A Graphical Design tool for Multi-Device User Interfaces based on UIML

Jan Meskens

promotor :
Prof. dr. Karin CONINX

co-promotor :
Prof. dr. Kris LUYTEN

Universiteit Maastricht

Eindverhandeling voorgedragen tot het bekomen van de graad
Master in de informatica human-computer interaction

universiteit hasselt

Abstract

A platform-independent *User Interface* (UI) would be desirable to enable the access to a wide range of services on many different devices with varying display sizes. It is not straightforward, however, to design such a UI because of the heterogeneity among these devices.

A *Model-Based User Interface Design* (MBUID) process can be used to (semi-) automatically generate an interface for a desired target platform from a platform independent user interface description. A lot of UI design tools have adopted this approach, however, many of them cannot have serious usability problems since the gap between the mental model of the designer and the presentation the tool offers is too big. Yet, most designers prefer the usage of traditional *Graphical User Interface* (GUI) builders because they provide a concrete representation during their design activities. Most GUI builders, however, do not support platform independence.

In this thesis, the strengths of both MBUID design tools and GUI builders are combined in order to create a new generation of flexible design tools. To establish this, we propose a domain specific GUI builder which can be generated from a domain vocabulary. This vocabulary contains a set of abstractions commonly used in the domain considered, each of them coupled to a concrete UI element. The domain specific GUI builder will allow the creation of a UI by combining these concrete elements, although the resulting UI description will still contain the necessary abstractions. The approach followed here is based on the *User Interface Markup Language* (UIML).

A concrete representation, however, shows the UI for one particular display size. To address multiple display sizes, a UI interpolation technique is integrated in the domain specific GUI builder. This technique generates UIs for arbitrary screen sizes from two previously created user interfaces. Consequently, the designed UI can easily be ported to a broad range of devices with varying screen sizes.

Our implementation of a domain-specific GUI builder and a UI interpolation mechanism is based on the open Uiml.net renderer. To validate our approach, the existing *System.Windows.Forms* (SWF) vocabulary is used as a mapping from the GUI domain vocabulary to specific SWF user interface elements. From this, we generate a SWF-specific GUI builder.

Acknowledgments

I would like to express my gratitude to a number of people who helped realize this thesis.

First, I want to thank my promotor *Prof. dr. Karin Coninx* for giving me the opportunity to realize this work. Furthermore, I thank my co-promotor, *Prof. dr. Kris Luyten*, for many insightful conversations during the development of the ideas in this thesis, and for helpful comments on the text. Particular thanks go to my supervisor, *Jo Vermeulen*, for his thorough, efficient and ongoing suggestions as well as for his support throughout the implementation and writing process of this thesis.

Furthermore, I like to thank *Bert De Decker* from the computer graphics group, for correcting the introduction to image based rendering in this tekst.

On a personal note, I am in great debt to my parents, *Fons and Hélène* and my sister *Lieve*: they supported and encouraged my studies and gave me sufficient freedom to make my own decisions. Also thanks to all my friends for the necessary diversions I needed during my studies. Special thanks go to *Lieven De Keyzer*, who convinced me to study computer science. Finally, I wish to thank my girlfriend *Kristien* for always being there for me and helping me through stressful times as well as for the corrections she made on my English text.

Contents

List of Figures

List of Tables

List of Listings

Part I

Preliminaries

Chapter 1

Introduction

1.1 Problem Description

Next to the rise of desktop computers, there has also been a widespread emergence of computing devices in the past few years [AnSA02]. Consequently, users want to use the same kinds of applications and access the same data and information on these appliances as on their desktop computers. As illustrated in Figure 1.1, an important characteristic of the different devices is the available screensize, which can vary from a very small display in mobile and embedded devices to a large display in a smartboard[1], widescreen projections, etc. The user interfaces for these devices may differ from the traditional interaction metaphors on desktops [AnSA02].

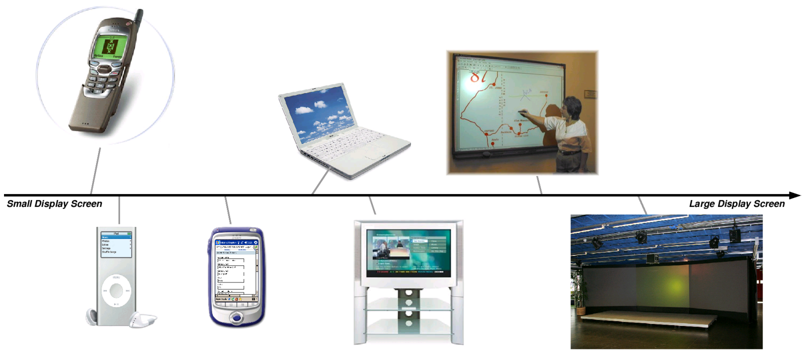


Figure 1.1: Devices ordered by their display size

The variance in display size between several devices complicates the deployment of a user interface to these devices. An interface designed for a desktop PC, for example, will fall partially outside the display region of a

[1]`http://smarttech.com/smartboard/`

PDA. Thus, depending on the available screen size, an automatic user interface adaptation should be performed. That way, the same application can be used from a broad range of devices. In order to accomplish this, several automated layout techniques have been introduced. In this context, layout means the process of determining the position and size of each visual object that is displayed in a user interface, and the result of that process [LF01]. Two popular automated layout techniques are *layout managers* and *constraint based systems.* Both techniques mostly result in a predictable and consistent interface, but they can only change the size or position of visual objects in an interface depending on the screen size. More advanced adaptations are impossible with automated layout techniques. An example of an advanced adaptation is the transformation of a listbox into a dropdown menu when the display size shrinks.

In order to become a usable interface for the desired platforms and devices, one can create a separate user interface for each device or platform. This approach, however, increases the development effort of an application significantly. Multi platform user interfaces address this issue by describing the user interface on a higher platform-independent level of abstraction. From this description, a (semi-) automatic reification process can be started towards the final interface on the desired target platform. This process, which starts with the design of the abstract models and progresses gradually towards the more concrete models is called *Model-Based User Interface Design* (MBUID) [CLC04].

Despite the reduction of development effort introduced by MBUID, the commercial world has not generally adopted this design method. One reason for the low acceptance of MBUID tools is that designers are forced to think at a high level of abstraction too early in the design process. This is contrary to *Graphical User Interface* (GUI) builders, which are better suited to support a creative user interface design process. A GUI builder uses direct manipulation operations to create the interface on a concrete level of abstraction. Yet, most GUI builders are constrained in the number of devices, platforms, toolkits or widget-sets that they support. Thus, there is room for a new generation of user interface design tools which combine the strengths of GUI builders and MBUID tools. This new tool will allow the designer to describe a user interface on a platform-independent level of abstraction without the necessity to define this description explicitly.

1.2 The Research covered by this thesis

We are searching for a new, flexible, design tool which can create a user interface on a concrete level of abstraction, but still reaches the desired platform independence. As well in litterature as in existing implementations, model based user interface design tools and GUI builders will be explored and evaluated. The proposed approach will automatically generate *domain-specific user interface builders.* These tools will be able to create interfaces on a concrete level, using domain specific abstractions.

A user interface, created on a concrete level of abstraction, is designed for a specific screen size. When this size changes, the user interface needs to be adapted. Several adaptation techniques will be researched, in order to combine these with the domain specific user interface builder. A new solution to this problem will be deducted, called *user interface interpolation.* This approach will be able to generate a new interface for a desired screensize from two previously created interfaces. The combination of the domain-specific user interface builders and user interface interpolation should result in a new development environment for multi-platform user interfaces.

The solutions described above will be based on the Uiml.net [LC04] UIML renderer. This renderer already allows the execution of a UIML interface description on different platforms and devices.

1.3 Outline

The next part covers the research that was conducted in this thesis. First, we provide a detailed comparison between model-based design tools and GUI builders. The strengths and weaknesses of both are discussed in detail. Chapter 3 discusses multi-platform user interfaces. Model-based user interface development and high level user interface descriptions are emphasized here. Next, demonstrational user interface design techniques and adaptable user interfaces are covered. A detailed overview of UIML and the Uiml.net renderer is provided in the final chapter of this part.

Details about the implementation part of this thesis are given in the next part. First, the domain specific user interface builder for UIML is introduced. Next, the integrated user interface interpolation mechanism is discussed in detail.

Finally, we draw the conclusions and provide some opportunities for future work.

Part II

Research

Chapter 2

A Comparison of User Interface Design Tools

Contents

User Interface Design Tools (UIDTs) help reduce the amount of code that programmers need to produce when creating a user interface and allow user interfaces to be created more quickly [MHP00]. This in turn enables rapid prototyping and therefore more iterations (or an iterative design process) which has been proven to be a crucial component for achieving high-quality user interfaces. Another important advantage of tools is that they help enforce a consistent look and feel, since all user interfaces created with a certain tool will be similar.

In this chapter, two categories of design tools are introduced: the traditional graphical user interface builders and model-based design tools. The former facilitates the development of UIs by intuitive direct manipulation techniques. The latter provides in the generation of a user interface from a high level declarative description. The strengths and weaknesses of these two tool categories are discussed in detail, in order to combine the strengths of both in a later stage of this thesis.

In order to compare both tool categories, a set of general user interface design tool principles is introduced first. Next, Sections 2.2 and 2.3 discuss respectively graphical user interface builders and model-based design tools. The application of the general principles in these tools is evaluated within each of these sections. Finally, a general evaluation of both categories is provided.

2.1 General Principles of User Interface Design Tools

In [HY91] some general principles for the construction of user interface design environments are stated:

- **There should be no hidden components:** this principle involves visibility and stems directly from the basic principles of direct manipulation.
- **Rich and immediate feedback should be provided:** the consequences of design actions need to be visible as those actions are being considered or carried out;
- **The designer should be in control of the design:** there is no substitute for the knowledge of a human designer and it is important to allow the designer to have full control over the design process;
- **The system should automate as much as possible:** automate as many of the tedious or repetitive steps as possible and provide guidance, direction and structure when is needed.

The final two principles are partially in conflict and can in fact represent an area of tradeoff: more control to the designer implies less automatization possibilities for the application and vice-versa.

In the remainder of this chapter, the application of these principles is evaluated within several tool categories. Each principle will be evaluated on a scale of zero to ten. A zero-score involves that the principle is not supported, a score of ten indicates the opposite: the principle is completely adopted.

2.2 Graphical User Interface Builders

Graphical User Interface (GUI) builders basically contain two main components [Pay98]. The first one is a library or toolbox with a wide variety of elements for a graphical user interface: e.g. windows, buttons, edit-controls, drawing areas, etc. The other one is a visual editor to combine these elements with the mouse by drag-and-drop or click-and-point actions. These tools can also be categorized as *What You See Is What You Get* (WYSIWYG) tools. In the next section, we will discuss a few examples.

2.2.1 Tool examples

Some well known GUI builders are Glade[1] and Qt Designer[2]. Glade enables quick and easy development of user interfaces for the GTK+ toolkit[3] and the Gnome desktop environment[4]. GTK+ is a multi-platform toolkit for creating graphical user interfaces. Offering a complete set of widgets, GTK+ is suitable for projects ranging from small one-off projects to complete application suites.

Besides Gnome, KDE is the other main desktop environment for Linux [Ant01]. While Gnome is built on top of the GTK+ toolkit, KDE[5] is built on top of the Troll Tech's Qt[6] widget library. Qt is a high-level cross-platform toolkit that provides an object-oriented graphical application library for C++ programmers. Qt Designer is the tool to automate most of the legwork of designing a user interface in Qt. Additionally Qt Designer can be integrated in the KDevelop integrated development environment, which gives you the facilities of a *Rapid Application Development* RAD environment for C++ programmers. Other examples include Visual Basic and the "resource editors" or "constructors" that come with Microsoft's Visual C++ and most other environments [MHP00].

2.2.2 User Interface rendering

After the construction of a user interface inside the tool, the interface should be made accessible to the application logic. The extraction of the interface outside of the tool is called *user interface generation.* Most tools store the interface structure in an intermediate *User Interface Description Language* (UIDL), a - commonly XML based - description of a user interface. From this intermediate format, the final interface should be rendered. The

[1]`http://glade.gnome.org/`
[2]`http://www.trolltech.com/products/qt/features/designer`
[3]`http://www.gtk.org/`
[4]`http://www.gnome.org/`
[5]`http://www.kde.org/`
[6]`http://www.trolltech.com/products/qt`

interface rendering process of a basic UIDL is straight forward: a specific XML tag is directly mapped on a corresponding UI element. In Figure 2.1 the rendering of a <*button*> tag to a *SWF* button is shown.

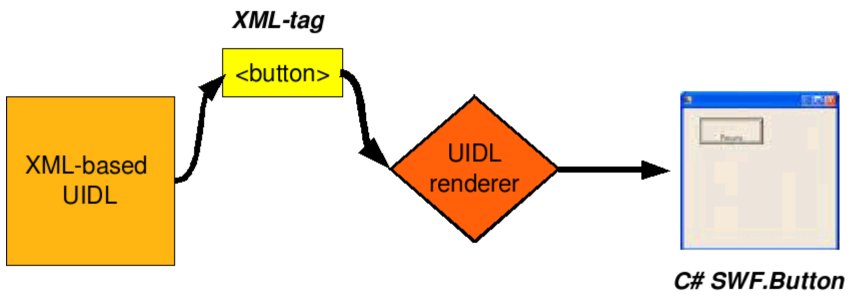


Figure 2.1: Rendering of a user interface description language

The rendering can be done by two approaches: compilation or interpretation. Although some literature associates *user interface rendering* with the interpretation approach only, it is defined as the overall UI generation concept here. The two rendering techniques are discussed below.

Compilation

The compilation approach involves the translation of the UIDL specification to programming code. The generated code can then be adapted in order to couple it to the application logic. In order to come to the final interface coupled to the logic, this code can then be built.

A major drawback of the compilation approach is the possible arise of an inconsistency between the generated code and the UIDL. Without the translation step, alterations to the UIDL are not reflected in the programming code. Otherwise, modifications in the code are never returned in the UIDL except when an inverse translation is available. Yet, this inverse translation is not commonplace and rarely available.

Both Glade and Qt Designer store the interface structure in an UIDL format, respectively *glade xml* and Troll Tech's own UI XML format. In Qt Designer, the *User Interface Compiler* (UIC) is used to generate a .h header file and the implementation file of the dialog class. Having done this, the KDevelop C++ IDE can be used to add callbacks, compile and debug the program. Glade, however, can process XML interfaces directly into C source code.

The Qt and Gtk+ toolkits have bindings in many languages: Qt is amongst others available in C++, the .NET platform (Qt#), python (PyQt) and java (Jambi), while Gtk+ has bindings for C, C++, Eiffel, Ada95, the .NET platform (Gtk#), etc. It is possible to convert the interface, described in the intermediate xml format, to some of these bindings. Besides the *uic* tool for *C++*, the Qt interface can be compiled to the Python bindings with *puic*. There is also a *juic* tool available to compile the interface to Java, but this requires a slightly modified version of the original Qt *ui* format, called *jui*. Glade can emit Eiffel, Ada95, and C++ source code.

Interpretation

Whereas the compilation approach generates source code, the interpretation approach generates the interface directly from the UIDL. The interface is rendered from the UIDL without any translation step. This involves that there can no longer be a inconsistency, any adjustment to the UIDL is directly reflected in the interface. Glade does not only support the compilation approach, but also the interpretation. To accomplish this it uses the *Libglade*[7] library, which builds the interface from an XML file (glade's save format) at runtime.

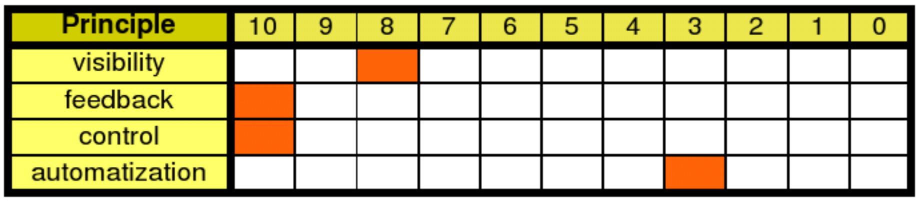
Another example of a UIDL is Microsoft's *Extensible Application Markup Language* (XAML[8]), which is a declarative XML-based language that defines objects and their properties in XML. XAML syntax focuses upon defining the UI for the *Windows Presentation Foundation* (WPF) and is therefore separate from the application code behind it. WPF is the new presentation API in .NET Framework 3.0 (formerly WinFX). Users can program directly against the API with .NET, but one can also use the WPF to interpret XAML documents.

2.2.3 Evaluation

Visibility the use of a widget panel, which shows all the possible widgets, avoids the presence of hidden components as stated in the first principle (see Section 2.1). Additionally, the properties of the different widgets are mostly listed in a property window, which reduce the time needed to look them up. One problem with GUI Builders is that, unless the interface builder is very limited in scope, the designer sees more than he needs to create the interface. Because of these redundant objects, the visibility-score is reduced with two points, which results in an eight.

[7] `http://developer.gnome.org/doc/API/libglade/libglade.html`

[8] `http://www.xaml.net`



Principle	10	9	8	7	6	5	4	3	2	1	0
visibility			🟧								
feedback	🟧										
control	🟧										
automatization								🟧			

Table 2.1: The four general principles applied to GUI builders

Feedback the second principle is completely supported in this type of tool because it uses direct manipulation to create interfaces. Direct manipulation involves immediate feedback when an action is undertaken by the designer. For instance, when a button is dragged to another location in a form, this is shown interactively to the designer. A user interface should not be build or executed before the result can be evaluated. The evaluation is seamlessly integrated in the design process of GUI Builders.

Control and Automation the designer is placed in full control of the design, he can use the direct manipulation techniques to develop the interface he has in mind. The full control given to the designer results in a minimal automated approach. This automation involves the replacement of programming code by direct manipulation. Intelligent techniques to reduce the development effort on repetitive steps are not provided in this type of tool.

2.3 Model-Based and Automatic Techniques

Although the UIDTs described in Section 2.2 facilitate the creation of user interfaces, the design itself stays a manual task: the fourth principle of user interface design environments (Section 2.1) is not well supported. A thread in user interface research that addresses this issue is the investigation of automatic techniques for generating interfaces [MHP00]. The goal of this work is generally to allow the designer to specify interfaces at a very high level of abstraction, with the details of the implementation to be provided by the system. Motivations for this include the hope that developers without user interface design experience could simply implement the functionality and rely on these systems to create high-quality user interfaces. The systems might allow user interfaces to be created with less effort (since parts would be generated automatically). Further, there is the promise of significant benefits such as automatic portability across multiple types of devices (see Chapter 3), and automatic generation of online help for the application.

2.3.1 Model-Based User Interface Design

The most well-known automatic user interface generation technique is *Model-Based User Interface* (MBUI) design. MBUIs are designed on an abstract model of the interface instead of its visual appearance [LL02]. These declarative models store a conceptual representation of the required interface. Recently, the models supported by *Model-Based User Interface Design Environments* (MB-UIDEs) have increased both in number and in expressiveness. The first-generation of MB-UIDEs is concentrated on modelling the underlying application domain through a domain model. Typically, such a limited view of the modelled domain produced simple menu or form-based interfaces. The past few years, however, MB-UIDEs have exploited a much wider range of interacting models, with consequently an increase in the quality and variety of their generated interfaces [GtMP$^+$98].

Most MB-UIDEs contain two important main parts: the conceptual description of the interface, the interface model, and a mapping function to map these abstract descriptions to the concrete interface [PE99]. The Model-based interface development process [CLC04] usually starts with the design of the abstract models and progresses gradually towards the more concrete models, resulting in the final interface when the design process is complete.

2.3.2 Interface Models

An interface model is an ordered collection of all the relevant elements of a user interface [PE99]. The elements of an interface model are grouped into model components. The basic components are the task model, the user model, the domain model, the presentation model, and the dialog model.

Task model

The *Task Model* (TM) expresses the activities that end-users of an application want to undertake and any temporal constraints that exist between tasks or sub-tasks[GtMP$^+$98]. A common task model is the *ConcurTask-Trees* (CTT) as proposed in [Pat00]. This notation offers a graphical syntax, an hierarchical structure and a notation to specify the temporal relation between tasks.

User model

A *User Model* (UM) represents the different types of users of a target application. It is not a cognitive model but a definition of the attributes and roles of users [PE99]. In SUPPLE [GW04], a different user model is used. User traces are used here to describe the way a user interacts with an application (see Section 4.2.3).

Domain model

The *Domain Model* (DM) is used to capture the functionality of the underlying application or database [GtMP$^+$98]. This model therefore specifies the application's public interface in terms of the low-level data sources and services the application makes available to the user. In nature, it is very similar to an application's data model [PE99] but it is also intended to explicitly represent the attributes of objects and the relationships among the various domain objects.

Presentation model

The *Presentation Model* (PM) [PE99] is a representation of the visual, haptic, and auditory elements that a user interface offers to its users. For example, a presentation element might be a window that contains additional elements such as widgets that appear in that window.

Dialog model

The *Dialog Model* (DiM) defines the way in which the presentation model interacts with the user. It presents the actions that a user may initiate via the presentation elements and the reactions that the application communicates via those same elements [PE99].

Mappings

In general terms, developers use model-based systems to define firstly one or more abstract models: a user-task model, a domain model and a user model. The model-based systems then attempt to generate from those model-components more concrete models such as presentation and dialog models that are then converted into an executable interface specification (i.e, a running user interface) [PE99]. This requires some sort of mapping to perform this conversion. These mappings can be realized at three levels:

- **between abstract models:** the assignment of users to tasks, for example, is a mapping process. At this level, there is also a mapping from task to domain model possible [PE99].
- **between concrete models:** the presentation elements and the dialog elements in an interface model must be linked to each other in order to specify a running user interface [PE99]. Therefore presentation-dialog mappings are needed.
- **from abstract models to concrete models:** for example, given task t in domain d find an appropriate presentation p and dialog D that allows user u to accomplish t [PE99]. At this level task-dialog,

task-presentation and domain-presentation are the most common mappings.

2.3.3 Model-Based Design Tools

The generation of user interfaces from domain models is the approach followed by Mecano [PEGM94]. Through a model editor, the developer can build, visualize and review a domain model. Mecano follows an iterative approach, which involves the continuous refinement of the domain model towards the final user interface. The model will be used as the input of an intelligent designer component, a tool which produces a dynamic dialog specification and a preliminary layout for the interface. A dialog specification is created at two levels: high- and low-level. A high-level dialog specification defines all interface windows, assigns interface objects to windows, and specifies the navigation scheme among windows in the interface. The low-level one defines specific dialog elements (widgets) to each interface object created at high level and specifies how the standard behavior of the dialog element is modified for the given domain. In a next step, the layout can be refined using NeXT's interface builder.

Besides the dialog model, Teallach [GtMP$^+$98] utilizes the four declarative interface models in the process of generating user interfaces. Teallach stores the different models in a shared model repository. Initial models can be generated from the ones already stored in the repository. This generation is supported by mapping rules, at different levels as described in Section 2.3.2, stored at several places in the Teallach's architecture. Thus, Teallach guides the developer through the construction of a presentation model, starting from the domain model. Some parts of the intermediate models can be generated automatically by *drag and generate* operations: for example, after dragging a part of the domain model to the task model editor, an initial task model will be generated. Other parts of the model should be made by hand and connected manually to the sibling models.

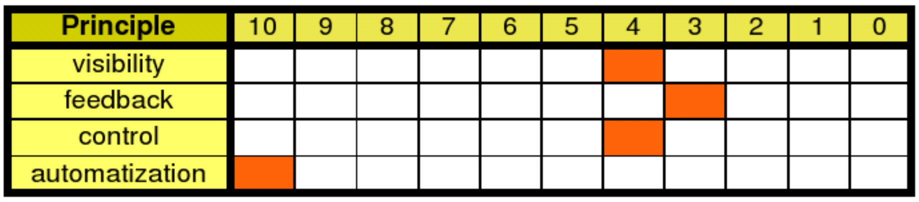
In MOBI-D [PE99], a general-purpose model-based interface development environment, all the different components of the interface model are used. Whereas the mappings are embedded into the system in Teallach's and Mecano's approach, MOBI-D allows interface developers to directly access and set the mappings according to their needs. In order to support the inspection and setting of mappings, MOBI-D extends the definition of an interface model to include a new model component called the *design model*: a declarative representation of all the mappings. A design editor is supplied to support the designer in the construction of same-level-of-abstraction mappings. For the more complex abstract-to-concrete mappings, MOBI-D provides a decision-support tool: *The Interface Model Mapper* (TIMM).

2.3.4 Evaluation

Control and Automation besides the construction of abstract interface models by model editors, a *Model Based User Interface Design Tool* (MBUIDT) supports the transformation of this model towards the final interface. These transformations are often based on heuristics [MHP00], which results in a difficult to understand connection between the model and the final result. Whereas this difficult relationship makes it hard for the designer to have control over the final UI, the automatization factor increases significantly. Several solutions to enhance the control of the designer are worked out in the tools discussed in Section 2.3.3. In Mecano [PEGM94] and Teallach [GtMP^{+}98], the designer can modify unexpected generations at different steps in the process. The MOBI-D [PE99] approach passes the mapping responsibility to the designer. Despite these creative solutions, only partial support for the control principle was realized.

Visibility the model-editors guide the developer in the model design process and typically show an intuitive interface with the possible options in the model. For example, in the CTT task modelling environment [Pat00], a panel with possible tasks and temporal operators between the tasks is shown. Although hidden components are avoided at model construction level, the components used to build the final interface stay hidden for the designer. Because of this, the first principle is only partially supported by MBUIDTs.

Feedback the impact of a change in the interface model, is rarely visualized immediately in the final interface or other models. For example, when a task model is modified, the result of this modification is only reflected in the dialog model after the execution of all the automatic transformations and related dialogs. Thus, the feedback principle is only minimally supported. The evaluation of each principle is shown in Table 2.2.



Principle	10	9	8	7	6	5	4	3	2	1	0
visibility							🟧				
feedback								🟧			
control							🟧				
automatization	🟧										

Table 2.2: The four general principles of user interface design environments applied to the model-based user interface development process.

2.4 Discussion

In this chapter, several sorts of design tools were discussed and evaluated. The decrease of control, visibility and feedback are the most important reasons that model-based user interfaces have not been widely adopted in the commercial software development world, which has instead gravitated towards GUI builders. An important reason for this success is that GUI builders use graphical means to express graphical concepts (e.g., interface layout) [MHP00]. By moving some aspects of user interface implementation from conventional code into an interactive specification system, these aspects of interface implementation are made available to those who are not conventional programmers. This allows many domain experts to prototype and implement interfaces highly tailored to their tasks, and allow visual design professionals to become more involved in creating the appearance of interfaces. Even the programmer benefits, as the speed of building is dramatically reduced.

Model-based UI tools do not match or augment the work practices of designers. They often force designers to think at a high level of abstraction to early in the design process [LL02]: for example, by specifying a task model which is then transformed in a concrete UI. This abstract thinking is contrary to the normal task of the designer to develop concrete interfaces. Designers must also learn a new language for specifying the models, which raises the threshold of use [MHP00]. Furthermore, automatically generating interfaces is a very difficult task: automatic and model-based systems have each placed significant limitations on the kinds of interfaces they can produce [MHP00]. A related problem is that the generated user interfaces are generally not as good as those that could be created with conventional programming techniques. Despite these drawbacks, model-based techniques have been widely adopted by multi-device UI development tools.

The comparison between the different tool-categories is summarized in Table 2.3.

Principle	Tools	10	9	8	7	6	5	4	3	2	1
visibility	*Gui Builders*			🟧							
	MBUIDTs							🟧			
feedback	*Gui Builders*	🟧									
	MBUIDTs								🟧		
control	*Gui Builders*	🟧									
	MBUIDTs							🟧			
automatization	*Gui Builders*								🟧		
	MBUIDTs	🟧									

Table 2.3: Comparison table between the different tool categories.

Chapter 3

Multi-Platform User Interfaces

Contents

Next to the rise of desktop computers, there has also been a widespread emergence of computing devices in the past few years [AnSA02]. However, users want to use the same kinds of applications and access the same data and information on these appliances that they can access on their desktop computers. The user interfaces for these platforms are different from the traditional interaction metaphors on desktops [AnSA02].

Multi-platform user interfaces are the subject of this chapter. These interfaces are described on an appropriate level of abstraction which omits platform-dependant terms. *Platform* refers to the hardware and software platforms [CCT$^+$03], which are, the computational and interaction devices that can be used. Multi-platform interfaces will adapt to the platform considered, without duplicating the development effort. For example, one interface description can be used for a desktop computer, a PDA and a smartphone. These type of interfaces belong to adaptable interfaces, which are discussed in Chapter 4.

First, the general approach of multi-target user interfaces is discussed. Next, a short introduction to high-level user interface descriptions is given. Finally, several tools which support the creation of multi-platform interfaces are discussed.

3.1 Multi-Target User Interfaces

The *context-of-use* [CCT$^+$03] of an interactive system is defined by three classes of entities: (1) *the users of the system* who are intended to use the system; (2) *the hardware and software platform(s)*, that is, the computational and interaction device(s) that can be used; (3) *the physical environment* where the interaction can take place. An interface which is capable of supporting multiple contexts of use is called a *multi-target user interface*. Multi-platform interfaces are a specialization of these multi-target user interfaces, which adapt the interfaces depending on the hardware and software platforms considered.

Initial models [CCT$^+$03], which are specified manually by the developer, serve as input descriptions to the multi-target development process. This process uses a combination of *operators* to transform initial models into transient models until the final context-sensitive interactive system is produced. A transient model is an intermediate model, necessary for the production of the final executable interface. Task-oriented descriptions, abstract and concrete interfaces are typical examples of transient models. Two kinds of operators are important in the production of transient models:

- *Vertical transformations* that may be processed along a top-down (reification) or bottom-up (abstraction) process;
- *Horizontal transformations*, such as those performed between HTML and WML content descriptions, correspond to translations between models at the same level of reification.

The final result of the multi-target design process is the *Final User Interface* (Final UI), expressed in source code, such as Java and HTML, or interpreted as a pre-computed user interface and plugged into a run-time environment that supports dynamic adaptation to multiple targets.

3.2 High-Level User Interface Descriptions

High-Level User Interface Descriptions (HLUIDs) provide the possibility to create a device-independent and abstract description of a user interface with a language that is easy to use in heterogeneous environments [Luy04]. Although there are also many User Interface Description Languages that do not use the *eXtensible Markup Language* (XML), XML is the most appropriate language to describe *High-Level User Interface Description Languages* (HLUIDLs) [Luy04]. This description language can reach easily *platform independence*, if there is an XML parser available on the system, the XML description can be used. Other desired properties of XML are: *consistency* through the use of a DTD[1], *unconventional I/O* (e.g.: WML or VoiceXML), *rapid prototyping*, *constraint definitions*, *easily extensible* and *re-usability.*

The contribution of HLUIDs to multi-platform interfaces is two-fold:

- an HLUID makes it possible to (semi-) automatically generate the code of the UI, as desired in *model-based* approaches for developing UIs [SV03];
- when an interface is required on different computing platforms, this can be done by exchanging the HLUID from one platform to another without any changes to avoid any extraneous development effort [SV03]. An HLUID renderer is responsible for generating the interface on the desired platforms.

Although an HLUIDL description should mainly stay the same for a wide range of devices, the same description generally cannot be reused in more extreme conditions such as very small screens or multiple displays [LVC06]. In practice, the description also contains widget-set specific layout information. This results in multiple alternative descriptions for the same user interface to keep them consistent for different widget-sets on different devices. To avoid the creation multiple alternative descriptions, another approach is to create a HLUIDL which targets a generic but limited set of widgets [LTVC06].

In the remainder of this section, some examples of HLUIDLs are provided

3.2.1 UsiXML

The *USer Interface eXtensible Markup Language* (UsiXML) [FMVM06] is an XML-compliant HLUIDL covering all four levels of abstraction: the *Final User Interface* (FUI), the *Concrete User Interface* (CUI), the *Abstract User Interface* (AUI), and the tasks and domain model. The FUI refers to the actual user interface, which will be rendered on a given computing platform. The CUI abstracts the FUI into a definition that is independent from any programming language. The CUI contains a detailed description of the user interface in terms of widgets, layout, navigation and behavior.

[1]Document Type Definition

The AUI abstracts the CUI into a UI definition that is independent from any interaction modality. This is done by the use of *interaction spaces*, i.e. the grouping of tasks that have to be presented together. In UsiXML, the AUI is populated by Abstract Components and Abstract Containers. The tasks and concepts model forms the last level of abstraction. In UsiXML the CTT notation [Pat00] is used to represent tasks, while class diagrams are used for the domain model.

3.2.2 XForms

XForms [Wik07] is an XML format for the specication of user interfaces, specically web forms. XForms was designed to be the next generation of HTML / XHTML forms, but is generic enough to be used in a stand-alone manner to describe any user interface, and even to perform simple and common data manipulation tasks.

3.2.3 UIML

The *User Interface Markup Language* (UIML) is an XML based meta-language to describe an interface in a device-, toolkit-, interaction- and platform-independent way. Instead of specific abstraction (part,property,...), UIML uses generic terms (e.g. window, button, etc) to define the interface. The mappings of these terms to the appropriate concrete interface elements is defined in a vocabulary section. The use of a vocabulary provides great flexibility: it allows to build a custom user interface language on top of UIML, similar to the way the XML meta-language supports building a custom data format. A more detailed description of UIML is given in Chapter 6.

3.3 Multi-Platform Interface Design Tools

In this section, tools are discussed to construct multi-platform interfaces. The *TERESA* [PMS03] tool reificates the task-models, specified in the ConcurTaskTree [Pat00] notation, to concrete user interfaces in four steps. First a high-level task model and a domain model should be constructed in the ConcurTaskTree notation. Whereas the former identifies the possible contexts of use and the various roles involved, the latter aims to identify all the objects and concepts that have to be manipulated to perform tasks and the relation among such objects. During the second step, the designer needs to refine or filter the high-level task-model towards a system task-model for the different platforms considered.

Next, the abstract user interface description, defined in the *XML AUI* format, should be generated from the system task-model. This transformation of task-models to abstract user interfaces is done by the detection of *Presentation Task Sets* (PTSs), [MPS04] which represent tasks that are enabled over the same period of time. Once these PTSs, and the transitions among them, are detected, the abstract interface can be derived. This will be done by mapping the tasks on corresponding interactors, depending on the task category. For example, a task classified as an "edit" type will be mapped on an interactor that allows information modification. An analysis of the CTT task model will build the composition among the different interactors. This composition is steered by the temporal operators between the tasks and various task-attributes (such as frequency).

The abstract interface description can be used as input for the concrete interface construction process. During this platform-dependent phase, the specific properties of the target-device should be considered: supported interaction techniques, operating system, toolkit, etc. The designer can choose the preferred level of control in this step. *TERESA* can render an interface depending on predefined criteria, but the designer can easily modify these criteria. The four steps are summarized in Figure 3.1.

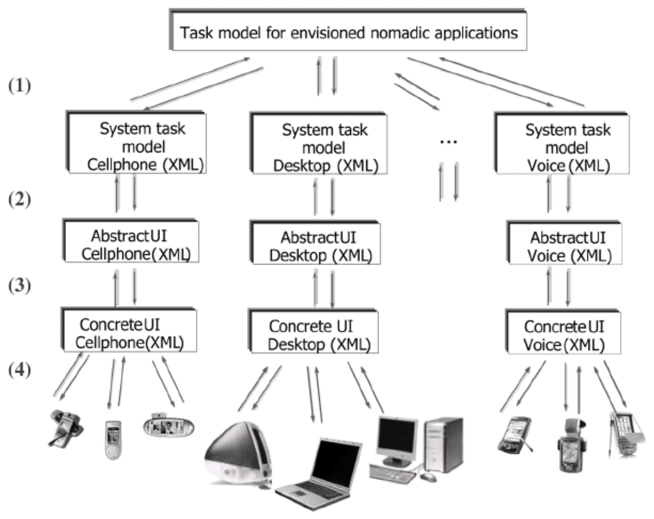

Task model for envisioned nomadic applications
(1)
System task model Cellphone (XML)
System task model Desktop (XML)
...
System task model Voice (XML)
(2)
AbstractUI Cellphone(XML)
AbstractUI Desktop (XML)
AbstractUI Voice (XML)
(3)
ConcreteUI Cellphone(XML)
Concrete UI Desktop (XML)
ConcreteUI Voice(XML)
(4)

Figure 3.1: The construction of multi-device interfaces by *TERESA* in four steps

Graceful Degradation [FMVM06] consists in specifying one source interface, designed for the least constrained platform, and to apply transformation rules to this source interface in order to produce specific interfaces targeted to more constrained platforms. By the use of splitting rules, interactor and image transformation rules, moving rules, resizing rules and removal rules it is possible to paginate an interface in logical units for different devices. Graceful degradation is possible at UsiXML's CUI level and at the combined AUI and task-model level. In order to become the most usable interface, the latter will be preferred to the former.

In TIDE [AnSA02] (*Transformation-based Integrated Development Environment*), the developer writes UIML (see Chapter 6) code and the IDE generates the interface, as expected. The assumption made here is that the process of creating an interface in an abstract language, such as UIML, which will be translated into one or more specific languages, undergo a process of trial and error. The developer builds what he or she thinks will be appropriate in UIML, renders to the desired language(s), and then makes changes as appropriate. TIDE is built to facilitate this process in three ways: (1) the UIML code of the abstract interface can be edited; (2) the result can be rendered for the desired platform; (3) after this rendition, the relationships between parts of the interface and blocks of UIML code are visualized by arrows connecting these two. The UIML code can be modified for the different platforms, which establishes the horizontal transformation.

In [JB03], a design environment is presented for implementing multi-device interfaces using custom tag libraries for *Java Server Pages* (JSPs), called the *Abstract User Interface Technology* (AUIT). An AUIT interface description contains generic, device-independent mark-up tags describing interface elements and layout information. Depending on the target platform, the information stored in this abstract description is used to render an interface in the suitable markup (e.g. HTML or WML) with the correct layout, using the right interaction, adornment (e.g. available fonts only, no colour if black-and-white device) etc. The design environment provides three editors – a tree structure-, text- and layout-editor – and two rendering views, one for a web browser and one for a PDA, to facilitate the creation of these specific type of interface. The most remarkable part is the layout editor which provides an intuitive manner to specify the relative positions and groupings of screen elements based on a grid model. While running the interface for a particular device, it may split into multiple parts using the grid specified, in order to fit the interface into the device's smaller screen. Thus, the horizontal transformation is done automatically by the splitting blocks defined in the grid editor.

3.4 Conclusion

The various concepts of multi-platform interfaces were introduced in this chapter. Also a short review of tools to create this type of interfaces was included. The design of multi-platform interfaces is an important goal of the design tool developed in this thesis.

Chapter 4

Adaptable User Interfaces

Contents

As discussed earlier in the problem description, an automatic adaptation of a user interface is necessary to make an application accessible from a broad range of devices with varying display sizes. To better understand this issue, existing adaptation techniques are studied in this chapter.

In Human-Computer Interaction, adaptation [TC99] is modeled as two complementary system properties: adaptability and adaptivity. While the former is the capacity of the system to allow users to customize their system from a predefined set of parameters, the latter is the capacity of the system to perform adaptation automatically without deliberate action from the user's part. Interfaces which support adaptability is called *user-adaptable* interfaces, the ones which support adaptivity are called *self-adapting* interfaces.

In this chapter, user-adaptable and self-adapting interfaces are introduced. Both are discussed in detail and several examples are provided. Finally, the strengths and weaknesses of these two adaptation techniques are discussed.

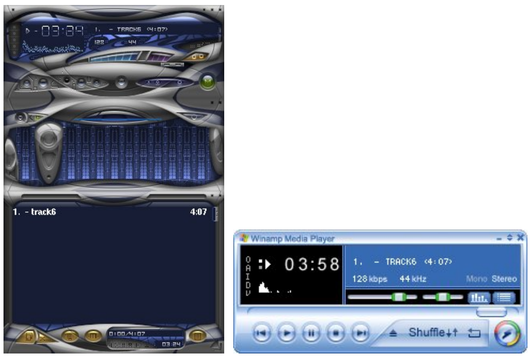
4.1 User-adaptable interfaces

4.1.1 General approach

An active interface customization approach can be established with user-adaptable interfaces. In this type of interfaces, one can adapt the *primary interface* through *secondary interfaces* [SCPR06]. The latter contains operations and interaction techniques to restructure the former, which is the interface to perform the application's main task. A well designed secondary interface with attractive adaptation options led to a great flexibility in the organizational structure of the primary interface. The more one can customize the primary interface to his personal needs, the better he will perform the task provided by the application.

4.1.2 Examples of tools

Two of the simplest forms of user interface customization are *skins* and *themes* [SCPR06]. A skin, or a theme, can simply consist of a set of colors or textures used by the existing drawing code. It can also partially or completely replace that drawing code, possibly adding complex output modifications. In addition to the visual style of interface elements, skins and themes can also specify the layout and to a lesser degree the behavior of these elements. However, although these allow visual designers to customize interfaces, these systems remain out of reach for end-users who can only choose between predefined theme options. Figure 4.1 shows two skins for the popular Winamp[1] multimedia player. In this example the primary interface is an interface to play multimedia, the secondary interfaces are a complex set of drawing tools - gimp, photoshop, . . . - and text editors - vim, notepad, . . . - to construct a skin.



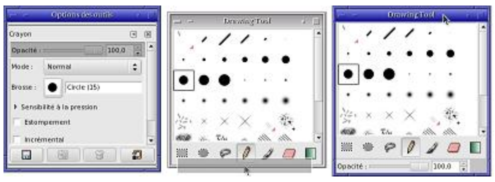
(a) futuristic skin (b) WMP skin

Figure 4.1: Winamp skinning examples

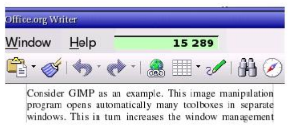
[1]http://www.winamp.com/

User Interface Façades [SCPR06] is a system that provides users with simple ways to adapt, reconfigure, and re-combine existing graphical interfaces, through the use of direct manipulation techniques. A user interface facade is a user-specified set of graphical interfaces and interaction techniques that can be used to customize the interaction with existing, unmodified applications. User interface Façades make it possible to:

- copy and paste screen regions,
- cut screen regions,
- use external components to interact with applications.

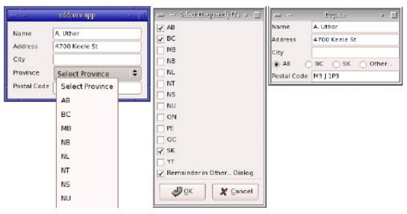


(a) Copy and paste screen regions



Consider GIMP as an example. This image manipulation program opens automatically many toolboxes in separate windows. This in turn increases the window management

(b) Creation of a hole in a word-processor interface



(c) Replacing of widget types

Figure 4.2: User Interface Façades (Images courtesy of [SCPR06])

User Interface Façades provide intuitive well designed secondary interfaces to adapt primary ones. In Figure 4.2a it is shown how to copy and paste screen regions between two Gimp[2] dialogs by direct manipulation. The opacite slider is selected and dragged to a new desired dialog. Figure 4.2b displays the creation of a hole in a textwriter interface. Through this hole, the result of an underlying calculator window can be consulted. It's also possible to change widget types as shown in Figure 4.2c. The dropdown widget in the left view is changed to a list of radiobuttons in the right view, the middle view is the user interface facade used to change the widget type.

[2]GIMP is the GNU Image Manipulation Program. It is a freely distributed piece of software for such tasks as photo retouching, image composition and image authoring. `http://www.gimp.org/`

4.2 Self-adapting interfaces

Whether adaptation is performed on human requests or automatically, the design space for adaptation includes three orthogonal axes [TC99] (see Figure 4.3):

- *The target for adaptation:* this axis denotes the entities for which adaptation is intended;
- *The means of adaptation:* the components of the system involved in adaptation;
- *The temporal dimension of adaptation,* which can be static, between sessions, or dynamic, at runtime.

During this section, several self-adapting interfaces will be discussed, classified by the target (see Figure 4.3) for which adaptation is intended.

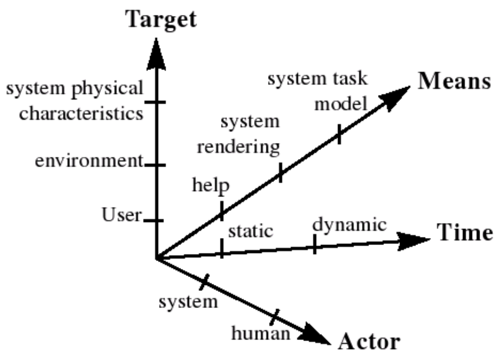


Figure 4.3: A design space for adaptation at high level of reasoning (image courtesy of [TC99])

4.2.1 User-Adaptive Interfaces

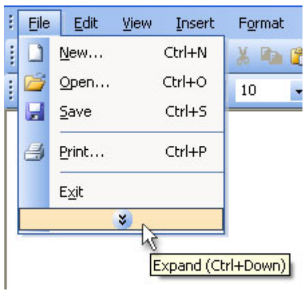


Figure 4.4: SmartMenus

The interfaces discussed here all have one idea in common [Jam03] : they learn something about each individual user and adapt its behavior to them. The most well-known example is provided by the *Smart Menus* feature that Microsoft introduced in Windows 2000, as shown in Figure 4.4. The idea behind these menus is that in the long run they should contain just the items that the user accesses regularly, so that the user needs to spend less time searching within menus. Another example are the *Split Menus* [SS94], which are organized into high- and low-frequency regions, with the several most frequently used options appearing at the top of the menu.

4.2.2 Physical Characteristics-Adaptive Interfaces

The physical characteristics of a system can be refined [TC99] in terms of interactional devices (e.g., mouse, keyboard, screen, video cameras), computational facilities (e.g., memory and processing power), and communicational facilities (e.g., bandwidth rate of the communication channels with other computing facilities).

The platform independent interfaces discussed in Chapter 3 can be categorized as physical characteristics-adaptive interfaces. Each of these interfaces will adapt to the device considered. Most of them do not adapt at runtime, but support static adaptation between sessions.

4.2.3 User- and Physical Characteristics-Adaptive Interfaces

Beside user-adaptation, the systems discussed here adapt to some specific physical characteristics changes too. SUPPLE [GW04][GCH$^+$05] is a user interface optimization tool, which is able to render UIs on different device types for different users. When asked to render an interface on a specific device and for a specific user, SUPPLE searches for the rendition that meets the device's constraints and minimizes the estimated cost (user effort) of the person's activity [GW04]. Not only the layout of the UI will be optimized, also the individual widgets are optimally chosen by SUPPLE during the rendering.

A functional interface model, a device model and a user model serve as the required inputs of SUPPLE [GW04]. The user model consist of *user traces*, which describes the sequences of elements manipulated by the user. Each of these inputs will be used to render an optimized UI as output. The objective is [GW04] to render each interface element with an available widget. Thus a *legal rendering* will be defined as a mapping from interface elements to widgets, which satisfies the interface and device constraints. More than one legal rendering will be possible, SUPPLE seeks the best: the one which minimizes the expected cost of user effort. A *cost function* is used as an estimate of the user effort involved in manipulating a particular rendering of an interface.

The bottleneck of decision-theoretic optimization techniques such as SUPPLE is the cost function [GW05]. In most cases, the numerous parameters of these functions are chosen manually, which is a tedious and error-prone process. While domain-specific learning techniques have been used occasionally, most practitioners parametrize the cost function and then engage in a laborious and unreliable process of hand-tuning. In the SUPPLE case, over forty factors and the corresponding weights has to be chosen manually to yield the desired solutions.

These cost function related issues are addressed by Arnauld[GW05]: a tool which facilitates the definition of cost functions. This reduces not only the burden on developers but it enables also wide-scale personalization capabilities, simply by defining separate cost functions for each user. One way to establish this is *example critiquing* which consists of recording the natural interactions of the user and use these to further improve the used cost function. In some cases, the user's natural actions provide insufficient feedback to learn a cost model. In these cases, the computer must facilitate preference elicitation by generating information-gathering questions to ask the user. With the use of these techniques, Arnauld can be integrated with SUPPLE to optimize SUPPLE's cost function's parameters.

4.3 Discussion

The biggest advantage of user-adaptable interfaces is that they leave the user in control, he can organize the interface in a way he or she likes. This is contrary to automated approaches, which will adapt the interface automatically according to some internal logic (optimization logic for example). However, these frequent changes in the arrangement of the interface can produce a variant of the general usability problem of *predictability* [Jam03]
.

Adaptable interfaces suffer from the problem that new 'secondary' interfaces and interaction techniques must be added to support the customization of the 'primary' interface [SCPR06]. These secondary interfaces should be designed so that they don't disturb the user in performing his main task through the primary interface. Further more they must be very intuitive and usable in order to convert the adaptation of the interface to a quick and light process. Otherwise, the user may not know what options exist or how to get them and trial and error with different settings can be time-consuming and frustrating [Jam03] . Self-adapting interfaces don't need secondary interfaces, the adaptation is a background process while the user performs his main task.

4.4 Conclusion

Different sorts of adaptation were introduced during this chapter: an active approach, user-adaptable interfaces, and a passive one, self-adapting interfaces. Self-adapting interfaces adapt dynamically or statically to a change in the adaptation target. This type of interface will prove useful during the construction of the interface interpolation mechanism in a later stage of this thesis.

Chapter 5

User Interface Design by Demonstration

Contents

To reach platform independence, a user interface needs to be adapted depending on the screensize. For the designer, it is not trivial to specify such an adaptation in an intuitive way. A possible technique to accomplish this is to demonstrate the desired adaptation behavior, in order to *train* this behavior to the interface.

A demonstrational design approach is used in *intelligent user interfaces*, which are the subject of this chapter. After a short introduction to intelligent user interfaces, the general approach of demonstrational interface design is discussed. Next, a selection of tools which adopt this approach is presented. Finally, the strengths and weaknesses of this type of user interface design are discussed.

5.1 Introduction to Intelligent User Interfaces

What happens when an end-user interacts with a user interface is called *user interface behavior*. For example, when a user clicks a button, this button should be enlarged; after clicking another button, a label should be hidden. Traditionally, interface behavior is specified by an abstract sequencing specification: for example, scripting- or programming languages. However, there are also interfaces which *know* how to react to user interaction. This type of interfaces are called *intelligent user interfaces*, since they are said to possess some sort of intelligence.

In order to create this intelligence, designers describe the desired interface behavior through examples. That is, the designers give concrete examples of the behavior rather than having to deal with an abstract sequencing specification directly [Fra95]. This concept is called *interface design by demonstration*.

5.2 General Approach

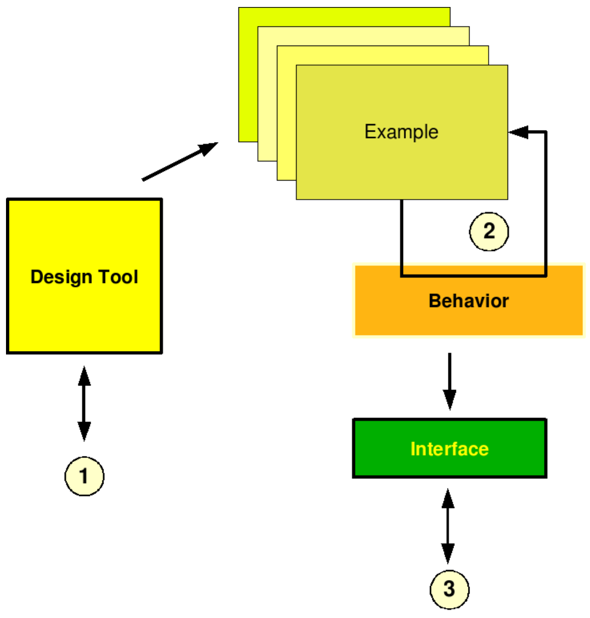


Figure 5.1: The general principle of demonstrational user interface designers

Most demonstrational tools follow the procedure illustrated in Figure 5.1. First the designer needs to interact with the design tool in order to provide the expected examples. Different tools use different methods to collect examples, for example stimuli response (DEMO [WF91]), sketching (Monet [LL05]) and before- and after snapshots (Inference Bear [FSF95]). In step two, (2) in Figure 5.1, these examples are used to learn the desired behavior. Popular learning techniques are linear generalization (DEMO [WF91]), reasoning engines (Inference Bear [FSF95]) and radial based function networks

(Monet [LL05]). This behavior is used to respond on the actions made by the user on the final interface, (3) in Figure 5.1. A more detailed description of some demonstrational designer tools is provided next in section 5.3

5.3 Demonstrational Interface Design Tools

The DEMO system [WF91] allows the developer to draw the graphical components of an interface using a standard drawing editor. Once the graphics are drawn, the developer then defines the behavior of the interface using stimulus-response demonstration. Stimulus-response demonstration is designed to be a simple and intuitive technique for defining the behavior of an interface. In using the technique, the developer first plays the role of an end user by demonstrating a stimulus that represents an action that an end user will perform on the interface. Fallowing the stimulus, the developer plays the role of the system by demonstrating the response(s) that should result from the given stimulus. DEMO does not use predefined widgets such as buttons and sliders but drawing primitives such as lines, circles and rectangles. The possible stimuli are atomic mouse events such as a mouse button presses and releases. For example, it is possible to specify that a line at a fixed position should appear when the left mouse button is pressed by specifying the mouse press as the stimulus and then drawing the line in response [Fra95].

Inference Bear [FSF95] (An **Inference** Creature based on **B**efor**e** and **A**fte**r** Snapshots) is built on top of a domain-independent reasoning engine. The examples provided as input to the engine consists of "before" and "after" states which describe a change of state, and of a parametrized event which describes when this change of state should occur at run-time. The output of the engine consists of an algorithmic description of how the state changes in response to the event. For example, it is possible to specify that a button moves one button length to the right every time it is clicked. In this example, the before snapshot is the button's original position, the after snapshot is the original position moved one button length to the right, and the click event is the description of when the transition should occur. Interference Bear provides an intuitive interface to record state changing events and the before- and the after states. Many examples can be given to describe the same behavior, in order to have an accurate training of the reasoning engine.

Monet [LL05] is a sketch-based tool for prototyping *continuous interactions* by demonstration. In contrast to discrete interactions that are triggered by events such as mouse click, continuous interactions give continuous feedback in response to the user's continuous input. In a climate viewer, for example, a floating bar can be moved freely on a map with its height changing to reflect the rainfall at a particular position on the map. In Monet, designers can prototype continuous widgets and their states of interest using examples. These examples are provided by sketching the state of the widgets and the mouse position, in order to estimate the continuous behavior. For example, the pointer of a clockwise widget can be sketched at different positions near the mouse cursor, which will infer a draggable pointer behavior. In the interaction mode the trained behavior can be tested and the set of examples can be refined to adjust the training. The learning technique used by Monet is based on *Radial Based Function Networks* (RFN).

5.4 Strengths and Weaknesses

User interface design by demonstration has several strengths and weaknesses. Each of these are covered in this section, mostly based on the overview in [Fra95].

5.4.1 Strengths

The primary strength is that giving examples of desired behavior takes less cognitive skill than formally specifying the same behavior. While learning how to give examples of behavior also takes skill, it compares favorably with having to learn a formal language [Fra95]. Another strength of programming by demonstration is that giving interactive examples is less intimidating then textually specifying the equivalent behavior. Visually-oriented designers in particular are more likely to be willing to give examples than they are to invest time up-front learning a formal language [Fra95].

5.4.2 Weaknesses

Programming by demonstration also has its drawbacks. For one, there are inherent theoretical limits on its expressive power [Fra95]. To understand this issue, imagine that you are defining a complex mathematical function by providing a number of its points. An external observer can now draw a curve that goes trough all of these points, but she can only approximate the function that you had in mind.

The practical limits on expressive power of the approach are even more severe because the number of examples for defining a single behavior must be very small (less than, say, ten) [Fra95]. Hence, assuming that the system has no prior knowledge of what will be demonstrated, it cannot infer complex behavior at all. Giving the system prior knowledge of what will be demonstrated does not solve this problem either because it is impossible to anticipate all possible demonstrations in a complex domain.

Another problem with programming by demonstration is that one cannot have complete confidence in what has been inferred without inspecting a static representation of the inferred behavior [Fra95]. Inference Bear [FSF95] for example, directly shows a static representation as the algorithmic output of the training process, which can be used by experts to verify the learned behavior. In most tools, however, there is only a test-drive mode available to test the behavior with a few cases. Yet, one can never have complete confidence in an inference without examining it in a symbolic form. Monet [LL05], for example, only provides an interaction mode which can be used to test the behavior.

Finally, demonstrating all of the behavior for a large design can be frustratingly tedious [Fra95] . This is because by demonstration alone the designer cannot easily define a common behavior once and then parametrize it for the future.

5.5 Conclusion

In this chapter, the specification of interface behavior by demonstration was introduced. Demonstrational techniques can be used to describe many types of behavior interactively, however, its difficult to demonstrate the complete behavior of a complicated interface. In this thesis, demonstrational interface design serves as a reference technique, which permits to describe a tough matter by using simple interactive techniques.

Chapter 6

UIML - User Interface Markup Language

The *User Interface Markup Language* (UIML) is an XML language that permits a declarative description of a user interface in a highly device-independent manner [AP99]. The goal of UIML [AH04] is to create an open standard user interface description language in XML that can be freely implemented by anyone. The motivation is to facilitate better tools for creation of user interfaces that work on any platform available today, but which also allow today's legacy user interfaces to evolve to new forms for use on platforms that are created years from now.

UIML is a meta language which contains only generic terms (part, property,...) instead of a specific abstraction (e.g. window, button, ...). A generic term can be used for any possible UI element and can support any metaphor. This allows UIML to prepare for the future, when new ways of interaction could be introduced. Yet, specific abstractions would restrict the use of UIML to a certain way of interaction (e.g. a specific modality, metaphor or device). Of course these generic terms must still be mapped on the right UI elements: this is accomplished by including a vocabulary section in the UIML document. The use of a vocabulary provides great flexibility: it allows one to build a custom user interface language on top of UIML, similar to the way the XML meta-language supports building a custom data format.

6.1 UIML Document Structure

6.1.1 General

The structure of a UIML document (fully described in [AH04]) is displayed in Figure 6.1. An UIML document is composed of the `peers` (also

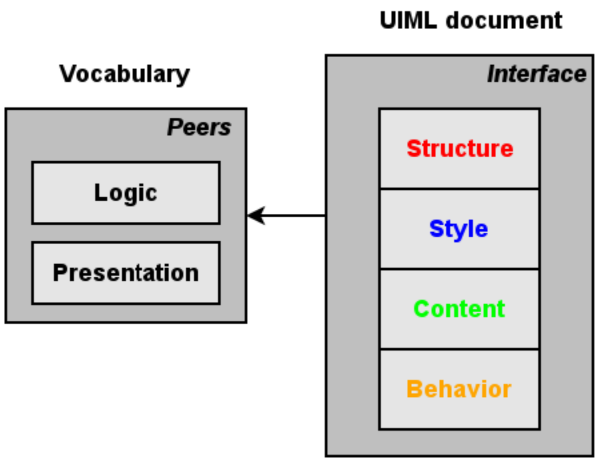
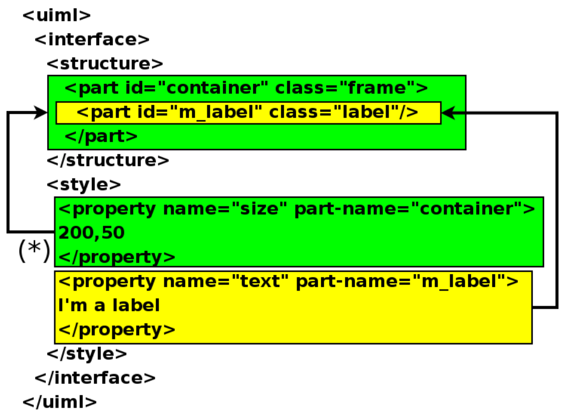


Figure 6.1: The global UIML document structure



```
<uiml>
 <interface>
  <structure>
   <part id="container" class="frame">
    <part id="m_label" class="label"/>
   </part>
  </structure>
  <style>
   <property name="size" part-name="container">
   200,50
(*) </property>
   <property name="text" part-name="m_label">
   I'm a label
   </property>
  </style>
 </interface>
</uiml>
```

Figure 6.2: An example of the connection between the `style` and `structure` section

known as the vocabulary section), which defines the mapping between the UIML tags and a widget set, and the `interface` section, which describes the user interface itself. Each following paragraph discusses a component of the UIML document structure.

6.1.2 Structure

The `structure` element is encapsulated in the UIML document's interface section, as shown in Figure 6.2. It defines the initial organization of the interface represented by the UIML document. "Organization" means the set of UI widgets that are present in the interface, and the containment relationship of those widgets to each other when the interface is initially rendered. This organization can be envisioned as a virtual tree of `parts` with each `part`'s associated content, style, behavior, etc. attached to it. An example of the `structure` `part` tree is shown in Figure 6.2, the arrows show the connection between the `part` and `style` properties. A `part` contains :

- A unique **identifier** to associate the part with the right style, content, behavior, ...
- A **class name** which will be mapped by the vocabulary on a concrete widget of the chosen widget set.

6.1.3 Style

In the `style` section of a UIML document, which is also located within the interface (as shown in Figure 6.2), there are several style `properties` defined about the different `parts`, eg : the size of a widget, position, color, border, ... Normally the `style` and `structure` are strictly separated, though UIML provides a mechanism to declare the `style` directly into the `part` tree (shown in Listing 6.1), which is categorized as the *inline style definition.*

Listing 6.1: `style` declared in the `part` tree

```
<UIML>
  ...
 <structure>
  <part class="Button">
    <style>
      <property name="label">Click me!</property>
    </style>
  </part>
 </structure>
  ...
</UIML>
```

A `property` contains following attributes:

- **name :** The name of the `property` to be set, eg. : size, color, ...
- **part-name :** The `part-name` attribute contains the identifier of a certain `part` element. This implies that the `property` element is connected to a `part` element.

The value inside the `property` tag contains the value to be set to the `property`. In Figure 6.2 we can reform the `property` marked with (*) to : *"set the **size** of the part with identifier **container** to **200,50**".*

6.1.4 Behavior

The behavior element describes what happens when an end-user interacts with the user interface. An example that describes what happens when a user clicks on a button is shown in Listing 6.2. The behavior element is a list of rule elements, where each rule contains two main elements.

1. **condition :** The condition specifies when the action, which is the other child of the rule element, should be undertaken. In Listing 6.2 the condition is the *ButtonPressed* event, which will be fired when the *hello* button is clicked. Events from the concrete widgets in the chosen widget set, are mapped on UIML event classes by the vocabulary.

2. **action :** The action that will be performed when the condition evaluates to true. The action in Listing 6.2 changes the text of the *helloLabel* part into *"Hello World!"*.

Listing 6.2: The behavior element describes end-user interaction

```
<behavior>
 <rule>
  <condition>
   <event class="ButtonPressed" part-name="hello"/>
  </condition>
  <action>
   <property part-name="helloLabel" name="text">
    Hello World !
   </property>
  </action>
 </rule>
</behavior>
```

6.1.5 Logic

The logic element is the first child of the peers section. The peers section is also called the UIML vocabulary, it's responsible for the mapping between UIML and the underlying layers. One part of this mapping, the one between the application logic and the user interface, is done by the logic element. It describes the calling conventions for methods in the application logic that the UI invokes. These methods could appear in different forms: OO-languages (Java, C++, C#, ...), CORBA objects, programs, legacy systems, server-side scripts, databases, ... A description of a power method with 2 arguments is given in Listing 6.3. The most important stakeholders in this example are:

- **d-component :** This element represents a set of d-methods, the *maps-to* attribute specifies the platform-specific type of component that is being bound. In the example in Listing 6.3 the d-component maps to the .NET System.Math class and will be available through the Math identifier for other UIML elements.

- **d-method :** The `d-method` element maps its identifier to the corresponding method of the parent component. In the example, the `power` method inside the `Math` component is mapped to `System.Math`'s `Pow` method.
- **d-param :** Passing parameters to the application logic is done with the `d-param` element. In Listing 6.3, two `System.Double` parameters, identified by `ground` and `power`, will be passed to the `System.Math.Pow` method.

Listing 6.3: The UIML description of the power method

```
<logic>
 <d-component id="Math" maps-to="System.Math">
  <d-method id="power" return-value="double"
   maps-to="Pow">
   <d-param id="ground" type="System.Double"/>
   <d-param id="power" type="System.Double"/>
  </d-method>
 </d-component>
</logic>
```

Once the mapping between UIML and the application logic is specified, the logic can be called with a `call` element. In Listing 6.4, the `power` method defined in Listing 6.3 is called with parameters `8` and `2`.

Listing 6.4: The `call` element

```
<call name="Math.power">
 <param name="ground">8</param>
 <param name="power">2</param>
</call>
```

6.1.6 Presentation

The `presentation` element is the other component of the `peers` section. This element translates the underlying toolkit or widget set to UIML and vice-versa. It defines the legal class names for parts and events in a UIML document, as well as the legal property names.

As an example, part of a button in the System.Windows.Forms vocabulary is given in Listing 6.5. The `presentation` element is a container of `d-class` elements. Each `d-class` binds a platform specific component to a name. In Listing 6.5 the `Button` name is bound to the `System.Windows.Forms.Button` class trough the `maps-to` attribute. Each `d-class` contains several `d-properties`, which represent a property or an event of the parent `d-class` component.

Listing 6.5: A button in the SWF 1.0 vocabulary

```
<presentation>
 ...
```

```
<d-class id="Button" used-in-tag="part"
 maps-type="class"
 maps-to="System.Windows.Forms.Button">
 <d-property id="label"
  return-type="System.String"
  maps-type="getMethod" maps-to="Text"/>
 <d-property id="label" maps-type="setMethod"
  maps-to="Text">
  <d-param type="System.String"/>
 </d-property>
  ...
 <d-property id="clicked" maps-type="event"
  maps-to="Click">
  <d-param
    type="System.Windows.Forms.Control.OnClick"/>
 </d-property>
  ...
</d-class>
...
</presentation>
```

6.2 The Uiml.net renderer

6.2.1 Introduction

Uiml.net[1] is a free software UIML renderer written in C#[2], developed at the *Expertise Centre for Digital Media* (EDM) [LC04]. It can render a UIML document using different widget sets and different platforms. Uiml.net supports Gtk#[3], System.Windows.Forms[4], System.Windows.Forms on the Compact .Net Framework, and a small part of Wx.Net[5].

The most mature implementation of a UIML renderer is the one provided by Harmonia, which is implemented in Java. However, at the time the .NET framework offered some new possibilities to develop a UIML renderer [LC04]. For example it supports on-the-fly executable code generation and better integration with web services. The Uiml.net project is the first attempt to write a UIML renderer for the .NET framework.

6.2.2 Architecture

The Uiml.net renderer , as described in [LTVC06], consists of one rendering core and multiple rendering backends that contains code that is only used by a specific vocabulary. A rendering core can process a UIML document and builds an internal representation of the UIML document. Since

[1]http://research.edm.uhasselt.be/kris/projects/uiml.net/
[2]http://www.ecma-international.org/publications/standards/Ecma-334.htm
[3]http://gtk-sharp.sourceforge.net
[4]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemWindowsForms.asp
[5]http://wxnet.sourceforge.net/

the mapping from abstract interactors to concrete widgets is defined outside the renderer, in the `peers` section (described in 6.1.6 and 6.1.5), it can be loaded dynamically and applied at runtime to the rendering core. The rendering backends have a very limited responsibility in the rendering process : they process the parts of a UIML document that can only be accomplished by widget-set specific knowledge.

[LTVC06] describes the rendering process as three different stages of processing :

pre-processing : during this stage, a UIML document can be transformed into another UIML document.

main processing : during this stage, a UIML document will be interpreted and a concrete instantiation of the document, using the UI toolkits that are available on the target platform, will be generated.

post-processing : during this stage the runtime behavior strategies of the UI will be selected.

The main processing stage is more specifically composed out of 4 steps :

1. The UIML-renderer takes a UIML document as input, and looks up the rendering backend library that is referred to in the UIML document.
2. An internal representation of the UIML document is built. Every part element of the document is processed to create a tree of abstract interface elements.
3. For every part element, its corresponding style is applied.
4. For every part element, the corresponding behavior is attached and the required libraries to execute this behavior will be loaded just-in-time.
5. The generated tree is handed over to the rendering module: for every part tag, a corresponding concrete widget is loaded according to the mappings defined in the vocabulary and linked with the internal representation. For the generated concrete widget, the related style properties are retrieved, mapped by the vocabulary to concrete widget properties and applied on the concrete widget.

6.3 Conclusion

As discussed earlier in the problem description, it is hard to construct platform-independent interfaces. After the discussion of UIML throughout this chapter, one can conclude that UIML is well suited to reach the required platform-independence. When one can construct a design tool which generates a UIML description of the designed interface, this description can be used to render the interface on different platforms. To accomplish this, a UIML renderer for the .NET platform, Uiml.net, was also presented in this chapter.

Part III

Development

Chapter 7

Domain-specific User Interface Builder

Contents

So far, several techniques were discussed to create a platform-independent design tool. Most of these tools describe the interface on a higher level of abstraction. Despite the platform independence which can be reached with these tools, many of them have serious usability problems since the gap between the mental model of the designer and the presentation the tool offers is too big. Most designers prefer to have a concrete representation during their design activities.

A more detailed evaluation of different types of design tools was given in Chapter 2. From this evaluation, it was concluded that both model-based and GUI builders have several strengths. GUI builders permit to edit a concrete representation of the interface during the design process, while model-based and automated techniques allow to reach the required platform independence. Thus, a combination of both tools will led to a new more flexible generation of design tools.

In this chapter, the main work of this thesis is introduced. A new type of design tool is presented, called the *Domain-specific User Interface Builder* (DSUIB), which combines the strengths of MBUID tools and GUI builders. Depending on the domain considered, these tools are adapted in order to facilitate the development of user interfaces for that domain. The tool will contain domain-specific abstractions, however, the designer can always edit a concrete representation of these abstractions. This implies that he is never forced to work with abstract models.

The origin of domain-specific user interface builders can be found in *Domain-Specific Visual Language* (DSVL) editors, e.g: Microsoft's Visio which contains multiple domain-specific toolboxes. While DSUIBs are specialized in creating user interfaces, DSVL editors are able to construct graphical models to describe the considered domain.

After an introduction to domain-specific visual language editors, the various aspects of the GUI domain are discussed. Next, the general characteristics of domain-specific user interface builders are introduced. After these preliminaries, the *Domain-specific User Interface Builder for the User Interface Markup Language* (DSUIB-UIML) is discussed in detail.

7.1 Domain-specific Visual Language Editors

7.1.1 Domain-specific Visual Languages

Domain-specific Languages (DSL) are tailored to a particular problem domain [EJ01] . Through the appropriate use of notations and abstractions, they provide the expressive power to better describe specific solutions to domain-related problems. These solutions can be expressed at an appropriate level of abstraction, and in the language of the problem domain, employing the concepts familiar to practitioners. In contrary to solutions expressed in a general-purpose language, which will be created by programmers with little domain knowledge, DSL programs can be created by domain experts.

The costs of educating DSL users as well as of designing, implementing and maintaining a DSL are high. In principle, visual languages are often better suited to use as domain-specific notations [EJ01] . For a large number of specialist applications or problem domains, there exists a natural and intuitive visual representation of artifacts in these domains. The combination of DSLs and these visual representations are called *Domain-specific Visual Languages* (DSVLs), which make it possible to describe complex information in a particular domain through a visual metaphor. DSVLs are more efficient an effective than general-purpose modelling languages [GHZL06].

Mostly, different domains use different DSVLs, each modelled DSVL uses a different modelling formalisms. A proven method to achieve the required flexibility for a modelling language that supports many formalisms and modelling paradigms is to model the modelling language itself [VdL02] . Such a model of the modelling language is called a *meta-model.* It describes the possible structures which can be expressed by the DSL and thus the corresponding DSVL. Tools which are able to create meta-models are called *meta-tools.* Consequently, one can consider a meta-tool as a tool to generate DSVL design tools.

7.1.2 Flexible Domain-specific Visual Language Editors

A naïve approach will construct one editor for each DSVL, the rationale being that a special-purpose tool using a special-purpose language is better than a general-purpose tool/language [GHZL06]. Although the construction of a tool for each DSVL is supported by several toolkits and frameworks [GMH00], it still is a very tedious and time-consuming activity. Therefore, a new approach will generate DSVL editors automatically from its meta-model, which can be defined by a *meta-tool.* Tools following this approach are discussed below.

Marama [GHZL06] allows users to rapidly specify or modify a desired visual language tool using an existing meta-tool, Pounamu, and then have the tool realized as a high-quality Eclipse-based editing environment. Marama editors look and feel like other Eclipse graphical editors, use Eclipse code generation support, and can be integrated with and extended by other Eclipse plugins. Microsoft's *Domain-specific Language Tools* [Jon05], integrated in the Visual Studio development platform, facilitate the creation of modeling languages, by a built in meta-tool, and generates a *Visio-like* editor for each language. *MetaEdit+* also contains a meta-tool which makes it possible to define custom domain concepts as the modeling elements, their properties and associated rules. These specifications will be used to generate a custom CASE tool.

7.2 The Graphical User Interface Domain

A GUI domain corresponds to the domain for which the GUI is created. A common way to describe this domain is by a *domain model* (see Section 2.3.2), which defines the objects that a user can view, access and manipulate trough a UI [BVE02]. A domain vocabulary specifies the naming scheme and visual representations of the domain objects (see Section 7.2.2).

7.2.1 Domain Objects

The *Concepts Model* [CCT$^+$02] captures information related to the domain of discourse: in this sense, it is similar to domain modelling. Because of this similarity, the terms *domain object* and *concept* can be exchanged. Throughout this text, domain object is preferred to concept because it emphasizes the relation with the domain for which the user interface is created. For example, a simplified music-player contains the domain objects: playlist, timer and song as shown in the domain model in Figure 7.1. As a concept in a domain model, a domain object can also have several attributes.

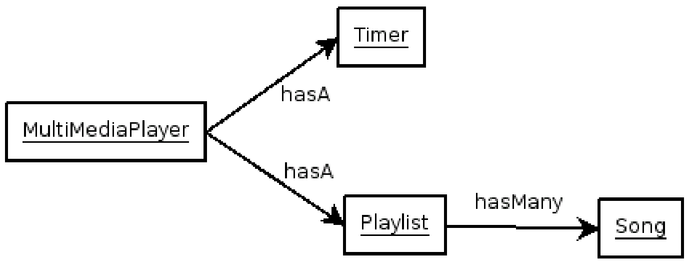


Figure 7.1: The domain model of a multimedia player

7.2.2 Domain Vocabulary

The domain vocabulary serves as the meta-model, which specifies the building blocks, to construct the GUI domain model. A domain vocabulary contains a set of domain objects that are commonly used in this GUI domain and a corresponding presentation of each of them. Thus, this vocabulary is responsible to map the domain objects to a visual presentation, called a component. The domain-objects used in the domain model of a music player are linked to a presentation in the MMVocabulary01 and MMVocabulary02 domain vocabularies as shown in 7.2. Beside the visual presentation of domain objects, the domain vocabulary also specifies the attributes which can be attached to a domain object.

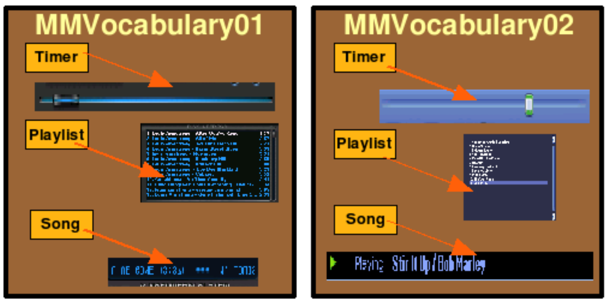


Figure 7.2: Several domain vocabularies for the music player domain

7.3 Domain-specific User Interface Builders

Domain-specific User Interface Builders are a new type of tools, which combine the flexibility of DSVL editors with the intuitivity of traditional GUI Builders. They facilitate the creation of a user interface for a specific domain.

7.3.1 General Characteristics

Domain-specific User Interface Builders are tools that allow the creation of flexible GUI designs, based on the domain vocabulary (see Section 7.2.2). A GUI Builder is generated for a specific vocabulary and contains a toolbox with all the graphical representations of the domain objects. These can be combined by drag and drop operations, in order to create an interface for the domain encapsulated in the vocabulary.

Although this approach is comparable with the one of the flexible DSVL editors discussed in Section 7.1.2, there is an important difference between DSUIBs and DSVL editors. With a DSVL editor, one can create abstract models which will be refined towards the final interface. Thus, DSVL editors can be plugged into the model-based design process. This is contrary to DSUIBs, which enable the creation of user interfaces at a concrete level of abstraction. The relation between DSUIBs and model-based design is discussed in Section 7.3.2.

Assume a vocabulary containing domain objects for 'classic' graphical user interfaces, for example a System.Windows.Forms[1] vocabulary, and one for a media-player, for example a vocabulary for the XMMS[2] music player. Typical objects contained in the System.Windows.Forms vocabulary are buttons,

[1]http://msdn2.microsoft.com/en-us/library/system.windows.forms.aspx
[2]http://www.xmms.org/

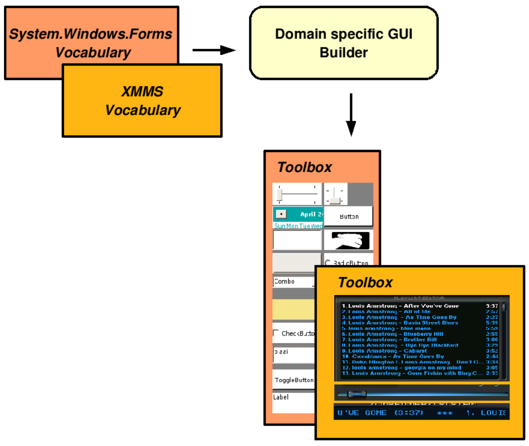


Figure 7.3: The Domain-specific User Interface Builder concept

textboxes, listboxes and panels. The ones encapsulated in the XMMS vocabulary can be a playlist, a timer and songs. Depending on the chosen vocabulary, another GUI Builder is generated for the domain encapsulated in this vocabulary (see Figure 7.3). This example illustrates the flexibility introduced by domain-specific user interface builders: the same tool can easily be used to design interfaces for different domains or abstraction levels.

7.3.2 Relation with Model-Based Design Tools

Although domain-specific user interface builders make use of domain objects and thus a domain model, it is no model-based design tool (which are discussed in Section 2.3). Model-based user interface development [CLC04] uses a multitude of models which are related to another in a certain way. Usually there is some kind of process that starts with the design of the abstract models and progresses gradually towards the more concrete models, resulting in the final interface when the design process is complete.

In DSUIBs, domain objects are directly mapped to the concrete interface level by the domain vocabulary, without progressing through a set of models. When this translation has taken place, the user interface designer should combine these representations in order to create the final interface. Thus, domain-specific abstractions are manipulated on a concrete level of abstraction, without the necessity to think abstractly.

7.4 Domain-specific User Interface Builder for UIML

In this section, DSUIB-UIML is presented, a Domain-specific User Interface Builder which is built upon the Uiml.net renderer (see Section 6.2).

7.4.1 Domain Objects in UIML

UIML is a meta language (see Chapter 6) which describes the mappings between generic user interface terms and concrete user interface elements in a vocabulary section. This makes that a UIML vocabulary can serve as a domain vocabulary, which couples domain objects (the generic user interface terms) to their visual representations (the concrete interface elements). The `d-property` element can be used to specify the attributes (properties in UIML) of the domain objects (UIML parts).

For example, the vocabulary described in Listing 7.1 makes it possible to create user interfaces containing one type of domain object, a button, which has two properties, `label` and `position`. The button will be visualized by a *System.Windows.Forms.Button*[3] component. A user interface described in accordance with this vocabulary, is shown in Listing 7.2: a combination of three buttons, each with several properties.

Listing 7.1: myVoc.uiml

```
<presentation>
<d-class id=‘‘Button’’ used-in-tag=‘‘part’’
 maps-type=‘‘class’’
 maps-to=‘‘System.Windows.Forms.Button’’>
 <d-property id=‘‘label’’ maps-type=‘‘setMethod’’
  maps-to=‘‘Text’’>
  <d-param type=‘‘System.String’’/>
 </d-property>
 <d-property id=‘‘position’’ maps-type=‘‘setMethod’’
  maps-to=‘‘Location’’>
  <d-param type=‘‘System.Drawing.Point’’/>
 </d-property>
</d-class>
</presentation>
```

Listing 7.2: The interface built on myVoc.uiml

```
<uiml>
 ...
 <interface>
   <structure>
     <part class=‘‘Button’’ id=‘‘button1’’/>
     <part class=‘‘Button’’ id=‘‘button2’’/>
     <part class=‘‘Button’’ id=‘‘button3’’/>
   </structure>
   <style>
     <property name=‘‘label’’ part-name=‘‘button1’’>
        Click me!
     </property>
     <property name=‘‘position’’ part-name=‘‘button1’’>
        10,10
     </property>
     <property name=‘‘label’’ part-name=‘‘button2’’>
        Click me tooo !
     </property>
```

[3]http://msdn2.microsoft.com/en-us/library/system.windows.forms.button(VS.71).aspx

```
    <property name=''position'' part-name=''button3''>
      20,20
    </property>
  </style>
 </interface>
 <peers>
  <presentation base=''myVoc.uiml''/>
 </peers>
 ...
</uiml>
```

7.4.2 GUI Builder Components

To facilitate the creation of user interfaces in UIML, the GUI Builder generated from a specific vocabulary should consist of three important dialogs:

- a *toolbox* containing a set of respresentations of domain objects.
- a *canvas* which shows the graphical representation of what the user interface should look like. Domain objects can be dragged from the toolbox to the canvas.
- a *property dialog* which shows the attributes of the domain objects: editing these attributes will result in the corresponding change in representation in the canvas as defined by the vocabulary.

In Figure 7.4, these three dialogs are shown in a GUI Builder generated from the *System.Windows.Forms* vocabulary[4].

Toolbox

In order to create the toolbox, a step-wise procedure is executed:

1. **creation of domain object instances:** for every domain object described in the vocabulary, a default instance is made. The created instance contains a UIML part and a list of properties, initialized with default values. For example, a default instance of the domain object described in the vocabulary of Listing 7.1 is given in Listing 7.3;
2. **off-screen rendering:** some minor changes were made to the Uiml.net renderer in order to allow per domain object rendering. Each domain object is rendered to an offscreen texture. This involves that every domain object is represented independently from the used widget-set or toolkit. A visual representation of a domain object is called a component.
3. **toolbox construction:** every texture is loaded in the designer, which results into the domain-specific toolbox.

[4]http://research.edm.uhasselt.be/ kris/projects/uiml.net/swf-1.1.uiml

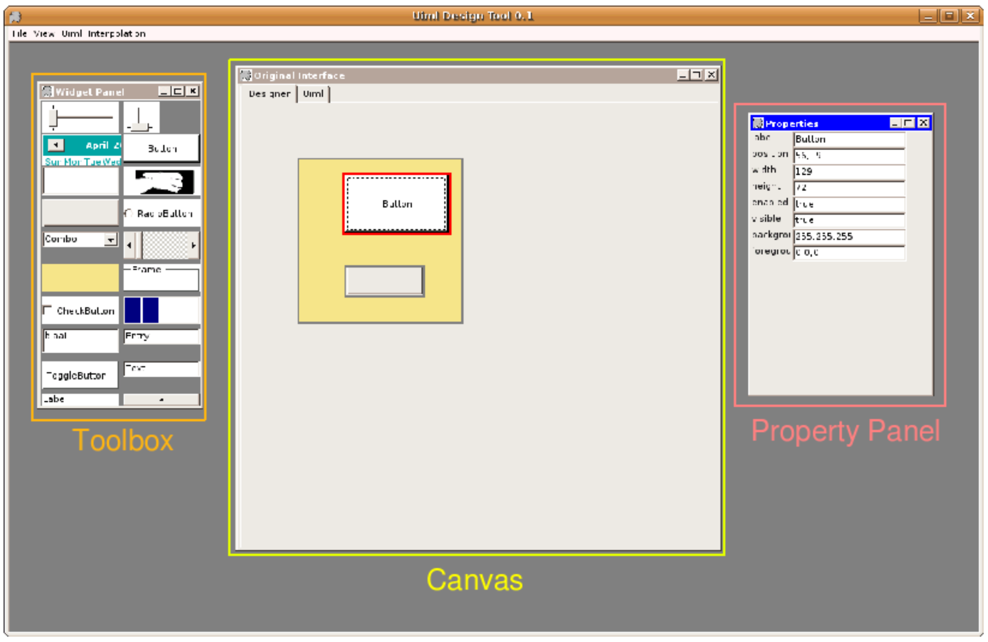


Figure 7.4: The three main dialogs of a GUI Builder, generated from the *System.Windows.Forms* vocabulary

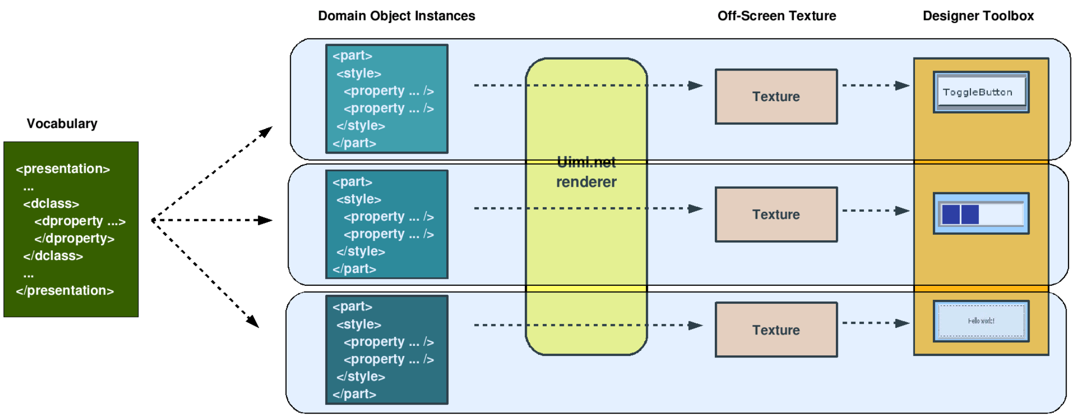


Figure 7.5: For the vocabulary in this Figure, the domain object rendering pipeline is executed three times in order to construct the toolbox

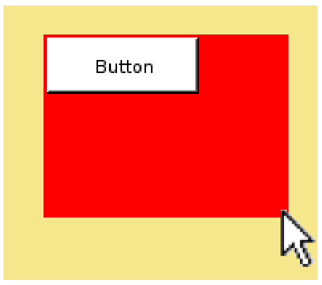
This procedure is illustrated in Figure 7.5. In this thesis, the procedure which creates a component from a domain-object's UIML description, is defined as the *domain object rendering pipeline*. Despite the usage of textures during design-time, the resulting interface will be fully functional during run-time.

Listing 7.3: A default instance of the domain object described in myVoc.uiml (listing 7.1)

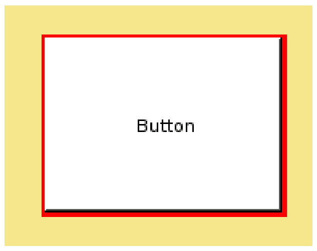
```
<part class=''Button'' id=''part_1''/>
  <style>
    <property name=''position''>
      0,0
    </property>
    <property name=''label''>
      Button
    </property>
  </style>
</part>
```

Canvas

A component, which is selected in the toolbox, can be placed and re-sized on the canvas by direct manipulation. Direct manipulation can be used to change the size or position of the component on the canvas. This is done by updating the `size` and `position` attributes of the domain object represented by the component. After updating these values, the component is re-rendered by executing the domain object rendering pipeline again. Since the continuous re-rendering during the re-size of a component would be too computation intensive, the new size of the component is first visualized by a red rectangle while dragging the mouse (see Figure 7.6(a)), and rendered when the mouse is released (see Figure 7.6(b)).



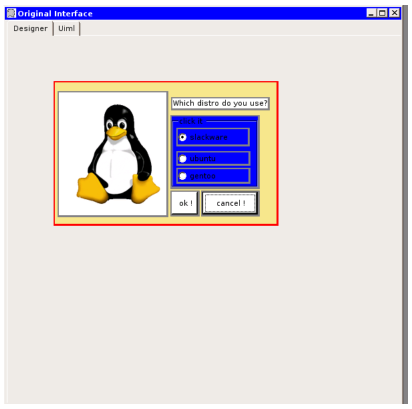
(a) while dragging the mouse



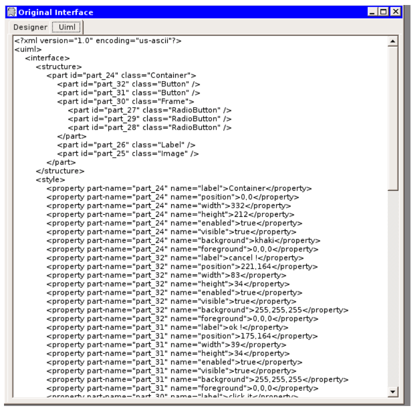
(b) after releasing the mouse

Figure 7.6: Smoothly resizing a component

The canvas contains two tab-pages, each representing another view of the interface. The *designer view* is used to create an interface by direct manipulation operation as discussed above. A UIML-representation of the user interface is given in the *UIML view*. Both views are shown in Figure 7.7.



(a) Design view



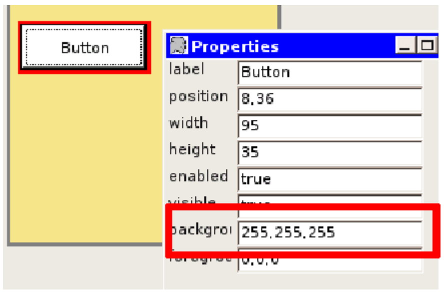
```
<?xml version="1.0" encoding="us-ascii"?>
<uiml>
  <interface>
    <structure>
      <part id="part_24" class="Container">
        <part id="part_32" class="Button" />
        <part id="part_31" class="Button" />
        <part id="part_30" class="Frame">
          <part id="part_27" class="RadioButton" />
          <part id="part_29" class="RadioButton" />
          <part id="part_28" class="RadioButton" />
        </part>
        <part id="part_26" class="Label" />
        <part id="part_25" class="Image" />
      </part>
    </structure>
    <style>
      <property part-name="part_24" name="label">Container</property>
      <property part-name="part_24" name="position">0,0</property>
      <property part-name="part_24" name="width">332</property>
      <property part-name="part_24" name="height">212</property>
      <property part-name="part_24" name="enabled">true</property>
      <property part-name="part_24" name="visible">true</property>
      <property part-name="part_24" name="background">khaki</property>
      <property part-name="part_24" name="foreground">0,0,0</property>
      <property part-name="part_32" name="label">cancel !</property>
      <property part-name="part_32" name="position">221,164</property>
      <property part-name="part_32" name="width">83</property>
      <property part-name="part_32" name="height">34</property>
      <property part-name="part_32" name="enabled">true</property>
      <property part-name="part_32" name="visible">true</property>
      <property part-name="part_32" name="background">255,255,255</property>
      <property part-name="part_32" name="foreground">0,0,0</property>
      <property part-name="part_31" name="label">ok !</property>
      <property part-name="part_31" name="position">175,164</property>
      <property part-name="part_31" name="width">39</property>
      <property part-name="part_31" name="height">34</property>
      <property part-name="part_31" name="enabled">true</property>
      <property part-name="part_31" name="visible">true</property>
      <property part-name="part_31" name="background">255,255,255</property>
      <property part-name="part_31" name="foreground">0,0,0</property>
      <property part-name="part_30" name="label">click it</property>
```

(b) UIML view

Figure 7.7: Multiple views of the canvas

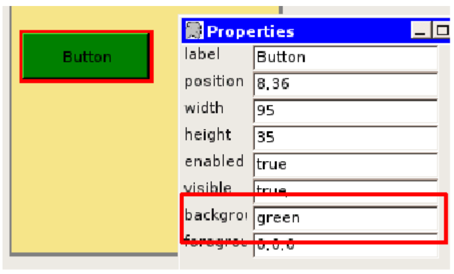
Property Dialog

After selecting a component, the attributes belonging to the component's domain object become visible in the property dialog. For a user interface designer, the term *property* is more familiar than the term *attribute*. Therefore, DSUIB-UIML has opted for properties instead of attributes. In the property panel, the value of every property can be changed, which results in a re-execution of the domain object rendering pipeline to make this change visible. Changing the color property from white to green is illustrated in Figure 7.8.



label	Button
position	8,36
width	95
height	35
enabled	true
visible	true
backgroı	255,255,255
foregroı	0,0,0

(a) The original color



label	Button
position	8,36
width	95
height	35
enabled	true
visible	true
backgroı	green
foregroı	0,0,0

(b) Color changed to green

Figure 7.8: Changing the properties of a component

7.4.3 Managing the Interface Structure

As described in Section 6.1.2, the structure of a UIML user interface can be seen as a virtual tree of parts. In order to manage the structure of this tree, the UIML design tool uses the z-order of the components placed on the canvas. The z-order is an ordering of overlapping two-dimensional objects, such as the components in a graphical user interface. Two important aspects engaged in this approach are:

- the algorithm used to build up the tree;
- a mechanism to manipulate the z-order of each component.

Building up the Tree

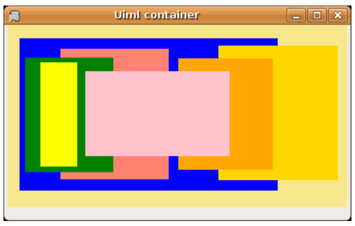
An interface consists out of several components with each a unique z-value. A low z-value indicates that a component lies below a component with a higher one and vice versa. To create a tree from these components, their spatial relationship will be considered: a component a, which is placed upon another component b, will be a child of b when it lies within the bounds of b and there are no disturbing components between both. A disturbing component x lies between a and b, intersects with both but lies not within the bounds of a. The algorithm which follows this approach, is described by the pseudo-code of Algorithm 1.

Algorithm 1 BuildTree

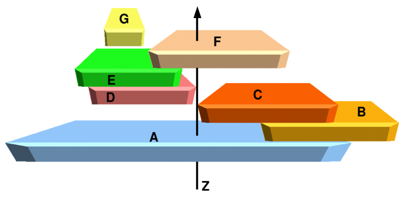
```
Require: A list of components $I$
  Order $I$ by descending z-value.
  Initialize an empty list $P$, in which components which have already a parent will be placed.
  for $x = 0$ and $x < length(I)$ do
    $ComponentA = I[x]$
    for $y = 0$ and $y < x$ do
      $ComponentB = I[y]$
      if $ComponentB$ does not exist in $P$ then
        if $ComponentB$ lies within the bounds of $ComponentA$ then
          if There are no disturbing elements between $ComponentA$ and $ComponentB$ then
            Add $ComponentB$ to $P$
            Set $ComponentA$ as the parent of $ComponentB$
          end if
        end if
      end if
    end for
  end for
```

To clarify this algorithm, assume the interface illustrated in Figure 7.9(a) containing seven colored components, from which the z-value is visualized in

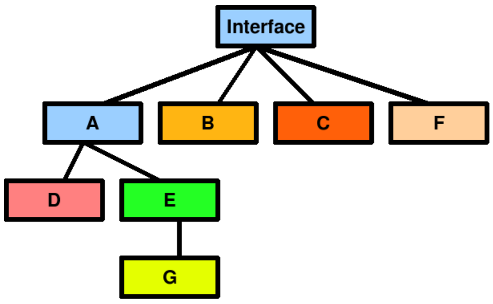
Figure 7.9(b). After the execution of the *BuildTree* algorithm, this interface will correspond to the tree shown in Figure 7.9(c). In this tree, C and F are no children of A, because B acts as a disturbing object between respectively C and A, and F and A. E is a child of A because D is no disturbing object, it lies completely within the bounds of A. The parent of G is E, and not D or A, because the components are investigated in descending z-order. Thus, at first F will be tested as the parent of G , than E, D, C, etc. Once a component found a parent, it will no longer be tested to other components, which involves that G will never be tested for other possible parents once it has discovered E as its parent.



(a) The interface



(b) A 3D visualization of the z-order



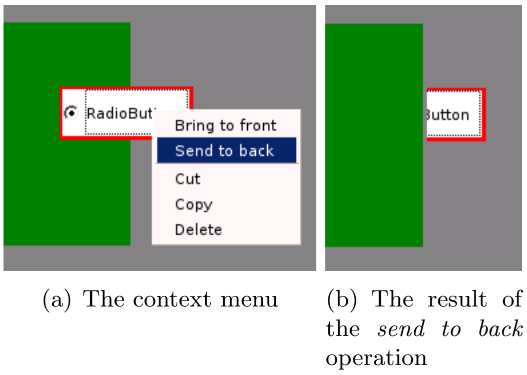
(c) The resulting tree

Figure 7.9: The z-order in a user interface

The algorithm described in this section is executed every time a component is moved, re-sized, added, deleted or z-manipulated.

Z-Order Manipulation

So far, we have created an algorithm which builds a widget-tree from a list of components with each a specific z-value. In order to allow a designer to change the organization of the UI, a mechanism must be introduced to manipulate a component's z-order after it has been places on the canvas. In DSUIB-UIML, this is done in the same way as in traditional drawing programs: by a context-menu which can *bring a component to the front*, or *send it to the back*. These simple operations can manipulate the z-order in an intuitive way. An example of the send to back operation is shown in Figure 7.10.



(a) The context menu (b) The result of the *send to back* operation

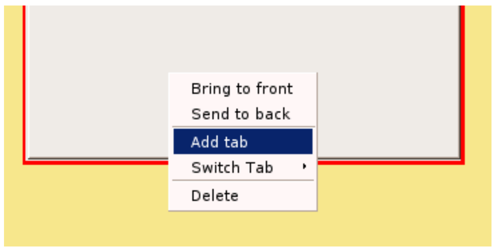
Figure 7.10: Manipulating the Z-value of the *RadioButton*

7.4.4 Serialization and Deserialization

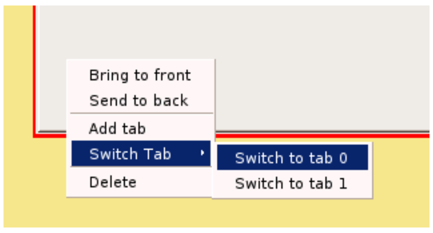
The designed interface is stored in an in-memory tree containing all the domain objects. This tree can be serialized into UIML, or UIML can be deserialized into the tree. The serialization procedure combines the part and property information of this tree into one UIML document. Deserialization follows the inverse procedure, the properties are attached to the parts in order to create the domain objects, which are then stored in the tree. These serialization and deserialization procedures take place when there has been a switch between the design and UIML-view of the canvas and also when a user interface is saved to or loaded from UIML.

7.4.5 Multi-Container Domain Objects

Some domain objects are multi-containers, consisting out of several containers, each holding some child objects. The most well-known multi-container is the tab-container, which holds different tab-pages, each having several children. In order to construct an interface with a multi-container, for example tab-pages, the design tool should be able to: (1) add tabpages to the tab-container, (2) switch between these tab-pages. These operations are encapsulated in the context-menu of the tab-container component, as shown in Figure 7.11.



(a) Add new tab-page



(b) Switch between tab-pages

Figure 7.11: An example of a multi-container: the tab-container with tab-pages

7.4.6 Shortcomings of the UIML Vocabulary

Problems

Although the UIML vocabulary seems to be the ideal solution to describe the domain vocabulary, we came across some shortcomings while designing the UIML user interface builder. The most common problems are:

- the lack of a mechanism to provide *default values* for a d-property in the vocabulary. This would be very practical in the useful of domain object instances for the toolbox;
- the lack of standardization concerning the names of the *size* and *position* properties. Thus it is not clear which property should be updated when resizing;
- not being able to determine which object can serve as a container;
- not being able to determine which object can serve as a multi-container.

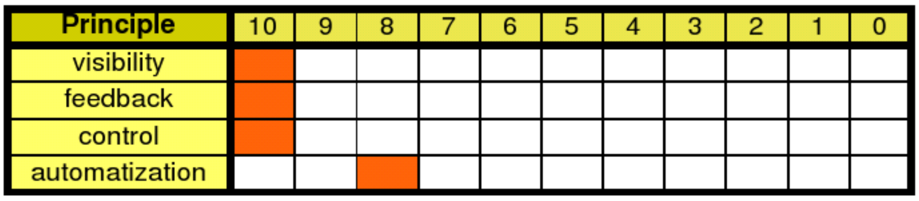
Designer Backends

As the Uiml.net renderer uses rendering backends [LTVC06] to get widget-set specific knowledge during the rendering process, the design tool will use designer backends to get additional information during the design stage. A designer backend is provided by a compiled [?] (DLL), which implements a backend interface trough which information can be accessed. This interface will provide functions to receive the property default values, size and position property names, the container and multi-container identifiers. During the implementation of DSUIB-UIML, a backend for the System.Windows.Forms vocabulary is realized.

7.5 Conclusion

In this chapter, the generation of GUI builders which are able to realize an interface for a specific GUI domain is discussed. DSUIBs provide the *visibility* of GUI builders, they even enhance this visibility by constraining the toolbox to the domain considered. Only components required for that domain can be used to construct the interface, which means that the design tool is not overloaded with general-purpose components. Furthermore, the *feedback* and *control* by DSUIBs is comparable to traditional GUI builders.

DSUIB-UIML generates interface descriptions in UIML, which in theory makes it possible to render these interfaces on multiple devices. However, no mechanism to adapt this description depending on physical constraints, such as an extreme increase in the display-size, is provided. Such a mechanism for DSUIB-UIML will be provided in Chapter 8. The evaluation is summarized in Table 7.1.



Principle	10	9	8	7	6	5	4	3	2	1	0
visibility	🟧										
feedback	🟧										
control	🟧										
automatization			🟧								

Table 7.1: Evaluation of Domain-specific User Interface Builders

Chapter 8

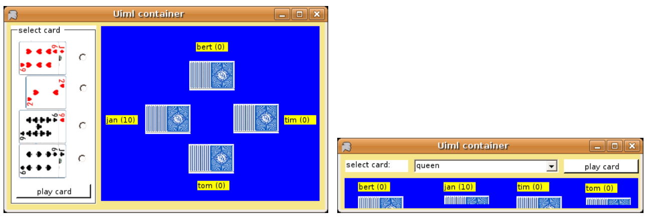
User Interface Interpolation

Contents

In chapter 7, a domain specific user interface builder for UIML was introduced. A user interface, constructed by this tool, can be rendered on multiple devices. For example, a user interface created for the System.Windows.Forms vocabulary can be rendered on a PDA or Desktop computer. The difference in screen size, however, does not always result in a proper rendering of this user interface on a small screen. In order to address this issue, DSUIB-UIML should be extended with an adaptation mechanism.

In this chapter, the term *User Interface Space* will be defined first to describe the adaptations of a user interface depending on the available display size. Next, several existing adaptation techniques will be discussed. Section 8.3.1 introduces a new theoretical model, *user interface based rendering*, to classify adaptation techniques which generate new interfaces from a set of previously designed ones. Within this model, a new adaptation technique is deducted: *user interface interpolation.* Finally, this technique is integrated with DSUIB-UIML and Uiml.net, which will result in a rule-based user interface interpolation mechanism.

8.1 User Interface Space



(a) $UIS(Width1, Height1)$ (b) $UIS(Width2, Height2)$

Figure 8.1: Two outcomes of the card-game *User Interface Space*

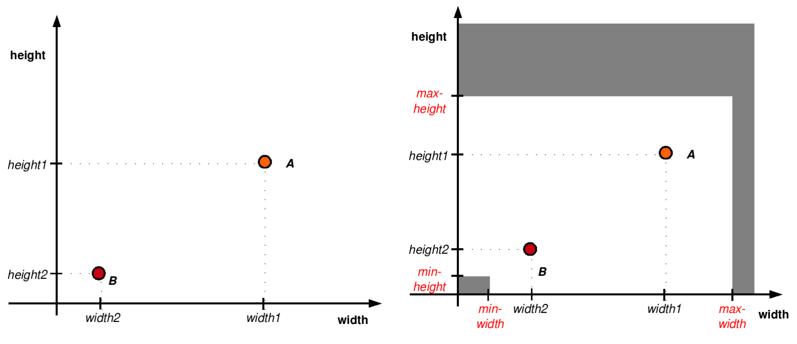
As discussed in section 4.2, user interfaces are adapted for a specific target, which can be the user, the environment or the system's physical characteristics. One important component of the system's physical characteristics is the display size of the user interface. After changing this size, an adaptation of the user interface may be required to preserve the usability [TC99] : for example, when the size shrinks, parts of the user interface may fall outside the display region. To describe the user interface adaptation caused by a change in size, the notion *User Interface Space* (UIS) is introduced.

8.1.1 Definition

In this thesis, we define the User Interface Space of a certain UI as the two-dimensional space which contains all the sizes a certain user interface can reach. In UIS, the UI is parametrized by its width and height, thus the rendition I at an arbitrary width W and height H can be noted as $I = UIS(W, H)$. Consequently, user interface space represents how the user interface should be rendered for every size. Assume for example, the user interface space for a card-game. At two different sizes, the card-game will be rendered like the interfaces shown in Figure 8.1.

UIS can be visualized on a Cartesian coordinate system: the user interface's height is located along a vertical axis, its width along a horizontal one.

For the card-game user interface space, the corresponding graph is shown in Figure 8.2(a). In many cases, user interface space will be constrained by a minimum and a maximum size, which is shown in Figure 8.2(b).

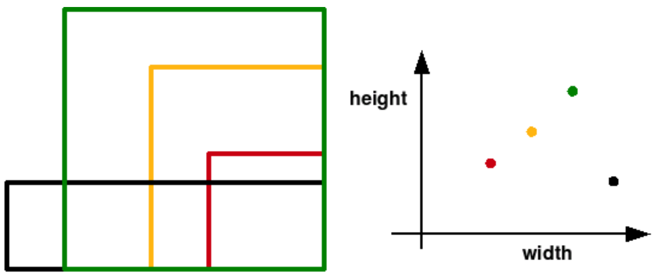


(a) (b)

Figure 8.2: User Interface Space

8.1.2 Adaptive Interfaces

Once the user interface space function is known, it is straightforward to create user interfaces which adapt to the display-size. Assume the display size is changed to $(widthx, heighty)$. To render the appropriate user interface, only $UIS(widthx, heighty)$ needs to be computed. This process is illustrated in Figure 8.3.



(a) Changing the display-size (b) Select the corresponding rendition in user interface space

Figure 8.3: User Interface Space and adaptation

8.2 User Interface Space Covering Techniques

One can create a user interface for every size in UIS. During the rendering stage, the user interface designed for the requested display-size will then be rendered. However, the creation of a user interface for every size in user interface space is almost impossible. Therefore, several *user interface space covering techniques* are pointed out to compute the user interface at every size in UIS automatically. A short review of existing techniques, which can be used as space covering techniques, is given in the remainder of this section.

8.2.1 Automated Layout Techniques

Layout means the process of determining the position and size of each visual object that is displayed in a user interface, and the result of that process. Thus, after the creation of a layout for every size in user interface space, an adapted user interface can be rendered for each size, using this layout. This process is facilitated by the *Automated layout techniques* [LF01], which automate the layouting process. Automated layout techniques can go from the use of *layout managers* included in a UI toolkit, to constraint based automated systems.

A layout manager [LF01] chooses positions and sizes at runtime for the objects that it controls, governed by a set of constraints imposed by a simple layout policy built into the manager and parameters specified by the programmer. Typical layout policies include strict horizontal (row) or vertical (column) layout, row-major or column-major layout, border layout and grid layout. A programmer designs a parametrized layout as a hierarchy of managed containers, chosen for their layout policies, and further constrained by programmer-specified parameters. Thus, a layout manager does not actually design a layout, but rather instantiates a layout at run time from the structure and parameters specified by the programmer.

In a constraint-based automated layout system [LF01], a *constraint solver* takes a constraint network and generates a set of positions and sizes for each of the components in the network. In such a constraint network, constraints are classified as *abstract* or *spatial.* Whereas the former indicate a high-level relationship between two components that are to be included in the layout, the latter enforces positions or size restrictions on the components. Constraint-based automated layout systems may get in trouble when the constraint network is *under constrained.* This means that the constraint solver can generate multiple solutions from the network, for example: the resulting position of a button can be $(10, 10)$, $(0, 0)$ or $(100, 100)$. Some of these solutions may result in a undesired layout, which can harm the usability of the UI. Thus, a constraint network must be created carefully in order to avoid under-constraining. Furthermore, the lack of designer knowledge implies that constraints may result in an undesirable layout.

Although the use of layouting techniques mostly results in a predictable and consistent interface, they can only change the size or position of visual objects in an interface. In many cases, other operations may be more suitable: for example, the transformation of a listbox[1] into a drop-down[2] list when the user interface shrinks; the transformation of a tab-container into a panel when the user interface enlarges. Automated layout techniques are not sufficient when advanced adaptations are required.

8.2.2 Intelligent User Interface Techniques

In chapter 5, intelligent user interface design was introduced. Several intelligent techniques can be used to cover user interface space. A short review of these techniques, related to the UIS covering problem, is provided in this section.

Depending on the screensize included in the device-model and the user, SUPPLE [GW04] will render the most optimal user interface. In contrary to automated layout techniques, which use simple rules to establish relations between components, SUPPLE uses a more complicated, less predictable, optimization mechanism. This results in less control for the designer concerning the final rendition. Yet, Supple can express more complicated operations than automated layouting techniques.

Demonstrational techniques can be used to demonstrate the desired adaptation behavior. However, demonstrating the complete behavior for a large design can be frustratingly tedious [Fra95].

8.3 The User Interface Based Rendering Approach

In this thesis, *User Interface Based Rendering* (UIBR) is defined as the user interface space covering mechanism, which generates a new user interface for an arbitrary screensize from a set of previously created UIs. This resembles in some way to *Image Based Rendering* (IBR), a technique used in the field of computer graphics. Consequently, a simplified introduction to IBR is given first in section 8.3.1. Starting with the IBR technique, we proceed to a description of the UIBR technique.

8.3.1 Image Based Rendering

Assume a photographer walking through a forest. At several viewpoints, he takes some photographs of the nature around him. After his walk, he

[1]a control to display a list of items

[2]a control which enables users to select from a single-selection drop-down list box

comes home and sends his photographs to *National Geographic*[3]. A couple of weeks later, *National Geographic* magazine published some of the photographs he made, however, it seems that these photographs were taken from different positions than the ones he took. The question is as follows, is it possible to generate new photographs of an environment from a set of pre-acquired imagery?

The answer is *yes*. The technique used here is called *Image based rendering* (IBR). In IBR, the *reference images* - the photographs - are considered as samples of the *plenoptic function* [McM97] . This function describes anything which can be seen from any point in space at any time[4]. Figure 8.4 illustrates a 2D plenoptic function for two points in space. The arrows describe the region which can be seen from these points. Once the plenoptic function is known, images can be generated for every point in space. However, the plenoptic function is almost always unknown.

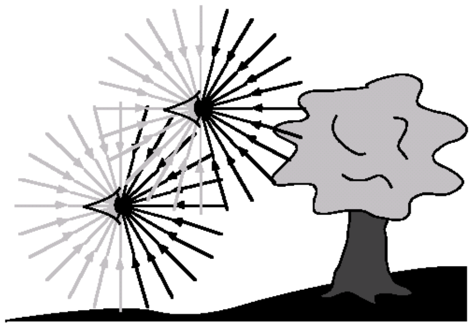


Figure 8.4: The plenoptic function

In IBR, techniques are used to approximate this plenoptic function from the sparse set of samples available. Two approaches which can be followed to obtain a *desired image* from the set of reference images are:

- *fit a model:* one can fit a model from the known data, which can be used to compute the new image from an arbitrary camera position. Lots of models are possible, e.g: a depth map, a triangle mesh, etc .
- *interpolation:* interpolate the known reference images to a new desired image. In image warping [McM97], for example, the visible points in the reference images are mapped to their correct position in the desired image.

[3]http://www.nationalgeographic.com/

[4]This is a simplified definition, for a complete description of the plenoptic function the reader is referred to [McM97]. Beside the plenoptic function, there are also other functions possible (for example [LH96])

8.3.2 User Interface Based Rendering

User Interface Based Rendering is a theoretical model which classifies the techniques that generate a new *desired user interface* from a set of *reference user interfaces*. Remarkable is the similarity with IBR, which generates a new desired image from a set of reference images. As for the difference between both approaches: a desired image is a view of the environment from a *desired position*, while a desired user interface is a UI designed for a *desired display-size*. Furthermore, a desired user interface is created for the same functionality as the reference user interfaces.

To clarify the UIBR technique, an example scenario is provided: assume a designer, who creates two reference user interfaces for a card game. Each reference user interface is designed for a specific display-size, as shown in Figure 8.1. Assume now a user, who utilizes the card game application and re-sizes the user interface to a desired size. For this desired size, UIBR will now generate a desired user interface based on the two previously designed interfaces.

While reference images are considered to be samples of the plenoptic function in IBR, UIBR considers reference user interfaces as samples of the user interface space function. Thus, these user interfaces can be used to approximate the UIS. Once the approximation of UIS is done, user interfaces can be generated for every size in UIS. To create these approximation methods, UIBR adopts the IBR approaches introduced in section 8.3.1. As shown in Figure 8.3.2, two paths can be followed to construct a desired user interface from a set of reference user interfaces: *re-engineering*, which corresponds to the IBR's model fitting approach, and *restructuring*, which is comparable to the IBR's interpolation approach. Each technique is discussed below.

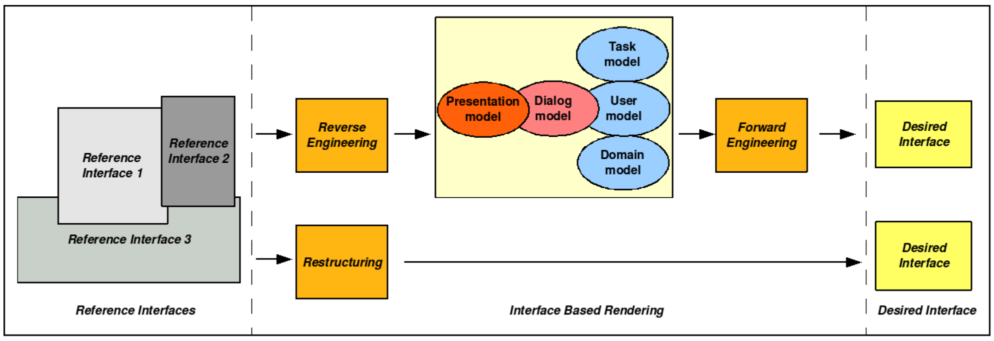


Figure 8.5: User Interface Based Rendering

Re-engineering

The re-engineering technique consists out of two stages: reverse and forward engineering. By reverse engineering [CI90], representations of the user interface at a higher level of abstraction are created: for example, a dialog model, presentation model or task-model. From these abstract models, the desired user interface for the desired display-size will be generated by forward engineering. A re-engineering approach for web pages is presented in [BVE02].

Restructuring

Restructuring [CI90] is the transformation from one representation form to another at the same relative abstraction level, while preserving the user interface's external behavior (functionality and behavior). In this approach, the reference user interfaces will be restructured towards the desired user interface, which omits reverse engineering to a more abstract level. This thesis contributes in the development of a new UIBR restructuring technique, called *User Interface Interpolation.* This approach is described in detail throughout section 8.4.

8.4 User Interface Interpolation

User Interface Interpolation is a new UIBR restructuring approach, which combines the components of two reference user interfaces in a new desired user interface. These two reference user interfaces often have widely varying sizes, for example a 320x240 (PDA) and a 1600x1200 user interface. UIs with intermediate sizes are then interpolated from the two reference ones. Each interpolated user interface serves as an adapted UI for its display size.

To illustrate this technique, assume two reference user interfaces A and B in user interface space, as shown in Figure 8.6. In this Figure, two elliptical regions are constructed around A and B. User Interfaces located in the gray region use components of A, the ones situated in the red region use components of B. Such a region is called the *Interpolation Region* around a reference interface. The intersection between both contains the user interfaces which combine components from user interface A and B. For example, as shown in Figure 8.6: UI $I3$ is constructed from components of B; $I2$ is an interpolation of A and B which contains components of both; $I1$ contains only components of A.

To create a User Interface Interpolation Algorithm, several problems need to be solved:

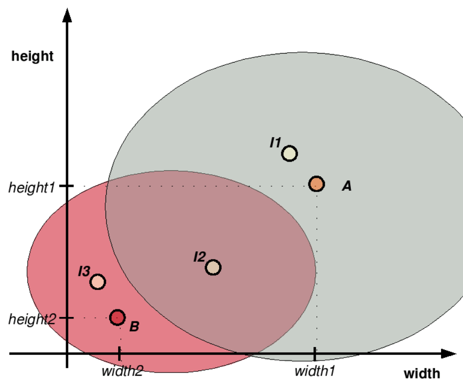


Figure 8.6: Interface Interpolation

- There are an infinite number of interpolation regions which can be constructed around A and B, some of them are listed in Figure 8.7. Thus, an interpolation algorithm must choose the appropriate interpolation regions, to know the components which should be combined at each size.
- Once found which components need to be combined, the question is to find out *how* these are combined?
- Which *temporal dimension* [TC99] is required: run-time adaptation when the display-size changes or static between sessions.

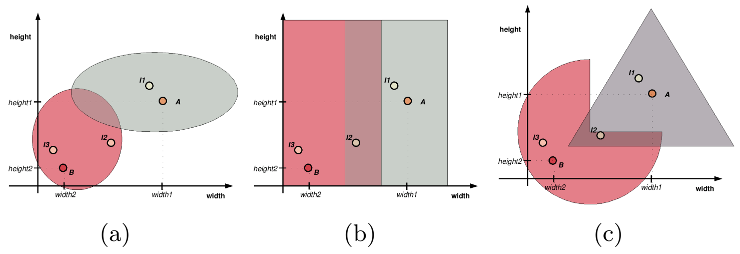


Figure 8.7: An infinite number of regions around the reference user interfaces are possible.

8.5 Rule-Based User Interface Interpolation

Rule-Based User Interface Interpolation is an interpolation technique which is implemented as an extension to the domain dependent user interface builder for UIML (see chapter 6) and Uiml.net (see section 6.2). Throughout this section, the several aspects of this interpolation technique are discussed.

8.5.1 Goals

The rule-based interpolation approach, integrated in the DSUIB-UIML environment, needs to reach several goals. All of these are introduced during this section.

Support for many Adaptations

As discussed in section 8.2.1, the adaptations supported by automated layout systems do not suffice in many cases. Therefore, rule-based user interface interpolation should support a wide range of adaptations:

- Changing a component's *position*;
- Changing a component's *size*;
- Changing a component's mapping, which is called *remapping*;
- Changing a component's *structural location*. The structural location is the location of a component in the user interface tree structure;
- *Adding or deleting* of components.

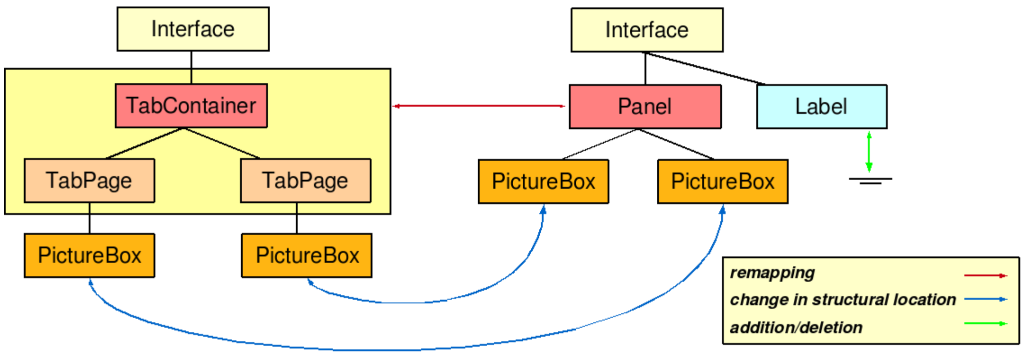


Figure 8.8: Adaptations between a small user interface, on the left, and a larger one, on the right

The change in a component's position and size corresponds to the adaptations supported by automated layout techniques. The other three adaptations are illustrated in Figure 8.8. This Figure contains two user interface tree structures: one of a small interface on the left, and one of a larger interface on the right. The adaptations from the small user interface to the large are visualized by the arrows between their components. In this Figure, a panel component is remapped to a tab container when the user interface shrinks. Yet, when the user interface enlarges, the tabcontainer transforms into a panel. The blue arrow shows a change in the structural position of the two picture boxes: they become a child of a tab page when the user interface shrinks. Otherwise, when the user interface enlarges, their parent is a panel component. The large user interface also contains a label, which is not included in the small one. This is an example of an addition/deletion and is indicated by the green arrow. There exists an inverse relationship between an addition and a deletion: an addition in one user interface will lead to a deletion in the other one and vice versa. Therefore, only additions are considered during this thesis, unless stated otherwise.

Full Control over the Interpolation

The designer should be in full control of the interpolation process. To establish this control, he needs to perform two tasks: (1) creating two user interfaces; (2) specifying which controls are to be used when to constructing an interpolated user interface. The first task should be supported in order to minimize the development effort. In DSUIB-UIML, this is done by *interface cloning*, as described in section 8.5.2.

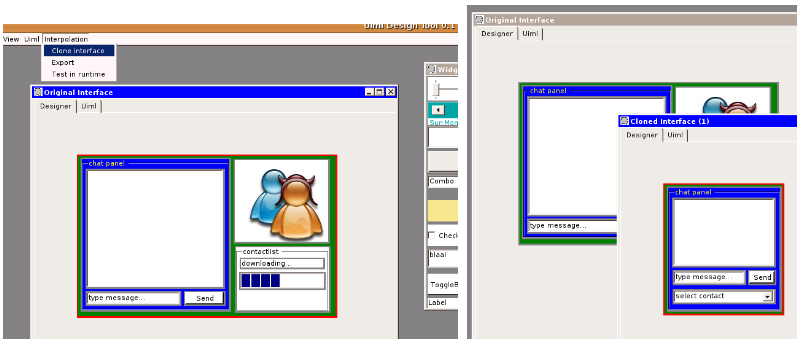
An existing interactive technique, which can be used to perform the second task, is demonstrational user interface design (see chapter 5). However, as discussed before, the demonstrational technique has some serious drawbacks such as the difficulty to demonstrate all the desired adaptations (see section 5.4). Thus, a new flexible and intuitive mechanism to specify the interpolation is used in DSUIB-UIML, which is presented in section 7.

Temporal-dimension Independence

It should be possible to interpolate in a dynamic or static way. For example, when the user interface is built for a desktop PC, runtime interpolation is desirable when the user interface re-sizes. Yet, it should be possible to generate a static interpolated user interface for a fixed screensize, for example for a PDA or smartphone.

8.5.2 User Interface Cloning

A requirement of the user interface interpolation approach is that both reference user interfaces are designed for the same functionality. In many cases, this implies that the second reference user interface will be an adapted version of the first one. To support this technique, DSUIB-UIML contains a *user interface cloning* function, located in the interpolation menu (as shown in Figure 8.9(a)). After the construction of the first user interface, this one can be cloned. The cloned user interface can then be adapted towards the desired screensize, which will finally result in the second reference user interface, as shown in Figure 8.9(b). Thus, user interface cloning reduces the development effort to a minimum, because the second user interface does not have to be built from scratch.



(a) The user interface cloning function (b) The second reference user interface, constructed from the first one

Figure 8.9: User Interface Cloning

8.5.3 Rules

For the different display-sizes in user interface space, the interpolation mechanism needs to know which components it should combine. In DSUIB-UIML, a user-specified rule-set is used to decide this. To describe these rules in detail, a more formal description of the user interface and component concepts is required:

Definition 8.5.1. *A **user interface** I_x is defined as a collection of n components: $I_x = \{C_{x1}, C_{x2}, ..., C_{xn}\}$. Furthermore, each **component** C_{xy}, whith $1 < y < n$, contains k properties P_{xy} : $P_{xy} = \{P^0_{xy}, ..., P^k_{xy}\}$.*

The relationship between components in both reference user interfaces can be defined as a link:

Definition 8.5.2. *A **link** $L^{jb}_{ia} = (C_{jb}, C_{ia})$ establishes a relation between two components C_{jb} and C_{ia}, each located in two different user interfaces I_j and I_i. The former component is defined as the **source component**, the latter as the **destination component**. Every component can be embodied in maximum one link.*

Definition 8.5.3. *A **null-link** $L^0_{ia} = (0, C_{ia})$, with a **null-source component**, is a special link which indicates that C_{ia} is not related to any other component in any other user interface.*

With these definitions in mind, the core principles behind rule-based user interface-interpolation can be explained. Given two reference user interfaces

I_1 and I_2, one can define a rule R as a tuple, $R = (L^{1a}_{2b}, Conditions, P^u_{1a}, P^u_{2b}) : 1 < a < n, 1 < b < n, 1 < u < k$. R will construct an interpolated component by combining the properties of the source and destination component. If the $Conditions$ are met, it uses P^u_{2b}. Otherwise, P^u_{1a} will be employed. In DSUIB-UIML, only four properties are contemplated: a component's size, position, structural location and mapping. Consequently there are *resizing-rules*, *repositioning-rules*, *restructuring-rules* and *remapping-rules*. Yet, there are also *addition-rules*, which contain a null-link. The rule-set, which specifies the components to use at a certain display-size is defined as $R(I_x, I_y)$, and contains all the rules between the components in I_x and I_y.

The rule-conditions specify an interval around a destination component, as illustrated in Figure 8.10(a). If the display-size falls within this interval, the property of this component will be employed. Otherwise, the source component's property will be used. It is not required to specify the interval by both a minimum and maximum size. In most cases, only a minimum width and/or height will suffice. Some alternative intervals are illustrated in Figure 8.10(b) and 8.10(c).

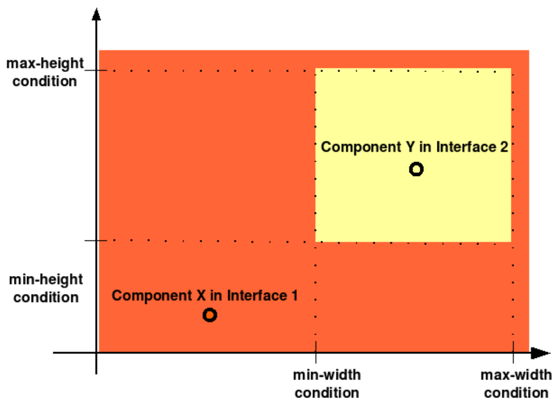
The main goal of rule-based interpolation is to support a wide range of adaptations. By choosing the source and destination components strategically, all the required adaptations can be established. An example of several linked components is given in Figure 8.10(d). The coloured lines represent the links between source and destination components. Further more, each colour represents an adaptation type. The green line represents a re-mapping between the *selecting cards component* and the drop-down list. All orange lines indicate a change in position and/or size between their end-points. A change in the structural location is envisioned by the purple line: the *play card* button can be a child of the panel or the user interface itself. Finally, additions are represented by the red crosses. An addition contains no source component, this component is replaced by a *null source component* (see also definition 8.5.3).

Thus, the rules presented in this section can be used to:

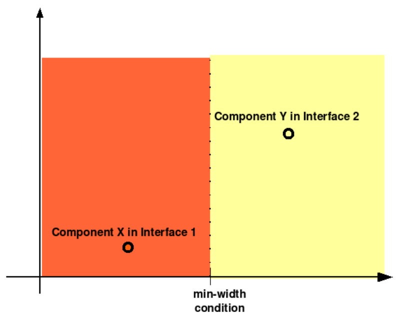
- specify *when* a certain adaptation property should be used, by constructing the right conditions;
- specify all the required adaptations, by selecting the source and destination components strategically.

8.5.4 Rules in DSUIB-UIML

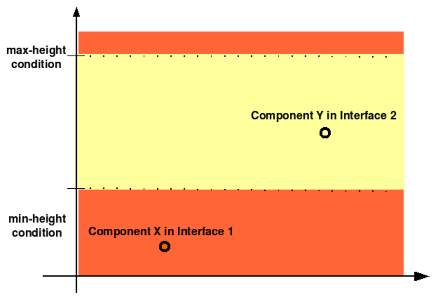
Despite the expressive strengths of rules, it will be a tedious and time-consuming job to construct a desired rule-set by hand. Therefore, DSUIB-



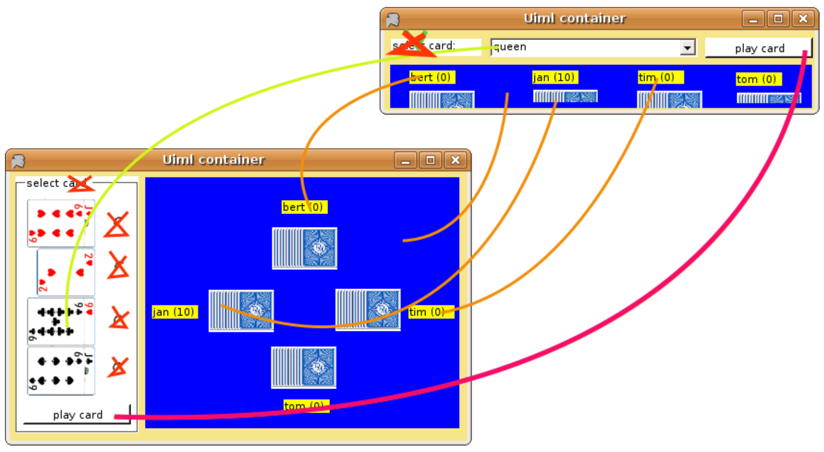
(a) An interval specified by a minimum and maximum size



(b) An interval specified by a minimum width condition



(c) An interval specified by a minimum and maximum height condition



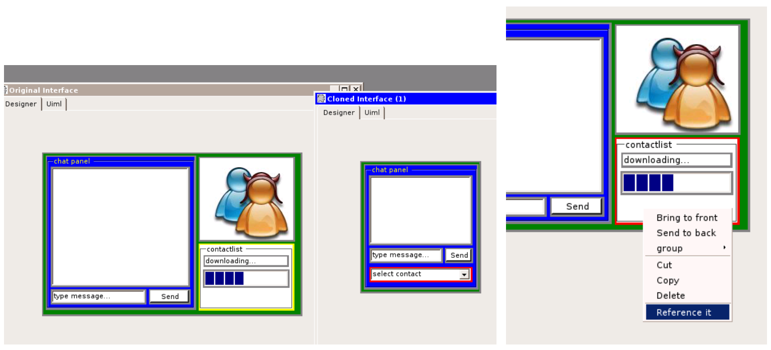
(d) Source and destination components in the card-game

Figure 8.10: Rules are used to divide user interface space

UIML contains some specific features to facilitate the creation and maintenance of a rule-set. This should result in an intuitive technique, which guarantees that the designer stays in full control of the interpolation process.

Specification of Source and Destination Components

The relation between source and destination components is established by *links*. A link, as stated in definition 8.5.2, connects a source component in one reference user interface to a destination component in the other one, as visualized in Figure 8.10(d). The second reference user interface is mostly constructed from a cloned reference user interface (as discussed in section 8.5.2). During the clonation, links are created automatically between the corresponding components of both user interfaces. UIML `parts` which have the same identifier in both user interfaces are defined as corresponding components.



(a) Visualization of a link

(b) Create a link between two components manually

Figure 8.11: Links between components

Although links are generated automatically during the cloning process, it should be possible to create links manually. For example, remappings are created by linking an existing source component to a destination component with another mapping. Therefore, a new *reference it* option is added to the component's context-menu, as shown in Figure 8.11(b). Once this option is clicked, the user can select a component in the other user interface, which will result in a link between these two components.

A visualization of a link between two components is provided when one of them is selected: its counterpart is coloured yellow (see Figure 8.11(a)).

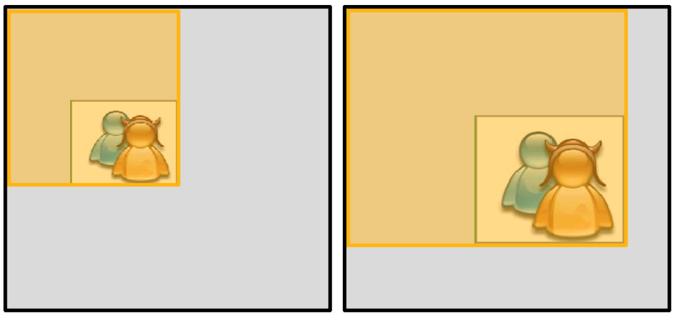
This improves the visibility, the designer can see a component's link anytime he selects it.

Creation of the Rules

Rules are created implicitly by DSUIB-UIML. Implicit creation is preferred to the explicit specification of rules, because it never disturbs the designer in his main activity, namely designing the user interface.

A background procedure inspects the reference user interfaces continuously: it compares four properties between each source and destination component: (1) size, (2) position, (3) structural location and (4) mapping. When a destination component's property differs from its source property, a rule is created for that property. For example, when the user re-sizes a component in one of the two user interfaces, a *re-size-rule* is created.

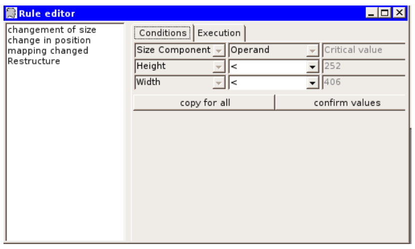
Specification of Conditions



(a) The source component (b) The destination component

Figure 8.12: The construction of a rectangular area around each component

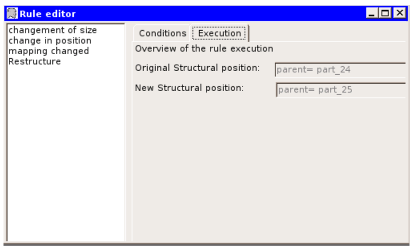
Once the rules are created, conditions should be added to complete these rules. DSUIB-UIML uses a heuristic which computes an initial interval for each rule. This heuristic constructs, for both the source and destination component, a rectangle from the origin of the user interface to the component's utmost corner, as shown in Figure 8.12. Next, the component which is embedded in the largest area is selected. For all the rules drawn for this component, a condition is formulated: the user interface size should be larger or equal than the rectangle size.



changement of size
change in position
mapping changed
Restructure

Size Component	Operand	Critical value
Height	<	252
Width	<	406

(a) The Conditions



changement of size
change in position
mapping changed
Restructure

Overview of the rule execution

Original Structural position: parent= part_24

New Structural position: parent= part_25

(b) The *effect* of the rule when the conditions are met

Figure 8.13: The Rule Editor

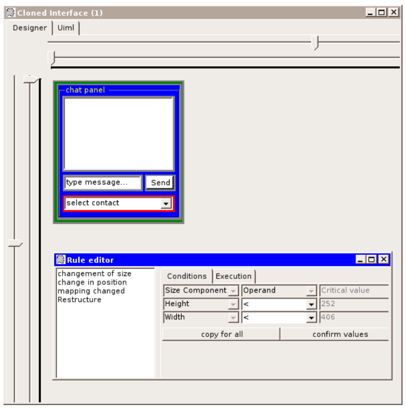
These initial intervals only suffice for simple user interfaces. However, for more complicated user interfaces, the designer should have the ability to customize the conditions. Therefore, DSUIB-UIML contains a built-in rule editor. After selecting a component, this editor shows all the rules applied to this component. For every rule, its pre-conditions and effects can be queried, as shown in Figure 8.13(a) and 8.13(b).

The user can specify the intervals interactively by slider components, which are placed near the user interface. For both width and height, two slider components are available: one for the minimum and one for the maximum value. By setting the slider to zero, the corresponding condition is deleted. Example intervals are illustrated in Figure 8.14(a), 8.14(b), 8.14(c) and 8.14(d). The slider components are preferred to the manual specification of width and height values, because they visualize the interval directly. Consequently, this results in a more natural interaction, without requiring a lot of attention from the designer.

Besides the sliders, Figure 8.13(a) shows a dropdown box, which can be used to modify the operand of a condition to one of the following values: $>, <, \leq, \geq$. Finally, the *copy for all* button copies the conditions to all the rules drawn for the selected component.

8.5.5 The Rule Syntax

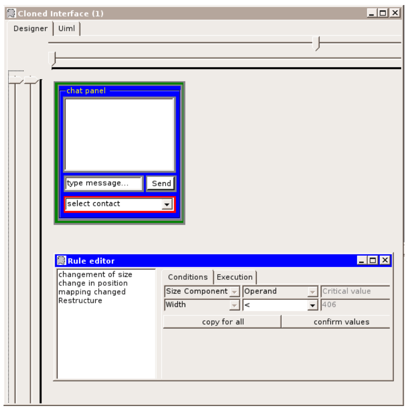
DSUIB-UIML serializes the user interface in standard UIML, which can be rendered by a UIML renderer. Through the renderer's API, one can couple the application logic to the user interface [LTVC06]. Yet, UIML provides no syntax to serialize the interpolation rules. In order to support this, the syntax could be extended, unfortunately, this implies that the tool will no longer generate standardized UIML. Consequently, there will be decided in favour of another approach, which does not affect the UIML syntax.



changement of size
change in position
mapping changed
Restructure

Size Component	Operand	Critical value
Height	<	252
Width	<	406

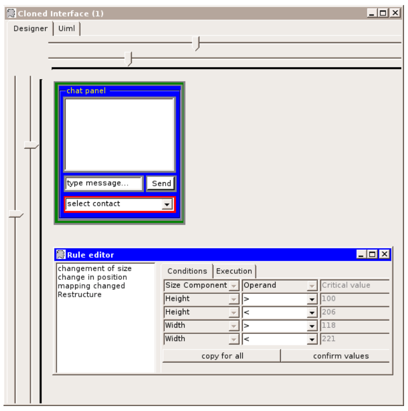
(a)



changement of size
change in position
mapping changed
Restructure

Size Component	Operand	Critical value
Width	<	406

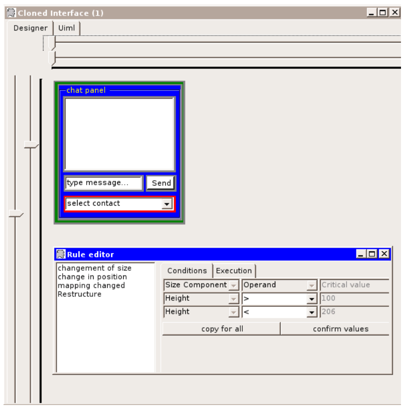
(b)



changement of size
change in position
mapping changed
Restructure

Size Component	Operand	Critical value
Height	>	100
Height	<	206
Width	>	118
Width	<	221

(c)



changement of size
change in position
mapping changed
Restructure

Size Component	Operand	Critical value
Height	>	100
Height	<	206

(d)

Figure 8.14: Four possible conditions for a selected mapping rule. If these are met, the mapping of the selected control will be used . Otherwise, the mapping of it's linked component is employed.

The rule-based mechanism can be accomplished straightforward, using simple elements for making a choice among other elements. Both the XML Schema specification[5] and the XSLT specification[6] define elements for this type of functionality. XML Schema defines the `xsd:choice` element and XSLT defines the `xsl:choice` element. The latter is the most suitable and flexible for the interpolation mechanism [Luy05]. Rule conditions can then be formulated by XPath[7] expressions, which can be embedded in the `xsl:when` statements. A pseudo-syntax to serialize the rules in an XSLT stylesheet is presented in Listing 8.1. The `width` and `height` parameters,

[5]http://www.w3.org/XML/Schema
[6]http://www.w3.org/TR/xslt
[7]http://www.w3.org/TR/xpath

used in the XPath conditions, should be passed by the application logic.

Listing 8.1: A pseudo-syntax for interpolation rules

```
...
<xsl:choose>
 <xsl:when test=`` $height > 100  and  $width < 200''>
   <!-- use the destination component's property -->
 </xsl:when>
 <xsl:otherwise>
   <!-- use the source component's property -->
 </xsl:otherwise>
<xsl:choose>
...
```

In the new approach, DSUIB-UIML exports the user interface into an XSLT stylesheet, which holds the interpolation rules. The stylesheet generates a proper UIML description depending on the screensize. The rendering is discussed in detail throughout section 8.5.6. The remainder of this section is devoted to the serialization of the interpolation rules.

Repositioning and Resizing Rules

The position and size of a component are specified in the `style` section of a UIML document. This means that the repositioning and resizing rules should be defined within this section. An example is provided in Listing 8.2.

Listing 8.2: The serialization of a repositioning rule

```
<style>
 <xsl:choose>
  <xsl:when test=" $height &lt; 252
           and  $width &lt; 406 ">
    <!-- destination property -->
    <property part-name="part_38"
     name="position">9,167</property>
  </xsl:when>
  <xsl:otherwise>
    <!-- source property -->
    <property part-name="part_38"
     name="position">255,148</property>
  </xsl:otherwise>
 </xsl:choose>
</style>
```

Restructuring and Addition Rules

The structure of a user interface is specified in the `structure` section of a UIML document. An example is given in Figure 8.15. In this example, `xsl:otherwise` cannot be used, because the part of the source component appears on a different location in the tree. This is resolved by the construction of an inverse condition around the source component. The example contains also an addition, which has no source component.

```
...
<structure>
 <part id="part_24" class="part_24">
 <xsl:choose>
  <xsl:when test=" $height > 140
                 and $width > 407 ">
   <!-- addition -->
   <part id="part_35" class="part_35"/>
  </xsl:when>
 </xsl:choose>
 <xsl:choose>
  <xsl:when test="$height >= 252 or $width >= 406 ">
   <!-- source component -->
   <part id="part_38" class="part_38">
   </part>
  </xsl:when>
 </xsl:choose>
 <part id="part_25" class="part_25">
  <xsl:choose>
   <xsl:when test=" $height < 252 and $width < 406 ">
     <!-- destination component -->
    <part id="part_38" class="part_38"/>
   </xsl:when>
  </xsl:choose>
  </part>
 </part>
</structure>
..
```

Figure 8.15: The serialization of a restructuring and addition

Remapping Rules

As shown in Listing 8.15, each `part` has a `class` attribute which contains its own identifier. In the vocabulary section, the remapping rules are responsible for the connection between this identifier and a concrete user interface element. An example is given in Listing 8.3. The remapping in this example corresponds to the one between the card-deck (a *System.Windows.Forms.GroupBox*) and a dropdown menu (a *System.Windows.Forms.ComboBox*) as illustrated in Figure 8.10(d) with the green line.

8.5.6 Uiml.net Interpolation Runtime

An XSLT stylesheet transforms an XML document into another XML document. In order to create the final UIML description of the user interface, the generated XSLT stylesheet transforms a *Platform Model* (PM) [Pue97] into a UIML document. This resulting document is a user interface

Listing 8.3: The serialization of a remapping rule

```
<presentation>
...
<xsl:choose>
  <xsl:when test=" $height < 252  and  $width < 406 ">
   <d-class id="part_38" used-in-tag="part"
     maps-type="class" maps-to="System.Windows.Forms.ComboBox">
     <d-property id="label" maps-type="setMethod" maps-to="Text">
       <d-param type="System.String"/>
     </d-property>
     ...
   </d-class>
  </xsl:when>
  <xsl:otherwise>
   <d-class id="part_38" used-in-tag="part"
     maps-type="class" maps-to="System.Windows.Forms.GroupBox">
     <d-property id="label" maps-type="setMethod" maps-to="Text">
      <d-param type="System.String"/>
     </d-property>
     ...
   </d-class>
  </xsl:otherwise>
</xsl:choose>
...
</presentation>
```

description for the platform considered. The PM contains the screensize of the target device, but can be extended to contain other platform-specific characteristics. An example of a platform model is presented in Listing 8.4.

Listing 8.4: The XML platform description

```
<platform-model>
  <display-size>
   <width>416</width>
   <height>262</height>
  </display-size>
</platform-model>
```

Time-dimension Independence is one of the goals which should be reached by DSUIB-UIML's rule-based interpolation. Static rendering of user interfaces for a specific screen-size is straightforward. First, one adapts the platform model in order to specify the desired display-size. Next, an XSLT processor uses the generated stylesheet to transform the PM into a UIML user interface description. The resulting user interface description can serve as the input of a UIML renderer, which visualizes the user interface.

To reach dynamic interpolation, run-time adaptation is required. According to [CCT$^+$03], a run-time adaptation-process consists out of three steps: (1) recognition of the situation, (2) computation of a reaction and (3) execution of the reaction. In order to support these steps, the Uiml.net renderer is extended towards the *Uiml.net interpolation runtime*. A schematic overview of the Uiml.net interpolation runtime is given in Figure 8.16. To accomplish the first step, the *re-size-event* of the displayed user interface is triggered. Yet, this re-size-event is widget-set specific and thus implemented in the rendering backends [LTVC06]. Next, if one re-sizes the user interface, this new size is serialized into the platform model during the second step. Finally, the reaction is executed by an XSLT transformation of the platform model. The resulting UIML document will then be used to re-render the user interface.

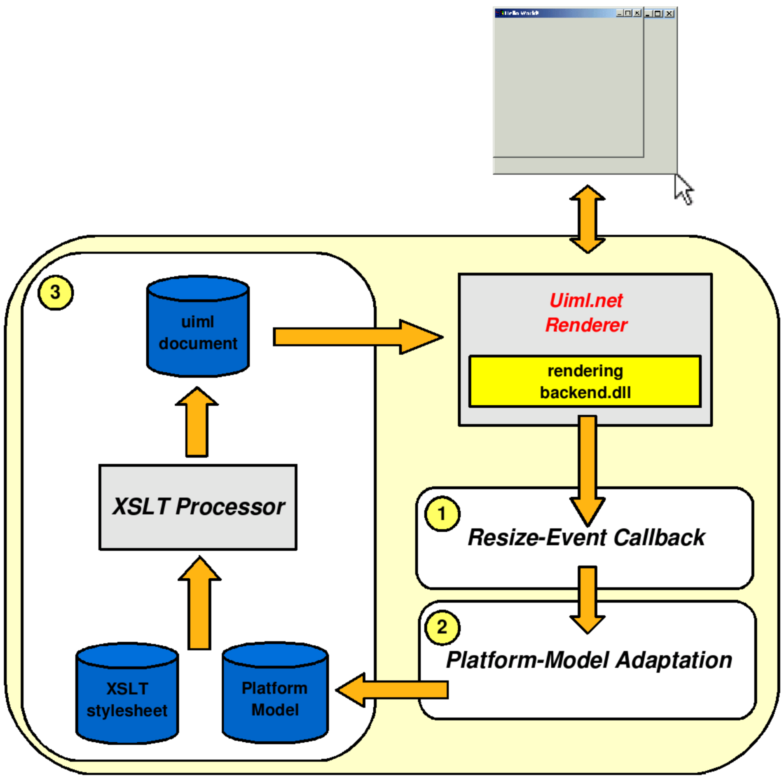


Figure 8.16: The Uiml.net interpolation runtime

8.6 Results

In this section, some examples are provided to illustrate the rule-based user interface interpolation mechanism. From the two reference user interfaces given in Figure 8.1, the interpolation at different screensizes is shown in Figure 8.17. The rule-set used here was constructed with few adaptations to the original one, generated by the heuristic. From the user interface in

Figure 8.17(a) towards the one in Figure 8.17(f), one can be see that the *select cards* component and the blue cards panel evoluate gradually towards their corresponding component in the other reference user interface.

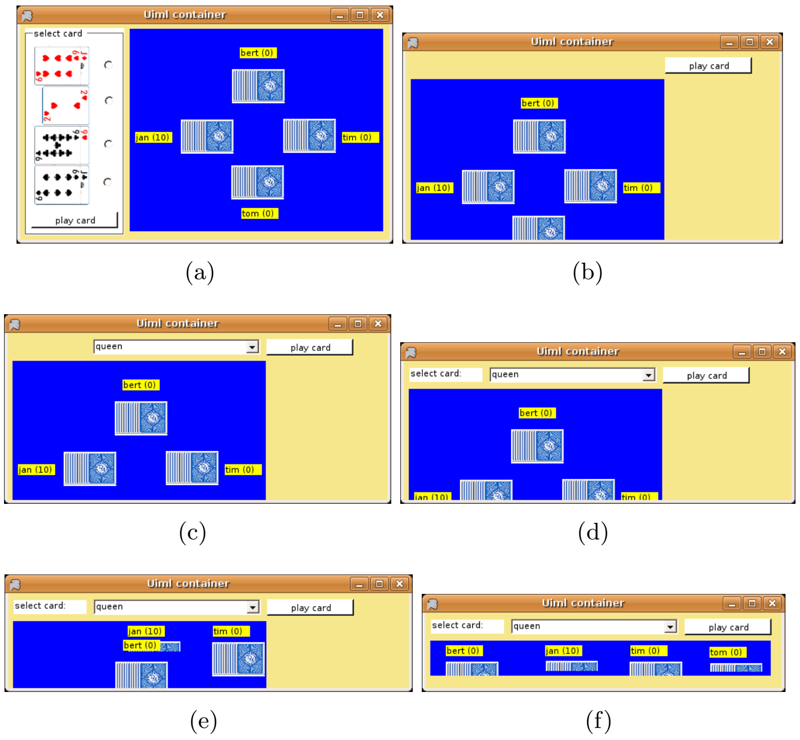
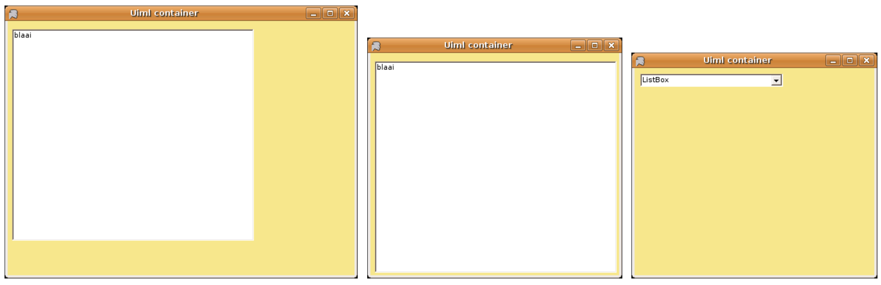


Figure 8.17: The interpolation of the card-game user interface at different screensizes

Figure 8.18 shows a remapping with the default heuristic as discussed in section 8.5.4. When the listbox is no longer completely visible, it is transformed into a combobox. In Figure 8.19 a birthdaybook interface is presented. The source interface, in Figure 8.19(a), evoluates gradually towards the smaller destination interface in Figure 8.19(f). A screenshot of DSUIB-UIML while designing this interface is provided in Figure 8.19(g).

8.7 Conclusion

In this chapter, an extension to the domain-specific user interface builder for UIML was presented: user interface interpolation. This technique can be used to support a broad range of adaptations and thus will be able to transform an interface depending on the available screensize.



(a) (b) (c)

Figure 8.18: The remapping between a combobox and a listbox with the default heuristic

A final evaluation of DSUIB-UIML in combination with the user interface interpolation mechanism is provided in Table 8.1. The score of the feedback and visibility principle stems directly from the the final evaluation of DSUIB-UIML (see Section 7.5). The combination of DSUIB-UIML and user interface interpolation results in a highly automated approach, however, the designer stays always in full control of the design. This control is accomplished by:

- a predictable user interface interpolation mechanism;
- DSUIB-UIML's intuitive user interface to specify the user interface interpolation behavior.

Principle	10	9	8	7	6	5	4	3	2	1	0
visibility	🟧										
feedback	🟧										
control	🟧										
automatization	🟧										

Table 8.1: The four general principles applied to DSUIB-UIML in combination with the user interface interpolation mechanism

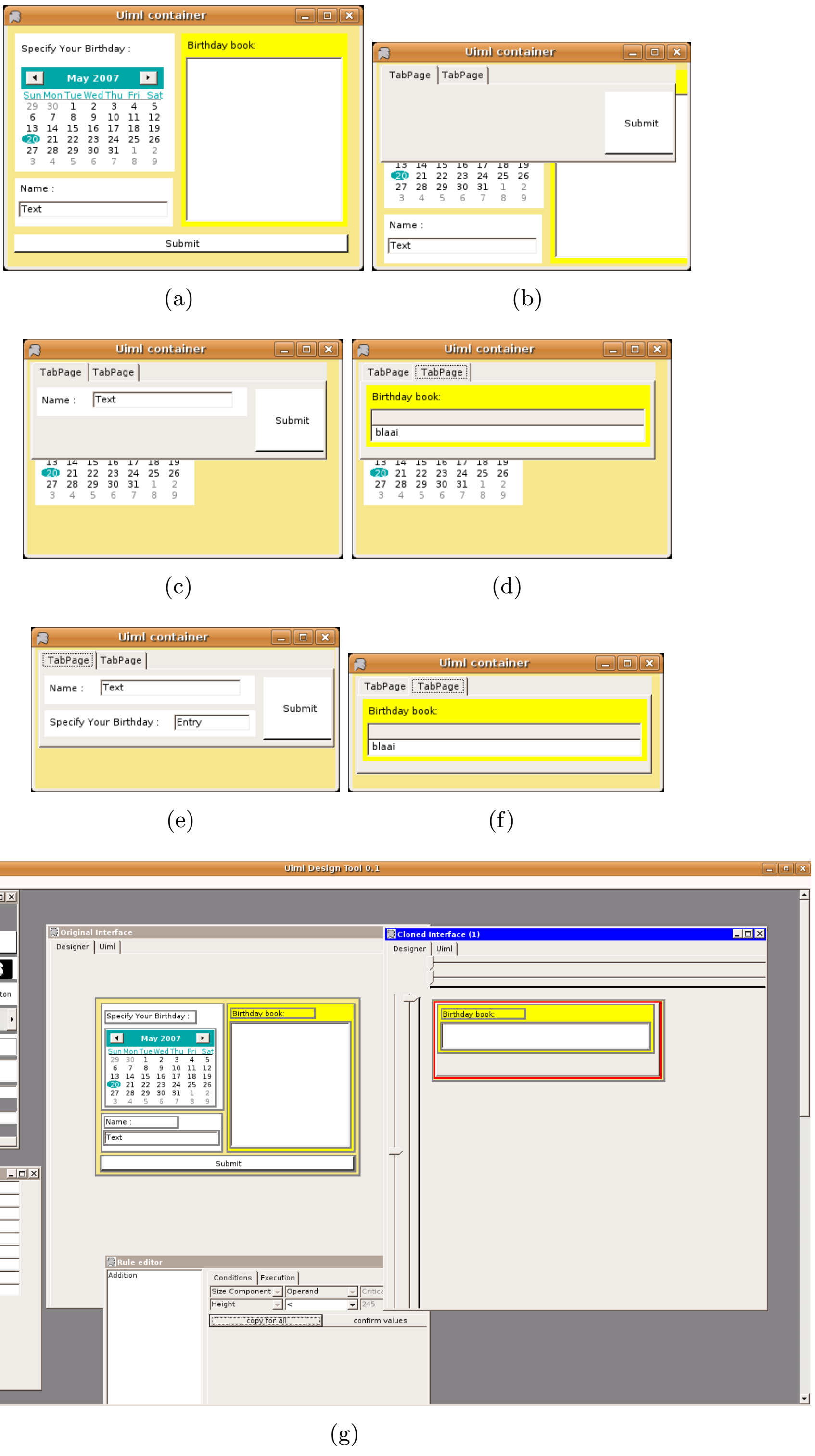


Figure 8.19: The birthdaybook application

Part IV

Conclusion

Chapter 9

Conclusion

9.1 Summary of the Results

In this thesis, a new type of user interface design tool was presented: the domain-specific user interface builder. This type of tool introduces the flexibility to generate a user interface builder for a certain domain, represented by a domain vocabulary. More specifically, a domain vocabulary contains a set of domain objects commonly used in the target domain as well as a user concrete interface component which represents this object visually.

A generated user interface builder contains three important dialogs: a toolbox, consisting of the domain objects' graphical representations; a canvas, to combine these domain objects towards a user interface; a property-panel, to manipulate the attributes of the domain objects. Compared with general-purpose user interface design tools, domain-specific interface builders speed up the construction of user interfaces for the domain considered.

A practical implementation of a domain-specific user interface builder was made on top of the Uiml.net renderer, called the *Domain-Specific User Interface Builder for the User Interface Markup Language* (DSUIB-UIML). To accomplish this, the UIML vocabulary serves as domain vocabulary. We showed that a traditional graphical user interface builder can be generated from the standard System.Windows.Forms vocabulary.

The UIML user interface description, generated by DSUIB-UIML, serves as input for the Uiml.net renderer. This involves that the user interface can be rendered on multiple devices, for example a mobile device and a desktop computer. Despite this multi-platform approach, a user interface designed for a desktop PC can fall partially outside the display-region on a small screen size.

A contribution to address this issue is made in this thesis with the implementation of a rule-based user interface interpolation mechanism, a new user interface based rendering technique. User interface based rendering is a theoretical model we deducted. It describes all the techniques which generate a user interface for a desired screen size from a set of previously designed ones. To accomplish this, user interface interpolation combines the components of two user interfaces, each designed for a specific user interface, towards a new user interface for an intermediate size. A user-defined rule-set is used to select the right component of both user interfaces at each size. This involves that a broad range of adaptations can be described at different sizes, such as resizing, repositioning, remapping, restructuring and adding a component.

The construction of the two required user interfaces and the user-defined rule-set is a non-trivial task for the user interface designer. Therefore, DSUIB-UIML was extended to minimize this effort. To accomplish this, the two major components are an integrated clone function and a rule-editor. The former clones the first user interface to avoid the creation of the second user interface from scratch. The latter supports the creation of rules and their conditions in an intuitive way.

DSUIB-UIML exports the constructed rule-set into an XSLT stylesheet. This stylesheet is able to transform a platform model, which contains the user interface's display size, into an UIML document. A UIML renderer can render this UIML document, which will result in the final user interface for the desired screensize. This process is integrated in the Uiml.net interpolation runtime, which is implemented as an extension of the Uiml.net renderer. Every time one re-sizes a user interface, the interpolation runtime will generate the appropriate user interface for that size.

During the research covered by this thesis, the most important lesson learned is that simplicity works. Initially, several *Artificial Intelligence* (AI) techniques, such as Markov models, decision trees and neural networks, were investigated to create the interpolation mechanism. Compared with these techniques, the rule-based mechanism proposed in this thesis is less complex. However, our approach is more predictable and controllable than most AI techniques.

9.2 Future Work

Despite the results presented in the previous section, there is still room for improvements. In this section, several issues of DSUIB-UIML are discussed.

9.2.1 Domain-Specific User Interface Builder

In the current UIML vocabulary, it is not possible to define one abstract interactor in terms of a set of other abstract interactors or to have differing `part` structures according to the selected mapping. In order to address this issue, a *1-to-N* mapping* [Luy05] mechanism is proposed. This new mapping mechanism maps one abstract interactor on a set of other abstract interactors that in turn should be mapped on the elements that are defined in the vocabulary. For example, an abstract interactor *playlist* can be mapped on several abstract interactors such as a *button*, *scrollbar*, *listbox* and *label*. Each of these abstract interactors will then be mapped on a concrete user interface object, as shown in Figure 9.1.

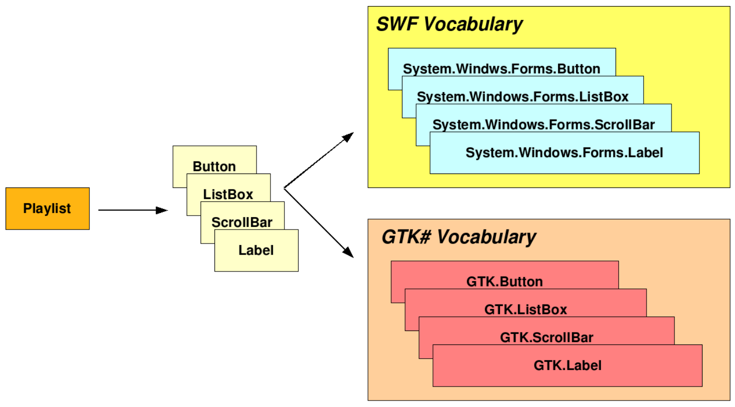

Playlist
Button
ListBox
ScrollBar
Label
SWF Vocabulary
System.Windws.Forms.Button
System.Windows.Forms.ListBox
System.Windows.Forms.ScrollBar
System.Windows.Forms.Label
GTK# Vocabulary
GTK.Button
GTK.ListBox
GTK.ScrollBar
GTK.Label

Figure 9.1: 1-N* mappings of an abstract playlist interactor

*1-to-N** mappings can be used to store patterns into the UIML vocabulary, which will represent more complex domain objects. For example, the playlist vocabulary proposed in section 7.2.2 can be constructed with *1-N** mappings. DSUIB-UIML will then be able to generate user interface builders for a wide range of complex domain vocabularies. In addition, it would be desirable to extend DSUIB-UIML with an intuitive mechanism which enables it to save patterns. If there is a part of the current designed user interface which can be reused in other designs, this mechanism will provide the possibility to select and save this reuseable part as a pattern in the vocabulary. Each pattern can be related to a certain name and iconic representation, which will appear in the toolbox. In this way, the designer can extend DSUIB-UIML *by practice* while he or she gains more knowledge in the problem domain.

9.2.2 Rule-Based User Interface Interpolation

Plasticity [CCT+02] is the capacity of an interactive system to withstand *variations of context of use* while preserving usability. The interpolation mechanism presented in this thesis does not guarantee anything about the usability of the intermediate user interfaces. Consequently, in some particular cases an interpolated user interface will be unusable. To resolve this, DSUIB-UIML can be extended to construct more reference user interfaces. We believe that an interpolation technique with more user interfaces will cover user interface space better, and thus will result in more usable intermediate user interfaces. This can be accomplished semi-automatically: using the current interpolation mechanism, a third user interface can be generated for a specific size from the two previously designed ones. This third user interface can then be adapted to resolve the usability problems for its particular size. This process can be repeated for an arbitrary number of reference user interfaces.

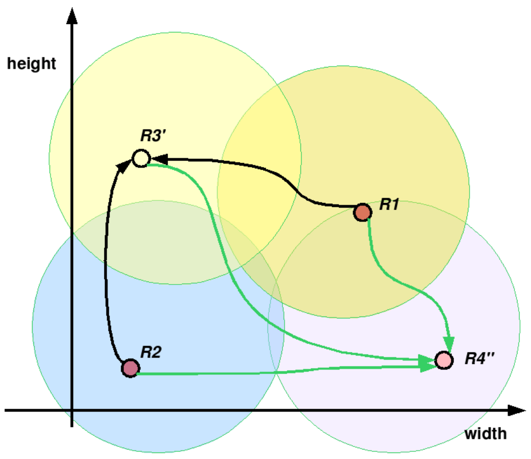


Figure 9.2: The extended interpolation mechanism

An example of this technique is given in Figure 9.2. $R1$ and $R2$ are the original reference user interfaces, which are used to generate $R3'$. After the adaptation of $R3'$, $R4''$ will be generated from the three already created user interfaces. For each user interface, a rule-set is created which is envisioned by the circular region around each user interface (see section 8.4).

Part V

Appendices

Appendix A

Dutch summary

A.1 Inleiding

Naast de traditionele desktopcomputers zijn er de laatste jaren verscheidene nieuwe computergestuurde toestellen ontwikkeld. Dit heeft bij de gebruikers de behoefte aangewakkerd om de applicaties die vooralsnog enkel op desktopcomputers aanwezig waren ook op deze nieuwe toestellen te kunnen draaien. Ten gevolge hiervan is het noodzakelijk om de gebruikersinterface van deze applicaties ook bereikbaar te maken op deze nieuwe toestellen.

Het ontwikkelen van een nieuwe gebruikersinterface voor elk van deze toestellen is een tijdrovende opdracht. Daarom wordt er onderzoek verricht naar technieken die het mogelijk maken om een gebruikersinterface te ontwikkelen voor meerdere platformen. Een veel gebruikte techniek bestaat uit het specifiëren van de gebruikersinterface op een hoog, toestelonafhankelijk niveau van abstractie. Vanuit deze specificatie kan dan via een geautomatiseerd proces een gepaste gebruikersinterface gegenereerd worden voor het gewenste toestel. Dit proces noemt men modelgebaseerde gebruikersinterface ontwikkeling.

Hoewel modelgebaseerde gebruikersinterface ontwikkeling platformonafhankelijkheid kan verzekeren, zijn de toepassingen die deze methodologie volgen niet echt populair. Een mogelijke verklaring hiervoor is dat de ontwikkelaars een te hoog niveau van abstractie moeten hanteren te vroeg in het ontwikkelingsproces. *GUI builders* daarentegen zijn dan weer populair, hoewel zij geen platformonafhankelijkheid garanderen. Een mogelijke verklaring hiervoor is dat bij *GUI builders* een gebruikersinterface op een laag niveau van abstractie moet gedefinieerd worden.

A.2 Domeinspecifieke Gebruikersinterface *Builders*

In deze thesis proberen we *GUI builders* zodanig uit te bereiden zodat zij ook platformonafhankelijke gebruikersinterfaces kunnen genereren, zonder deze te moeten specifiëren op een hoog abstractieniveau. Om dit te bewerkstelligen, onderzoeken wij de mogelijkheid om automatische *design tools* te genereren voor een bepaald domein, welk dat vooraf gedefinieerd is in een domein *vocabulary*. Deze *vocabulary* bevat een aantal vaak gebruikte abstracties in het gekozen domein alsook de gebruikersinterface elementen die deze abstracties voorstellen. Een domeinspecifieke *design tool* maakt het mogelijk om deze gebruikersinterface-elementen te combineren tot een volwaardige gebruikersinterface. Dit impliceert dus dat een gebruikersinterface ontwikkeld wordt op een concreet niveau van abstractie, maar toch wordt er gebruik gemaakt van domeinspecifieke abstracties. Deze aanpak resulteert finaal in een platformonfhankelijke gebruikersinterface.

Om een domeinspecifieke gebruikersinterface *builder* te implementeren baseren wij ons in deze thesis op de open Uiml.net renderer. Deze renderer kan gebruikersinterfaces genereren vanaf een declaratieve beschrijvende gebruikersinterfacetaal, genaamd de *User Interface Markup Language* (UIML). UIML is een metataal die gebruik maakt van generieke termen als `part`, `property` in plaats van specifieke elementen als `button`, `window`, etc. UIML bestaat enerzijds uit een `interface` gedeelte dat de structuur, stijl en het gedrag van gebruikersinterface bevat. Anderzijds is een `vocabulary` sectie aanwezig die de generieke termen *mapt* naar concrete gebruikersinterface-elementen. Dit houdt in dat op basis van UIML een nieuwe beschrijvende gebruikersinterfacetaal gemaakt kan worden door in het *vocabulary* gedeelte de juiste links tussen de generieke termen en concrete gebruikersinterface-componenten te leggen.

Om een domeinspecifieke `vocabulary` te beschrijven maken wij gebruik van de standaard UIML *vocabulary*. Deze wordt ingelezen en gebruikt om een serie domeinobjecten aan te maken. Vervolgens worden deze domeinobjecten één voor één gevisualiseerd (of liever *geconcretiseerd*) door de Uiml.net renderer. Een visuele representatie van een domeinobject noemen we een *component*. De nu bekomen verzameling van componenten wordt gebruikt om een *toolbox* aan te maken voor de gebruikersinterface *design tool*. Deze *toolbox* wordt dus dynamisch aangemaakt naargelang de gebruikte domein *vocabulary*. Om een gebruikersinterface te construeren kan de ontwikkelaar nu componenten selecteren in de *toolbox* en slepen naar een *canvas*. Op deze *canvas* wordt dus de eigenlijke structuur van de gebruikersinterface bepaald. De attributen van de op het *canvas* aanwezige componenten kunnen gewijzigd worden door deze te modifiëren in een attributenpaneel. Aangezien elke component correspondeert met een domeinobject, dat op zijn beurt een stukje UIML is, is het triviaal om vanuit een verzameling componenten een UIML gebruikersinterfacebeschrijving te genereren. Deze gegenereerde beschrijving kan vervolgens met behulp van de Uiml.net renderer, en de juiste *vocabulary*, op verscheidene platformen getoond worden. In deze thesis hebben we een gebruikersinterface *builder* gegenereerd voor de bestaande *System.Windows.Forms Vocabulary.*

A.3 Gebruikersinterface-interpolatie

Het ontwikkelen van een gebruikersinterface op een laag abstractieniveau impliceert dat er slechts één mogelijke situatie van de interface beschreven kan worden. Wanneer dezelfde interface getoond wordt op een toestel met een andere schermgrootte kan het gebeuren dat deze gebruikersinterface hier onbruikbaar wordt. Om dit te voorkomen moet een intelligente techniek voorzien worden die een gebruikersinterface kan aanpassen voor willekeurige - onbekende - schermgroottes.

In deze thesis wordt een gebruikersinterface-interpolatiemechanisme voorgesteld om nieuwe gebruikersinterfaces te genereren voor arbitraire schermgroottes. Dit mechanisme neemt als input twee gebruikersinterfaces, elk ontwikkeld voor een bepaalde schermgrootte. Het gebruikersinterface-interpolatie mechanisme gaat vervolgens de componenten van beide interfaces combineren om zo nieuwe interfaces te genereren voor (gedurende het ontwikkelingsproces) onbekende schermgroottes.

De moeilijkheid in het bewerkstelligen van een gebruikersinterface interpolatiemechanisme is de manier waarop de componenten dienen gekozen te worden bij de verschillende schermgroottes. Wij stellen een regelgebaseerd systeem voor, dat aan de hand van een vooraf opgestelde verzameling van regels de juiste componenten gaat kiezen voor een specifieke schermgrootte. Een regel kan bijvoorbeeld zijn: *'Voor de component I van de geïnterpoleerde interface: kies de grootte van* ***component A*** *uit interface 1 wanneer de schermgrootte ligt tussen 100 en 200, opteer in het andere geval voor de grootte van* ***component B****'*. Uit dit voorbeeld kunnen we de belangrijkste componenten van een regel deduceren:

- de **broncomponent:** component A in de eerste interface;
- de **doelcomponent:** component B in de tweede interface;
- de **condities:** wanneer de schermgrootte ligt tussen 100 en 200.

Regels kunnen opgesteld worden voor 4 verschillende eigenschappen:

1. het veranderen van de grootte van een component;

2. het veranderen van de plaats van een component in de gebruikersinterface;
3. het veranderen van de plaats van een component in de boomstructuur van een gebruikersinterface;
4. het transformeren van een component: bijvoorbeeld, een *dropdown* component die verandert in een *listbox* wanneer de schermgrootte verkleint.

Wanneer de voor een regel opgestelde condities gelden, wordt de geobserveerde eigenschap van de broncomponent genomen. Aan de andere kant, wanneer deze conditie niet geldt, wordt de eigenschap van de doelcomponent gekozen.

Het spreekt voor zich dat het opstellen van een verzameling regels met de bijhorende eigenschappen geen eenvoudige taak is voor een ontwikkelaar. Daarom is onze domeinspecifieke gebruikersinterface *builder* op een zodanige manier uitgebreid dat dit zo eenvoudig mogelijk kan verlopen. In de eerste plaats is er een eenvoudig systeem voorzien om de eerst ontwikkelde gebruikersinterface te klonen. Deze gekloonde gebruikersinterface kan vervolgens aangepast worden, wat zal resulteren in de tweede gebruikersinterface. Tijdens het klonen van de eerste interface worden de overeenkomstige componenten tussen beide gebruikersinterfaces gelinkt: de componenten in gebruikersinterface 1 worden de broncomponenten, diegenen in de andere gebruikersinterface zijn de doelcomponenten. Dit volstaat echter niet. Wanneer er bijvoorbeeld bij een van de twee gebruikersinterfaces een nieuwe component wordt toegevoegd, moet het mogelijk zijn om deze te linken aan een component in de andere gebruikersinterface. Dit wordt in de domeinspecifieke gebruikersinterface *builder* ondersteund door een *reference it* operatie in het contextmenu van een component.

Tot hiertoe hebben we twee gebruikersinterfaces, elk bestaande uit een set van componenten met telkens een link tussen een component in gebruikersinterface 1 naar een component in gebruikersinterface 2. Nu moet op basis van deze verzameling links een setje van regels opgebouwd worden. Dit wordt bewerkstelligd door een continu proces, dat tijdens het opmaken van de interface voortdurend de bron- en doelcomponenten in beide interfaces vergelijkt. Wanneer voor één van de hierboven vermelde eigenschappen een verschil tussen de bron- en doelcomponent wordt gevonden, wordt er voor deze eigenschappen een regel aangemaakt. Zo is het dus mogelijk dat er voor elk van de vier bovenvermelde eigenschappen een regel wordt aangemaakt. Door een impliciete regelcreatie te verkiezen boven een expliciete wordt de designer niet gestoord in zijn hoofdactiviteit, zijnde het creëren van de gebruikersinterface.

Eens de regels gecreëerd zijn moeten er nog de gepaste condities aan toegekend worden. De domeinspecifieke gebruikersinterface *builder* bevat een basisheuristiek die initiële intervallen gaat berekenen. Voor elke bron- en doelcomponent wordt de minimale schermgrootte berekend die nodig is om deze component te kunnen weergeven. Vervolgens wordt paarsgewijs de component C^i_{max} gekozen die de grootste schermgrootte $Size^i$ vereist. Voor elke regel kunnen dan de volgende initiële condities worden opgesteld: indien de schermgrootte groter is dan $Size^i$, opteer dan voor C^i_{max}, anders dient de aan C^i_{max} gelinkte component gekozen te worden.

Hoewel de hierboven beschreven heuristiek werkt voor eenvoudige gebruikersinterfaces, voldoet deze niet in meer complexe situaties. Daarom bevat de domeinspecifieke gebruikersinterface *builder* een eenvoudige regel *editor*. Deze laat toe om op een intuïtieve manier de intervallen voor de impliciet gecreëerde regels te bewerken. Wanneer in de regel *editor* een regel geselecteerd wordt, verschijnen er schuifbalken naast de ontworpen gebruikersinterfaces. Deze schuifbalken stellen de discrete intervallen voor waarbinnen de geselecteerde regel geldt. Door de schuifbalken te manipuleren, kunnen de condities eenvoudig aangepast worden zonder dat de ontwikkelaar manueel bepaalde intervallen moet berekenen.

De aangemaakte regels dienen geëxporteerd te worden naar UIML opdat er een adaptatie van de gebruikersinterface zou kunnen plaatsvinden. Wij kozen ervoor om de regels te exporteren in een XSLT *stylesheet*. Deze *stylesheet* wordt gebruikt om een platformmodel, dewelke de aanwezige schermgrootte bevat, om te vormen tot een UIML document. Door het platformmodel aan te passen telkens wanneer de schermgrootte wijzigt en vervolgens de XSLT transformatie opnieuw uit te voeren wordt steeds een geschikte gebruikersinterface bekomen voor de beschikbare schermgrootte. Dit proces werd ondergebracht in de Uiml.net renderer, dewelke nu kan omgedoopt worden tot de *Uiml.net interpolation runtime*.

A.4 Toekomstig Werk

De concepten voorgesteld in deze thesis zijn nog voor verbetering vatbaar. Zo is het in de huidige versie van UIML niet mogelijk om samenstellingen van abstracte interactoren te specifiëren in de *vocabulary*. Eens dit mogelijk gemaakt is zou een ontwikkelaar dynamisch herbruikbare gebruikersinterface elementen kunnen toevoegen aan de *vocabulary*. Ook zou een iconografische specificatie per gebruikersinterface element een handige meerwaarde bieden.

Het opgestelde interpolatiesysteem kan in sommige gevallen ongebruiksvriendelijke gebruikersinterfaces genereren. Om dit te voorkomen kan in de toekomst geopteerd worden voor een interpolatiesyteem dat meer dan twee interfaces als input neemt. Van de eerste twee opgestelde gebruikersinterfaces kan een derde gegenereerd worden voor een gewenste schermgrootte. Deze derde kan dan aangepast worden om de gebruiksvriendelijkheid ervan te verhogen. Van de drie aanwezige interfaces kan vervolgens een vierde gegenereerd worden, enz. Hoe meer gebruikersinterfaces als input, hoe hoger de gebruiksvriendelijkheid van de tussenliggende gebruikersinterfaces zal liggen.

Bibliography

[AH04] Marc Abrams and James Helms. Uiml 3.1 language specification. Technical report, Oasis UIML TC, 2004.

[AnSA02] Mir Farooq Ali, Manuel A. Pérez-Qui nonez, Eric Shell, and Marc Abrams. Building multi-platform user interfaces with uiml. In *Proceedings of CADUI 2002*, May 2002.

[Ant01] Antipope.org. Rapid application development tools, 2001. [Online; accessed 15-March-2007].

[AP99] Marc Abrams and Constantinos Phanouriou. Uiml: An xml language for building device-independent user interfaces. In *XML '99*, Philadelphia, USA, 1999.

[BVE02] Laurent Bouillon, Jean Vanderdonckt, and Jacob Eisenstein. Model-based approaches to reengineering web pages. In *TAMODIA '02: Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design*, pages 86–95. INFOREC Publishing House Bucharest, 2002.

[CCT$^+$02] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Nathalie Souchon, Laurent Bouillon, Murielle Florins, and Jean Vanderdonckt. Plasticity of user interfaces: A revised reference framework. In *TAMODIA*, pages 127–134, 2002.

[CCT$^+$03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[CLC04] Tim Clerckx, Kris Luyten, and Karin Coninx. The mapping problem back and forth: customizing dynamic models while preserving consistency. In *TAMODIA*, pages 33–42, 2004.

[EJ01] R. Esser and J. Janneck. A framework for defining domain-specific visual languages. In *Workshop on Domain Specific Visual Languages, in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-2001)*, Portland, USA, October 14-18 2001.

[FMVM06] M. Florins, F. Montero, J. Vanderdonckt, and B. Michotte. Splitting rules for graceful degradation of user interfaces. In *Proceedings of 8th Int. Working Conference on Advanced Visual Interfaces AVI'2006*, pages 59–66, New York, May 23-26 2006.

[Fra95] M. Frank. *Model-Based User Interface Design by Demonstration and by Interview.* PhD thesis, Georgia Institute of Technology, 1995.

[FSF95] Martin R. Frank, Piyawadee Noi Sukaviriya, and James D. Foley. Inference bear: Designing interactive interfaces through before and after snapshots. In *Symposium on Designing Interactive Systems*, pages 167–175, 1995.

[GCH+05] Krzyszstof Gajos, David Christianson, Raphael Hoffmann, Tal Shaked, Kiera Henning, Jing Jing Long, and Daniel S. Weld. Fast and robust interface generation for ubiquitous applications. In *Proceedings of Ubicomp 2005*, Tokyo, Japan, 2005.

[GHZL06] John C. Grundy, John G. Hosking, Nianping Zhu, and Na Liu. Generating domain-specific visual language editors from high-level tool specifications. In *ASE*, pages 25–36, 2006.

[GMH00] John C. Grundy, Warwick B. Mugridge, and John G. Hosking. Constructing component-based software engineering environments: issues and experiences. *Information and Software Technology*, 42(2):103–114, 2000.

[GtMP+98] T. Gri, t McKirdy, N. Paton, J. Kennedy, R. Cooper, B. Barclay, C. Goble, P. Gray, M. Smyth, A. West, and A. Dinn. An open model-based interface development system: The teallach approach. In *Proceedings of DSV-IS'98*, pages 32–49, June 1998.

[GW04] Krzyszstof Gajos and Daniel S. Weld. Supple: Automatically generating user interfaces. In *Proceedings of IUI'04*, January 13-16 2004.

[GW05] Krzyszstof Gajos and Daniel S. Weld. Preference elicitation for interface optimization. In *Proceedings of UIST 2005*, Seattle, USA, 2005.

[HY91] Scott E. Hudson and Andrey K. Yeatts. Smoothly integrating rule-based techniques into a direct manipulation interface builder. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 145–153, November 1991.

[Jam03] Anthony Jameson. Adaptive interfaces and agents. pages 305–330, 2003.

[JB03] A. John and G. Biao. An environment for developing adaptive, multidevice user interfaces. In *Proceedings of the Fourth Australasian User Interface Conference AUIC'2003, Vol 18*, pages 59–66, Adelaide, Australia, 2003.

[Jon05] Edwin Jongsma. Domain specific language, software factories ontraadseld. *Microsoft .NET Magazine*, (11):33–35, december 2005.

[LC04] Kris Luyten and Karin Coninx. Uiml.net: An open uiml renderer for the .net framework. In *Proceedings of Computer-Aided Design of User Interfaces (CADUI'2004)*, pages 221–230, Funchal, Madeira Island, PT, 2004.

[LF01] S. Lok and S. Freiner. A survey of automated layout techniques for information presentations. In *Proceedings of Smart Graphics 2001 (Int. Symp. on Smart Graphics)*, Hawthorne, NY, march 2001.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42, New York, NY, USA, 1996. ACM Press.

[LL02] James Lin and James A. Landay. Damask: A tool for early-stage design and prototyping of multi-device user interfaces. In *Proceedings of The 8th International Conference on Distributed Multimedia Systems (2002 International Workshop on Visual Computing)*, pages 573–580, San Francisco, CA, September 26-28 2002.

[LL05] Yang Li and James A. Landay. Informal prototyping of continuous graphical interactions by demonstration. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 221–230, New York, NY, USA, 2005. ACM Press.

[LTVC06] Kris Luyten, Kristof Thys, Jo Vermeulen, and Karin Coninx. A generic approach for multi-device user interface rendering with uiml. In *Proceedings of Computer-Aided Design of User Interfaces (CADUI'2006)*, Bucharest, Romania, June 5-8 2006.

[Luy04] Kris Luyten. *Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development.* PhD thesis, Expertise Centre for Digital Media, LUC, october 2004.

[Luy05] Kris Luyten. Proposition: Adaptable 1-n* mappings for uiml vocabularies. `http://research.edm.uhasselt.be/kris/research/uiml.net/adaptable1onNmappings.pdf`, 2005.

[LVC06] Kris Luyten, Jo Vermeulen, and Karin Coninx. Constraint adaptability of multi-device user interfaces. In *Workshop on The Many Faces on Consistency, CHI'2006 workshop*, Montreal, Quebec, Canada, April 22-23 2006.

[McM97] Leonard McMillan. *An Image-Based Approach to Three-Dimensional Computer Graphics.* PhD thesis, University of North Carolina, Apr 1997.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*, 7(1):3–28, 2000.

[MPS04] G. Mori, F. Paternó, and Carmen Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. In *Proceedings IEEE transactions on software engineering*, August 2004.

[Pat00] Fabio Paternò. Model-based design and evaluation of interactive applications. In *Springer*, 2000.

[Pay98] Bernd Paysan. Mino$\sum$-system integration. `http://www.jwdt.com/~paysan/minos-eng-2.ps.gz`, August 1998.

[PE99] Angel R. Puerta and Jacob Eisenstein. Towards a general computational framework for model-based interface development systems. In *Intelligent User Interfaces*, pages 171–178, 1999.

[PEGM94] Angel R. Puerta, Henrik Eriksson, John H. Gennari, and Mark A. Musen. Model-based automated generation of user interfaces. In *Proceedings of National Conference on Artificial Intelligence*, pages 471–477, 1994.

[PMS03] F. Paternó, G. Mori, and C. Santoro. Tool support for designing nomadic applications. In *Proceedings of 7 Int. Conf. on Intelligent User Interfaces IUI'03*, pages 12–15, January 2003.

[Pue97] Angel R. Puerta. A model-based interface development environment. *IEEE Softw.*, 14(4):40–47, 1997.

[SCPR06] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Philips, and Nicolas Roussel. User interface façades: Towards fully adaptable user interfaces. In *Proceedings of UIST 2006*, Montreux, Switzerland, October 15-18 2006.

[SS94] Andrew Sears and Ben Shneiderman. Split menus: effectively using selection frequency to organize menus. *ACM Trans. Comput.-Hum. Interact.*, 1(1):27–51, 1994.

[SV03] Nathalie Souchon and Jean Vanderdonckt. A review of xml-compliant user interface description languages. In *DSV-IS*, pages 377–391, 2003.

[TC99] D. Thévenin and J. Coutaz. Adaptation and plasticity of user interfaces. In *Workshop on Adaptive Design of Interactive Multimedia Presentations for Mobile Users*, 1999.

[VdL02] Hans Vangheluwe and Juan de Lara. Meta-models are models too. In *Winter Simulation Conference*, pages 597–605, 2002.

[WF91] D. Wolber and G. Fisher. A demonstrational technique for developing interfaces with dynamically created objects. In *Proc. of the 4th Annual Symposium on User Interface Software and Technology (UIST'91)*, pages 221–230, Hilton Head, SC, 1991.

[Wik07] Wikipedia. Xforms, 2007. [Online; accessed 20-March-2007].