Chapter 1

A GENERIC APPROACH FOR MULTI-DEVICE USER INTERFACE RENDERING WITH UIML

Kris Luyten Kristof Thys Jo Vermeulen Karin Coninx
Hasselt University – transnationale Universiteit Limburg
Expertise Centre for Digital Media – IBBT
Wetenschapspark, 2
B3590 Diepenbeek (Belgium)
{Kris.Luyten,Kristof.Thys,Jo.Vermeulen,Karin.Coninx}@uhasselt.be

Abstract We present a rendering engine for displaying graphical user interfaces on multiple devices. The renderer interprets a standardized XML-based user interface description language: the User Interface Markup Language (UIML). A generic architecture for the renderer is defined so that deployment of the engine on different devices implies only little effort. We show that our rendering engine can be used on iDTV set-top boxes, mobile phones, PDAs and desktop PCs, and smoothly integrates with both local and remote application logic. As a test bed for the UIML specification we also explore support for extensions to UIML that enable the user interface designer to maximize accessibility and target multiple devices and different types of users at once.

1. Introduction and Motivation

There is a real need for user interface (UI) design tools to support multi-device UI creation, even though there are many different existing solutions for this purpose. Most solutions are limited however since they specifically target a predefined range of platforms, e.g. a browser environment like Mozilla XUL[1], specific programming languages (e.g. bound to Java like Jaxx[2] or jXUL[3]) or a limited set of interaction modalities (mostly graphical UIs). In contrast, UIML is a *metalanguage* that can

[1] http://developer.mozilla.org/en/docs/MDC:About
[2] http://www.jaxxframework.org
[3] http://jxul.sourceforge.net/

be used to specify the four different aspects of a UI (structure, style, content and behavior) independently [2]. None of these parts contain information that is related to a specific platform, programming language or modality. Although the language is syntactically generic, it allows some semantic constructs that imply a dependency on a certain platform, programming language or modality. In this paper we show how these limitations can be overcome and we discuss the foundations of the UIML rendering engine we provide to the community.

To our knowledge, only little has been published about the actual architecture of similar rendering engines. Roughly, there are two approaches for XML-based UI specification languages: the language can target a generic but limited set of widgets or the language is very close to the final widget set and thus specific for a certain type of platform. The former is often used for multi-device UI description languages (e.g. [3] and [7]), while the latter often relies on a certain rendering platform such as a browser. Most implementations that are publicly or commercially available are also limited to use with one particular programming language (or, in many cases, one virtual machine) and widget set.

Section 2 provides an introduction to UIML and explains the basic structure and constructs of this XML-based language. Section 3 gives a short introduction and overview of the UIML rendering engine architecture. Next, sections 4 and 5 discuss the extensions we created to enhance support for multiple devices and multiple users. Finally, section 6 provides the conclusions.

2. The User Interface Markup Language (UIML)

UIML [1] is a high-level markup language to describe the structure, style, content and behavior of a UI. For a thorough discussion of the UIML language we refer to the specification [1]: we will limit the UIML overview in this paper to the essential constructs. UIML offers a clear separation of concerns which is reflected in the four different aspects of a UI that can be specified in this metalanguage. Figure 1.1 shows a skeleton of a UIML document at the top of the figure.

In the past there were attempts to create a generic vocabulary by Abrams et. al [3]. In our approach we try to support a generic set of interactors that can be used across vocabularies, but also allow to use vocabulary specific interactors. If the designer wants to differentiate the interface for a platform because of the availability of certain widgets, she or he can do so with our implementation. The multi-vocabulary approach that our implementation supports is described in [4].

3. UIML-renderer Architecture

Multi-device Architecture

A prerequisite for the renderer to work is the availability of a .Net or Java Virtual Machine on the target device. Most mobile and even embedded devices support one of these two virtual machines. Our implementation is built in such a way that there are no further dependencies on exactly which version or type of virtual machine, it simply relies on the virtual machine as a dynamic execution environment that allows to load new functionality on demand while running the UI. This enables us to support multiple mapping vocabularies on-the-fly.

Our UIML renderer consists of one rendering core and multiple rendering backends that contain 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. Notice that the mappings from abstract interactors to concrete widgets are defined in the peers section (the vocabulary) of a UIML document. Since the mapping information is provided from outside of the renderer, it can be loaded dynamically and applied at runtime to the rendering core. The rendering backends have a very limited responsibility: they process the parts of a UIML document that can rely on widget-set specific knowledge. Notice that due to the structure of a UIML vocabulary, most mappings can be executed by the rendering core. The reflection mechanism allows to use the information from the vocabulary to detect and load the required widgets at runtime without any platform-specific code. This approach overcomes the limitations of a generic vocabulary by allowing the designer to also use widget-set specific parts.

To overcome the limitations of devices that have limited capabilities and no means to use a reflection mechanism (for example the MIDP2.0[4] platform does not include a reflection API), the rendering core needs to include ad-hoc mappings. This approach limits the extensibility of the UIML vocabulary: if a new mapping is included in the vocabulary, the rendering core should be updated while this is not necessary when reflection can be used.

Multi-device Rendering Behavior

We define an *internal representation* of a UIML document as a set of objects that represent an in-memory tree of the UIML document which can be rendered at any time without further transformations.

[4] `http://java.sun.com/products/midp/`

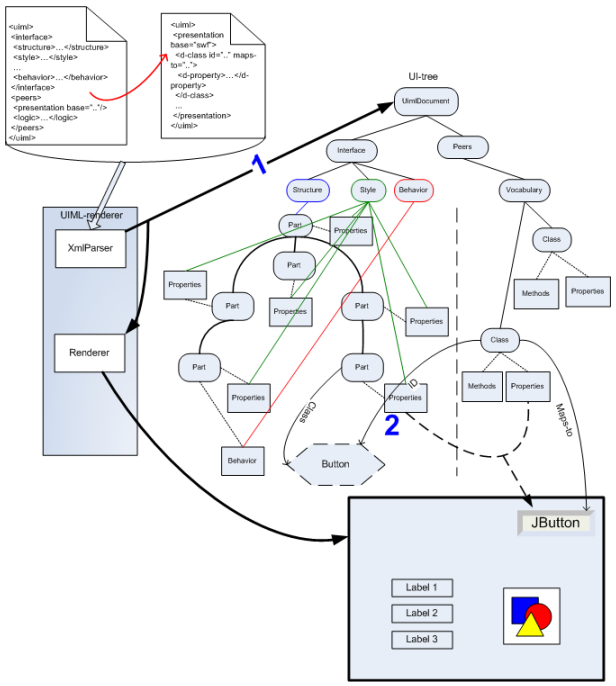

Figure 1.1. An overview of the UIML rendering engine

This reflects step 1 in figure 1.1. We define a *concrete instantiation* of a UIML document as a set of objects that represent an in-memory tree of the UIML document where each object that represents a UIML part has a reference to a final widget on the target platform. This reflects step 2 in figure 1.1.

Figure 1.1 shows the overall behavior of a UIML-renderer. The renderer uses several stages, where each stage processes a certain aspect of the UIML document. We identified three different stages of processing that are required for a flexible rendering engine:

pre-processing : during this stage a UIML document can be transformed into another UIML document. Section 4 gives some examples of transformations that require a pre-processing stage.

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 UI will be selected.

The main processing stage is more specifically composed out of the following steps, illustrated in figure 1.1;

1 The UIML-renderer takes an 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.

This section focused on the main processing stage of the rendering engine. The next sections will discuss the pre-processing and post-processing stages. Both are important not only to support multiple devices, but also to be able to transform the UIML document according to some parameters before rendering and to support a smooth integration with the target environment.

4. The Pre-processing Stage

In the pre-processing stage the renderer will apply transformations on the UIML document before creating a concrete instantiation of the user interface. The transformations will be applied on this input document: this results in an intermediate document version that can be pre-processed by other predefined transformations.

In this section we discuss two different pre-processing stages that are supported by the UIML.net rendering engine: applying user profiles for more accessible UIs and applying layout constraints to maximize UIML document portability across platforms.

The support for more accessible user interfaces is accomplished by using an MPEG-21 description profile of the usage environment (a separate part of the MPEG-21 part 7 specification). The usage environment can

describe the accessibility characteristics that the UI needs to take into account for a particular user. A user agent reads the characteristics from the MPEG-21 file and transforms the values found in the properties of the internal representation of the UIML document so that they convey to the required characteristics. For example; a MPEG-21 profile can define the colour vision deficiency for the user and provide required values for foreground and background colours. The user agent will transform the matching properties that are found in the internal representation into values that make sure the colours that cause the colour vision deficiency are not used. A full description of this approach can be found in [5].

For the layout management preprocessor to act as a pre-processing stage like described in this section, we consider the specification of the layout constraints to be a part of the UIML specification [1]. We allow the designer to use spatial layout constraints to specify the layout of the UI. The layout management preprocessor will generate a platform specific layout using absolute positioning that is consistent with the predefined set of constraints. The usage of spatial layout constraints makes a UIML document even more portable across platforms and adds a certain degree of plasticity to it. In the past, the layout specification in a UIML document was embedded in the structure and style descriptions and relied on platform specific constructs (e.g. the availability of the Java GridBagLayout class). A full description of this approach can be found in [6].

5. The Post-processing Stage

Once a concrete instantiation is created by the rendering engine, it can still be further manipulated. The primary function of the post-processing stage is to bind the UI with the application logic.

Our renderer allows the application logic to subscribe to events from the UIML user interface. This is realized through reflection and aspect-oriented programming techniques. Arbitrary object instances can be connected to the UIML document after the main processing stage, and it is possible to tag their methods with information declaring interest in specific events from certain parts of the user interface. Upon connection, the renderer inspects the connected object to check whether any of its methods are interested in user interface events, and if so, a hook between that particular event and the object's method will be dynamically created, similar to a callback function. Furthermore, a method can state that it wants to receive the internal representation of a part element, when the event gets fired. The properties of this part element can be queried or manipulated, and the application logic can even act upon

the concrete widget instantiation. This mechanism allows for interesting applications. For instance; suppose we have a UIML login form, which connects to a database. When the authentication mechanism changes, the developer can just replace the existing authentication library. Due to the dynamic nature of this binding, nor the user interface, nor the renderer has to be changed.

As an extension module we added support for adaptive navigation through the UI. In implementing UIML renderers on resource-constrained devices for different types of users, navigation through the UI should be carefully defined. On devices with limited navigation input possibilities, e.g. arrow-buttons only as found on the remote control of a television, UI designers want to specify how users can navigate between different interface elements. The post-processing stage to support adaptable navigation uses a separate navigation specification that can be applied on the final UI. Since the internal representation is still available during the post-processing stage, this information can still be used. For example, the hierarchy of parts from the UIML document indicates which concrete widget is a container for other ones.

The navigation specification defines how the focus is handled, using *init* and *focus-transition* rules similar to the working of a state transition network. The *init* rule specifies the element that gets the focus when the parent container is entered. A *focus-transition* specifies the element that gets the focus when a navigation event is generated (e.g. left button pressed). The UIML-renderer applies these rules by capturing the events generated by the UI and checks whether the event is a navigation event and can be processed. One of the benefits of this approach is that navigation through the generated UI can be optimized for on the screen sizes or user preferences.

6. Conclusions

Our demonstrator shows a functional UIML renderer on multiple devices that is capable of generating personalized and platform-specific user interfaces from a high-level user interface description. The renderer will be demonstrated on a mobile phone, PDA, desktop PC and for a set-top box. We provide a UIML renderer that processes the UIML document on the client device. This approach results in greater flexibility and better support for just-in-time render- and runtime adaptations of the UI.

The .Net version of the UIML renderer presented in this paper is available as free software at `http://research.edm.uhasselt.be/uiml`. As far as we know it is the only free (as in freedom) implementation of

the UIML 3.0 standard that is available. It has been used and tested in several projects, and we want to encourage everyone to use the rendering software and contribute in any possible way. Although the renderer runs on Java virtual machines as well as .Net virtual machines, only the latter is publicly available.

Acknowledgments. Part of the research at EDM is funded by the European Fund for Regional Development, the Interdisciplinary Institute for Broadband Technology (IBBT) and the Flemish Government.

References

[1] Marc Abrams and James Helms. User Interface Markup Language (UIML) Specification version 3.1. Technical report, Oasis UIML TC, 2004.

[2] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An Appliance-Independent XML User Interface Language. *WWW8 / Computer Networks*, 1999.

[3] Mir Farooq Ali, Manuel A. Pérez-Quiõnes, Marc Abrams, and Eric Shell. Building Multi-Platform User Interfaces with UIML. In *Computer-Aided Design of User Interfaces*, pages 255–266, 2002.

[4] Kris Luyten and Karin Coninx. Uiml.net: an Open Uiml Renderer for the .Net Framework. In *Computer-Aided Design of User Interfaces*, 2004.

[5] Kris Luyten, Kristof Thys, and Karin Coninx. Profile-Aware Multi-Device Interfaces: An MPEG-21-Based Approach for Accessible User Interfaces. In *Accessible Design in the Digital World*, 2005.

[6] Kris Luyten, Jo Vermeulen, and Karin Coninx. Constraint Adaptability for Multi-Device User Interfaces. In *Workshop on The Many Faces on Consistency*, 2006.

[7] Roland A. Merrick, Brian Wood, and William Krebs. Abstract User Interface Markup Language. In *Developing User Interfaces with XML: Advances on User Interface Description Languages*, pages 39–46, 2004.