Meta-GUI-Builders: Generating Domain-specific Interface Builders for Multi-Device User Interface Creation

Kris Luyten
Hasselt University
transnationale Universiteit Limburg
Expertise Centre for Digital Media
Diepenbeek, Belgium
kris.luyten@uhasselt.be

Karin Coninx
Hasselt University
transnationale Universiteit Limburg
Expertise Centre for Digital Media
Diepenbeek, Belgium
karin.coninx@uhasselt.be

Jan Meskens
Hasselt University
transnationale Universiteit Limburg
Expertise Centre for Digital Media
Diepenbeek, Belgium
jan.meskens@uhasselt.be

Jo Vermeulen
Hasselt University
transnationale Universiteit Limburg
Expertise Centre for Digital Media
Diepenbeek, Belgium
jo.vermeulen@uhasselt.be

CHI 2008, April 5 – April 10, 2008, Florence, Italy
ACM 978-1-60558-012-8/08/04.

ABSTRACT
Nowadays, there is a growing demand to design user interfaces that run on many devices. However, existing multi-device design approaches are not suitable for domain experts, whose input can be invaluable to come to a suitable user interface for a specific domain. Existing techniques often require the manipulation of high-level models and transformations which are difficult to interpret and predict by a domain expert without a technical background. We present Meta-GUI-Builders, a new generation of graphical user interface builder tools that allows domain experts to create multi-device GUI designs themselves. These tools automatically adapt their workspace to a specific domain by encapsulating domain-specific elements in the designer's tool palette. Engaging domain experts in a multi-device design approach is a first step towards creating aesthetic user interfaces that can be deployed on many devices, a combination that is hard to achieve with previous approaches.

ACM Classification Keywords
H5.2. **Information interfaces and presentation**: User Interfaces – Graphical user interfaces, Prototyping

```
<uiml>
  <interface>
    <structure>
      <part id="player" class="Container">
        <part id="current"
              class="Container">
        <part id="song_id" class="Song"/>
        <part id="controls" class="Timer"/>
        <part id="controls"
              class="PlayControls"/>
      </part>
      <part id="list" class="PlayList"/>
    </structure>
    <style>...</style>
  </interface>
  <peers>
    <presentation
          base="http://purl.org/uimlvocs
                /audio-simple.uiml"/>
  </peers>
</uiml>
```

Table 1: An example UIML document describing an audio player

Keywords
UIML, multi-device interface design, prototyping

INTRODUCTION

The input of a domain expert can be invaluable to create an appropriate user interface for a specific domain. Most of the current graphical user interface (GUI) creation tools are geared toward a software developer and are not suitable for a domain expert that has no technical background. GUI builders such as the form editor included in Microsoft Visual Studio (http://msdn.microsoft.com/vstudio/) and the Eclipse Visual Editor (http://www.eclipse.org/vep) focus on integration with the underlying software instead of translating domain concepts into a suitable user interface presentation. Imagine building an advanced piece of audio processing software that should be deployed on multiple platforms: this would require a domain expert to define the domain concepts that should be reflected in the user interface presentation. Currently, the software developer has to map these domain concepts manually onto an appropriate presentation in the final user interface. Moreover, this mapping process has to be repeated for every target platform on which the user interface will be deployed.

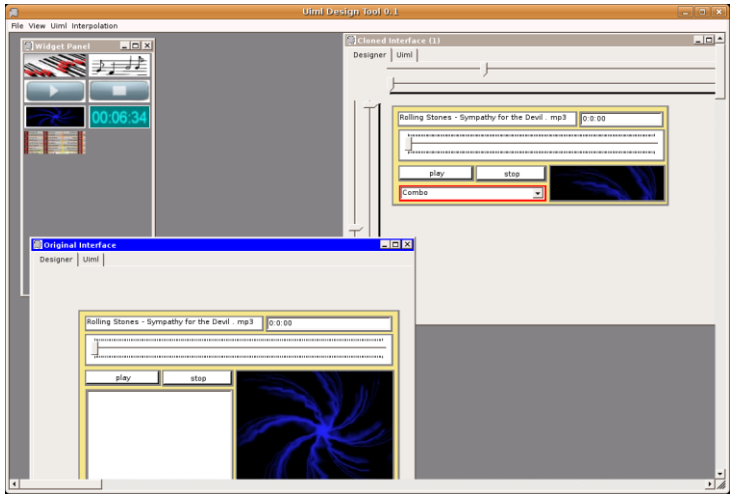
In this paper we describe an approach which allows for focusing on designing a user interface from the viewpoint of the domain concepts instead of the software structure. Our method is independent of the computing platform (amongst others the programming language and end-user device) and smoothly integrates the user interface with the functional core, even if these are created by different experts as is often the case.

Underlying this idea is a new type of user interface authoring tool: a *meta user interface builder tool* that adapts its workspace according to a specific domain. A visual builder tool is generated for the user interface designer according to a domain vocabulary that defines domain concepts for a certain problem domain. The core language that is used to enable automatic authoring tool generation is itself a meta-language: the User Interface Markup Language (UIML) [7].

THE USER INTERFACE MARKUP LANGUAGE

UIML is a canonical XML-based user interface description language that supports a custom naming scheme according to the problem domain. A UIML description expresses the structure, style, content and behavior of a user interface independent of platform, widget set and programming language. For this purpose a *mapping vocabulary* containing mapping rules from *domain objects* onto *concrete representations* is defined. As such, this language is the cornerstone of our approach: UIML itself specifies the different aspects of a user interface that should be defined, but does not dictate the vocabulary that is used to design the user interface. The mapping vocabulary can be changed according to the target domain.

We use the rendering engine Uiml.net [4] that transforms a UIML document into a concrete working user interface. Uiml.net is an open source implementation that is suitable for rendering UIML 3.0 compliant user interface descriptions [7]. This rendering engine will query a mapping vocabulary and instantiate the appropriate widgets from the selected widget set at run-time. UIML separates the user interface specification from its concrete representation. The concrete user interface can be automatically

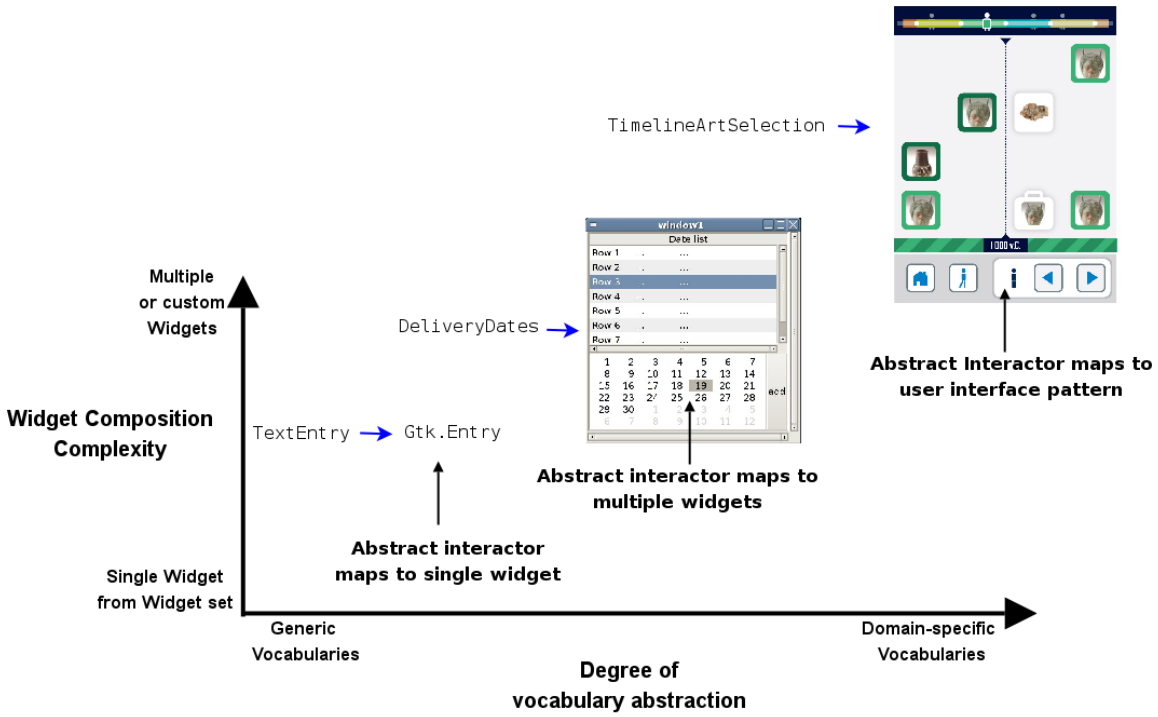


adapted when another mapping vocabulary is used that contains different mapping rules with other concrete widgets. The Listing on this page shows an extract from a UIML file that specifies the user interface for an audio player, with at the bottom of the example a reference to the specific vocabulary that should be used (indicated by the *<peers>* tag).

The figure on this page shows a GUI builder whose workspace has been generated from a specialized audio vocabulary. The level of abstraction that can be used in the vocabulary is not constrained: in this example the objects "timer", "playlist" and "song" are used. Each domain object is mapped onto a visual representation that can be used to build a concrete user interface. A different presentation for the same set of domain objects can be obtained by replacing *audio-simple.uiml* with *audio-mobile.uiml* for example.

MAKING DOMAIN OBJECTS EXPLICIT

Some changes were made to the UIML standard to have better support for the generation of domain-specific GUI tools. Since the mapping vocabulary contains a concrete representation for each domain object, they can be presented directly in the GUI Builder. However, the standard vocabulary only allows mapping one domain object onto one widget class. Our approach allows mapping a domain object onto a composition of widgets or even a user interface pattern. Typically the vocabulary encodes rules such as *PushButton → Gtk.Button* indicating that a part of the class PushButton in the UIML structure description (see the example listing) should be mapped onto a Gtk implementation of a button. Our extension allows to specify rules such as *DeliveryDates → { Gtk.List | { Gtk.List, DateEntry → {Gtk.Calendar, PushButton}} }.* The curly brackets "{}" indicate a composition when the delimiter is "," or a choice when the delimeter is "|". This is encoded in an XML format in the vocabulary which allows the mapping rules to be hierarchically structured as shown in the example rule above. This also implies that the level of abstraction supported by the tool can differ. A "traditional" GUI Builder tool, intended for a designer that is used to work with regular design tools such as the Visual Studio Forms Editor or Eclipse Visual Editor, can be generated from a vocabulary which maps GUI domain objects (e.g. a "Button") onto concrete widgets (e.g. a Gtk.Button). The toolbox would then consist of a collection of single widgets that can be dropped onto a canvas. However, the real value comes from creating a tool for other domains by mapping domain concepts onto suitable user interface patterns, e.g. a user interface design tool for a "car navigation system" or for "audio software". Figure 2 shows a design tool generated from the audio-simple vocabulary. The tool palette clearly shows some iconic representations of domain objects that occur within the domain of audio applications. The most powerful property of this approach is that a vocabulary that only provides a low degree of abstraction can evolve into a vocabulary of a high degree of abstraction as the designer gains more knowledge of the domain. The figure on the next page provides a visual



representation of the type of mappings realized in a vocabulary.

GENERATION OF A DOMAIN-DEPENDENT GUI BUILDER

GUI Builder tools typically allow designers to compose a user interface by using drag-and-drop operations that move objects from a toolbox into a graphical canvas. The most common dialogs (be it integrated in a single view or a multi-window view) are the *toolbox*, the *canvas*, the *property dialog* and the *tree view*. The toolbox contains a set of representations of domain objects that can be dragged onto the canvas and their properties can be changed in the properties dialog.

The above description of the different dialogs is identical to the traditional structure of a GUI Builder tool. However, both the toolbox and property dialog are automatically generated from the domain vocabulary. The other windows contain different views on the UIML description of the user interface that is being designed. Designers interact mainly with the canvas that provides a concrete, graphical view on the user interface. All changes on the canvas (e.g. resizing a widget) are automatically reflected in the underlying UIML description. This reveals another important advantage: although the underlying language abstracts the user interface away from its final platform or device, the designer can still benefit from a concrete graphical view on her/his design.

We have observed that many multi-platform tools were hard to use since the gap between the mental model of the designer and the presentation the tool offers was too big. Most multi-platform tools do not present a concrete view to the designer but rather abstract their visualization. Examples of such tools are CanonSketch [3], VisiXML [2] and Damask [1]. With CanonSketch the designer can focus on content instead of presentation since it provides an abstract iconic notation instead of a concrete graphical prototyping environment and focus on interactive aspects of the dialog being designed. Damask focuses on early phases in the design process that require a high level of creativity. VisiXML is a graphical editor for designing mid-fidelity prototypes on top of the Microsoft Visio environment and that can save the design in a XML-based language that can also specify the user interface at an abstract level like UIML. Although these

approaches have their own benefits, we feel most designers prefer to also have a concrete representation during their design activities. This way they are able to polish the user interface without having to imagine first what the final user interface would look like [8], which reduces the mental burden on the designer.

The user interface builder tool presents the domain objects as a set of items in a tool palette from which the designer can drag and drop items on a canvas. The canvas will directly use the concrete user interface representations as encoded in the vocabulary. This allows the designer to use the concrete representations for domain objects instead of the abstractions.

MULTI-PLATFORM DESIGN BY EXAMPLE

Since a concrete representation shows the user interface for one particular situation, namely the screen size shown by the tool builders' design canvas, we need a more intelligent tool that can support a user interface design for multiple situations. Our tool supports this by allowing the designer to create multiple user interface design examples for the same application. Imagine we want to use an audio player which has a user interface designed by our tool on many devices that have different screen sizes. A designer can freely choose for which screen sizes she or he provides alternative user interface designs. Unlike most other approaches that try to automate this process with one single user interface specification, we support many user interface designs. This means the designer has more control over the presentation of the user interface on different platforms, while the user interface design is still flexible enough to be deployed on a wide range of other devices. Because of the multiple examples the designer can provide, we can ensure the user interfaces generated from the UIML specification adhere as close as possible to the decisions made during the design stage.

Two different techniques are used: first of all we can take advantage of multiple mappings that are included in the vocabulary and thus choose an alternative presentation for the same domain. A second technique is based on the multiple user interface design examples that can be created as discussed in the previous paragraph. The Uiml.net renderer is extended with a user interface interpolation mechanism which can combine the properties of two different user interface designs and create a new user interface for another screen size while maximizing the preservation of the properties of the two example designs. The user interface interpolation mechanism uses a set of rules to decide which properties should be selected and adapted for the new screen size. Each rule couples a user interface property to a minimum and maximum screen size within which the property is valid. The designer specifies these rules which are later encoded in XML and added to the UIML specification. The rules in the UIML specification can be interpreted by our UIML renderer and will select the appropriate properties according to the screen size for which the renderer is generating a final user interface.

At design-time, the designer can specify these rules for several screen size intervals in the generated design tool. Assume, for example, an audio player: when the screen size reduces, it would be preferable to map the playlist to a smaller component and to shrink the visualization component. Other possible transformations include resizing, replacing or removing user interface elements. In our tool, a set of sliders indicate the screen size intervals for which an example is valid. The effects of changing the screen size intervals can be tested at

design-time, since the user interface builder continuously renders the user interface that is being designed with the Uiml.net rendering engine.

DISCUSSION

We presented a Meta-GUI-builder tool that enables a designer to design multi-device user interfaces for a certain domain. The GUI builder tool is generated from a domain vocabulary and as such can be used for different problem domains. Furthermore, it considers a user interface design as an example user interface for an application using a specific screen size. Considering the growing importance to deploy a user interface on different devices, we want to point out the fact that there is little or no designer intervention possible in current design tools to accomplish this without the designer losing controls over her or his design. Different examples for different screen sizes can be created by our tool so the user interface can be used for multiple devices with different screen sizes. For the "intermediate" screen sizes where there are no examples, an interpolation is calculated between the next example for a larger screen size and the next example for a smaller screen size. Inspired by existing approaches such as Supple [5] and Uniform [6], we seek to give the designer more power in the design process in order to constrain the generated designs to the examples given by the designer. Finally we want to emphasize that both the extensible multi-platform UIML renderer and the design tool are available as free software from http://research.edm.uhasselt.be/uiml.

Acknowledgments

Part of the research at EDM is funded by ERDF (European Regional Development Fund), the Flemish Government and the Flemish Interdisciplinary institute for BroadBand Technology (IBBT). The AMASS++ (Advanced Multimedia Alignment and Structured Summarization) project IWT 060051 is directly funded by the IWT (Flemish subsidy organization).

REFERENCES

[1] 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)*, San Francisco, 2002, pp. 573-580.

[2] Adrien Coyette and Jean Vanderdonckt. A Sketching Tool for Designing Anyuser, Anyplatform, Anywhere User Interfaces, Proc. of 10th IFIP TC 13 Int. Conf. on Human-Computer Interaction Interact'2005, Italy, Rome, 2005, pp. 550-564.

[3] Campos, P. and Nunes, N. Canonsketch: a User-Centered Tool for Canonical Abstract Prototyping**.** In *Proceedings of DSV-IS'2004, 11th International Workshop on Design, Specification and Verification of Interactive Systems*, 2004.

[4] Kris Luyten and Karin Coninx. Uiml.net: an Open Uiml Renderer for the .Net Framework, CADUI'2004, Funchal, Madeira Island (Portugal), 2004

[5] Krzysztof Gajos and Daniel S. Weld. SUPPLE: Automatically Generating User Interfaces. In Proceedings of IUI'04. Funchal, Portugal, 2004

[6] Jeffrey Nichols, Brad A. Myers and Brandon Rothrock. "UNIFORM: Automatically Generating Consistent Remote Control User Interfaces," In *Proceedings of CHI'2006*. pp. 611-620.Montreal, Canada, April 22-26, 2006.

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

[8] Luca Cardelli. Building User Interfaces by Direct Manipulation, ACM Symposium on User Interface Software and Technology, 152-166, 1988