A Domain Specific Language for Multi User Interface Development

Alexander Behring, Andreas Petter, Max Mühlhäuser

{behring, a_petter, max}@tk.informatik.tu-darmstadt.de

Abstract: User Interface Development is increasingly facing the demand that an application must provide different User Interfaces (UIs) for different contexts of use, e.g., interaction device and primary task. This leads to two key challenges: how to create these multiple UIs for one application (creation challenge), and how to consistently modify them (modification challenge). The creation challenge has been addressed in various works utilizing automatic UI generation. We present a domain specific language (DSL) suitable to address the modification challenge. The DSL makes use of explicit relations between different UI versions, along which modications of the UIs can be propagated. With the presented approach, modifications can be applied more easily, which is important for iterative (UI) design.

1 Introduction

Technological advances allow us to interact with computing platforms in a great variety of situations. Form factors of devices are getting smaller and the devices are more and more affordable for a bigger audience. We are currently seeing an increase in the number of user interfaces an application has to provide. Commercial services already are providing different interface versions for one application – not only for desktop computers but also for mobile devices. Examples beyond simple train, airplane or bus schedule services are electronic boarding-passes and the possibility to buy electronic public transportation tickets for and with mobile devices.

These user interfaces (*UI*s) differ depending on the *context of use*. Characteristics of the user, the used computing platform (incl. software) and the environment make up the context of use. We take over this notion from Calvary et al. [CCD+04]. On an abstract level, we divide the development of such UIs for multiple contexts of use into two challenges: the creation challenge, and the modification challenge.

The **creation challenge** focuses on the initial creation of a UI. The user interface to be created can be build on the green grass. On the other hand, for the **modification challenge** it is assumed that an existing UI is to be updated. To cope with the updating and allow the developer to control how the existing UI is updated, is the issue for the modification challenge.

Our approach to address the modification challenge is based on the key idea to allow one modification to change multiple UIs. In order to control the propagation of a modification,

UIs are ordered in a tree. The modification is ”passed” down the tree along associations between the UIs. Changing these associations allows the UI engineer to adapt the propagation of modifications to suite the problem at hand. The unique feature of our domain specific language (*DSL*) is that its application constructs this tree and thus allows one modification to be applied to multiple UIs.

In this paper, we present a DSL to model user interfaces and address the modification challenge. The DSL is based on a set of basic requirements, presented in the following. In section 2, the concepts of the DSL (its semantic) is introduced. In the subsequent section, the metamodel (abstract syntax), expressed in EMF (Eclipse Modeling Framework)[1] is described. Constraints (well-formedness rules) presented thereafter also belong to the DSL and ensure the validity of the metamodel instances. Finally, the use of the DSL, with its concrete syntax, and its integration in tools is laid out. We conclude with a discussion of the presented work and related approaches.

1.1 Requirements

The DSL was designed based on a set of four basic requirements, presented in this section. Further discussion of the topic can be found in [BPFM08, BPM09].

Requirement 1: *UI engineers provide information at the level of abstraction suitable for the problem at hand.*

The Model Driven Architecture (*MDA*) also pursues this goal [KUW02]. Especially, changes to the UI can be situated at different levels (e.g., for all UIs or just for the iPhone version, cf. figure 1). Hereby, the *1.1) number of abstraction levels* (cf. section 2) suitable for a UI may vary depending on domain and application. Furthermore, the *1.2) nature of abstraction* shall not be fixed: the developer should be free to choose whether to abstract from, e.g., user characteristics, the computing platform or the UI toolkit.

Requirement 2: *The approach must easily be extendable to new UI toolkits.*

In order to not limit the scope of the approach, easy extension to new UI toolkits is an important property, as Myers et al. note in [MHP00].

Requirement 3: *Support full control over the UI look and feel for UI engineers.*

Automatic approaches hold great potential to increase efficiency, but their interfaces are prone to usability problems and lack aesthetic quality[2] [MHP00, MVLC08, DMLC08]. Thus, *3.1) the UI engineer must be able to manually modify all UIs* (also generated ones), and *3.2) be in control of how modifications are applied*. Furthermore, according to Myers

[1] http://www.eclipse.org/emf

[2] Of course depending on the definition of aesthetic quality and the degree of automatization.

et al [MHP00], the control over the "low-level pragmatics of the interactions look and feel" is important for UI engineers. Therefore, *3.3) the approach should rather focus this detailed control* than on the commonalities of different UI descriptions.

Requirement 4: *Conceptualize for ease of use for UI engineers.*

Ease of use is crucial for the adoption of an approach. Especially a *4.1) closest-as-possible resemblance between edited artifact and resulting UI* guarantees that the UI engineer is not working on abstract artifacts that isolate her from the concrete interface. It lowers the threshold of use and reduces unpredictability problems. Myers et al note these problems in [MHP00] and conclude that this was one of the reasons, why User Interface Management Systems (UIMS) did not catch on.

2 Modeling Concepts (DSL Semantic)

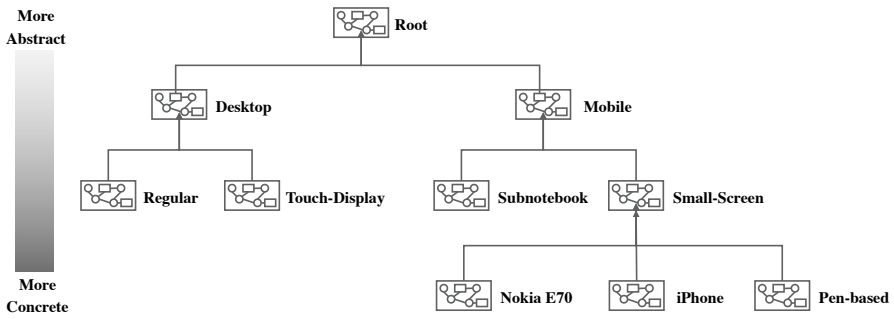

Figure 1: An exemplary *refinement tree* showing multiple levels of refinement. More abstract UI models (at the top) are refined to more concrete versions.

This section introduces, after a brief overview, the concepts used in the DSL (i.e. its semantic), before the corresponding metamodel (abstract syntax) is presented in the next section.

When creating a UI for a new context of use, the UI engineer bases the new UI on an already existing one (she refines it). This new **UI refinement** can be modified freely (e.g., elements added, removed, and properties changed) to suite the new context of use at hand. Along with the new refinement, associations between the two UIs and their constituents are created to document the refinement relationship of the new refinement and the UI it is based on. Applying this refinement step for multiple contexts of use creates a tree of UIs ordered according to the refinements made (cf. figure 1). Inside each UI, Interaction Objects (introduced below) describe the UIs look and feel. To apply a modification to multiple UIs, it is passed along the associations between the different UIs. The UI engineer can influence this propagation by changing the refinement associations between the UIs.

The next sections discuss the elements to describe a UI in detail (Interaction Objects), how they are classified (Interaction Object Class) and how refinement is accomplished in our approach.

2.1 Interaction Objects

An *Interaction Object* is an entity representing a form of interaction between the user and the application. They can appear in a concrete form (Concrete Interaction Objects), which Users can perceive and/or manipulate, e.g., a combo box, a GUI button, a sound or a button on an interaction device. They also appear in an abstract form (Abstract Interaction Objects) abstracting away from one or many concrete representation, e.g., a select-one-of-n or an action invocation. Interaction Objects are the basic building block used in our approach for UI modeling, they are instantiated to model the user interface.

In our opinion, a clear division into Abstract and Concrete Interaction Objects is infeasible. A concrete element like an HTML Combo Box for example has different representations on different platforms, e.g., Mac OS Safari versus Windows Internet Explorer. Elements of the XForms toolkit[3] (e.g., select, group and submit) can be mapped to different representations. The distinction between abstract and concrete thus becomes blurry: given the set of an AUI element (CTT approach, [MPS04]), an HTML element, an XForms element and a Swing element, this set cannot clearly be divided into abstract and concrete elements – where do you draw the line? As further an arbitrary number of refinements steps is required (requirement 1), we do not use the distinction into concrete and abstract, but allow Interaction Objects at arbitrary abstraction levels to be used.

2.2 Interaction Object Class

Similar to UML Classes [Obj05b], Interaction Objects must be assigned a type. The type can, e.g., be HTML Combo Box, XForms Select or Swing JPanel. In order to type an Interaction Object, it is assigned an Interaction Object Class. The class implies the attributes an Interaction Object can have.

Interaction Object Classes are organized in libraries, allowing an easy extension to new toolkits (requirement 2). A library provides all classes relevant to support a given UI toolkit (e.g., Swing, SWT or XForms). Hereby, inheritance relationships between the classes can be modeled, as well. The specializing class inherits all attributes of the generalized[4] class.

[3] http://www.w3.org/MarkUp/Forms/

[4] Please note that we use the term *generalized* in the context of class inheritance, in order to distinguish it from *abstracted*, as used in the context of refinement.

2.3 Refinement and User Interface Boxes (UI Box)

The user interface adapted to a given context of use is defined by arranging Interaction Objects in a UI Box. The UI Box is a container, which has a context of use assigned to it.

To specialize a UI for a new context of use, a more abstract UI is refined. For example, the iPhone UI in figure 1 refines the more abstract Small-Screen UI. Hereby, the UI engineer modifies the UI to make it more specific for the new context of use. Repeated refinement results in a *Refinement Tree* of UIs, as shown in figure 1. The nodes in the figure represent user interface models (UI Boxes) for different contexts of use. UIs in the tree are ordered from abstract to more concrete (refined) UIs, with a single topmost (*root*) UI. The more concrete (*refining*) version refine the more *abstract* versions.

This refinement allows the UIs to be at any level of abstraction (cf. our remark on Abstract and Concrete Interaction Objects). The UI engineer can build refinement trees as deep as needed – no number of refinement levels is prescribed – and thereby provide the information at the level of abstraction suitable (requirement 1). This includes choosing the nature of abstraction. For example, whether to abstract from a specific platform, or a user characteristic. Our approach thus allows the refinement as needed by the concrete problem at hand.

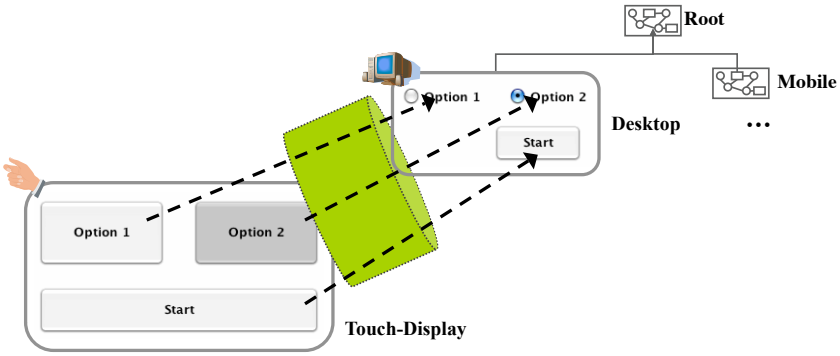

Figure 2: Refinement associations on the UI Box level (big dotted tube) guard ("tube") the refinement associations on Interaction Objects level (dashed arrows), as formulated in the Nesting Consistency Constraint (cf. section 3.1.5).

The key idea to address the modification challenge, is to allow one modification to change multiple UIs (connected in the Refinement Tree). Edges connecting the UIs for the different contexts of use are called *Refinement Associations*. Their semantic is "concretization for a given context of use". Besides the association on UI level, more detailed information on Interaction Object level is needed to unambiguously identify which elements are refining each other. The Interaction Objects themselves are connected via refinement associations, too, as illustrated in figure 2. Using this concept, one modification can change multiple UIs at once by being propagated along the refinement associations.

When applying a modification in a UI, it can either affect *i)* the properties of Interaction Objects, or *ii)* add or remove Interaction Objects. Property modifications are propagated, as retained in the model by the UI engineer through the use of refinement associations. These associations can be interpreted as consistency conditions: if refinement is in place, modifications occur in all refining UI versions – the refinement association describe what features of the UIs are automatically kept consistent. The latter modification to add and remove Interaction Objects is of a different nature and thus supported differently. In our opinion, it can only be addressed through also providing adequate tool support.

3 Metamodel for the Domain Specific Language (Abstract Syntax)

Based on the concepts of the previous section, we developed a metamodel (abstract syntax) for describing user interfaces. It is formulated using EMF (Eclipse Modeling Framework), a framework for model-driven development in the Eclipse[5] project. EMF was used to produce model code – to be able to instantiate metamodel instances at runtime – and build editors to create and modify user interface models using the presented approach.

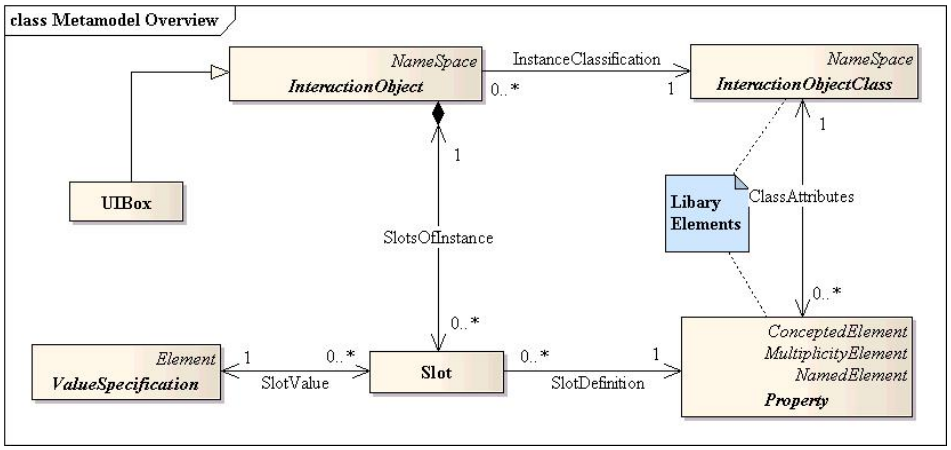

Figure 3: Overview of the metamodel. An Interaction Object can have properties, must be typed, and can be nested. Interaction Object Classes are used to type Interaction Object and define which properties an Interaction Object can have.

In figure 3, an overview of the metamodel is given. Interaction Object Classes are represented by an element of the same name. Classes can have properties by attaching them via the Class Attributes association. A Property can only belong to one class. Both, Interaction Object Classes and Properties, are held in libraries and are referenced by Interaction Object and Slots. These references are not bidirectional, libraries are independent of UI descriptions (Interaction Objects in UI Boxes).

[5] http://www.eclipse.org

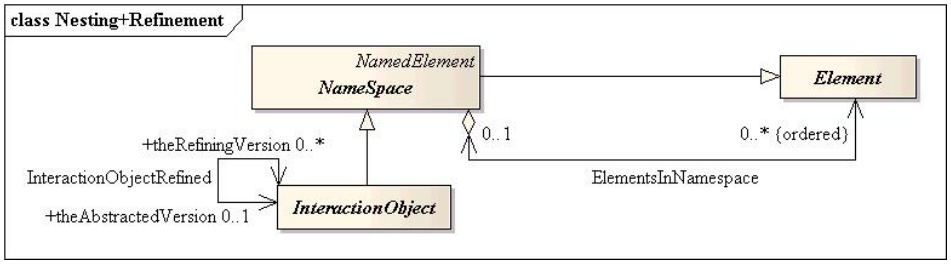

Figure 4: Implementation of the nesting and refinement features in the metamodel.

The actual UI model consists of instances of UI Boxes, Interaction Objects, ValueSpecifications and Slots (depicted in the middle and the left side of figure 3). An Interaction Object references its *classification* via the association Instance Classification. Assigning *property values* is similar to UML [Obj05b]: Slots mediate Values Specifications with Properties. A Slot references the Property it sets a value for, as well as the Value Specification (i.e. the value) to be set. Value Specifications and Properties can have Slots associated with them, but do not have to (e.g., no value is set).

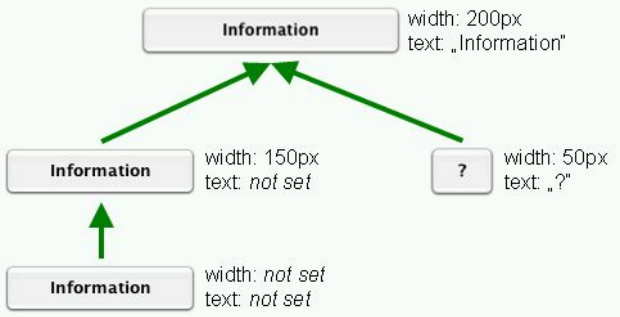

Figure 5: Example illustrating the refinement of properties. The refinement on the lower left inherits its width from the more abstract version and its text from the topmost version.

The *nesting* of Interaction Objects is illustrated on the right side of figure 4: an Interaction Object inherits from Namespace, which can contain Elements. In turn, Interaction Objects are Elements. In the same figure (left side), our implementation of *refinement* is depicted: an Interaction Object refines another Interaction Object via the *Interaction Object Refined* association. Hereby not only Interaction Objects can refine each other, but also UI Boxes, as they inherit from Interaction Object. Furthermore, the *refinement association is transitive*: if a property value is looked up in the more abstract version and no value is found, the next more abstract version will be consulted and so on (cf. figure 5).

Refinement of Properties is implemented without a special association. We assume that if an interactor does not set a property value, the value of its abstract version should be used.

So the critical question is, whether the abstract version's property can be unambiguously identified from the refining version's property. It can, because the abstract version has the same property and thus, its value can be accessed and used. Using this approach, a property can either be:

set locally, i.e. the Interaction Object has a value set for the property, or
refined, when it has no value set.

Multiple refinement, similar to multiple inheritance, is not supported in order to allow consistency by construction: an Interaction Object can only refine 0..1 abstract versions.

3.1 Constraints and Formalism (Well-Formedness Rules)

To further clarify the use and well-formedness of an instance of the presented metamodel, constraints are used. In this section, constraints relevant for our approach are presented. The first two constraints are independent of the refinement itself, whereas the latter constraints are all targeted at providing a consistent Refinement Tree.

3.1.1 Library Consistency

When interpreting a UI, and rendering it, toolkits cannot be mixed (e.g., an HTML Combo Box inside a Swing JPanel). We call the corresponding constraint the Library Consistency Constraint. It consists of two sub-constraints: *i)* the library itself may only contain elements of one UI toolkit and *ii)* all elements in a UI Box must belong to one library. Only when both sub-constraints are satisfied, the Library Consistency Constraint holds.

Using our metamodel, we collect all elements belonging to one library inside a library-element using an ownership-association (not in the scope of this paper). The first sub-constraint is fulfilled, if all elements owned by a library element indeed belong to the same UI toolkit. To fulfill this, the library itself has to be specified correctly, which cannot be checked on model-level, but must be done by externally (by a human).

The second sub-constraint is fulfilled, if all elements nested inside a UI Box are typed by classes that all belong to the same library, i.e., all these Interaction Object Classes are owned by the same library element. In contrast to the previous sub-constraint, this can be checked on model-level.

3.1.2 Property Resolvability

As modeling properties in the presented metamodel is based on UML, a similar constraint must be enforced. We adapt the UML constraint [Obj05b], pg. 127, to "a Slot specifies the value of its defining Property, which must be a Property of an Interaction Object Class of the Interaction Object owning the Slot." This can be illustrated in figure 3: when following associations from a Slot element, on the one hand via the Property element, on the other hand via an Interaction Object, the same Interaction Object Class must be met.

3.1.3 Circular Refinements

In order for the presented concepts to work, the tree structure of refinement associations on UI Box and Interaction Object level must be preserved. This means that no circular refinements may exist. For all Interaction Objects, it must thus hold that for every Interaction Object they are refined by, they may not be the refinement of. Hereby, refinement has to be considered *transitively*: if Interaction Object A refines Interaction Object B and Interaction Object B refines Interaction Object C, Interaction Object A refines (transitively) Interaction Object C.

When regarding transitivity of refinement between UI Boxes, three possible relations between two UI Boxes in a Refinement Tree exist. UI Box *A* relates to UI Box *B*:

refined,	*A* (transitively) is the refinement of *B*,
abstracted,	*B* (transitively) is the refinement of *A*, and
related,	*B* and *A* (transitively) refine the same UI Box *C* (not identical to *A* or *B*).

When regarding Interaction Objects that are not UI Boxes, a fourth type is possible: *unrelated*. That is, two Interaction Objects are not connected via refinement associations. Taking a different view point, we can also note that for a given Interaction Object that is not a UI Box, there is not necessarily a related Interaction Object in the other UI Boxes.

3.1.4 Refinement Condition

Naturally, not all combinations of refining Interaction Objects are possible when refining. Trying to refine an HTML Combo Box to a Swing JPanel must lead into inconsistencies; *i)* their purpose is different, the combo box is used to select an item, the panel to group elements, and *ii)* their properties are incompatible. Checking whether both Interaction Objects serve the same purpose is currently ongoing research in our group, but the property compatibility can be formulated: an Interaction Object B can only refine an Interaction Object A, if matching properties of both Interaction Objects (as they are defined by their classes and the classes generalizations) are compatible. Compatible means that the types of the properties can be converted into each other. Matching means that the properties have the same name.

If we assume that upward modification propagation is not possible, i.e., the UI engineer cannot modify a UI and expect more abstract UIs to reflect this modification, compatibility can be specialized. Compatible then means that all possible values of the property of *A* can be converted to values of the property of *B*.

3.1.5 Nesting Consistency

When refining a UI Box, the nesting order of the affected Interaction Objects should not be changed. For example, a panel contained inside a tab should not be changed to the tab being inside the panel in the refined version. This would produce inconsistent semantics of the groupings (”why elements are grouped in the panel, what is their commonality”).

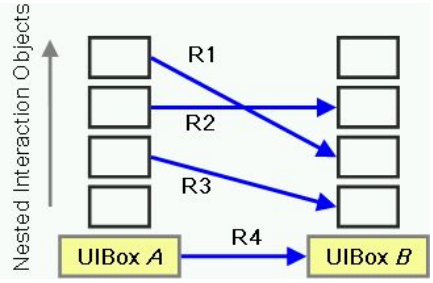

Figure 6: Illustration of a violation of the Nesting Consistency Constraint. Arrows depict refinement associations (R1 – R4), nested Interaction Objects (small boxes) sit on top of their parents. The crossing of R1 and R2 violates the constraint.

Note that, on the other hand, insertion and deletion of Interaction Object must be allowed. Figure 6 illustrates this: the lowest element in UI Box A is removed, whereas the topmost element in UI Box B is added (neither has a refinement partner in the other UI).

Thus, the constraint is formulated: for a refinement of UI Box *A* to UI Box *B*, for every nested element X in *A* it must hold that the refinements of X's nested element are not parents of X's refinement. A violation of the *Nesting Consistency* constraint is illustrated in figure 6.

3.2 Integration of the Domain Specific Language and Concrete Syntax

Besides a clear definition of the metamodel (abstract syntax) and its constraints (well-formedness), it is important to consider the use of the DSL, especially it's appearance to the UI engineer (concrete syntax). Especially in the area of UI development, adequate tool support is crucial, otherwise, the approach will not survive (for some examples, see [MHP00]).

The UIs created and modified using the presented metamodel can either be transformed into code or an intermediary artifact (*generator approach*) or be directly interpreted (*runtime interpretation*), as noted in [Pin00]. We chose the interpreter approach, because using the generator approach, code and model have to be kept synchronized (known as the *round-trip problem* [HT06]). Using the interpreter approach, model executability is achieved automatically. This facilitates an experimental approach to learning the use of the language and more direct feedback [Sel03]. The UI engineer can more quickly evaluate changes to the UI, which is beneficial for them [DRO07], especially in iterative processes. [6]

For editing models, we put forward the use of WYSIWYG editors (cf. requirement 4). Myers et al. identified in [MHP00] user interface builders as a successful tool approach and noted that their advantage is the conceptual match of manipulating graphical interfaces

[6] Regarding performance, our current implementation of a Swing interpreter shows only very small delay when loading and initially interpreting the UI, while the interaction with the UI has no perceivable delays.

by graphical means. Furthermore, direct feedback on changes and experimentation is possible. Thus, the concrete syntax in our case is determined by the implementation of the Swing UI toolkit, or – more general – by the editor used.

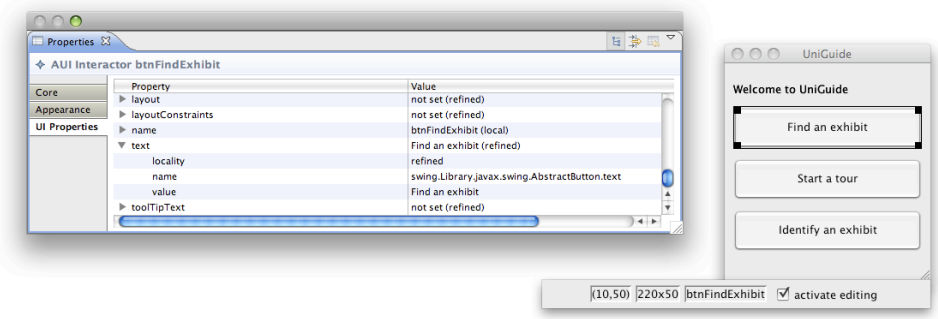

Figure 7: The interpreter and editor (right side) together with the synchronized Eclipse properties page. The property page shows the currently selected item (button "Find an exhibit").

The refinement concept presented in this paper has been implemented in Eclipse-based tools. A WYSIWYG-editor and a special Eclipse view to investigate refinement associations were created. Part of the tools can be seen in figure 7. The interested reader can refer to [BPM09] for more details.

4 Discussion

After implementing some first case studies, we conducted a conceptual user study to collect feedback on the presented approach. Seven participants with a great variance in GUI building skills (ranging from no experience to very skilled interface builder users) had to complete two tasks. They were given a set of ready-made UIs they had to modify. Task one consisted of replacing text in all UIs, whereas task two was to add tooltips to existing elements. After a brief introduction, the participants were asked to complete both tasks without the refinement concept in an interface builder[7], and with utilizing the refinement concept in the Eclipse-based tools.

All participants successfully completed both tasks using the Refinement Associations and had no delays in applying the concept with regard to the use of the interface builder. We can conclude that the concept is easily comprehensible (wrt. industrial-grade UI builders) and thus has a low threshold of use. Furthermore, we discovered through measurements that using the tools with the refinement concept, the error rate was reduced for the tasks: $4\% \pm 6\%$ for our approach versus $21\% \pm 20\%$ for Netbeans. We attributed this to the repetitive and tedious nature when performing the tasks with traditional tools, which was

[7] We used Netbeans for this purpose, because Eclipse does not provide an industrial-grade interface builder, cf. http://www.netbeans.org.

confirmed by the participants feedback. In [BP09], the interested reader can find a detailed description and discussion of the study.

A larger case study was conducted in the context of the SoKNOS project[8]. We integrated the adaptation framework into the SoKNOS portal (UI) and created a SoKNOS plugin to support the users in messaging tasks. Using the approach presented in this paper, we created various refinements for the plugin's UI (e.g., for special roles and different screen sizes and custom hardware). The presented approach allowed us to solely modify the parts of the UI model that changed, inheriting other parts of the UI layout and behavior from more abstract UI versions.

The work presented uses the **library metaphor**, which has the advantages of a stable and simple core language, as well as maximum flexibility with respect to extensibility [AK05] (requirement 2). New Interaction Object Classifiers do not have to be encoded directly into the metamodel, but can be integrated by adding them via a library. This is an important property, as Myers et al. note in [MHP00] in order to not limit the scope of the approach.

Our approach transfers concepts of MDA [KUW02] to the domain of user interfaces. In MDA, a platform independent model is transformed into a platform specific model. When interpreting the different contexts of use as "platforms" in the MDA sense, the presented refinement approach can be seen as multiple applications of MDA-like transformations. The result is a Refinement Tree (cf. figure 1). This tree has no fixed levels of abstraction, does not constrain the nature of abstraction and thus allows the UI engineer to provide the information at the level of abstraction suitable for the problem at hand (requirement 1).

Using the approach presented, the UI engineer has full control over the UI look and feel (requirement 3). The refinement can be controlled by the developer through modifying the Refinement Associations between UI elements. Furthermore, all aspects of the UI elements (properties, type) can be modified, so that the UI engineer has full control over the UI look and feel. Ease of use for the developer (requirement 4) is addressed through tool support, primarily through WYSIWYG editing, avoiding isolation of the UI engineer from the resulting interface by the use of abstract artifacts. This structured and comprehensible approach to the modification challenge allows UI engineers to easily predict the outcome of their modifications and quickly modify multiple user interfaces. Especially iterative processes will thus benefit.

The modification challenge is specifically addressed by our DSL: using the DSL generates the Refinement Tree, in which modifications can be propagated. This allows to apply one modification to multiple UIs simultaneously.

5 Related Approaches

Other model-based approaches for UI development discuss the concepts used for building metamodels such as the one presented here, but do not provide concrete metamodels [SCF+06, CCT+02]. Another important difference to other works is the number of sup-

[8] http://www.soknos.de

ported refinement levels (requirement 1). Most other works on model-based UI development allow two levels of abstraction – an abstract plus a concrete level. In contrast, the presented approach supports an arbitrary number of refinement levels. This allows the developer to choose levels of abstraction suitable for the problem at hand. Also, the UI engineer using our approach is free to choose the nature of abstraction, e.g., whether to abstract from a specific platform or user characteristics.

A number of works focusses on the **creation challenge** - the initial creation of a UI. Questions related to this challenge are about what elements are suitable for the UI, how UIs are structured and how the developer is involved into the creation process. A strong motivation for most works in this area is the adaptation to a newly encountered context of use. Thus, the aim is to support as many as possible contexts of use, often without necessarily knowing and specifying them in advance. This is frequently accomplished by automating the UI creation and adaptation, e.g., as in [CCD+04, NMH+02, MPS04, LVM+04, ZZHM07, MVLC08]. Hereby, modification of existing UIs is not explicitly investigated in these works.

On the other hand, the **modification challenge** focuses on updating an already existing UI. Questions about how the UI developer can apply modifications to UIs, in what way the update is executed, how she can evaluate her modifications and what the impact of a single modification is are prevailing. Damask [LL08] addresses this challenge and provides a pattern and layer concept. Sukaviriya et al. address the modification challenge in [SSRM07] by reflecting changes to business models in UI models.

Many approaches that rather focus on the creation challenge make use of model-to-model transformations (e.g., [MPS04, LVM+04, CCT+02]). These transformations could, in principle, be used to synchronize different artifacts after modifications have been made. Transformation languages like QVT relations [Obj05a], Solverational [PBM09] and triple graph grammars (TGG) [Sch95] can be employed. For example, Sottet et al. [SCF+06] use ATL[9], and Limbourg applies a generic graph-transformation language [Lim04]. Approaches that provide such generic solutions, as transformations do, can be applied to a great range of problem domains. But since they are not focused on the UI, their syntax, and more important their semantic, does not address UI specific issues. Consequently, they are very abstract for the UI engineer to use. But when using abstract descriptions, the connection to the concrete interfaces is often not clear to the engineer [MVLC08, MHP00]. The approach suffers from unpredictability. Furthermore, a new (often complex) language has to be learned, thus raising the threshold of use [MHP00] (requirement 4).

Mori et al. [MPS04] on the other hand encode their transformations into their tool – which implies that they are very specific, but the UI engineer cannot fully influence the transformations anymore. By using the presented mechanisms, our approach tries to connect the benefits of both sides: the UI engineer can fully influence the way modifications are applied and at the same time, the concepts to do so are easily comprehensible and UI specific.

UsiXML [LVM+04] by Quentin Limbourg is a well-known metamodel for UI modeling. The metamodel includes task, abstract and concrete UI modeling. Its Interaction Objects

[9] ATL – Atlas Transformation Language, cf. http://www.eclipse.org/m2m/atl/

Classes are hardcoded into the metamodel. The focus of usiXML is on the creation challenge, hereby a generic graph-transformation language is used. In contrast, our metamodel allows arbitrary refinement levels so that the UI engineer can choose the nature of abstraction suitable to the problem at hand (requirement 1). Furthermore, our approach applies the library metaphor and does not hardcode Interaction Object Classes into the metamodel, which allows for more easy extensibility (requirement 2).

Damask [LL08] is a tool mainly targeted towards UI prototyping. The modification of UIs that already are used in running applications is not Damask's focus. However, the layer concept presented in Damask is similar to the Refinement Tree of our approach. By introducing a DSL able to support arbitrary toolkits and allowing more than two layers of refinement, our work goes beyond Damask and allows a wide range of different contexts of use.

Gummy [MVLC08] is a UIML-based [HSL+08] tool for creating different UIs for different contexts of use. The tool allows creation of the UI in a WYSIWYG-fashion and maintains a UIML description, i.e., the abstract user interface, together with a platform mapping (e.g., to Java Swing), i.e. the concrete user interface. The abstract description can be used to generate new UI versions, but after creation the connection to the source UI is lost: modifications can only be applied to a single UI. In contrast, our approach keeps these connections and thus allows to apply one modification to multiple UIs. The authors of Gummy look into integration of a layer concept (like Damask or our work) into their approach [MVLC08].

Our previous work in [BPFM08] discusses the motivation and requirements behind refinement in more detail, but does not elaborate on the approach, and [BPM09] focusses on tool support. In contrast, the paper at hand presents the refinement approach including metamodel and constraints.

6 Conclusion and Outlook

We presented a DSL to describe user interfaces explicitly targeting the modification challenge. The DSL consists of the metamodel presented (abstract syntax), constraints that constitute its well-formedness rules. The semantic of its modeling concepts was described. The usage of the metamodel was described, its concept and implementation discussed, as well as its concrete syntax. The DSL adheres to the introduced requirements.

We currently research the integration of transformation approaches and the concepts presented in this paper. Because hand-crafting user interfaces for multiple target platforms is a costly task, transformations can be used for automatic generation of UIs. But the aesthetic quality of automatically generated UIs often stands behind that of manually crafted interfaces [MHP00, MVLC08, DMLC08]. Thus, easy manual modifications must be made possible.

References

[AK05] Colin Atkinson and Thomas Kühne. Concepts for Comparing Modeling Tool Architectures. In *MoDELS*, pages 398–413, 2005.

[BP09] Alexander Behring and Andreas Petter. Mapache Dialogue Refinement Tools: a Preliminary User Study. Technical Report TR-10, Telecooperation Research Division, TU Darmstadt, Darmstadt, May 2009. ISSN 1864-0516.

[BPFM08] Alexander Behring, Andreas Petter, Felix Flentge, and Max Mühlhäuser. Towards Multi-Level Dialogue Refinement for User Interfaces. In *Workshop on User Interface Description Languages*, 2008. April 5-10, 2008, Florence, Italy.

[BPM09] Alexander Behring, Andreas Petter, and Max Mühlhäuser. Rapidly Modifying Multiple User Interfaces of one Application. In *ICSOFT 2009*, pages 344–347. INSTICC Press, Jul 2009.

[CCD+04] Gaëlle Calvary, Joëlle Coutaz, Olfa Dâassi, Lionel Balme, and Alexandre Demeure. Towards a New Generation of Widgets for Supporting Software Plasticity: The "Comet". In Rémi Bastide, Philippe A. Palanque, and Jörg Roth, editors, *EHCI/DS-VIS*, volume 3425 of *Lecture Notes in Computer Science*, pages 306–324. Springer, 2004.

[CCT+02] Gaelle 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 '02: Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design*, pages 127–134. INFOREC Publishing House Bucharest, 2002.

[DMLC08] Alexandre Demeure, Jan Meskens, Kris Luyten, and Karin Coninx. Design by Example of Plastic User Interfaces. In *Proceedings of CADUI2008, the 7th International Conference on Computer-Aided Design of User Interfaces*, 2008.

[DRO07] Jr.Ev Dan R. Olsen. Evaluating user interface systems research. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 251–258, New York, NY, USA, 2007. ACM.

[HSL+08] James Helms, Robbie Schaefer, Kris Luyten, Jean Vanderdonckt, Jo Vermeulen, and Marc Abrams (Editors). User Interface Markup Language (UIML) Version 4.0 Committee Draft. Available at http://www.oasis-open.org/committees/download.php/28457/uiml-4.0-cd01.pdf, 2008. Last Access 19.02.2009.

[HT06] B. Hailpern and P. Tarr. Model-driven development: the good, the bad, and the ugly. *IBM Syst. J.*, 45(3):451–461, 2006.

[KUW02] Thomas Koch, Axel Uhl, and Dirk Weise. Model Driven Architecture, 2002.

[Lim04] Quentin Limbourg. *Multi-Path Development of User Interfaces*. PhD thesis, Universit catholique de Louvain, 2004.

[LL08] James Lin and James A. Landay. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In *CHI*, pages 1313–1322, 2008.

[LVM+04] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. USIXML: A Language Supporting Multi-path Development of User Interfaces. In *EHCI/DS-VIS*, pages 200–220, 2004.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

[MPS04] Giulio Mori, Fabio Paternò, and Carmen Santoro. Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. *IEEE Trans. Softw. Eng.*, 30(8):507–520, 2004.

[MVLC08] Jan Meskens, Jo Vermeulen, Kris Luyten, and Karin Coninx. Gummy for multiplatform user interface designs: shape me, multiply me, fix me, use me. In *AVI '08: Proceedings of the working conference on Advanced visual interfaces*, pages 233–240, New York, NY, USA, 2008. ACM.

[NMH+02] Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joseph Hughes, Thomas K. Harris, Roni Rosenfeld, and Mathilde Pignol. Generating remote control interfaces for complex appliances. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 161–170, New York, NY, USA, 2002. ACM.

[Obj05a] Object Management Group. MOF QVT Final Adopted Specification 2.0. Available at: http://www.omg.org/docs/ptc/05-11-01.pdf, 2005.

[Obj05b] Object Management Group. Unified Modeling Language: Superstructure 2.0. Available at: http://www.omg.org/docs/formal/05-07-04.pdf, 2005.

[PBM09] Andreas Petter, Alexander Behring, and Max Mühlhäuser. Constraint Solving in Model Transformations. In Richard F. Paige, editor, *International Conference on Model Transformation, ICMT 2009*. Springer, 2009. to appear.

[Pin00] Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. *Lecture Notes in Computer Science*, 1946:207–226, 2000.

[SCF+06] Jean-Sbastien Sottet, Gaelle Calvary, Jean-Marie Favre, Joelle Coutaz, and Alexandre Demeure. Towards Mapping and Model Transformation for Consistency of Plastic User Interfaces. In Kai Richter, Jeffrey Nichols, Krzysztof Gajos, and Ahmed Seffah, editors, *Workshop on The Many Faces of Consistency in Cross-platform Design, ACM conf. on Computer Human Interaction, CHI 2006*. ACM Press, 2006.

[Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, London, UK, 1995. Springer-Verlag.

[Sel03] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, 2003.

[SSRM07] Noi Sukaviriya, Vibha Sinha, Thejaswini Ramachandra, and Senthil Mani. Model-Driven Approach for Managing Human Interface Design Life Cycle. In *MoDELS*, pages 226–240, 2007.

[ZZHM07] Xulin Zhao, Ying Zou, Jen Hawkins, and Bhadri Madapusi. A Business-Process-Driven Approach for Generating E-Commerce User Interfaces. In *MoDELS*, pages 256–270, 2007.