Model-driven User Interface Development with the Eclipse Modeling Project

Andreas Wolff
Institut für Informatik
Universität Rostock
Andreas.Wolff@uni-rostock.de

Peter Forbrig
Institut für Informatik
Universität Rostock
Peter.Forbrig@uni-rostock.de

ABSTRACT
Model-driven development nowadays often is done using the tools from the Eclipse Modeling Project (EMP). We developed a number of meta-models and transformations to support a model-driven user interface development within EMP. This paper presents our Swing and XUL meta-models and demonstrates what they can be used for.

Author Keywords
UI Model Transformation, CUI Models, Reverse Engineering

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: Miscellaneous

INTRODUCTION
Modeling user interfaces (UI) is a well investigated research topic. For more than twenty years user interfaces are specified in terms of instances of meta-models. Available approaches differ in level-of-detail, some are tailored to a specific domain or interface modalities or target devices, others try to cover general interfaces. A general consensus in model-driven user interface design (MD-UID) exists to distinguish several levels of abstraction. As well as in general model-driven software development, a distinction is made between platform independent (PIM) and platform specific models (PSM). A platform in MD-UID is often considered to be a certain device, or rather the interface source code for that device, itself. Platform independent UI models are called abstract user interfaces (AUI) and platform specific UI models concrete user interfaces (CUI). UI source code is not necessarily source code of a certain programming language, but may as well be an interface description written in a markup language.

There exist numerous markup languages and often they are derived from XML, common examples include XAML, XUL or UsiXML. The main advantage of those languages is the provision of a well-defined grammar and the inherent hierarchical structure, i.e., they provide a suitable meta-model. These characteristics makes them ideal candidates for a model-driven UI development, but not necessarily ideal languages for UI design.

Figure 1. MDD process overview

Non-XML based user interface description languages are even more numerous. They range from comparatively simple template languages to large frameworks as for example Swing or SWT. Creating meta-models, for use within a MD-UID process, of those languages is not as straightforward as with XML-based languages, but we think it might be worth the effort.

In general, we consider model-driven software development as a sequence of transformations of models. We do not think that those transformations can be performed in a fully automated way, but should be executed by humans using interactive tools. We also think these persons, i.e., software engineers and user interface designers; have to base their work on the same models. Our work is especially focused on methods and tools supporting transformations by patterns. Figure 1 sketches all models we consider important within this process and how they are interrelated.

In this paper we focus on the user interface part of the overall process[6]. It is located in the lower right part of Figure 1. UI creation starts off from the task model. A task model itself is not sufficient to express the dynamic relations within an user interface. To capture these, an auxiliary model, the

Pre-proceedings of the 5th International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2010): Bridging between User Experience and UI Engineering, organized at the 28th ACM Conference on Human Factors in Computing Systems (CHI 2010), Atlanta, Georgia, USA, April 10, 2010.

Copyright © 2010 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners. This volume is published by its editors.

dialog graph, was introduced. The dialog graph model eventually is transformed into an abstract user interface, our PIM.

The idea presented in Figure 1 as such is a process model. In the last years we created different implementations for it. A common problem with these implementations was, that they would only cover quite small portions of the overall problem. We think this was mainly due to a scaling problem. With increasing level of detail, the implementations became too large and too complex to handle.

In our recent implementation we make use of the Eclipse Modeling Project (EMP). EMP is ”a unified set of modeling frameworks, tooling and standards implementations”[1]. ”Standards implementations” refers to OMG MDA standards whose reference implementations are developed by EMP.

Figure 2 is a details view on the transformation from ”abstract UI” to ”UI model” in Figure 1. The meaning of the line styles is as follows: a solid line is a model-to-model transformation. Dotted lines denote a model-to-text transformation. Reverse Engineering (RE) is marked using dashed lines. Abstract and concrete user interface (AUI, CUI_x) are EMF models each.

Next to an arrow the applied technique for this transformation was annotated. QVTo[5] stands for OMG’s ”Query View Transformation operational” standard. Acceleo[1] is the EMP implementation of OMG’s ”Model to Text Language” (MTL[4]) standard. PLML[2] itself is a pattern language; we developed an EMF model for it. The PLML model contains transformation specifications in either of EMP’s model-to-model transformation engines. For details on PLML and our usage of it see [9].

Forward engineering from task models typically only yields quite simple user interfaces. To test our ideas against more complex UIs, and for other reasons, we developed some mechanism to reverse engineer the GUI of existing legacy software into instances of our own CUI meta-models. This reverse engineering is not a standard x-to-y transformation.

The rest of the paper is organized as follows: First, we present the XUL EMF meta-model and some of its applications. Afterwards some details about our Swing meta-model and its uses in reverse engineering are discussed.

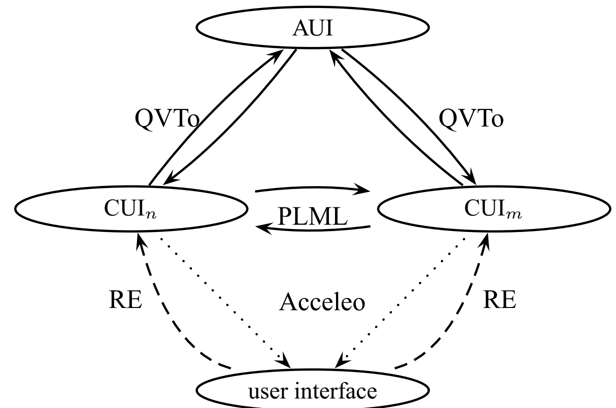

Figure 2. Model relations and transformation techniques

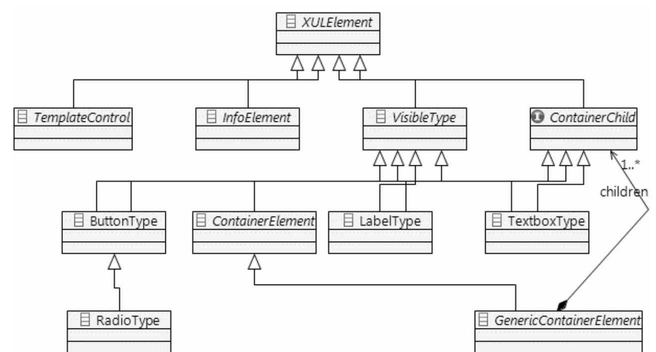

Figure 3. Excerpt from the XUL meta-model

CONCRETE USER INTERFACE MODELS

We consider meta-models for the platform-specific user interface model to be the core models of a model-driven user interface development. In continuation of our previous experience with MD-UID, we decided to develop CUI meta-models for Mozilla’s XUL and Java Swing.

XUL Meta-Model

For a long time our work is focused around XUL. However, most of the time we did not care about creating a complete and correct meta-model of XUL. We adhered to a minimal implementation that was only extended if absolutely necessary. So, the XUL standard evolved over time, but our minimal implementation did not. This started to create a number of problems and errors.

A good starting point to create an EMF meta-model for XUL is its grammar. Using freely available sources, it is possible to build this grammar as XML schema definition (XSD) for XUL.

Having a schema definition, a first Ecore model for XUL was easily generated. This automatically generated model had some disadvantages, e.g. anonymous types, multiple types for the same purpose etc. Also, containment relations were not transformed properly and had to be corrected and inserted by hand.

Figure 3 visualizes the basic hierarchy within the XUL Ecore model, many types were omitted. To give an idea about the size: the model consists of 31 data types, 151 classes and interfaces with 443 attributes in total.

XULElement is the root element of the type hierarchy; it contains attributes which are valid for every XUL tag. Each attribute is initialized with a sensible default value or it remains unset if no value is required explicitly.

TemplateControl is the root type for any tag that controls XUL's template engine. InfoElement is the super type of every tag that is not displayed but defines actions, key bindings and such. Any type that implements ContainerChild can be placed into a visual container. Most implementors are concrete VisibleType objects, some are visual containers themselves.

ContainerElement is a marker class for all elements which are able or require to contain sub-elements. For example: a menu-element requires that menu-items are defined. Subtypes of GenericContainer object are a visual containers that include children of type ContainerChild.

RadioType is a radio-button, ButtonType a plain button, LabelType a label and TextboxType a textbox. Those four types are the only concrete classes of Figure 3.

EMF provides a facility to generate editor plug-ins for Ecore models. Using this standard mechanism unfortunately does not yield a valuable XUL editor. Because the default serialization mechanism of Ecore is XMI, the generated editor would not be able to read or write valid XUL files. We had to write our own implementation of the model de-/serialization.

An attempt to use the XUL Ecore model as source to generate an GMF editor was aborted. The resulting generated editor was hardly useful. While it was possible to do some basic editing of labels or buttons, it became very complex to implement a correct layout mechanism. Also, things like the unlimited nesting of group- or tab-boxes proved to be a serious problem. Nevertheless, it would be interesting to build a GMF XUL editor using the XUL Ecore model. We resorted to continue to use our existing graphical XUL editor, but to generate its internal model by customized JET transformations from the XUL Ecore model.

The complete XUL meta-model can be used as a descriptive model for XUL user interfaces, it is freely available from [1].

Meta-Model for Swing

As mentioned earlier, we don't think that XML-derived UI-languages are the best choice for every problem. Also, we acknowledge the fact that a large part of existing user interfaces was not specified using a markup language, but by other means.

The GUI framework Swing of the Java programming language is one example of a user interface framework. It has been included as part of Java since version 1.2. In contrast to its competing GUI framework SWT, Swing is completely platform-independent. For this reason we decided to build a Swing CUI model instead of an SWT model, although the latter is closer related to Eclipse.

Creating the Swing CUI meta-model means to create an Ecore model of Swing. Such a model can be built using a number of methods. EMF has so called model importers which are able to construct Ecore models from XMI, XML Schema, class models of Rational Rose or from annotated Java source code.

So, to define a meta model for Swing there are several possibilities. For example, it is possible to define it manually; nevertheless we dropped this possibility due to the foreseeable large size of the resulting model.

Importing a rational rose model should work well, but it certainly requires to have such a model in the first place. Preparing an XML Schema definition (XSD) for transformation into an Ecore model has its own challenges, as mentioned for the XUL meta-model, but in the absence of such a schema definition this also does not apply.

This leaves two sources for model creation, either using another XMI model or annotated Java code. As most parts of Java Swing are available in source code, it should be possible to create either of these thereof.

EMF's model importer for Java source code requires that each identifier, method, class or interface that is to be transferred into an Ecore model is marked using the annotation @model. This is a highly flexible approach and very useful if one extracts only small parts of existing source code into a model. However, for Swing's about 620 files this method seemed to be almost as laborious, time-consuming and error-prone as defining the model manually.

Many UML tools serialize their models using XMI. Also examining source code to derive a model thereof is a common feature of these tools. In combination, EMF's importer for XMI and an external tool for source code to XMI transformation should provide the desired model of Swing. Unfortunately it was not that simple. The XMI-output produced by UML tools we tested could not be imported by EMF. Looking back on a history of known problems [3] with XMI incompatibilities we did not investigate why exactly the import failed and if it could possibly be repaired by some XSL transformation or the like.

Instead we were searching for a method to create Ecore from Java sources in one transformation, without any intermediate steps, tools or importers.

Developing yet another Java source code parser apparently would not have been a good idea, because there already is a number of parsers available as library or in source code. After some research we developed a small tool which makes use of Java's own JavaDoc parser and is able to convert any Java code into an Ecore model, for details and special considerations see [8].

Swing's source code, in Java version 1.6, has 620 files, and since Swing makes heavy use of AWT, its 368 source code files had to be included into the model as well. The comprehensive Ecore model now consists of more than 2000 types. Additionally it references about 200 external types which are not defined within the source code of AWT and Swing.

Working with such a huge model reveals some weaknesses

[1]http://wwwswt.informatik.uni-rostock.de/metamodels/

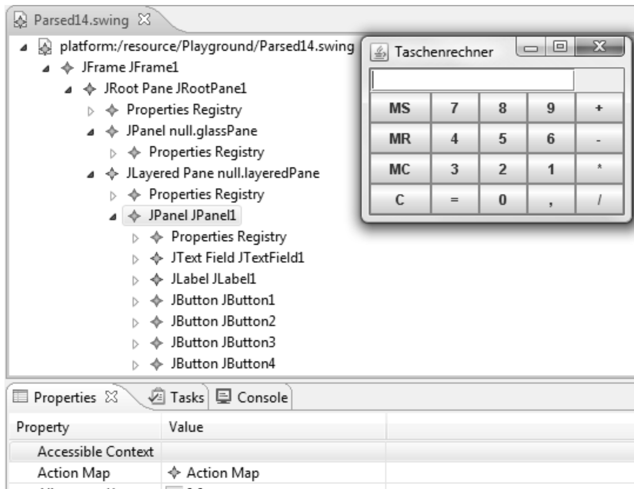

Figure 4. Calculator application and their model screenshot

of the tools of the Eclipse Modeling Project. For example with code generation: The current implementation becomes almost unusable with respect to generation time. Also, some generated methods became larger than allowed by Java, i.e. their compiled byte-code exceeded 64kByte.

To reduce size and complexity the meta-model was pruned using the approach of Sen et al.[7]. Roughly, their idea is as follows: they start with a minimal pruned model that only contains a selection of desired must-have features and classes from the source model. Then their start model is extended step-by-step with features from the source model until their pruned model is valid and is a subset of the source model.

Running this pruning process on the Swing CUI meta-model greatly reduces its size. Size reduction is of course dependent on the selected must-have features, but nevertheless we found that more than 90 percent of the types can be safely eliminated from the CUI model without loosing much expressiveness. This fairly high percentage can be explained by the generation process of the source model. Obviously a lot of implementation-specific types had been generated into the model that were not directly related to properties of UI elements. The complete Swing meta-model, as well as the pruned meta-model, can be obtained from [10].

Populating a Swing model

One use of the Swing meta-model is in reverse engineering. We developed an application which uses the meta-model to capture the static layout of a frame in a Swing application into a meta-model instance. Such a meta-model instance might be considered as a screenshot.

The principle is to traverse the hierarchy of Swing objects within a certain screen and to create the corresponding meta-model instance for each Swing object. Since the meta-model now is a true reflection of Swing itself, no special mappings or other tricks are required. However, some technical problems did arise, because all of the mapping is done at runtime and relies on reflection.

Figure 4 is a combined screenshot of a sample application and its user interface as an Ecore model instance. Beside layout properties, the Swing CUI model does also cover dynamic aspects, like action listener and action command. By a sequence of transformations, we can now edit the user interface in a GUI editor and afterwards re-generate a calculator which would feature another GUI, but uses the same business logic.

CONCLUSION

In this paper we presented our approach to model-driven user interface development using the tools from the Eclipse Modeling Project. We introduced Ecore models for XUL and Swing user interfaces and explained how we derived, build and refined those models. It was also shown that those models can be used for a number of different purposes. These usages include reverse engineering, generating internal models for other tools and also direct editing of user interfaces.

In the future we may resume our attempt to create a GMF-based XUL editor. Beside that, there is a lot of work to do to improve and extend the existing transformations, model-to-model and model-to-text.

REFERENCES

1. Eclipse modeling project (last visited on 22-02-2010). http://www.eclipse.org/modeling/.
2. S. Fincher. Perspectives on hci patterns: concepts and tools (introducing plml). In *Workshop at CHI 2003*, 2003.
3. B. Lundell, B. Lings, A. Persson, and A. Mattsson. Uml model interchange in heterogeneous tool environments: An analysis of adoptions of xmi 2. In *Proc. of MoDELS 2006*, pages 619–630, 2006.
4. Omg standard: Model to text (last visited on 22-02-2010). http://www.omg.org/spec/MOFM2T/1.0/.
5. Omg standard: Query view transformation (last visited on 22-02-2010). http://http://www.omg.org/spec/QVT/1.0/.
6. D. Reichart, P. Forbrig, and A. Dittmar. Task models as basis for requirements engineering and software execution. In *Proc. of Tamodia 2004*, pages 51–58, 2004.
7. S. Sen, N. Moha, B. Baudry, and J.-M. Jezequel. Meta-model pruning. In *Proc. of MoDELS 2009*, pages 32–46, 2009.
8. A. Wolff and P. Forbrig. Deriving emf models from java source code. In *Proc. of Reverse Engineering Models from Artifacts 2009*, 2009.
9. A. Wolff and P. Forbrig. Pattern catalogs using the pattern language meta language. In *Proc. of Visual Formalisms for Patterns 2009*, 2009.
10. Meta-model download (last visited on 22-02-2010). http://wwwswt.informatik.uni-rostock.de/metamodels/.