Rapid UI Development for Enterprise Applications: Combining Manual and Model-Driven Techniques

Arne Schramm, André Preußner, Matthias Heinrich, and Lars Vogel

SAP Research Center Dresden
{arne.schramm,andre.preussner,matthias.heinrich,lars.vogel}@sap.com

Abstract. UI development for enterprise applications is a time-consuming and error-prone task. In fact, approximately 50% of development resources are devoted to UI implementation tasks [1]. Model-driven UI development aims to reduce this effort. However, the quality of the final layout is a problem of this approach, especially when dealing with large and complex domain models. We share our experience in successfully using model-driven UI development in a large-scale enterprise project. Our approach mitigates the problems of model-driven UI development by combining manual layout with automatic inference of UI elements from a given domain model. Furthermore, we provide means to influence the UI generation at design time and to customize the UI at runtime. Thus, our approach significantly reduces the UI implementation effort while retaining control of the resulting UI.

Keywords: Model-Driven UI Development, UI Generation, UI Customisation.

1 Introduction

Enterprise applications, such as the SAP Enterprise Resource Planning or the SAP Transportation Management, are used by companies to execute and manage their business processes. One of their main tasks is to provide CRUD (Create, Read, Update, Delete) functionality for business objects, such as products, customers, business partners, etc. The properties and relations of these business objects are captured in *domain models*. Enterprise applications provide different views on the domain model of a company. Developers of user interfaces (UIs) for enterprise applications are facing a number of challenges.

1. **Development Costs:** Implementing user interfaces for enterprise applications is a labour-intensive and therefore costly task. Developers have to repeat similar working steps, such as choosing widgets and binding data to them, many times. To reduce the effort of UI creation, many model-driven approaches [2,3,4,5] have been proposed. They use several intermediate models to describe user interfaces at different layers of abstraction [6]. Given an appropriate tool support, developers can derive the UIs from models much

D.C. Petriu, N. Rouquette, Ø. Haugen (Eds.): MODELS 2010, Part I, LNCS 6394, pp. 271–285, 2010.
© Springer-Verlag Berlin Heidelberg 2010

more efficiently and in a less error-prone way than implementing them from scratch. A drawback of this methodology with respect to the effort of UI creation is that every UI widget still has to be modelled manually. In particular, the mapping of data fields of the domain model to UI widgets is a recurring task that can be automated to a great extent. However, a fully automated UI generation based on a domain model produces significantly limited interfaces in terms of clarity, understandability, and usability [7], especially for complex models.
2. **Diversity of Users:** Users of an application can have different roles that may require role-specific views on the data they are working with. This forces software companies to develop diverse UIs for different groups of users, instead of providing one UI that satisfies the expectations of all users [2]. Furthermore, the users of an application may have different skills and experience levels, preferences, or cultural backgrounds, and therefore different expectations regarding a convenient UI. Thus, flexible mechanisms for UI adaptation, and customisation are needed.
3. **Software Evolution and Maintenance:** Companies need to be able to quickly respond to trends and changes in the market. Therefore, the IT landscape of an enterprise has to support constant evolution [8]. An example is a transportation company extending their services from nation-wide to international transports. This affects the company's domain model, since addresses of partners and customers now need a new attribute indicating their country. To support setting or reading this attribute, the company needs to change, re-compile and re-install the UI for its transportation management system.

In our approach we address the above mentioned challenges as follows.

1. We combine the benefits of model-driven UI development with automatic UI generation to create meaningful and user centric results with minimal effort. The UI designer creates the layout structure of the application which ensures the clarity, while the atomic UI widgets are automatically generated using information from a domain model.
2. We enable developers to rapidly create a business application for a certain domain, and adapt it to the needs of a specific user role. The user can individually customise the resulting UI.
3. Using our approach, minor changes in the domain model, such as adding or deleting an attribute, are automatically reflected in the UI after a restart of the application. Structural changes, such as adding or deleting business objects, can be incorporated in an application with little effort thanks to our tool support.

The remainder of this paper is organised as follows. Section 2 introduces a real-world example that is used to explain the challenges tackled by our approach. Section 3 describes our hybrid approach for UI development combining manual and model-driven techniques. Section 4 points out the advantages and disadvantages of our approach by means of two industrial use cases. Section 5 lists related work

in the field of model-driven UI development and ad-hoc UI generation. Finally, Section 6 summarizes the paper and proposes ideas for future work.

The ideas presented in this paper were partly developed in the scope of the EU-funded ServFace project.[1]

2 Organisational Context and Running Example

This section gives an overview of our organizational context and its challenges from the UI development perspective. SAP Transportation Management (SAP TM) is a solution for transportation management. This includes applications for the management of transportation requests, which contain, e.g., the business partners, the source and destination locations, or the items to be transported. The customers of SAP TM, typically transportation companies, have very different business models. Some offer complex transports over various locations using diverse means of transportation, while others provide only limited transport services. The differences in the business models are reflected in different requirements for the software support. While the services offered by the SAP TM solution are suitable for most of the customers, and thus the underlying domain model is the same for all these customers, the UIs have to be adapted for each customer to match the needs imposed by the various business models. Until today more than one million UIs have been built on top of the transportation solution. For one customer even 1145 different UI screens have been developed. The development of each of these UIs is a tedious and expensive effort. This motivated us to find a solution for efficient UI creation and customisation. Existing UIs for the transportation management solution consist mainly of elements for editing data (approximately 80%) and only a minor part (approximately 20%) is used for navigation and administration. For this reason, we focus on the creation of form-based editors that provide CRUD functionality for DM elements.

Running Example. To better illustrate our approach, we introduce an example based on a simplified scenario. The application enables users to manage transportation requests. The domain model for this scenario is shown in Figure 1.

The model consists of eight classes containing in total 45 attributes. Creating a form-based editor manually would take some time even for such a small model. Assuming that each attribute needs at least a label and a widget for editing its value, a programmer has to define nearly 100 widgets and bind them to elements in the domain model. In the following section we introduce our UI development approach that reduces this effort significantly.

3 Our UI Design Methodology

In this section we describe how UIs are created following our approach. First, we briefly outline the steps needed to create a UI from an application designer's perspective. In the following subsections we explain in detail the important concepts and components of our approach.

[1] http://www.servface.eu, last visited July 9, 2010.

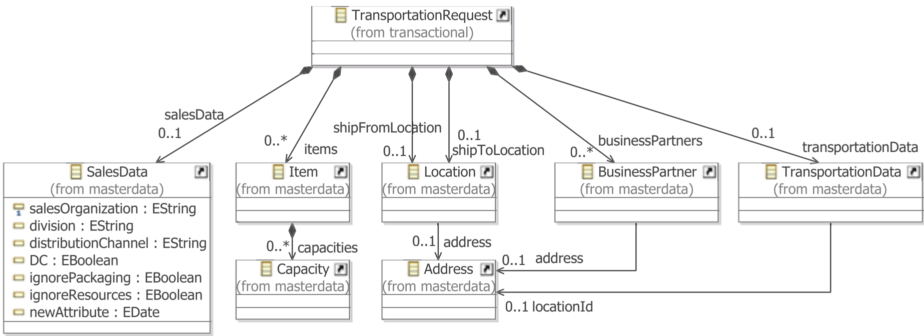

Fig. 1. Excerpt of the Simplified Domain Model for Transportation Requests

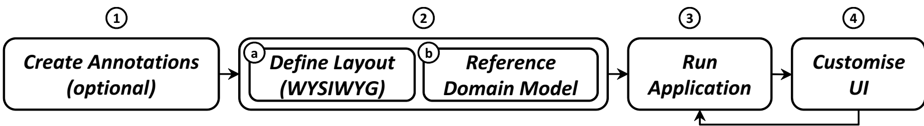

Fig. 2. Application design process

3.1 Application Design Process

Figure 2 gives an overview of the four steps in our application design process.

1. **Create Annotations:** As the starting point we assume that the domain model for the domain, for which an application shall be developed, is given as an EMF model (EMF = Eclipse Modeling Framework)[2]. The developer may add annotations to the model to influence the appearance of the generated UI.We do not have tool support for this step, since EMF provides excellent tools itself.
2. **Define Layout and Reference Domain Model:** To create the UI, the designer defines the layout (a), and sets references to elements of the domain model (b) using a WYSIWYG editor.
3. **Run Application:** Thanks to our interpretive approach the application can now be used without any additional generation or compilation steps. Besides the manipulation of domain model elements, the application provides persistency of data, and convenience features such as auto-completion.
4. **Customise UI:** The UI can be customised individually by the users to adapt it to their needs. After this customisation step the application can be used without a restart.

3.2 Model and Tool Overview

Figure 3 gives an overview of the models and tools that are used to create and customise applications. The numbers refer to the application development steps in Figure 2. The Final Application is generated using an interpretive approach.

[2] http://www.eclipse.org/modeling/emf, last visited May 2, 2010.

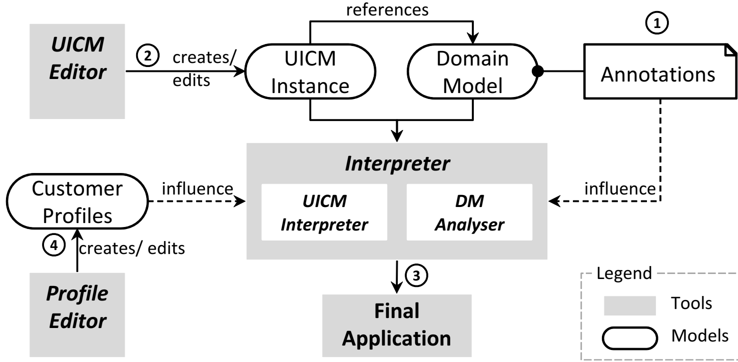

Fig. 3. Model and tool overview

Input for the Interpreter are two models, the UI Container Model (UICM) and the Domain Model (DM). While the UICM is different for each new application, the DM is created once by domain experts, and reused for all applications for that specific domain. The UICM is created using the UICM Editor, which is explained in Section 3.3. The UICM contains two kinds of information, the UI container structure, and the references to DM elements. The generation of the container structure is detailed in Section 3.4, and the generation of UI widgets based on the DM is explained in Section 3.5. To influence this generation, developers can enrich the DM with UI specific meta-data using annotations, which are described in Section 3.6. The resulting application can be customised individually using the Profile Editor as explained in Section 3.7.

3.3 Modelling the Application – The UICM Editor

The UI of an enterprise application must be well-structured to enable users to work efficiently. Automatic UI generation fails to create such well-structured UIs for large and complex domain models. Therefore, we decided to let the designer create the layout of an application's container structure by creating instances of the UICM.

Figure 4 shows an excerpt of the metamodel of the UICM, which we defined using EMF. Although the number of offered UI container elements is quite limited, it is our experience that the available elements are sufficiently powerful. Furthermore, it requires only a very low introductive training and can be used efficiently by designers.

Figure 5 shows the UICM Editor, a WYSIWYG editor for creating form-based applications. It consists of 5 parts: The Controls palette (1) providing the container elements, the Data Objects palette (2) offering the DM elements, the GUI outline containing a tree-view of the UI, the Properties view (4) for manipulating the properties of the currently selected UI element, and the Editor view (5) that provides the modelling area and displays the WYSIWYG representation of the application that is currently built.

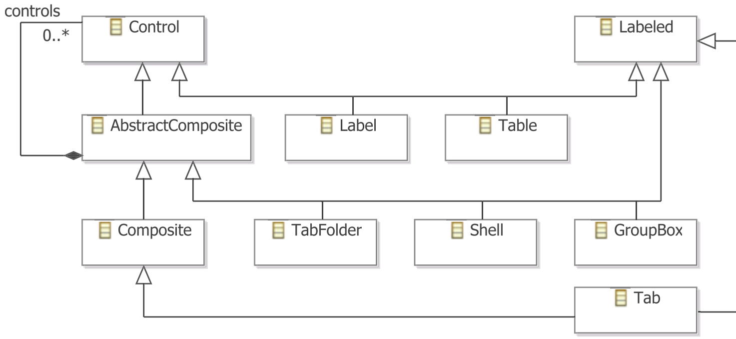

Fig. 4. Excerpt of the UI Container Model

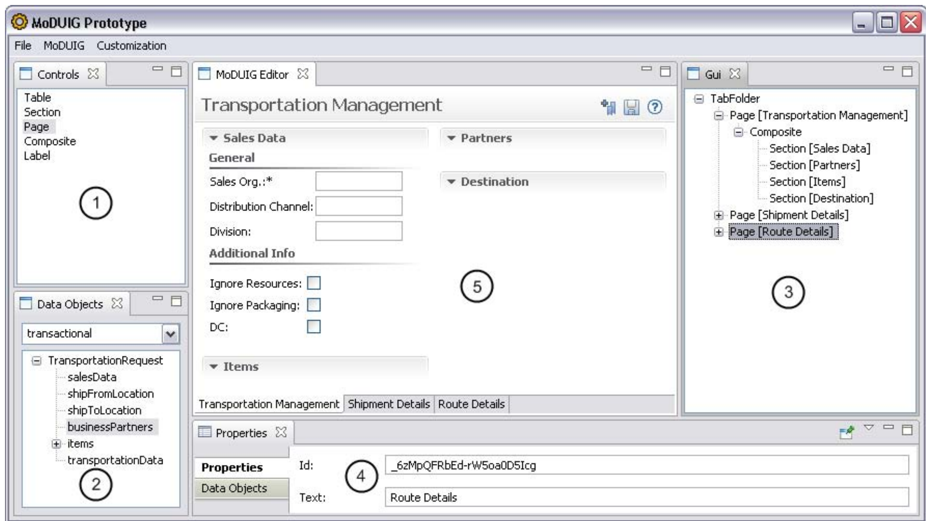

Fig. 5. UICM Editor

To describe the container structure of an application, the designer drags and drops different container elements from the controls palette onto the modelling area (Step 2a in Figure 2). In this manner the designer creates a hierarchy of nested composites on different tabs[3], which can contain several sections. Then he drags and drops DM elements onto the containers. The DM elements are anaylsed and a suitable UI representation is generated and displayed inside the container (Step 2b in Figure 2).

The example application in the modeling area in Figure 5 contains such a container structure. Figure 6 shows an excerpt of the XML representation of the corresponding UICM. The application has three tabs, "Transportation Management" (Line 3), "Shipment Details", and "Route Details". The first tab contains

[3] Our interpreter displays tabs as pages in a multi-page editor.

```
1  <tm.widgets:Shell ...>
2   <controls xsi:type="tm.widgets:TabFolder">
3     <controls xsi:type="tm.widgets:Tab" text="Transportation Management" ...>
4       <controls xsi:type="tm.widgets:Composite" ...>
5         <controls xsi:type="tm.widgets:GroupBox" text="Sales Data" ...>
6           <dataObjects href=".../salesData" />
7         </controls>
8         ...
9       </controls>
10    </controls>
11    ...
12  </controls>
13 </tm.widgets:Shell>
```

Fig. 6. Excerpt of the example UICM

4 sections, “Sales Data” (Line 5), “Partners”, “Items”, and “Destination”. The “Sales Data” section is bound to the “salesData” business object from the domain model (Line 6). The final application is shown in Figure 7.

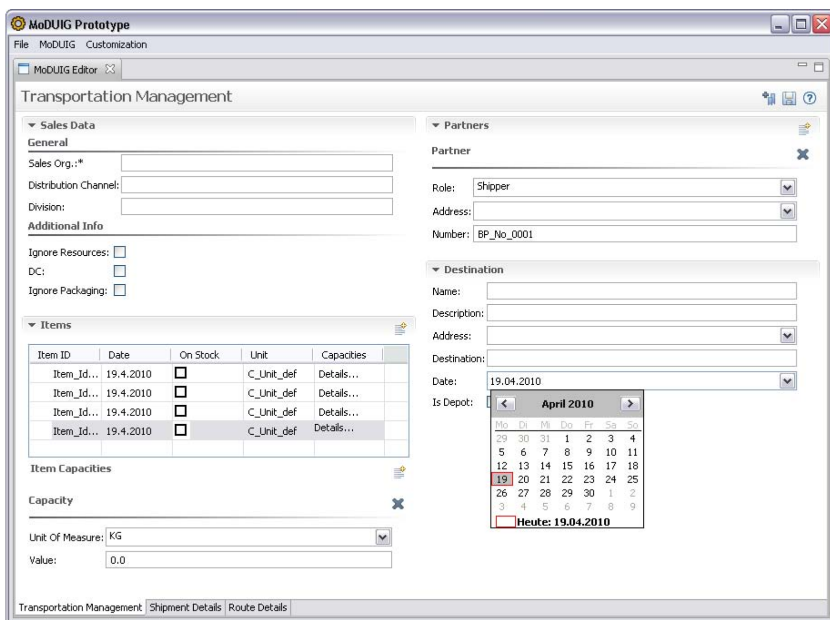

Fig. 7. Final application

3.4 Interpreting the Container Model – The UICM Interpreter

Due to the reasons discussed in Section 5 we decided to use an interpretive approach, and implemented a concrete interpreter using the Standard Widget

Toolkit (SWT).[4] The interpreter follows a strict set of rules to create widgets for the elements in the UICM. The UICM itself, however, is technology-agnostic, and different interpreters could create different UIs from it. The transformation rules applied by our interpreter are shown in Table 1.

Table 1. Mapping of UICM types to UI container widgets

UICM Element Type	UI Container Widget
`Composite`	`org.eclipse.swt.widgets.Composite`
`TabFolder`	`org.eclipse.ui.part.MultiPageEditorPart`
`Tab`	`org.eclipse.ui.forms.widgets.ScrolledForm`
`GroupBox`	`org.eclipse.ui.forms.widgets.Section`
`Table`	`org.eclipse.jface.viewers.TableViewer`

3.5 Inferring the UI Elements – The Domain Model Analyser

If the UICM interpreter finds a reference to a DM element in the UICM, the DM Analyser takes care of creating the atomic UI widgets. Here we take advantage of the fact that attributes of certain data types are usually mapped to a predictable set of widgets. For example, an attribute named "customer" of type "string" is typically represented by a label named "Customer" and an editable textbox. The DM Analyser inspects the structure of the DM element using reflection to get all attributes. Based on the name and type of these attributes, UI widgets are created and placed inside the container that references the DM element. Table 2 lists the most common mapping rules for data type to widgets at runtime, and Figure 8 shows the steps executed by the generator to process one referenced element of the domain model. The UI designer can add annotations to the DM to influence the behaviour of the DM Analyser (see Section 3.6).

In the example application in Figure 7 the designer dragged the DM elements "SalesData", "BusinessPartner", "Item", and "Location" into the corresponding sections "Sales Data", "Partners", "Items", and "Destination". This set references from the sections in the UICM to the respective elements in the DM. They are analysed and the appropriate widgets are generated.

Table 2. Mapping of data types to UI widgets

Data Type	Generated UI Widgets
`string`	`org.eclipse.swt.widgets.Text`
`int`	`org.eclipse.swt.widgets.Text`
`double`	`org.eclipse.swt.widgets.Text`
`boolean`	`org.eclipse.swt.widgets.Button (SWT.CHECK)`
`date`	`org.eclipse.swt.widgets.DateTime`
`enumeration`	`org.eclipse.swt.custom.CCombo`

[4] http://www.eclipse.org/swt/, last visited July 9, 2010.

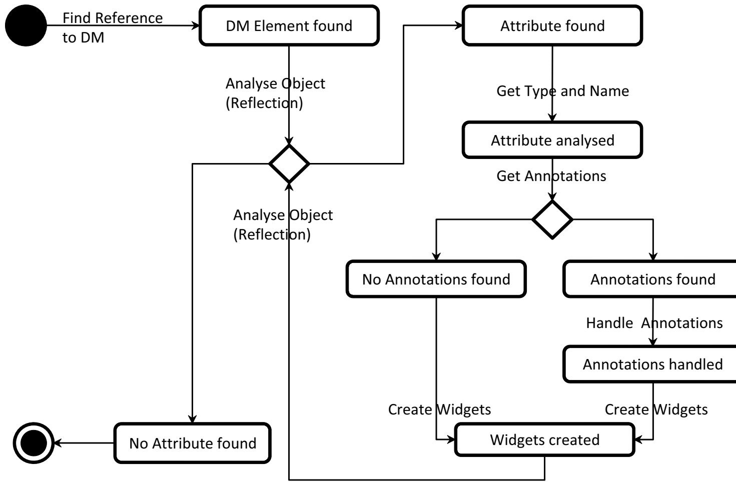

Fig. 8. Activity diagram of the UI generation

3.6 Influencing the Generated UI – Annotations

Annotations are one possible way of influencing the appearance of the generated UI. The annotations are created, stored, and passed to the DM Analyser using the standard EMF annotation mechanism. Each DM element can have its own annotations, which influence the appearance of the resulting UI element. Currently, we support the following set of annotations.

Label. Defines the label for the UI element. Labels are displayed as read-only textfields in front of the UI element.

Group. Defines a group of UI elements that are placed nearby together and separated from other UI elements by a special separator, as can be seen e.g. in Figure 7 for the UI elements grouped under “General” and “Additional Info” in the “Sales Data” section. A group has a unique identifier and a label.

GroupElement. Indicates that the UI element belongs to a certain group by referring to its group identifier.

Ordering. Defines an order in which UI elements should be placed on the UI. The order is given as an index. DM elements, which have no ordering annotation assigned, are placed below all ordered DM elements in the order of their appearance in the EMF model.

Hide. Indicates that no UI element should be generated for this particular DM element.

Default. Defines a default value that will be displayed in a new instance of the corresponding UI element, and be used as value if the application user does not specify another value.

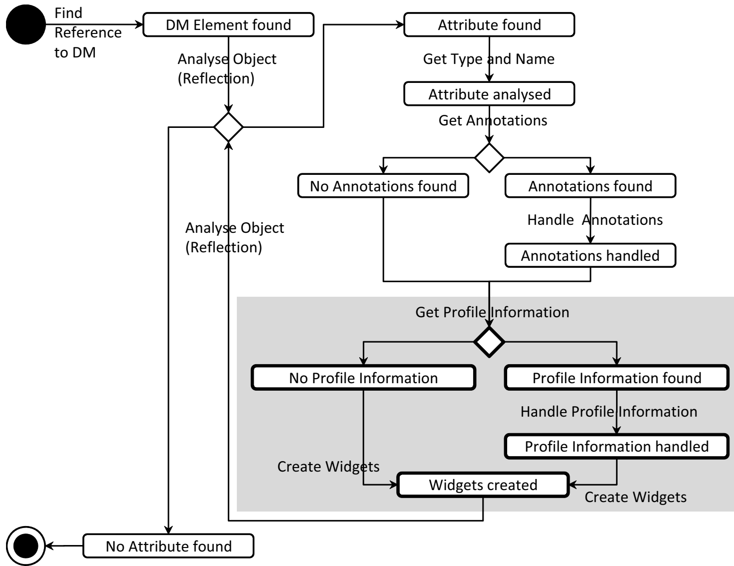

Fig. 9. Analysing DM elements using profile information

3.7 Customising the Resulting UI – The Profile Editor

The methodology described in the previous sections enables the application designer to quickly create and adapt UIs according to the needs of the customer. However, these UIs still have to be customisable on an individual basis, since users might have different roles in the company, and therefore may need different views on the domain data. Creating different UIs for each and every role would require significant effort for both development and maintenance. It is also desirable to have only one annotated instance of the domain model per customer, which eliminates the possibility of adding individual annotations per user. To enable individual customisation nonetheless, we introduced the concept of user profiles. The Profile Editor currently provides the following customisation possibilities to the user: Change layout, hide elements, change labels, and set individual default values.

The profiles influence the behaviour of the UICM Interpreter and the DM Analyser during the UI generation. They contain pieces of information that modify the appearance of UI elements. The effects of the profiles on the UI always overrride contradictory annotations. Figure 9 shows the activity diagram of the UI generation by the DM Analyser when taking profiles into consideration.

The Profile Editor uses a modified version of the Interpreter described in the previous subsection. This Interpreter generates a modified version of the application's UI with additional widgets, such as checkboxes to indicate whether a UI widget should be hidden, or textboxes to change labels. The benefit of that approach is the consistency of the look&feel.

4 Evaluation

In the project period of less than one year we have realised two different real-world scenarios. No extensive user study was available at the time of writing this paper, but the realisation of a larger evaluation involving customers is planned.

Development Costs and Productivity

As a very first evaluation of the productivity of our approach, we re-built a form-based editor for the creation of instances of the Universal Service Description Language (USDL) metamodel[5]. This metamodel consists of 46 classes with a total of 243 attributes. The manual implementation of the editor took 10 person days. It consists of four pages providing CRUD functionality for the elements of the four perspectives of USDL. Creating the same editor with our approach took less than two hours. One fact that speeded up the reimplementation is of course that the layout for the editor's UI was already given by the existing implementation. Taking this into account our approach needs approximately 1/10 of the time for the manual implementation.

Quality

The implementation of graphical UIs is usually the most error-prone part of the application development [9]. These implementation errors are completely eliminated as there is no implementation part in our UI creation process. The structure of our UI is modelled by defining instances of the UICM using a WYSIWYG editor and the atomic UI widgets are generated automatically. However, from a design point of view, it is still possible to create disadvantageous UI layouts, which are misleading or badly structured. To mitigate that, we decided to provide a limited set of container elements in the UICM, so that the designer is also guided in the design choices up to a certain level.

Tool Support

Our approach allowed us to create a powerful tool support for the development of form-based CRUD applications. The UICM Editor enables the creation of applications in a WYSIWYG manner, and the Profile Editor provides an easy-to-use customisation of the final application. Both tools are integrated in a workbench that also runs the final application itself. In this way, the designer as well as a user can switch between the different tools and the final application, and thus immediately see and use the result of the design or customisation step. The tools and the final application use the UICM Interpreter and the DM Analyser as their foundation, which ensures a consistent look&feel of the UI.

Usability of the Final UI

The usability of UIs created with our approach is similar to the usability of manually created UIs for DMs with low to medium complexity. The DM Analyser

[5] http://www.internet-of-services.com/index.php?id=24, last visited May 2, 2010.

generates reasonable UI widgets that improve the usability, such as comboboxes for enumerations, or calendar widgets for dates. Furthermore, the application provides convenience features such as autocompletion, and tab support (i.e. navigating to the next input field using the tabulator key). Of course, the clarity and understandability of the UI depends to a great extent on layout created by the application designer, and the additional information available via annotations during the UI generation, such as grouping and ordering.

For complex DMs with deeply nested or recursive data structures the restricted number of available structuring elements in the UICM currently hinders the designer from creating clear UIs. One possible improvement is the usage of the grouping and ordering annotation to enhance the visual structuring of the generated UI elements. The available annotations as well as the UICM container elements will be further evaluated and optimized.

Consistency

As mentioned in Section 2, roughly 80% of our UI elements are used to edit or display data. The creation of these UIs in a predefined way leads to a consistent look&feel of the different screens and applications. The look&feel of our generated UIs is defined by the different mapping rules applied by the interpreter. Attributes of the same data type are always mapped to the same combination of widgets, e.g., dates are always visualised in a calendar widget. Another benefit of our approach is the consistency of error messages, hints or tooltips, as they are also generated along with the widgets.

Customisability

One main requirement of our approach was that the resulting UI has to be customisable individually, including the renaming and hiding of elements, setting default values or changing the layout of the container structure. Providing these possibilities allows the creation of one UI for a group of users, e.g., the employees of one company, which can later be customized without implementation effort by every user individually. Due to the interpretive approach the changes in the UI are immediately reflected in the final application. The customisation can be done by the user employing an adapted version of the interpreter. The benefit of that is again a good usability and a consistent look&feel, also for the customise view of the resulting application. The changes done by the user are stored in a separate profile and taken into consideration by the interpreter and DM analyser immediately.

Extensibility, Maintainability, and Flexibility

The flexibility of the created UI with regard to changes in the application's requirements or the DM is improved. Small changes in the DM, e.g., adding or deleting attributes or changing their data type do not even require the UI to be modified, since the DM elements will be analysed using reflection every time

the application is run. For displaying new DM elements, the UICM has to be changed, so that it references the new model element. Again, there is no need for code generation or adaptation.

5 Related Work

Work from different fields of research was taken into account for our approach. The CAMELEON reference framework introduced by Calvary et al. [6] is the foundation for several works, e.g., by Vanderdockt [2] and Paternò [10], [11]. It defines several transient models, which are transformed into each other to create the final UI. Concept and task models are used to derive an abstract UI model (AUI), which is independent of any platform and implementation. A concrete UI (CUI) uses this information and turns it into an interactor-dependent UI description. In the last step, the runnable final UI is created.

There are two approaches to create the final UI. According to the methodology proposed in [6], the final UI code should be generated based on the CUI. This strong separation between the modelling and runtime environment however leads to insufficient support of evolution, especially in large-scale systems. The regeneration of complete systems is often not feasible and in some cases even not possible [12]. The second approach is to interpret the CUI instead of generating code. Interpretive model-driven approaches are discussed in [13], [14], and [15]. Meijler et al. [12] give a detailed comparison of generative and interpretive methods and present a hybrid approach supporting fine-grained changes in the model without re-generation of code. The main advantage of interpretation over code generation is flexibility. Because interpreters execute commands line by line, changes in any of these lines are directly reflected in the resulting application. There is no need for regeneration of code or model transformation. This allows for a fast adaptation of the UI to new tasks or focuses. The shortcomings of that are similar to our work, as they are bound to one preferred runtime and cannot generate different runtime platforms from one model according to MDA.

The aspect of automated UI generation is covered by Spillner et al. [16], [17]. The proposed methodology inspired us in developing the concepts of the DM Analyser. They retrieve their data model from WSDL files. The structure of the data model is then used to derive the UI elements from it. The mapping of data types to certain UI widgets is predefined to create highly usable UIs. We developed the annotation model based on the concept of additional UI hints introduced in this work.

6 Summary and Future Work

In this paper we presented our experience in the development of UIs for enterprise applications. The main challenge in this field are the high costs for development, customisation and maintenance. Our approach to reduce development effort combines explicit modelling of the UI's layout structure with automatic generation of the fine grained elements like labels and input fields based on a

given domain model. The models are interpreted at runtime providing an increased flexibility. The interpreter and domain model analyser can be influenced by annotating the domain model at design time or creating custom profiles at runtime. We discussed the lessons learnt in detail. The benefits are a significantly reduced UI development effort, and the elimination of programming errors, since no manual implementation is required. Other enhancements are the consistency of the used UI elements and the flexibility of the resulting application. However, highly complex and deeply nested domain models can currently not be visualized in a satisfactory manner.

For future work we have a number of improvement ideas. The introduction of complex widgets like maps or graphs would improve the usability and efficiency of the UIs for end-users. The modelling of navigational elements has not been discussed in this work, but will be part of future work. Especially for UIs handling large parts of the DM, intuitive navigation between different screens and sections will improve the usability significantly. For very complex DMs, the usability of UIs created with our approach can be improved, as mentioned in the evaluation (Section 4). Different aspects of our methodology have to be reconsidered for that. One way would be the extension of the UICM, providing UI designers more possibilities for structuring the UI. An extended set of annotations could also help to accomplish that goal. Another idea to be further evaluated is the storage of the annotations separate from DM. This enables the provision of multiple annotation sets for the same model, or the separation of different annotation aspects, such as layout (e.g. Group and Order) from language (Label). In this way labels for different languages can be provided in separate files, one for each language, and separated from the language-independent annotations. Finally, the easy and fast adaptation of the UI's appearance and the application of corporate design to all screens of one customer could be enabled by supporting style sheets. For this we will investigate the possibilities provided by the Eclipse e4 project supporting CSS.

References

1. Myers, B.A., Rosson, M.B.: Survey on user interface programming. In: SIGCHI 1992: Human Factory in Computing Systems (1992)
2. Vanderdonckt, J.: A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In: Proceedings of the 16th Conference on Advanced Information Systems Engineering (2005)
3. Sousa, K., Mendonça, H., Vanderdonkt, J.: Towards Method Engineering of Model-Driven User Interface Development. In: Winckler, M., Johnson, H., Palanque, P. (eds.) TAMODIA 2007. LNCS, vol. 4849, pp. 112–125. Springer, Heidelberg (2007)
4. Lu, X., Wan, J.: Model Driven Development of Complex User Interfaces. In: Proceedings of the MoDELS 2007 Workshop on Model Driven Development of Advanced User Interfaces (2007)
5. Ali Fatolahi, S.S.S., Lethbridge, T.C.: Towards a Semi-Automated Model-Driven Method for the Generation of Web-based Application from Use Cases. In: Proceedings of the 4th International Workshop on Model-Driven Web Engineering MDWE (2008)

6. Calvary, G., Coutaz, J., Bouillon, L., Florins, M., Limbourg, Q., Marucci, L., Paternó, F., Santoro, C., Souchon, N., Thevenin, D., Vanderdonckt, J.: The CAMELEON Reference Framework. Technical report (2002)
7. Myers, B., Hudson, S.E., Pausch, R.: Past, present, and future of user interface software tools. In: ACM Transactions on Computer-Human Interaction (TOCHI) (2000)
8. Lehmann, M., Ramil, J.: Evolution in Software and Related Areas. In: 4th International Workshop on Principles of Software Evolution (2001)
9. Mohan, R., Kulkarni, V.: Model Driven Development of Graphical User Interfaces for Enterprise Business Applications – Experience, Lessons Learnt and a Way Forward. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 307–321. Springer, Heidelberg (2009)
10. Paternò, F., Santoro, C., Scorcia, A.: Automatically adapting web sites for mobile access through logical descriptions and dynamic analysis of interaction resources. In: Proceedings of the Working Conference on Advanced Visual Interfaces (2008)
11. Paternò, F., Santoro, C., Spano, L.D.: Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments, vol. 16, pp. 1–30. ACM, New York (2009)
12. Meijler, T.D., Nytun, J.P., Prinz, A., Wortmann, H.: Supporting fine-grained generative model-driven evolution. Software and Systems Modeling (2010)
13. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation 12, 290–321 (2002)
14. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software 50, 36–41 (2003)
15. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. In: 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (2001)
16. Spillner, J., Braun, I., Schill, A.: Flexible Human Service Interfaces. In: Cardoso, J., Cordeiro, J., Filipe, J. (eds.) Proceedings of ICEIS (5), pp. 79–85 (2007)
17. Spillner, J., Feldmann, M., Braun, I., Springer, T., Schill, A.: Ad-Hoc Usage of Web Services with Dynvoker. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 208–219. Springer, Heidelberg (2008)