Proceedings of the 25th IASTED International Multi-Conference
SOFTWARE ENGINEERING
February 13–15, 2007, Innsbruck, Austria
ISBN Hardcopy: 978-0-88986-641-6 / CD: 978-0-88986-643-0

A PRACTICAL APPROACH TO TASK-DIALOG AND ABSTRACT PRESENTATION MODELING OF GRAPHICAL USER INTERFACES

Stefan Römer, Johannes Mayer, Franz Schweiggert
Institute of Applied Information Processing
Ulm University
D–89069 Ulm, Germany
email: {stefan.roemer,johannes.mayer,franz.schweiggert}@uni-ulm.de

Martina Maier, Tobias Vollmer
eXXcellent solutions gmbh
Beim Alten Fritz 2
D–89075 Ulm, Germany
email: {m.maier,t.vollmer}@exxcellent.de

ABSTRACT
While Model Driven Architecture (MDA) has already been adopted by companies for the development of business and persistence layers, there has been no general adoption of a model-driven approach for graphical user interface (GUI) development. Therefore, the present paper describes a practical and pragmatic approach for task-dialog and abstract presentation modeling of GUIs. The presented model has already been used to generate user interfaces. It will be shown how UML activity diagrams can be extended in order to enable automated code genertation from the model—with some manual intervention (with a GUI builder to layout the GUI components). A sample application will be described in order to show how the presented model can be used.

KEY WORDS
Graphical User Interface, Task-Dialog Model, Abstract Presentation Model, Model Driven Architecture

1 Introduction

Graphical user interfaces (GUI) usually consist of many windows, dialogs, wizards, etc. which are interconnected and are usually nested over several levels. In order to give developers a better overview over the GUI, a visual modeling approach is required. Furthermore, productivity improvements can only be achieved by an upward shift in the level of abstraction in which problems are solved [9] and some kind of automation. Visual modeling languages can satisfy these requirements (overview and automation) if they operate on a higher level of abstraction than the code [9] and contain enough semantics of the GUI in order to enable automated code generation.

Visual modeling languages are currently used mainly for modeling certain parts of an application, including persistence layer, business objects, and business logic. The standard notation for modeling these aspects of an application is the Unified Modeling Language (UML) [11]. There are also many CASE tools and IDEs that support the UML. However, UML only covers part of an application. Up to now, other parts of an application, such as the GUI, are not or rarely modeled in practice. Especially, task modeling, dialog modeling, and presentation modeling of GUIs are not yet supported by the UML—the quasi-standard in model-driven [10] and model-based software development. There are neither suitable UML diagram types nor UML profiles. If we want to close this "modeling gap" and come to a practically applicable solution, we have to extend the UML. Then we can also use tools available for the UML and extend them. Otherwise, it would be necessary to develop custom tools—which is not an approach viable in industry. Thus, it will be necessary that our solution is based on UML. Thereby, we also get two additional advantages for free: If we use the UML with only slight modifications, it will not be difficult for the average developer to understand our models. Furthermore, our UML modeling approach for GUIs can be used to construct a unified, integrated model-driven development approach, which will in each layer (presentation, business, and persistance) be based on the UML (without trying to model everything which is not yet supported by the UML and by the currently available tools).

A suitable practical modeling language should thus fulfill the following requirements:

- It should be suitable for modeling GUIs with many top-level UI components like windows, dialogs, wizards, views, etc.
- It should offer a good overview of the tasks that can be performed with the GUI.
- It should be simple and intuitive to avoid a long period of training.
- It should be practical, i. e. there should be tool support.
- It should be possible to link the application model with the model for the GUI.
- Modeling should be supported at several levels of detail.

The following section presents a new approach for task-dialog and abstract presentation modeling based on UML activity diagrams. In the Section 3, a case study with the introduced modeling language is presented. Section 4 discusses related work. Finally, Section 5 concludes the present paper.

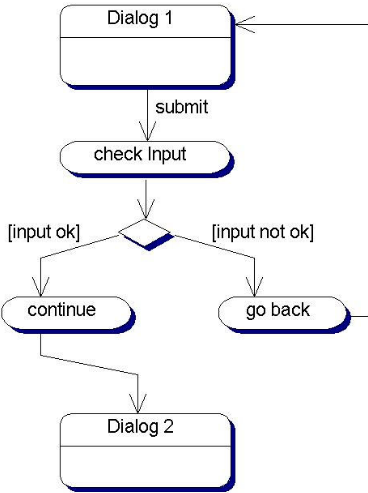

Figure 1. A first task-dialog model of a GUI with an UML activity diagram

2 A Practical Task-Dialog and Abstract Presentation Modeling Approach with UML Activity Diagrams

In order to enable linking of the application model with the GUI model as well as having tool support, UML should be used as a basis. There are many CASE Tools for UML and it has already been adopted as a quasi-standard modeling language in industry. These tools often provide mechanisms for extending the UML.

In the following, we take a look at UML 1.4 activity diagrams[1] and find out whether they are suitable for task-dialog and abstract presentation models of GUIs.

We could model the dialogs of an application with states. As soon as an event occurs, states may be left or the status of a state is changed. The dialogs of an application react similarily. The user "fires" events by interacting with the GUI and reaches other dialogs or the current dialog is changed. To get an overview of a GUI, especially the events which lead to other dialogs are important.

However, the same events do not always lead to the same following dialogs. Often inputs have to be processed by the application in order to determine which dialog has to be displayed next. In consequence, activity diagrams should offer possibilities to model conditional branches. We can use decisions and transitions with conditions of UML activity diagrams for this purpose. Further elements provided by activity diagrams are activities. They can be used for modeling actions and instructions.

Now we can make a first approach to model a GUI. Figure 1 shows a small example. The modeled GUI consists of two dialogs: 'Dialog 1' and 'Dialog 2'. In 'Dialog 1', the user has obviously the option to trigger an event called 'submit'—persumably, he can click on a button which results in the event 'submit'. After the inputs have been checked, the GUI displays 'Dialog 2' if the inputs were correct or goes back to 'Dialog 1' otherwise.

[1] Our approach should be practical which requires full tool support. This was only available for UML 1.4 at the time, this approach was developed.

Figure 2. The login dialog

At present, the sample model could be used to do task-dialog modeling for a GUI. However, since it is not clear what e. g. a state should be, a GUI cannot yet be automatically generated from this description. Therefore, it will be necessary to extend UML activity diagrams in order to describe specific parts and features of GUIs, in order to enable task-dialog and abstract presentation modeling being automatically transformed into code.

Tables 1 and 2 show the (extended) set of elements of UML activity diagrams which are necessary for modeling tasks, dialogs, and abstract presentation of GUI. These enhancements were achieved by adding stereotypes and new properties.[2]

3 Case Study

In the following it will demonstrated, how a simple application could be modeled with the modeling language presented so far. In the first part of this section the specification of the modeled application are specified. In the second part it is shown how these requirements could be modeled in three diagrams.

3.1 Specification of the Modeled Application

In the following, the functionality of the modeled sample application will be described. The focus of this description is on the possible interactions on the GUI—and not on the functionalities "behind" the GUI.

Before the main application will be displayed, a login dialog (cf. Figure 2) should appear. This dialog should not be closeable by clicking ☒ and has two text fields to enter login name and password. As usual, the login dialog has 'OK' and 'cancel' buttons. If a user confirms by clicking 'OK', the password for the specified login is retrieved from the data base and compared with the entered password. If the login does not exist or the password does not match, a

[2] This comes close to an UML light weight extension.

Figure 3. A warning message displayed after entering a wrong login or password

Figure 4. The main application (back) and a dialog of one dialog sequence (front)

warning will be displayed (cf. Figure 3) and the login dialog should not be closed in this case. If the 'OK' button is clicked and login and password match, the main application window displayed in full mode. Optionally, 'cancel' could be clicked. In this case the login dialog is closed, but the main application should continue in a restricted mode.

In the main application window (cf. Figure 4) shown after leaving the login dialog, the following tasks can be performed:

- Create new users
- Search for users
- Edit users
- Delete users

For each of these four tasks there should be a sequence of dialogs. Hereby we always want to have the ability to leave the current dialog by clicking 'close' or ☒. Dialogs which enable modifying user data should additionally have a 'Save' button.

3.2 Modeling of the sample application

Figure 5 shows the model for the login dialog, containing the action 'login' referenced from the model of the main application. 'login' is the Action that shows the Login view

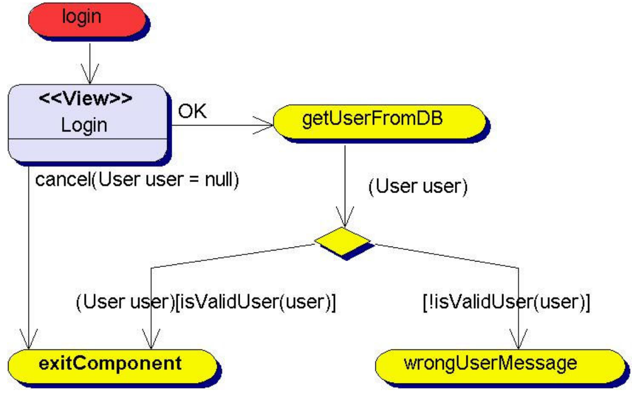

Figure 5. The model for the login dialog

and is called from the main application via '*login'. The login dialog has two events ('OK' and 'cancel') not being special events. That means they will be interpreted as buttons (by the code generator).

After clicking on the 'OK' button, the logic 'getUserFromDB' is triggered. In the method generated for 'getUserFromDB' (which is implemented by the developer), the data from the text fields login name and password are acquired (from the GUI) and the corresponding data is retrieved from the data base (it is the binding to the business/persistence layer). In the model, it is denoted that this logic produces a User object. The Conditions following the Decision have to check if 'user' is valid or not. Because the generator isn't able to resolve these Conditions they are treated as complex conditions[3]. Then the login dialog returns 'user'. If we leave the dialog via 'cancel' the instance 'user' is explicit set to 'null' before being returned. After the 'wrongUserMessage', however, the login dialog is still active.

Figure 6 shows the model for the main application. First we create a Component with the stereotype 'Application' and name it 'ApplicationExample'. Each Component fires an 'init' special event before it is displayed. This is the right place to insert the login dialog. But we decide not to model the login dialog in this diagram, because we want to keep it small and clear. Therefore we use an ActionLink called '*login', which links to an Action in another diagram (cf. Figure 5). We only define the return type as 'User' and the name of the instance returned as 'user'. After returing from '*login', it will be checked if the returned User instance represents an administrator (abstract condition) or a user not being logged in (simple condition).

The Logics 'initAdmin' and 'initNotLoggedIn' are modeled to have methods generated, in which the developer can implement own code. These model elements serve as placeholders in the diagram for later manual insertion of code.

[3]The generator generates an abstract method `abstract boolean isValidUser(User user)` in the implementation.

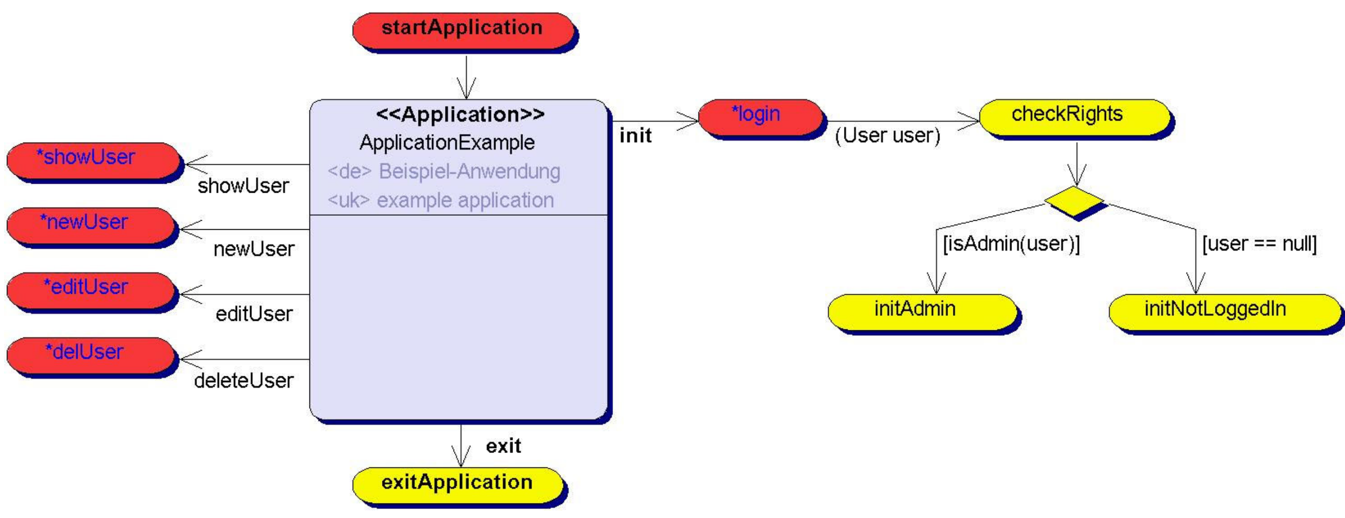

Figure 6. The model for the main application

In order to enable that the application window can be closed by clicking on ☒, the special event 'exit' is used with the special logic 'exitApplication'.

The only things missing now are the four events triggering the four dialog sequences. These four events will cause buttons being generated within the main application window. To keep the diagram small we used ActionLinks again.

The model for the four dialog sequences is shown in Figure 7. The dialog sequences can be activated by calling the Actions 'newUser', 'showUser', 'delUser', and 'editUser', respectively. They are called only from the main application (cf. Figure 6). There are altogether eight dialogs involved in the four dialog sequences. However, there are only three different dialogs (namely 'searchUser', 'showUser', and 'editUser'). Therefore, we can take advantage of reusing Components as well as of extending Components for modeling. The common behavior of the views has thus only to be modeled once and can be reused and extended each time it is required. When reused, events 'next', 'back', and 'delete' are added.

3.3 From the Model to the Code

After the generation step, the complex logics has still to be implemented manually. In our example, we had to manually implement the methods `boolean isAdmin(User user)` (cf. Figure 6) and `boolean isValidUser(User user)` (cf. Figure 5).

What is still left is to construct the concrete GUI layout—our modeling approach only specifies the abstract presentation, i. e. the elements of the GUI. The layout can be done manually or with a GUI design tool. The layout of the login dialog in our example (cf. Figure 2) was constructed with a GUI designer.

The screenshots in Figures 2–4 show the generated GUI of the application. The red marked area shows the place in which manual GUI design should take place (cf. Figure 4).

4 Discussion and Related Work

A good overview of different approaches for modeling of graphical user interfaces is provided by the survey papers [19, 13, 6].

UMLi ([18, 14, 16, 12, 17]) is a modeling approach for GUIs based on the UML. One aim of UMLi is to be a minimal extension of the UML notation to support detailed modeling of GUIs. Neither the semantics of existing UML elements, nor the UML meta-model is modified. Therefore the UMLi notation only uses UML extension mechanisms. The basic idea of the UMLi approach is that user interface designers and application designers should work within a single environment and that the UI designers should do their work in the presentation model. Our approach however is intended to provide better overview because only the high-level perspective of a GUI is modeled (i. e. task-dialog modeling and abstract presentation modeling), and the concrete presentation layout is constructed in a GUI design tool after generation. Our approach seems to be more convenient for GUI designers. There are some more differences to our approach. UMLi has introduced one new diagram in UML, called the user interface diagram, to model the presentation model. This user interface diagram is composed of one 'FreeContainer' that is a top-level window, 'Containers', 'Editors', 'Displayers', 'Inputters', and various diagram elements ('Widgets'). The structure of the Widgets is modeled in a user interface diagram and its behavior is modeled in an activity diagram, which represents the task-dialog model. Whereas activity diagrams are used in UMLi mainly to model the logics within a dialog, they are used in our approach to model the control flow between

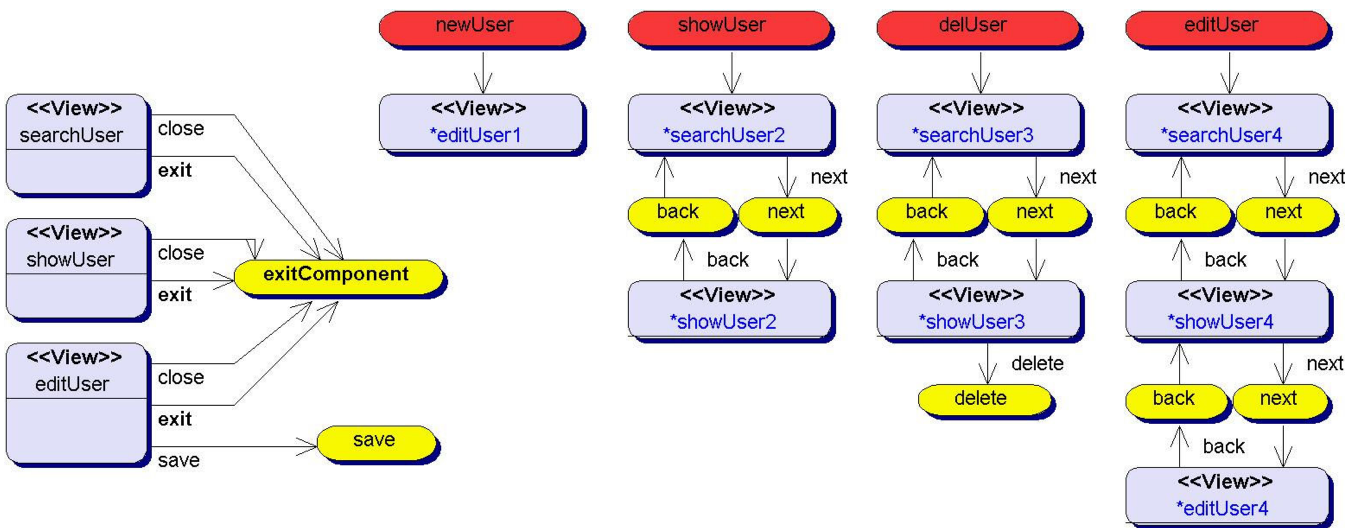

Figure 7. The model for the dialog sequences

dialogs, which is not really modeled by UMLi. Thus, our model is more suitable to get an overview over the GUI flow. Furthermore, UMLi introduces many new elements and stereotypes to UML activity diagrams which makes them difficult to understand. Our approach uses activity diagrams only with some custom stereotypes. This makes it easy for developers familiar with the UML to understand our models. Finally, the UMLi activity diagrams are partially too close to the implementation (and contain method calls). In our approach, activity diagrams only contain business logic that is relevant for the dialog flow. Thus, there are lots of differences between our approach and UMLi. Our approach certainly gives a better overview, is not that close to the implementation details, and requires only slight changes to UML activity diagrams. Altogether, our approach is clearly preferable to UMLi given our objectives as mentioned at the beginning.

TADEUS (TAsk-based DEvelopment of USer interface software) [20, 5] provides four explicit declarative models: the task, domain, user, and dialog model. Tasks are modeled in a hierarchic tree structure. The presentation model called dialog model is divided into a navigation dialog model which describes the possible interactions between dialog views, and a processing dialog model which deals with the description of the processing within a dialog view. Both of these models are based on petri nets. The navigation dialog model is similar to the presented modeling approach, however it is based on petri nets. Thus, the application and the presentation cannot be modeled in the same environment and TADEUS requires custom graphical editors.

Teallach [7, 15] and TRIDENT [2][1] allow to model even details for concrete UI design and have therefore a concrete presentation model. Since Teallach and TRIDENT are not based on UML, the binding to an application model in UML is difficult. Furthermore, custom design tools are required for both approaches.

Book et al. [3] introduced the Dialog Flow Notation (DFN) for the hierarchical description of the dialog flow of web applications. This approach is specifically tailored to the development of (multi-channel) web applications. The DFN notation is not based on the UML. However, diagrams similar to UML activity diagrams resp. state charts are used.

There are also new and completely different concepts for GUI modeling as for example NakedObjects [8, 4]. It is an object oriented approach. NakedObjects is an object browser which generates generic user interfaces at runtime and is well suited for rapid prototyping, but it is not usable for expressing workflow oriented applications. In addition, the produced UIs are hardly adaptable and it is questionable whether such GUIs are usable by end-users.

Many of the modeling approaches for GUIs are not based on UML. Therefore, integration into standard tools is not available and also not easily possible. For this reason, a practical approach has to be based on the UML in order to also have standard tools. A further reason is that in this case modeling of several layers can be done in one modeling language and environment. This will enable a more general model-driven approach than it is usual nowadays, where mostly the business layer and the persistence layer is modeled.

It is possible in our approach that a developer can create a model which makes no sense for the generator. This is caused by the fact, that we had not introduced a completely new modeling language, but extended an existing one. We could solve this problem by creating an own language using (for example) a meta modeling language like the MOF (Meta Object Facility). But up to now, there are no CASE tools which support the MOF. Another solution could be

using a new breed of tools called Meta Case Tools. They give the ability to develop own problem specific modeling languages [9]. A pragmatic solution to the above problem could be to develop a software that guides the developer and forbids semantically incorrect models, such that there can be no model that contains two “Application” components.

5 Conclusion

A practical and pragmatic modeling approach for task-dialog and abstract presentation modeling of graphical user interfaces has been introduced in the present paper. The main goals of this approach are simple tool integration and good overview over the modeled GUI. Therefore, the UML was used, especially activity diagrams have been adapted. New stereotypes and new properties have been introduced in order to add sufficient semantics to UML activity diagrams such that code can be generated automatically from the model. Since our modeling approach does not cover concrete presentation, in order to achieve a better overview, a GUI designer tool has to be used with the generated code in order to layout the GUI components. Furthermore, specific controller logics has also to be implemented manually. The generator only generates empty methods for this purpose. If using sub-classing, the generated code has not to be changed but extended. This enables us to generate the GUI once again without detroying the hand-written code. Our approach has already been implemented and integrated within a common IDE. After some tweaking with the custom configuration language, the integration was not problem. Therefore, the presented approach is really practically and can be used to provide a model-driven development environment that is not only able to the model business and persistence, but also the presentation layer.

References

[1] F. Bodart, A. Hennebert, J. Leheureux, I. Provot, and J. Vanderdonckt. A model based approach to presentation: A continuum from task analysis to prototype. In *Proceedings of 1st Eurographics Workshop on Design Specification and Verification of Interactive System (DSVIS 1994)*, pages 25–39. Eurographics, Carrara, Italy, 1994.

[2] F. Bodart, A. Hennebert, J. Leheureux, and J. Vanderdonckt. Computer-aided window identification in trident. In *Proceedings of the 5 IFIP TC13 Conf. on Human Computer Interaction Interact 1995*, pages 331–336. Chapman and Hall, London, 1995.

[3] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 100–109. IEEE Computer Society Press.

[4] B. Coyner. Introduction to naked objects. *java.net*, July 2003. http://today.java.net/pub/a/today/2003/07/15/naked-address.html.

[5] T. Elwert. Continuous and explicit dialogue modelling. In *Proceedings Conference on Human Factors in Computing Systems*, pages 265–266. ACM Press, New York, 1996.

[6] M. Gomaa, A. Salah, and S. Rahman. Towards a better model based user interface development environment: A comprehensive survey, 2005. http://www.cs.uwec.edu/mics/.

[7] T. Griffiths, P. Barclay, J. McKirdy, N. Paton, P. Gray, J. Kennedy, R. Cooper, C. Goble, A. West, and M. Smyth. Teallach: A model-based user interface development environment for object databases. In *Proceedings of UIDIS'99*, pages 86–96. IEEE Press, Edinburgh, UK, September 1999.

[8] A. Haase. Nackt und pur: Naked objects - objekte pur. *JavaMagazin*, 5(5), 2004. http://www.javamagazin.de/itr/online-artikel/psecom,id,543,nodeid,11.html.

[9] M. Iseger. Platform-based development on a high programming abstraction level, September 2004. http://www.embedded-control-europe.com/pdf/ecesep04p30.pdf.

[10] S. J. Mellor, A. N. Clark, and T. Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, September/October 2003.

[11] Object Management Group. Unified Modeling Language (UML) specification, 2004. http://www.omg.org/.

[12] P. Pinheiro da Silva. *Object Modelling of Interactive Systems: The UMLi Approach.* PhD thesis, University of Manchester, 2002. http://www.cs.utep.edu/paulo/publications.html.

[13] P. Pinheiro da Silva. User interface declarative models and development environments: A survey. In *Interactive Systems: Design, Specification, and Verification (7th International Workshop DSV-IS)*. Limerick, Ireland, June 2000. http://www.cs.utep.edu/paulo/publications.html.

[14] P. Pinheiro da Silva. Umli: Integrating user interface and application design. In *Electronic proceedings of the UML2000 Workshop on Towards a UML Profile for Interactive Systems Development (TUPIS2000)*. York, United Kingdom, October 2000. http://www.cs.utep.edu/paulo/publications.html.

[15] P. Pinheiro da Silva, T. Griffiths, and N. W. Paton. Generating user interface code in a model

based user interface development environment. In L. T. V. Gesu, S. Levialdi, editor, *Proceedings of the International Working Conference on Advance Visual Interfaces2000 (AVI2000)*, pages 155–160. ACM Press, Palermo, Italy, June 2000. http://www.cs.utep.edu/paulo/publications.html.

[16] P. Pinheiro da Silva and N. W. Paton. A uml-based design environment for interactive applications. In *Proceedings of the 2nd International Workshop on User Interfaces to Data Intensive Systems (UIDIS'01)*. Zurich, Switzerland, 2001. http://www.cs.utep.edu/paulo/publications.html.

[17] P. Pinheiro da Silva and N. W. Paton. Improving uml support for user interface design: A metric assessment of umli. In *Proceedings of ICSE-2003 Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction*. Portland, USA, 2003. http://www.cs.utep.edu/paulo/publications.html.

[18] P. Pinheiro da Silva and N. W. Paton. Umli: The unified modeling language for interactive applications. In *UML2000 - The Unified Modeling Language: Advancing the Standard (3rd International Conference on the Unified Modeling Language)*. York, United Kingdom, October 2000. http://www.cs.utep.edu/paulo/publications.html.

[19] E. Schlungbaum. Model-based user interface software tools - current state of declarative models. Technical report, Visualization and Usability Center, Georgia Institute of Technology, Atlanta, 1996. http://www.gvu.gatech.edu/gvu/reports/1996/abstracts/96-30.html.

[20] E. Schlungbaum and T. Elwert. Dialogue graphs - a formal and visual specification technique for dialogue modeling. In *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface of OZCHI 2000*. Springer, London, 1996.

Table 1. Modeling elements used for task-dialog and abstract presentation modeling of GUIs (part 1)

Modeling Element	Syntax and Semantic Descrition	Possible Generator Implementation
<<Component>> Name	A **Component** represents a general GUI object. A Component can be used as placeholder.	All Components are realized as own classes. A generator cannot generate code for a Component unless a stereotype is used.
<<Component>> *Name	An **ExtendedComponent** is linked to another Component. The ExtendedComponent inherits the behavior of the linked Component. Furthermore, own behavior can be modeled for an ExtendedComponent and the inherited behavior can be "overwritten".	This idea of inheritance can be implemented by the generator in the form of type and implementation inheritance of classes.
<<Application>> Name	The stereotype of this Component is **Application**. An Application represents the main application. This stereotype has to be used exactly once in a project.	For this stereotype a main application window will be generated.
<<Menu>> Name	A **Menu** is usually nested. Its sub-components must have the same stereotype. With Menu it is possible to model menus. The order of Menus can be specified through ComponentOrders.	Menus can be generated in various forms: main menus, pop-down menus, context menus, navigation bars, etc.
<<Group>> Name	With **Group** any kind of Components (except that with stereotype 'Application') can be grouped together.	A Group could be implemented as a tabbed panel with an own panel for each element or a desktop with individual windows for each element. The possibilites highly depend on the target platform.
<<View>> Name	A **View** represents a simple dialog and that cannot have sub-components.	A modal or non-modal dialog is generated—depending on how it is invoked.
<<Wizard>> Name	A **Wizard** contains 'Steps' and is used to model dialog sequences, which are displayed in linear order.	Usually, wizards are not natively supported by a lot of target platforms. Therefore, the wizard and its steps have to be built from dialogs and panels by the generator.
<<Step>> Name	A **Step** is comparable with a View, but is exclusively used as a sub-component of 'Wizard'. The order of Steps is specified by ComponentOrders.	A Step could be implemented by a panel and being dynamically exchanged in a Wizard at runtime.
	A **ComponentOrder** specifies the order of Components. It is mainly used to define a sequence of Steps and Menus.	Each Component is associated with an index according to the ComponentOrder. Based on these indices, the order of the components within the GUI can be determined.
Event	An **Event** can only be triggered by a Component. It represents an event triggered by the user via interaction with the GUI, e. g. by clicking on a button.	A button can be generated if the event is not a special event.
init	An **'init' Event** is triggered when a Component is entered. This happens before the Component is displayed.	This special event is triggered by the framework of the target platform.

Table 2. Modeling elements used for task-dialog and abstract presentation modeling of GUIs (part 2)

Modeling Element	Syntax and Semantic Descrition	Possible Generator Implementation
exit	An **'exit' Event** is triggered when a user closes a Component (e. g. with the button).	This special event is triggered by the framework of the target platform.
Action	An **Action** displays a graphical component (i. e. a Component). Actions can also be called from other diagrams via ActionLinks.	All Actions are realized as derived classes of an action class and started with an action method.
*Action	An **ActionLink** links to an Action. It allows to call the same Action from various diagrams. An ActionLink can be called modal or not modal (cf. Fork).	An ActionLink cannot be extended. The linked Action is simply instantiated and started via the action method.
startApplication	**startApplication** is a special action (denoted by the bold text). **startApplication** has to be used exactly once per project and has a Component with the stereotype Application as successor. The framework of the target platform calls this special action when the application is started.	This action is performed when the application is started.
Logic	A **Logic** describes a task and represents a controller element. It always belongs to the previous Component. If a Logic has no successor the control flow is directed back to the associated Component, which again waits for events to occur.	The generator implements a method in the class of the associated Component.
exitApplication	**exitApplication** is a special logic (denoted by bold text style). If called, the application is ended.	A method is generated which exits the application.
exitComponent	**exitComponent** is a special logic (denoted by bold text style). If called, the associated Component is closed and parameters are returned.	A method which closes the current Component and returns parameters (if any) will be generated.
Null	A **NullActivity** can be used as a placeholder.	The generator adds comments to the code.
(Argument)	An **Argument** declares object flows, i. e. data passed along Transitions between the model elements (not modeled as in UML activity diagrams). Assignments are allowed, too. Object flows can be combined with Events and Conditions.	The output has to match the input of the following element.
	A **Decision** has exactly one incoming Transition and at least one outgoing Condition.	If constructs can be generated from Decisions and associated Conditions.
[Condition]	A **Condition** leads to conditional invocation of its successor if the condition (simple or abstract) is true. If there is more than one outgoing Condition for a Decision, only one must be true.	The generation of simple conditions (e. g., `user == null`) is straightforward. Abstract conditions are implemented by abstract methods (e. g., `abstract boolean isAdmin(User user)`).
	With **Fork**s, parallelism can be modeled. If a Fork precedes an Action or an ActionLink, the following Component is invoked non-modal.	