Easy Model-Driven Development of Multimedia User Interfaces with GuiBuilder

Stefan Sauer and Gregor Engels

s-lab – Software Quality Lab
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
{sauer,engels}@s-lab.upb.de

Abstract. GUI builder tools are widely used in practice to develop the user interface of software systems. Typically they are visual programming tools that support direct-manipulative assembling of the user interface components. We have developed the tool GuiBuilder which follows a model-driven approach to the development of graphical (multimedia) user interfaces. This allows a meta-design approach where user interface developers as well as prospective users of the system are supported in modelling the desired functionality of the GUI on a high level of abstraction that is easy to understand for all involved stakeholders. The model consists of compositional presentation diagrams to model the structure of the user interface and hierarchical statechart diagrams to model its behaviour. GuiBuilder then supports the transformation of the model to Java, i.e., the generation of a working user interface and the simulation of the modelled behaviour. Interactive sessions with the user interface can be recorded and replayed.

Keywords: Model-driven development, meta-design, user interface, prototype generation, capture-replay.

1 Introduction

Recently, *meta-design* has been proposed as a novel approach to system development where end users play an active role not only in using a software system but also in designing it. In [2], G. Fischer et al. state: "Meta-design characterizes objectives, techniques, and processes for creating new media and environments allowing 'owners of problems' (that is, end users) to act as designers. A fundamental objective of meta-design is to create socio-technical environments that empower users to engage actively in the continuous development of systems rather than being restricted to the use of existing systems."

In [1], M.F. Costabile et al. refine this approach and introduce the notion of "Software Shaping Workshops (SSW)", where groups of stakeholders focus on certain aspects of system development. They state: "We view meta-design as a technique, which provides the stakeholders in the design team with suitable languages

C. Stephanidis (Ed.): Universal Access in HCI, Part I, HCII 2007, LNCS 4554, pp. 537–546, 2007.

and tools to favour their personal and common reasoning about […].” Futhermore, they follow G. Fischer’s arguments, who characterizes end users as persons who want to be a “consumer” (i.e., user) of a software system in some situations, and in others a “designer”, who adapts the software system to her personal needs and desires.

In our approach, we exemplify these ideas by presenting a model-based development approach for graphical user interfaces (GUI). The overall idea is to provide high-level sophisticated design languages and tools, which allow end users to be involved in designing and testing graphical user interfaces of a software system.

Following the approach of model-driven development (MDD) techniques [4,9], such a platform-independent model of a GUI is automatically transformed into an executable GUI realisation in a common programming language like Java.

Graphical user interfaces of (multimedia) software applications provide users with the presentation of information and interaction capabilities with (media) content and functionality. The user interface is a complex part of the overall system and often requires software engineering effort comparable to building the application functionality itself. In addition, the user interface has to meet the user’s requirements and expectations in order to yield a high acceptance rate by future users. Thus, user interface development should be done cooperatively by software engineers and prospective end users. Due to the inherent complexity of user interfaces, model-based development processes which are nowadays well-accepted in software development should be applied for user interfaces, too. GUI builder tools that merely support visual programming of the user interface are overstrained with this task.

Model-based development of user interfaces promotes structuring of the resulting implementation and allows developers and prospective users in teamwork to prevent errors or to detect errors earlier and more easily by already analysing the model of the user interface. The models can also be used as documentation and for guiding the maintenance of the software system. *Model-driven* development even goes a step further by automatically generating from the model an executable user interface in a common programming language like Java.

The objective of this work is to develop a model-driven and tool-based development technique for graphical user interfaces (GUI). The model of the GUI combines structural and behavioural aspects. The model-driven development of the GUI is then supported by a tool called GuiBuilder. GuiBuilder provides developers and prospective users with an editor for GUI modelling and an execution environment for GUI simulation. A prototype user interface can be generated from the model, executed and tested. External tools can also connect to the simulation and are notified about the simulation progress. Simulation runs can be recorded and replayed. The simulation logs can also be used to support regression testing based on the capture-replay paradigm.

A number of model-based approaches have been proposed in past years to deal with user interface modelling at different levels of abstraction (see e.g. [10]). GuiBuilder is targeted towards concrete user interface modelling. The idea of combining statechart and presentation diagrams originally stems from the OMMMA approach [8]. Statecharts have also been used in [3] for describing GUI behaviour. UsiXML (e.g. [11]) uses graph transformations instead. It provides a variety of GUI elements which are currently not completely supported by GuiBuilder due to its early development state. In MOBI-D [7] the process of constructing a GUI is guided and

restricted by domain and task definitions, which are the building blocks of user interfaces in MOBI-D. A UML-based approach towards model-driven development of multimedia user interfaces is described in [5]. Recently, model-driven development of user interfaces has attracted wider interest in the research community [6].

In the next section, we will introduce the different models that are supported by GuiBuilder and their interplay. Section 3 presents the tool GuiBuilder. We draw conclusions and outline future perspectives in Section 4.

2 Models of GuiBuilder

The model of the multimedia user interface in GuiBuilder consists of two parts: the presentation model and the dynamics model. The *presentation model* captures the structure and layout of the user interface, the *dynamics model* uses UML statecharts to specify the behaviour of the GUI. Dynamic behaviour is enacted by user interaction or other events that cause a change of state in the user interface (and the application). Events that are caused by user interaction are modelled as signals which can be handled by the presentation elements. Signals can also be sent as the actions of triggered state transitions.

The basic concept of the compound model is to assign a presentation design to a state, which describes the structure and layout of the user interface while in that state. At any point in time, the GUI of an application is in a specific, possibly complex state. An event occurrence causes a state change and thus a change of the presentation.

The presentation model consists of presentation elements (see Fig. 1). Typically they are graphical elements that are part of the application's presentation. Such elements can e.g. be geometric shapes, widgets, or graphics elements for rendering images or video. In addition to graphical elements, audio elements can be included for playing music or sound effects. The presentation elements have properties which can be assigned with values. The properties depend on the type of presentation element and determine the presentation of the element. The types of presentation elements are organized in a class hierarchy.

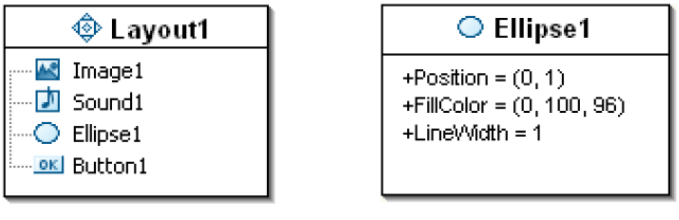

Fig. 1. A presentation consists of presentation elements (left), where each presentation element is characterized by its property values (right)

The presentation elements within one presentation diagram are ordered. The topmost element is upfront and possibly covers parts of other elements if they overlap.

If the GUI is in a simple state, the presentation is a composition of presentation elements with their property values. The presentation is completely described by the presentation diagram that is assigned to this state.

However, it is also possible to assign presentation diagrams to complex states in our model. Complex states allow us to hierarchically structure the state of the user interface. The actual presentation is then composed from the presentation diagrams that are assigned to the current simple state and all its parent states, where the complex states can even be concurrent (i.e., AND-superstates). Fig. 2 shows an example, where the presentation diagrams Layout1 and Layout2 are assigned to State1 and its substate State2, respectively.

The actual composition of the presentation is determined by the hierarchical structure of the statechart diagram. If the behaviour of a superstate is refined by substates, the assigned presentation is also refined by the presentation diagrams that are assigned to the respective substates.

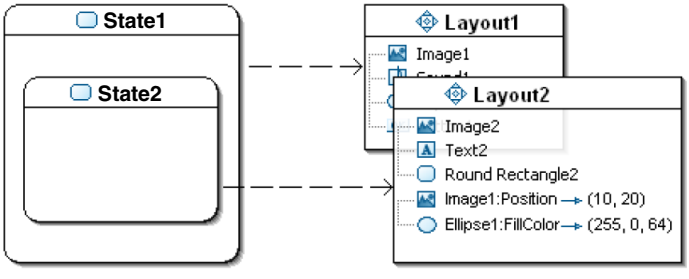

Fig. 2. Presentation diagrams can be assigned to hierarchical states, new presentation elements can be added for substates or properties of existing elements be modified

The composition of the presentation diagrams according to the state hierarchy works as follows:

First, presentation diagrams are stacked on top of each other. The order is determined by the state hierarchy: presentation diagrams of substates are put on top of presentation diagrams of their superstates. The former are intended to be the more specific. Their presentation elements override (cover) the presentation elements of the latter. For concurrent states, an order is not defined.

Secondly, since presentation diagrams can not only contain new presentation elements, but also property changes (i.e., modify the properties of presentation elements contained in presentation diagrams that are assigned to superstates), the modified value also overrides the ‘inherited’ value. All presentation elements that are introduced in presentation diagrams of the superstates of a state can be altered by modifying their property values. A property change thus specifies the modification of a property value of an inherited presentation element in a substate (see Fig. 3).

Consequently, a hierarchical presentation can be interpreted as a list of modifications where the instantiation of a presentation element is a specific case. This list can then be processed to construct and compose the actual presentation for a particular state: for each presentation diagram, the list of modifications is processed,

whereby the order of the lists of different presentation diagrams is determined by the hierarchical state structure from superstates to substates.

With respect to execution semantics, this means that when a state is left, the modifications of its presentation diagram to the user interface become ineffective and are replaced by the modifications of the presentation diagram of the successively entered state. Modifications of the presentation diagram of the possibly still active superstates remain unaffected, yet may be overridden.

Structured specification of a user interface is facilitated by this composition mechanism. GUIs typically contain a limited number of fundamentally different views which are then subject to a larger number of smaller (local) modifications for representing the particularities of different states within the overreaching context. Our incremental composition mechanism eases the specification of such modifications and prevents the developer from having to specify the complete presentation design for each, even simple modification. The GUI design thus requires less effort and the GUI models become easier to extend and modify even by end users, especially since redundancy is limited and controlled.

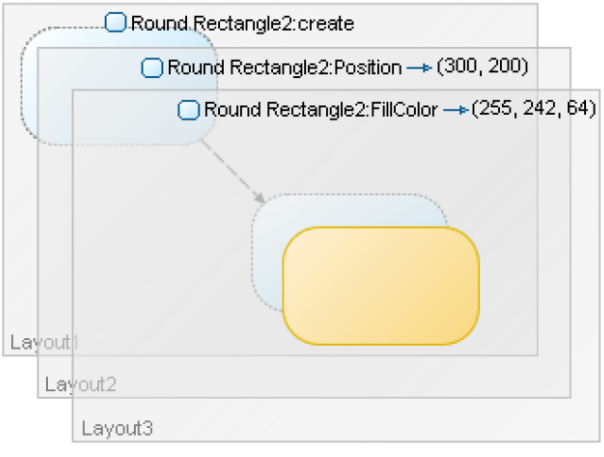

Fig. 3. Stepwise modification of a presentation element through hierarchical presentation diagrams

In addition to presentation, interaction also profits from the incremental specification. User interaction results in events which are received by presentation elements as signals. Since signals are properties of the presentation elements as well, they can be 'inherited' and modified like presentation properties. Functionality can thus be adapted in the same way by modifying the signal specification.

Thus, since we follow a clearly structured approach toward user interface construction and limit the GUI modelling language to a selected number of modelling concepts and elements, it is suited for professional software developers and end users as well. The integration of end users in the software development tasks is further promoted by the strict distinction of interactive control behaviour that is modelled here and possibly complex algorithmic computations of the system that are developed separately.

3 GuiBuilder - The Tool

GuiBuilder has been developed as a plug-in of the Eclipse tool environment and platform. We used the Plug-in Development Toolkit PDT for its implementation and the Graphical Editor Framework GEF for implementing the graphical editor of the GuiBuilder plug-in.

GuiBuilder supports user interface software developers as well as prospective users in the development of graphical (multimedia) user interfaces. Audio and video can be integrated in the presentation of the application that is developed. In the current version of GuiBuilder, executable GUIs are generated from the model and executed using Java SWT, and the Java Media API is deployed for rendering of multimedia artefacts.

The main view of GuiBuilder is the GUI editor. Additional views of GuiBuilder are the Eclipse standard views problems view, outline view, and properties view as well as the presentation view. The problems view lists the detected errors and warnings. The outline view presents an outline page for each window of the GUI editor when it is selected. The properties view shows properties of a selected model element (statechart element or presentation element). Properties can be edited directly in the properties view or in an explicit properties dialog. The presentation preview is a GuiBuilder-specific view that presents a preview of the presentation (see Fig. 6). In our development of GuiBuilder we tried to keep the tool as simple as possible—despite its diverse functionality—to be usable even by end users.

The tight integration of editing and simulation tools allows users to dynamically switch between the roles of software developers who design the structure and behaviour of the interactive graphical user interface by the use of design models and users who interact with the application that is being designed.

3.1 Editor

The editor of GuiBuilder is a graphical tool that supports the direct-manipulative construction of dynamics and presentation diagrams. The GUI editor is a multi-page editor that can manage windows of two different types for statecharts and presentation layouts, respectively.

3.2 Model Validation

The GUI editor calls the validator to validate the correctness of the edited model. The diagrams of the dynamics model have to be valid UML statechart diagrams where only a limited subset of modelling elements is used to control the model's complexity. In addition, we require that the specified behaviour is deterministic. Thus, the statechart diagrams are validated before code is generated from them by the generator function and the simulation can be started. To effectively support the developer as well as prospective user, we provide syntax-directed editing to prevent from fundamental syntactic errors and static model analysis to detect more complex and context-sensitive problems. For example, missing start events or non-deterministic transitions are identified by our model analysis. Two categories of problems, errors and warnings are recorded and presented to the user of the editor in the central

problems view, in the outline view (see Fig. 4), and directly at the relevant modelling elements in the editor view. The identified problems are accompanied by correction procedures (i.e., quick fixes). These features are altogether intended to support users of the editor as much as possible in detecting and correcting defects. Only after all errors have been resolved, the generation can be enacted. Warnings need not to be resolved; however, they should not be ignored since they mark weaknesses of concept or style within the model. Thus, the static analysis supports both the syntactic correctness of the model and its quality in accordance to modelling guidelines.

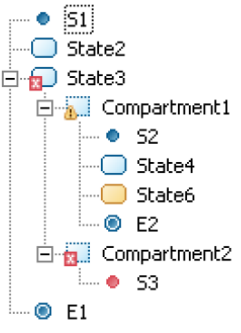

Fig. 4. Problems are marked at the causing model elements

Since the static analysis is the powerful core of the validation module, the syntax-directed editing restrictions can be kept low, not to unnecessarily hinder the flexibility of model editing. For example, inconsistencies or incorrectness can be temporarily tolerated as long as the developer does not want to start the prototype generation process.

Despite the wide range of checks in the static analysis, some problems can still only be detected during dynamic analysis. Dynamic analysis is integrated with the simulation and executed at runtime. Dynamic errors that are detected then are for example infinite loops or non-deterministic behaviour. Such errors cause the termination of the simulation run.

3.3 Generation and Simulation

The simulator view shows the simulated GUI. The GUI editor passes the GUI model via a generator function to the simulator view. The generator function flexibly implements the transformation rules to build a prototype GUI from the GUI model. It can be replaced for generating a different target language or for tailoring of the generation results.

The simulator view starts the simulation in the simulator and registers the GUI editor with the simulator. The simulator then notifies the GUI editor about state changes.

Simulation of the user interface is accomplished by interpreting the model. The GUI simulator uses a statechart simulator which interprets the statecharts of the dynamics model. Connected objects are notified by the statechart simulator about

state transitions and triggered actions (signals). The GUI simulator constructs the composite presentation view for the active state configuration and passes it to the simulator view of GuiBuilder. The simulator view renders the current GUI view. The user can then start the simulation in the simulation view, and the simulator executes the generated GUI. Events can be raised either by interacting with the simulated GUI elements directly or by using the 'remote control' that we implemented as an external plug-in. It can be operated remotely to generate the required signals.

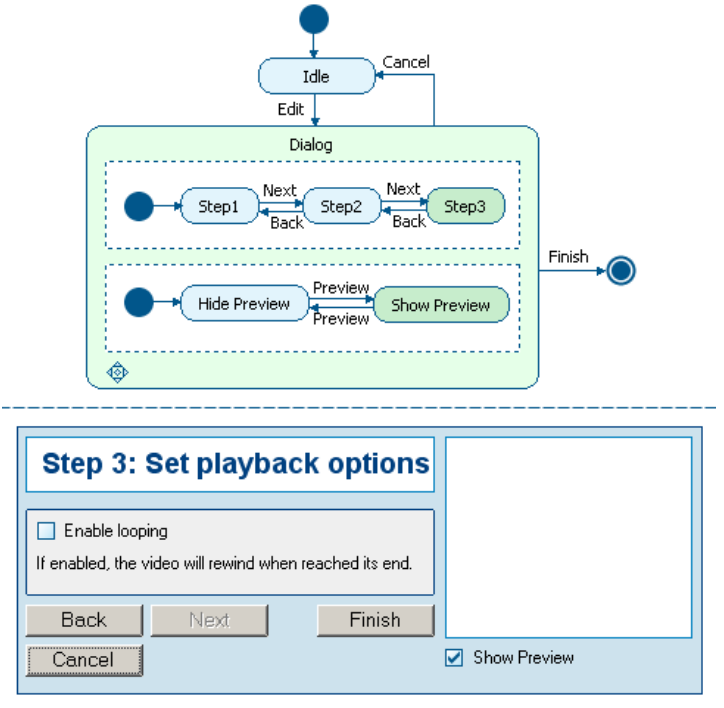

Fig. 5. Simulations can be tracked in the model

The interpretative approach has the advantage that the user interface model can be altered at runtime, and these changes can directly influence the succeeding simulation behaviour. GuiBuilder provides this functionality in a separate hot-code replacement mode.

External tools and other Eclipse plug-ins can connect to the simulator and are thus notified about state changes in the simulated model. They can assign specific actions to the signals and specifically respond to their occurrence. With this mechanism it is possible to actually control a fully fledged application. Besides, the simulation recorder that we developed uses this mechanism for recording a simulation run. The recorder logs the simulation execution. The recorded log can later be used to replay the simulation or to do regression testing after the GUI model has been modified.

External tools can themselves send signals to the simulation and raise events to change the state of the simulated GUI. Thus, the GUI can react to application or external events, too.

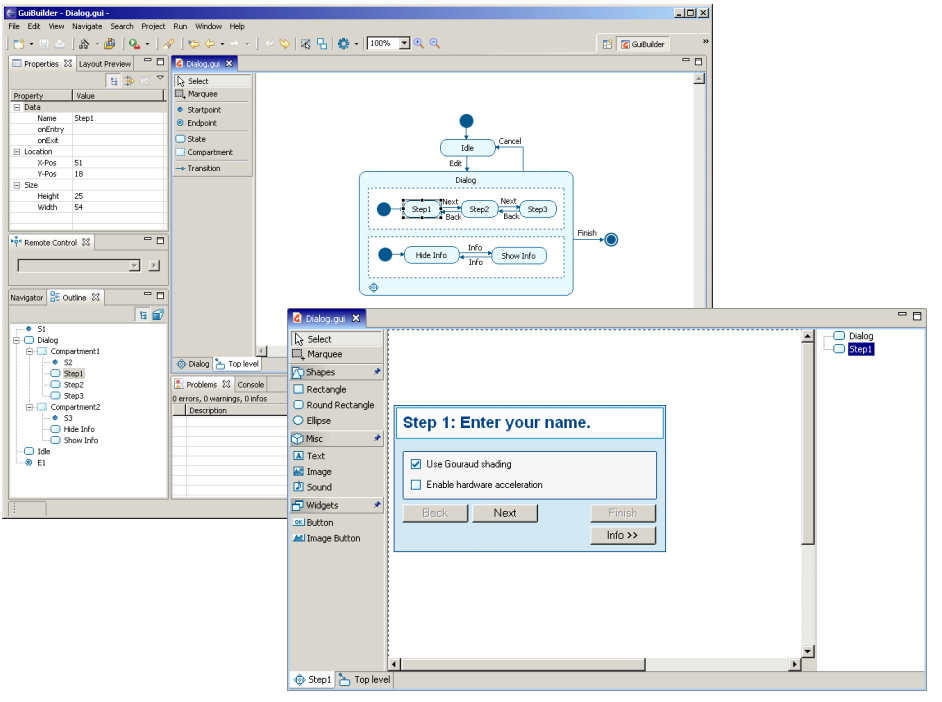

Fig. 6. The GUI of GuiBuilder

While a simulation is running, the editor view of GuiBuilder highlights the current state of the dynamics model in its statechart in green colour (composite state “Dialog” and its concurrent substates “Step 3“ and “Show Preview” in the example of Fig. 5). Thus, dynamic information is fed back into the model representation and can be used e.g. for model debugging.

4 Conclusions

We have integrated the model-driven development paradigm with the GUI-builder tool concept. This provides user interface developers as well as prospective end users with a tool for constructing graphical (multimedia) user interfaces in practice. The GUI model consists of presentation and dynamics models from which a prototype user interface can be generated and simulated.

In a next step, we plan to further improve the capabilities of multimedia processing by extending the dynamic model to deal with timed procedural behaviour. We also want to demonstrate the flexibility of the transformation approach by tailoring the generator function to different target representations.

We have evaluated GuiBuilder in several workshops with high-school students and people who are interested in software development, but not professional software developers or programmers. After a presentation of the tool of about half an hour they were capable of using the tool for constructing, changing and simulating simple applications like a traffic light control with only very limited support by our tutors. Thus, the tool has shown its capability to support end users with little programming skills in building and simulating interactive graphical user interfaces.

Additional information about GuiBuilder can be found at http://www.s-lab.upb.de/Tools/GuiBuilder/

Acknowledgments. The authors are indebted to the three computer science students Dennis Hannwacker, Marcus Dürksen, and Alexander Gebel, who contributed to the concepts of GuiBuilder and implemented the tool.

References

[1] Costabile, M.F., Fogli, D., Mussio, P., Piccinno, A.: A Meta-Design Approach to End-User Development. In: Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), pp. 308–310. IEEE Comp. Soc, Washington (2005)

[2] Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A.G., Mehandjiev, N.: Meta-Design: A Manifesto for End-User Development. CACM 47(9), 33–37 (2004)

[3] Horrocks, I.: Constructing the User Interface with Statecharts. Addison-Wesley, London (1999)

[4] Mellor, S.J., Scott, K., Uhl, A.: MDA Distilled: Principles of model-driven architecture. Addison-Wesley Professional, London (2004)

[5] Pleuß, A.: Modeling the User Interface of Multimedia Applications. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 676–690. Springer, Heidelberg (2005)

[6] Pleuß, A., Van den Bergh, J., Hußmann, H., Sauer, S (eds.).: MDDAUI '05. In: Proc. of the MoDELS'05 Workshop on Model Driven Development of Advanced User Interfaces, CEUR Workshop Proc. 159. CEUR-WS.org, (2005)

[7] Puerta, A.R.: A Model-Based Interface Development Environment. IEEE Software 14(4), 41–47 (1997)

[8] Sauer, S., Engels, G.: UML-based Behavior Specification of Interactive Multimedia Applications. In: Proc. IEEE Symposium on Human-Centric Computing Languages and Environments (HCC'01), pp. 248–255. IEEE Comp. Soc, Washington (2001)

[9] Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20(5), 19–25 (2003)

[10] van Harmelen, M. (ed.): Object Modeling and User Interface Design. Addison-Wesley, London (2001)

[11] Vanderdonckt, J., Limbourg, Q., Michotte, B., Bouillon, L., Trevisan, D., Florins, M.: UsiXML: a User Interface Description Language for Specifying Multimodal User Interfaces. In: Proc. W3C Workshop on Multimodal Interaction WMI'2004 (2004), http://www.usixml.org