The Guilet Dialog Model and Dialog Core for Graphical User Interfaces

Jürgen Rückert and Barbara Paech

Institute of Computer Science, University of Heidelberg, Germany
{rueckert,paech}@uni-heidelberg.de
http://www-swe.informatik.uni-heidelberg.de

Abstract. Model-based approaches to graphical user interfaces (GUIs) achieved poor acceptance of software engineers because the offer models, architectures, components, frameworks and libraries that restrict the flexibility of development too much. We propose a dialog model which enables flexible development with no restrictions on presentation and application layer and without any implementation-technology dependence. The dialog model supports GUI designers and developers in understanding the behavior of the GUI. The dialog model controls the dialog core component. The dialog component relieves GUI developers of re-implementing the coordination of presentation and application layer.

Key words: Model-based user interfaces. Dialog models. Dialog cores. UI engines

1 Introduction

Model-based approaches to graphical user interfaces (GUIs) achieved poor acceptance of software engineers because they restrict the flexibility of development too much. They rarely offer models, architectures, components, frameworks and libraries that can fully be adapted to customer needs [11]:

- P1 GUI designers are not able to describe the presentation that usability engineers defined (in mock-ups). For instance, the approach does not support the modeling of complex graphical components which would be necessary in order to guarantee the usable GUI.
- P2 Software architects are not able to integrate existing application layers and their application services into the GUI. For instance, the approach does not consider the different application service technologies and their corresponding different integration mechanisms.
- P3 GUI Developers are not able to transfer a GUI to another platform as the behavior is hidden in the platform-specific implementation parts and hardly changeable. In this case it is hardly possible to re-use components that are responsible for controlling the GUI. For instance, the approach does not allow for a desktop application to be transferred to a PDA application by splitting few large screens into many small screens and does not allow adding a wizard-like behavior to walk through the small screens.

P4 GUI Developers use development tools that are not integrated with the modeling tools of the designers which easily results in implementations that no longer reflect the actual design.

We propose an approach to develop GUIs that puts dialog modeling in the center of design and implementation. The *Guilet Dialog Model* (*GDM*) (1.) allows designers to model the behavior of the GUI graphically (P1, P4) and (2.) allows developers to realize the presentation and application layer using any implementation technologies (P2) by identifying abstract behavioural building blocks (P3), namely the *Guilets*. An optional reusable *Guilet Dialog Core* (*GDC*) component that is controlled by the *GDM* (3.) relieves developers of re-implementing the coordination of presentation and application layer without restricting the GUI architecture too much (P2) and (4.) allows developers to transfer (P3) the GUI between applications of a specific platform (e.g. inside the Java platform between Java Swing, Java Web and Eclipse Rich Client applications).

Section 2 defines major functional and non-functional requirements of dialog core models. Section 3 presents an easy to understand example that shows the graphical notation of the *GDM* (3.1). Afterwards, the modeling elements of the *GDM* are explained (3.2). Section 4 presents first experiences gathered in an in-house and in a commercial project. Section 5 summarizes the article and gives an outlook.

2 Requirements of Dialog Core Models

The major functional requirements for dialog cores models can be retrieved from the articles on GUI architectures like the Model-View-Control pattern, the Presentation-Abstraction-Control pattern [5], the Arch model [10] and OpenQuasar [14]. The requirements focus on the coordination of presentation and application layer: A dialog core should be able (F1) to create and destroy graphical components (like views) and their sub-components (like widgets), (F2) to create and maintain the communication channels with application services, and finally (F3) to process events that are created by users in the presentation layer or by application services in the application layer. Processing events encloses (F3.1) sending events to views and widgets in order to change their status, (F3.2) sending data to views or retrieving data from views, and (F3.3) call application services and interpret the results or exceptions. Usually, the event processing is specific to each event source (e.g. graphical component) and each event type (e.g. click, focus).

The major non-functional requirements for dialog cores models are outlined in Figure 1. We detailed the ISO/IEC 9126 quality categories (at the top) by requirements that we elicited from the literature on model-based UI approaches [8] [1] [13] [15] [3] [12] [16] [6] [9] [17] [2] [4]. These requirements are software requirements but not end-user requirements because dialog core models are artefacts that are used hidden inside the GUI.

The quality attribute set *Functionality* describes in how far the DM implements the demanded functionality (see above). The quality attribute set *Reliability* describes in how far the DM is able to model a certain level of performance under defined conditions for a stated period of time. The quality attribute set *Usability* describes the designer's effort of creating and manipulating the DM. The quality attribute set *Maintainability*

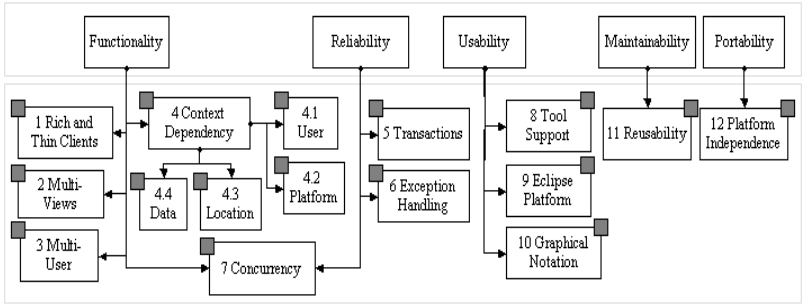

Fig. 1. Non-functional Requirements for Dialog Core Models (gray boxes at the left mark conceptual requirements, gray boxes at the right mark engineering requirements)

describes the effort needed to make necessary changes in the DM. The quality attribute set *Portability* describes the ability of the DM to be transferred from on environment to another.

1 Rich and Thin Clients: The DM should be reusable for stand-alone applications as well as Web applications.
2 Multi-Views: The DM should be able to handle multiple views (e.g. panels), that are visible at the same time and are part of a screen (e.g. a frame or a web page).
3 Multi-User: The DM should be able to model the influence of access rights of users and roles on the GUI behavior.
4 Context Dependency: The DM should be able to model the dependency between page structure, page flow and inserted data, user, computing platform and work environment.
5 Transactions: The DM should support the modeling of two transactions types: transactions during the period of processing multiple events and during processing single events.
6 Exception Handling: The DM should allow modeling expected exceptions during event processing.
7 Concurrency: The DM should support modeling concurrent event processing.
8 Tool Support: The DM should be maintainable with a domain-specific (dialog) modeling tool to shorten design time.
9 Eclipse Platform: The DM should be maintainable on the Eclipse platform because of the high acceptance and usage experience of software developers and the seamless integration of design and implementation.
10 Graphical Notation: The DM should be graphically editable instead of textually (including XML) because this ensures faster understandability.
11 Reusability: The DM should not constrain the usage of implementation technologies for presentation and application layer technologies.
12 Platform Independence: The DM should not contain any presentation and application layer specific information in order to be reusable for a variety of applications in (Web, desktop and mobile) or between platforms (Sun Java, Microsoft .NET).

3 Guilets

3.1 Application example

In this section we introduce an application example that does not illustrate all of the solution ideas of the *GDM* but instead is easy to understand (transactions are left out e.g.). Figure 2 shows the flow of events between presentation and application layer and the coordination of the flow by the *GDC*. Figure 3 shows a screen shot of the view *Lecture*

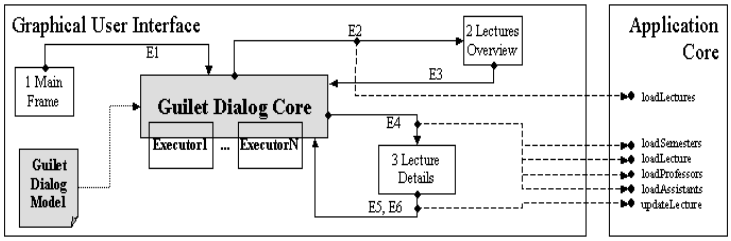

Fig. 2. The *GDC* as central component controlling the behavior of a GUI

Details of our in-house application for administrating students and lectures. Figure 4

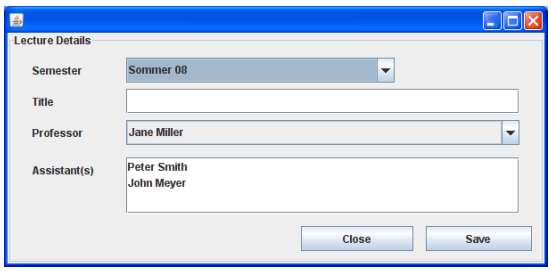

Fig. 3. Screen shot of the view *Lecture Details*

shows the *GDM* of the view *Lecture Details* that describes the view's behavior. The user starts the application (*Main Frame*), retrieves a list of lectures (*Lectures Overview*) and requests the details of a certain lecture by sending event E3 to the *GDC*. The *GDC* reacts on E3 by triggering event E4 (*ShowAndInitialize*). As shown in Figure 4, the *GDC* invokes 4 executors in parallel. 3 executors query lists of business objects from application services and forward the lists into these 3 inner *Guilets* that are able to handle lists of data (e.g. combo boxes or multi line fields). The *LoadLecture* executor reads the variable *LectureId* (circle at the left) that contains the ID of the selected lecture, queries the appropriate lecture data from an application service (a property defines the connection reference) and forwards this data to all 4 inner *Guilets*. The inner *Guilets*

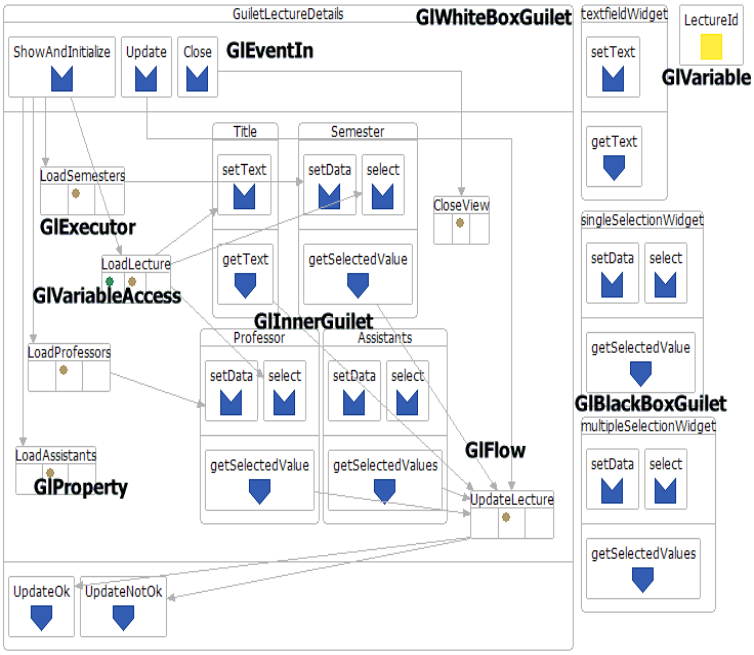

Fig. 4. *GDM* of the view *Lecture Details*

are either of type *textfield*, *singleSelection* or *multipleSelection* as the inner *Guilet* type denotes (not shown). The user triggers the update of the modified data by sending event E5 (*Update*) to the *GDC*. The *GDC* invokes the executor *UpdateLecture* that first reads the data of the 4 inner *Guilets*, then calls an application service and finally sends either the event *UpdateOk* or *UpdateNotOk* that may be used for refreshing other views that are interested in an update (would be modeled as `GlTransition`). The user closes the view *Lecture Details* by sending event E6 (*Close*) to the *GDC*. The *GDC* invokes the executor *CloseView* that invokes a suitable GUI service.

3.2 Guilet Dialog Model

The *GDM* is based on the major elements `GlWhiteBoxGuilet`, `GlBlackBoxGuilet`, `GlEventIn`, `GlEventOut`, `GlExecutor`, `GlInnerGuilet` and `GlFlow`.

Designers model a `GlWhiteBoxGuilet` whenever they want to model the behavior of a 2d-container like a view, partial view or widget. Widgets are graphical components that receive or provide data and very often allow user input. Partial views are the smallest composition units of logically related widgets, their size is often determined by reuse. Views are a composition of partial views and may contain additional widgets themselves. The hierarchical structure of views needs not to be fixed, the enclosed partial views/widgets and their amount (of recurrence) may depend on the context of

usage. The simplest case e.g. is a view that is not shown until a certain data value was inserted/selected in another view. **Designers** model a `GlInnerGuilet` whenever they want to add a view, partial view or widget to a whitebox *Guilet*. An inner *Guilet* enables reuse because it is either of type `GlWhiteBoxGuilet` or `GlBlackBoxGuilet`. It just layouts incoming and outgoing events, but never behavior, in order to avoid redundant information layout. Designers use a `GlBlackBoxGuilet` whenever they want to model a view, partial view or a widget but are not interested in modeling its behavior (of e.g. a complex widget of a fixed graphical library).

Designers model a `GlEventIn` or a `GlEventOut` whenever they want a start- or endpoint for a certain processing logic. They only need to model events when they require a processing logic that needs implementation. They do not need to model events that are processed automatically by the presentation or application layer. For instance, they do not need to model the event *sort rows* if the table widget is already capable of sorting. This decision for the *GDM* was made in order to reduce the amount of modeled elements. Events are created by users or by application cores or by the *GDC*. Typical events are the initialization of a view and the call of semantical functions (e.g. *save data*).

Designers model a `GlExecutor` whenever they require a behavioral component during processing an event. The implemented executors call application services, in order to query data or invoke semantical functions, or they call GUI services in order to change the status of graphical components. **Designers** model a `GlVariableAccess` (referencing a `GlVariable`) whenever they want to store the output of an executor or want to use a stored value as input of an executor. Variables have a scope of validity property.

Designers model a `GlFlow` whenever they want to link an event with an executor or an executor with an event, or an executor with a inner guilet or an inner guilet with an executor. A flow calls several executors always concurrently because sequential executors can be merged into one executor. A flow cannot split into two flows by a condition element because we do not want to overload the model with too much detailed information. The conditional cases have to be implemented in the executor, the documentation property of the executor serves to forward this information from designer to developer. A special case of a flow is a `GlTransition` which is an event-to-event call between two *Guilets*.

Designers add one or more `GlProperty` to any of the modeling elements above whenever they want to enrich the elements with information that they need for processing. Usually, executors use properties for the configuration of application services connections.

4 Experiences

We applied the *GDM* and the *GDC* to develop several desktop applications: (1) an in-house data management application for students and lectures and (2) an application for a pick list creation which is an add-on application for an existing commercial fashion logistic solution. The specifications consist of a task model and a domain data model, virtual windows [7] (task-based mock-ups), system functions and a state chart diagram

(page flow). The virtual windows and the state chart diagram are a well-suited starting point to design *Guilets*. The systems were realized as client-server systems. The clients are implemented in Java Swing and include a *GDC* in Java. The *GDC* was reused in both projects. The presentations are loosely coupled with Java Web Services by the *GDC*, more precisely by the executor implementations. The *GDC* is driven by the *GDM* which was modeled using the *GMT*. During these two projects we were able to check the fulfillment of some requirements, as follows. **Functionality:** (+) Multi-views are fully supported. (+) Multi-users can be modeled by using event properties (edited in XML directly). **Reliability** (+) GUI Transactions (useful e.g. for wizards) are supported by tagging the flow elements with transaction IDs (edited in XML directly). (+-) Several exceptions of an executor can be modeled. The reason for an exception cannot be modeled, the reason is only accessible in the executor implementation. (+) The assumption of parallel execution of executors as default is acceptable because sequential executors can be merged into one. **Usability:** (+) The *GDM* definitely should be expressed in a graphical notation and the *GMT* must remain a substantial part of the design because it is very hard to ensure a semantically correct XML using pure text or XML editors. (+-) On the one hand, the *GDM* exempts from too many details because of missing conditional elements. On the other hand, the executor hides the conditional information in its implementation. (-) A graphical modeling support for transaction (e.g. path high-lighting) might be very useful for immediate visualisation of a transaction and its flows. **Maintainability:** (+) The executor elements can be mapped to well maintainable code structures that easily can be understood even weeks later. We expect that executors additionally allow collaborative, parallel implementation by several developers and a transparent tracing of implementation progess. (-) Depending on the level of the modeled presentation details, the *GDM* tends to become very large. We learned that *Guilets* should not be used to model the presentation hierarchy as a whole, but should model instead only these event-sending presentation parts that require an event processing by the *GDM*. (+) The integrated modeling and implementation in Eclipse allowed incremental development of the GUI. (+) *Guilets* made development fast because the hard-to-implement part of coordinating the presentation and application layer is available as the out-of-the-box component *GDC*. (-+) 100% reuse of the modeled elements seems to be rare because usually the properties of same-named *Guilet* sub-elements (e.g. properties of executors) of two *Guilets* differ. Despite, the amount of reuse of executor implementations is high. **Portability:** (+) The blackbox *Guilets* serve as a nice mechanism to model widget libraries and can easily be reused in other *Guilets*.

5 Summary and Outlook

In this article we introduced a design approach to describe the behavior of graphical user interfaces. Designers create a *Guilet Dialog Model* in a graphical notation using the *Guilet Modeling Tool*. Developers apply the *Guilet Dialog Model* as a feed for a reusable *Guilet Dialog Core* component that controls the presentation and application layer using implemented, partially generated *executor components*. In the future, we will, due to encouraging project realizations, continue the evaluation of *Guilets* in order to evaluate the missing requirements in the area of context dependency.

References

1. P. Barclay, T. Griffiths, J. McKirdy, N. Paton, R. Cooper, and J. Kennedy. The Teallach Tool: Using Models for Flexible User Interface Design. In *CADUI*, pages 139–158. Kluwer, 1999.
2. M. Brambilla, S. Comai, P. Fraternali, and M. Matera. *Designing Web Applications with WebML and WebRatio*. Springer, October 2007.
3. T. Browne, D. Davila, S. Rugaber, and K. Stirewalt. The Mastermind User Interface Generation Project. GVU Technical Report GIT-GVU-96-31, Georgia Institute of Technology, 1996.
4. S. Comai and G. T. Carughi. A behavioral model for rich internet applications. In *ICWE*, pages 364–369, 2007.
5. J. Coutaz. PAC: An object oriented model for dialog design. In H.-J. Bullinger and B. Shakel, editors, *Human-Computer Interaction: INTERACT'87*, pages 431–436. North-Holland, Amsterdam, 1987.
6. ISO. Lotos - a formal description technique based on temporal ordering of observational behaviour (ISO 8807). Technical report, Information Processing Systems - Open Systems Interconnection, 1989.
7. S. Lauesen. *User Interface Design. A Software Engineering Perspective*. Addison-Wesley, 2004.
8. F. Lonczewski and S. Schreiber. The FUSE-System: an Integrated User Interface Design Environment. In *CADUI*, pages 37–56, 1996.
9. F. J. Martinez-Ruiz, J. M. Arteaga, J. Vanderdonckt, J. M. Gonzalez-Calleros, and R. Mendoza. A first draft of a model-driven method for designing graphical user interfaces of rich internet applications. In *LA-WEB '06: Proceedings of the Fourth Latin American Web Congress*, pages 32–38, Washington, DC, USA, 2006. IEEE Computer Society.
10. D. Navarre, P. Palanque, P. Dragicevic, and R. Bastide. An approach integrating two complementary model-based environments for the construction of multimodal interactive applications. *Interact. Comput.*, 18(5):910–941, 2006.
11. F. Paternò and S. Sansone. Model-based Generation of Interactive Digital TV Applications. In *MoDELS'06, Workshop on Model Driven Development of Advanced User Interfaces. Genova, Italy*, 2006.
12. A. R. Puerta. The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In *CADUI*, pages 19–36, 1996.
13. G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, editors. *Web Engineering: Modelling and Implementing Web Applications*, volume 12 of *Human-Computer Interaction Series*, pages 263–301. Springer, 2008.
14. J. Siedersleben. *Moderne Software-Architektur*. Dpunkt, 2004.
15. P. A. Szekely, P. N. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. Declarative interface models for user interface construction tools: the MASTERMIND approach. In *EHCI*, pages 120–150, 1995.
16. J. Vanderdonckt and et al. User Interface Extensible Markup Language (UsiXML) 1.8. Université catholique de Louvain, February 2007. http://www.usixml.org.
17. J. Vanderdonckt, D. Grolaux, P. V. Roy, Q. Limbourg, B. M. Macq, and B. Michel. A design space for context-sensitive user interfaces. In *IASSE*, pages 207–214, 2005.