A Dialog Control Framework for Hypertext-Based Applications

Matthias Book
mb@matthiasbook.de
Lehrstuhl für Software-Technologie
Universität Dortmund

Betreuer: Dipl.-Inform. Malte Hülder, Lehrstuhl für Software-Technologie, Universität Dortmund
Art der Arbeit: Diplomarbeit
Fachbereich innerhalb der GI: Software-Technologie (SWT)

Abstract Summary

This thesis introduces a notation for specifying the dialog flow of hypertext-based applications, i.e. the succession of hypertext pages displayed on the client and application logic operations performed on the server. A key concept is the encapsulation of multiple dialog steps in context-independent dialog modules, which can be nested arbitrarily. The notation is accompanied by a framework that implements a dialog control logic capable of handling multiple presentation channels, leaving only the tasks of implementing the business logic, designing the user interface and specifying the dialog flow to the application developer.

1. Motivation: Need for a Dialog Control Framework

Since the introduction of the World Wide Web about twelve years ago, the services that web sites provide to users have continuously become more sophisticated: From presenting static information, sites have evolved to offer simple one-step interactions (e.g. search engines), then multi-step transactions (e.g. shopping carts), and now complex applications (e.g. e-mail readers, customer support systems, portal systems), where the user interface (UI) consists of web pages presented in a browser. Compared to traditional window-based UIs, hypertext-based UIs require only modest client capabilities, making them suitable for a wide variety of client platforms, especially thin clients such as mobile devices with their strict energy, memory, input and output limitations. Furthermore, the simple information elements and interaction techniques of hypertext-based UIs can easily be ported to various presentation channels, meeting the increasing demand of users to be able to work with applications very flexibly – ideally, any service should be available on any device, anywhere, anytime. With reasonable effort, this requirement can virtually only be fulfilled by employing the thin client principle, where the application logic is implemented presentation channel-independently on a central server, while the UI is rendered individually on various client devices.

However, when developing applications with hypertext-based UIs, software engineers need to be aware that their implementation differs in some important characteristics from applications with window-based UIs [RF96]: Firstly, different presentation channels have different input and output capabilities (e.g. regarding screen size), possibly restricting the amount of information users can work with at a time. Consequently, presentation channel-independent applications must not only implement different UIs, but also be able to handle different interaction patterns. Secondly, hypertext-based UIs present information on pages instead of in windows. Consequently, interactions that would be performed without involving the application logic in a window-based UI (e.g. closing a window) require the generation of a new page in a page-based UI and thus involve the application logic for every interaction step. Thirdly, hypertext-based UIs employ a request-response mechanism to pull data from the server. Since the application logic cannot push data to the client, it can only react passively to user actions (e.g. clicking on a link) instead of actively initiating dialog steps (e.g. opening a new window). Finally, HTTP is stateless: The protocol only transports data, but does not maintain any information on the state of the dialog system. Consequently, the application itself has to manage the dialog state for each user session, which requires complicated logic for more complex dialog structures.

Regarding the impact of these characteristics on the user experience, one of the most notable effects is the limitation to simple dialog structures in many hypertext-based applications today: Linear and branched dialog sequences can be easily implemented and are therefore commonplace, but already simple nested structures (e.g. an authorization form inserted at the beginning of a sensitive transaction, leading the user to different pages depending on his credentials, or returning him to where he came from in case of an invalid login) require a lot of dialog control logic, and no application that the author is aware of is capable of nesting arbitrary dialogs on multiple levels.

Since users have a long-established conceptual model of nested dialogs from window-based applications, they will likely transfer that model to hypertext-based applications. However, because of insufficient dialog control logic, many applications still violate users' expectations today when they send them to other pages than they intended to reach (in some web applications, for example, login forms return users to the homepage after a successful login instead of sending them to the area that required authorization, requiring the user to navigate back to the desired area). This violation of the dialog principles of controllability and conformity with user expectations imposes a high cognitive and memory load on the user.

Based on the hypothesis that nested dialogs can improve the usability of hypertext UIs, this thesis therefore introduces a notation for specifying complex, nested dialog flows, and presents a framework to control them.

2. The Dialog Control Framework

Hypertext-based applications usually employ a front controller architecture, as described by the Model-View-Controller (MVC) paradigm [Kr88] which suggests the separation of user interface, application logic and control logic. However, many MVC implementations do not fully separate application and control logic. For example, in the Jakarta Struts framework, the action objects which implement the application logic also decide where to forward a request, and the controller just executes that forward command (Figure 1).

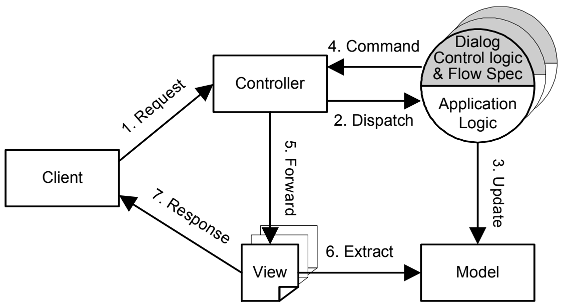

Figure 1: Coarse architecture of the Jakarta Struts framework (dialog control logic and flow spec shaded)

In this approach, the dialog control logic and dialog flow specification is distributed over all actions. This allows the actions to make only relatively isolated dialog flow decisions, and exacerbates the implementation of more complex dialog structures with higher-order constructs like nested dialog modules. Also, the hard-coded decentral implementation of the dialog control logic is relatively inflexible, almost unsuitable for reuse and hard to maintain. Finally, achieving presentation channel independence would require additional effort and possibly redundant work: Since the dialog flow depends on the presentation channel, while the application logic does not, their close coupling forbids the reuse of actions on multiple presentation channels.

In contrast, the Dialog Control Framework presented in this thesis features a very strict implementation of the MVC paradigm, completely separating not only the application logic and user interface, but also the dialog flow specification and control logic: Here, the *controller* decides where to forward requests by using a central dialog flow model to look up the receivers of events generated by dialog masks and actions (Figure 2).

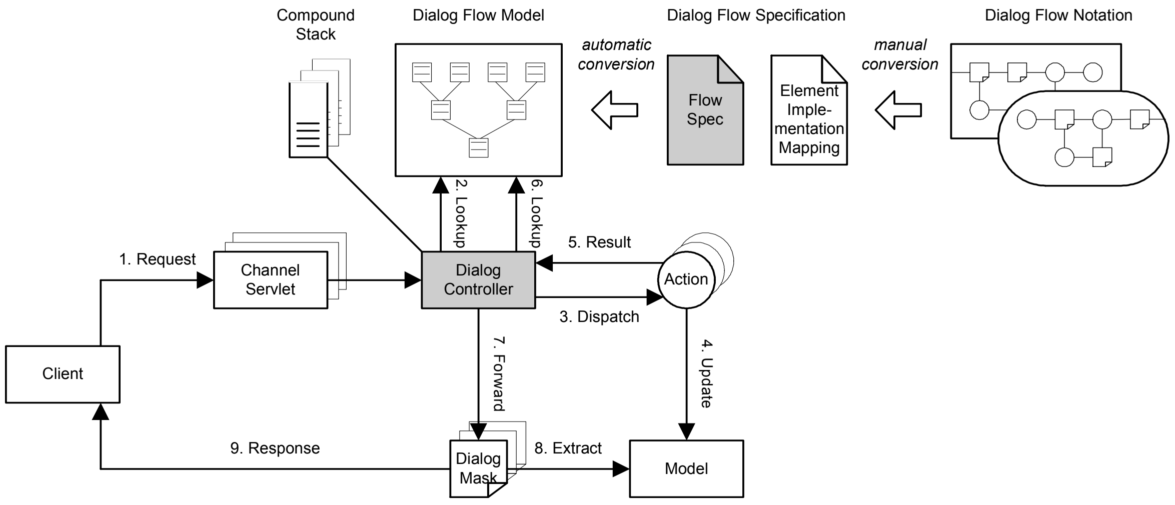

Figure 2: Coarse architecture of the Dialog Control Framework (dialog control logic and flow spec shaded)

The Dialog Control Framework was implemented in the Java programming language. Channel servlets receive the requests coming in from clients on various presentation channels, extract the events encoded in them and send them to the central Dialog Controller. The Dialog Controller keeps track of the state of the users' nested dialog modules in compound stacks, and determines how to handle incoming events by looking them up in an object structure that models the application's dialog flow. Depending on the type of event and the user's current dialog state, the Dialog Controller may dispatch the request to an action, forward it to a dialog mask, or nest or terminate dialog modules by pushing them onto the compound stack or removing them again. To use this framework, the application developer only needs to provide a set of Java Server Pages representing the dialog masks, a set of action classes containing the application logic, and two XML documents containing the dialog flow specification and the mapping of elements to their implementations. These documents are machine-readable representations of dialog graph diagrams drawn in the Dialog Flow Notation, and parsed upon initialization of the framework to build the dialog flow model's object structure.

3. The Dialog Flow Notation

Inspired by Harel's Statecharts [Ha87], the Dialog Flow Notation represents the dialog flow within an application as a directed graph of states connected by transitions. The notation refers to the transitions as **events** since in the implementation, states cannot specify which state shall become active next – as explained above, the dialog controller decides which transition to follow upon receipt of an event generated by a state. The notation refers to the states as dialog elements, discerning four types with different semantics: **Dialog masks** are hypertext pages generated by the server and rendered by the client, while **actions** are operations performed by the application logic on the server. Note that dialog graphs constructed out of these atomic elements do not need to be bipartite, since applications might want to display multiple masks without intermittent application logic operations, or perform a series of operations without displaying pages.

Dialog graphs can be encapsulated in **dialog modules** and **dialog containers**, which can be part of greater dialog graphs themselves. When these compound elements receive events from the exterior dialog graphs that they are integrated in, their interior dialog graph is executed, thereby realizing the key concept of nested dialogs. When the interior dialog graph of a dialog module terminates, it generates a terminal event that continues the traversal of the module's exterior dialog graph. Figure 3 illustrates these concepts by example:

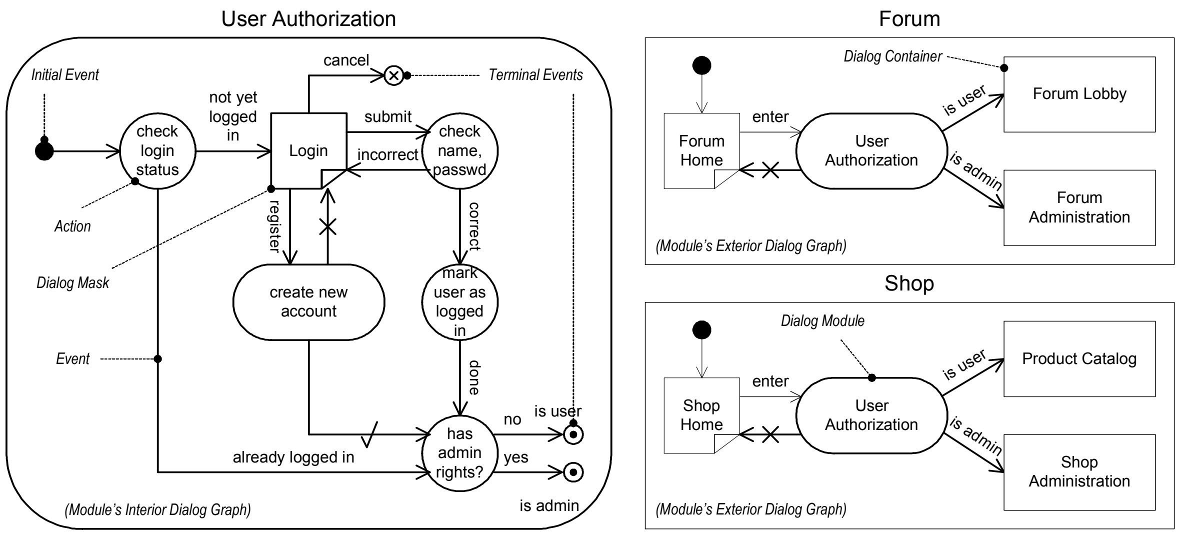

Figure 3: Example of interior and exterior dialog graphs of a User Authorization *dialog module*

Here, the framework activates the *User Authorization* module (left) when the *Forum Home* or *Shop Home* mask (right) generate the *enter* event. Traversal of its interior dialog graph begins at the initial event, which leads to the *check login status* action that determines if the user is already logged in and generates a corresponding event. Assuming the *Login* mask features *submit*, *register* and *cancel* buttons, it can generate the respective events which either lead to an action checking the entered credentials, initiate traversal of the *create new account* module on yet another nesting level, or terminate traversal of the interior dialog graph, generating a *cancelled* terminal event in the exterior dialog graph (marked by an X) which leads back to the respective *Home* mask. If the credentials entered into the *Login* mask were correct, the user is marked as logged in, and a final action determines which access rights are associated with his account (Note that by splitting the application logic into these relatively fine-grained operations instead of implementing them all in one action, the module can react flexibly to different situations, like bypassing the credential check when the user is already logged in or just created an account). Depending on the user's rights, the module finally terminates generating either an *is user* or an *is admin* terminal event. According to this event, the framework activates the *Forum Lobby* or *Forum Administration* dialog container (if the module was called in the *Forum* container), or the *Product Catalog* or *Shop Administration* container (if the module was called in the *Shop* container). The demonstrated reuse of the *User Authorization* module in different exterior dialog graphs is only possible because the module does not specify the following element itself, but leaves that decision to the Dialog Controller which knows the specification of the exterior graph that the module is nested in. The same event generated by the same dialog element can therefore lead to different elements in different contexts.

The above-mentioned containers differ from modules in that their interior dialog graphs do not terminate by themselves. Containers are intended for encapsulating sub-applications with a "closed" dialog graph lacking a natural exit point. Traversal of their interior dialog graph (including traversal of all their currently nested compounds) can only be aborted by the framework if another container shall be activated on the same nesting level. In order to abort compounds in a controlled way, a special **abort dialog graph** can be specified for each of them, which might for example ask the user if he really wants to abort (also giving him a chance to resume the original dialog graph where he left off), or if he wants to save any unsaved data before aborting.

4. Conclusion

The strict separation of user interface design, application logic implementation, dialog flow specification and dialog control logic in the Dialog Control Framework presented here has a number of advantages: Firstly, it enables a high degree of flexibility, reusability and maintainability for the components of all four layers. Secondly, presentation channel-independent applications can be built with minimal redundancy: Only the dialog masks and dialog flow specifications have to be adapted for different channels, while the application logic can be implemented only once and the dialog control logic is already provided by the framework. Finally, since the central dialog control logic is aware of the whole dialog flow specified for an application, it can provide mechanisms for the realization of complex dialog constructs. For example, the application developer can use context-independent dialog modules that may be nested, aborted and resumed without having to worry about event handling, stack management and resume point identification.

The associated Dialog Flow Notation is essential for providing the specification of the dialog flow to the framework. Since it does not require a detailed knowledge of the underlying protocols and technologies, but instead works with three relatively intuitively understandable concepts ("masks contain what the user sees, actions contain what the system does, and compounds contain transactions the user can perform"), it can also be used by people without programming experience, such as representatives of the application's target audience, usability experts and user interface designers. Therefore, the notation's dialog graph diagrams can be used as a communication tool throughout the software development process. In addition, the graphical dialog flow specifications can be transformed into XML documents by following a set of simple rules, allowing for a very efficient transition from specification to implementation.

This thesis provides the foundation for this approach to dialog control in hypertext-based applications by defining the basic concepts, elements and rules of the Dialog Flow Notation and providing a prototypical Dialog Control Framework implementation. Further research should examine how the robustness of the framework in unforeseen situations can be increased (e.g. when the user clicks the browser's Back button), how parameters can be associated with events to make them more flexible means for transporting information, and how the framework can be optimized to handle high load in a production environment.

References

[Ha87] Harel, D.: Statecharts: A visual formalism for complex systems. In *Scientific Computer Programming* 8(3), 231-274

[Kr88] Krasner, G.E.: A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk. In *Journal of Object-Oriented Programming* 1(3), 1988, 26-49

[RF96] Rice, J., Farquhar, A., Piernot, P, Gruber, T.: Using the web instead of a window system. In *ACM Conference on Human Factors in Computing Systems (CHI '96) Proceedings,* ACM Press 1996