Form-Based User Interface - The Architectural Patterns

A Pattern Language

Jens Coldewey

sd&m GmbH & Co.KG,
Thomas-Dehler-Str. 27
D-81737 München,
Germany
email: jensc@sdm.de
Phone: +49-89-63812-125
http:/www.sdm.de/g/arcus/

Ingolf Krüger

Technische Universität München,
Fakultät für Informatik,
Arcisstr. 21
D-80290 München,
Germany,
email: kruegeri@informatik.tu-muenchen.de
Phone: +49-89-2892-8166
http://wwwbroy.informatik.tu-muenchen.de/~kruegeri/

Abstract

Despite all the benefits of object-oriented user interfaces, there are still domains that call for a form-based user interface. Business information systems that support fast processing of few, well-defined use cases are typical examples. This pattern language helps to develop the software architecture for such systems.

The paper is part of a larger effort to collect patterns for business information systems, currently pursued by the ARCUS team.[1]

1 Introduction

1.1 Form-Based and Object-Oriented User Interfaces

When you design user interfaces you have to decide for a basic interaction style. In the recent years object-oriented user interfaces (OOUI) have become more and more popular [Col95]. These interfaces feature sophisticated selection and navigation techniques to select objects and context-sensitive menus to manipulate them. Figure 1 shows an example of an OOUI.

1 This work is sponsored by the German Ministry of Research and Technology under project name ENTSTAND

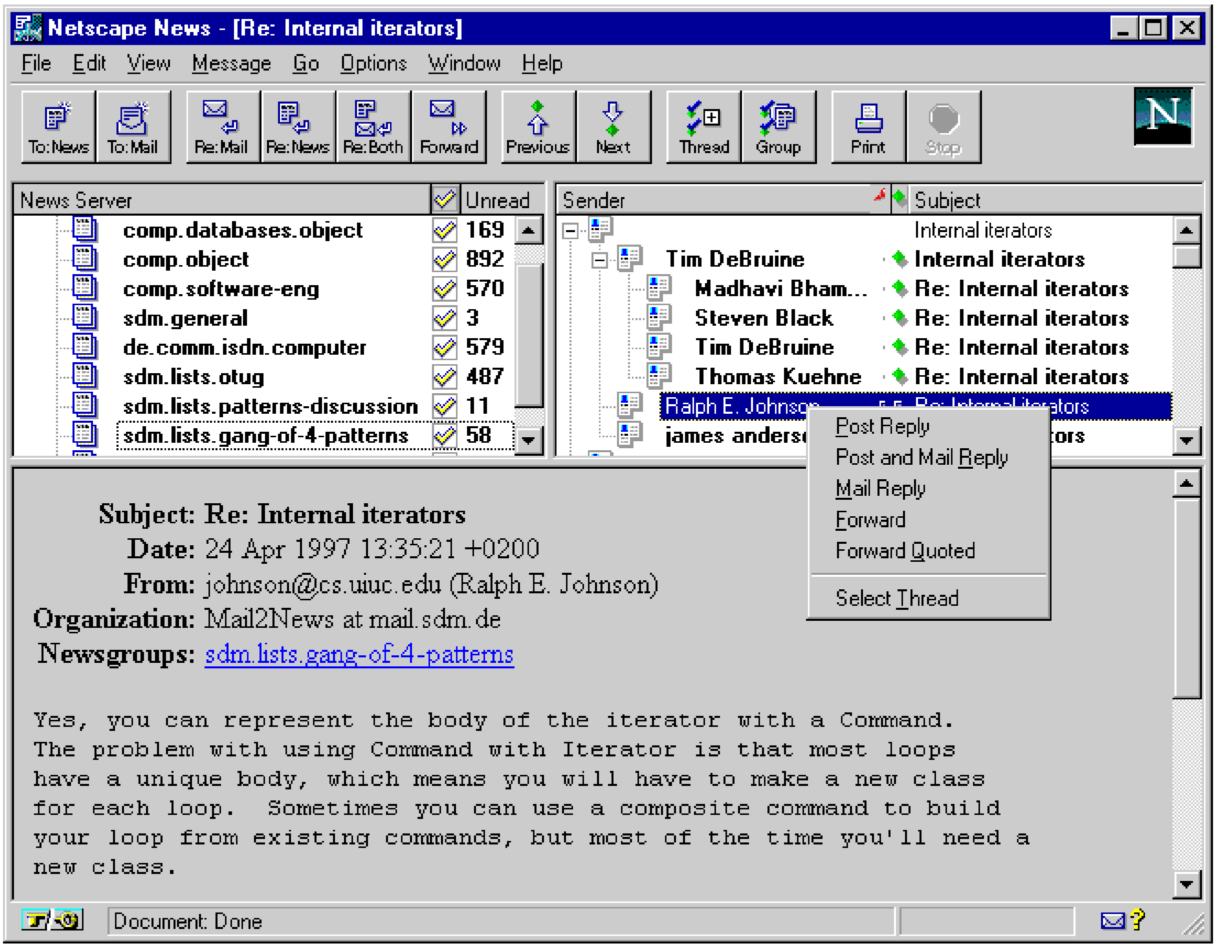

Figure 1: Example of a news reader with an object-oriented user interface. Beneath the menu bar there are two subwindows for navigation among object groups and selection of objects. The larger window below shows the contents of the object. Context sensitive pop-up menus provide actions to manipulate the selected objects.

An alternative to OOUIs are form-based user interfaces. Instead of context-sensitive menus and object navigation they provide a set of forms organized in dialog sequences, which model the work flow of the user. Have a look at Figure 2 on page 4 for an example of this interaction style.

It is tempting, to confuse the interaction style with the implementation technique of the interface. Both examples use the elements of a graphical user interface and both have an object-oriented implementation. However, Figure 1 shows the typical elements to navigate between objects without suggesting any workflow. Figure 2 on the other hand shows no objects but is one particular step in the workflow.

As the following table shows, you may implement both styles on GUIs. Many 4GL tools generate FBUIs on a GUI. Alphanumeric terminals and Web-Browsers imply certain restrictions.

	Alphanumeric Terminal	GUI	Web-Browser
Form-based User Interface	✓	✓	HTML forms
OO User Interface	×	✓	only with JAVA or Active-X

Though most modern user interfaces provide OOUIs, the need for traditional form-based interfaces persists for several reasons:

- *Often, skilled users need to process well-defined use cases as fast as possible.* The users know what data the system needs and have all the data at hand; they need a user interface, which reflects their work flow and needs as few key strokes as possible. "Media switching" between keyboard and mouse slows down their work. For instance, consider a passenger check-in system at an airport.

- *Untrained or semi-skilled users often need step-by-step assistance while performing their tasks.* For instance, an ATM supports only one or two use cases (getting money and displaying account information) but should serve customers of all skill-levels. Because FBUIs direct the workflow, the user does not necessarily need to know the next step to go. However, well-designed FBUIs provide enough shortcuts to serve skilled users too.

- *The system might not have enough resources to support an OOUI.* These can cause significant load to a database server while retrieving information of which only a small fraction may be relevant to the user. For instance, you need to evaluate consistency rules to determine the selectable menu items or push buttons, as well as data to fill list boxes and navigation controls. This may sum up to several times the load you need to process the use case. Therefore, OOUIs cause more traffic between the client and the server. World-wide distributed applications, such as reservation systems or WWW servers, often have low-bandwidth connections between clients and servers. You can use elaborate caching techniques to at least reduce the traffic but that increases design, implementation, and test effort considerably.

- *The existing hardware infrastructure may not support an OOUI.* You need a high-resolution screen and a pointing device to use an OOUI effectively. Yet, many industrial customers still have thousands of alphanumeric terminals. Replacing the latter with graphical terminals may sum up to several million dollars in hardware costs alone.

In all these cases it may be a good choice to implement a form-based user interface (FBUI) and to accept several drawbacks:

- *An FBUI is less flexible than an OOUI.* Because the user interface determines the work flow, any unforeseen use case is hard to handle - if it is manageable at all. New use cases often require changes of the user interface, resulting in changes of the software. Minor flaws in the analysis often lead to non-optimal support for the users and impede acceptance, because the system dictates a non-intuitive work-flow. Hence you have to take care during analysis and design to make the FBUI extensible, which is harder than with an OOUI.

- *Working with a complex FBUI is harder to learn.* An FBUI usually works with a sequence of forms you have to fill out to complete a task. The user has to be familiar with the use cases and the corresponding forms to work with the system. This may imply an increased training effort if the users are not too familiar with the workflow. On the other hand, it enables the user to process a known use case quickly.

- *FBUIs are not optimal to support strategic decision systems.* If users work with the system only every now and then to analyze data and collect information for strategic decisions, they can take advantage from the flexibility of an OOUI. Hence, OOUIs are preferable for management information systems or data warehouse applications.

Of course, you may mix form-based elements into an OOUI and vice versa to tune for the user's requirements.

This pattern language describes an architecture for the design and implementation of FBUIs.

1.2 Running Example

We will use a running example of a flight reservation system to describe the patterns. The reservation form depicted in Figure 2 corresponds to the layout of the flight tickets. There are three subdialogs to find reservations, to have a look at available flights, and to get help. The user can issue three different domain level actions: She can start a new reservation, change an existing one, or cancel a reservation. Figure 3 models this behavior as a state event diagram. Note that there are two transitions named "Find...": If the user presses the `Find...` button *with* a reservation id entered, the system just fills the current form. If she presses the button *without* a reservation id, the system opens a new form to find reservations. We have skipped additional transitions in the subdialogs, as well as general actions, such as processing help.

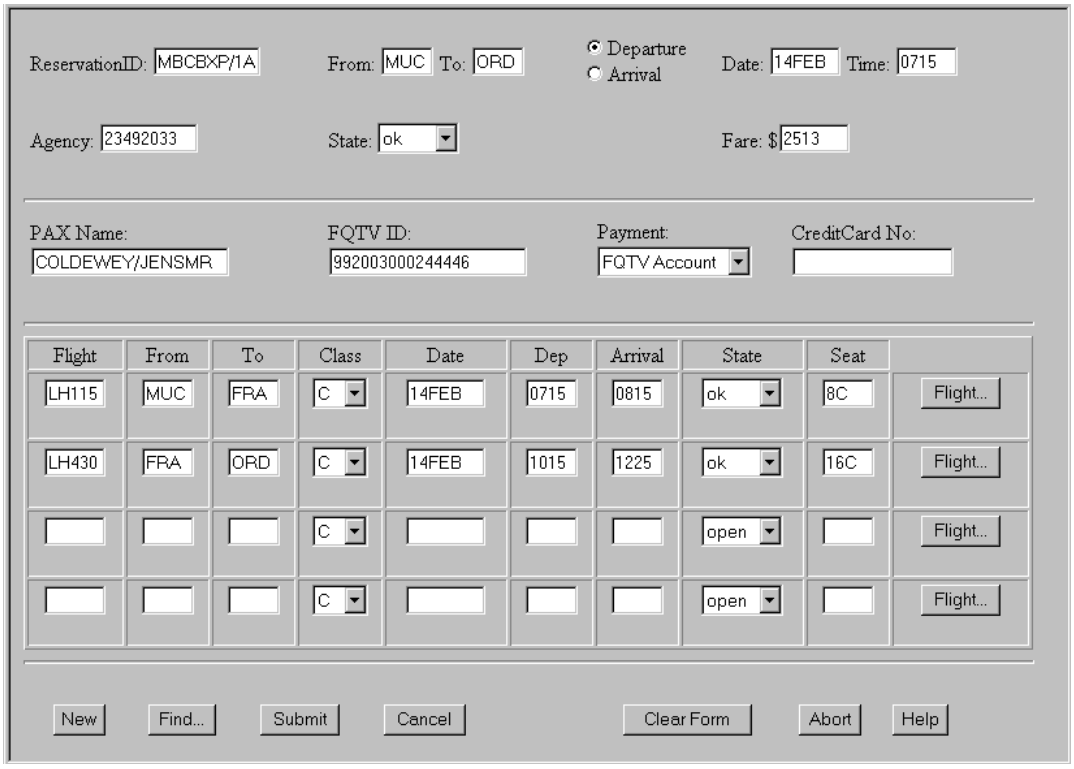

Figure 2: Reservation form of a flight reservation system.

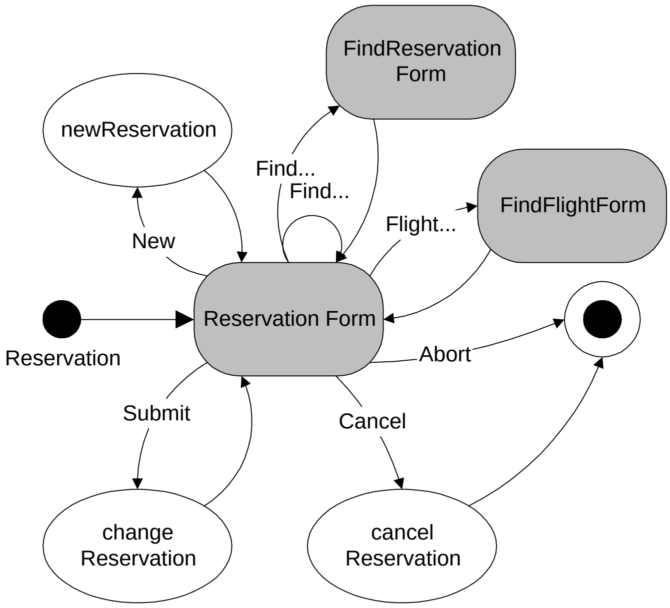

Figure 3: State diagram of the flight reservation dialog. The shaded states represent forms while the unshaded ellipses represent actions. The reservation form of Figure 2 is the central form with two subdialogs to find a flight and a reservation respectively.

2 The Pattern Language

The pattern language consists of ten patterns, shown in Figure 4. This paper describes only the shaded patterns in depth and provides summaries ("pattlets") for the rest.

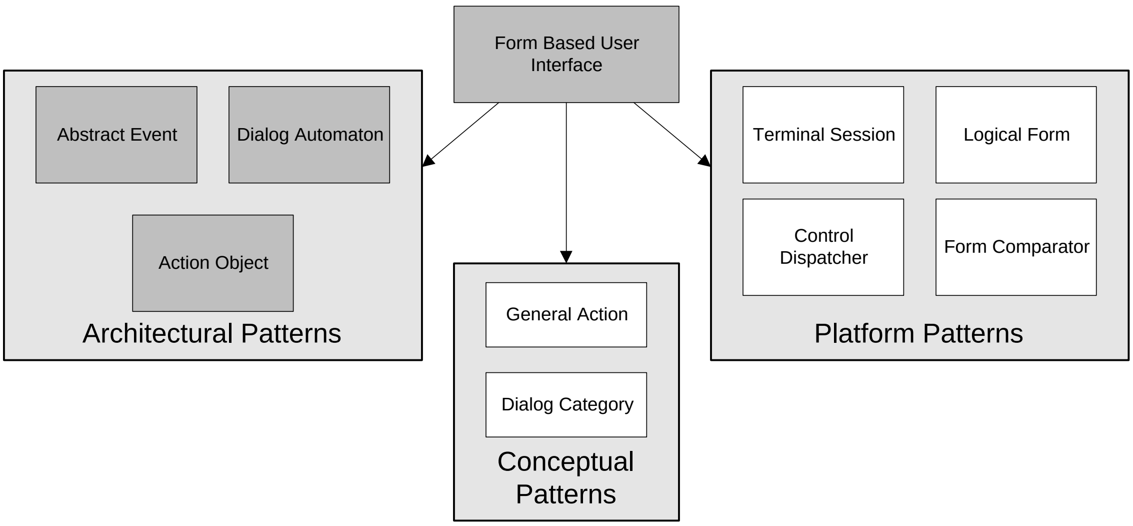

Figure 4: The patterns of the Form-Based User Interface language.

The language is contains three parts: *Architectural* patterns describe classes and subsystems of the architecture that do not depend on a specific platform. *Conceptual* patterns describe analysis concepts that help to organize the architecture. The *Platform Patterns* encapsulate different hardware and operating system aspects.

This paper starts with the Form Based User Interface pattern. The three architectural patterns follow in the order, in which events are processed: Abstract Event, Dialog Automaton, and Action Object. Finally, we summarize the other patterns.

Known Uses[2]

[Den91] describes a 3GL architecture conforming to this language. Several mainframe projects at sd&m use the architecture, for instance the Thyssen/IAB project [Zeh88] and the TLR project [Bis96]. They all use Dialog Automaton, Dialog Category, General Actions, Form Comparator, Control Dispatcher, and Terminal Session.

You find some of the patterns in most user interface frameworks. Microsoft Foundation Classes and StarView use Abstract Event, Logical Form, and General Action. The DD35 project [KuR96], as well as the EASY-C project [SaZ96] also use elements of the pattern language. Action Object is a variant of the `Command` object of ET++ [Gam92], which is also used in the SIGMAPLAN project [RCo95]. Taskmaster [MNR97] uses Dialog Automaton.

2.1 Form-Based User Interface

Abstract

This pattern describes an architecture for systems using form-based user interfaces. It shows how to combine the other patterns of this paper.

Context

You are developing a business information system with a form-based user interface.

Problem

How do you structure the user interface software?

Forces

- *Technical reuse* versus *domain level reuse:* We distinguish two different approaches to reuse for our purposes; the *technical approach* tries to identify technical components, such as containers or state machines and makes them as generic as possible. This approach offers reusability across a large variety of domains. On the other hand it usually does not encourage a design that identifies recurring domain level components, such as customers or orders. If you design software for a specific domain area, you may earn more value from *reusing domain level components*. However, you may scale down most technical concepts to subsystems of a component. Therefore, these two approaches are not necessarily contradictory.

- *Supporting additional interfaces:* The user interface is only one of several access points to a large system. Batch interfaces and neighborhood systems may also rely on services of your system. A user interface architecture should allow isolated changes of the user interface without affecting the other interfaces. Changes at the domain level should propagate consistently to all interfaces.

2 Most known uses of our pattern language are very similar. To save space we collect them here.

- *Communication cost* versus *software distribution cost:* This is a particular force for client server systems. If the server controls the user interface, both user interface control and every user activity cause network traffic. Shifting control to the client relieves the server from processing and, therefore, raises additional freedom for user interface design. However, with this approach, you need a homogenous client landscape and a technique to distribute updated software to the clients. With several thousands of clients, this may be a severe challenge.

Solution

Use a layered architecture [BMR+96], separating the user interface and the domain kernel. Partition the user interface into five subsystems according to Figure 5. *Control Dispatching* and *I/O Wrappers* encapsulate the operating system. These two subsystems are specific to the platform you work on. *Event Abstraction*, *Dialog Control* and *Domain Kernel Interface* represent the logical part of the user interface. They are independent of any specific implementation platform.

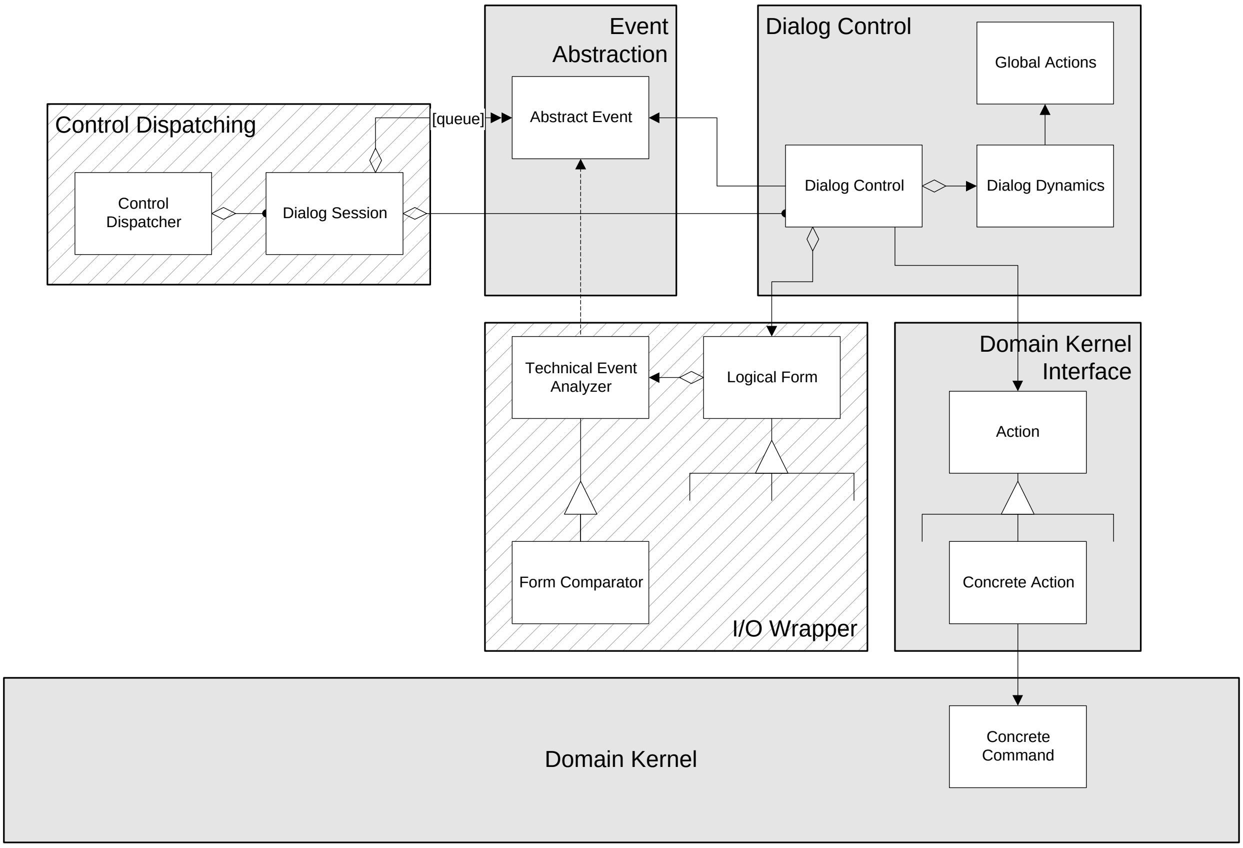

Figure 5: Class design of the form based user interface. Shaded areas indicate generic subsystems, dashed areas denote subsystems specific to the implementation platform. The `Logical Form` and `Action` classes need to be subclassed according to the application. This class design is one possible instantiation of the pattern language, tailored for mainframe systems. We have skipped minor helper classes, such as factories and adapters.

Logically, any user activity arrives at the `TechnicalEventAnalyzer`. This class generates `AbstractEvents`, representing the semantic of the user activity. This event is forwarded to the `DialogControl`, which raises the appropriate `Action`. Finally the `LogicalForm` provides a platform independent way to access the forms. The `ControlDispatcher` and the `DialogSession` provide session and state information for transaction monitors, according to the <u>Control Dispatcher</u> and the <u>Terminal Session</u> patterns.

Consequences

- *Technical reuse:* The pattern language promotes a technical reuse strategy. It is sensible in a component based approach only if the components have user interface parts of considerable complexity. If the dialogs of the system span several components, this architecture makes it is hard to integrate their dialogs: the design does not follow the object structure of the domain kernel adequately, but corresponds to the use case structure of the domain, instead.

- *Other interfaces:* Due to the layered architecture, the pattern language separates domain level issues from user interface issues. This enables you to provide additional interfaces to the same domain kernel without affecting the user interface code.

- *Communication cost and software distribution:* The design enables you to optimize the traffic between the operating system and the user interface control as well as between the subsystems of the user interface. Therefore, you can place a client server cut anywhere between the subsystems. Consequently you can tailor communication traffic to your needs. In particular, the design works well with mainframe systems and WWW-Servers.

Related Patterns

Abstract Event discusses the event abstraction subsystem. Dialog Automaton, General Action and Dialog Category form the Dialog Control subsystem. Action Object deals with the domain kernel interface. Control Dispatcher and Terminal Session discuss control dispatching, Logical Form and Form Comparator address issues of I/O wrapping.

2.2 Abstract Event

Abstract

Abstract Event presents a way to process events regardless of their physical origin.

Context

The operating system signals user interactions by means of technical events, such as keystrokes or menu selections. In general, these technical events do not directly correspond to a single domain specific task or to a user interface activity. Several technical events may cause the same domain task and a single technical event may result in several domain tasks.

Problem

How do you facilitate a flexible mapping between technical events and the corresponding actions?

Forces

- *Flexibility* versus *complexity:* A mapping mechanism increases flexibility. This pays off during maintenance if you change the physical appearance of the interface without changing the logical behavior. It may also avoid code duplication. On the other hand the mapping adds an additional level of indirection. You have to define and maintain the mapping semantics and you need additional code to interpret the semantics.

- *Call technique of the user interface API:* Most user interface APIs have their own mechanism to transfer technical events to the application program. For instance, Web-Servers or CICS transaction monitors, call a certain script or program according to the address or the transaction code the user entered. Windows environments usually supply the main application loop with messages that describe the technical events. Most user interface frameworks provide mechanisms to distribute these messages.

Solution

Define a separate level of abstraction called *Abstract Event*. A `TechnicalEventAnalyzer` maps technical events to `AbstractEvents` (Figure 6). The reaction of the system depends on these `AbstractEvents` rather than on the technical events of the operating system. The `AbstractEvents` contain arbitrary parameters. An `AbstractEventInterpreter` interprets the events. An `AbstractEventQueue` buffers prioritized abstract events. Model the queue as a Singleton [GHJ+94].

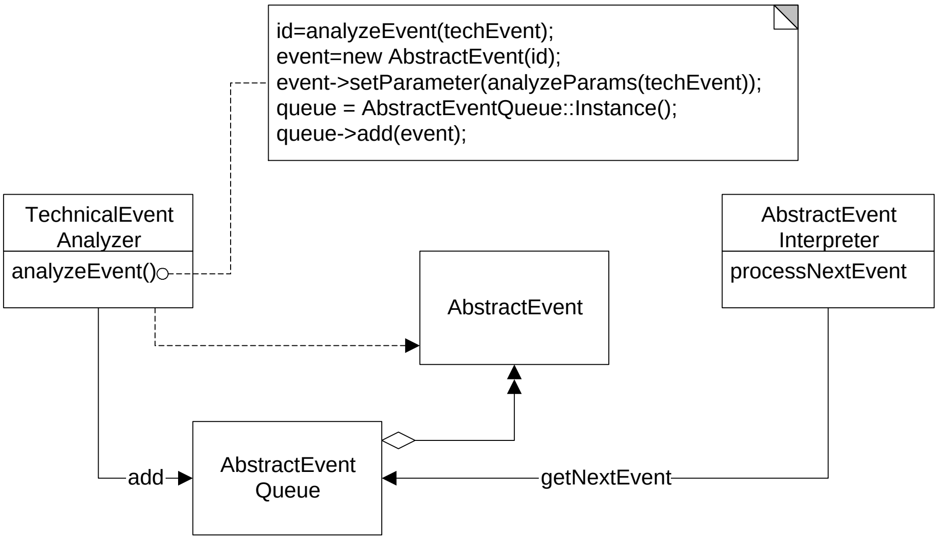

Figure 6: Structure of an Abstract Event

Example Resolved

You can derive the abstract events for flight reservation from the state event model in Figure 3. The following table shows the technical events, the corresponding abstract events and the parameters that come (at least) with every abstract event. Note that the same technical event “Find... button pressed” leads to two different abstract events, depending on the contents of the reservation id field (second and third row).

Technical Event	Abstract Event	Parameter
“`New`” button pressed	`New`	
“`Find...`” button pressed with reservation id field set	`Fill`	reservation id
“`Find...`” button pressed with reservation id field empty	`Find`	all data the user has entered
Any of the “`Flight...`” buttons pressed	`Flight`	flight number in the current row
“`Abort`” button pressed	`Abort`	
“`Cancel`” button pressed	`Cancel`	Reservation number
“`Submit`” button pressed	`Submit`	All data the user has entered

Consequences

- *Flexibility:* The extra level of abstraction allows arbitrary mappings between the physical action the user invokes and the domain level effect. You can handle function keys and other events easily by changing the `TechnicalEventAnalyzer` according to the current state of the interface. Making the `TechnicalEventAnalyzer` configurable facilitates user definable key assignments. The sophistication level of the `TechnicalEventAnalyzer` determines the level of flexibility you can achieve.

- *Complexity:* The `AbstractEvent` and the `AbstractEventQueue` are simple classes you can code in a few lines or draw from a container library. The most complex part is the `TechnicalEventAnalyzer`. Its complexity determines the overall complexity of an AbstractEvent.

- *Compatibility to user interface API call technique:* If the user interface API does not support the additional level of indirection this pattern may add significant complexity. Especially many 4GL tools may cause trouble. They couple user actions or transaction codes to program entry points, such as methods or program modules, respectively. Depending on the flexibility of the tool, implementing AbstractEvent may just be a matter of convention or it may result in considerable design and implementation effort.

Implementation

- *Increased flexibility with other sources of abstract events:* So far, we have considered the `TechnicalEventAnalyzer` as the only source of `AbstractEvents`. The concept is flexible enough to allow other sources, too. In particular, the `AbstractEventInterpreter` may generate new `AbstractEvents`. This technique results in a cascade of `AbstractEvents` in reaction to a single technical event, providing a powerful approach to structuring the reaction of a system to user input.

- *Several abstract events from one technical event:* In mainframe environments you often find forms containing a list of items. For every item you can specify an action you would like to perform. Changing the data of the item may result in an update operation; checking a delete row may result in a delete action. This technique gives the user the chance to issue several use cases with one submit key, thus reducing the number of transactions. It is straightforward to extend the `TechnicalEventAnalyzer` to generate more than one `AbstractEvent` in reaction to a single technical event.

- *Prioritizing Abstract Events:* Consider two queued `AbstractEvents`: The first one results in a new form, which shows order items instead of customers, the second deletes the currently displayed object. Clearly, the order of these two operations is crucial for the result: If you change the form first, the order item will be deleted, if you perform the delete operation first, the customer object is erased. In implementations that support this kind of interaction, you have to prioritize the `AbstractEvents` to ensure predictable results. You may put events into the same priority category that do not interfere with one another.

Variants

Instead of the sketched producer-consumer solution, you may also split the abstraction process into several levels. For example, in a window system a control may abstract technical events to low-level abstract events and propagate them to their parent window. The parent - perhaps a complete form or a part of it - turns this abstract event into another higher-level event and transfers it to the enclosing window and so on. This is a Chain of Responsibility [GHJ+94, p. 223]. In this scenario, the system does not interpret the abstract events in one big step, but every part of the user interface interprets an event on its own level of abstraction. This is a suitable approach if the user interface contains many different parts that work independently from one another.

Related Patterns

Dialog Automaton is an approach to design the `AbstractEventInterpreter`. Form Comparator shows how to analyze technical events on a web server or a mainframe.

2.3 Dialog Automaton

Abstract

Dialog Automaton describes how to implement the dynamic aspects of dialogs[3] and the coordination of a dialog's forms using a finite state machine.

Context

You have specified the dialogs of your system using a state event model. Every state corresponds to a form while every event corresponds to an Abstract Event. The resulting state event model is of considerable complexity.

Problem

How do you implement sequences of forms flexible and maintainable?

Forces

- *Table-driven* versus *hard-coded behavior:* You have the choice either to define the behavior in a table, interpreted by general classes, or to write specialized code that directly implements the behavior. A table-driven solution offers easy maintenance but usually results in complex data structures that are hard to understand. Hard-coded behavior often is easier to understand but requires recompilation after every change.

- *Maintainability:* The users often ask for changes that affect mainly the state event model of the interface. Consider additional shortcuts and navigation possibilities as an example. Note that this demand is specific to form-based user interfaces, because they are more restrictive with respect to the interaction sequence than carefully designed object-oriented interfaces.

- *Extensibility:* Adding additional forms is another important aspect of maintainability. It is not as often as changes to the navigation. Often the extension requirements arise after the system worked for several years and nobody of the original design team is in duty anymore.

Solution

Use a finite state machine to control the sequence of forms and the actions to perform in reaction to a certain `AbstractEvent`. An `AutomatonDescription` table contains the state event model of the dialog; the `DialogAutomaton` interprets this table and starts the appropriate actions.

---

3 We use "dialog" denotes a sequence of forms the user may traverse to complete a certain task. Note that this is different from a "dialog box" in GUI environments. Their counterpart in this document is a "form".

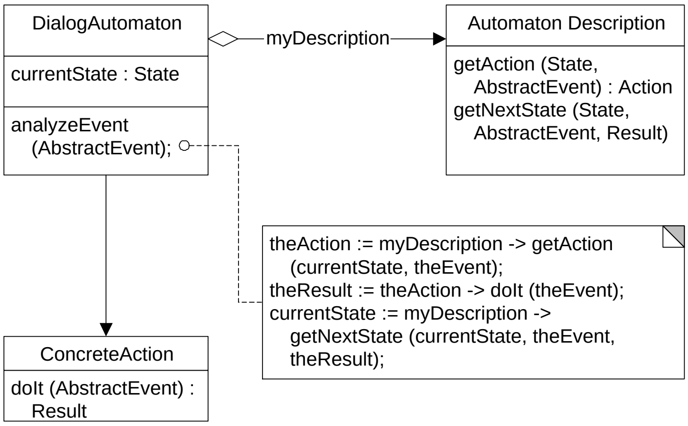

Figure 7: The structure of the Dialog Automaton. The pattern links Abstract Events to Action Objects using a finite state machine.

Example Resolved

There are six Abstract Events in the preceding flight reservation example: `New`, `Submit`, `Cancel`, `Fill`, `Find`, `Flight` and `Abort`. The `AutomatonDescription` contains the following table to model the state event diagram:

State	Abstract EventId	ActionId	Result	NextState
`ReservationForm`	`New`	`ActNewReservation`		`ReservationForm`
	`Submit`	`ActChangeReservation`		`ReservationForm`
	`Cancel`	`ActCancelReservation`	`OK`	`end`
			`Error`	`ReservationForm`
	`Fill`	`ActFindReservation`		`ReservationForm`
	`Find`	`ActOpenForm`		`FindReservationForm`
	`Flight`	`ActFindFlight`		`FindFlightForm`
	`Abort`	`ActCloseForm`		`end`
`FindReservationForm`	`OK`	`ActPopForm`		`ReservationForm`
	`Abort`	`ActCloseForm`		`ReservationForm`
`FindFlightForm`	`OK`	`ActPopForm`		`ReservationForm`
	`Abort`	`ActCloseForm`		`ReservationForm`

Note that the `ActCloseForm` action replaces the end state of the state event model. `AutomatonDescription` extracts the `EventId` from the Abstract Event it gets. It transfers the `ActionId` to a `ConcreteAction` to initiate the corresponding action. The `AbstractEvent` also contains the parameters of the action. The `ReservationForm` allows two possible states on a `Cancel` event, depending on the result of the `ActCancelReservation` action.

Consequences

- *Table-driven solution:* The pattern uses tables to represent the state event model. This enables you to implement the generic parts by code while a table contains the dialog specific parts. The solution decreases code size on the expense of introducing constant data. Changes to the state event model result in changes of the data and need no recompilation. However, you have to keep the state event table consistent, which is a notable issue with large automata.

- *Maintainability and Extensibility:* Dialog Automaton supports changes of the dynamic behavior adequately. Integrating additional forms into a dialog results in changes of the table entries of all states from which the new form is accessible. There are no code changes necessary in either case.

- *Separation of concerns:* The pattern takes a technical approach to separate concerns. The `AutomatonDescription` represents dynamic behavior, while other classes represent static aspects of the user interface. However, you may cluster static and dynamic aspects of a dialog in one component as long as all the forms of a dialog belong to the same component. If a dialog spans several components, the automatons of the components have to interact, which may be difficult to accomplish.

Variants

Instead of implementing the automaton with table, you can apply the State pattern [GHJ+94, p. 305]: For every form define a subclass of `LogicalForm` (Figure 5). These subclasses correspond to the `ConcreteStates` of the State pattern, while the `LogicalForm` takes the role of the `State` participant. The `DialogAutomaton` class is the `Context` of the `State`.

Note that this approach is not data-driven. Instead, the `ConcreteForms` "know" their specific transitions. This variant does not separate layout from dynamics. Therefore, it supports independent forms and thus increases reusability of individual forms. On the other hand, this approach does not support Dialog Categories and leads to impenetrable code if you have a complex state model. Use it with form-based user interfaces only if you cannot apply Dialog Categories and the state model is not too complex.

Related Patterns

The pattern works best if you combine it with Abstract Events and Action Object. You can simplify the state event table using General Action.

2.4 Action Object

Abstract

Action Object provides a standardized interface to all activities of a user interface. It addresses both accessing the domain kernel, and performing actions inside the user interface, such as opening and closing forms.

Context

You are using Abstract Event to decouple technical events from logical events and a DialogAutomaton to interpret the Abstract Events.

Problem

How do you provide a uniform way to start actions?

Forces

- *Decoupling* versus *number of classes:* Different actions require different parameters. It is good practice to hide these differences from the interpreter of the Abstract Events, thus making the interpreter generic. On the other hand, the information hiding raises the need for additional classes.

- *Maintenance:* User interfaces vary faster than the domain kernel, because most changes of the domain requirements result in a change of the user interface but not vice versa. Typical changes ask for additional information presented in a form or additional actions the user can start in a dialog step.

- *Uniform reaction:* Actions may raise errors and other spontaneous events, which the user interface needs to process (see [Rnz96, chapter 2.5]). A user interface should provide a uniform reaction to errors without code duplication.

Solution

Build up a hierarchy of Action Objects. Define `ConcreteActions` for every action an Abstract Event can cause (Figure 8). The `ConcreteAction` determines the appropriate parameters, method calls, and result processing. The common superclass `Action` defines the abstract protocol of all Action Objects.

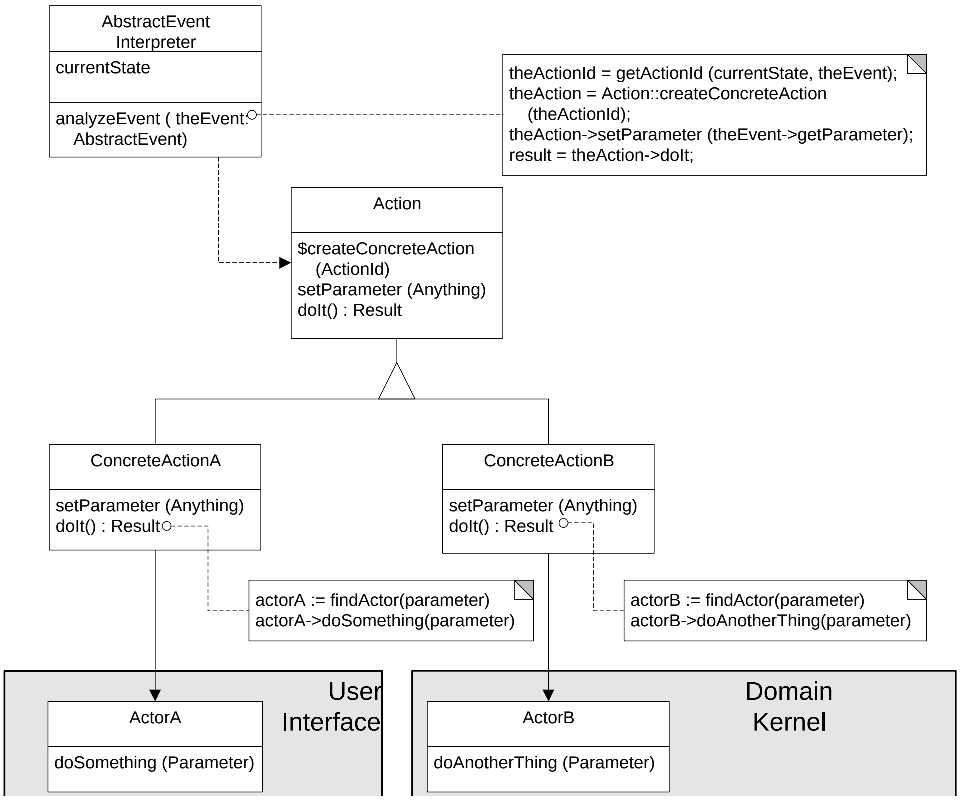

Figure 8 The Structure of the Action Object pattern.

Example Resolved

The reservation form of our example leads to the `ConcreteActions` described in the following table. The table also contains the parameters the actions need. Note that the actions either retrieve the parameters from the `AbstractAction` object or from global objects, such as the `DialogAutomaton`.

Concrete Action	Parameter	Description
`ActOpenForm`	`AbstractEvent`	Opens a new form according to the `AbstractEventId`. If the `AbstractEvent` contains any parameters the action offers the data to the new form.
`ActCloseForm`	---	Closes the current form discarding changes
`ActNewReservation`	---	Calls the domain kernel, which instantiates a new reservation object returning the reservation id. Writes the id into the current form
`ActChangeReservation`	all the data the user has entered	Forwards the data to the reservation object of the domain kernel for an update. In case of failure, returns an error code
`ActCancelReservation`	`Reservation ID`	Calls the cancel method of the reservation object
...	...	...

Consequences

- *Decoupling:* The `ConcreteActions` decouple the action specific code from the control code of the user interface. They hide differences of the actions behind a uniform class interface. Thus it is easy to implement a generic subsystem to control the user interface. The Action Object hides the specific domain kernel interfaces from the rest of the user interface. This is especially useful when the `AbstractEventInterpreter` follows a data driven approach, because you have only one single abstract method call for all actions. The `AbstractEventInterpreter` needs no knowledge about the semantics or even the parameters of the action.

- *Number of Classes:* Using a separate class for every action may lead to more than hundred primitive subclasses of `Action` that have no further structure. In the variants section, we discuss ways to deal with this drawback on the expense of decoupling.

- *Maintenance:* Adding a new domain level action means to add another `ConcreteAction` but does not affect any other code. Reusing an existing action in another form means only to extend the `AbstractEventInterpreter` for that form. An additional field on a form involves a change of the respective `AbstractEvent`. If the domain level interface does not support the corresponding extra parameter, you also have to change the corresponding `ConcreteAction`.

- *Uniform reaction:* There is one `ConcreteAction` for every domain task instead of one call for every possibility to invoke this task from the user interface. Hence, the pattern enforces a uniform reaction to user inputs.

- *Action Object and Command:* The pattern resembles the Command pattern [GHJ+94, p. 233]. However, in most domain kernels there are separate `Command` classes that address undo operations and coordination of control. In these cases the `Command` objects take the role of the `Actors` in Figure 8. You may consider the `ConcreteActions` the user interface part of a Command. This separation allows several interfaces to work with the same domain kernel. A batch interface or a legacy system interface may use the `Command` objects of the domain kernel without taking details of the user interface into account.

Implementation

- *Action Object without polymorphism:* If you are using a 3GL, such as COBOL or C, you may choose from two implementation strategies for the action objects. One approach is to provide a large case statement with the `ActionId` as discriminator. You may nest levels of case statements to structure the actions. Another approach is to use function pointers as `ActionId`. This emulates polymorphism and avoids the large case statement. However, an erroneous function pointer may cause hard-to-detect errors. Both approaches suffer from a maintainability problem: Every new action enforces recompilation of the complete case statement or of the `AbstractEventInterpreter`, respectively. This may cause considerable turnaround times during system construction.

- *Which parameters to pass?* The easiest way is to let an `AbstractEvent` hold exactly the parameters the `ConcreteAction` needs. However, this solution forces changes of both parts when you have to add further fields to the signature of the action. You achieve better maintainability if the `AbstractEvents` carry all data they can get. The `ConcreteActions` use the data they find and apply reasonable defaults to all other parameters. This strategy results in good decoupling but buries the risk that a `ConcreteAction` uses erroneous default values.

- *Passing parameters:* There are three ways to retrieve the parameters for an action. In the solution section we have already shown the `AbstractEventInterpreter` passing anonymous objects using `AbstractEvents`. Another possibility is to let the `ConcreteActions` call other parts of the system to determine their parameters. For instance, the `ConcreteAction` may call the `LogicalForm` to retrieve information. However, be cautious not to mingle the responsibilities of the `ConcreteAction` with those of the `AbstractEvent`. In COBOL environments you also have to avoid recursive calls. A third alternative is to enable the `ConcreteActions` to open new forms and to ask the user for information. This way works only if the forms can return results to the `ConcreteActions`.

- *Finding Actors:* To perform the action a `ConcreteAction` has to know the `Actor`. There are several alternatives where to get a reference. The most simple case is to pass a reference as a parameter. This approach is appropriate, for instance, if the Action Object is part of a Model-View-Controller [BMR+96, p. 125]. Another way is to pass a domain level key. To retrieve the reference the `ConcreteAction` calls a domain kernel function and passes the key. Finally, the `ConcreteAction` may address a root object, which is a globally known Singleton, and then navigate to the actor. This approach is reasonable to address other parts of the user interface.

- *Iterating over several instances:* If the user interface allows multiselection in any way, you have to decide, which subsystem is responsible for iterating over the selection. There are two alternatives: The `ConcreteAction` iterates the set and submits several kernel calls, or the domain kernel supplies set operations. The former simplifies the domain kernel interface, because the domain kernel only has to provide single object operations. However, set operations frequently have different constraints and semantics than single object operations. While changing the data of a single customer is straightforward, a change of several customers may be sensible only for certain attributes. Transaction oriented systems often bracket the domain level action with a transaction. Hence, iterating in the `ConcreteAction` invokes a separate transaction for every item. Another advantage of domain level set operations is the possibility to offer this service to other interfaces, too. A batch interface or a legacy system may also need set operations.

Variants

A common variant is to merge the `ConcreteActions` and the `AbstractEventInterpreter`. Here, the interpreter initiates the appropriate activities using a State pattern [GHJ+94, p. 305]. This avoids the large number of separate `ConcreteActions` but causes a closer coupling between the domain kernel interface and the analyzer.

Another variant is to integrate the actions as methods into the Logical Form. Every Logical Form provides a set of methods to perform all the actions available on that form. This variant structures the available actions and avoids a plethora of primitive classes. However, in this approach starting an action is not a generic task anymore. The `AbstractEventInterpreter` needs to know the names of the action methods and needs recompilation for every new action. Though this does not seem to be very attractive, it is a good solution in a 3GL environment, where you have to tackle the recompilation problem anyway. To avoid duplicate code, identify frequently occurring actions and put them in a separate module.

Related Patterns

Individual Instance Method [Fow97, p. 106] discusses similar issues on analysis level. You may use AbstractInterface [Col96] to encapsulate creation of `ConcreteActions`.

2.5 Further Pattlets

Logical Form

Context

Most operating systems and frameworks offer an easy-to-use interface for controls and form layout. This interface leaves many options the user code has to specify. The system needs a representation of only the contents of a form.

Problem

How do you implement layout control without cluttering the user interface with layout details?

Solution

Wrap every form with a `LogicalForm` class. This class is responsible for all layout aspects and other layout related tasks, such as focus control. It provides a set of methods to set the fields of the mask. The protocol of the Logical Forms use domain level data types, such as `SocialSecurityNumber` or `ZipCode` instead of technical types, such as integer and string.

General Action

Context

The user interface provides certain actions that are valid in most conditions. Calling for help or aborting a dialog are two examples. The operating system processes some of these actions but others are application specific.

Problem

How do you provide consistent processing of these general actions without duplicating code?

Solution

Provide a central component within the Dialog Control that handles requests common to all dialogs.

Control Dispatcher

Context

You are using an environment, such as CICS, which encourages an architecture with a separate executable for every transaction. Having General Actions this approach forces you to implement the General Actions in every executable.

Problem

How do you avoid code duplication for processing ?

Solution

Provide a single executable, the `ControlDispatcher`, that the environment of the application calls. A parameter, such as a transaction id, specifies the request. The `ControlDispatcher` forwards requests to an analyzer that dispatches the calls to the appropriate transactions or domain level objects. This analyzer also handles the General Events.

Related Patterns

Consider using Abstract Events and Dialog Automaton to implement the analyzer. Consider Action Object to access the domain level part.

Usually the analyzer needs information about the state of the dialog to analyze a request. TerminalSession provides a way to maintain state information even if the environment supports only stateless protocols.

Terminal Session

Context

The user interface runs on top of a stateless server, such as a transaction monitor or a web server. Every user session is in a different state, which you have to maintain.

Problem

How do you handle the state of individual sessions?

Solution

Associate a logical Terminal Session with every physical connection of the server. Identify the session with either the physical id of the terminal or with a session token, transferred as a hidden field. Maintain the state information of every terminal session in an area that is persistent over several transactions.

Form Comparator

Context

You are using Abstract Events. The input device does not notify the interface control of every single action that a user performs on her terminal. Instead, the server receives the modified form contents when the user presses the confirmation button.

Problem

How do you determine the Abstract Events the user has initiated?

Solution

Keep a before-image of every form, i.e. its contents prior to user modifications. When the user submits changes, compare the new form to the before-image to determine which fields have changed. Create the corresponding Abstract Events. Use a generator or a reflective approach to minimize implementation effort for large systems.

Dialog Category

Context

The dynamic structure (i.e. the states and the possible transitions) of several dialogs is identical. The dialogs vary only in the data they process or in the screen layout.

Problem

How do you avoid code duplication when implementing dynamically equivalent dialogs?

Solution

Organize the dialogs of your user interface into dynamically equivalent categories. Specify the actions the user may invoke abstractly. For a concrete dialog supply the following information: the dialog category it belongs to, the concrete actions corresponding to the abstract specifications, and the concrete screen layouts of the participating forms.

Use Dialog Automaton to get a natural implementation of Dialog Categories: Define one table for every category. The table you instantiate for a certain dialog determines its category.

3 Acknowledgments

This pattern language would not have been possible without the profound know how and experience, Andreas Hess and Uli Zeh provided. We mined additional information from documents written by Jan Bis, Thomas Kunst, Martin Reichert, Jordi Romano, Thomas Salzberger, Johannes Siedersleben, and Markus Zobel. Ernst Denert also provided helpful input.

Klaus Renzel and Günter Palfinger reviewed earlier versions of these patterns. Wolfgang Keller urged our last rewrite (while eating all of Jens’ chocolate cake) with many valuable suggestions to improve readability.

Our EuroPLoP shepherd Joe Yoder gave valuable hints to improve the structure of the paper and the introduction.

4 References

[BMR+96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal:** *Pattern-Oriented Software Architecture - A System of Patterns;* John Wiley & Sons Ltd., Chichester, England, 1996; ISBN 0-471-95869-7

[Bis96] **Jan Bis:** *TLR-WGLV - Entwicklerhandbuch Teil II, Systemkonstruktion;* sd&m GmbH & Co.KG, 1996

[Col95] **Dave Collins:** *Designing Object-Oriented User Interfaces;* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1995; ISBN 0-8053-5350-X

[Col96] **Jens Coldewey:** *Decoupling of Object-Oriented Systems - A Collection of Patterns;* sd&m GmbH & Co.KG, Munich, 1996; available via http://www.sdm.de/g/arcus/

[Den91] **Ernst Denert:** *Software-Engineering - Methodische Projektabwicklung;* Springer-Verlag, Berlin Heidelberg New York; 1991, ISBN 3-540-53404-0

[Gam92] **Erich Gamma:** *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge;* Springer-Verlag Berlin Heidelberg New York, 1992; ISBN 3-540-56006-8

[GHJ+94] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:** *Design Patterns - Elements of Reusable Object-Oriented Software;* Addison-Wesley Publishing Company, Reading, Massachusetts, 1994; ISBN 0-201-63361-2

[Fow97] **Martin Fowler:** *Analysis Patterns- Reusable Object Models;* Addison-Wesley Publishing Company, Reading, Massachusetts , 1997; ISBN 0-201-89542-0

[KuR96] **Thomas Kunst, Martin Reichert:** *Architecture - DD35 Sidepanel Software;* sd&m GmbH & Co.KG, Munich, May, 28th 1996

[MNR97] **Robert C. Martin, James W. Newkirk, Bhama Rao:** *Taskmaster: An Architecture Pattern for GUI Applications;* C++ Report, March 1997; SIGS Publications, Inc, New York,NY; ISSN 1040-6042; available via Internet at http://www.oma.com/PDF/taskmast.pdf

[RCo95] **Jordi Romano, Jens Coldewey:** *SIGMAPLAN Gesamtarchitektur - Bedienoberfläche;* Siemens AG, sd&m GmbH & CoKG, Munich, 1995

[Rnz96] **Klaus Renzel:** *Error Handling for Business Information Systems;* sd&m GmbH & Co.KG, Munich, December, 12th 1996; available via WWW at http://www.sdm.de/g/arcus/

[SaZ96] **Thomas Salzberger, Markus Zobel:** *Dialoge - Vorstellung eines Konzepts;* sd&m GmbH & Co.KG, Munich, June, 16th 1996

[Zeh88] **Uli Zeh:** *Grobspezifikation der Basisfunktionen Thyssen/IAB;* sd&m GmbH & Co.KG, Munich, 1988