Hans-Werner Six, Josef Voss

A Software Engineering Approach to User Interface Development
A Software Engineering Approach to User Interface Development

H.-W. Six and J. Voss
Praktische Informatik III
FernUniversität Hagen
P.O. Box 940, W - 5800 Hagen, Germany
email: six@fernuni-hagen.de fax: (+49) 2331 804 317

Abstract

To guide user interface construction, concepts are needed that provide a conceptual basis for modeling, abstract notation, and implementation of tools and concrete interfaces. In this paper, we discuss how general software engineering principles apply in this context. Following these principles, we have developed an object-oriented user interface framework called DIWA which consists of a design model enhanced by a declarative language for the specification of dynamic dialog behavior. We present the main components of the DIWA framework and provide an extended discussion of the underlying design rationales setting them against revealed software engineering principles.

Keywords: User Interface Management Systems, Object-Oriented Design, Software Design Principles

1 Introduction

The development of high quality user interfaces is a time consuming task, especially in the case of direct-manipulation interfaces. Since the beginning of the 80's numerous tools with varying levels of abstraction have been developed to support user interface construction. However, they are not very satisfactory yet. Low level toolkits overload the designer with too many details, high level user interface development systems (UIDSs) suffer from a lack of flexibility and comprehensiveness.

The difficulties in constructing appropriate tools at a sufficiently high level of abstraction are mainly due to the inherent complexity of direct manipulation interfaces. To illustrate the complexity we want to sketch three facets of the problem:

Concurrency: The user can arbitrarily change focus from one activity to another, e.g. interrupt a form-filling activity to inspect another source of information, e.g. by performing a database query, and eventually return to the first activity. This kind of interaction is called "multi-threaded dialogs".

Dependencies: Consider a tool for database design consisting of an editor for entity-relationship diagrams in one window and another window presenting the relation tables automatically generated from the ER-diagram. The relation window should always reflect the state of the ER-diagram.

Semantic feedback: Consider the same example and the user connecting entity symbols and relationship symbols by drawing lines with the mouse. The rubber-band following the mouse should ideally reflect the final layout of the connection, i.e. a rectilinear line without intersections with other lines and symbols.

From the developers point of view, user interface construction has at least two dimensions: screen layout and dynamic behavior.

To guide user interface construction, concepts are needed that provide a conceptual basis for modeling, abstract notation, and finally, for the implementation of tools as well as concrete interfaces. Basically, such a conceptual model must be able to model screen layout, dynamic behavior and naturally lead to an appropriate software structure. Moreover, to efficaciously reduce complexity, the model must provide few simple concepts which are still powerful enough to enable the UI developer to cope with complex direct manipulation user interfaces.

As far as user interface software structure is concerned, the logical separation (different modules or processes) of the user interface from the functional core of an application and the decomposition of the user interface (UI) into smaller components are widely accepted basic ideas. We will refer to the UI components as "user interface objects" (UIOs) whereby the term "object" refers to object-oriented design principles. A UIO software component encapsulates a certain part of the dialog behavior and a corre-
sponding part of the screen layout. Very naturally, the
heart of a conceptual model centers around an abstract
specification of UIOs.

In this paper, we take a software engineering perspective
to the user interface development problem. To this end, we
discuss how general software engineering principles and
rules apply to the user interface field. To make our consid­
erations concrete, we introduce DIWA, a framework for
the design of user interfaces, which comprises:

- an object-oriented hierarchical model with emphasis on
  homogeneity of components and restricted communica­
tion between components,
- an integrated language for the specification of dialog be­
  havior which is implementation independent, declarative
  and executable.
- an extensible class library containing building blocks of
  various levels of abstraction, and
- a simple user interface development system.

The main purpose of this paper is the discussion of the de­
sign rationales behind the DIWA system. In the next chap­
ter, we illustrate the basic design principles that guided our
development. In chapters 3 and 4, we present the main
components of the DIWA system - the design model and
the language for dialog behavior. Chapter 5 comprises an
extended discussion of the DIWA design concepts setting
them against the principles from chapter 2. The paper clos­
es with a description of the current implementation status
and some ideas for future work.

2 Software Engineering Principles for User
Interface Construction

We start this chapter with a classification of problems oc­
curring in UI design. Then, based on this classification we
reveal software engineering principles which should guide
UI design. Among the specific problems imposed by the
complexity of direct manipulation user interfaces the fol­
lowing are most important:

- **Topology of structure**: User interfaces decomposed to
  many UIOs tend to have rather complex interconnections
  because of the high degree of dependencies between
  UIOs. These dependencies do not obey simple structures
  like pipelines or hierarchies.

- **Nature of components**: UIOs appearing in interactive
  applications, like windows, scrollbars, forms, or application
  specific graphical objects, considerably differ in their ren­
dering and behavior. Is it possible and reasonable to identi­
fy communalities between UIOs or are specialized UIOs
more appropriate? Specialized components however, tend
to restrict their composability.

- **Separation of tasks**: The main tasks of a user interface,
namely input/output processing, data conversion and data
transfer between application and user interface, are closely
related. For example, user input often refers to objects dis­
played on the screen. Main question is how and to what
extent can a separation of tasks be achieved.

- **Concurrency of components**: Multiple threads yield con­
currently working components (e.g. event handlers). Main
problem is the reduction of concurrency to as few com­
ponents as possible.

- **Communication between components**: Clearly, asynchro­
nous communication is required in many situations but
tends to be slow and is difficult to manage. It is important
to identify situations where synchronous communication
suffices. However, can such a mixture of synchronous and
asynchronous communication be managed in a reliable
manner?

What design alternatives are to be selected should be guid­
ed by established software engineering design principles.
The rationale of any approach based on software engineer­
ing principles should be to keep concepts as simple as pos­
sible without restricting the expressiveness such that
realistic situations can still be modeled adequately. Be­
sides our general assumption of an object-oriented envi­
ronment, we claim the following design principles to be
appropriate:

- **Hierarchical structure**: The hierarchical arrangement of UIOs is widely accept­
ed. It yields a manageable structure and nicely corre­
sponds to geometric containment of the UIOs' screen
rendering.

- **Client server architecture**: A client server architecture provides for a clear separa­
tion of tasks. It applies to various levels, i.e. among
UIOs and between user interface and application. To­
gether with a hierarchical structure the client server ar­
chitecture imposes a clear, non-cyclic call structure.

- **Homogenous components**: This principle corresponds to the simplicity of concepts.
Small UIOs like simple buttons should be treated in the
same way as large composite UIOs like graphical edi­
tors. All interfaces between UIOs should basically have
the same protocol. Besides simplicity of concepts, ho­
mogeneity of components is a prerequisite to compos­
ability.
Local responsibility of components
A UIO should work locally, i.e., with minimal knowledge about other UIOs. Interdependencies are organized by the concept of responsibility [24]: A UIO is responsible for as few (subordinate) UIOs as possible.

Restricted Communication
Due to the complex structure, especially to the concurrency of UIOs, communication between UIOs should be restricted to simple patterns: Top Down communication whenever possible, asynchronous communication between as few components as possible.

Identification of universal properties
In an object-oriented context it is advisable to identify universal structures and behavior of UIOs in order to define them as abstract classes. The success of this procedure clearly depends on the separation of independent tasks of UIOs.

3 The DIWA Architecture
To make the previous discussion concrete, we introduce an object-oriented user interface framework called DIWA whose basic concepts follow the design principles explained above. DIWA consists of a conceptual model enhanced by a declarative language for the specification of the dynamic dialog behavior. At the moment, we are working on the integration of a screen layout description technique but this topic will be excluded from further discussions.

In this chapter, we illustrate the conceptual model while in the next chapter the dialog language will be presented. The global structure of a user interface according to the DIWA model is a hierarchy of UIOs. UIOs are organized in classes. We first describe our notion of UIO, i.e., the task and the structure of a DIWA UIO.

3.1 The structure of a DIWA UIO
The decomposition of a DIWA UIO into its three parts - dialog control, presentation and application interface - and the interconnections between these components is illustrated by figure 1. In particular, the components have the following tasks:

- The dialog control component makes all decisions concerning the course of the dialog, i.e., possible sequences of events and the appropriate reactions on them. Events are user inputs, signals from other UIOs and return parameters from calls to presentation and application methods. Typical reactions on an event are, for example, calling the presentation to change the screen rendering of the object, calling the own UIO to re-arrange subobjects, calling application methods or sending signals to (dialog controls of) other UIOs.

- The presentation component is responsible for drawing the object. Additionally, it arranges its subobjects (direct descendents of the UIO). Arranging a subobject means computing its shape and position. The presentation uses data it gets by method calls to the application interface.

- The application interface (AI) of a UIO provides the presentation and subobjects with suitable access to data. The AI has access to data of the parent's AI (via method calls) or it directly calls functions of an application process. An important task of this component is the support of semantic data and application specific functionality inside the user interface. This is essential to support semantic feedback. Otherwise semantic feedback would require intense communication between user interface and application. This is not feasible, especially when user interface and application are running in different processes possibly on different machines.

- The application - also referred to as functional core - resides in its own process outside the UI. In general, any UIO can have its own application process. However usually there is one application process which is connected to the application interface of the root UIO.

We use a generalized form of application function with the following characteristics:

- Application functions return results asynchronously.
- A single function call may return a varying number of results. These are stored in a shared queue and can
be treated by the user interface, i.e. the application interface component that initiated the call, as soon as they arrive.

Summarizing, the roles of the internal components of an UIO can be characterized as follows:

- The dialog control is the only asynchronously working component. It makes use of services provided by the presentation and application interface.
- The presentation provides services to its dialog control and makes use of services provided by the application interface.
- The application interface provides services to several components: dialog control and presentation of the UIO it belongs to, and to application interfaces of subordinate UIOs. It makes use of services from the application interface of the parent UIO.

On the other hand, the external interface of a DIWA UIO consists of two parts:

- Events for control flow: A UIO understands certain user events and signals from other UIOs. Furthermore, it sends certain signals to other UIOs. However, we restrict the exchange of signals between UIOs to parent child communication.
- Message calls for data flow: A UIO sends messages to its parent UIO or to an application process to deliver or receive data.

### 3.2 Abstract Notation of UIO Classes

To support the development of user-interfaces according to the DIWA architecture we have invented a high level description language to define UIO-classes. This language renders that only the presentation and application interface of a UIO have to be programmed. Object structure and dialog behavior can be expressed in a high level, programming language independent style.

DIWA is an object-oriented system and supports multiple inheritance, i.e. a new class is defined as subclass of one or more existing classes. The most basic predefined class, residing at the root of the inheritance lattice, is the class DialogObject.

As an example, we present a class realizing horizontal scrollbars. The (simplified) object structure of a scrollbar consists of three subobjects - the slider and two buttons for left shifts and right shifts - as illustrated by the following figure:

![Scrollbar Example](image)

A corresponding class definition in the DIWA description language is:

```plaintext
Scrollbar class [ DialogObject ]
  subobjects: [ Slider: Button, LeftArrow: Button, RightArrow: Button ]
  eventhandlers: ...
  presClass: ScrollBarPresentation
  appClass: ScrollBarAppInterface
end Scrollbar
```

- In the first line the name of the class and a list of its superclasses is given. In our example DialogObject is the only superclass.
- The subobjects section describes the UIO hierarchy, i.e. the list of subobjects of a Scrollbar instance. Each subobject in the list - in our example, Slider, LeftArrow and RightArrow are the subobjects of a Scrollbar instance - is specified by its name and its type (the name of another UIO class).
- The eventhandlers section describes the dynamic behavior of instances of this class, i.e. the dialog control component of the UIO. This task will be explained in the next section.
- The presClass entry contains the name of a class that implements the presentation of instances of the Scrollbar class. This presentation class has to be implemented separately using an object-oriented programming language that depends on the implementation environment.
- Analogously, the appClass entry gives the name of a class that implements the application interface of instances of the Scrollbar class.

### 3.3 Protocol of UIOs

Since user interfaces require dynamic modifications a UIO hierarchy is not static. To impose rules on the construction and modification of UIO structures we allow ordered sets of subobjects to appear in the subobject list of a UIO class. These sets determine legal structure modifications: During run-time subobjects may only be added to or deleted from a subobject set.

The protocol of a UIO as a whole mainly comprises operations to modify or re-arrange its subobject structure. The
protocol reflects the local responsibility of a UIO. Some examples of the operations are:

subobjToTop (subob)
Re-arrange the subobjects such that the given subobject will lie in front of its siblings.
deactivateSubob (subob)
Suspend the given subobject and all its descendents from event processing.
newObInSet (setname)
Create a new UIO and add it to the set of subobjects indicated by setname.
deleteOb (subob)
Remove the given subobject from its set and destroy it.

All operations of the UIO protocol are predefined in the class DialogObject and may not (and need not) be redefined in subclasses.

3.4 Protocol of Presentation Components
A presentation component is mainly concerned with two (related) tasks: The determination of positions and shapes and the actual screen rendering of the UIO it belongs to. The computation of shape and position of a UIO is distributed between the presentation of that UIO and its parent’s presentation. In most cases the parent presentation computes the children’s layout to which each child agrees. In some cases a more flexible protocol is of advantage:

A parent’s presentation computes the position and shape of a child and delivers it to the child’s presentation which takes these data as a basis on which it computes its own position and shape.

As an example for a shape that is determined by the child we consider a menu UIO. The menu shape - width and height - depends on the number of items (and eventually on the maximum length of the items). The menu itself, not its owner, computes the shape because it has the necessary knowledge about the menu items. The position of a menu, however, is usually determined by the menu owner. The menu itself may not even know whether it is used as a popup or static menu.

Some examples from the presentation protocol are:

display
Draw this UIO. The method assumes that a certain portion of the screen is available. Usually, it ignores that parts of this region may be obscured by other UIOs.

setPosition (point): point
Determine the origin of the screen area of this UIO expressed in the coordinate system of the parent UIO.

3.5 Protocol of Application Interface Components
Besides data exchange and manipulation, a typical task of an application interface is the mapping of UIOs to application objects. Consider, for example, a PolygonEditor UIO maintaining a set PolySet of Polygon subobjects. Each Polygon UIO corresponds to a polygon object in the application. Changes on either side have to be communicated between PolygonEditor and the application. The application interface of the PolygonEditor has to perform the mapping between UIO ids and polygon ids of the application.

Some examples from the application interface protocol are:

registerNewObInSet (setname, subob)
Register a new subobject created by newObInSet, i.e. the application interface takes precautions to forthcoming requests of the new subobject.

disableOb (setname, subob)
The given subobject has been destroyed (via deleteOb).

All operations of the presentation and application interface protocol are implemented by the user interface designer. However, a number of generic operations provide default versions that can be inherited from the predefined classes Presentation and Application.

4 DIWA Dialog Descriptions
Among typical specification techniques for dialog behavior (e.g. context-free grammars [16], state-transition nets (STN) [9], event handlers [5], event-response-systems [8]) the event handler concept appears to be the most flexible technique. In DIWA, one or more event handlers can be associated with a UIO. The corresponding class definition contains a description for each event handler. Thus, in contrast to other approaches [2], [5], each event handler is bound to an object.

The syntax for event handlers represents a textual (non-graphical) notation of STNs which has been enriched such that events and reactions on events can be specified precisely. Instead of enriching the expressive capabilities of state transition nets by adding complete programming language power (resulting in so called ATNs), our approach tries to preserve a symbolic notation and a high level of abstraction.

The main additional feature is the concept of object expressions appearing, for example, in object-related events like moving the mouse into a certain object, or as parameters of presentation method calls, e.g. to graphically invert a certain object. Thus, our notation enables the precise and high level specification of expected events and reactions on event arrivals, i.e. changes in dialog state, changes in object properties and calls to application functions.

To give an example, we enhance the scrollbar class from above by adding an event handler that manages slider movements. Once a movement is finished, the event handler emits a changed signal which will be received by the parent UIO that will then initiate an appropriate update of its presentation.

Scrollbar class [DialogObject]

subobjects: [Slider: Button, LeftArrow: Button, RightArrow: Button]
eventhandlers: [SliderMoveHandler]

SliderMoveHandler: eventhandler

stateDesc:

Waiting =
    ( LeftDownln(Slider) → initMove to pres ) → Dragging 

Dragging =
    ( MouseMove → moveToCursor to Slider → Dragging 
      | LeftUp → save to pres; ) → Waiting
      | signal changed ) → Waiting

end SliderMoveHandler

presClass: ScrollBarPresentation
appClass: ScrollBarAppInterface
end Scrollbar

- The eventhandlers section contains a list of names for all event handlers associated with instances of this class. Each event handler is specified in its own section. The Scrollbar class contains SliderMoveHandler as the only event handler.
- An event handler definition has a stateDesc entry which contains a description for each state of the event handler. Note that at any time exactly one state is active, i.e. is able to receive an event. In our example, we have two states, Waiting and Dragging. Each state is defined by one or more reaction schemes separated by a vertical bar.
- A reaction scheme consists of an event, a list of reactions to that event and the name of the state the event handler will enter after that event has been received and reacted upon. Consider, for example, the Waiting state with one reaction scheme. The event LeftDownln(Slider), i.e. the left mouse button has been pressed inside the slider, will cause one reaction - a call of the presentation method initMove. The next active state is Dragging.

We have mentioned that more than one event handler can be associated with a UIO. These handlers are again hierarchically organized to control their cooperation. Additionally, the inheritance of event handlers follows certain rules. Due to space restrictions, we cannot go into further details. The interested reader is referred to [20].

5 Design Rationales

In this chapter, we discuss the basic design decisions made in the DIWA framework in the light of the software engineering principles listed in chapter 2. For this purpose, we point out some problems that do not have an obvious solution and consider alternatives to our design decisions.

5.1 External vs. Internal Control

As an application of the hierarchical structure and the client server architecture principle, DIWA provides a purely external control strategy. However, it is frequently argued [3], [2] that purely external control is not feasible. One argument against external control is that the application may require additional information from the user or detect an error situation at times not predictable for the user interface. Another argument is the maintenance of changes in the structure of the UIO hierarchy according to changes in application data.
Asyncronous requests from the application

We believe that in a thorough UI design all kinds of possible asynchronous requests from the application must be taken into account. However, the time when such a request occurs is unpredictable.

The two possibilities to handle an asynchronous request are an event handler waiting for an event from the application or an event handler waiting for a return value from an application function. From the user interface point of view both versions are equivalent. However, the latter fits better to the DIWA approach because it preserves the client server principle. Actually, the DIWA update mechanism follows the external control strategy by performing the following steps:

1. An event handler of some UIO detects the request based on the return value of an application function.
2. If other UIOs may be concerned as well the event handler sends a signal to the parent UIO.
3. One or more UIOs send update messages to their application interfaces and presentations.

This mechanism is similar to the MVC update mechanism [10] but is shifted from the application into the user interface.

The functional core has to be programmed accordingly. For instance, for the implementation of an event driven simulation the application programmer has to consider user interface calls, i.e. requests for data to be displayed, user requests to interrupt a simulation or to change parameters at virtually any time. From the perspective of the application, these calls appear as events.

Influence of the application on the UIO structure

Although the external control concept does not allow the application to know about the structure of the user interface, the UIO hierarchy must be able to change according to changes in application data. Changes in UIO positions and shapes are rather straightforward following the update strategy from above. However, creation and deletion of UIOs as a result of an application change must be performed as well.

Consider for example the PolygonEditor. Application data determines what subobjects are contained in PolySet. To this end, the PolygonEditor application interface has to provide two methods that report the necessary changes in the set of polygon subobjects. These operations are further examples of the application interface protocol:

```
subobsToDeleteFromSet (setname): list of subobs
Compute a list of elements from the given set that are
to be deleted according to the latest change of application data.

numberOfNewObsInSet (setname): integer
Compute the number of new subobjects to be created
and inserted into the given set according to the latest
change of application data.
```

The PolygonEditor UIO processes the actual creations and deletions by an updateSubobStructure operation which is another generic operation in the UIO protocol:

```
updateSubobStructure (setname)
Create and/or delete subobjects from the given set.
```

Necessary informations are provided by the application interface via the above two methods.

The creation and deletion of more complex UIOs like windows is performed in the same way.

From the discussion it should become clear, that the external control strategy following the hierarchical structure and client server principle works well even in a complex user interface setting. In our opinion, the basic problem with external control seems to be a psychological one, namely, that the functional core does not determine the course of the action any more, and is actually implemented as a set of functions/methods that are called in an "unpredictable" order.

5.2 The Nature of Dialog Control

DIWA event handlers deal with the behavior of UIOs at various levels of abstraction - from low level syntactic to application specific behavior. We present some examples illustrating the structural similarities between the different levels of behavior. Extracting these similarities allows to use the same description language on different levels.

As an example for simple syntactic behavior consider the following event handler to detect double clicks with the middle mouse button when the cursor is inside a subobject:

```
MiddleDClick: eventhandler
stateDesc:
Waiting =
( MiddleDownln (allsubs) \rightarrow nop \rightarrow Between )
Between =
( timer 300 \rightarrow nop \rightarrow Waiting |
  MiddleDownln (allsubs) \rightarrow signal MiddleDClick (eventob) \rightarrow Waiting )
end MiddleDClick
```

The event handler MiddleDClick is defined in the basic UIO class WithDoubleClicks. Other classes re-use this be-
behavior by adding WithDoubleClicks to their superclass list. An example for a more elaborate syntactic behavior is the cooperation of a Scrollbar UIO with a PolygonEditor using the scrollbar. Fig. 3 shows how these UIOs exchange signals and send messages to each other in order to communicate the effect of a scrolling operation. The sequence of actions is indicated by numbers.

![Fig. 3](image)

Data exchange is accomplished by messages to the application interface of the PolygonEditor UIO. The exchange of signals is handled by the SliderMoveHandler of the Scrollbar UIO (cf. chapter 4) and the UpdateHandler of the PolygonEditor UIO:

```plaintext
PolygonEditor class [DialogObject]
  subobjects: [PolySet: setOf Polygon
                   Scrollbar: Scrollbar]
  eventhandlers: [UpdateHandler]
  UpdateHandler: eventhandler
    type: simple
    stateDesc:
      (changed from Scrollbar \rightarrow
       update to pres;
       moveSubobj to PolySet)
  end UpdateHandler
  presClass: PolygonEditorPresentation
  appClass: PolygonEditorAppInterface
end PolygonEditor
```

The update call lets the presentation get the new start position from the application interface (via getStartPos). The moveSubobj call initiates the re-positioning of the polygon subobjects.

As an example for application specific behavior, consider dependencies between the ER window and the relation window mentioned in the introduction. The whole application corresponds to a UIO class DbDesigner defined as subclass of the predefined class WindowManager. DbDesigner introduces an additional event handler that receives changed signals from the ER-window and reacts on these signals by sending an update signal to the relation window. Information about the kind of change is communicated via the application interface of the DbDesigner UIO.

The examples show that there is a continuum of more or less application specific behavior. As a consequence, purely syntactic aspects should not be treated conceptually different from higher levels, i.e. application specific aspects of dialog control. The appropriate mechanism to separate different levels of abstraction is inheritance of classes following the basic rules:

- Low level syntactic dialog behavior should be handled in basic classes.
- More application specific behavior should be handled in derived classes.

Summarizing, DIWA dialog control components gather structurally similar tasks that other approaches [3], [10] distribute between presentation, dialog control and functional core. Thus, a more homogenous structure and clearer separation of tasks is achieved.

### 5.3 Communication between Dialog Control Components

In the DIWA framework, the principle of top down communication is widely applied. Messages strictly follow this pattern. Signals between dialog control components, however, may as well be sent from a child to its parent UIO, i.e. violate the top-down principle. To have a closer look at the situation, we have to distinguish synchronous signals, i.e. those blocking the sender until an answer from the receiver has arrived, and asynchronous signals that do not require an answer. The latter do not cause problems; the receiver treats them like user events.

However, synchronous signals from a child to its parent may be more dangerous. Hence, we restrict their use to few situations. Consider for example a UIO requiring that its siblings are not active for some time. The child sends a corresponding signal to its parent UIO and waits for the answer which indicates that the siblings are actually deactivated. These situations which do not appear too often, can be characterized as follows: A UIO cannot proceed until some coordination has taken place. The responsibility for the coordination is located at the parent or some other superior UIO.

Summarizing, the restricted communication principle is realized by top down communication for messages and almost top down communication for signals. The exceptions
from this rule are as few as possible.

5.4 Separation of Tasks between Presentation and Dialog Control

An important difference between DIWA and approaches like PAC [3] or Serpent [1] is the role of presentation and dialog control with regard to event handling and data exchange.

One point of discussion is whether the presentation should handle - besides output - user input events, too. It is argued that separating is not appropriate because input and output handling are tightly coupled and input should not be treated by a dialog control component because it is media dependent [4]. However, the DIWA event handler descriptions show that user input can be described media independent and yet detailed, and that a separation is reasonable and feasible. DIWA event handlers use symbolic names - mostly generic - to issue presentation operations. The implementation of these operations is completely left to the presentation.

The most important point concerns the communication of components. The DIWA approach - all events are handled by the dialog control - guarantees that a presentation does not exchange events with other components, i.e. communication between components is considerably simpler.

Another issue is data exchange between UIOs. DIWA dialog control components are not involved in data transfer or the mapping of application objects to UIOs, and hence, are not concerned with application specific data formats. Data transfer only takes place between presentation and application interface components. Besides a clearer separation of tasks, this property is a prerequisite for abstract specifications of dialog control, i.e. dialog behavior.

6 Related Work

We moved this section to the end of the paper because we feel that after the illustration and discussion of the DIWA concepts the understanding of this topic is easier for the reader.

A first attempt to structure user interface software has been presented in the well-known Seeheim model [17]. It suggests a global decomposition into three parts - presentation, dialog control and application interface model. The idea of hierarchically composed UIOs with an internal decomposition - appearing in various approaches [7], [3], [10] including DIWA - can be interpreted as a kind of inversion applied to the Seeheim model. Instead of bundling all code relevant to dialog control, presentation and application interface in one large component each, the three aspects - presentation, dialog control and application interface - are locally bundled for each UIO.

Object-Oriented Architecture Models

The Smalltalk MVC model [10] has substantially influenced our approach. MVC shows that it is possible to isolate the functional core from input and output processing. However, our approach extends MVC in several ways:

- Although Smalltalk's view and controller objects hold semantic data there is no component, like DIWA's application-interface, explicitly designated to this purpose.
- In MVC, application specific dependencies between views are maintained in the model object (the functional core). In DIWA we use application specific event handlers inside the user interface.
- The model is not rigorously carried through. Objects like scrollbars and menus are not built according to the MVC model and it is difficult or apply the model to application specific graphics inside a window.

PAC [3] is a conceptual model, that in contrast to DIWA and MVC does not comprise a concrete framework. It suggests a global structure similar to DIWA. As in DIWA, a component capturing semantic data - missing in MVC - is associated with each "interaction object". However, PAC's separation of tasks and, hence, its call structure is considerably different from the DIWA and MVC approaches. The main differences have been discussed in the chapter above.

Most important, both, PAC and MVC, do not provide mechanisms for an abstract specification of dialog behavior.

Object-Oriented Toolkits

We consider InterViews [12] as a typical example of an object-oriented toolkit. Besides the lack of an abstraction for dialog behavior, an important difference to DIWA is that InterViews uses three different categories of UIO classes - interactors (which are able to process user input), graphics and text objects - each with its specific protocol. These categories are furthermore divided into different classes for primitive objects and for objects able to compose groups of objects. In contrast, DIWA user interfaces are build up from homogenous UIOs. The abilities to compose UIOs and to communicate with the user are generic.
properties of all UIOs.

Object-Oriented UIMSs

Tube [7] and Garnet [15] are examples of systems providing abstractions for dialog behavior in an object-oriented environment. The composite object architecture of Tube (influenced by PAC) is similar to the DIWA approach. However, Tubes's mechanism for dialog descriptions - the rule-based language ERL [8] - is less declarative than the DIWA language because it does not provide explicit dialog states and makes use of implicit communication via flags. Garnet uses a fixed set of so-called interactors all running the same STN [14] with three states. A syntax for dialog behavior is not provided.

Neither Garnet nor Tube uses an explicit local decomposition of objects. For instance, Tube holds all aspects of a UIO in slots of one object. However, the behavior part of a Tube UIO - a set of ERL rules - is encapsulated. But there are global ERL modules as well.

Generally compared to object-oriented UIMSs, DIWA preserves the benefits of a toolkit - a transparent and modular architecture - while introducing a higher level of abstraction for dialog behavior descriptions. This is achieved by integrating runtime-facilities into each UIO. Basically, these are facilities for initializing and re-arranging the UIO hierarchy and for the interpretation of dialog descriptions. This means that DIWA preserves local responsibilities because it does not have a global run-time component.

A benefit of Tube and Garnet is their support for layout specification, especially a constraint mechanism. This issue is currently under work for the DIWA system.

7 Implementation Status

The DIWA system is implemented on top of the NeWS window system [6]. At the moment, it provides a Smalltalk-like browser that allows to inspect, edit and create UIO classes including classes for presentations and application interfaces. UIO classes are denoted in the DIWA language and automatically compiled to PostScript classes. Classes for presentations and application interfaces are directly implemented in PostScript.

The class library comprises common UI building blocks like buttons, text fields, menus, scrollbars and windows, but is still under development. We have built sample applications, i.e. more complex UIOs, like a window manager, a polygon editor and an ER-editor. Our experiences show that - although event handlers interpret their descriptions at run-time - response time is very good. This is due to the fact that UIOs live inside the NeWS server, especially, that event handlers are realized as lightweight processes.

8 Conclusion and Future Work

In our opinion, an abstraction mechanism and conceptual modeling is the most important prerequisite to cope with the inherent complexity of user interface construction. From this point of view, we have developed the DIWA framework following established software engineering principles. The design of the framework and experiences with the development of concrete user interfaces show that these principles can be successfully applied to user interface construction.

Our future work concerns extensions of the conceptual model and the development system. In both areas screen layout is the main interest. The existing conceptual model and description techniques will be enhanced by layout specifications, especially declarative notations for graphical constraints, to be associated with UIOs similar to event handlers. The primary extension of the DIWA development system will be a tool for the interactive definition of UIO structures and screen layout. This extension has to be well-integrated with textual layout specifications and the existing browser. We believe that DIWA's separation of tasks provides a sound basis for these extensions.

9 References


[18] Six, H.-W. and Voss, J. DIWA- A Hierarchical Ob-


