User Interface Modelling with UML

Paulo Pinheiro da Silva and Norman W. Paton

Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK
{pinheirp,norm}@cs.man.ac.uk

Abstract. The Unified Modeling Language (UML) is a natural candidate for user interface (UI) modelling since it is the standard notation for object oriented modelling of applications. However, it is by no means clear how to model UIs using UML. This paper presents a user interface modelling case study using UML. This case study identifies some aspects of UIs that cannot be modelled using UML notation, and a set of UML constructors that may be used to model UIs. The modelling problems indicate some weaknesses of UML for modelling UIs, while the constructors exploited indicate some strengths. The identification of such strengths and weaknesses can be used in the formulation of a strategy for extending UML to provide greater support for user interface design.

1 Introduction

UML [10, 2] is the standard language for object-oriented modelling of software applications. The user interface, as a significant part of most applications [17], should also be modelled using UML. In fact, UML is a natural candidate notation for UI modelling. However, it is by no means always clear how to model user interfaces using UML. It is not easy to identify how user interface elements, such as user tasks and presentations, are supported in UML application models. There are few reports on projects specifically applying UML for modelling the UI. Moreover, many modelling problems that appear during the design of a UI, such as in the case study described in this chapter, are not completely addressed by UML-based design methods or by the UML specification.

Many proposals have been made for models that support the design of UI elements, using several different notations. For instance, there is research concerning the design of user tasks, as in Kirwan and Ainsworth [14] and in Johnson [13]. Moreover, there are several proposals for designing UIs using declarative models, as described in Griffiths *et al.* [9] and Szekely [19]. Therefore, it would be best not to have to invent new modelling constructs for the UI if existing ones can be used effectively. Further, it would be good to be able to use the same constructs for the UI as for the rest of the application. Indeed, a single notation could be useful for consolidating the complete design of an object-oriented user interface. There is some research in this area, e.g. Kovacevic [15], but the identification of which UI aspects can and cannot be described using UML is still not clear.

The aim of this paper is to present a summarised description of a comprehensive UI modelling case study using UML. This case study has the purpose of identifying: (1) common UI modelling problems when using UML; and (2) a set of UML constructors and diagrams that may be used by application developers to design UIs. From the modelling problems we can identify some aspects of UIs that are not covered by the UML. From the set of constructors we can identify the aspects of UIs that are covered by the UML. Therefore, the case study produces an insight into the ease with which the UML can be used to model UIs. Moreover, it provides elements that may be used to develop a strategy for extending UML in order to provide better support for user interface design.

The case study intentionally does not describe any method that was used during the UI design. Indeed, many methods were considered in a way to find out alternative modelling approaches, with a view to overcoming difficulties identified. The idea here is to solve the identified UI modelling problems using only UML.

The case study considers only form-based user interfaces. In fact, very important categories of application user interfaces such as database system UIs and web application UIs are mainly form-based. Restricting the scope to form-based user interfaces means that user interfaces for applications

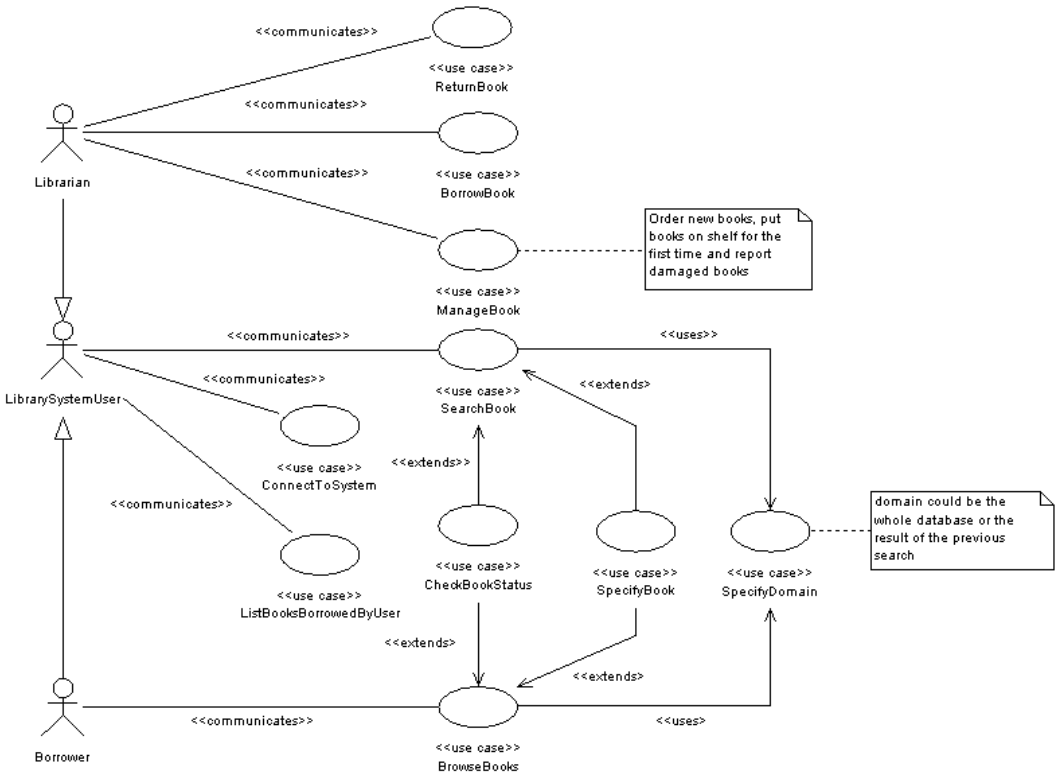

Figure 1: The use case diagram.

such as games, word processors and simulators are not in the scope of this study. Despite this, the UI models introduced in the case study can be used as a UI model baseline for the development of more comprehensive UI models based on UML. Further, the case study considers that modelling the use of visual components (widgets) is more important than the modelling of the widgets themselves for most UI designers. Thus, this case study discusses how widgets may be used by the application, and not how widgets can be modelled.

2 Case Study: The Library System

A library case study is used to identify user interface modelling problems [9]. The Library System in the case study could be considered too simple to catch real problems faced during the modelling of user interfaces. Indeed, the Library System is simple compared with real systems. However, it has proved sufficiently complex to allow a range of problems to be identified in the area of user interface modelling.

A use case diagram in Figure 1 shows the actors of the Library System and their use cases. Actors are *Librarians* and *Borrowers*. The actor *LibrarySystemUser* is a generalisation of *Librarian* and *Borrower*. Librarians use the Library System to manage the book catalogue and the loan records. Librarians only need to inform to the Library System when books are checked into and checked out of the system to be able to manage loan records. Thus, the use cases *BorrowBook* and *ReturnBook*, associated with *Librarian*, are created. *LibrarySystemUsers* can connect to the system, list the books borrowed by a library user and search for books by author, title, year or a combination of these. Thus, the use cases associated with *LibrarySystemUsers* are *ConnectToSystem*, *ListBooksBorrowedByUser* and *SearchBook*. *ConnectToSystem* is considered as a use case since a *LibrarySystemUser* can login to the system just to check his/her password. *Borrowers* can browse the book catalogue without specifying any condition. Thus, the use case *BrowseBooks* is associated with *Borrowers*. Search and browse operations can be repeated over the result of the last search or browse operation.

The Library System must guarantee that only registered users can login to the system. Further, the system must guarantee that borrowers can only perform services associated with borrowers, and that librarians can only perform services associated with librarians.

Books can be selected while users are searching or browsing the book catalogue. Once a book is

selected, users can check its availability. The use case *CheckBookStatus* is modelled as an extension of *SearchBook* and *BrowseBooks*. Indeed, a book must be selected by one of these operations before the user can check its status.

Some use cases have similar features in their behaviour. For instance, *BrowseBooks* and *SearchBook* are both use cases where books can be specified. Thus, a new use case called *SpecifyBook* has been created to model this shared behaviour. Unidirectional associations are created to model the ≪*extends*≫ relationship between *SpecifyBook* and *BrowseBooks*, and between *SpecifyBook* and *SearchBook*. Similarly, *SpecifyDomain* is also a common behaviour of *SearchBook* and *BrowseBook*.

From the use case diagram and the system specification not entirely described in this paper is obtained the design of the *domain model* represented by the class diagram shown in Figure 2. This class diagram is composed of the following ≪*entity*≫ classes: *Person*, *Librarian*, *Borrower*, *Book*, *Loan* and *StockItem*. The three first ≪*entity*≫ classes correspond to the *LibrarySystemUser*, *Librarian* and *Borrower* actors, respectively.

The existence of an instance of *Book* means that the book has an entry in the library catalogue. To manage its stock, the Library System has a *StockItem* class that represents copy versions of the books the library has. However, it is possible that some books in the library catalogue are not in stock, e.g. when newly ordered books have not yet been delivered, or when books are damaged. Indeed, books are inserted in the library catalogue when they are ordered. Thus, the use case *ManageBook* associated with *Librarian* is created.

An instance of *Loan* is created by the process modelled by the *BorrowBook* use case and destroyed in the process modelled by the *ReturnBook* use case. A *Loan* object indicates essentially the day a book should be returned.

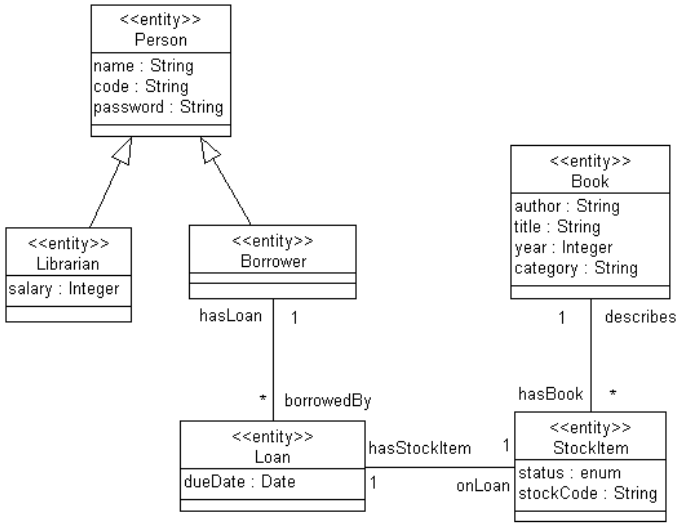

Figure 2: The domain model.

The ≪*entity*≫ stereotype, ≪*control*≫ stereotype and ≪*boundary*≫ stereotype are used throughout this paper. They were introduced by Ivar Jabcobson in his Object-Oriented Software Engineering [12] and incorporated by UML. The ≪*entity*≫ stereotype identifies classes and class instances that model things or objects that exist in their own right. The ≪*control*≫ stereotype identifies classes and class instances that perform system behaviour. The ≪*boundary*≫ stereotype identifies classes and class instances that handle the interaction between system users and systems.

The case study description provides the context for the introduction of the UI design.

3 Task Modelling

The *BrowseBooks* use case in Figure 1 shows that a borrower can browse books. However, borrowers must be logged in to perform any other function. Using UML terminology this means that the actor *Borrower* can only use the *BrowseBooks* use case if he/she previously used the *ConnectToSystem* use case. In fact, the situation is slightly more complex than that. The fact that a borrower used *ConnectToSystem* before does not mean that s/he is logged into the system. This could happen, for example, because the borrower tried to login but failed. The same use case diagram shows that the *CheckBookStatus* use case extends the *SearchBook* and *BrowseBooks* use cases, but it doesn't explain how this extension happens.

The problem described above is that use case diagrams were designed for requirements analysis, but they do not provide control flow information related to tasks. The activity diagram in Figure 3(a) shows how a borrower can interact with the user interface of the Library System. There we notice that activities are not the same thing as use cases, since the activity *Log in* means use of the *Connect-*

ToSystem use case where the system user successfully logged into the system. The activity diagram also shows that after logging into the system a user needs to select one of the following options: search for a book, browse books or quit the interaction with the application. Furthermore, the user can only check the status of a book if the book was selected by the activity *Select book*.

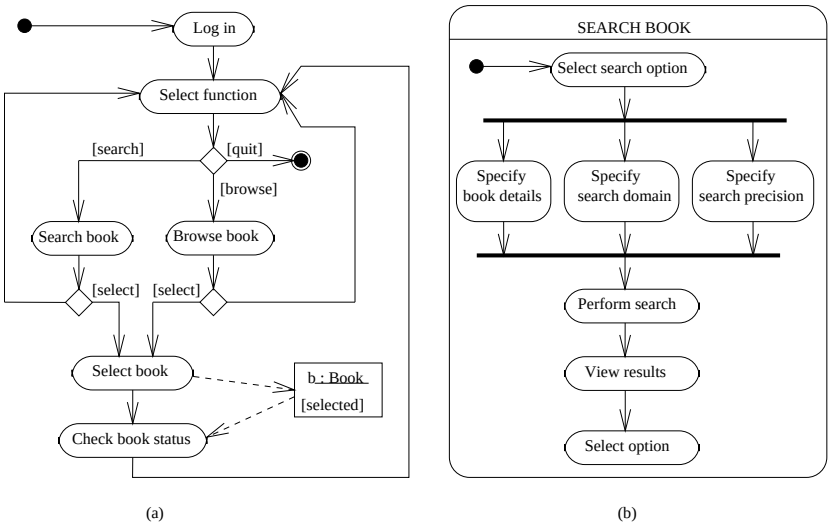

Figure 3: A partial view of the task model.

Applying activity diagrams to control user interface navigation resembles traditional Hierarchical Task Analysis (HTA) [13, 14, 4] widely used to describe user task models. Activities, use cases and tasks, however, are not exactly the same thing although they have similar characteristics. Use cases, for example, can be considered high level tasks, and some activities are also similar to tasks. However, the relationship between use cases and activities in UML is not particularly clear. In fact, use cases do not provide some features often associated with user requirements, like goals, pre-conditions and post-conditions, that may help the design of activity diagrams.

A single activity diagram could model the whole task flow control, but activities can also be decomposed. In fact, activities are not atomic, which means that they can be interrupted as well as taking some time to execute [2]. Using this decomposing facility, the activity *Search book* of Figure 3(a) can be explained more precisely by an additional activity diagram as shown in Figure 3(b).

Many tasks require information from the domain model as well as information provided by the users [8]. For instance, in our general activity diagram of Figure 3(a), a selected book should be passed from activity *Select book* to activity *Check book status*. This data flow within activity diagrams can be modelled using object flows. There, in Figure 3(a), the activity *Select book* identified the object `b` of class *Book* that was passed to activity *Check book status*. These data items can also be acquired from interaction diagrams associated with the use case. However, the object flow technique provided by activity diagrams avoids the necessity of checking interaction diagrams to discover how tasks access information.

4 Abstract Presentation Modelling

The need to model UI presentation arises naturally while modelling the application. Even for very simple scenarios, the modelling of part of the UI presentation is essential. At this stage we do not need a detailed model of the UI presentation, but only to know what kind of components compose the UI, how many components there are, and how they may be grouped. We also need to know which operations these UI elements should have. Therefore, we need an *abstract presentation model*.

The modelling of a user successfully logging into the Library System can be used to exemplify the use of an abstract presentation model. Figure 4 shows the sequence diagram for this scenario of the *ConnectToSystem* use case. According to the use case diagram in Figure 1, *ConnectToSystem* is associated with the $\ll actor \gg$ *LibrarySystemUser*. Hence, a *LibrarySystemUser* object initiates this interaction, sending a message to an instance of *Library System*, which acts as the whole Library System.

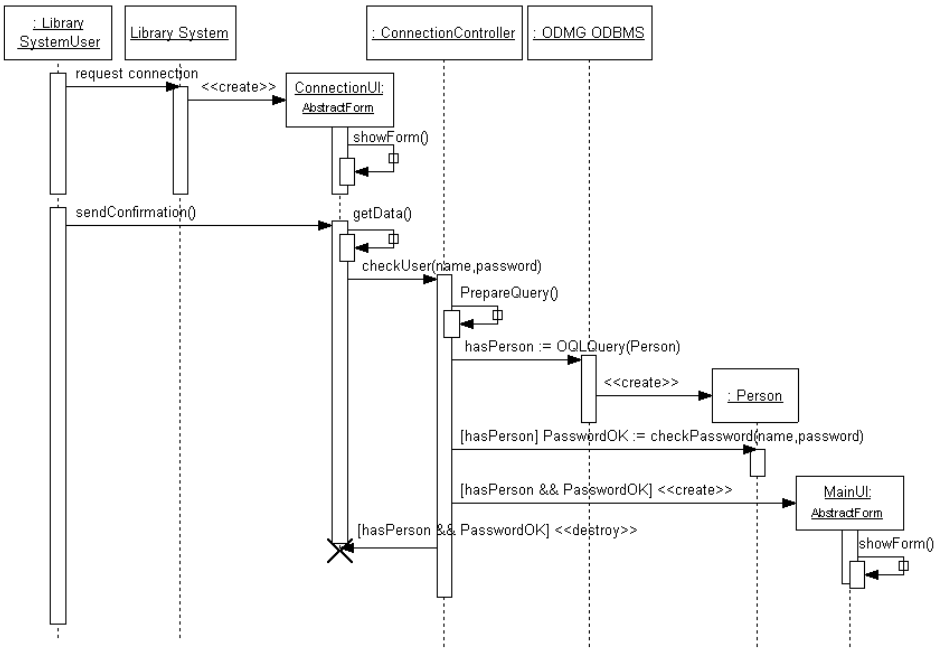

Figure 4: A sequence diagram for the *ConnectToSystem* use case.

Practically speaking, the *request connection* message can be, for example, a double click on the Library System's icon in a Windows environment. The system immediately creates a ≪*boundary*≫ object of the class *ConnectionUI*, that executes an operation called *showForm()*. The creation of an object is modelled by one object sending a message ≪*create*≫ to the new object. Once created, the *ConnectionUI* object presents to the user a connection user interface requesting a login name and a password. This user interface can be something like the form shown in Figure 5. Figure 5 is not a UML diagram since UML does not specify any notation for designing UI presentations. In fact, we are not claiming that UML should have a UI mock up notation that can lead to early commitment in terms of UI layout and component selection. However, we argue that UML needs a notation that can describe better the structure of abstract user interfaces than class and object diagrams. In fact, such notation could be used early in the UI design even to support the task design using activity diagrams.

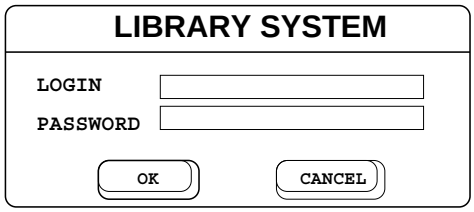

Figure 5: The display of the *ConnectionUI*.

The explanation of the *ConnectToSystem* sequence diagram requires some auxiliary definitions, which are provided below.

4.1 Abstract Presentation Structure

The abstract presentation model, shown in Figure 6, has a top-level container, which is the ≪*apm*≫ *AbstractForm*, that can have many components, ≪*apm*≫ *AbstractComponent*, and other containers, ≪*apm*≫ *AbstractContainer*. In fact, containers provide a grouping mechanism to the structural elements of the UI presentation. A generic abstract component is represented by the ≪*apm*≫ *AbstractComponent*. In Figure 6, *AbstractComponent* is specialised into three categories: *StaticDisplay*, *ActionInvoker* and *InteractionControl*. The ≪*apm*≫ stereotype identifies the abstract presentation model classes.

- The *StaticDisplay* category is related to those components that just provide some visual information, such as labels.

- The *ActionInvoker* category is related to those components that can receive system events that are propagated as system operations, such as buttons.

- The *InteractionControl* category is related to those components that can receive system events that normally model user options concerning navigation through the UI, such as menus.

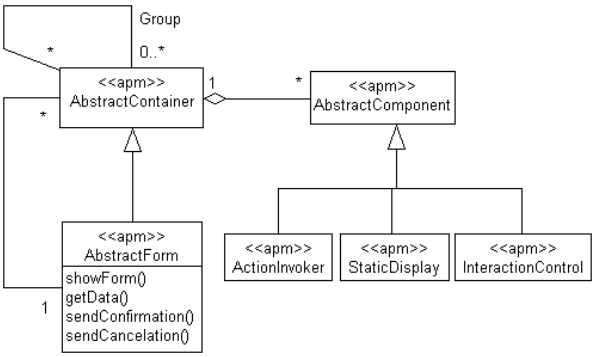

Figure 6: The abstract presentation model.

Bodart and Vanderdonckt [1] provide a more precise discussion of the categorisation of abstract components.

The class diagram shown in Figure 6 is the framework used to describe a conceptual user interface. An object diagram of this class diagram provides the conceptual description of the user interface. The *ConnectionUI* is conceptually described by the model shown in Figure 7. The links labelled with `compose` are those between *AbstractComponents* and *AbstractContainers*. The links between two instances of *AbstractContainers* are labelled with `integrate`.

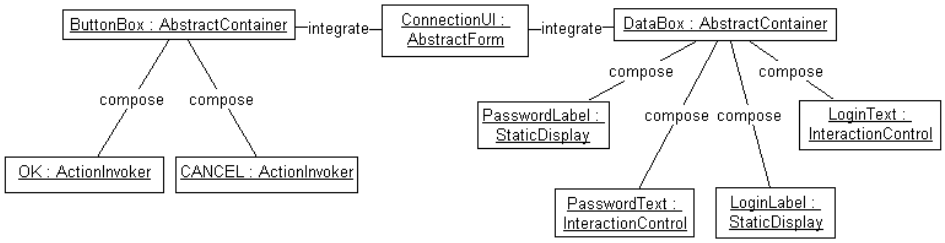

Figure 7: The abstract model of the *ConnectionUI*.

4.2 Abstract Presentation Behaviour

A set of four operations are defined for the *AbstractForm* class: *showForm()*, *getData()*, *sendConfirmation()* and *sendCancellation()*. At this stage of the design process, these four operations are enough to provide a basic understanding of a possible set of operations that should be implemented in some way by ≪*boundary*≫ objects.

- *showForm()* is normally used as a self-delegation message that draws the form on the output device. It is automatically performed when ≪*boundary*≫ objects are created.

- *getData()* collects information provided by the user after an interaction, doing any required transformation on the information provided into suitable parameters for system operations.

- *sendConfirmation()* and *sendCancellation()* messages model low-level system operations on the interaction of a system user with ≪*boundary*≫ objects. They are generated due to system events raised by aggregated components of the ≪*boundary*≫ objects. The *sendConfirmation()*

operation informs ≪*boundary*≫ objects that the system's user is submitting information to the system, while *sendCancellation()* indicates that the system user wants to abort the interaction with the form without submitting any information.

4.3 Using the Abstract Presentation Model

Returning to the *ConnectToSystem* sequence diagram in Figure 4, the *Library System* creates the ≪*boundary*≫ *ConnectionUI* object of class *AbstractForm*, which executes the *showForm()* method. This method draws the *ConnectionUI* form that is presented to the user. Interacting with the UI the user sends a *sendConfirmation()* message to the *ConnectionUI* object. The *sendConfirmation()* message can be an event associated with the `OK` button shown in Figure 5, but this is not specified during the abstract presentation modelling. The *ConnectionUI* object performs a *getData()* operation that picks up the data provided by the user. After collecting the data, the *ConnectionUI* object sends a system operation message *checkUser()* to the ≪*control*≫ *ConnectionController* object, passing the login name and password as parameters. The *ConnectionController* object prepares a query that is submitted to a database management system. If there are objects of class *Person* with the provided name in the database, the database instantiates *Person*. Then, the ≪*control*≫ *ConnectionController* object sends a message to the ≪*entity*≫ *Person* object checking the provided password. If the password is correct, the *ConnectionController* object creates a *MainUI* object and destroys the *ConnectionUI*. The presented sequence diagram is restricted to the scenario where the user successfully logs into the system. Unsuccessful attempts to log into the system scenario can be modelled as described in Section 6.1.

Activities, as presented in Section 3, and abstract presentation models are weakly connected by the flow objects in the activity diagrams. Indeed, *AbstractComponents* should be used in activity diagrams to explain the data flow between the UI and the underlying application. However, we believe that a well-defined relationship between activities and instances of *AbstractForms* can facilitate the design of tasks and abstract presentation. For instance, activities that involve user interactions should be supported by ≪*boundary*≫ objects. However, it is difficult to identify ≪*boundary*≫ objects from an activity or to identify activities from ≪*boundary*≫ objects.

5 Concrete Presentation Modelling

Abstract presentation models do not describe which components compose each ≪*boundary*≫ class. They also don't provide any description of layout. Further, they do not describe how events of user interface components relate to operations of ≪*control*≫ classes. Therefore, *concrete presentation models* are required sometime during the UI design process.

5.1 Concrete Presentation Structure and Layout

From the concrete presentation point of view, the abstract presentation model presented in Figure 6 is the design pattern specification for the UI presentation model. This pattern has been called *Presentation Framework*. In fact, the design pattern approach, as presented in Gamma *et al.* [5] and incorporated by UML, provides a way to describe how different environments can be accommodated within the diagrams that use elements of the abstract presentation model, e.g. the sequence diagram in Figure 4. Indeed, concrete presentation models are environment-dependent since they are described in terms of environment classes and components. An environment in our terminology can be classes of a object-oriented programming language, components or both. We are going to use Java [6] to show how UI classes may be related to *environment* classes. The ≪*cpm*≫ stereotype is used to identify these environment classes.

Figure 8 shows a concrete presentation model using the *Presentation Framework* and some Java AWT components. The *Presentation Framework* is represented using a collaboration symbol of UML with five different roles: AbstractForm, AbstractContainer, StaticDisplay, InteractionControl and ActionInvoker. The *Presentation Framework* pattern provides a clear description of how abstract presentation classes are replaced by concrete presentation classes, respecting the relationships of the abstract presentation model (Figure 6). In fact, Figure 8 also shows that the ≪*cpm*≫ *Frame* is bound to the ≪*apm*≫ *AbstractForm*, the ≪*cpm*≫ *Container* is bound to the ≪*apm*≫ *AbstractContainer*, the ≪*cpm*≫ *Label* is bound to the ≪*apm*≫ *StaticDisplay*, the ≪*cpm*≫ *TextField* is bound to the ≪*apm*≫ *InteractionControl* and the ≪*cpm*≫ *Button* is bound to the ≪*apm*≫ *ActionInvoker*.

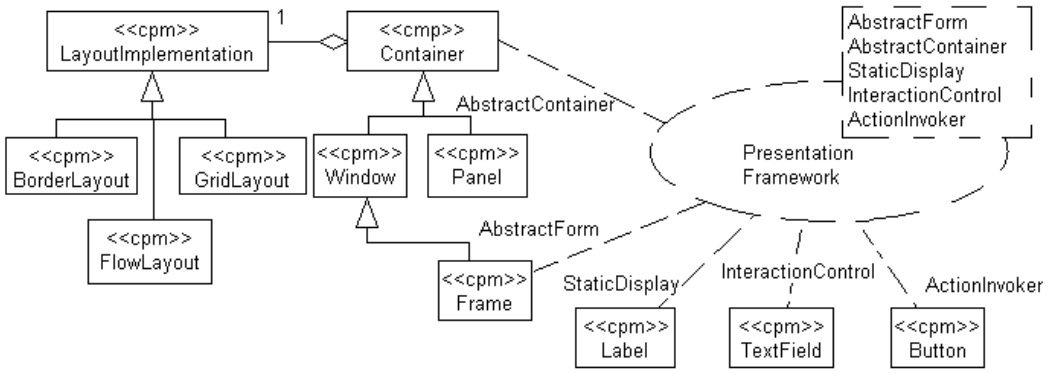

Figure 8: The concrete presentation models.

The *Presentation Framework* pattern can be extended to allow many components to be bound to StaticDisplays, InteractionControls and ActionInvokers. This extension can provide a declarative notation for modelling the mapping between abstract and concrete components. However, the concrete presentation model, as presented in Figure 8, provides what is required to model our case study. Additionally, the concrete presentation model provides a description of how presentation layout can be specified. Every instance of a class acting as a *Container* must have an instance of *LayoutImplementation* aggregated to it. Several categories of layout object can be added to the concrete presentation model by adding them as subclasses of *LayoutImplementation*. The modelling of the UI presentation layout is not entirely explained. Indeed, the concrete presentation model relies on the semantics of the environment. In the case of the concrete presentation model of Figure 8, Java provides algorithms embedded in methods that work as templates to model layouts for the presentation model of the UI.

The concrete presentation model based on the *Presentation Framework* does not cause large parts of the design to be environment-dependent, providing a model with a flexible and well-established relationship between the $\ll apm \gg$ classes and the component classes. For instance, the components presented in Figure 8 model Java's AWT components. The Swing components, however, can replace the AWT components naturally without breaking the abstract presentation model.

Figure 9 presents the concrete presentation model for the *ConnectionUI* presented in Figure 5. This model is an object diagram where the links are: the `compose` and `integrate` links introduced in Section 4.1, and the `organise` link that relates instances of *Frame* (playing the role of the *AbstractContainer*) with their respective instances of *LayoutImplementation*. This link is mandatory for each instance of *Frame*. Further, Figure 9 shows that *Panels* are being used instead of *Containers* to model non top-level containers. This is possible since the subclasses of the bound classes can also be considered as part of the concrete presentation model.

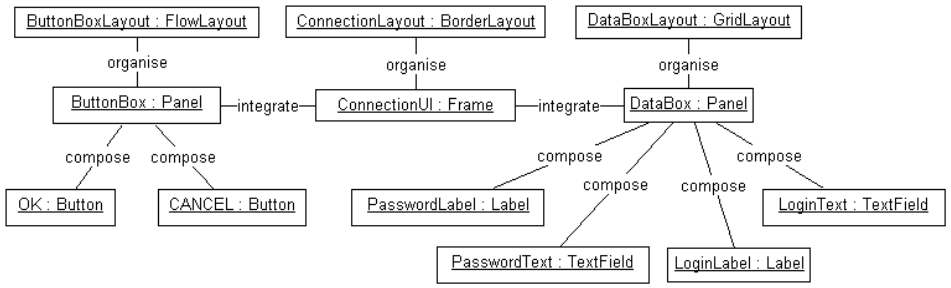

Figure 9: The concrete presentation model of the *ConnectionUI*.

5.2 Concrete Presentation Behaviour

Once we know how to model the structure of the concrete presentation model, we need to model the presentation's behaviour. Recalling Figure 9, there are a number of components that have behaviour

associated with them. For example, the *OK* object of class *Button* has an event associated with it that is triggered when the user presses the button. On the other hand, the object *PasswordLabel* of class *Label* does not have any event associated with it that needs to be handled by the application.

The first problem concerning presentation behaviour modelling is how to identify from UML diagrams possible events associated with ≪*boundary*≫ objects. In our application model every ≪*boundary*≫ object is an instance of *Frame* and has two operations that have events associated with them: *sendConfirmation()* and *sendCancellation()*.

Inspecting the application model, it can be seen that the *sendConfirmation()* message was described in the sequence diagram in Figure 4. The *sendCancellation()*, however, was not modelled at all. In fact, every message sent by an actor to a ≪*boundary*≫ object represents an event associated with an UI component. Therefore, one possible way to model all presentation behaviour is by producing one interaction diagram for each UI event. In the *ConnectToSystem* use case we only modelled one scenario: the successful logging of the user into the system. Therefore, we need to model a *ConnectToSystem* scenario where the `CANCEL` button is pressed.

Figure 10 shows the sequence diagram for the *ConnectToSystem* use case where the user presses the button `CANCEL`. The `CANCEL` button pressing event is represented as a *button pressed* message sent by the user to the graphical component `CANCEL` button. In fact, the *button pressed* message is a concrete presentation model event since it specifies which kind of event the component button should use to trigger the *sendCancellation()* message. Thus, the *sendCancellation()* message is sent to the *ConnectionUI* object, interrupting the interaction of the user with the Library System.

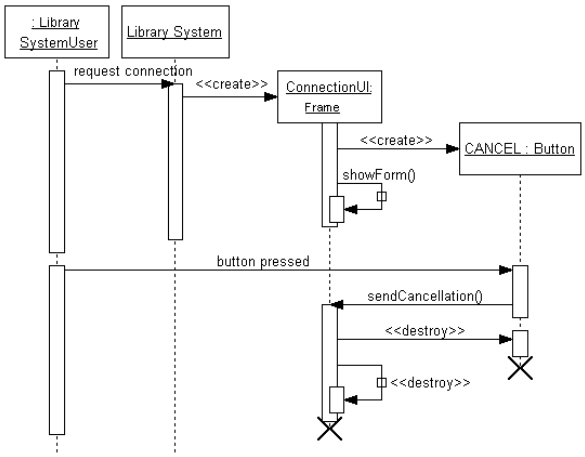

Figure 10: A second sequence diagram for the *ConnectToSytem* use case.

However, modelling using one interaction diagram for each UI event is not a good strategy since in a single use case scenario a user can interact with many ≪*boundary*≫ objects, and a single ≪*boundary*≫ object can have tens, or even hundreds, of events associated with it. Therefore, the strategy is to handle all combinations of events using a minimum number of interaction diagrams that describe all possible events. The problem is not as bad as it seems, though. Many components encapsulate part of the application behaviour. Complex components may reduce the number of events that a ≪*boundary*≫ object needs to handle.

6 Event Modelling

As described in Booch [2], *events* are "things that happen". Indeed, many things happen when we are using an application: keys and buttons are pressed, the mouse is moved, messages are sent to the network, etc. We are calling these things that happen *events*.

In a object-oriented user interface, inputs and outputs are streams of events [7]. Figure 11 shows a general event model where *user actions* and *synchronisation events* are sent to an object-oriented user interface as input events. The application, through its user interface, reacts to these input events generating output events that are presented as *visual feedback*. Visual feedback can be *normal feedback* or *abnormal feedback*. Abnormal visual feedback, such as error messages, is that associated with difficulties encountered during the enactment of a user's task.

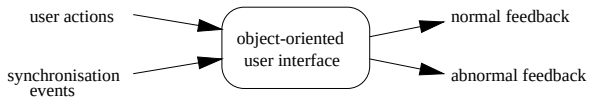

Figure 11: The event model.

User actions, system events and normal visual feedback events have been discussed throughout this paper. For instance, in Figure 4, users interact with ≪*boundary*≫ objects, ≪*control*≫ objects send and receive messages of ≪*boundary*≫ objects and of ≪*entity*≫ objects, and ≪*entity*≫ objects send and receive messages of ≪*control*≫ objects. Therefore, the aim of this section is to discuss a strategy for modelling user interface features related to:

- Exceptions, which are a special kind of event since they are created when something unexpected happened in the system. Indeed, a significant amount of the functionality of interactive applications may involve catering for the unexpected.
- Synchronisation events, which are generated inside the application to guarantee that data displayed in user interfaces are synchronised with data in the application.

Abnormal visual feedback can happen with synchronisation events, and synchronisation events can be generated from system actions. However, they are modelled independently. Therefore, the following sections discuss each of these special events in turn.

6.1 Modelling Exception Handling

Exceptions, as defined by Meyer in [16], are run-time events "that may cause a routine call to fail". Moreover, a routine call fails when it terminates its execution in a state not satisfying the routine's contract. These definitions are complex since they require further definitions such as the *routine's contract* definition, which is itself complex. Hence, identifying what is a fail and what is an exception are not obvious tasks. Despite the formal definition, exceptions here are more akin to those used in object-oriented programming languages such as Java [6] and C++ [18].

In terms of user interfaces, the important aspect of exceptions is that sometimes they are not entirely solved by exception handlers, leading the application to provide visual feedback to users that something is going wrong (or, at least, not going as expected). In fact, once activated, the exception handlers try to solve the problems identified by the exceptions without notifying the users. Unfortunately, exception handlers do not solve every kind of problem. Therefore, the user should be notified of those unsolved exceptions or involved in choosing a solution to the problem.

The problem now is how to model the aspects of the user interface that are related to exception handling.

6.1.1 Structural Aspects of the UI of Exception Handlers

In the application model there are many situations where exceptions and exception handlers can be used. For example, the designer could choose to display an error message somewhere in the *ConnectionUI* form due to an exception raised during the execution of a database query.

In UML notation, exceptions are modelled as a stereotyped ≪*send*≫ dependency from a class operation to an exception handler class [2]. Figure 12 shows a ≪*send*≫ dependency that links the operation *checkUser* in the ≪*control*≫ *ConnectionController* class with the ≪*exception*≫ *DatabaseFail* class. Moreover, Booch *et al.* [2] proposes a hierarchy of exception handlers identified by the ≪*exception*≫ stereotype. Usually, non-caught exceptions are sent to higher-level exception handlers in the hierarchy until they are caught by an exception handler or until they reach the top-level handler of the hierarchy. If some exception is not handled by ≪*exception*≫ *DatabaseFail* in Figure 12, then it must be handled by ≪*exception*≫ *Exception*.

Exceptions can be generated in any class, since classes generally have methods (that are routines), which have contracts that can be broken. Despite the fact that exception handlers can act as ≪*control*≫ classes, they are not modelled exactly as ≪*control*≫ classes. Instead, they can catch exceptions (events) from classes of any category. In this case, ≪*exception*≫ classes are introduced.

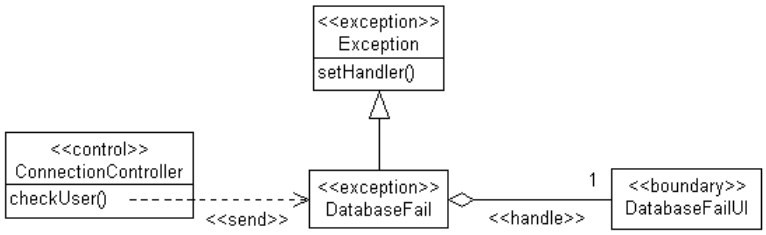

Figure 12: The relationship between the UI and the exception handler.

The operations of these classes can be called from any method of any class, even from methods of other ≪*exception*≫ classes.

One of the roles of ≪*exception*≫ classes is to act as ≪*control*≫ classes to ≪*boundary*≫ classes when exceptions happen. However, there are situations where ≪*exception*≫ classes cannot control a ≪*boundary*≫ class. For instance, if the exception handler requires some decision such as `quit` or `retry` from the user, and the original ≪*boundary*≫ object does not have components to deal with such an interaction, then a new ≪*boundary*≫ object should be created to provide the communication between exception handlers and users.

In terms of the user interface, however, it is important to know how ≪*boundary*≫ classes are related to this hierarchy of exceptions. Objects of ≪*exception*≫ classes can act as objects of ≪*control*≫ classes. Therefore, ≪*boundary*≫ classes can be aggregated to ≪*exception*≫ classes. In Figure 12, the ≪*exception*≫ *DatabaseFail* acts as a ≪*control*≫ class, handling the ≪*boundary*≫ *DatabaseFailUI* class. The ≪*handles*≫ stereotype is used to identify the relationship between ≪*boundary*≫ classes and their controllers.

6.1.2 Behavioural Aspects of the UI of Exception Handlers

Exceptions also affect the task model of the user interface since they can modify the flow of control from activity to activity during a user interaction. For instance, the activity *Perform search* in Figure 3(b) can raise a database exception [3] since a query is performed there.

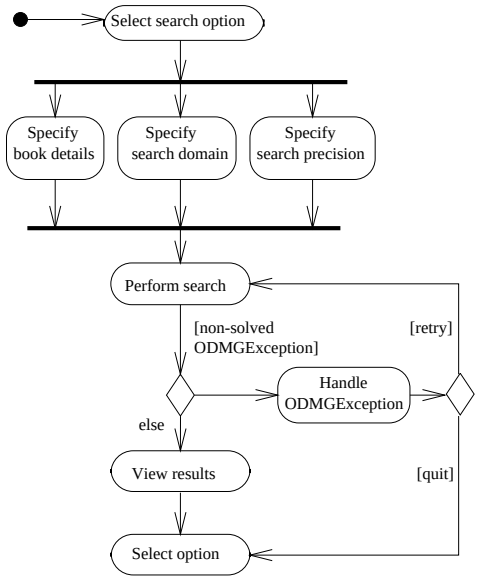

Figure 13: Exceptions in the task model.

The modelling of possible modifications to the flow of control of the task model is a straightforward task since UML's activity diagrams provide a branching notation. The outgoing transitions can be re-routed to different activities, depending on boolean guard expressions. Figure 13 shows the activity diagram of Figure 3(b) extended to model exception handling. The branch after the activity *Perform*

search (rendered as a diamond) re-routes the flow of control when exceptions happen during the execution of *Perform search*. The guard `[non-solved ODMGExceptions]` re-routes the flow of control to an activity called *Handle ODMGException* when an ODMGException is not solved by its handler. Otherwise, the flow of control follows the usual route identified by the keyword `else`.

6.2 Synchronisation Event Modelling

We are calling synchronous UIs those UIs where displayed data is frequently updated while the ≪*boundary*≫ objects are visible. Otherwise, they are asynchronous UIs.

User interfaces, and especially graphical UIs, are usually implemented using asynchronous messages [4]. Therefore, one additional problem concerning synchronous UIs is how to model them using asynchronous messages only. The general idea for solving this problem is to refresh the ≪*boundary*≫ objects with updated data as frequently as required. Therefore, the generation of events that produce UI updating is a possible approach to modelling synchronous UI. In this case, the generated event is called a *synchronisation event*. The natural candidates for synchronisation event generation are the ≪*entity*≫ objects, since they are the location of the updated data. Synchronisation events could also be generated by ≪*boundary*≫ and ≪*control*≫ objects, but they will need to query the ≪*entity*≫ objects to get the updated data for each generated synchronisation event. Therefore, we are only considering here the case where synchronisation events are generated by ≪*entity*≫ objects.

One possible approach to modelling synchronous events generated by ≪*entity*≫ objects is presented by the class diagram shown in Figure 14.

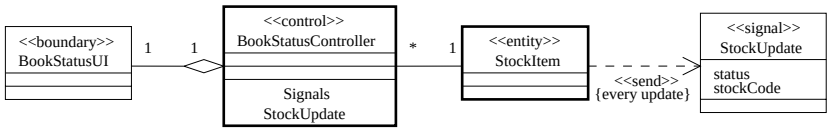

Figure 14: The model of the synchronous *BookStatusUI*.

Some UML features not previously discussed are used to model the synchronous UIs: classes rendered with heavy lines, classes rendered with a signal compartment, a time constraint in a dependency relationship and the ≪*signal*≫ stereotype. Thus, we need to explain these features before explaining the model.

- The heavy lines in *BookStatusController* and *StockItem* mean that the instances of these classes are active objects. In fact, instances of *BookStatusController* must have some active behaviour to "receive" the *StockUpdate* events generated by the instances of *StockItem*.

- The constraint {every update}, in the ≪*send*≫ dependency between *StockItem* and *StockUpdate*, means that the instances of *StockItem* must generate a *StockUpdate* event every time the *StockItem* is updated. This constraint means that the instances of *StockItem* have an active mechanism for identifying state updates.

- The ≪*signal*≫ stereotype means that the instances of these classes are signalling to the system that an event of type *StockUpdate* is happening. The meaning of this event, in this context, is that the objects of class *StockItem* are pushing the updated *status* of the stock items to those classes that are listening for this kind of event.

- The *Signals* compartment in the *BookStatusController* identifies that the instances of this class can be listeners for the *StockUpdate* events. This means that the instances that are also listeners are notified every time a *StockUpdate* event is generated.

Returning to the class diagram in Figure 14, we notice that every time that a *StockItem* update happens the instances of *StockItem* generate a *StockUpdate* synchronisation event, that has the current status of the stock item. Then, these events are listened for by the instances of *BookStatusController*, which sends messages to the instances of the ≪*boundary*≫ class, updating the displayed data. The attribute *stockCode* inside the *StockUpdate* signal works as an identification mechanism preventing the *BookStatusControl* objects from displaying the status of different stock items.

This modelling approach is especially suitable for systems where synchronous UIs are essential. In fact, the complexity of the pushing mechanism inside the ≪*entity*≫ objects is always the same. For instance, complex ≪*entity*≫ objects can have many operations that change their state. However, using this approach, as the state is monitored for updates, it is not necessary to identify the state modification of the objects inside their methods. Moreover, the complexity of the pushing mechanism is the same if the ≪*entity*≫ objects are displayed by one or several ≪*boundary*≫ objects. Indeed, the event is sent to every object which has a signature for it. Therefore, one single synchronisation event can notify however many listeners it has.

7 Putting All Together: Packaging the Application

Figure 15 shows a package diagram that provides an overview of the whole system. Further, this package diagram shows dependencies between several components of the system.

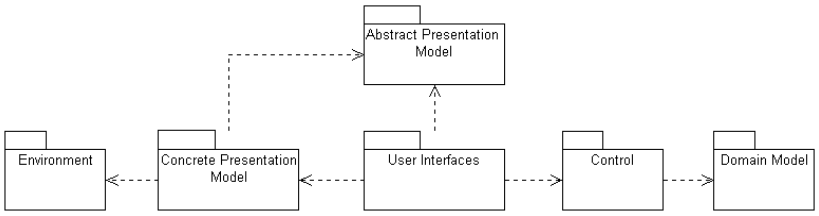

Figure 15: The package diagram of the Library System.

The classes and class instances are grouped into six packages, as follows.

- The *User Interface* package composed of ≪*boundary*≫ classes and objects.
- The *Abstract Presentation Model* package composed of ≪*apm*≫ classes as shown in Figure 6.
- The *Control* package composed of ≪*control*≫ classes. These classes are presented in sequence diagrams such as in Figure 4.
- The *Domain Model* package composed of ≪*entity*≫ classes. The class diagram of these classes forms the domain model in Figure 2.
- The *Environment* package composed of those classes used to build the user interface. The *environment* could be an object-oriented programming language such as C++ [18], Java [6] or Smalltalk, some components such as ActiveX and Java Beans, or even a composite set of components and object-oriented programming languages. The environment concerns mainly the visual part of the user interface, but it could be responsible for important user interface behaviours, especially when considering the use of complex components.
- The *Concrete Presentation Model* package composed of those classes of the *environment* package that are bound to ≪*apm*≫ classes of the *Presentation Framework* pattern.

8 Conclusions

This paper discussed user interface modelling using a Library System case study. The application system was modelled using the Unified Modeling Language that has proved to be useful for modelling user interfaces. In fact, UML has a rich set of constructors complete enough to model the architectural aspects of form-based user interfaces. However, such UI modelling may not be as straightforward a process as expected and desired. Indeed, some modelling problems were identified from the case study:

- UML does not describe clearly the relationship between use cases and activities (see Section 3). Use cases do not provide some aspects of user requirements like goals, pre-conditions and post-conditions that may help the design of activity diagrams.

- UML does not have a notation to describe abstract presentations (see Section 4). In fact, we believe that UML needs such a notation in order to support the design of UI presentations.

- UML does not provide a relationship between classes providing an abstract presentation (*AbstractForms* in Figure 6) and activities. In fact, it is difficult to identify which UI is related to each activity that involves user interaction, as described in Section 4.3.

Additionally, the case study provides an illustrative example of the use of many UML constructors, in terms of diagrams, for modelling the user interface. The summary of the UML diagrams used is presented in Table 1, and the constructors are those used in the diagrams presented throughout the paper.

User Interface Element	UML Resource
Domain Model	class diagram
Task Model	activity diagram
Presentation Model (abstract and concrete)	class diagram with design patterns interaction (sequence) diagram object diagram
Event - UI related with Exception Handler	class diagram activity diagram
Event - UI synchronisation	class diagram

Table 1: Summary of the UML diagrams used to model UI elements.

We are using some of the Statecharts [11] constructors since we are using activity diagrams [10]. However, we are not considering the design of widgets in this paper. For this reason we are more interested in inter-object transitions (activity diagrams), than in intra-object transitions (statecharts) while modelling widgets.

There are also some lessons that can be learned from the modelling of the Library System:

- The design of an user interface is a complex process since it requires complete comprehension of the elements that compose the user interface. Indeed, UIs in general have many elements that are not clearly required from the beginning of the design.

- The elements of the user interface have many dependencies among them, as shown in Figure 15. Therefore, the design process should consider UI modelling as integral.

There is room for further discussion of how to model user interfaces using UML. Indeed, there will be other ways of representing user interfaces using UML. The study in this paper presents only one approach.

Acknowledgements. The first author is sponsored by Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Brazil) – Grant 200153/98-6.

References

[1] F. Bodart and J. Vanderdonckt. Widget standardisation through abstract interaction objects. In *Advances in Applied Ergonomics*, pages 300–305, Istanbul - West Lafayette, May 1996. USA Publishing.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.

[3] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Francisco, CA, 1997.

[4] D. Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cummings, Redwood City, CA, 1995.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[7] M. Green. A survey of three dialogues models. *ACM Transaction on Graphics*, 5(3):244–275, July 1986.

[8] T. Griffiths, P. Barclay, J. McKirdy, N. Paton, P. Gray, J. Kennedy, R. Cooper, C. Goble, A. West, and M. Smyth. Teallach: A model-based user interface development environment for object databases. In *Proceedings of UIDIS'99*, pages 86–96, Edinburgh, UK, September 1999. IEEE Press.

[9] T. Griffiths, J. McKirdy, G. Forrester, N. Paton, J. Kennedy, P. Barclay, R. Cooper, C. Goble, and P. Gray. Exploiting model-based techniques for user interfaces to database. In *Proceedings of VDB-4*, pages 21–46, Italy, May 1998.

[10] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. Version 1.3.

[11] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 97.

[12] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, Reading, MA, 1992.

[13] P. Johnson. *Human computer interaction: psychology, task analysis and software engineering*. McGraw-Hill, Maidenhead, UK, 1992.

[14] B. Kirwan and L. Ainsworth. *A Guide to Task Analysis*. Taylor & Francis, London, UK, 1992.

[15] S. Kovacevic. UML and user interface modeling. In *Proceedings of UML'98 International Workshop*, pages 235–244, Mulhouse, France, June 1998. ESSAIM, Mulhouse.

[16] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, second edition, 1997.

[17] B. Myers and M. Rosson. Survey on user interface programming. In *Proceedings of SIGCHI'92*, pages 192–202, 1992.

[18] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, third edition, 1997.

[19] P. Szekely. Retrospective and challenges for model-bases interface development. In *Computer-Aided Design of User Interfaces*, pages xxi–xliv, Namur, Belgium, 1996. Namur University Press.