Generating User Interface Code from Declarative Models: The Teallach Approach

Paulo Pinheiro da Silva Tony Griffiths
Norman W. Paton

Department of Computer Science,
University of Manchester
Oxford Road, Manchester M13 9PL, England, UK
Phone: +44-161-2756139 Fax: +44-161-2756236
e-mail: {pinheirp,griffitt,norm}@cs.man.ac.uk

August 13, 1999

Abstract

Declarative models can provide abstract descriptions of user interfaces. Therefore, it is desirable to use declarative models for designing user interfaces since complex details of the user interfaces can be avoided at the design time. However, declarative models are usually not able to describe all aspects required to generate user interfaces. This paper describes the code generation of complete user interfaces from declarative models. The runtime context and the code generation process are presented and explained. Details of the declarative models and the runtime class library used by the code generator are described. Further, an illustrative example of the use of the code generator is also provided. The code generator is part of the Teallach model-based user interface development environment.

1 Introduction

There are significant improvements in the development of graphical user interfaces during this last decade. Widgets, for example, are even able to easy handle images and sounds now. Moreover, there are powerful tools such as user interfaces (UI) builders and toolkits that facilitate the hard work that are the widgets handling and the UI layout customisations [10]. Despite the improvements on graphical user interfaces and their developing tools, they still are a challenging part of the development of most data intensive applications. User interface management systems (UIMSs) appeared as a category of tools trying to support a systematic design of user interfaces [14, 18]. UIMSs improve the UI development when compared with UI builders and toolkits [10]. However, UIMSs are not able to describe UIs providing a suitable level of abstraction. For instance, UIMSs usually require a description of the complex dialogue between users and user interfaces instead of

a description of the tasks performed by users and applications. Model-based user interface development environments (MB-UIDEs) are the state-of-art in terms of UI development [6, 17]. Using MB-UIDEs, users can design UIs abstracting details normally required by UIMSs.

The MB-UIDE technology, however, is not a stable technology. In fact, most MB-UIDE prototypes don't achieve a running user interface. Therefore, it is hard to show that their declarative models have the information required for generating running interfaces. It is even hard to prove that it is possible to generate user interfaces from their declarative models. Teallach [6, 7, 5] is a MB-UIDE that can generate Java user interface code for data intensive applications from UI declarative models. This paper present details of how the code generator successfully generates user interface code from declarative models in Teallach.

The paper is structured as follows: Section 2 presents the reasons to implement a code generator to achieve running user interfaces from declarative models. Section 3 describes the declarative models used by the code generator. Section 4 describes the runtime context, which includes the class library used by the generated code. Section 5 describes the code generation process. Section 6 presents a code generation example. Section 7 presents some conclusions and indicates possible improvements for the presented code generator.

2 Background

MB-UIDE Development Life-cycle. The development of UIs using MB-UIDEs is centred in the design of the UI. The running interfaces are automated generated from the user interface models by the MB-UIDEs [17]. The running interface, as a UI prototype, can be used by designers and expert users to validate the models. Any design refinement must be carried out in the declarative models. Indeed, the declarative models must contemplate the whole UI design. At the end of the design phase, the user interface code can be generated from the complete and validated models.

Further UI refinements can be carried out directly into the generated code at the implementation time. It is not expected to be a hard work to refine, and even to maintain, the generated code. In fact, the code is usually written in a standard way, and in a well-known object-oriented programming language (OOPL).

Code generation is not the unique way to achieve running interfaces from declarative models. In Humanoid [16], for example, the running interfaces are generated by Amulet [11], that is a UIMS. In this case, the Humanoid generates the specification of a UI for Amulet. In ITS [19] the models are interpreted by the application. In this case, the ITS runtime interpreter is linked with the application.

Code Generation Choices. The code generation approach have some benefits when compared with the other approaches for generating running interfaces. The code generation can make the UI easy take advantage of the benefits of the selected OOPL. For instance, the UI can be platform-independent if it is coded in Java [4], or it can have a high-performance if coded in C++. No overheads are introduced by the use of UIMSs or

runtime interpreters. Additionally, the user interface code can be freely modified without restrictions due to other softwares responsible for the execution of the UI.

In terms of the declarative models validation, the code generation is also a good choice since the running interfaces are entirely generated from the models. In this case, don't have a single piece of software between the models and the running interface that can insert or modify the behaviour of the generated interfaces. These are some reasons for choosing the code generation approach. However, the use of UIMSs and runtime interpreters also present some benefits such as the dynamic reconfiguration of the user interface and their components.

Code Generation in other MB-UIDEs. There are many researches improving the MB-UIDE technology. Most of these researches has at least a strategy for generating running interfaces. ITS [19] is one of the few examples of MB-UIDE that interpret declarative models at runtime. The use of UIMSs is a popular choice, indeed. AME [9] generates code for Open Interface. Humanoid [16] and the first version of Mastermind [17] generate code for Amulet [11]. TADEUS [3] generates code for ISA Dialog Manager. The code generation is the other popular choice. AME [9], JANUS [1] and the second version of Mastermind [15] generates C++ code, while MOBI-D [12] and Teallach [6] generates Java code.

Despite these efforts, none of them completely describes how to generate code from the declarative models. Schlungbaum and Elwert [13] presents details of how TADEUS generates a UI description file for ISA Dialog Manager, that is a UIMS. However, it is not possible to show that a UI code can be generated only from the TADEUS models. The Stirewalt's thesis [15] also describes part of the code generation problem. There, Stirewalt describes how the second version of Mastermind generates code using the Mastermind Toolkit (MMTK), that is based on C++. However, only part of the declarative models are used to generate code using MMTK. The other part of the models are still used to generate code using Amulet.

3 Teallach Declarative Models

The Teallach is a development environment where designers can graphically model the user interface. The models described here are those implemented into the development environment.

The Declarative Models. There are three models that describe different aspects of a user interface: the *domain model* that describes the domain application properties that are relevant to the user interface; the *task model* that describes the tasks required to execute the application functions; and the *presentation model* that describes the visual part of the user interface.

The domain model can be defined by persistent ODMG objects, distributed CORBA IDL objects or transient JAVA objects. A wide range of domain models can be used as a Teallach domain model.

The task model is composed of a hierarchical tree of tasks. Tasks that have subtasks are called composite tasks, while leaf tasks are primitive tasks. Composite tasks can be: *sequential tasks* where the subtasks are executed in order; *order independent tasks* where the subtasks must be executed, but in any order; *choice tasks* where only of the subtasks is executed; *optional tasks* where any number of subtasks might be executed, even none or all of them; *repeatable tasks* composed of only one subtask that is repeated the number of times specified by its parent task; *conditional tasks* that specify conditions to execute their children tasks that evaluate specific conditions. Additionally, composite tasks can have state objects that are objects from the domain model or from the presentation model.

Primitive tasks can be *action tasks* or *interaction tasks*. State object methods are invoked into action tasks. The interaction between users and application is performed into interaction tasks using presentation objects declared as state objects in ancestors' composite tasks.

The presentation model is composed of interaction objects. There are two categories of interaction objects, defining two distinct level of abstraction: the concrete interaction objects (CIO) and the abstract interaction objects (AIO). The concrete interaction objects are the widgets that compose the UI. However, as object-oriented programming languages usually have many CIOs, and these CIOs require many customisation, the UI designer can model the UI only specifying AIOs that describe the relevant properties of the interaction objects at the design time.

The presentation model still require some grouping components to aggregate the interaction objects. Like interaction objects, there are abstract and concrete grouping components.

A comprehensive discussion about the Teallach models is described in [5]. Details of how the Teallach models are edited are described in [].

Model Relationships. The task model is responsible for the integration of the Teallach models. Precisely, the state objects inside the composite tasks are the integration elements of the Teallach model. Each state object is a domain object or a presentation object. Therefore, there are implicit links between the tasks where the state objects are defined and members of the domain and the presentation models. One special kind of link that exists inside the task model is that linking the state objects to the primitive tasks where they are effectively used. Figure 1 shows how a link is specified in the Teallach development environment.

The links between the domain model and the other models are restricted to the links provided by the state objects. However, it is required additional links between the task model and the presentation model. Each interaction task requires one CIO. Action task can be linked to CIOs. Composite tasks can be linked to grouping components.

4 Runtime Context

The generated user interface is coded in Java using Swing. The Model-View-Controller (MVC) architecture is used to code the user interface components. A specialised class

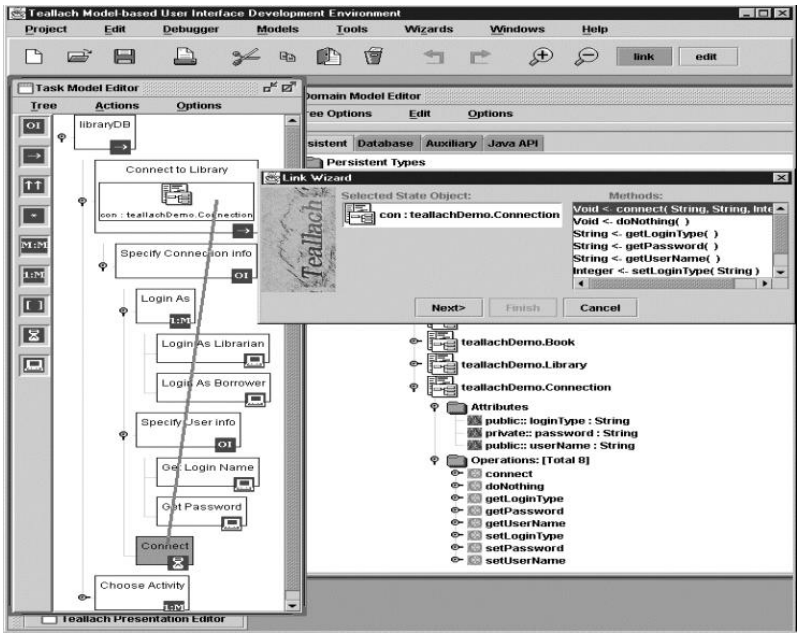

Figure 1: Snapshot of a Teallach screen when the user is explicitly linking an object from the domain model with a state object in the task model. A wizard window helps the user modelling the link.

library coded in Java and using the MVC approach is also used by the generated interface.

Model-View-Controller Architecture. The MVC architecture [8] provides a suitable strategy for developing user interfaces using OOPL. Indeed, the MVC specify how the user interface software should be separated into components, each one with a specific function. The *model* components are responsible for handling the state of objects used by the user interface. The *view* components are responsible for the user interactions that display the states of the *model* components. The *controllers* are responsible for handling the user interactions that can modify the state of the *models*. One additional benefit of the MVC architecture is that it describes the possible relationships between the MVC's components. In particular, the MVC architecture provides a clear distinction between the visual part of the user interface – the views and controllers – and the state of the user interface – the models.

Java and Swing. Nowaday there are many powerful and reliable OOPLs that can be used for implementing graphical user interfaces. Further, there are many OOPLs that also provide facilities to implement UI code using the MVC architecture. Java is the chosen OOPL in the Teallach project. The main reason for this choice is that Java is a platform-independent OOPL due to the Java Virtual Machine (JVM) [4]. In fact, the JVM avoid the complex problem that is the migration of UIs developed from a specific platform to

another platform.

Moreover, the Teallach project is also considering the use of Swing components. These components are *lightweight* Java components, which means, they are entirely coded in Java, not relying on native code as the Java's Abstract Windowing Toolkit (AWT). Additionally, the Swing components provide a clear distinction between *view* components and *model* components since they implement the MVC architecture. Therefore, it is possible to dynamically move from a Motif-like UI to a MS-Window-like UI just replacing the set of *views*, called the *look-and-feel* of the UI. Thus, the Java with the Swing components provide the required programming resources that are used by the Teallach Code Generator.

Class Library. A runtime class library has been created with the aim of reduce the necessity of generate the whole user interface code. The strategy is to keep the complexity of the user interface code into the runtime class library avoiding the generation of complex classes. The runtime class library contains a task type class hierarchy, presented in Figure 2, and a state object wrapper class.

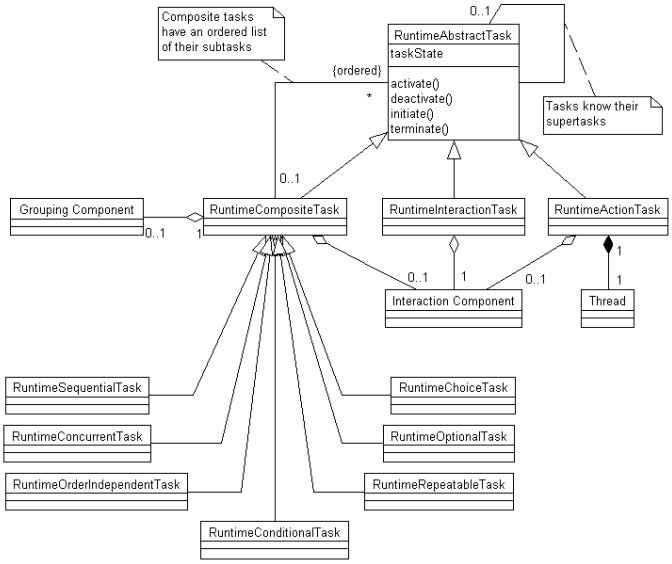

Figure 2: The task type hierarchy.

The tasks of the task model become subclasses of the `RuntimeAbstractTask` class. More precise, the composite tasks become subclasses of the `RuntimeCompositeTask`, the action tasks become subclasses of the `RuntimeActionTask` and the interaction tasks become subclasses of the `RuntimeInteractionTask`. The `RuntimeCompositeTask` provides a method to aggregate a Swing grouping component into their subclasses. The `RuntimeCompositeTask` also provides a method to aggregate subtask classes into their subclasses. The `RuntimeActionTask` provides a thread to execute the assigned state object operation. The `RuntimeInteractionTask` provides a method to aggregate a Swing

component.

As subclasses of the `RuntimeAbstractTask` class, the task classes implement the operations `activate()`, `deactivate()`, `initiate()` and `terminate()`. Basically, the `activate()` operation enables the interaction of the user with the part of the UI responsible for the activated task. On the opposite way, the `deactivate()` operation disables the interaction. The `terminate()` operation, usually associated with the `deactive()` operation, notifies the parent task class that the current task class has finished. The `initiate()` operation returns the UI to the state it had when initially created. The `initiate()` operation is invoked on the children tasks of a composite task that has been activated. Specific behaviours for these operations are performed for task classes of different categories. For instance, composite task classes invoke the `setVisible(true)` operation for their aggregate grouping component, and interaction task classes invoke the `setEnable(true)` operation for their aggregated CIO, when these task classes are activated.

Still in Figure 2, action and composite tasks can have aggregated interaction components. In this case, the interaction components are called *initiators*. Initiators are required to: (1) fire action tasks that are designed to be started on demand; (2) fire composite tasks that are subtasks of choice tasks, optional tasks or order independent tasks.

In terms of state objects, the class library provides the `RuntimeStateObject` class that is a state object wrapper. There are two reasons to wrap an state object: (1) the user interface code require static references of state objects that, as instances of domain objects or CIOs, can be dynamically instantiated, modified and destroyed during the execution of the user interface; (2) there are no mechanism to identify state object state transitions.

The user interface code must have a static reference to the state objects since the links between the state objects of a task class and the state objects of its subtask classes is established in the task class constructors. As state object wrappers are instantiated with the task classes and are not destroyed during the application execution, then they can be used as a static reference to the state objects.

The necessity to identify state object state transitions is due to the MVC architecture. State objects that are domain objects are MVC models. State objects that are presentation components can be MVC views, controllers, or both view and controllers. Then, the interaction task classes need to identify MVC model's modification to update their views and/or need to identify MVC controller's action to update their models. Therefore, the `RuntimeStateObject` class provides a method used by the interaction task classes to register their adaptor classes as listeners of the required state objects. The MVC architecture also require static references to state objects as the task classes.

5 Code Generation Process

From the Task Model to the UI Code. The code generator executes four modules, in this order: the `ApplicationGenerator`, the `StateObjectMapper`, the `CompositeTaskGenerator` and the `PresentationPacker`. The `ApplicationGenerator` creates a standard class responsible for the invocation of the root composite task at runtime. The `StateObjectMapper` provide a mapping of the state object wrappers between parents and

children tasks. The `CompositeTaskGenerator` creates the code of the composite task classes and also invokes the `ActionTaskGenerator`, the `InteractionTaskGenerator` and the `PresentationGenerator`, when required. The `PresentationPacker` add into the presentation classes the presentation components not linked with tasks.

The task model describes in which composite task the state objects are declared, and in which primitive task they are used. At the design time, it is assumed that state objects are visible in the descendant tasks of the composite task where they were declared. The `StateObjectMapper` identifies the state objects that are requited for each composite task. The mapper performs a preorder traversal of the task model populating a mapping table of each composite task with their declared state objects. For each primitive task reached in this preorder traversal, a new process is initiated. This new process populates the mapping table of the tasks between the primitive tasks and the tasks where the state objects were declared.

The `CompositeTaskGenerator` performs recursively another preorder traversal of the task model. During this traversal the `CompositeTaskGenerator`: (1) generates the code for the reached composite tasks; (2) invokes the `PresentationGenerator` for the composite tasks that are explicitly linked to a grouping component of the presentation model; (3) invokes the `ActionTaskGenerator` for the reached action tasks; and (4) invokes the `InteractionGenerator` for the reached interaction tasks. The standard code of a composite task class creates an array of subtasks that are invoked according to the temporal relation specified by the superclasses of the composite task classes. The composite task classes receive an array of `RuntimeStateObjects` that are the state objects wrappers, forwarding a proper array of `RuntimeStateObjects` for each subtasks, according to the mapping table created during the execution of the `StateObjectMapper`.

The `PresentationGenerator` is invoked for a specific grouping component of the presentation model. During an execution of the `PresentationGenerator`, it performs a breath-first traversal of the presentation model identifying the immediate children of the provided grouping component. A code is generated for the provided grouping component and its children that are interaction components. The children that are grouping components are not contemplated during this execution of the `PresentationGenerator`. The generated code is composed only of CIOs. Therefore, the translation of AIOs into CIOs is performed at the code generation time. The code for the children grouping components are generated when they are the provided grouping components of the `PresentationGenerator`, or after the execution of the `CompositeTaskGenerator`, when the code for the presentation model components not explicitly referred by the task model are generated.

Extended Interaction. There are widgets that are not modelled in the presentation model that are coded in the generated interfaces. These widgets belong to a special category of widgets responsible for controlling the composite tasks. There are three possible controls that these widgets inform to the composite task. The `Cancel` button notifies the task that it was aborted by the user. The `Ok` button notifies the task that it has been finished. The `Next` button, used only in concurrent tasks, notifies that the user wish to see the UI of the next running subtask. These buttons are added at the bottom of the

grouping component aggregated to the composite task. Table 1 shows how each one of these buttons are used in the Teallach.

Composite Task Category	Cancel	OK	Next
Choice	yes	no	no
Concurrent	yes	no	yes
Conditional	yes	no	no
Optional	yes	yes	no
Order independent	yes	yes	no
Repeatable	yes	no	no
Sequential	yes	no	no

Table 1: Extended interaction for composite tasks.

The behaviour of the `Cancel` button depends on the category of the parent task of the composite task that is been cancelled. If the parent task is a choice task, the parent task is restarted. If the parent task is an optional task, the current task is finished. If the parent task is a concurrent, repeatable or an order independent task, the current task is restarted. In fact, the initialisation of subtasks of concurrent and order independent tasks only activate the initiators of the subtasks. Finally, if the parent task is a sequential task, the parent task is finished.

6 An Example

The development of UIs using Teallach is exemplified by the design and the code generation of the UserConnection user interface of a library system. Executing the Library System, the user must specify if s/he is connecting as a librarian or a borrower. Further, the user must provide her/his login name and password. Figure 3 shows a possible set of Teallach models for the UserConnection UI.

In the task model, the `connect` composite task has three subtasks: the `newConnectionData` action task, the `specifyInfo` order independent task and the `tryConnection` action task. Additionally, the `connect` task has a state object `Con` of the type `ConnectionData`. The `Con` state object is a domain object. Therefore, the domain model provides the information concerning the `ConnectionData` type. Figure 3 also shown that the `specifyInfo` task, as a composite task, has its own subtasks, that also have other subtasks.

The Teallach presentation model can be composed of abstract and concrete components. In the presentation model of Figure 3, there are abstract grouping components, such as the `Container`, and concrete grouping components, such as the `JPanel`. The presentation model also has interaction components such as the `Inputters`, that are AIOs, and the `JRadioButton`, that are CIOs. Abstract components require less customisation than concrete objects, which facilitates the design of the UIs. However, concrete components are required to refine the user interface design, fixing some preferences in the model. Abstract components are replaced by their defaults concrete components at code generation time. Default settings are used to customise the concrete components. Concrete objects fixed in the presentation model and their settings are preserved by the code generator.

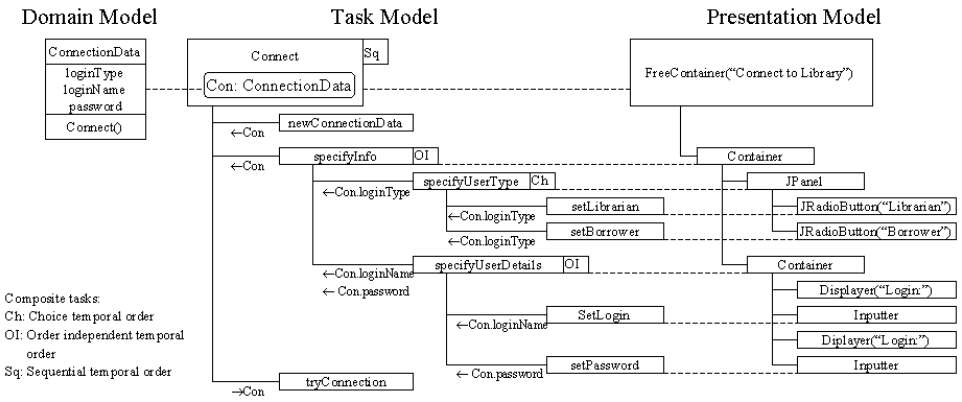

Figure 3: The Teallach models for the UserConnection user interface.

The Generated Code. Figure 4 presents a UML [2] class diagram displaying the relationships between the generated user interface classes. The `LibrarySystem` class was generated by the *ApplicationGenerator*. The `LibrarySystem` class, as the application class, is the class that implements the `main()` method and that activate the root task `connect`. The classes which names are suffixed with `'Container'` were generated by the *PresentationGenerator*. The classes which names are suffixed with `'Interaction'` or `'Adaptor'` were generated by the *InteractionGenerator*. The classes which names are suffixed by `'Action'` were generated by the *ActionGenerator*. The classes which names are the same of the task model are composite tasks that were generated by the *CompositeTaskGenerator*.

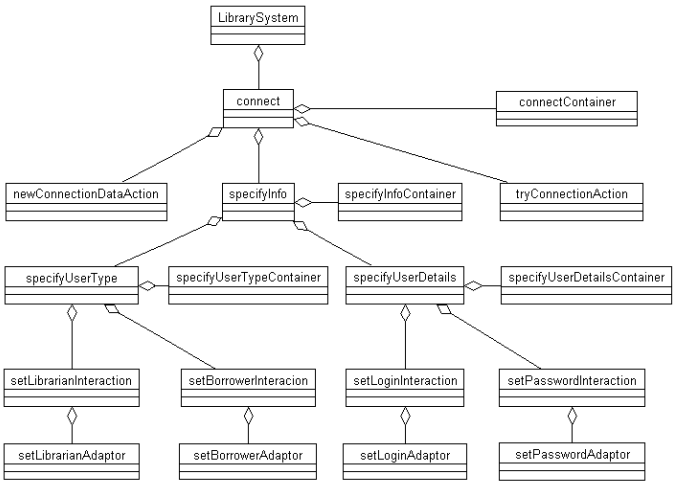

Figure 4: The generated user interface classes.

Excepting `'Container'` and `'Adaptor'` classes, the other classes in Figure 4 are subclasses of the `RuntimeAbstractTask` in Figure 2. For instance, the action tasks are subclasses of the `RuntimeActionTask`, the interaction tasks are subclasses of the `RuntimeInteractionTask` and each composite task is a subclass of one of the subclasses of the `RuntimeCompositeTask`.

The running user interface. The path of the class library and the domain application classes must be added into the environment variable `ClassPath`. After this setting, the compilation and execution of the generated classes will produces a running user interface for the application. The domain classes are naturally used by the user interface since the declarative domain model provides the information required by the user interface. Figure 5 shows the running UserConnection user interface. The analysis of the generated user interface indicates that some refinements concerning `Cancel` buttons can improve the quality of the UI. In fact, some optimisations related to extended interactions, in general, are been considered. However, these optimisations are not been considered in this paper.

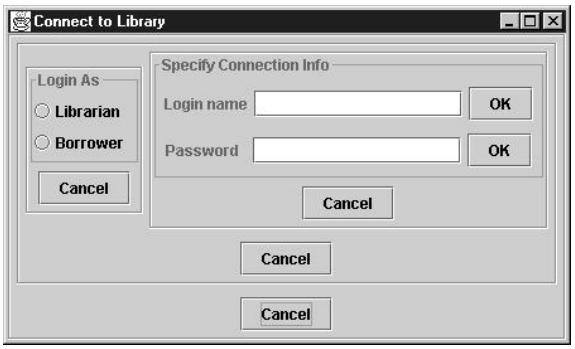

Figure 5: The UserConnection user interface.

7 Conclusions

The use of the code generator has proved that the Teallach declarative models are a suitable set of models to describe user interfaces. Only the control buttons of the composite task classes are not explicitly described in the models, but well-defined rules are provided describing when and how these buttons are used. The whole task and presentation models are used in the code generation process. Parts of the domain model are used, when required.

Library System user interfaces more complex than the UserConnection user interface has been designed and generated with success. The use of Teallach has proved that the environment is a suitable environment for the systematic development of UI since: (1) it provides an design environment where developers can model UI abstracting the details concerning the implementation of the UI; (2) the implementation time is drastically reduced if compared with the implementation time without the Teallach environment.

Further assessments and improvements of Teallach are planned. In terms of assessments, it is planned to use Teallach for the development of the UIs of a system that requires UIs more complex that those required by the Library System case study. In terms of improvements, it is planned to use UML diagrams [2] to describe the UI models.

References

[1] H. Balzert, F. Hofmann, V. Kruschinski, and C. Niemann. The JANUS application development environment — generating more than the user interface. In *Computer-Aided Design of User Interfaces*, pages 183–206, Namur, Belgium, 1996. Namur University Press.

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

[3] T. Elwert and E. Schlungbaum. Modelling and generation of graphical user interfaces in the TADEUS approach. In *Designing, Specification and Verification of Interactive Systems*, pages 193–208, Vienna, 1995. Springer.

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

[5] 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*, Edinburgh, UK, September 1999.

[6] 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*, Italy, May 1998.

[7] T. Griffiths, J. McKirdy, N. Paton, J. Kennedy, R. Cooper, P. Barclay, C. Goble, P. Gray, M. Smyth, A. West, and A. Dinn. An open model-based interface development system: The Teallach approach. In *Supplementary Proceedings of DS-VIS'98*, Cambridge, UK, June 1998.

[8] G. Krasner and S. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *JOOP*, August/September 1988.

[9] C. Märtin. Software life cycle automation for interactive applications: The AME design environment. In *Computer-Aided Design of User Interfaces*, pages 57–74, Namur, Belgium, 1996. Namur University Press.

[10] B. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, March 1995.

[11] B. Myers, R. McDaniel, R. Miller, A. Ferrency, A. Faulring, B. Kyle, A. Mickish, A. Klimovitsky, and P. Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):346–365, June 1997.

[12] A. Puerta and D. Maulsby. Management of interface design knowledge with MODI-D. In *Proceedings of IUI'97*, pages 249–252, Orlando, FL, January 1997.

[13] E. Schlungbaum. Model-based user interface software tools - current state of declarative models. Technical Report 96-30, Graphics, Visualization and Usability Center, Georgia Institute of Technology, 1996.

[14] G. Singh and M. Green. A high-level user interface management system. In *Proceedings of SIGCHI'89*, page 1989, May 1989.

[15] K. Stirewalt. *Automatic Generation of Interactive Systems from Declarative Models.* PhD thesis, Georgia Institute of Technology, December 1997.

[16] P. Szekely, P. Luo, and R. Neches. Facilitating the exploration of interface design alternatives: The HUMANOID model of interface design. In *Proceedings of SIGCHI'92*, pages 507–515, May 1992.

[17] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. Declarative interface models for user interface construction tools: the MASTERMIND approach. In *Engineering for Human-Computer Interaction*, pages 120–150, London, UK, 1996. Chapman & Hall.

[18] P. Wellner. Statemaster: A UIMS based on statecharts for prototyping and target implementation. In *Proceedings of SIGCHI'89*, pages 177–182, May 1989.

[19] C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Green. ITS: A tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*, 8(3):204–236, July 1990.