Generating User Interface Code in a Model Based User Interface Development Environment

Paulo Pinheiro da Silva, Tony Griffiths, Norman W. Paton
Department of Computer Science
University of Manchester
Oxford Road, Manchester, M13 9PL UK
+44 161 2756139
{pinheirp, griffitt, norm}@cs.man.ac.uk

ABSTRACT

Declarative models play an important role in most software design activities, by allowing designs to be constructed that selectively abstract over complex implementation details. In the user interface setting, Model-Based User Interface Development Environments (MB-UIDEs) provide a context within which declarative models can be constructed and related, as part of the interface design process. However, such declarative models are not usually directly executable, and may be difficult to relate to existing software components. It is therefore important that MB-UIDEs both fit in well with existing software architectures and standards, and provide an effective route from declarative interface specification to running user interfaces. This paper describes how user interface software is generated from declarative descriptions in the Teallach MB-UIDE. Distinctive features of Teallach include its open architecture, which connects directly to existing applications and widget sets, and the generation of executable interface applications in Java. This paper focuses on how Java programs, organized using the model-view-controller pattern (MVC), are generated from the task, domain and presentation models of Teallach.

Keywords

Model-based user interface development environments, user interface development tools, automatic code generation.

1. INTRODUCTION

The use of declarative models in the development of user interfaces shows promise in providing implementation independent descriptions of important features of an interface, such as user tasks [13]. Such declarative descriptions play the same role as declarative models in other software development activities, such as the use of entity-relationship models in database design. However, as in other contexts, the declarative descriptions, by their very nature, do not always provide a direct or natural mapping onto mainstream implementation technologies for execution.

MB-UIDEs seek to provide a setting within which a collection of complementary declarative models can be used to provide a systematic approach to the description of user interface functionality [6]. A MB-UIDE generally includes task, domain, presentation and dialogue models, and an environment in which such models can be constructed and manipulated

However, MB-UIDE technology is not yet mature, and is not yet widely used by interface developers. This paper seeks to address two issues that we believe are important to the wider acceptance and uptake of MB-UIDEs, namely the role of such systems within existing software environments, and the generation of executable programs from declarative models. These two issues are closely related, as generated user interface programs must coexist with existing software. Indeed, many new user interfaces are interfaces to existing software systems.

One of Teallach's fundamental design objectives is to maintain an open relationship to existing software systems through the APIs supported by its domain and presentation models. Although most proposals for MB-UIDEs include a domain model, it is common for domain model constructs to impinge in only a limited way on the other models in the system. One contention of this paper is that domain models should be fully available to and accessible within other models, and that the abstract domain model description used within a MB-UIDE should be able to be used as a view of existing software components. In Teallach, the domain model is the industry standard ODMG object model, which has been used to provide access both to existing object databases and to mainstream applications via class libraries.

As well as existing domain constructs, an increasing range of generic or application-specific user interface components are becoming available for mainstream programming environments, from comboboxes to molecule viewers. In Teallach, the presentation model is open, in that mechanisms are provided for importing into the system interaction components conforming to the Java Beans component architecture. This means that there is no barrier to designers selecting and reusing useful interface components.

These characteristics of Teallach's domain and presentation models provide a framework that allows user interfaces generated from Teallach models to run in conjunction with existing software systems and interface components. The Teallach code generator produces as output Java programs that are organized using the MVC design pattern. These Java programs in turn make use of software systems accessed through the domain model – in particular, Teallach is being used to construct user interfaces to object databases stored using the Poet Object Database Management System.

In general terms, the principal contribution of this paper is to describe a role for MB-UIDEs in which they bridge the gap between an underlying application and its user interface, providing user interface designers with the ability to declaratively specify the *dynamics* of the application-interface dialogue. In particular, Teallach allows a designer to define declaratively how information is passed between the

interface end-user and the underlying application, and how this information is subsequently processed. In Teallach, the application is viewed as an object-oriented domain model (as it happens, with its origin in databases, but with much in common with the CORBA object model) and the interface is an object-oriented widget set. The more detailed technical contribution is a description of how MVC code is generated from declarative models in the Teallach MB-UIDE.

More details on Teallach, other than the code generator, can be found in [5, 2], which provide an overview of the Teallach models and present the environment that is used to create and manipulate the models.

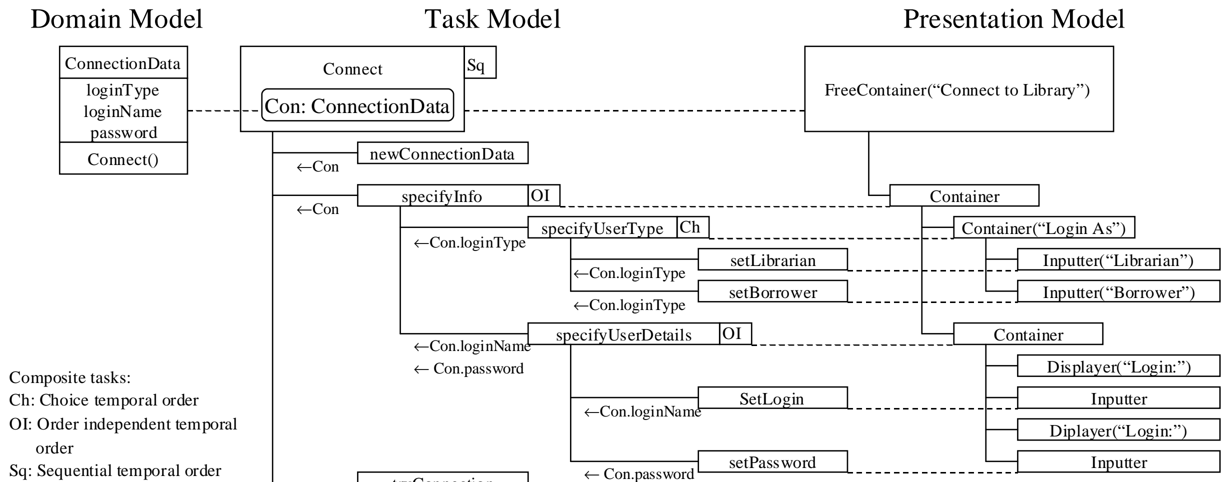

Figure 1: The Teallach Declarative Models for the ConnectUser User Interface.

2. BACKGROUND

The role that MB-UIDEs can most effectively play within the user interface development process is not universally accepted. For example, in the task-based system ADEPT [13], the emphasis is principally on the identification and refinement of user tasks, rather than on the provision of facilities for interface generation, whereas in MOBI-D [9] the task modelling process is augmented through the use of a knowledge base of interface design guidelines to produce detailed interface specifications which can subsequently be hand coded. However, most MB-UIDEs provide some mechanism for generating user interfaces from their declarative models. The generated interface can be used to validate the models, for use in evaluations, or ultimately as delivered applications.

There are three main approaches to obtaining running interfaces from declarative models:

1. *Interpreters*: the declarative models are interpreted directly, as in ITS [12].
2. *UIMS Generators*: the declarative models are mapped into alternative, generally lower-level, representations of the interface, which are subsequently interpreted, as in Humanoid [11], TADEUS [4] and FUSE [8].
3. *Source Code Generators*: the declarative models are mapped to programs in an imperative programming language, as in JANUS [1], and later versions of Mastermind [10] which generate C++ code.

Arguments can be made for and against each of the above approaches. For example, in interpreted systems the runtime environment can be tailored directly to the needs of the declarative models, but performance may be problematical in the presence of a substantial runtime system and interpretation of high level constructs. When generating code for a UIMS, the UIMS may provide useful facilities, for example, for changing the interface of an application at runtime, but may or may not support a clean or efficient mapping of all constructs from the declarative models. When generating programming language source code, generated applications have access to the full facilities of the programming language (e.g. for access to databases or remote systems), but the level of direct support for interface functionality is likely to be less than when generating UIMS code.

Although significant effort has been directed at the provision of effective facilities for producing running interfaces from interface models, interface generation is not a solved problem for MB-UIDEs. Published descriptions of approaches are often more indicative than thorough, and it is common for generated interfaces to have limited functionality. For example, although Elwert and Schlungbaum [4] present details of how TADEUS generates a UI description file for the ISA Dialog Manager, it is not shown that fully functional interfaces can be generated from the TADEUS models alone. TADEUS models contain little on how domain information is processed within the interface. There are other proposals for systems which exploit an integrated suite of models, with the ultimate objective of generating code, e.g., TRIDENT [3], but the means by which such generation will be realised are not yet described.

In particular, Teallach differs from other MB-UIDEs since its models support a description of how data flows between the user interface and the underlying application, and vice-versa. Furthermore, the Teallach domain model is not only a description of the application objects, but *it is* the underlying application itself.

This paper thus seeks to complement the earlier literature on interface generation from MB-UIDEs by providing details of how complete Java applications can be generated from a representative MB-UIDE. The scope of the generated code is also considered to be significant, in that interfaces reuse both existing domain models implemented in a database system and off-the-shelf interface components.

3. TEALLACH DECLARATIVE MODELS

In Teallach, interfaces are described using task, domain and presentation models that are constructed and related using interactive tools [2]. In contrast to other MB-IDEs, Teallach supports a flexible design method that allows designers the freedom to construct its models in any order. Once the models have been developed, the designer can automatically generate a user interface corresponding to these models

by invoking the code generator as described in this paper. The developer can either choose to accept the generated interface, or can return to the various models and continue the design process in an iterative cycle.

3.1 The Declarative Models

The Teallach models are illustrated using a simple library system case study, and in particular the task of connecting to the system, where users specify if they are connecting as librarians or borrowers, and provide a login name and password.

Figure 1 shows a set of Teallach models for the connection activity. There are three models that describe different aspects of the user interface. The *domain model* (DM) describes the data that is obtained from the user and used in the attempt to connect to the library system. The *task model* (TM) is a hierarchical representation of the tasks and subtasks undertaken, and the *presentation model* (PM) describes the visual aspects of the interface.

The DM represents application concepts as a collection of object classes described using the standard ODMG data model. This model can be used to provide access to object databases, or to Java classes from the application or from generic class libraries.

A TM is composed of a hierarchy of composite (e.g., sequential, repeatable, etc.) and primitive (action or interaction) tasks. The interaction between users and the application is performed by interaction tasks using PM components. Interaction tasks specify the nature and origin of the information they are displaying or receiving in terms of state objects.

In the example in Figure 1, the *Connect* composite task has three subtasks, and a state object *Con* of the DM type *ConnectionData*. In a more elaborate example, the Con state object could be the subject of further manipulation - e.g., to log user activity.

Presentation models are composed of hierarchies of interaction and grouping objects. Interaction objects can be concrete interaction objects (CIOs) and abstract interaction objects (AIOs). The CIOs are the widgets that compose the UI. The AIOs are abstractions of these widgets that describe if interaction objects are used for data input (*inputter*), data output (*displayer*) or both (*editor*) in presentation models. AIOs avoid premature commitment to specific presentations, and leave open the prospect of alternative visual presentations for different environments or user groups. Presentation models also use abstract and concrete grouping objects to aggregate interaction objects.

The PM in Figure 1 contains just abstract components. *FreeContainers* and *Containers* are abstract grouping objects, where *FreeContainers* are abstractions of top-level windows, while *Containers* must be specified in the context of a *FreeContainer*. AIOs have fewer properties and operations than typical CIOs, so a description of an interface in terms of AIOs is generally much more concise than one expressed in terms of CIOs. However, running interfaces use only CIOs, so AIOs are either linked to specific CIOs by the designer, or assigned default concrete representations by the system.

For example, by default, a *FreeContainer* AIO is replaced by a Swing *JFrame*. The PM can however view more complex widgets such as JTree and JTable as simpler *Displayer*, *Inputter* or *Editor* AIOs (or indeed any CIO that has been appropriately registered with the PM). Designers are however allowed to fix their preferences at any point in the design process.

3.2 Model Relationships

The TM has a central role in the integration of the Teallach models. The integration of the models makes extensive use of state objects declared in composite tasks. Each state object is an instance of a DM or a PM type. In the Teallach tool the designer can create explicit links between the models, and can then use a wizard to guide them through the process of fully specifying the link, including specifying which underlying domain methods are called and how the data passed and generated by such methods are utilized in the models. This process is fully explained in [5].

The links between the DM and the other models are restricted to the links provided by the state objects. However, additional links are required between the TM and the PM. For example, each interaction task requires one AIO, which in turn can be associated with one or many CIO. Composite tasks can be linked to grouping components in the PM.

4. RUNTIME CONTEXT

The generated user interface is coded in Java using the Swing widget set, or other imported interaction objects. The Model-View-Controller (MVC) pattern is used to organize the user interface components. A specialized class library coded in Java and using the MVC approach is also used by the generated interface.

4.1 Model-View-Controller Architecture

The MVC pattern [7] is a design pattern for the organization of user interface programs. MVC specifies how user interface software should be separated into components, each 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*. An additional benefit of the MVC pattern is that it describes the possible relationships between the component objects. In particular, the MVC pattern 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.

4.2 Class Library

The code generator produces as output Java code that exploits the Swing widget set. In addition to the basic Swing classes, a class library has been created to reduce the quantity of code produced by the code generator. The strategy is to locate as much as possible of complexity of the user interfaces code in the class library, thereby avoiding the generation of complex classes. The class library, which is illustrated in Figure 2 using UML notation, contains a class hierarchy that mirrors the task types of the TM.

This class library complies with the MVC pattern, such that an *Interaction Component* associated with a *RuntimeCompositeTask* class will always act as the MVC controller of the *RuntimeCompositeTask*'s children (which in turn act as the *RuntimeCompositeTask*'s MVC model), and a *Grouping Component* associated with a *RuntimeCompositeTask* class will always act as its MVC view. In addition, CIOs associated with *RuntimeActionTask* classes are also always controllers of *RuntimeStateObject* classes (the models), whereas CIOs associated with *RuntimeInteractionTask* classes can be MVC controllers or MVC views. In fact, if the AIO associated with the related interaction task in the declarative models is an *inputter* then the CIO is a MVC controller. If the AIO is a *displayer* then the CIO is a MVC view. If the AIO is an *editor* then the CIO is both a MVC controller and view.

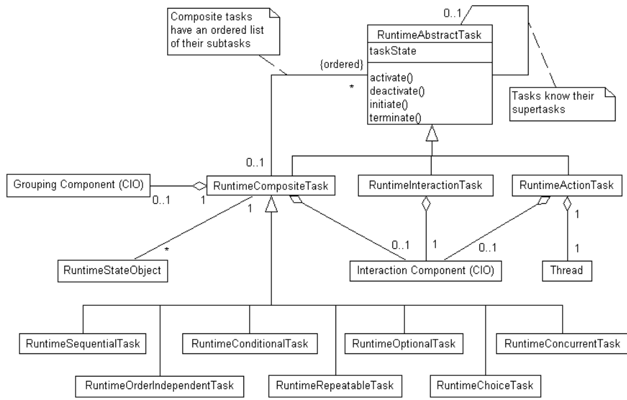

Figure 2. The Task Type Hierarchy.

TM tasks are implemented using subclasses of the class *RuntimeAbstractTask*. More precisely, composite tasks become subclasses of *RuntimeCompositeTask*, action tasks become subclasses of *RuntimeActionTask*, and interaction tasks become subclasses of *RuntimeInteractionTask*. A *RuntimeCompositeTask* can have many subtask classes. *RuntimeActionTask* provides a thread that is used to execute the assigned state object operation. Every *RuntimeInteractionTask* is associated with a CIO, and a *RuntimeCompositeTask* can be associated with a CIO.

As subclasses of *RuntimeAbstractTask*, the task classes implement the operations *activate()*, *deactivate()*, *initiate()* and *terminate()*. Basically, *activate()* enables the interaction of the user with the part of the UI responsible for the activated task. In the opposite way, *deactivate()* disables the interaction. The *terminate()* operation, usually associated with the *deactivate()* 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 it was first created. The *initiate()* operation is invoked on the children tasks of a composite task that has been activated. Specific behaviors for these operations are provided in 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 *setEnabled(true)* operation for their aggregated CIOs, when these task classes are activated.

Still in Figure 2, *RuntimeActionTask* classes and *RuntimeCompositeTask* classes can be associated with CIOs. In this case, the CIOs are called *initiators*. Initiators are required to: (1) fire action tasks that are designed to be started on demand, for example, a CIO that may be associated with the *tryConnection* action task; or (2) fire composite tasks that are subtasks of choice tasks, optional tasks or order independent tasks. For example, a user action should be recognized by a CIO in the active window in order to activate an optional task associated with another window. Through facilities provided by the PM, initiators can be customised to specify the event type (e.g., on mouse click, etc.) used to invoke the action task's associated operation.

In addition to the classes that relate directly to task types, the class library provides the class *RuntimeStateObject* for wrapping state objects. There are two reasons for wrapping a state object. (1) The user interface code must be able to reference state objects that may not yet be instantiated. However, state objects as instances of domain objects or CIOs, can be dynamically instantiated and destroyed during the execution of the user interface, so instances of *RuntimeStateObject* act as handles on the actual values. (2) There needs to be some mechanism to identify state transitions in state objects.

The necessity to identify state transitions in state objects is due to the MVC pattern, such that interaction task classes need to recognise MVC model modifications and subsequently update their views. The *RuntimeStateObject* class therefore provides an *addChangeListener()* operation that notifies the registered classes that the state of the *RuntimeStateObject* has changed. Interaction task classes that have CIOs acting as MVC views only need to register their adapter classes with the *RuntimeStateObject* classes to access this functionality.

5. CODE GENERATION PROCESS

5.1 The Generated Code

Figure 3 presents a class diagram for the user interface code generated for the example application in Figure 1. With the exception of the *Container* and *Adaptor* classes, the other classes in Figure 3 are subclasses of *RuntimeAbstractTask* from Figure 2. For example, the action tasks classes implementing action tasks, namely *tryConnectionAction and newConnectionDataAction*, are subclasses of *RuntimeActionTask*. The classes implementing interaction tasks, such as *setBorrowerInteraction* and *setLoginInteraction* are subclasses of *RuntimeInteractionTask*. The classes implementing composite tasks are indirect subclasses of *RuntimeCompositeTask* – for example, *specifyInfo* is a subclass of *RuntimeOrderIndependentTask*.

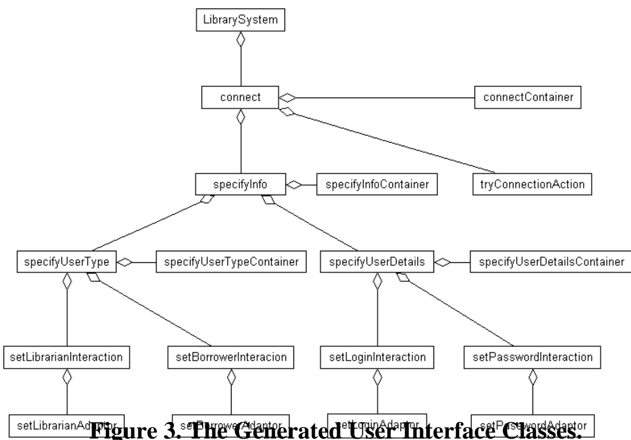

Figure 3. The Generated User Interface Classes.

The class *LibrarySystem* implements the *main()* method that invokes the *activate()* operation of the *connect* root task. This task activation invokes the *initiate()* operation in the root task subclasses, that in turn invokes the *initiate()* operation in their subclasses, and so on. As *connect* is a *RuntimeSequentialTask* class, it initially only activates the *specifyInfo* class invoking its *activate()* operation. When the *specifyInfo* class finishes normally (without being cancelled), its *deactivate()* and *terminate()* operations are invoked. The *deactivate()* operation mainly sets the *taskState* attribute of the *RuntimeAbstractTask* class (shown in Figure 2) to idle. The *terminate()* operation generates a change event that is listened for by the *connect* class, so that on detection it can activate its next subtask. In this case, the *tryConnectionAction* class is activated. On completion of the *tryConnectionAction* class, the *connect* class also terminates since it does not have more subtasks. Having identified that the *connect* class has finished, the *LibrarySystem* class finally closes the application.

5.2 Components of Code Generator

The code generator consists of four modules, 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. In the example, this is the class *LibrarySystem*.

The *StateObjectMapper* identifies the state objects that are accessible within each composite task. In essence, a state object defined in a composite task is visible in all descendent tasks. *StateObjectMapper* performs a preorder traversal of the TM populating a mapping table associating each task with the state objects visible in the task. For example, *StateObjectMapper* identifies that the *Con* state object declared in the *connect* task (Figure 1) must also be accessible by the *specifyInfo* task since the *Con* object is also used by the *specifyInfo* subtasks.

The *CompositeTaskGenerator* performs another preorder traversal of the TM, during which the *CompositeTaskGenerator*: (1) generates the code for the visited composite tasks; (2) invokes the *PresentationGenerator* for the composite tasks that are explicitly linked to a grouping component of the PM; (3) invokes the *ActionTaskGenerator* for action tasks; and (4) invokes the *InteractionGenerator* for interaction tasks. The standard code for a composite task class creates an array of subtasks that are invoked according to the temporal relation specified by their parent task. The composite task classes receive an array of *RuntimeStateObject*s that are the state object wrappers, and forward an array of *RuntimeStateObject*s to each subtask, according to the mapping table created during the execution of the *StateObjectMapper*.

The *PresentationGenerator* is invoked for a specific grouping component of the PM. During an execution of the *PresentationGenerator*, it performs a breath-first traversal of the PM identifying the immediate children of the provided grouping component. Code is generated for the provided grouping component and its children that are interaction components. The children that are grouping components are not considered during this execution of the *PresentationGenerator*, but in a subsequent call of *PresentationGenerator* by *CompositeTaskGenerator*. The generated code is composed only of CIOs. Therefore, wherever the designer has not associated an AIO with a CIO, the code generator applies the default mapping. For example, the *specifyInfoContainer* class does not specify a CIO since its children components are also grouping components. In contrast, the *specifyUserDetails* container class (shown in Figure 1) has four CIOs, and is itself associated with a grouping component.

The *ActionTaskGenerator* utilises the information provided by the domain operation associated with an action task. If the designer has specified that an action task is *automatic*, then the action is automatically fired when activated, otherwise a CIO is used to fire the action. The *ActionTaskGenerator* is also responsible for identifying any state objects necessary to store an operation's arguments and the result of the operation.

The *InteractionGenerator* is responsible for mapping how data is displayed or input in the generated interface. The CIO associated with an interaction task acts as a MVC controller or view (or both depending on the interaction type), whereas a state object acts as the MVC model. Since DM state objects often are frequently the objects of an object database, the *InteractionGenerator* usually relates the database objects with Swing components so that database data may be openly interacted with.

5.3 Extended Interaction based on Task Model Semantics

There are generally widgets in generated interfaces that are not modeled explicitly in the PM of the application. These widgets are used for controlling composite tasks. The three main control activities are:

1. *Confirmation*: allows the user to indicate that a task has been completed; this is normally depicted using an *OK* button.
2. *Cancellation*: allows the user to exit a task without completing it; this is normally depicted using a *Cancel* button. The semantics of the *Cancel* button depends on the category of the parent task of the composite task class that is been cancelled. If the parent task class is a choice task class, the parent task class is restarted invoking its *activate()* operation. If the parent task class is an optional task class, the current task class is finished invoking the *deactivate()* operation, but not the *terminate()* operation. If the parent task class is a concurrent, repeatable or an order independent task class, the current task class is restarted invoking its *activate()* operation. Finally, if the parent task class is a sequential task class, the parent task class is finished invoking both its *deactivate()* and *terminate()* operations.
3. *Change Task*: allows the user to swap from one concurrent task to another; this is normally depicted using a menu or a combo box to be used as with a cardstack metaphor.

Different task types have different behavior in terms of confirmation and canceling; in Teallach there are default controls generated for the different task types, but some of the defaults can be overridden by the designer.

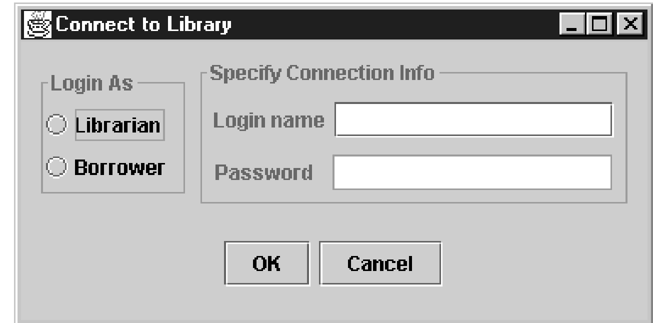

Figure 4. The UserConnection User Interface.

The interface produced for the models in Figure 1 is given in Figure 4. Here it can be noticed that the code generator has assumed the default mappings for each AIO to its default CIO.

6. RESULTS OF INITIAL EXPERIMENTS

One of the longer-term goals of the Teallach project is to evaluate both the Teallach tools and models, the flexible design method, and the generated interfaces in terms of their functionality and usability. At the present time we have only limited results on each of these factors from the experiences of two developers. In terms of the generated code for the small example illustrating this paper, we have some initial results which indicate that the design lifecycle for interfaces produced using Teallach is shorter than for interfaces produced using either hand crafted code or using an integrated development environment, and that subsequent changes to interface functionality (e.g., changing a sequential task to an order independent task, or changing the data type of displayed data) are easily supported. The result of this is that an interface design can be tested with an end user and amendments to the models can be presented to them in just the time it takes to re-compile the code.

When presented with this simple case study, an expert developer implemented the code in 776 lines of code in a few days. Using Teallach, the same interface was developed in a couple of hours with no actual writing of code. Since the generated code utilises the Teallach runtime library, it consisted of only 353 lines of code. Although it is not intended that the code generated by Teallach will be used as *pro-*

duction code, there are indications that the reduction of application-specific code may well improve the maintainability and testability of the code.

While we do not yet have any empirical evidence to support our claims that UI development is made simpler using a MB-UIDE such as Teallach, initial observations have shown that the time taken to learn the UI development process using Teallach is much shorter than the time taken to train a computer scientist how to program using Java and the ancillary APIs. Moreover, developers were able to easily improve the produced UI code just by refining the models.

7. CONCLUSIONS

The experience of developing a code generator for Teallach has illustrated that its models contain sufficient information to allow fully functional interfaces to be generated. In addition, since designers can either leave most presentational aspects of the generated interface to system defaults or can make explicit assignment of CIOs to tasks means that interface generation in Teallach can adapt to the preferences of different designers. In fact, as described in [2], an important feature of interface development in Teallach is the flexibility provided by the development environment – for example, the design tools allow models to be developed and linked in different orders and ways, reflecting different design preferences.

An additional distinctive feature of Teallach is the mainstream setting within which it operates. Java code produced by the code generator interacts with existing object databases and application class libraries through the DM interface, and makes use of existing Swing or designer-supplied interaction objects that are accessed through the PM.

Although interface builders that allow developers to construct applications by drawing displays using components such as those provided by Swing are in widespread use, it is still the case that applications constructed using such environments are often complex to code, paying no attention to the dynamics of the interface.

MB-UIDEs have received considerable attention from researchers working on computer-aided design of user interfaces in recent years, as the use of specialized declarative models can assist developers in partitioning a substantial design activities into manageable chunks. However, proposals for MB-UIDEs often fail to provide comprehensive interface generation capabilities, or generate interfaces that are difficult to link to existing applications or component sets. Furthermore, there are few detailed descriptions of code generation for MB-UIDEs in the literature.

This paper has described a code generator for the Teallach MB-UIDE that generates Java applications using the MVC design pattern, whose interfaces can exploit the Swing widget set. The contention of the paper is that the provision of effective interface generation facilities in mainstream settings is important to the long-term uptake of model-based interface development techniques.

8. ACKNOWLEDGMENTS

This work is funded by the UK Engineering and Physical Sciences Research Council, whose support we are pleased to acknowledge. The first author is sponsored by Conselho Nacional de Desenvolvimento Cientifico e Tecnologico – CNPq (Brazil) – Grant 200153/98-6, whose support the first author is pleased to acknowledge. We also thank our partners on the Teallach project for their contributions to the development of the overall Teallach system. They are Peter Barclay, Richard Cooper, Carole Goble, Phil Gray, Jo McKirdy, Michael Smyth and Adrian West.

9. REFERENCES

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

[2] Barclay, P., Griffiths, T., McKirdy, J., Paton, N., Cooper, R. and Kennedy J. The Teallach Tool: Using Models for Flexible User Interface Design. In *Proceeding of CADUI'99*, Kluwer.

[3] Bodart, F., et al., Towards a Systematic Building of Software Architecture: the TRIDENT Methodological Guide. In *Proc. DSVIS'95*, Vienna, Springer, 1995, pp. 237-25

[4] Elwert, T., Schlungbaum, E. Modelling and generation of graphical user interfaces in the TADEUS approach. In *Proc. DSVIS'95*. Vienna, Springer, 1995, 193-208.

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

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

[7] Krasner, G., Pope, S. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*. 1 (3), 1988, 26-49.

[8] Lonczewski, F. and Shreifer, S., The FUSE System: an Integrated User Interface Design Environment, In *Proc. CADUI'96*, 1996, 37-56.

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

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

[11] Szekely, P., Luo, P., Neches, R. Facilitating the exploration of interface design alternatives: The HUMANOID model of interface design. In *Proceedings of SIGCHI'92* (May 1992). 507-515.

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

[13] Wilson, S., Johnson, P. Bridging the generation gap: From work tasks to user interface designs. In *Computer-Aided Design of User Interfaces*. Namur University Press, 1996, 77-94.