Teallach: A Model-Based User Interface Development Environment for Object Databases

Tony Griffiths[1], Peter J. Barclay[2], Jo McKirdy[3], Norman W. Paton[1], Philip D. Gray[3], Jessie Kennedy[2], Richard Cooper[3], Carole A. Goble[1], Adrian West[1] and Michael Smyth[2]

[1]Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK.
Email: { griffitt, norm, carole, ajw} @cs.man.ac.uk

[2]Department of Computing Studies, Napier University,
Canal Court, 42 Craiglockhart Avenue, Edinburgh EH14 1LT, UK.
Email: { pjb, jessie, michael} @dcs.napier.ac.uk

[3]Department of Computing Science, University of Glasgow,
Glasgow G12 8QQ, UK.
Email: { jo, rich, pdg} @dcs.gla.ac.uk

Abstract

Model-based user interface development environments show promise for improving the productivity of user interface developers, and possibly for improving the quality of developed interfaces. However, model-based techniques have rarely been applied to the important area of database interfaces. This lack of experience with data intensive systems may have led to model-based projects failing to support certain requirements that are essential in data intensive applications, and has prevented database interface developers from benefiting from model-based techniques. This paper presents a model-based user interface development environment for object databases, describing the models it supports, the relationships between these models, and the tool used to construct interfaces using the models.

1. Introduction

In many organisations a significant proportion of the user interfaces developed are user interfaces to databases [20]. Although most commercial interface development packages include facilities tailored for use with databases (e.g., [2]), and most database vendors supply interface building tools (e.g., [16]), such products rarely build upon the most recent research activities on user interface development environments, and provide little support for specifying the dynamics of the developed interface. This paper provides an overview of the Teallach project, which is developing a model-based user interface development environment for use with object databases.

Model-based user interface development environments (MB-IDEs) [19] seek to describe the functionality of a user interface using a collection of declarative models. In such a context, constructing a user interface involves building and linking a collection of models. In practice, these models often include, but may not be limited to, a domain model, a task model, a dialogue model and a presentation model. It can be argued that the model-based approach improves upon current component-based programming environments, for example, by making the interface development process more systematic, and by shielding developers from certain low-level aspects of interface implementation through judicious use of abstract models. The features of MB-IDEs are discussed more fully in Section 2. At this point, we note only that MB-IDEs are the focus of significant attention in the user interface development environment community, and that very little work has been done to date that seeks to bring this distinctive technology to bear on database problems [8].

Object databases are now in reasonably widespread use in a range of advanced applications [5]. These applications are typically characterised by the presence of rich information models and complex processing or analyses. User interface requirements vary widely from application

to application, but many applications use complex custom-built visualisations (e.g. in geographic, scientific or design domains), which indicates a need for an open architecture into which different visual components can be incorporated. Furthermore, many applications must make use of operations associated with database objects, and displays often have to present information that is not directly associated with a small number of persistent objects. This means that interface development environments for object databases must incorporate effective facilities for communication and temporary storage of application data within the interface.

This paper describes how techniques from MB-IDEs can be used to support the development of user interfaces to object databases. In particular, the paper provides an overview of the models and tools of the Teallach system, which is a MB-IDE designed specifically for use with object databases. The development of a MB-IDE in this context has highlighted a number of limitations in earlier systems, and thus hopefully advances the area of model-based user interface development, as well as bringing recent interface development techniques into contact with data intensive applications.

The paper is structured as follows. Section 2 provides an introduction to model-based user interface development environments. Section 3 describes the Teallach models, of which there are three – a domain model, a task model and a presentation model. Section 3 also describes how the models relate to each other, which is important to the functionality of the overall environment. Section 4 provides an overview of the method that directs interface development, and the application development tools that support the method. Section 5 presents some conclusions.

2. Model-Based User Interface Development Environments

Early interactive design environments [3,12] provided programmers with a way of developing user interfaces semi-automatically via a specification of the dialogue, or interactive behaviour, of the run-time system. Although clearly superior to manual coding in a general-purpose language, these systems generated rather crude interfaces, and the designer is left with the substantial job of configuring the system to take into account the nature of the application data and operations, the resources of the graphics or window system, and the abilities of users. Over the past ten years, one strand of research has focused on enhancing the user interface design environment with additional information organised into components which model design-relevant aspects of the target system and its proposed context of use. The MB-IDEs arising from this work exploit the information in their models to provide full- or semi-automated generation of user interface prototypes, and design assistance tools such as computer-based design critics, simulators and model-checkers.

MB-IDEs may be compared with respect to the number and expressiveness of their models, and the design tools which exploit the model-based information. For comparisons of collections of proposals, see [15, 8].

Early MB-IDEs, such as UIDE [6] and Humanoid [17], incorporate relatively simple application, dialogue and presentation (i.e., user interface component) models, plus a rule-based user interface generator. ADEPT [9] adds a model of user-system tasks, at a higher level of abstraction than dialogue, to produce prototype interfaces for evaluation and subsequent refinement. DRIVE [11] includes a user model that captures capabilities and limitations of the user population for automatic selection of an appropriate interaction technique from several candidates. As well as using sophisticated application, task, dialogue and user models, TADEUS [15] partitions presentation specification into abstract and concrete presentation models, representing respectively the dialogue role and the toolkit-dependent appearance and behaviour of interface objects.

In addition to semi-automatic interface generation, a variety of forms of design assistance can be found in the family of MB-IDEs. Humanoid produces application-related and context-sensitive help and undo/redo. TRIDENT [1] uses a knowledge-base of design guidelines to offer to the designer a set of appropriate presentation components plus a human-readable rationale for the selection. TADEUS's dialogue model can be translated into a Petri Net for simulation and model-checking.

With the exception of some highly domain-specific applications such as those found in the avionics industry, MB-IDEs have yet to receive acceptance in an industrial setting. The most common tool for interface construction remains the GUI-builder, which typically holds no information about the components of the system beyond the composition and configurable properties of interaction objects. It has been hypothesised that model-based systems may not yet be expressive enough [14]. Clearly, the models must be able to capture sufficient relevant information to provide useful assistance, and to do so via a specification language and tools which retain a positive cost/benefit ratio for the developer. Also, generation rules may produce poor initial interfaces via automatic generation. A MB-IDE must support, or at least accommodate, the construction of user interfaces as sophisticated as those which can be built by other means. Additionally, the tools must not impose an unacceptable method of working on developers (e.g., a rigidly top-down approach which leaves prototype construction until the end of the design process [8]).

3. The Teallach Models

3.1 The Domain Model

Teallach assumes the pre-existence of the underlying application for which a user interface is to be developed. The application may take the form of standard Java classes or may involve the services of an object-oriented database management system (OODBMS). Irrespective of its guise, if the application is to inform and link into the user interface, it must be modelled within the Teallach system. The Teallach domain model is the means by which this is realised. In other words, the domain model reflects the structure and functionality of the underlying application and, where applicable, database connectivity and interaction.

3.1.1 The Structure of the Domain Model. With a view to providing a measure of platform independence, the domain model represents applications in terms of the concepts specified in the ODMG object database standard [4]. The ODMG specification lists a collection of interfaces covering a range of issues, including the fundamental methods used for manipulating every component type in the model.

The domain model is essentially a meta-model that realises each of the ODMG concepts as an instantiable class. For example, consider the metadata for `Operation`, which is provided by ODMG to describe the methods available on a type or class. To realise this, Teallach has defined a class called `dm_Operation` that captures the ODMG specification for `Operation`, including its relationships with multiple `Parameters`, a result `Type` and a list of `Exceptions`.

A similar process was followed for all the constructs of the ODMG object model, resulting in a collection of classes that are the core building blocks for domain models. An individual domain model is a collection of related instances of these building block types as dictated by the structure and functionality of the underlying application.

The principal role of the domain model is to represent the underlying application, and database connectivity and interaction. In addition, however, it models *auxiliary* data types that may be required to describe transient data vital to the runtime operation of the application and interface. Examples of auxiliary domain information include class libraries that may be needed to manipulate the data input to the system via the user interface before it is used directly with the underlying application. These facilities are required for the runtime operation of the user interface, but are not an inherent part of the application itself. Auxiliary data is also modelled using the ODMG derived building blocks in order that the representation of domain components is orthogonal to their source and persistence.

3.1.2 Using the Domain Model. Where Teallach is being used to build an interface to an OODB application, the schema of the database has to be analysed and converted into a domain model using the building blocks discussed above. Of those OODBMSs that claim ODMG compliance, few, if any, have adopted the ODMG standard schema definition language ODL as the means by which the schemas of their databases are specified, and instead have devised individual means for schema definition. This in turn means that Teallach, via the domain model, has to interface to each ODMG compliant OODBMS using a different mechanism. Teallach is currently focusing on POET [13]. POET's specific manner of schema description uses a configuration file that lists, amongst other things, the name of the application database and schema, and the fully qualified names of Java classes that are to be persistent within the application database. The schema information is extracted from a POET database in a two stage process. Firstly, the configuration file is parsed to ascertain the names of the persistent classes defined for the database. Secondly, each of the classes is then introspected over to discover its internal structure in terms of its fields, methods and exceptions. This information is then used to guide the construction of the appropriate instances of the building block types mentioned previously. Figure 1 shows a very simple class called `Person` which has a single field of type `String`, a constructor method, and two access methods. Given this information, the model as shown in figure 2 is constructed to represent the class `Person` within the domain model.

```
public class Person
{
    private String name;
    public Person(){name = “”;}
    public String getName() {return(name); }
    public void setName(String n){name = n;}
}
```

Figure 1. Definition of class `Person`

When modelling the auxiliary types mentioned previously, introspection over the required classes is again used to inform the instantiation of the appropriate domain model constructs.

3.2 The Task Model

The task model provides support for modelling both the structure of user tasks and the flow of information between the models when carrying out the user's tasks.

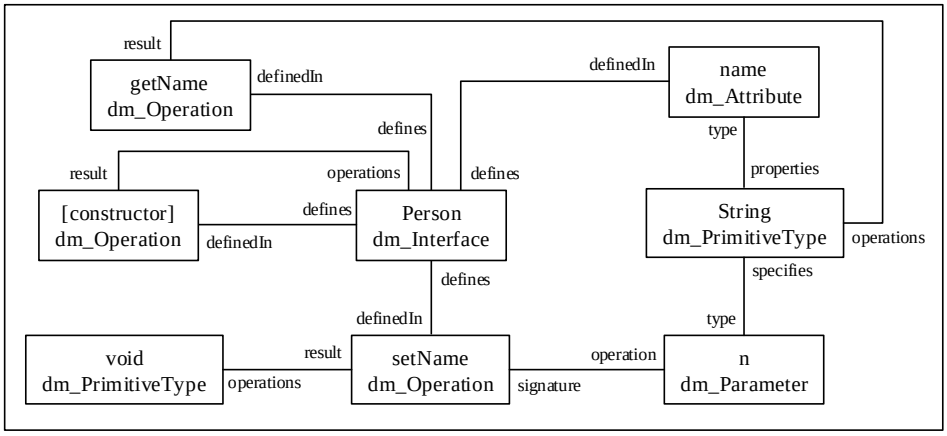

Figure 2. Domain Model of class Person

3.2.1 Basic Task Model Structure. The task model shares its basic structure with the task models of several MB-IDEs such as Adept [10], Mastermind [17] and TADEUS [7]. An example of a task model in the context of the Teallach tool is given in figure 3. The task model is a goal-oriented task hierarchy, with its leaf nodes (termed *primitive* tasks) representing interaction or action tasks. The temporal relationship between sibling tasks is specified by their parent task.

The task model provides seven temporal relations, namely:

- *Sequential:* The subtasks must be performed in the specified order; all subtasks must be completed before the task's goal is considered achieved.
- *Order-independent:* The subtasks may be performed in any order; all subtasks must be completed before the task's goal is considered achieved.
- *Repeatable:* The sub-tasks are repeated a specified number of times, or until a condition is satisfied.
- *Concurrent:* The subtasks are performed in parallel; all subtasks must be completed before the task's goal is considered achieved. Two kinds of concurrent task are distinguished by a flag: truly concurrent tasks occur at the same time, while interleaved tasks have only one sub-task at a particular time, although all are progressing.
- *Choice:* The user must decide which one of the sub-tasks is to be performed; the chosen subtask must be completed before the task's goal is considered achieved.
- *Optional:* Zero or more (including all) of the subtasks of a choice task may be chosen. This task type therefore has an implicit concurrency controller; all chosen subtask(s) must be completed before the task's goal is considered achieved.
- *Conditional:* There exists a choice between sub-tasks that is dependent on a specified condition.

3.2.2 State Information and Information Parameters. One of the main shortcomings of many MB-IDE task models lies in their inability to represent and handle user or application information flow within the user interface design. The task model therefore allows the declaration of local state, and the association of this state with a task. Once specified, the designer can indicate how this state information is set and utilised by tasks in the task model.

State information can be associated with any non-primitive task, with the associated task providing its scope. A state object has type information corresponding to a domain model class definition, and can therefore correspond to either an application class, a programming language class (such as those provided by a Java library), a user defined class, or a presentation model class. In the task model, information flow is specified by utilising a task's input and output parameters. In general, the output of a task is linked to the state of its parent task.

Since Teallach's primary concern is interface development for object database applications, the task model requires a means of creating and invoking database-specific functionality such as sessions, transactions and queries. Since the domain model provides a description of these concepts in terms of the ODMG meta-model, the task model can create state objects which correspond to database sessions, transactions and queries; the required functionality can be invoked through calls to a state object's methods. For example, a state object `q: Query` associated with a non-primitive task can be invoked through its `q.execute()` method.

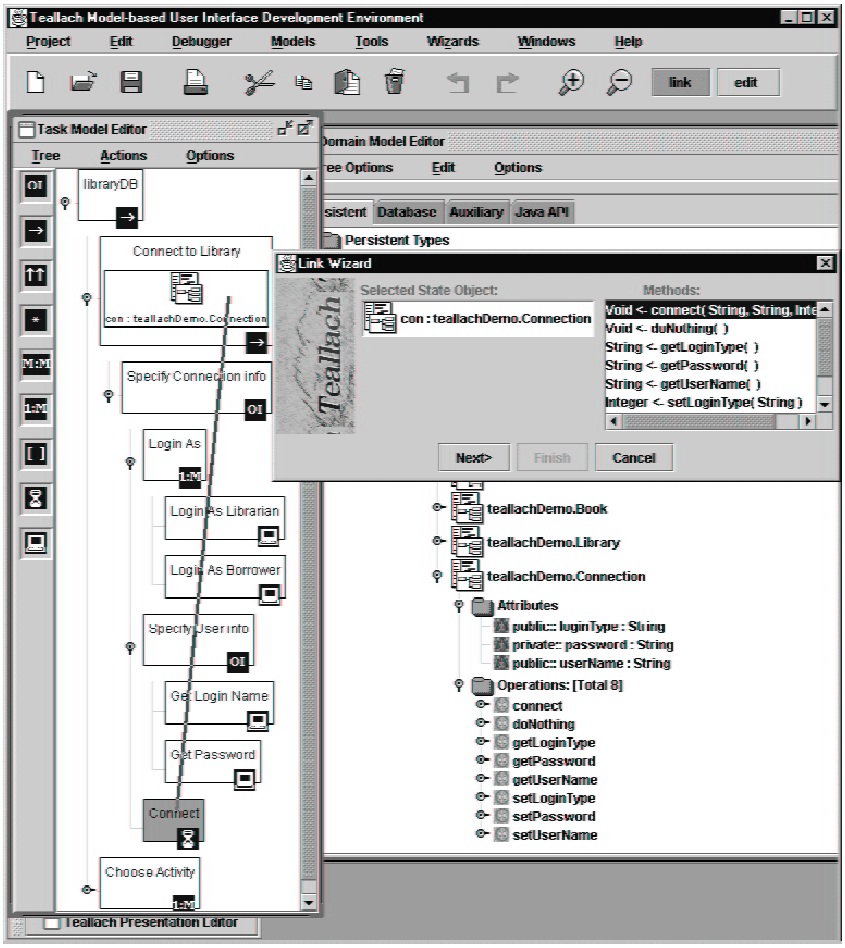

Figure 3. Screen shot of Teallach tool, showing a link between the task and domain models

3.2.3 Primitive Task Types. The lowest level building blocks of the task model are its primitive tasks. While a primitive task can be created in the task model, there may be a link to a corresponding domain model operation (either from the application database, a user-defined type, or a programming language primitive type). There are several such task types, which can generally be classified as action or interaction tasks.

An *action task* corresponds to some low-level activity carried out by the application. An action task is associated with a domain model operation.

An *interaction task* involves some degree of human-computer interaction, either by the user providing input to the computer, the computer providing feedback to the user, or a combination of the two. An interaction task is associated with a presentation model component.

3.2.4 Exception Handling. Once a task model primitive task been linked to a domain model operation, the task model can ask the domain model if the operation can raise any exceptions. Typically this results in the designer specifying where in the task hierarchy control flow will continue once the exception has been handled. Any information associated with the exception is passed along the control flow.

3.3 The Presentation Model

Teallach's presentation model is used to describe the presentation of the user-interface being designed; in other words, it describes the appearance and surface behaviour of the generated user interface. There are two levels of presentation, a concrete presentation model and an abstract presentation model.

3.3.1 The Concrete Presentation Model. The elements of the concrete presentation model are the widgets of the toolkit being used to build the user-interface. Teallach uses the Java Swing widget set. There are several advantages to using Swing, as it provides a rich set of widgets, has an efficient underlying implementation, and has a *pluggable look-and-feel*, facilitating the construction of different styles of interface. Teallach also allows the use of other, custom widgets when building interfaces; this is essential if

we are not to rule out domain-specific widgets that particular users may require (e.g., a standard molecule-viewer application). Teallach uses the Java Beans conventions as an adapter between itself and external widgets to allow such user-supplied widgets to be registered within the presentation model. Fragments of interfaces that the user has created earlier may also be saved for later reuse.

The concrete presentation model is represented in two ways: there is a *GUI view*, which may be considered to be a "preview" of part of the user-interface being constructed; and there is a *tree view*, which represents the interface as a hierarchical structure, based on the containment hierarchy of its components. The tree view shows a higher-level view of the concrete presentation, where the types and interrelations of widgets are shown, but details such as position, font and colour are omitted. The tree view is contained in the main Presentation Editor window, whereas the GUI view is free-floating. Both views are shown in figure 4.

The Teallach user may construct parts of interfaces by hand, or use automatic generation to create parts that are subsequently modified interactively. To facilitate the first of these approaches, a palette of Swing widgets is provided, which may be inserted into the interface as desired (Figure 4 shows a subset of the palette, for demonstration purposes; a complete, customisable, free-floating palette is to be added). Methods for automatically generating presentation fragments from fragments of other models are described in Section 3.4.

3.3.2 The Abstract Presentation Model. Teallach's abstract presentation model defines abstract categories to which concrete presentation widgets are assigned. For example, radio-buttons, a drop-down list and a scrollable list are all concrete widgets that may allow the end-user to choose one option from several; hence, they are all examples of an abstract category *Chooser*. The abstract presentation model pre-defines a number of such standard abstract categories, such as *Displays* (for showing data), *Editors* (for changing data), and *ActionItems* (for invoking behaviour). Furthermore, Teallach users may customise the model by adding their own categories. Note that a single concrete widget may be associated with more than one abstract category.

These abstract categories have features that characterise all their members. To use a widget with the abstract presentation model, is it necessary first to register it. Registering a widget involves declaring the category to which it belongs and, for each category, specifying the relationship between the properties of the concrete widget and the features of the abstract category. For example, the abstract `TextDisplay` category has the feature `input`, representing the text to be displayed. To use a Swing `JTextField` widget as a `TextDisplay`, it is necessary to indicate that the `JTextField`'s input is provided by its `setText` method. Therefore, the registration requires specification of the mapping `TextDisplay.input -> JTextField.setText`. Such mappings are specified in a script file, which the user can author directly. It is intended to add a Registration Wizard to Teallach that guides the user through this registration process and generates the script file automatically. However, only user-supplied custom widgets need be registered by the user; in the Teallach system, the widgets provided by Swing are pre-registered with the standard categories.

In the current prototype, the abstract presentation model may be shown in the presentation editor's tree view, by rendering the abstract category of a widget instead of its concrete Swing type; the user may toggle between the abstract and the concrete views of the presentation.

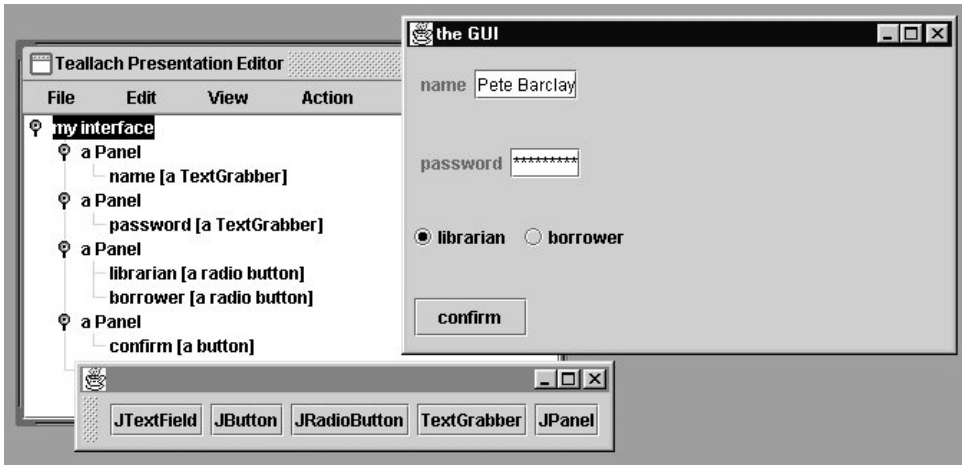

Figure 4. Example concrete presentation model views

A benefit that derives from the use of the abstract presentation model is that (a notion of) *style* is supported in the design of the end-user interface. For example, where several different concrete presentations may represent the same abstract presentation (as in the chooser example above), a choice must be made; a collection of such choices may be collected together into a *style-sheet*. A style-sheet defines a named style as a collection of such choices, which may then be applied to a fragment of an interface. The style-sheet also describes any fixed customisations of widgets to be used within the style it defines. This approach provides three advantages: style decisions can be applied consistently across an interface; when mapping between models, presentation fragments can be generated in terms of abstract categories, which are then made concrete by applying an appropriate style; and the designer is relieved of the need to specify the same choice repeatedly.

Style-sheets are again to be represented as script files showing mappings, but a planned style editor will not only allow the user to view and apply different styles, but will also contain a Style Wizard to guide the user through the process of creating additional styles.

3.4 Interaction Between Models

This section briefly reviews the ways that the three Teallach models can interact. There are two ways of associating elements from different models: *linking* and *deriving*. In *linking*, an association is made between existing components in two models. In *deriving*, components in one model are constructed based on components in another; the derive mode also creates associations between the newly created component and the source of the derivation. In the modelling environment, linking and deriving are different modelling tasks, expressed using different interaction techniques. The following subsections outline the ways in which components from the different models can be associated and derived. It should be noted that the domain model is fixed, and as such cannot be changed by a derive operation.

3.4.1 From Domain to Task. Linking a domain operation to a leaf task specifies that the task will be an invocation of this operation on a particular *instance* of a domain model class `T:<Type>` that defines the operation. The designer must therefore first construct a new state object `t:T` in the leaf tasks's immediate parent, and then link the leaf task to the state object. Teallach then uses a Link Wizard to guide the designer through the process of creating the link. Such a link is shown between the '*Connect*' task and the `con:teallachDemo.Connection` state object to create the link to the `con.connect(String, String,Integer)` method in Figure 3.

Using a domain class as the source of a derive operation with a non-leaf node of the task model as its target creates a new task structure, rooted in a new sub-task, that will perform the default action (usually edit) on the domain class. The public operations and attributes of the class are added as subtasks to this new task. Figure 6 shows the task hierarchy generated by using the *Librarian* class from the domain model to derive a new sub-task (called *edit librarian*) in the task model.

3.4.2 From Presentation to Task. Linking a node of the presentation model to a node of the task model specifies that an *instance* of the presentation object is to be used to visualise the task. An example could be the linking of a pre-built, custom editor for *Librarian* instances to the task '*edit librarian*'. In this case, both elements are considered as leaves, as we do not wish to view them at a finer granularity. The designer will therefore create a state object which corresponds to an instance of the presentation model information in the task model and link using the Link Wizard.

Deriving a non-leaf node in the task model from the information contained in a presentation model node generates a new task structure which describes the dynamics of the presentation object. For example, using the librarian editor shown in Figure 5 as the source of the derive operation with the root of the task model as the target would generate the task structure shown in Figure 6.

3.4.3 From Task to Presentation, Linking a leaf of the task model to a leaf of the presentation model has the same effect as creating a link in the opposite direction, as described in Section 3.4.2.

A task model node can be used to derive the structure of a presentation model node whose purpose is to create a default presentation for the task sub-hierarchy. For example, the login dialog shown in Figure 4 could be derived from a task structure like the one shown in Figure 3.

3.4.4 From Domain to Presentation. Domain items may be used to generate presentation items directly, or may be linked to them through the task model. For example, linking a task *set salary* (which uses a method `setSalary(int amount)` defined in the domain model) to a presentation leaf representing an action-item (eg, a button) causes the task, and hence the domain method, to be invoked by the presentation item. If the presentation item is not capable of invoking behaviour, an error is reported. If the operation is associated with a non-leaf presentation, a new action-item (leaf node) is added to the presentation, which can be used to invoke the task.

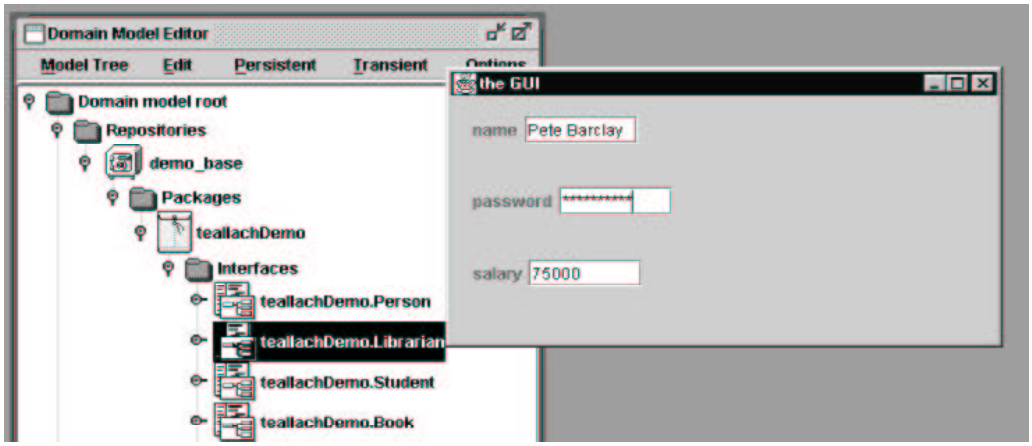

Figure 5. Basic editor for *Librarian* generated from the domain model

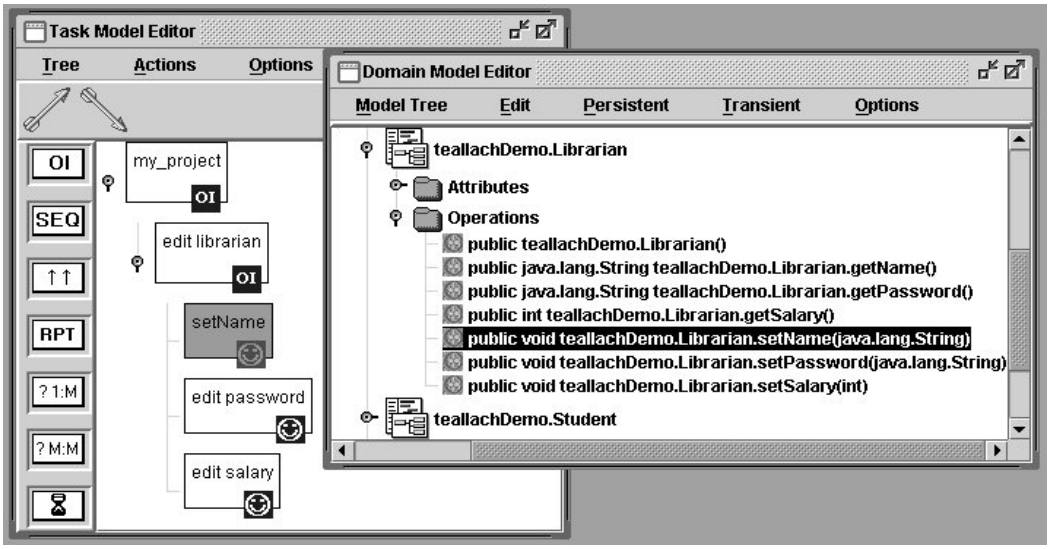

Figure 6. Task hierarchy generated from a domain class

Deriving a node in the presentation model from the information contained in a domain model class (e.g., *Librarian*) creates a basic default editor for *Librarian*. This editor is constructed by examining the get- and set-methods of the class, and inserting appropriate display or edit fields in the generated presentation object. Figure 5 shows the result of deriving the *Librarian* class from the domain to the root of the presentation model. Of course, the user may wish to treat this generated object only as a starting point, and may subsequently modify it.

4. The Teallach Design Method and Tools

4.1 The Method

MB-IDEs typically support a single fixed method for developing their component models, frequently stipulating that their task model must be constructed before any other model. This results in an inflexible interface development lifecycle, imposing a methodology that not all developers will be comfortable with.

Teallach attempts to circumvent this problem by removing constraints on the order in which its three models must be constructed. This flexibility removes restrictions on the order in which links between related model components can be defined. For example, one mode of working could be to create a high level task model, and then to use the presentation model to articulate each of these high level tasks. Once this process has been completed, the designer must then specify how the presentation model's designs relate to the task model's high level tasks by linking the related components, as described in Section 3.4.

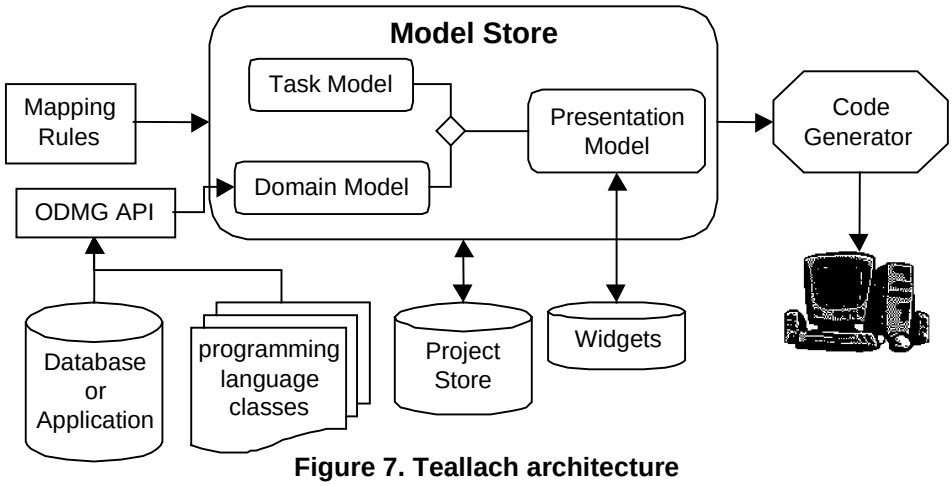

Figure 7. Teallach architecture

Teallach's ability to provide clean and transparent access to its underlying application stems from the domain model's abstract view of the application. This shields the remaining Teallach models from the complexities and idiosyncrasies of external systems. Although there is no stipulation that the domain model must be constructed before the other models, Teallach assumes that all references to domain model concepts must be resolved before an interface can be generated.

Figure 7 presents an overview of the Teallach architecture, which has been heavily influenced by the requirement to support a flexible methodology. Central to this architecture is the Teallach model store, which provides several core facilities, including:

- *Model linkage information*: The store is responsible for maintaining links created by the designer between related model components.
- *Mapping information*: The store acts as a centralised repository of information about the possible mappings between the Teallach models. These mappings are utilised when automatically generating model fragments in one model from the information captured in another.

Once models have been constructed and linked, the user interface to the underlying application can be generated. This is achieved through the use of a code generator, which outputs Java programs corresponding to the developed models. During the process of code generation the links between the Teallach models are verified, and calls to underlying domain data and functionality are resolved.

4.2 The Tools

Teallach provides an integrated toolkit that a designer can use to construct and link the individual models that form the model-based UI design. This toolkit supports Teallach's flexible design method. As shown in figure 3, the Teallach toolkit consists of editors for the three models within an overall tool environment providing project management, editing, model linking and code generation facilities. The toolkit uses a desktop metaphor and direct manipulation techniques for model construction.

As previously discussed, a designer can create links between related model components. In the Teallach tools this is achieved by switching to *link mode* and drawing a rubber-banded line from an element in the task model to a state object representing the other model construct, to associate the two. This can be seen in Figure 3, which shows a link being created between the task model `Connect` action task and the domain model `connect` operation defined on the `con: Connect` state object. The Link Wizard guides the designer through this process. Once an operation has been associated with a task, the icon in the task model is updated to reflect this change. Teallach currently uses a simple hyperlink metaphor so show associations between linked model components; this allows the designer to jump to an associated component by invoking its *show linked components* operation from a popup menu.

To derive information from one model into another, the designer simply drags a component from one model and drops it at the desired location in the target model. For example, the designer may construct a partial task hierarchy corresponding to some constructed presentation (or vice versa). Once the new structure has been generated, the relationships between components are maintained through the services provided by the Teallach store.

5. Conclusions

This paper has provided an overview of the Teallach MB-IDE for object databases. Distinctive features of Teallach include:

- A rich domain model that provides an abstract view of structural and behavioural features of applications in terms of the ODMG object model.
- A hierarchical task modelling language that is fully integrated with the domain model so that not only can

user tasks be described, but also the data associated with those tasks.

- A presentation model in which abstract descriptions of displays can be associated with task and domain model concepts, and which is associated with an extensible concrete presentation based on Java Beans and Swing.
- A flexible methodology, in which the models provide abstract facilities for describing the user interface without imposing a prescriptive approach to model construction.
- A design environment that supports the flexible methodology, and in particular the association of components from different models.
- An open architecture, in which both application classes and auxiliary functionality can be made accessible to other Teallach models through the domain model, and in which additional presentation components can be registered using the Java Beans protocol.

The design of the Teallach system has from the start been motivated by the desire to make the development of user interfaces to object database systems more systematic and more efficient. However, although we were willing to trade some measure of generality in the Teallach system for greater productivity in the database context, we believe that there is very little that is database specific about Teallach. While the domain model is clearly a data model from a database, the ODMG model has much in common with object models from other settings. What the database orientation has provided is a requirement to take the exchange of information between models seriously, and the early identification of a library case study has acted as a useful "reality check" throughout.

A further feature of Teallach is that it is decidedly non-radical in a number of respects. In particular, the domain model and the (concrete) presentation model are, respectively, an existing data model and an existing widget set, and the basic building blocks of the task model are familiar from other MB-IDEs. We consider this conservatism to be a virtue, as it allows reuse of software systems and experience, and has allowed the Teallach development effort to focus on what is required to integrate these existing components effectively. As a result, where Teallach makes a contribution in the area of MB-IDEs, it is not in its individual components, but rather in the way these components have been combined. Teallach is unusual both in the extent to which the different models are integrated (e.g. few other systems address issues relating to the transmission of domain model concepts through the task model), and in allowing associations to be constructed between models in such a flexible manner. Thus Teallach can be seen as contributing to model-based interface development research by: clarifying how models can be integrated; demonstrating how the use of model-based systems need not lead to the imposition of a specific development method; and by showing how tools can support flexibility in model construction.

Acknowledgements: This research is funded by the UK Engineering and Physical Sciences Research Council (EPSRC), whose support we are pleased to acknowledge.

References

[1] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, B. Sacre, J. Vanderdonckt, "Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide", *Interactive Systems: Design, Specification and Verification*. Springer, 1995, pp. 77-94.

[2] Borland Delphi. http://www.borland.com/delphi/

[3] W. Buxton, M.R. Lamb, D. Sherman, and K.E. Smith, "Towards a comprehensive user interface management system", *Computer Graphics, 17(3)*, 1983,pp. 35-42.

[4] R.G.G. Cattell et al, *The Object Database Standard: 2.0*, Morgan Kaufmann Publishers Inc., 1997

[5] A.B. Chaudhri and M. Loomis, *Object Databases in Practice*, Prentice-Hall, 1998.

[6] J. Foley, W.C. Kim, S. Kovacevic, and K. Murray, "Defining Interfaces at a High Level of Abstraction", *IEEE Computer* , 1989, pp. 25-32.

[7] T. Elwert, T. Schlungbaum, "Modelling and Generation of Graphical User Interfaces in the TADEUS Approach", *Designing, Specification, and Verification of Interactive Systems* (Palanque , P., Bastide, R. Eds.). Wien, Springer, 1995, pp. 193-208.

[8] T. Griffiths et al., "Exploiting Model-Based Techniques for User Interfaces to Databases", *Proc. Visual Database Systems 4*, (Y. Ioannidis and W. Klas (eds)), Chapman & Hall, 1998, 21-46.

[9] P. Johnson, H. Johnson, and S. Wilson, "Rapid Prototyping of User Interfaces Driven by Task Models", *Scenario-Based Design*, (Carroll, J. ed). John Wiley & Son, 1995, pp. 209-246.

[10] P. Markopoulos, J. Pycock, S. Wilson, and P. Johnson, "Adept - A task based design environment", *Proceedings of the 25th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, 1992, pp. 587-596.

[11] K. Mitchell, J. Kennedy, P. Barclay, "A Framework for User Interfaces to Databases". *Proc. AVI*, ACM Press, 1996.

[12] D.R. Olsen, "MIKE: The Menu Interaction Kontol Environment", *ACM Transactions on Graphics, 5(4)*, 1987, pp. 318-344.

[13] Poet Software. http://www.poet.com

[14] A. Puerta, H. Eriksson, J. Gennari, and M. Musen, "Model-Based Automated Generation of User Interfaces", *Proc. National Conference on Artificial Intelligence (AAAI)*, 1994.

[15] E. Schlungbaum, T. Elwert, "Automatic User Interface Generation from Declarative Models" *Proc. CADUI*, 1996, pp. 3-18.

[16] Visual dBase http://www.dbase2000.com/

[17] P. Szekely, P. Luo, and R. Neches, "Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design", *Proc. CHI 92*, 1992, pp. 507-515.

[18] P. Szekely, P. Sukaviriya, P. Castells, J. Muhtkumarasamy, E. Salcher, "Declarative Interface Models For User Interface Construction Tools: The MASTERMIND Approach Engineering For Human-Computer Interaction", *2nd Workshop on Database Issues for Data Visualization* (A. Wierse, G.G. Grinstein, U. Lang, (eds.)), Springer-Verlag, 1996.

[19] P. Szekely, "Retrospective and Challenges for Model-Based Interface Development", *Proc. DSVIS*, (F. Bodart and J. Vanderdonckt (eds)), Springer-Verlag, 1996, pp. 1-27.

[20] M.M. Zloof, "Selected Ingredients in End-User Programming", *Proc. Visual Database Systems (VDB) 4*, (Y. Ioannidis and W. Klas (eds)), Chapman & Hall, 1998.