The Teallach Tool:
Using Models for Flexible User Interface Design

Peter J. Barclay[2], Tony Griffiths[1], Jo McKirdy[3], Norman W. Paton[1], Richard Cooper[3], Jessie Kennedy[2]
[1]*Department of Computer Science, University of Manchester,*
Oxford Road, Manchester M13 9PL, UK.
Email: { griffitt, norm } @cs.man.ac.uk

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

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

http://www.dcs.gla.ac.uk/research/teallach/

Abstract Model-based user interface development environments aim to provide designers with a more systematic approach to user interface development using a particular design method. This method is realised through tools which support the construction and linkage of the supported models. This paper presents the tools which support the construction of the Teallach models in the context of the Teallach design method. Distinctive features of the Teallach tool include comprehensive facilities for relating the different models, and the provision of a flexible design method in which models can be constructed and related by designers in different orders and in different ways.

1. INTRODUCTION

The development and maintenance of user interface software is challenging. Although interface development environments provide facilities that allow individual components within an interface to be constructed without recourse to programming, the behaviour of user interfaces is

generally implemented by complex, hand crafted software systems. Although design patterns can be used to provide an organisational framework for interface software, it is still the case that user interface software is intrinsically complex, and that changing an existing interface to reflect changing requirements and to take account of user feedback is a laborious and often somewhat ad-hoc process.

Model-based user interface development environments (MB-UIDEs) have been developed with a view to providing a more systematic approach to user interface development, building in particular on abstract models of different aspects of user interface functionality (e.g., TADEUS [11], FUSE [7] and MOBI-D [10]). Typically, a MB-UIDE will include domain, task, dialogue and presentation models. The benefits that it is hoped will arise from the use of MB-UIDEs include the generation of interface software based on the abstract models, and more seamless integration of the interface design and implementation processes.

However, although MB-UIDEs have a range of promising characteristics for easing user interface development, they introduce a number of new challenges. The development of effective tools for the construction and linking of a collection of abstract models is itself a substantial challenge, which must be addressed in the context of a design method that directs the interface developer in the construction of a coherent collection of models. This paper seeks to address these two issues – tools for model construction and tool support for a design method – in the context of the Teallach MB-UIDE [6]. Distinctive features of the Teallach tool include comprehensive facilities for relating the different models, and the provision of a flexible design method in which models can be constructed and related by designers in different orders and in different ways.

The paper is structured as follows. Section 2 sets the scene by introducing the Teallach system. Section 3 describes a case study that will be used throughout the paper. Section 4 outlines the flexible design method supported by Teallach, and which must be accounted for in the tool. Section 5 describes the facilities provided by the tool for editing and relating the Teallach models. Section 6 presents some conclusions.

2. TEALLACH BACKGROUND AND MOTIVATION

The Teallach MB-UIDE is primarily concerned with constructing user-interfaces to object oriented databases. The Teallach user is the designer of interfaces to database applications, not the end-user of these interfaces. In order to meet the needs of such design efforts, Teallach provides three

models: a domain model, a task model, and a presentation model. These models are described in detail in [6].

The presence of underlying models gives some advantages to an interface-building tool, such as a clear semantics for the interface under construction, and facilities for the automatic checking of consistency in the models (and hence in the resulting interfaces), together with support for 'help' and 'undo' functionality. However, we have identified a number of weaknesses in existing MB-UIDEs. Some of these are described below; for more details, the reader is referred to [4].

- Some systems impose a rigid methodology on the interface-designer.
- Most systems do not provide the facilities to work with database-specific concepts, such as transactions.
- Many systems have a fixed set of widgets from which interfaces can be constructed, thereby disallowing the use of application specific widgets which may be required in some domains.
- Few systems have a clear method for representing flow of state information within the interface to be generated.

One of the goals of the Teallach project is to develop models that address some of these shortfalls, and to build a prototype tool that illustrates our solutions. In particular, we do not wish to impose a particular style of working on the interface designer. For example, one designer may wish to proceed from specifications to implementations, whereas another may wish first to sketch forms to be used in the interface, and then connect these to application functionality. Both these approaches, and others, are allowed by Teallach. This is achieved by 1) treating all of the models in an even-handed manner, and 2) performing consistency checking as late as possible, so that the designer is free to work through 'inconsistent' designs towards consistent ones. In particular, automatic generation of model components may be used as the designer desires: it is possible to generate large parts of the interface automatically (and optionally modify these generated components), or to 'wire together' user-built substructures with no use of automatic generation.

Teallach's interfaces are realised as compiled Java applications, giving the efficiency of compiled code while running on any major platform. The widgets used are taken from Java's Swing widget set [2], but additional user-supplied widgets may be added to the toolkit and used at any time. Access to database-specific concepts is supplied through Teallach's ODMG-style [1] domain model.

3. CASE STUDY

To provide a tangible explanation of the manner in which the Teallach tools operate and are used, subsequent tool discussion will be conducted in relation to a case study, which is a library database application – the UML class diagram of this is shown in figure 1.

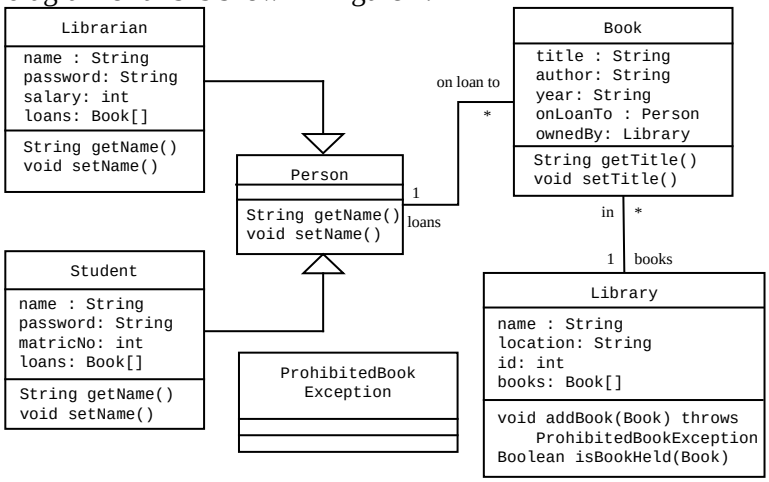

Figure 1: The Library Database Schema

For brevity, consider a single task that the user of a library application might perform – that is *searching for a book*. Using the application, the user indicates that a search is to be performed, and subsequently specifies a collection of search parameters which constitute the attributes on which the search is to be based – for example, a search based on a named author. The user then initiates the search (amounting to the running of a query parameterised by the specified information) and is presented with the resultant information. Assuming one or more books were returned, the user can browse through them. The remainder of this paper demonstrates, in terms of one possible traversal of the Teallach method using the tools, how a designer might construct a user interface to support this task.

4. THE TEALLACH METHOD

One of the principle aims of Teallach is the provision of a flexible design method such that designers using Teallach are not restricted to a single developmental strategy. Although some might question this approach given the more rigid methods promoted by other MB-UIDEs [4], the methodological stance adopted by Teallach arose from the following observations. Firstly, there is little evidence to support the notion that the

less flexible methods forwarded by other MB-UIDEs are indeed the best (or only) approaches; forcing designers along a linear developmental path appears overly restrictive and does not support the characteristics of the software development life cycles that designers often favour. For example, one could indicate that TADEUS [11] operates by successively refining a task model through a dialogue model and a number of interaction tables. Secondly, Teallach recognises that if it is to be adopted as a means of user interface development, then it needs to observe the developmental habits of software developers who often work in iterative cycles of development where various aspects of their artefacts are developed in parallel or in an interleaved manner [8]. Through its flexible methodology, Teallach aims to support (as far as possible) the observed working practice of software developers.

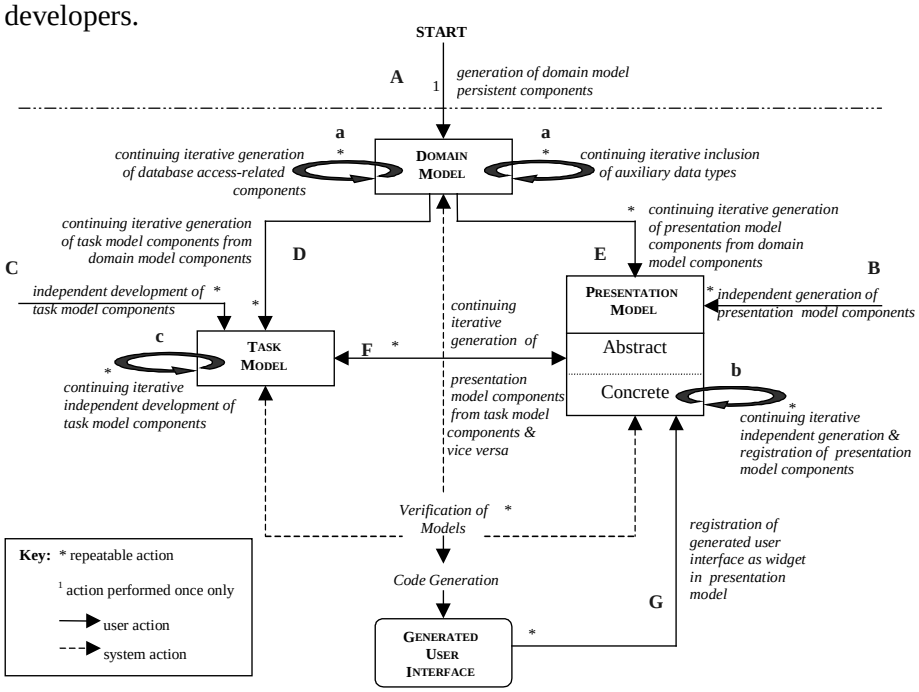

Figure 2: The Teallach Method

Figure 2 shows the flexible methodological structure proposed by Teallach and which is subsequently realised in the Teallach tools. Due to the number of potential routes through the method – made possible by its inherent flexibility – it is not feasible to discuss each. Instead, the interactivity and dependencies between the steps in the method will be discussed at a high level, with the intention that the discussion provides a feel for the developmental freedom available to the designer. Later

discussion of the actual Teallach tools, by which the method is realised, will demonstrate one possible path through the method.

Teallach refers to a design effort as a *project* – that is, a collection of models which contribute to the development of a specific user interface. Projects can be saved during the course of development, and components of one project can be imported into another to facilitate reuse. The remainder of this discussion will assume that the developer is creating a new project. It should be noted that the aim of this section is to outline the Teallach method – the *means* by which the actions may be performed using the tools is discussed in section 5.

Teallach has been developed to facilitate the design of user interfaces to pre-existing object oriented databases (OODBs). There is therefore a basic assumption that the schema and classes for the underlying database must exist prior to user interface development. The Teallach tool therefore permits one entry point to the developmental cycle (as shown in Figure 2). This allows the structure of the underlying database application to be established within a project in the form of the *persistent* components of a project-specific domain model (step A). Having determined the persistence capability of the application, the developer is then free to design each of the individual Teallach models.

At *any* stage in the design of a user interface, each of the models can be independently developed. Consider first the domain model. As required, the developer can create components to facilitate access to the underlying database (database connectivity components) and can import information about auxiliary data types which may be required for the runtime operation of the application – see the steps labelled (a) in figure 2. It should be noted that the domain model also provides the facility to view and utilise the types provided by the Java API – a subset of the auxiliary information available to Teallach. The inclusion of these components is a one-off action which can be performed at any stage during development.

Consider now the task model. Independently of the other Teallach models, components (and hierarchies of components) within the task model can be created, manipulated, and deleted – as shown by steps (C) and (c) in figure 2. Similarly, the developer can, independently of the domain and task models, create, manipulate and delete components (and hierarchies of components) within the presentation model and can register new widgets (steps (B) and (b) in figure 2). Each of these activities can be performed at any stage during user interface development.

At any point during development the designer can either associate components from distinct models (thus linking the models to generate a cohesive user interface design) or can generate new components in a model from a component previously constructed in another model. Such activities

are represented by steps (D), (E) and (F) in figure 2 (and also labelled in the actual tool shown in figure 3). These activities can be performed repeatedly and in any order.

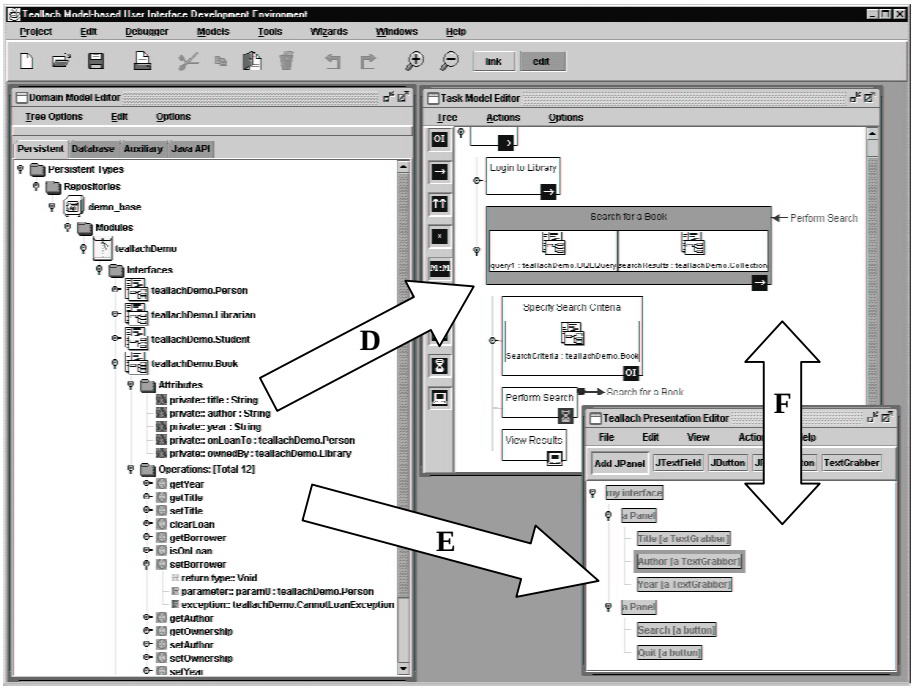

Figure 3: The Teallach Tool and Possible Inter-model Operations.

Consider first step (D), which shows the association of task model components with domain model components. The designer can use a domain model structure to automatically generate an initial task hierarchy, or can link components in the two models through the use of state objects. A state object is the means by which the task model represents constructs imported from the other Teallach models. Greater detail of these associations is given in section 5. Step (E) supports a similar scenario where the domain model components are used to automatically generate a presentation. Further details of these associations are also given in Section 5.

In Teallach, the task model is not tied to a specific visual representation of a user interface. This maxim is realised in the tool by linking task model components to high-level abstractions of concrete presentation model constructs (termed the *abstract* presentation model). In step (F) components in the abstract presentation and task models can be linked together, or can be used to automatically generate a new component in the other model. In all such operations, the designer will create a state object in the task model to represent the linked presentation model component.

At any stage during design, the developer can choose to verify the various models; a step which determines the consistency and completeness of the models with respect to one another. Assuming model verification has succeeded, the designer can automatically generate a user interface as described by the models. 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. The designer can choose to register any generated user interface as a self-contained "black-box" widget within the presentation model (see step (G)), thus facilitating a bootstrapping process of development and reuse of generated components.

From the above discussion it can be seen that, with the exception of step (A), provided that the required components exist within each model, any of the discussed steps can be performed in any order and any number of times. Hence designers are given the freedom to work in the manner most suited to themselves and their projects, and are not restricted by an overly prescriptive developmental strategy.

5. THE TEALLACH TOOL

5.1 General

The Teallach tool has been implemented using Java 2.0 and the Swing GUI tool-set. This tool-set has provided us with a rich library of GUI primitives that facilitate design using the model-view-controller pattern; our experiences with both this tool-set and Java in general have been mainly favourable. It has been designed so that Teallach itself, and the interfaces it generates which are also implemented in Java, will run on all major hardware and OS platforms. Teallach interacts with the underlying application (typically an OODB) through its domain model, which provides an interpretation of the contents of the application through the concepts of ODMG. In the current prototype Teallach has been designed to interact with the Poet OODBMS [9].

As shown in figure 3, the tool provides separate editors for each of the three models, implemented using a desktop metaphor. In addition, the presentation model provides further, free-floating windows, such as a preview of the interface under construction, and a palette of widgets the designer can use. Model constructs can be exchanged between the editors either by drag-and-drop or by cut-and-paste metaphors using a single system clipboard.

The semantics of inter-model associations are described in more detail below, but the basic scheme is as follows: when a fragment of one model is

dropped into the editor of a different model, some new structure is generated in the target editor, derived from the source model (drag-and-generate). It is also possible to 'link' components from different models, for example to show that a particular widget is to be used to perform a particular task. This is achieved by switching the tool into *link* mode and drawing arcs between the associated components (click-and-link). The large arrows in figure 3 show the possible ways in which the three Teallach models can interact.

5.2 The Domain Model Editor

A project-specific Teallach domain model reflects the structure and functionality of the underlying application, database connectivity, and auxiliary data types such that they can inform and link into the user interface. To provide a measure of platform independence, the domain model represents these factors using constructs derived from the concepts specified in the ODMG object database standard.

The domain model editor within the Teallach tool comprises four independent panels, representing: persistent data components, imported auxiliary classes, auxiliary classes derived from the Java API, and the database connectivity aspects of an application. A domain model serves purely as an information source (as shown in figure 2), and as such the domain model tool is not concerned with receipt of information from the other models. Instead, its role is to make available information to the other models in a uniform manner such that the persistence of the data is transparent.

Upon start up of the Teallach development environment, as mentioned in Section 4, a model of the persistent data related components of the application is generated. This is done automatically through an analysis of the schema of the underlying database. The domain model editor shown in figure 3 shows the persistent data components of the domain model that represent the schema described in the case study.

There are also two panels concerned with the representation of auxiliary data types, that is, data structures which are not database classes, but which provide functionality required for the runtime operation of the application. The first of the two auxiliary data panels provides the designer with the means to import, as required, any user-defined classes or packages that provide additional functionality. To import a package or class, the designer must simply specify its fully qualified name. The screen shot in figure 4 shows an auxiliary class that has been imported for use in the case study. It provides the facility for authentication of a string as a valid year. Once again, the designer is able to copy or drag this domain model component and paste or drop it into one of the other models so that its functionality can be

exploited. The second of these panels represents the data types in the Java API.

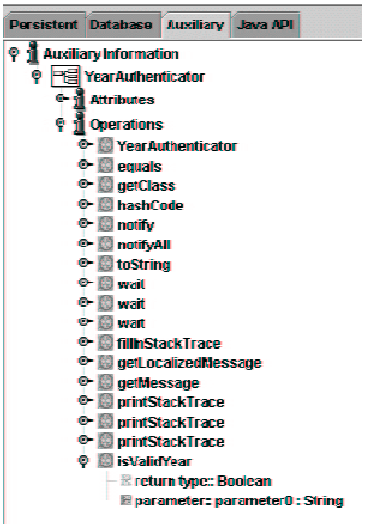

Figure 4: Importing Auxiliary Information

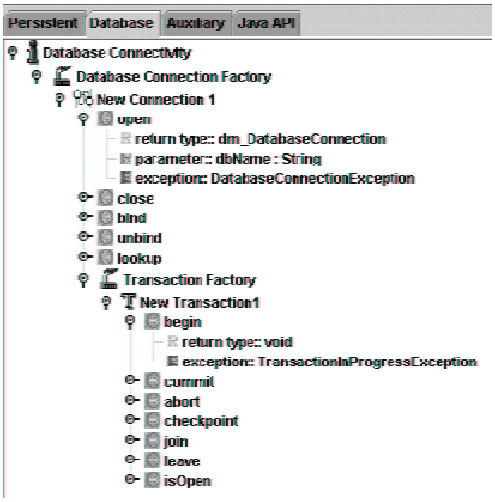

Figure 5: Database Connectivity Pane

The final panel within the domain model editor concerns the establishment of components to support database connectivity. Once again, such components are modelled in terms of the appropriate ODMG concepts. Within this panel, the developer can instantiate database connections and therein transactions. Similarly, the developer can create OQL queries that can be run over the underlying database. Once established, the developer can treat database connectivity components in the same manner as other domain model components. Figure 5 shows the representation of a database connection and transaction required for the *search for book* case study. Having established the connectivity with the database, the developer would then be required to build the query using this same panel in the domain model editor.

5.3 The Presentation Model Editor

Teallach provides both a *concrete* presentation model (CPM) and an *abstract* presentation model (APM). The CPM contains real widgets such as those available in Swing, and user-supplied custom widgets. For example, the widget JPasswordField (for capturing users' passwords) is available from Swing, whereas the widget TextGrabber (for inputting text) is a user-supplied widget. Arbitrarily complex Interactors, such as 3D molecule-viewers, may be used as concrete widgets provided they have been registered with the presentation model tool.

Teallach's APM extends the light-weight presentation model described in [3]. This model defines abstract categories of widgets, designed to offer a particular functionality. An abstract category may be realised by many different concrete widgets. For example, the category *Inputter* represents anything which may capture the user's input; both a JPasswordField and a TextGrabber may serve as realisations of Inputter.

The designer may use either concrete or abstract presentation objects, and intermix these freely. Of course, where abstract interactors are used, a decision must be made as to which possible realisation will be used in the final interface; a default is always available, so a valid interface is defined at all times. Details of how abstract categories are realised by concrete widgets are recorded in a *style*, so that consistency of look-and-feel can be achieved, and differing interfaces can be easily generated which support the same functionality.

5.3.1 An Overview of the Presentation Model Tools

The interface designer interacts with Teallach's presentation model through a collection of related tools. The presentation model editor allows the designer to construct presentation fragments, whether by hand or automatically; the widget palette provides access to the components which may be used for building interfaces; and the presentation meta-editor allows the designer to edit the meta-model of the presentation, as described below.

5.3.2 Constructing Interface Fragments by Hand

The designer may construct presentations by hand, by explicitly assembling components. An example can be seen in figure 6**,** where the designer is constructing the form to be used when specifying the criteria to be used when searching for a book. From the presentation model's widget palette, the designer has selected three *TextGrabber* components, to capture the title, author, and year fields for the book sought. These components have been placed within a JPanel, which is a (usually invisible) container provided by Swing for grouping together related items, and ensuring that they behave as desired when the window is resized. An ancillary class, from the domain model, may be employed to ensure that the format of the text entered into the year field can represent a valid year.

The designer has then added a second JPanel to the search window, grouping two buttons that allow the user either to confirm the search criteria and proceed to performing the search (*search*), or to quit from this window, if desired (*quit*).

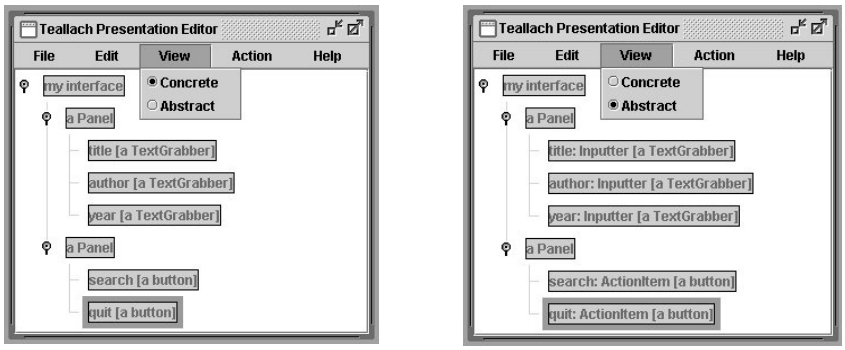

Figure 6: Constructing Presentation Model Elements

Once constructed, this interface fragment can be used in a variety of ways: 1) it can be 'shrink-wrapped' for later use in this and other applications; 2) it can be linked to constructs in the domain and task models to form part of the final user-interface; or 3) it can be dropped into the task model to automatically generate task structures corresponding to the activity of searching for a book.

In addition to the tree-structured view provided by the presentation editor, Teallach also provides a preview of the end-interface under construction. For example, an automatically generated preview of the search form is shown in figure 7; this allows the designer to see an immediate result when altering properties of the presentation such as colour, font, and layout.

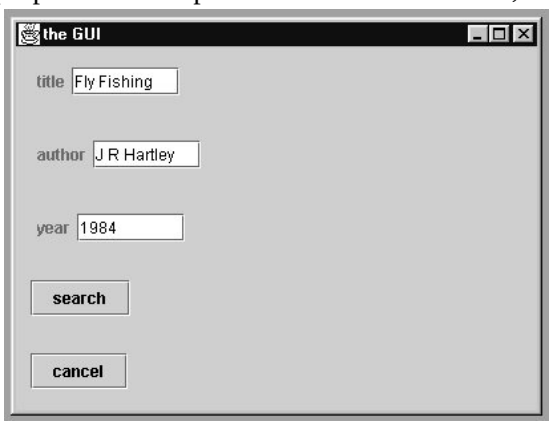

Figure 7: The Presentation Model Preview Window

5.3.3 Editing the Presentation Meta-model

Teallach's presentation meta-editor allows the designer to modify the abstract presentation model. The meta-editor is shown in figure 8**.** This tool has both a *categories* tab and a *register* tab. The categories tab allows new abstract categories to be defined and added to the presentation model; once

defined, they become available for use in the abstract widget palette. Figure 8 shows part of the definition of the category Inputter, derived from the category Item; this has a method getValue() returning an object (of any type) which has been input by the user of the widget.

Figure 8: the Presentation Model Meta-Editor

The register tab allows new concrete widgets to be registered within existing categories. Once registered, a widget becomes available for use in the concrete widget palette. This allows the designer to use custom-built and third-party widgets in an interface.

The meta-editor uses the meta-data it collects to automatically write Java code implementing the newly defined meta-objects. This code is then compiled reflectively, so that the new objects can become available in the system without need for interpretation.

5.4 The Task Model Editor

The Teallach task model tool provides an environment for constructing and editing hierarchical task models, the semantics of which have been presented in [5]. A hierarchy constructed using the task model tool is a temporally ordered representation of the goals and subgoals a user wishes to achieve in the developed interface. The Teallach task model is novel in that it provides a designer with the ability to declare local state and associate this with a task, and subsequently to indicate how this state information is initialised and utilised within a task.

To realise the task model for the case study, our designer constructs a high-level specification of the task they intend application end-users to perform through the modelled interface. To achieve this, the designer drags a task of the required type from the task model's palette of task types (shown in figure 3) and drops it at the required location in the task model construction area.

At the lowest level in the task hierarchy, the designer creates *interaction* and *action* tasks which represent how the application processes information, and how the end-user and the application participate in the task. These tasks may have links to domain or presentation model functionality which is realised and invoked through a suitably initialised state object (as described in section 5.5.1).

5.5 Establishing Model Interaction Using Link Mode

At any time during the model development process, the designer may create links between components specified in any of the Teallach models. By creating a link, the designer is stating, for example, that a widget is to be used to perform a particular task, or that an action task corresponds to an invocation of an operation on an application class. Links between the Task and Presentation models are also used to describe dialogue dynamics. This section will show how, through the use of state objects representing both domain and presentation model constructs, the task model tool provides the facility to bind together the concepts in the three Teallach models.

5.5.1 Creating and Using State Information in the Tool

A state object is the means by which the task model tool maintains references to constructs from the other Teallach model tools, and is constructed through either a *paste as state* menu option, or as a side-effect of Teallach automatically generating a task construct from another Teallach model construct. A state object refers to a named instance of either a domain or abstract presentation model class, and is realised as a uniquely named rectangle within the scope of a non-primitive task.

Once a state object has been created it can be utilised in several ways. For example, one of the state object's methods can be invoked (equivalent to invoking underlying application or widget functionality), or one (or more) of a state object's public attributes can be read from or written to (equivalent to specifying the flow of information between the user and the underlying application).

For the purposes of our case study, the designer needs to specify that the search criteria provided by the end-user should be stored in a named object of type `Book`, and that a named query should be invoked on the database with this `Book` object as the search parameter. The designer therefore copies the `Book` persistent domain class from the domain model tool and pastes in into the *specify book information* non-primitive task selected in the task model tool using the *paste as state* option from the Edit menu. Similarly, using the Database Connectivity editor pane, a state object corresponding to

a new OQL query is copied from the domain model to the *Search for a Book* task. The designer will also need to create state objects representing the database session and transaction in which the OQL query will be performed. Once the designer has provided a suitable name (e.g., `currentBook`), new state objects are created at the required locations. In figure 3, the task model editor shows that the Search for a Book task contains two state objects corresponding to `searchResults:Collection` and `query1:OQLQuery`.

The following sections illustrate *some* of the ways in which state objects can be used to link constructs in the three Teallach models using both the link and generate mechanisms.

5.5.2 Linking Task and Domain Model Constructs

Once our designer has created the necessary state objects, they can link action or interaction tasks with them. For example, the designer needs to show that the *Perform Search* action task invokes the `execute()` method on the `query1:OQLQuery` state object. This is achieved by the designer selecting the *link* toggle button on the main toolbar to switch to link mode, and subsequently clicking on the *Perform Search* action task and `query1:OQLQuery` state object – an extending arc is drawn between the two constructs to give the designer visual feedback.

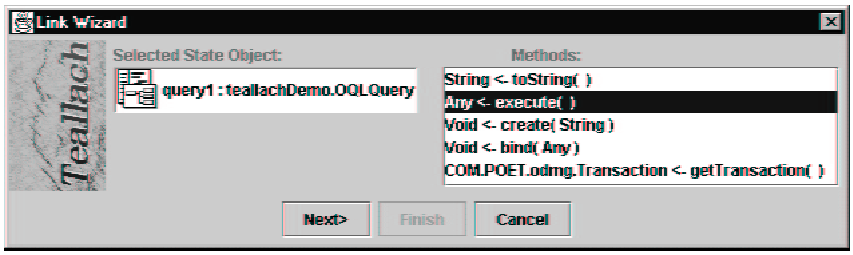

Figure 9: Assigning Method calls to an Action Task Using the Link Wizard

If the link operation is successful, then Teallach invokes its *Link Wizard* to guide the designer through the potentially complex task of creating the link. As shown in figure 9, the Link Wizard recognises that the designer is creating a link between an action task and a state object, and asks the designer to choose which of the selected state object's methods they wish to invoke by providing them with a list of possible methods from which to choose. Once a method has been selected, the Link Wizard checks if the selected method requires any parameters, or if it has a return value. In either case it asks the designer which state objects will provide the information for the parameters, and optionally, which state object will be used to store the return value. For both of these questions the Link Wizard will provide a list

of suitable alternatives to the designer. An example of this is shown in figure 10, where the Link Wizard is enquiring where the collection of Objects (i.e., Books) returned by the `query1.execute()` method will be stored; the designer selects the `foundBooks:Collection` state object within the *Search for a Book* task.

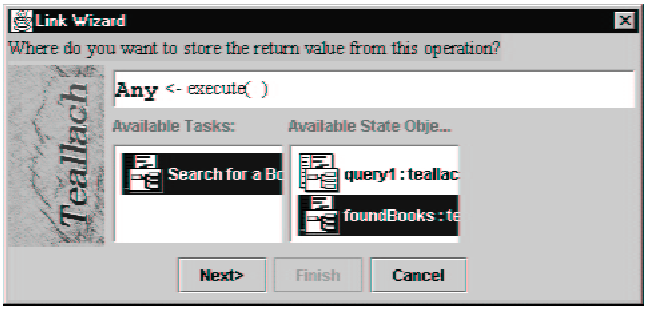

Figure 10: Handling return Values Using the Link Wizard

If the chosen domain method raises an exception (i.e., an IllegalOperationException), then the task model editor will display a red circle next to the action task for each exception that it raises. The designer is then free to specify what should happen if the exceptional circumstance arises; the task model editor can be used to specify that if the `query1.execute()` method (as utilised by the *Perform Search* action task) raises an exception, then the *Search for a Book* task should be performed again – this is shown in figure 3.

5.5.3 Linking Task and Presentation Model Constructs

If the designer wishes to declare that an interaction task is to be realised by a particular widget (e.g., that the *Get Author Name* interaction task corresponds to a particular Swing JTextField widget in the CPM), then in a similar manner to the previous section, the designer will create a state object corresponding to the JTextField widget in the required location within the task model editor. It should be noted that it is actually the APM construct which corresponds to the CPM widget that is used. If the link operation is accepted, then the Link Wizard will once again be activated.

Since the semantics of this link operation are different to that discussed in the previous section, the Link Wizard will ask the designer a different set of questions. For example, if the designer creates a link between the *Get Author Name* interaction task and the `author:Inputter` state object (realised by a JTextField widget in the CPM), then the Link Wizard will ask if the task is receiving or outputting (or both) information, and will proceed with a dialogue which will ascertain the type of the information being processed, and which state object(s) will provide this information.

By linking task and presentation model components the designer is also specifying the dynamics (dialog) of the interface. This is achieved by the semantics of non-primitive task model nodes (i.e., sequential, concurrent, etc.) being applied to the non-primitive presentation model nodes to which they are linked.

5.6 Establishing Model Interaction Using Generate Mode

To assist the designer in the process of constructing a consistent set of models, Teallach provides a drag-and-generate mode. This mode is invoked by dragging a fragment of one model into a suitable location within another. As a result of this operation a new model structure is created in the target model. Since the domain model is immutable, it cannot act as a target model. When the target model is the task model, Teallach creates appropriate state objects in addition to the newly constructed task hierarchy (i.e., domain classes or presentation widgets), and automatically creates links between these state objects and any action or interaction tasks.

For example, our designer may decide to drag the `Book` class into the task model editor to create a new first child of the *Search for a Book* task: this will create an order-independent task called *Edit Book*, with an action task child corresponding to each of the class's methods, and interaction tasks corresponding to each of the class's public attributes. The designer is then at liberty to edit the new constructs required. In this case the designer will simply remove any unwanted action or interaction tasks, and will rename the top-level task *Gather Search Criteria*.

Once this task has been constructed, the designer can then drag the new task construct into the presentation model to create a default form to represent the required task.

6. SUMMARY

This paper has presented the flexible design method forwarded by the Teallach MB-UIDE, which is realised through a rich set of tools that support the construction of the Teallach models. In particular the Teallach method and its supporting tools remove the rigid methodological constraints imposed by other MB-UIDEs, providing user interface designers with a method and design environment that more closely meets their modes of working.

This inherent flexibility has posed many challenges for the Teallach design team, as a flexible design method requires often complex control facilities. We have therefore concentrated on providing a core set of design

primitives (e.g., building models individually using no automatic generation, providing a simple *paste as state* operation for inter-model linking), and subsequently providing higher-level design functionality (utilising the core primitives) which supports a more rapid design method (e.g., automatically generating model constructs from the information modelled in another Teallach model). The often complex process of inter-model linkage has also been greatly simplified through the use of a Wizard metaphor.

Acknowledgements

This work is funded by UK's Engineering and Physical Sciences Research Council (EPSRC), whose support we are 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 Carole Goble, Phil Gray, Michael Smyth and Adrian West.

7. REFERENCES

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

[2] Eckstein, R., Loy, M., & Wood, D., *Java Swing*. O'Reilly & Associates, Sebastopol, CA. 1998.

[3] Gray, P., Cooper, R., Kennedy, J., McKirdy, J., Barclay, P., & Griffiths, T. (1998), *A Lightweight Presentation Model for Database User Interfaces*, ERCIM'98, Stockholm, October 1998.

[4] 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 Databases*, in Proceedings of VDB-4, Chapman & Hall, London. pp. 21-46. 1998.

[5] Tony Griffiths, Norman W. Paton, Carole Goble, Adrian West, *Task Modelling for Database Interface Development*. To appear in Proceedings HCI International'99.

[6] Tony Griffiths, Peter J. Barclay, Jo McKirdy, Norman W. Paton, Philip D. Gray, Jessie Kennedy, Richard Cooper, Carole A. Goble, Adrian West and Michael Smyth, (1999), *Teallach: A Model-Based User Interface Development Environment for Object Databases*, in Proc. User Interfaces to Data Intensive Systems (UIDIS), IEEE Press. pp. 86-96. 1999.

[7] F. Lonczewski and S. Schreiber, *The FUSE-System: an Integrated User Interface Design Environment*, in Proc. CADUI, J. Vanderdonckt (Ed.), pp. 37-56. 1996.

[8] McKirdy, J., *An Empirical Study of the Relationships Between User Interface Development Tools & User Interface Software Development*, Technical Report TR-1998-06, University of Glasgow, Department of Computing Science, March 1998.

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

[10] Puerta, A.R., *A Model-Based Interface Development Environment*. IEEE Software, 14(4). pp. 41-47. 1997

[11] E. Schlungbaum and T. Elwert, *Automated User Interface Generation from Declarative Models*, in Proc. CADUI, J. Vanderdonckt (Ed.). pp. 3-18, 1996.