Teallach — a flexible user-interface development environment for object database applications

P.J Barclay\textsuperscript{a,\*}, T. Griffiths\textsuperscript{b}, J. McKirdy\textsuperscript{c}, J. Kennedy\textsuperscript{a}, R. Cooper\textsuperscript{c}, N.W. Paton\textsuperscript{b}, P. Gray\textsuperscript{c}

\textsuperscript{a}Department of Computing, Napier University, Merchiston Campus, 10 Colinton Road, Edinburgh, EH10 5DT, UK
\textsuperscript{b}Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK
\textsuperscript{c}Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow G12 8QQ, UK

Received 2 February 2001; received in revised form 5 April 2002; accepted 30 May 2002

Abstract

The Teallach project has adapted model-based user-interface development techniques to the systematic creation of user-interfaces for object-oriented database applications. Model-based approaches aim to provide designers with a more principled approach to user-interface development using a variety of underlying models, and tools which manipulate these models. Here we present the results of the Teallach project, describing the tools developed and the flexible design method supported. Distinctive features of the Teallach system include provision of database-specific constructs, comprehensive facilities for relating the different models, and support for a flexible design method in which models can be constructed and related by designers in different orders and in different ways, to suit their particular design rationales. The system then creates the desired user-interface as an independent, fully functional Java application, with automatically generated help facilities.

\textcopyright{} 2003 Elsevier Science Ltd. All rights reserved.

Keywords: Model-based systems; User-interfaces; Data-intensive applications

1. Introduction

\textit{The importance of providing powerful,} appropriate and usable user-interfaces to software applications is well recognized in particular, the database community

\*Corresponding author. Bassoe Offshore Consultants, 10 The Enterprise Centre 1, Dryden Road, Loanhead Edinburgh EH20 9LZ, UK Tel.: +44-131-440-2848, fax: +44-131-440-4318.
E-mail address: peter@bassoe.com (P.J. Barclay).
URL: http://www.dcs.gla.ac.uk/research/teallach/.

1045-926X/03/$-see front matter \textcopyright{} 2003 Elsevier Science Ltd. All rights reserved.
PII: S1045-926X(02)00056-3
has recognized the need for more thoroughgoing research on the development of user-interfaces for database applications [2]. The Teallach project has addressed this area by making use of model-based user-interface development techniques in the object-oriented database arena.

In previous publications, we have described the utility of model-based techniques for database design [3], the approach taken by the Teallach project [4], the philosophy underlying Teallach’s presentation [5] and task [6] models, the purposes and interactions of the models on which Teallach is based [7] and Teallach’s flexible approach to design [8].

As the project is now completed, the purpose of this paper is to provide a summative review of the outcomes of our research, focusing on the functionality of the software developed and showing how this supports our novel approach to design. The paper is structured as follows. Section 2 provides a brief introduction to model-based user-interface development environments, and identifies the focus of the Teallach project; Section 3 introduces Teallach, highlighting some similarities to and differences from previous systems; Section 4 explains the design philosophy underlying Teallach; Section 5 outlines an example used in the remainder of the paper; Sections 6 and 7 describe the Teallach software in detail; Section 8 describes how interfaces are generated; Section 9 describes how the Teallach environment can be customized and extended; Section 10 describes how ‘help’ functionality is provided; the implementation of the system is sketched in Section 11; Section 12 mentions some future work and Section 13 concludes with a summary.

Additional information about the project may be found on the Teallach homepage at: http://www.dcs.gla.ac.uk/research/teallach/

2. Model-based user-interface development environments

Model-based user-interface development environments (MB-UIDEs) have been investigated with a view to provide a more systematic approach to user-interface construction, by relating the interface developed to the underlying abstract models. Some examples of this approach are TADEUS [9], FUSE [10], TRIDENT [11], Adept [12], DRIVE [13] and MOBI-D [14].

In this approach, the process of developing an interface becomes a process of constructing and linking instances of abstract models; the final interface may then be verified against the models or automatically generated from them. Furthermore, models or parts of models can be reused, facilitating subsequent development.

The models used reflect different aspects of user-interface functionality, and each proposed MB-UIDE has its own set of models. Typically, an MB-UIDE will include the following models:

- **Domain model**: This describes the underlying application for which the interface is being developed, and may also include auxiliary functionality required by this application.
• **Task model**: This describes the user’s activities in interacting with the intended software system.

• **Dialogue model**: This describes the flow of control and information between the user and the application, and may also include information-flow *within* the application.

• **Presentation model**: This describes the on-screen appearance of the user-interface.

However, some systems merge or omit some of these models, and other systems have additional models. In Teallach, for example, the task and dialogue models are combined.

Benefits, it is hoped, that will arise from the use of MB-UIDEs include the generation of interface software based on the models, reducing development efforts; a more seamless integration of the interface design and implementation processes, increasing the correctness of the resultant interfaces; using the models to facilitate communication amongst designers; and the automatic generation of additional functionality, such as ‘help’ and ‘undo’ facilities, supported by the presence of the underlying models.

Since the model-based approach transforms interface-building into model-building, model-based projects usually advocate a particular design method to guide the user through this model-building process. The software tools developed aid the developer by providing support to the design method advocated. Another perspective on this is that the nature of the design approach advocated tends to influence the structure and functionality of the software tools developed.

The methods used by existing model-based systems usually enforce a particular sequence of working on the developer. Whereas there are sound reasons for such an approach when programming in the large, user-interface development is normally a creative, iterative process, and we feel that highly prescribed design methods are likely to hinder the uptake of MB-UIDEs by user-interface designers. Therefore, a major theme of the Teallach project, and of this paper, is to support more flexible approaches to design, and develop software which can support such flexibility. This idea is developed further in the following section.

### 3. Teallach

A distinguishing feature of the Teallach MB-UIDE is that it is primarily concerned with constructing user-interfaces to object-oriented database applications. This means that, unlike in other MB-UIDEs, database-specific concepts such as transactions and queries must be incorporated within the models used.

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 (incorporating dialogue and runtime models), and a presentation model. These models are outlined below, and differences from the generic definitions of these models given in the...
previous section are explained. Teallach’s models are described in detail in [7], and examples of the models in use may be found in Sections 6 and 7.

Despite the potential advantages offered by the model-based approach, we have identified a number of weaknesses in existing MB-UlDEs. Some of these are listed below; for more details, the reader is referred to [3].

- Some systems impose a rigid design method 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 application domains.

One of the goals of the Teallach project has been to develop models and methods that address some of these shortfalls, and to build a prototype tool that illustrates our solutions. In order to support our approach, Teallach uses the models described below.

3.1. The domain model

This model describes the schema of the underlying database, the structure of non-persistent auxiliary functionality (such as utility classes in the Java API), and issues relating to database connectivity. All components are expressed in terms of the concepts outlined in the ODMG 2.0 standard object model [13] — an approach which carries two distinct advantages: (a) the Teallach system may interface with different underlying object-oriented database management systems (OODBMSs) in a uniform manner and (b) the persistence of the data modelled is orthogonal to its representation. The domain model contains types representing meta-level constructs of the ODMG (such as classes, properties and relationships); particular instantiations of these constructs describe the schema of a particular database. As the ODMG model is well documented in [13], the domain model is not described further here.

3.2. The task model

The task model describes the tasks performed by the end-user when interacting with the database application, and how the associated information is processed. Tasks range from high-level, human-oriented tasks to low-level, system-oriented tasks. The Teallach task model also incorporates functionality that other MB-UlDEs have placed in their dialogue models, and certain runtime information describing some dynamic aspects of the interface under construction, such as the propagation and handling of exceptions.

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. Sequential tasks are shown with an → icon on the rectangle that represents them in the task model editor.

• **Order-independent:** The subtasks may be performed in any order; all subtasks must be completed before the task’s goal is considered achieved. Order-independent tasks are shown with an OI icon.

• **Repeatable:** The subtasks are repeated a specified number of times, or until a condition is satisfied. Repeatable tasks are shown with an * icon.

• **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 subtask at a particular time, although all are progressing. Concurrent tasks are shown with a ↑↑ icon.

• **Choice:** The user must decide which one of the subtasks is to be performed; the chosen subtask must be completed before the task’s goal is considered achieved. Choice tasks are shown with a 1:M icon.

• **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. Optional task are shown with an M:M icon.

• **Conditional:** There exists a choice between subtasks that is dependent on a specified condition. Conditional tasks are shown with a || icon.

• The two primitive task types are:
  - **Action tasks:** An action task indicates that the system will perform some internal processing (such as a calculation or a database update). Action tasks are indicated by a small hourglass icon.
  - **Interaction tasks:** An interaction task indicates that there will be an exchange of data between the system and the user (i.e. an input or output operation). Interaction tasks are indicated by a small VDU icon.

Examples of the various task types can be seen later in [Fig. 10](#).

3.3. **The presentation model**

This model describes the appearance of the interface under construction; it is shown both in a schematic, hierarchical format and as a WYSIWYG ‘preview’ of the end-interface. A distinguishing feature of Teallach is that its presentation model is a dual-level model, which provides abstract interactors as first-class objects as well as concrete widgets.

The abstract presentation model consists of a set of categories, based on those identified in [5]. The categories used are:

- **FreeContainer:** A FreeContainer represents a ‘top-level window’. Typical examples are dialogue boxes and main application windows.
**Inputter:** An Inputter is used to receive data from the end-user and transfer it into the system. A typical example is a text input field, into which the user may type alphanumeric data such as an address or date of birth.

**Display:** A Display is used to present data from the system to the end-user. A typical example is a (read-only) table, which may be used to present tabular data such as bank account information.

**Editor:** An Editor is used by the end-user to alter existing data. A typical example would be an *updatable* table.

**Chooser:** A Chooser allows the end-user to make a constrained choice, by presenting a list of alternatives for selection. A typical example is a drop-down selection or a pick-list.

**ActionItem:** An ActionItem allows the end-user to initiate some behaviour in the system. Typical examples are buttons and menu items.

Within the APM, these abstract interactors are shown simply as rectangles with the category type following the name of the interactor in square brackets. For example, a TextGrabber widget used to input the name of the author of the book sought would be labelled Author [Inputter]. This can be seen in Fig. 2(a) later.

These abstract categories can be represented by any concrete interactor from the Swing widget set, provided that widget is capable of realizing the category in question. The choice of concrete interactor used can be switched dynamically at runtime, allowing sweeping changes to the look and feel of the interface without affecting its functionality. Further details of the presentation model may be found in [25].

### 3.4. Using the models

As described in the remainder of the paper, Teallach provides tools to author the individual models, and a rich set of techniques for generating and linking these models. Structures in one model can be used automatically to generate corresponding structures in another model; these generated structures may then be used as-is, or used as the basis for further refinement. Existing structures in different models may be linked together, to indicate their intended interaction; and state objects (see Section 7) may be used to represent information-flow within the models, and between the user and the interface. Model development is subject to incremental validation, and when finalized, a complete running interface with ‘help’ functionality is automatically generated.

The user-interfaces generated by Teallach are realized as compiled Java applications, giving the efficiency of compiled code while running on any major computing platform. The widgets used are taken from Java’s Swing widget set [16], but additional user-supplied widgets may be added to the toolkit and used at any time. The user-interface accesses the database through Teallach’s ODMG-style domain API.
4. Designing with teallach

One of the principal aims of Teallach is the provision of a flexible design method such that designers using Teallach are not restricted to a single developmental strategy. Early in the project, we interviewed several practising user-interface designers, and soon realized that different people work in different ways. 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 back to application functionality. Teallach can support both of these approaches, and others.

We believe that forcing designers along a linear developmental path is restrictive and does not support the characteristics of the software development life cycles that designers often favour. For example, TADEUS [9] operates by successively refining a task model through prescribed stages. Teallach recognizes that if it is to be adopted, then it needs to accommodate the developmental preferences 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 [17]. Through its flexible methodology, Teallach aims to support as far as possible the varied observed working practices of software developers.

Teallach’s flexible design approach 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 alternatively to ‘wire together’ user-built substructures without using automatic generation.

Fig. 1 shows an overview of the flexible method proposed by Teallach, which is supported by the Teallach tools. Due to the number of potential routes through the method, 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 will demonstrate some possible paths 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.

Teallach has been developed to facilitate the design of user-interfaces to pre-existing object-oriented database (OODB) applications. There is, therefore, a basic assumption that the schema and classes for the underlying database must exist prior to the process of user interface development. The Teallach tool therefore permits one entry point to the developmental cycle (as shown in Fig. 1). 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). Thereafter,
the developer is then free to design and augment 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 (described in Section 9) which may be required for the runtime operation of the application — see the steps labelled (a) in Fig. 1. The inclusion of each of these additional 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 Fig. 1. Similarly, the developer can, independent of the domain or 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 Fig. 2]. Each of these activities can be performed at any stage during the user-interface development process.

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 Fig. 1, and described in detail in Section 7. These steps can be performed repeatedly and in any order.

Finally, a complete, independent user-interface can be generated from the resultant design.

5. Running example

The use of the Teallach tool will be explained with reference to a simple running example, which is a library database application — the UML class diagram of this is shown in Fig. 2.

For brevity, consider a single task that the user of a library application might perform — that of 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 parameterized by the specified information) and is presented with the resultant information. Assuming one or more books were returned, the user can then 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.
6. The teallach tools

6.1. Overview

Teallach provides separate editors for its domain, task and presentation 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).

As the order of development is free, the models are stored individually, and links between models are stored as they are created. The system attempts to maintain the integrity of the models, for example by allowing only meaningful links to be made, and by ensuring consistency when elements of models are updated or deleted (prompting the user for information as necessary). However, since models may be developed incrementally and piecewise, it is finally necessary for the system to verify their overall consistency before generating code for the end-interface.

6.2. 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, so 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 2.0 object database standard [15].

The domain model editor within the Teallach tool comprises four tabbed panels representing distinct but related concepts: persistent data components; imported auxiliary classes; auxiliary classes derived from the Java API and the database connectivity aspects of the application. The domain model as a whole serves only as an information source thus its description of the above features is used by the other models but is not affected by them in any way. The use of auxiliary classes and the Java API is described in Section 9, so let us briefly consider the persistent and database connectivity components now.

When creating a new project, Teallach automatically analyses of the schema of the underlying database to create a model of the persistent types used (see Section 4).

Even database systems that claim ODMG compliance vary considerably in their treatment of metadata, so this process of creating the domain model has to be unique for each supported product. However, the results of the analysis are presented in an
identical manner thus shielding the designer from any need to understand the complexities of the specific underlying database management system. The domain model editor in Fig. 3(a) shows the persistent data components of the domain model that represents the schema described in the case study.

The database panel in the domain model editor [see Fig. 3(b)] is concerned with the establishment of classes required to support database connectivity and querying. As with the other domain model structures, the database connectivity components adopt and extend the ODMG standard to further strengthen database connectivity correctness. That is, database connections are created by means of a database-connection factory; each database connection has a nested transaction factory, which allows generation of transactions scoped to the encompassing connection; and, perpetuating the isolation achieved by nesting, the transactions all have a dedicated query factory to enable scoped query construction. The developer creates instances of the connections, transactions and queries as required for the runtime operation of the application. These can then be linked to the task model in the guise

---

Fig. 3. Teallach’s domain model editor, showing the Persistent panel (a) and the Database panel (b).
of state objects and the runtime invocation of their associated methods specified. The domain model’s enforced nesting ensures that the scoping of queries inside transactions and transactions inside database connections is always in place and thus helps reduce runtime exceptions in the generated application. Fig. 3 shows the representation of a database connection and nested transaction as required for the search-for-book example.

6.3. 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 through the developed interface (i.e. searching for a book). The task model tool also allows the designer to specify information-flow by declaring local state objects and associate these with tasks, and indicating how this state information is initialized and used within the task model. This process is used to describe both the flow of information between the user and the database application, and the user-interface’s internal information flows.

To realize the task model for the case study, the designer constructs a high-level specification of the task he intends end-users of the application 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 Fig. 4(a)] and drops it at the required location in the task model construction area. The task model is a hierarchy of task instances, as shown in the figure. Each task instance is labelled by an icon showing its type (e.g. the → symbol for sequential task, OI for order-independent tasks).

Fig. 4. (a) Teallach’s task editor, and (b) pop-up task-customization form.
At the lowest level in the task hierarchy, the designer creates interaction and action tasks which represent how the application processes information and invokes behaviour, 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 realized and invoked through a suitably initialized state object.

At any point in its development, the characteristics of a task can be edited through customized, pop-up forms such as that shown in Fig. 4(b). This allows the user to specify the attributes appropriate to a task instance of this type, such as whether it is cancellable, or whether any interaction object or help text is associated with the task. The pop-up may also be used to change the type of the task if required.

If an invoked method is capable of raising an exception, then the task model tool will place a red ball icon by the corresponding action task; exception handling is specified in the task tool by the designer dragging a thread from the red ball towards the task that catches the exception. The designer can then specify what should happen if such an exception is in the generated interface at runtime, i.e. the application could either exit or continue with another task. A pop-up dialogue allows the designer to specify a text message that will be shown at runtime to explain what has happened.

6.4. 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

---

(b) Fig. 4 (continued).
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, with a displayed label) is a user-supplied widget. 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 realized by many different concrete widgets. For example, the category Inputter represents anything that may capture the user’s input; both a JPasswordField and a TextGrabber may serve as realizations of Inputter. A fuller description of the abstract categories available, and how they can be realised, may be found in [25]. Fig. 5 shows both the abstract and the concrete views of the presentation model.

In Fig. 6(a), the designer is specifying which concrete widget is to be used to realise an Inputter; in this case, a TextGrabber. This is done using the automatically customized pop-up menu which shows all available realizations for a particular abstract interactor in the presentation model.

The designer may insert either concrete or abstract presentation objects, and intermix these freely. Where abstract interactors are used, a decision must be made as to which possible realization will be used in the final interface; a default is always provided, so a valid interface is defined at all times. Details of how abstract categories are realized by concrete widgets effectively define a notion of style, so that consistency of look-and-feel can be achieved and differing interfaces can be easily generated which support the same functionality.

The designer may reuse existing presentation structures, generate them from other models or explicitly assemble them. Figs. 5 and 6 show this explicit assembly, 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 [seen in Fig. 7(b)], the designer has selected three Inputter components, to capture the title, author and year information for the book sought. (Section 8 describes how such input can be validated.) These components have been placed within a JPanel, which is a concrete (but usually invisible) container provided by Swing for grouping together related items and ensuring that they behave as desired when the window is resized.

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).

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. Option (1) could indicate a presentation-first approach to interface-construction or simply that the

---

1The designer may explicitly create quit buttons to invoked any specific appropriate functionality, such as closing a database connection; in addition, where tasks are designated as cancellable, the code-generator (described in Section 8) automatically adds a cancel button to allow termination of the task.
designer has recognized that a compound widget can be re-used in this or other projects. Option (2) is appropriate where the models are developed fairly independently, possibly by different designers, and then subsequently integrated. Option (3) is useful to bootstrap the development of the task model, where the presentation model has been substantially developed first.

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 Fig. 7; this allows the designer to assess the immediate visual impact of properties of the presentation such as colour, font and layout.

7. Integrating the models

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 particular widget is to be used for a particular
task, or that an particular action task corresponds to an invocation of a particular operation on an application class. Links between the task and presentation models are also used to describe dialogue dynamics. This section will show how state objects can be used to integrate the various models.

A state object is the means by which the task model maintains references to constructs from another Teallach model, and is constructed through either a paste as state menu option, or as a side-effect of Teallach’s automatically generating a task construct from another Teallach model construct. A state object refers to a named instance of either a domain or (abstract or concrete) presentation model class, and is shown as a uniquely named rectangle within the scope of a non-primitive task. This can be seen in the ‘specify search criteria’ task in later Fig. 10.

Once a state object has been created it can be used 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 properties can be read from or written to (equivalent to specifying the flow of information between the user and the underlying application, or an intra-application information-flow).

For the purposes of our example, 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 it into the specify book information task selected in the task model tool using the paste as state option from the main window’s Edit menu. Similarly, using the Database Connectivity editor pane, a state object corresponding to a new query is copied from the domain model to the Search for a Book task. This query is expressed in ODMG’s standard query language, OQL.
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 Fig. 10, the task model editor shows the currentBook state object.

The following subsections 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 the generate mechanisms.

7.1. Linking task and domain model constructs

Once the designer has created the necessary state objects, he can link action or interaction tasks with them. For example, it is necessary to show that the Perform Search action task invokes the execute () method on the query1: OQLQuery state object. The designer achieves this by 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.

If the link operation is permissible, Teallach invokes its Link Wizard to guide the designer through the potentially complex task of creating the link. As shown in Fig. 8, the Link Wizard recognizes 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 he wishes to invoke by providing 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 Fig. 9, where the Link Wizard is enquiring where the collection of Books returned by the query1: execute () method will be stored. This method has return type Any, ODMG’s universal

![Link Wizard](image)

Fig. 8. Teallach’s link-wizard.
union type,\textsuperscript{2} indicating that the type of object returned is not, in general, knowable in advance. The designer selects the \texttt{foundBooks: Object} state object within the \textit{Search for a Book} task.

If the chosen domain method raises any exceptions (e.g. an \texttt{IllegalOperationException}), then the task model editor will display a red ball 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; for example, the task model editor can be used to specify that if the \texttt{query1: execute} method (as used by the \textit{Perform Search} action task) raises an exception, then the \textit{Search for a Book} task should be performed.

\subsection{7.2. 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 \textit{Get Title} interaction task corresponds to a particular Inputter in the presentation model), then these can be linked, and a connecting state object is created automatically. It should be noted that it is actually the APM category that corresponds to the CPM widget that is used, thereby preserving flexibility in the presentation model: any appropriate concrete widget can be selected to realize the abstract category at or before interface-generation time. If the link operation is accepted, then the Link Wizard will once again be invoked. This can be seen in \textbf{Fig. 9} which shows the designer drawing a line between the task and Inputter to link them.

Since the semantics of this link operation are different from 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 \textit{Get Title} interaction task and the \texttt{title: Inputter} state object (realised by a TextGrabber or JTextField widget in the CPM), then the Link Wizard will detect that the task is receiving information, and will proceed with a dialogue which will ascertain the type of the information

\textsuperscript{2}Java’s universal union type is called \textit{Object}.
being processed, and which state object(s) will provide this information. This can be seen in the Link Wizard dialogue shown in Fig. 11, where the designer is in the process of specifying that the state object’s setTitle(String) method should be used to store the title information gathered through the Inputter.
By linking task and presentation model components the designer is also specifying the dynamics (dialogue) 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.

7.3. Establishing model interaction using generate-mode

To assist the designer in the process of constructing a consistent set of models, Teallach also 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, the 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 user-interface form to represent the required task.

7.4. Model integration and development approaches

It can be seen that the various modes of generating and linking models can be marshalled in support of the different possible paths in the development of a project, as outlined in Section 4.

During model-generation, the domain model always serves as a source, never a target. This is because Teallach is used to design user-interfaces to existing OODB applications, not to design these applications themselves. Thus we expect the domain model already to exist at the start of a project, but we can use Teallach’s generation mode to bootstrap the elaboration of corresponding model structures for task and presentation.

As the task and presentation models can serve both as sources and targets of generation, the designer is free to develop either of these first, and may then use one model to bootstrap the elaboration of the other. Of course, an entire model not need be developed before another, so fragments of one model may be developed and used as the basis of corresponding fragments in the other model, whilst on other subareas of the two models this ordering may be reversed. Finally, the developer is free to use no model-generation, and develop the task and presentation models entirely
independently; this is particularly useful where more than one developer collaborate on the same project.

Once models or model fragments have been developed by any of these approaches, Teallach’s model linkage facilities may be used to associate components in each; of course, some linkage may have been automatically performed if model-generation was used. Since the only constraint on model linkage is that the models must be completely and consistently linked before the end-interface can be generated, linkages may be performed in any order. As the models are linked, automatic checking is applied to ensure the consistency of the models; however, to allow freedom of working, completeness-checking is delayed until code-generation time.

Furthermore, appropriate use of generation and linking facilities allows subcycles of construction/generation and linking to be intermixed, permitting interfaces to be developed part by part, if the designer prefers.

8. Generating interfaces

Although many MB-UlDEs make no attempt to generate running interfaces from their declarative models, a number of proposals include mechanisms for supporting the production, and not only the design, of user-interfaces. Several different strategies have been adopted. For example, the models of ITS [18] are executed directly by an interpreter, whereas in Humanoid [19], TADEUS [20] and FUSE [10] the models are mapped to an alternative representation that is interpreted by a user-interface management system. There is also a small number of examples where programs are generated in a conventional high-level language such as C++, for example JANUS [21] and Mastermind [22].

The Teallach code-generator produces a complete Java source program. This provides a complete implementation of the interface, in contrast to other systems such as MOBI-D [14], which generate only stubs as a basis for code that the developer must write. Teallach’s approach is possible since we rely only on a complete set of (consistent) models as input for the code-generator; the process whereby these models were generated has no impact on the code-generation process.

We have chosen to generate executable code directly to minimize runtime overheads, to maximize portability, and to allow use to be made of existing mainstream widget sets and class libraries from running interfaces, and to allow direct calls to the underlying OODB application.

8.1. Generating java code

The code-generator produces as output Java code that exploits the Swing widget set. Swing applications are organised using the model-view-controller (MVC) pattern. MVC is a widely used organizational framework for the development of user-interface-programs [23]. 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.
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
of the complexity of the user-interface code in the class library as possible, thereby
avoiding the generation of complex programs.
To implement the interface developed in the running example, a number of Java source
files are created; these files contain generated code, based on the generator’s class library.
The application is then created by compiling and running these files independently of
Teallach. More details of the code-generation process may be found in [26].

9. Customizing the environment

Teallach allows its user to design an interface using the basic constructs provided
in each of its models. However, a degree of further extensibility is also possible.
The domain model allows the designer to import classes from the standard Java
API for use in interfaces; furthermore, additional user-supplied classes may also be
imported. The domain editor supplies a tab for each of these functionalities. This
facility is sited in the domain model, as these auxiliary classes are, like the database
classes, external objects described in terms of ODMG constructs which can be linked
into the generated code.
These auxiliary classes are not database classes, but provide functionality required
for the runtime operation of the application. To import a package or class, the
designer must simply specify its fully qualified name. The screen shot in Fig. 12
shows YearAuthenticator, an auxiliary class that has been imported for use in the
example. It provides the facility for validation of a string of characters as
representing 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 ability to import additional classes is very important, as Teallach does not
provide a general-purpose programming environment. However, during the
specification of an interface, various application-specific formatting or validating
functions are often required. The auxiliary classes of the domain model provide a
way to meet this requirement, using either standard classes of the Java API, or the
developer’s own purpose-written classes.
When constructing presentations, the designer may sometimes require widgets not
provided by Swing. Specific applications may require complex, customized
interaction items; for example, a chemistry database application may require a
molecule-viewer widget. Such third-party widgets can be made available to the
designer by registering them with the presentation model. In order to be registered, a
component must be a JavaBean — in other words, it must conform to Java’s
component model. Other kinds of components may be used by creating a Java
wrapper for them which presents them as a JavaBean. Once registered, the new item
becomes available for use under the ‘User’ tab of the widget palette.
The registration process may be facilitated by a tool such as the meta-editor shown in Fig. 13. Here the designer is in the process of registering the TextGrabber widget as a kind of Inputter, and has just specified that the concrete widget’s getText() method should be called whenever the abstract interactor is interrogated for the data the end-user has input.

The meta-editor uses the meta-data it collects by introspection to automatically write Java code implementing the newly defined meta-objects, relieving the implementor from this coding effort.

10. Providing help

Contemporary help systems are commonly developed separately from the application that they are to support. This results in considerable time and effort
being spent in manually coding such systems. Furthermore, complex programming is required as the help system will have to continually interact with and access the underlying application. Additionally, there is the maintenance cost of continually updating the help system whenever the application is modified. Teallach provides a means by which an application and its associated help system may be generated automatically, given the declarative models upon which it has been designed.

Ideally, Teallach should be able to automatically infer all the information necessary to generate a help system from the information contained in its three models. Indeed, the dialogue information inherent in the task model allows Teallach to explain the interrelationships between task items. Sometimes however, a particular task will require further explanation by the interface designer. In this case, the task model provides a task customization form [Fig. 4(b)] to allow the input of help information specific to a particular task. When the designer is ready to generate an interface, he can optionally indicate (through the use of a simple tick box in the application generation dialogue) that help should also be generated for the interface. During the interface generation process the help generator produces a set of XML files, which provide the required help descriptions for each task.

If the designer has specified that help should be generated, then an extra Help menu item will be added to the generated interface, as illustrated in Fig. 14(a). When the end-user requests help on a particular item, the help viewer is invoked. The help viewer is a standard component provided by the Java Help API, which allows the HTML help files which were generated along with the application to be viewed according to the relationships specified by the task model’s semantics. Additionally, the help browser contains an automatically generated table of contents, an index, and a search engine. This is shown in Fig. 14(b).

The Help menu in the generated application basically allows the invocation of the help browser to answer the following questions.

- Help — invokes the help system to display an overview of the help available for the generated interface.
- **Help on Item** (also available through the F1 accelerator key) — invokes the help browser with the currently active task as the focus of the browser.
- **What tasks can I perform now?** — invokes the help browser with a dynamically generated set of files which tell users what tasks are available to them at the present time (given the current state of the interface).
- **How can I perform this task?** — invokes the help browser when users click on a particular widget in the interface and tells them the steps necessary to achieve the particular task.
- **Why is this task not allowed?** — invokes the help browser when users click on a particular widget in the interface and tells them why a particular item is greyed out or otherwise unavailable (e.g. they have not yet completed all the steps necessary to activate the task).
• How may I activate this task? — If a widget is unavailable, then the help browser will automatically generate a help file which tells users the steps necessary to activate the specified widget.

11. The implementation

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 [24].

The models are implemented as instances of Swing’s JTree component, used for working with hierarchical data. The JTree nodes maintain pointers to instances of classes representing the constructs in each of the models. Data are exchanged between the models using Java’s Transferable protocol; projects are saved and recovered using Java’s serialization facilities.

The domain model contains classes representing the standard ODMG constructs; these classes are instantiated to create any particular database schema. These classes link to the database using its Java Application Programmers’ Interface, allowing Teallach to access the stored data.

State objects within the task model maintain references to objects in the other models, encoding the associations required by the code-generator.

The presentation model represents abstract categories as first-class objects, and uses a switching mechanism to handle their realizations. The presentation tool traverses the structure of the hierarchical model in order to construct the preview of the interface as required. The code-generator essentially traverses the task model, generating code for each node, which in turn involves the planting of calls to the domain model and further traversals within the presentation model. The code-generator may also create widgets that are not modelled explicitly in the presentation model, which are required for dialogue-control, such as additional confirm and cancel buttons. A detailed description of the code-generator may be found in [26].

12. Future work

One of the chief goals of the Teallach project was to devise a tool that would allow user-interface designers to work in unconstrained and varied ways. We have demonstrated this goal in our own experiments, using the tool in different ways to build interfaces and fragments of interfaces. However, the very flexibility we have
achieved makes it difficult to evaluate the system overall in a deterministic way: the criterion of success can be only ‘how well did it help you do what you wanted to do?’

However, as we gain experience over time with particular aspects of the tool, we plan to conduct more restricted evaluations on how particular features of the system might impact on particular subactivities involved in the process of developing user-interfaces. The results of these studies will be reported to the community in due course. In addition, many of the insights gained during the course of the project are now informing further research in user-interfaces at each of the partner sites.

13. Summary

This paper has described the software produced by the Teallach project, and shown how we have addressed the issues requiring research that we identified in Section 3. In particular, we have emphasized the need to support a flexible approach to design, to support designers’ preferred methods of working. The Teallach system provides a rich set of tools that support the construction of models and the specification of their interactions.

The flexibility we desired has posed many challenges for the Teallach design team, as a flexible design method often requires complex control facilities. We have therefore concentrated on:

- providing a core set of design primitives, including database-specific concepts (e.g. for building models individually, providing simple click-and-link or paste-as-state operations for inter-model linking);
- allowing users to extend the set of available building-blocks (e.g. by importing auxiliary classes or new widgets);
- enhancing the system with higher-level design functionality based on the core primitives, supporting more rapid design (e.g. by automatically generating model constructs from the information modelled in other Teallach models) and
- automating the final stages of the development cycle (e.g. by generating the end-interfaces and implanting help functionality).

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 other members of the Teallach team for their contributions to this work; they are Grant Forrester, Andrew Dinn, Michael Smyth, Carole Goble and Adrian West. We also wish to thank Paulo Pinheiro da Silva for his work on the code-generator, and Brendan Tourish for his work on the help system.
References


