An Open Model-Based Interface Development System: The Teallach Approach

T.Griffiths[?], J. McKirdy[?], N.Paton[?], J.Kennedy[?], R.Cooper[?], P.Barclay[?], C.Goble[?], P.Gray[?], M. Smyth[?], A. West[?], A.Dinn[?]

[?] Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, England. http://img.cs.man.ac.uk
[?]Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow G12 8QQ, Scotland. http://www.dcs.gla.ac.uk/ {~pdg | ~jo | ~rich }
[?] Department of Computer Studies, Napier University, Canal Court, 42 Craiglockhart Ave, Edinburgh, EH14 1LT, Scotland. http://www.dcs.napier.ac.uk/osg/

Abstract. The goal of the Teallach project is to provide facilities for the systematic development of interfaces to object databases in a manner which is independent of both a specific underlying database and operating system. Teallach's open architecture also allows the creation of interfaces to non-database applications in a platform-independent manner. To this end Teallach adopts model-based techniques in the process of interface construction, exploits the cross-platform capabilities of Java, and utilises the Java Beans API to allow third-party interface components to be exploited. Through the use of a simple case study this paper introduces the Teallach approach to interface development, providing an overview of the system, its motivations, and underlying technology.

1 Introduction

1.1 Motivation

Recent research and development work in database management systems has emphasised the extension of conventional database systems with enhanced data modelling features or languages [9]. Furthermore, the behaviour modelling facilities of databases are no longer limited to simple query languages, as database programming languages and systems support the storage of imperative programs, active rules, deductive rules and integrity constraints in the database alongside the stored data with which they are associated. This increase in the functionality of database systems has been motivated by the needs identified in advanced applications in a wide range of scientific, engineering, geographic and commercial domains.

This increase in the facilities supported by database systems has, however, been associated with a corresponding increase in their complexity. This, in turn, presents new challenges to the designers and developers of user interfaces to databases, who must seek to provide effective tools for the design, browsing, querying, updating, maintenance and debugging of ever more sophisticated database applications. It is

clear, however, that the increasing capabilities of database management systems has not been accompanied by corresponding improvements in the utility of their interfaces. This mismatch in the rates at which database systems and their interfaces are developing, if not addressed, is likely to limit the effective exploitation of advanced database technologies, even in applications where there are clear needs for enhanced database capabilities.

Teallach[1] is a model-based user interface development environment (MB-UIDE) whose primary aim is to provide a software workbench facilitating the rapid development of interfaces in a manner which is *independent of both a specific underlying database system and operating system.* To this end, Teallach has adopted model-based techniques in the process of interface construction, exploits the cross-platform capabilities of Java, and utilises the Java Beans API to allow third-party interface components to be exploited. By these means Teallach will provide a means of moving database interface development from an ad hoc, system-specific context towards more general and well founded solutions.

1.2 Model-based User Interface Development Environments

MB-UIDEs have emerged as a promising paradigm for supporting the systematic and efficient development of user interfaces [6, 8, 4, 7, 15]. MB-UIDEs support both rapid prototyping through automatic generation of (preliminary) interfaces from partial descriptions of applications, and a methodology that encourages discipline in interface design. However, MB-UIDEs are not yet mature, and proposals differ significantly in the range and nature of the models supported.

From the viewpoint of the developer, the key components of a MB-UIDE are the declarative models which store a conceptual representation of the required interface. During recent years, the models supported by MB-UIDEs have increased both in number and in expressiveness. The first-generation of MB-UIDEs (typified by UIDE [6], MECANO [15], AME [10] and JANUS [2]) concentrated on modelling the underlying application domain through a domain model. Typically, such a limited view of the modelled domain produced simple menu or forms-based interfaces. In recent years however, MB-UIDEs such as TADEUS [16, 5], DRIVE [11] and MASTERMIND [18] have exploited a much wider range of interacting models, with a consequent increase in the quality and variety of their generated interfaces. These models allow the designer to exploit fully the information gathered during the requirements analysis phase of the development process.

1.3 Interfacing MB-UIDEs to External Applications

A MB-UIDE is a system that automatically generates a user interface from a set of declarative specifications (models) which describe the tasks that the end-users of an application wish to perform in an interface in terms of either a task or application domain specification (or a combination of both). Typically, a MB-UIDE may be expected to be used in conjunction with existing software systems (e.g. databases) and established display techniques. As such, MB-UIDEs should be developed in an open manner which supports interfacing to external components. As illustrated in figure 1,

[1] In Gaelic, teallach (pronounced: *tyaloch*) is usually taken to mean a smith's forge although, in the past, it was also used to refer to an anvil or furnace; it is a place where tools are made.

if a MB-UIDE is to be considered *open*, it must allow efficient and transparent interfacing to both external applications and interface components.

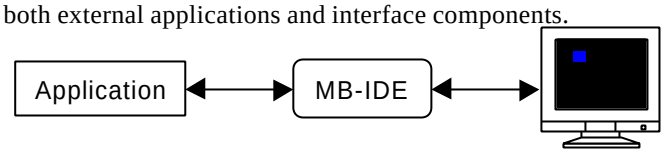


Fig. 1. Interfacing to External Applications

While the Teallach project has its roots in generating user interfaces to databases, the project has identified several important features which must be considered when interacting with external applications in general, namely:

- ***Application knowledge***. A MB-UIDE has little or no control over the internal workings of an external application. If an application is capable of raising exception conditions or being interrupted, then the generated user interface must be capable of responding appropriately.
- ***Limited interface components***. The components or libraries from which the user interface is constituted in MB-UIDEs are typically fixed, often non-extensible, and frequently platform-specific. An open MB-UIDE should be capable of using, extending or customising existing interface components.
- ***External Data flow***. The user interface needs to consider and handle interactive dialogue with the application. An open MB-UIDE should be capable of both gathering information from an application, and presenting information to the application.
- ***Internal Data flow***. The user interface must handle the flow of application, interface, and transient data that is relevant to user tasks.

To date, many MB-UIDEs have concentrated on capturing the declarative semantics of users tasks using both task and domain-centric methods, yet have failed to provide comprehensive support for the four core activities stated above.

This paper is structured as follows. Section 2 provides an overview of the Teallach models and their relationships. These are discussed in greater detail in section 3 through the use of a case study. Section 4 discusses how information captured by the Teallach models can be used to generate a user interface. Finally section 5 draws some conclusions and identifies research directions for the Teallach project.

2 An Overview of Teallach

Teallach has adopted a model-based approach to user interface development. To this end Teallach utilises four declarative models in the process of generating a user interface, namely the *domain*, *user*, *task*, and *presentation* models. In the Teallach approach many of the concepts that other MB-UIDEs capture in an explicit dialogue model are partitioned between the task and presentation models. Unlike other MB-UIDEs, Teallach does not impose a highly structured methodology on its users, rather Teallach aims to provide developers with a flexible design methodology, allowing them to construct the models in an order that reflects their preferences or needs. Once completed, the developer can automatically generate a user interface from the complete model specifications. The relationships between the Teallach models are shown in figure 2.

2.1 Shared model Repository

Teallach uses a shared model repository to allow each model to populate the repository with the specific concepts it is responsible for capturing. Where possible this approach also allows the initial state of each model to be created from the concepts already stored in the repository as populated by other previously constructed models. This is possible since many of the stored concepts are analogous to the concepts utilised by other models. For example, an optional task (as seen by the task model) is viewed by the presentation model as a Java container class whose components are options.

If the underlying application is a database, then the model repository can be stored in the database along with the schema for the application. This is possible because all Teallach models are represented internally using the information structuring facilities of the domain model. This allows Teallach to utilise the storage management and transaction processing mechanisms provided by the database. If no database is present then the storage facilities provided by the Java Serialisation API or persistent Java through PJama [1] will be utilised.

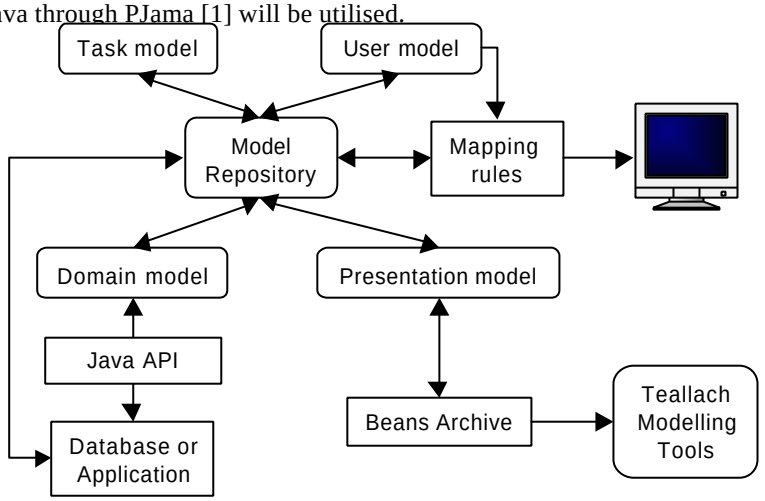


Fig. 2. The Teallach Models and their Relationships

2.2 Domain Model

The domain model (DM) is used to capture the functionality of the underlying application or database. This model therefore specifies the application's public interface in terms of the low-level data sources and services the application makes available to the user. Since Teallach's prime motivation is to facilitate the creation of interfaces to object databases which conform to the ODMG standard [13], the DM views the database (through its ODMG Java binding) as a set of Java classes which reflect the structure and functionality of both the underlying object database and its meta-data. If Teallach is to act as an MB-UIDE for non-database applications, then the applications must provide methods that allow them both to appear as a set of Java classes and that allow the interrogation of their classes used system through appropriate meta-data access facilities. One means by which this may be achieved is through a CORBA [14] application interface, as the CORBA object model is essentially a subset of that provided by ODMG. In Teallach the DM is visually

represented using a subset of the class diagram notation of UML [19].

2.3 Task Model

The task model (TM) expresses the activities that end-users of an application want to undertake and any ordering constraints that exist between tasks or sub-tasks. In addition to this task hierarchy, the TM captures cyclic tasks, conditional tasks, data flow, exceptions, interrupts, help, modality, entry and exit points, and (in the specific case of databases) transactions. At its lowest level, TM primitives are bound to DM concepts. The TM can also make reference to transient state information and can pass this state information within its internal structure.

The complexity of the TM is managed by allowing the TM to be viewed as a multi-layered structure, with each layer allowing the expression of further concepts. For example, the designer may start the process of task modelling by creating a task hierarchy and the ordering constraints between these tasks; once completed they may add exceptions or state information using a different view of the same model. This approach facilitates the design process by allowing the designer to visualise either the whole TM or layers of related concepts.

2.4 Presentation Model

The presentation model (PM) exists at two distinct yet complimentary levels. The first level (termed the *Abstract* Presentation Model) provides an abstract view of a generic interface which represents a corresponding task model. The second level (termed the *Concrete* Presentation Model) is realised as a concrete instance of an interface which can be presented to a user; there may be many concrete instances of an abstract presentation model. The PM has two primary responsibilities. Firstly it serves to act as a repository for information relating to the Java Beans[2] with which it can create interfaces, and secondly it allows the designer to alter and fine-tune the appearance of an interface through a design-time tool. The PM can be partially generated from the structure of the TM using a set of mapping rules in combination with the concepts captured by the user model. The PM allows the designer to: edit, change and fine-tune the Java Beans used in the interface; alter the grouping of the Beans; specify event handling mechanisms (since each Bean interacts using a specific event interface); define navigation between windows or frames; and add non-functional enhancements to the interface such as borders. The PM is responsible for maintaining and registering new Java Beans.

It should be noted that the Bean repository maintained by the PM serves a dual purpose since the Teallach's graphical model construction tools will themselves be constructed using Beans. These tools will in turn utilise Bean technology in the definition of the constructs which they utilise to realise their modelled concepts. For example, the TM can use Beans to represent its task types.

2.5 User Model

The user model (UM) captures information about the style of interface preferred by

[2] JavaBeans [3] is the platform-neutral, component architecture for Java. They allows developers to create reusable software components which the PM can use as its interface building blocks.

the intended users or user groups. The UM is used in conjunction with the TM and a set of mapping rules to generate the initial state of the PM. When the underlying application is an object database the UM captures information about the users of the database in terms of their authority level and how this affects the data they can access. The UM can also be used to record information about a user's interface configuration and preferences.

2.6 Mapping Rules

Teallach uses mapping rules in several places in its architecture to allow mappings between the various models. For example, a set of mapping rules exist between the task model and its abstract presentation model counterpart. In addition to these mappings, an additional set of rules exist between the abstract and concrete presentation models. These mapping rules take into consideration the information captured in the user model, to provide the intended users of the system with a generated interface suitable to their requirements.

These mapping rules are simple in nature, selecting from a 1:m correspondence between abstract PM concepts and PM Beans. The mapping rules consult the UM to decide which Bean from those applicable should be used, and reflect the implicit structure of many task model ordering constraints. The mapping rules can also utilise environmental information, such as the target display medium, to affect the characteristics of the generated PM layout. In addition to defining mappings between tasks and Beans, the mapping rules define which layout manager should be used by each container Bean. The notion of layout managers is discussed in section 3.5.3 of this paper.

3 The Teallach Models

3.1 A Simple Case Study

This case study is based upon a library database. Access to the database is assumed to be through an ODMG call interfaces. Rather than attempt to describe an entire application based on this domain, the case study focuses on a single task - *searching for a book*.

After connecting to the library database, to search for a book, the user must first specify that a search is to be performed. A series of search parameters must then be specified. These include: the attribute(s) of the book on which the search is to be based - i.e. author, title or year (or a combination thereof); and the required accuracy of the search - whether approximate results should be returned or only those results which constitute an exact match. By specifying these parameters, the user has effectively constructed a query that can then be submitted to the database. Upon retrieval of the results of the query, the system must present the results to the user in a manner appropriate to the particular interface style, or inform the user that no books were found to match the parameters given. Where results were obtained, the user is then at liberty to browse through them.

3.2 The Domain Model

The DM provides a description of the underlying application through a Java API. For the case study this is realised as a UML description of the set of Java classes

constituting the library schema together with auxiliary classes representing the facilities provided by an ODMG-compliant database. These auxiliary classes allow access to the database meta-data and the services provided by the database. The UML description of these classes is shown in figure 3.

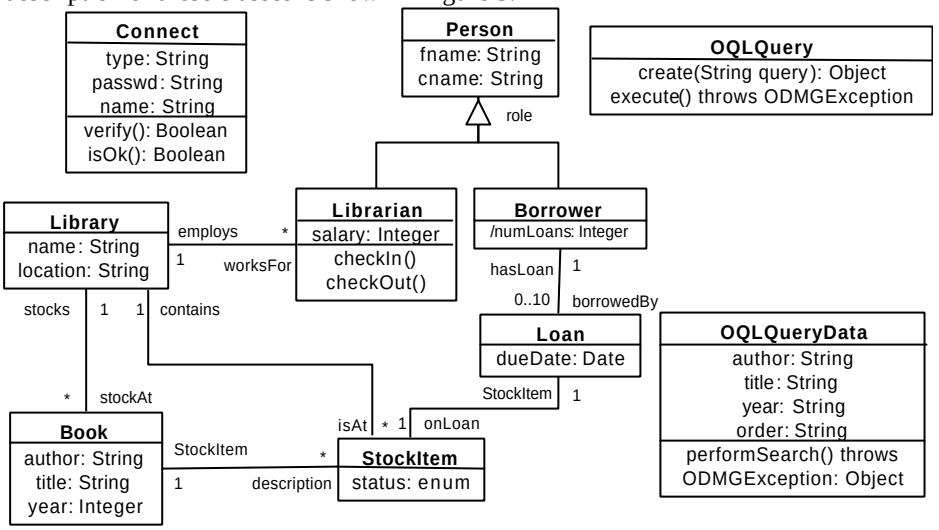


Fig. 3. The Library Case Study Domain Model

The Connect class is responsible for making the connection to the library database given the user type (either librarian or borrower) and their login information, whereas the OQLQuery class executes the given OQL query string and returns its results as a set of Java objects. In addition to modelling the domain schema and meta-data, the domain model allows the definition of *transient* classes. These classes facilitate the modelling of task model concepts such as state information and data flow. An example of a transient domain class is the OQLQueryData class whose purpose is to collect the information necessary for a book search.

3.3 The Task Model

The Teallach TM describes the tasks that application users want to undertake through an interface. The TM is realised using a multi-layered approach capable of capturing many concepts including temporal ordering, state, data flow, exceptions, conditional tasks and undo. In the TM tool these concepts are stacked in concept layers to allow the potential complexity of the model to be managed. The structure and semantics of the initial task hierarchy layer of the TM is similar to the task structures used by other MB-UIDEs such as ADEPT [8] and TADEUS [5].

3.3.1 Task Trees

When viewed in its simplest form, a TM expresses the temporal ordering constraints between sub-tasks. An ordering constraint is specified by the parent task. Initially, we have used seven temporal ordering constraints: sequential, order-independent, repeatable, parallel, interleaved, choice and optional. These are informally defined as:

Sequential	The sub-tasks are performed in strict sequence.
Order-independent	The sub-tasks can be performed in any order, but all sub-tasks must be completed.
Repeatable	The sub-tasks are to be repeated a specified number of times, or until a condition is satisfied.
Parallel	The sub-tasks are to be performed concurrently.
Interleaved	The sub-tasks are to be performed concurrently, but they may be synchronised at specified points in the task hierarchy.
Choice	Only one of the sub-tasks is to be performed, the user must decide which.
Optional	Between zero and all of the sub-tasks are to be performed, the user must decide which.
Conditional	There exists a choice between sub-tasks which is dependent on a specified condition.

Each task type raises an implicit 'task complete' event when its goal is achieved. In the case of the optional and choice task types any corresponding interface will need to specify the actual event which signals that the user has completed the task. The TM allows previously declared tasks to be referred to at lower levels in the hierarchy, thus allowing task reuse and ensuring consistency across common sub-tasks.

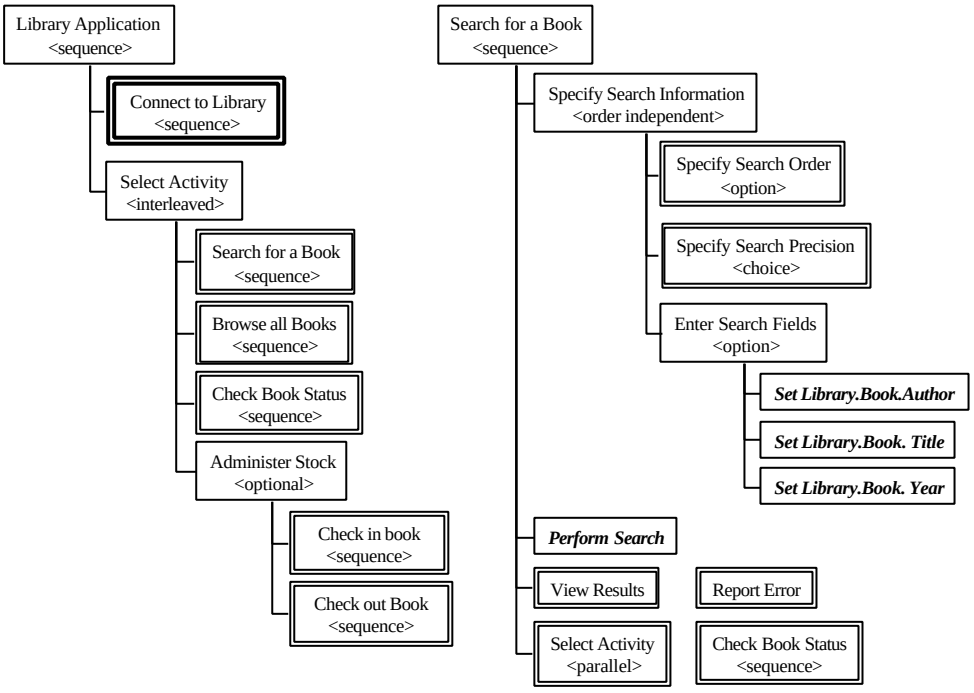


Fig. 4. Case Study Task Tree

The task tree for the case study is shown in figure 4. In this diagram leaf tasks are

mapped to DM concepts, resulting in an implicit TM type system. The direction of information flow in leaf tasks is inferred through the naming conventions adopted from Java. For example, set *Xxxx* infers that information is to be passed from the user to the application whereas get *Xxxx* infers the opposite. When a conditional ordering exists the conditional tasks are shown as adjacent tasks at the same level; the conditional part of this relationship is declared in a finer grained layer of the TM. In figure 4, tasks which require further decomposition are enclosed by double borders, and modal tasks[3] are shown by bold border.

3.3.2 The Task Model Layers

The information contained in the task tree allows the basic task hierarchy to be constructed. The task model allows the capture and visualisation of additional interface and object database concepts by utilising a layered interface.

3.3.2.1 Adding State Information and Data Flow

The conducting of many tasks requires access to information from the DM or provided by the user of the developed interface. To support this, it is necessary to be able to declare local state associated with tasks and to indicate how this state information flows between tasks.

As can be seen from figure 5 the designer can specify both state information and the flow of information into and from sub-tasks. This information is shown superimposed on the basic task hierarchy structure using labelled arcs to indicate the direction of information flow. State information is shown through typed DM concepts whose scope is defined by the enclosing task.

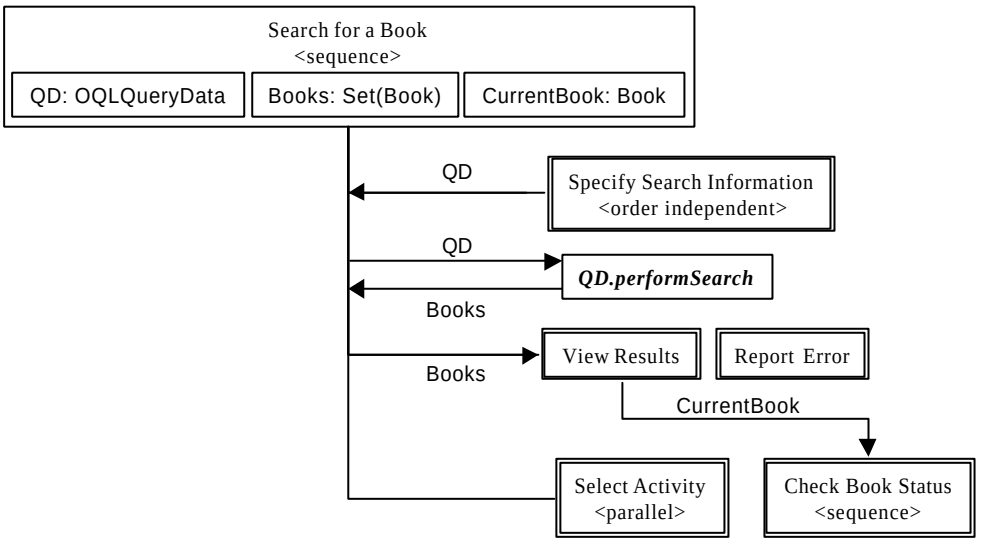


Fig. 5. Case Study Task Model – State and Information Flow Layer

[3] A modal task locks out the execution of any other possibly concurrent tasks.

In figure 5, the fields of an `OQLQueryData` state object are set by the sub-tasks of the `Specify Search Information` task. The populated object is passed to the next task in the sequence. In this figure the `Perform Search` task has been further specified to be a call on the `OQLQueryData` state object's `performSearch` method. This method (realised in Java) constructs an OQL query string from its gathered state information and constructs and executes an `OQLQuery`. The result of this method is another transient state object representing a set of Book objects. These are passed as the input to the `View Results` task. Figure 5 further shows that the View Results task passes individual `Book` objects to the Check `Book Status` task.

3.3.2.2 Adding Navigation Information

Although the task model constructs provide the general control flow information, the designer may periodically need to adapt this control flow in a manner that extends the facilities described above. Figure 6 shows the general structure of the top-level of task model for the case study. In this figure nodes representing interface start and exit points have been explicitly added. The semantics of the exit nodes may be read as: *each sub-task of the `Select Activity` task will have exit available unless otherwise specified*.

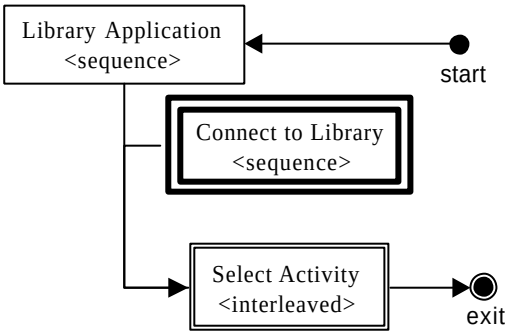


Fig. 6. Case Study Top-level Task Model

Additional task information can be added to the TM through the definition of navigation and conditional transitions; however, only one transition can be taken from a given node. Conditional transitions between task nodes are labelled with a guard condition or an explicit event name. In this example the transition with the guard condition `Books.empty()` refers to a Boolean method of the `Books` state variable declared within the `Search for a Book` task node. This state information is accessible through the state layer of the TM. Figure 7 shows two navigation transitions called `another search`. These allow a designer to specify task branches which do not conform to the temporal ordering specified by a parent node, and may have an optional label. Conditional transitions are labelled with a Boolean guard condition (enclosed by square braces) which allows a decision to be made based upon state information; in this case whether any books were found by the specified search.

Additional concepts can be added to the task graph in any order, but for the sake of this case study we will continue by adding the data flow and state information.

3.3.2.3 Specifying Exceptions and Help

Any external source may fail in some way, giving rise to an exception. As sources in Teallach are wrapped from Java, it is assumed that the Java exception mechanism will be used by them to signal difficulties. The handling of exceptions must therefore be propagated to the descriptions of the tasks that may give rise to them.

By default, exceptions are implicitly thrown to their parent task which may provide a means of catching them. This process continues until a top-level task is reached, upon which a general exception is caught. This default behaviour can, however, be overridden by explicitly providing a modal event handler task for an explicit exception. Figure 8 shows that the `ODMGException` thrown by the `OQLQueryData` object's `performSearch` method uses an explicit exception handler. This handler declares that the `Search for a Book` task will be the next task called after the exception has been successfully handled.

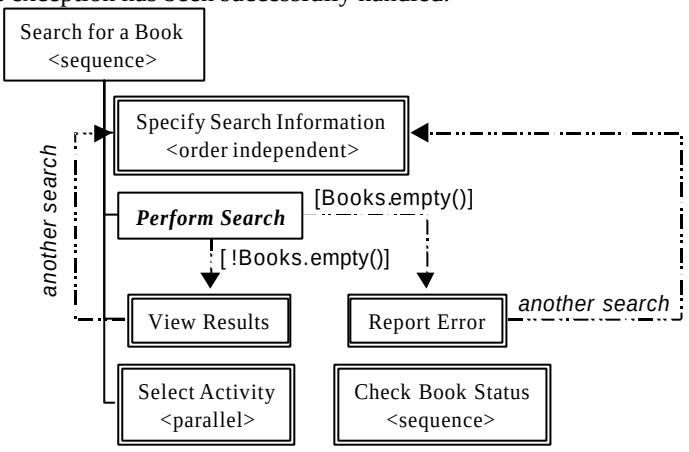


Fig. 7. Adding Navigation Transitions

Figure 8 further declares that help should be available for all sub-tasks of `Select Activity` unless explicitly specified, as indicated by the `View Results` task.

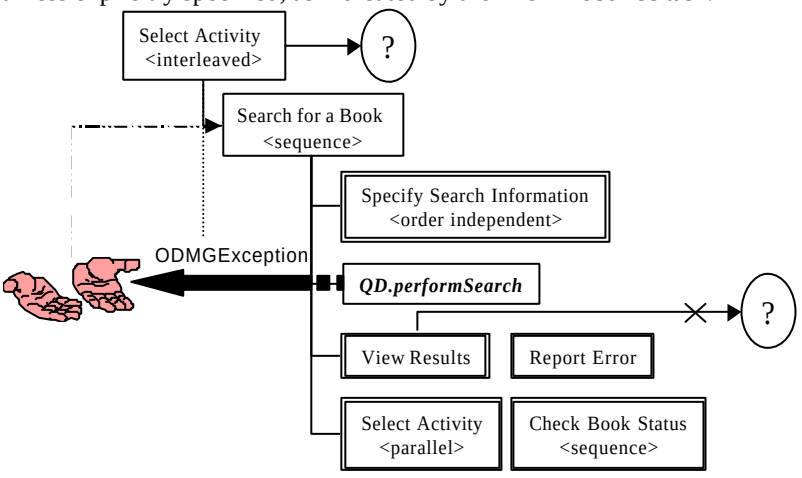


Fig. 8. Case Study Task Model – Exceptions and Help Layer

3.4 The User Model

The user model (UM) allows a designer to specify information about users and user groups in terms of their preferences and authority. The notion of authority is of particular interest and importance when considering user interfaces to databases, where users may have restricted access to the data and restricted permissions on their use of that data. In the library case study the librarians user group will have access to the AdministerStock sub-task hierarchy, whereas the borrowers group will not. By recognising this inherent feature of database systems, the UM can affect the user interface components (and hence the presented interface) which are visible to each group of users. If this feature were not present, the interface generated for each user group would be identical, and would rely on database security features to disallow access to restricted information.

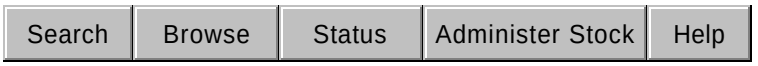
The UM is used in conjunction with the defined mapping rules to allow the construction of the initial state of the PM. The UM is not intended to rely on using a fixed set of attributes with which to model users since this would be both prescriptive and limiting. Rather, in the longer term, an extensible knowledge base of user information will be utilised which will be linked to the set mapping rules. An example of this type of system is the user model of the Adept MB-UIDE [8].

For the purposes of the present prototype, the UM will specify abstract user interface preferences such as "Librarians prefer menus" or "Borrowers prefer iconic displays". Whilst the simple user model proposed for the present prototype does not exploit the full potential of user modelling techniques, it does have the advantage of producing a simple mapping between the TM and PM, thus providing the potential for reverse engineering of a TM from a PM. The case study assumes that the initial state of the PM has been created using mappings such as:

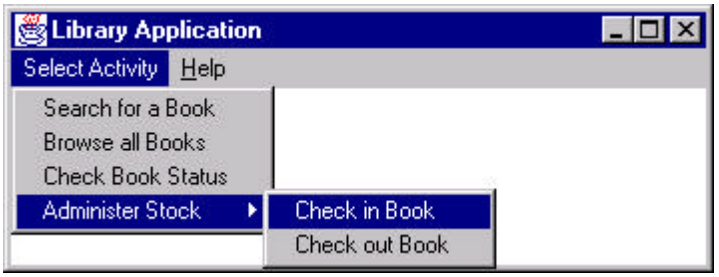
Optional task	→	< Optional Container, Grid layout manager >
Procedure call	→	< LabelledButton >
Interleaved tasks	→	< CardStack Container, Card layout manager > or < menubar, null >

where mappings are of the form < Container Bean, layout manager >, or < Component Bean >. Where a choice is available, the preferences specified in the UM will determine which mapping rules are chosen for each user. For example, figure 9 shows the two interface components corresponding to a set of interleaved tasks according to the above mapping rules. The mapping rule which specifies the CardStack container uses a toolbar to control the stack of interface cards corresponding to the task mapped to each toolbar button.

The mappings utilise the implicit type definitions and information flow direction of the TM concepts in the process of generating a PM. For example, the type of the `setLibrary.Book.Author` implies that a String data input Bean should be used whose label is 'Author'.



Interleaved Tasks Mapped to a Toolbar Controller for a CardStack



Interleaved Tasks Mapped to a Menubar

Fig. 9. Potential Interface Mappings

3.5 The Presentation Model

To further its openness policy, the PM utilises an extensible set of Java Beans in the process of generating a user interface to an application. The Beans used in this process are often diverse in functionality and appearance, but in general can be categorised as either *container* or *component* Beans, with a wide range of interface functionalities being offered by both type of Bean. In the process of developing a presentation the PM must register information about the Beans it is using. This information is gathered using two methods.

- **Introspection**: Java Beans provide methods by which their publicly accessible interface can be discerned. This *introspection* API allows the PM to register information about a Bean's methods and variables and their type system. The introspection API also allows the PM to know about the events that a Bean can generate and respond to. In addition the introspection API provides facilities which allow the designer (through the PM) to customise a Bean's appearance and functionality. These customisations can be saved through Java's serialisation API.
- **Interactive Dialogue**: When the PM is constructing an initial interface specification from the information stored in the shared model repository, it needs to know which of a Bean's methods to bind to the specified task events. Java Beans do not provide any notion of default behaviour for event generation; for example, a Button Bean can respond to many different mouse and keyboard events. When each Bean is initially registered with the PM, the PM administrator must therefore specify which of a Bean's methods will form its default behaviour if any ambiguity is present. The PM administrator can also specify whether the Bean is a container or a component. If it is a container then it is possible to specify the type of the container. For example, will its components be optional or choices.

Through the above methods, Teallach can avoid many of the pitfalls associated with working with external interface components whose closed access to their functionality

is likely to prove restrictive.

The case study assumes that the PM has Beans available which allow the realisation of the required containers (such as a toolbar Bean), and a simple LabelledButton Bean. Within the Teallach project, more sophisticated Beans for visualising generic query results, database schemas, etc, will be developed.

3.5.1 Design Scenarios

Teallach will allow a designer to work in many ways, including generating an application interface from existing task and/or domain model specification(s), or creating an interface in isolation from the other Teallach models, and allowing Teallach to generate the structure of the TM from the developed interface.

Using the first scenario, the initial interface design is realised by a combination of mapping rules and style and preference information gathered by the user model. This 'first-stab' interface it fully editable, allowing the designer to change the selected Beans, alter their placement and appearance, add any missing components that are identified at this stage, and customise the event handling mechanisms utilised by each Bean. Any such changes will be propagated (through the shared model repository) to other interested models. By adopting the second scenario, the designer must 'reverse engineer' the task structure of the constructed interface. This potentially complex prospect is achievable since each Bean has previously registered its default functionality with the PM, and the mappings used by the UM are simple in nature.

It has been noted that one of the shortcomings of MB-UIDEs in general is their lack of ability to propagate any changes made to the designed interface to any underlying models [17]. Teallach's shared model approach to interface design moves towards a solution to this problem by informing the other models of any changes to any concepts in which they have an interest.

Figure 10 shows a possible interface for the `Search for a Book` task using the mapping rules specified in section 3.4.

3.5.2 Presentation Model Structure

The structure of the abstract PM and TM reflect each other, with a mapping existing between each task in the TM and concept in the abstract PM. Once the abstract PM has been constructed, a set of corresponding concrete PMs can be generated using the abstractions defined in the user model.

In its simplest form the concrete PM can be seen to be a hierarchy of containers and components which reflect the required functionality of the TM. The PM however captures many more concepts than this simple structure through its ability to specify events, physical groupings of task concepts, inter and intra-component navigation, and non-functional ornamentation. The PM also allows the designer to specify the dynamic layout of each container through its layout manager feature as discussed in the next section.

In some cases the PM may possess a Bean which has enough functionality to process a complete sub-task hierarchy. This is illustrated in the case study, where it is assumed that the PM has access to a generic query visualisation Bean which will provide the functionality required for the `View Results` task. Alternatively, the designer may define a leaf task which is too complex for any single Bean to capture the required functionality. If this is the case then designer must either continue

decomposing the task (in the TM) until it is defined to a level of detail where individual Beans are available in the PM, or obtain a Bean with the required functionality.

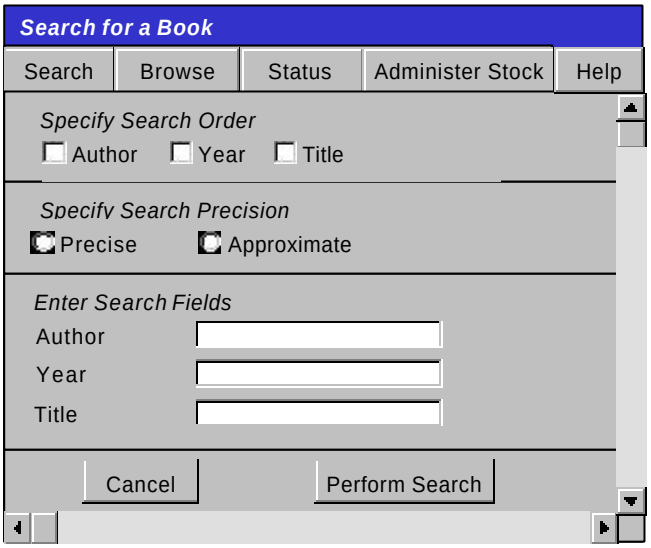


Fig. 11. Possible Interface for the Search for a Book Task

3.5.3 Layout Managers

The PM exploits Java's automatic and dynamic layout facilities, termed *layout managers*. In the PM, a layout manager can be applied to a container Bean to provide a means of defining where interface components will be physically located in the interface. The designer can utilise layout managers by either defining user layout preferences in the user model, or can interactively specify or override which layout managers should be used for each container Bean through PM facilities. Through the use of layout managers, Teallach can simply and effectively achieve the dynamic layout functionality of user interface management systems (UIMS) such as AMULET [12].

For the case study, several layout managers can be utilised according to the required functionality. For example, the `Enter Search Fields` container shown in figure 9 uses the standard flow layout manager which dynamically updates the number of columns and rows needed to view its components as the `Search for a Book` window is resized.

4 Generating the User Interface

There are several options that are available for generating an interface corresponding to the requirements specified in the Teallach model repository, including whether to use an interpreted or compiled solution. Teallach's requirements identify that an interpreted design-time solution for the interactive PM tools is indicated, whereas an optimised compiled version of the production interface has many performance-related benefits. The compiled solution does have drawbacks however, since each time the

designer wishes to generate a test application an expensive re-build operation is necessary. This problem can however be overcome by using techniques such as incremental and background compilation. It is envisaged that the advantages that an interpreted design-time environment will bring will reduce the need for such a requirement.

5 Conclusions and Research Directions

This paper has presented the Teallach approach to developing user interfaces to object databases in particular, and external applications in general. The current research has recognised the need for MB-UIDEs to consider an open approach to interface development, in terms of both interfacing to existing software systems and interface display techniques.

The Teallach MB-UIDE will provide software tools which allows the realisation of the models and methodology discussed in this paper. At the present time the project (having constructed a throw-away prototype for experimental purposes) is completing the design of the second prototype and will shortly commence development of the necessary software tools.

Acknowledgements

This work is funded by UK's Engineering and Physical Sciences Research Council (EPSRC), whose support we are pleased to acknowledge.

6 Bibliography

1. Atkinson, M., Daynes, L., Jordan, M., Printezis, T., Spence, S.: An Orthogonally Persistent Java. ACM SIGMOD Record, Volume 25, Number 4, December 1996
2. Balzert, H., Hofmann, F., Kruschinski, V., Niemann, C.: The Janus Application Development Environment Generating More than the User Interface. In *Computer-Aided Design of User Interfaces* (Vanderdonckt, J. Ed.). Namur University Press, Namur, 1996, pp. 183 – 205.
3. The Java Beans Specification: http://splash.javasoft.com/beans/docs/beans.101.ps
4. Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I., Sacre, B., Vanderdonckt, J.: Towards a Systematic Building of Software Architectures: The TRIDENT Methodological Guide. In *Interactive Systems: Design, Specification and Verification*. Berlin: Springer, 1995, pp 77 – 94.
5. Elwert, T., Schlungbaum, T.: Modelling and Generation of Graphical User Interfaces in the TADEUS Approach. In *Designing, Specification, and Verification of Interactive Systems* (Palanque, P., Bastide, R. Eds.). Wien, Springer, 1995, pp. 193-208.
6. Foley, J. Sukaviriya, P.: History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. In *Interactive Systems: Design, Specification and Verification*. Berlin: Springer, 1995, pp 3 – 14.
7. Janssen, C., Weisbecker, A., Zeigler, J.: Generating User Interfaces from Data Models and Dialogue Net Specifications. In *Bridges between Worlds, Proceedings of InterCHI'93* (Ashlund, S. et al. Eds.), Amsterdam, April 1993. ACM Press, New York 1993, pp 418 – 423.

8. Johnson, P., Johnson, H., Wilson, S.: Rapid Prototyping of User Interfaces Driven by Task Models. In *Scenario-Based Design*, (Carroll, J. Ed). John Wiley & Son (London), 1995, pp. 209 – 246.
9. Kim, W. (Ed.): Modern Database Systems. Addison Wesley, 1995.
10. Märtin, C.: Software Life Cycle Automation for Interactive Applications: The AME Design Environment. In *Computer-Aided Design of User Interfaces* (Vanderdonckt, J.Ed.). Namur University Press, Namur, 1996, pp. 57 – 74.
11. Mitchell, K., Kennedy, J., Barclay, P: A Framework for User Interfaces to Databases. In *ACM International Workshop on Advanced Visual Interfaces*. Gubbio, Italy, 1996.
12. Myers, B.A., et al.: The Amulet Environment: New Models for Effective User Interface Software Development. Carnegie Mellon University School of Computer Science technical report No. CMU-CS-96-189. 1996.
13. Cattell, R.G.G. et al.: The Object Database Standard: 2.0. Morgan Kaufmann Publishers, Inc. 1997.
14. CORBA: Architecture and Specification. Object Management Group Publication Services, 1995.
15. Puerta, A.: The Mecano Project: Comprehensive and Integrated Support for Model-based Interface Development. In *Computer-Aided Design of User Interfaces* (Vanderdonckt, J. Ed.). Namur University Press, Namur, 1996, pp. 19 – 36.
16. Schlungbaum, E., Elwert, T.: Automatic User Interface Generation from Declarative Models. In *Computer-Aided Design of User Interfaces, Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces CADUI'96*, (Vanderdonckt, J. Ed.) Namur, 5-7 June 1996, Presses Universitaires de Namur, Namur, 1996, pp. 3-18.
17. Szekely, P.: Retrospective and Challenges for Model-Based Interface Development. In Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces, (Vanderdonckt, J. Ed.). Namur University Press, Namur, 1996.
18. Szekely, P., Sukaviriya, P., Castells, P., Muhtkumarasamy, J., Salcher, E.: Declarative Interface Models For User Interface Construction Tools: The MASTERMIND Approach. In *Engineering For Human-Computer Interaction*, 1996.
19. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modelling Language Reference Manual. Addison Wesley Longman, Inc. 1998.