ELSEVIER

Interacting with Computers 14 (2001) 31–68

Interacting with Computers

www.elsevier.com/locate/intcom

Teallach: a model-based user interface development environment for object databases☆

Tony Griffiths[a,*], Peter J. Barclay[b], Norman W. Paton[a], Jo McKirdy[c], Jessie Kennedy[b], Philip D. Gray[c], Richard Cooper[c], Carole A. Goble[a], Paulo Pinheiro da Silva[a]

[a]*Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK*
[b]*School of Computing, Napier University, 219 Colinton Road, Edinburgh EH14 1DJ, UK*
[c]*Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK*

Received 7 July 2000; revised 22 March 2001; accepted 4 May 2001

Abstract

Model-based user interface development environments show promise for improving the productivity of user interface developers, and possibly for improving the quality of developed interfaces. While model-based techniques have previously been applied to the area of database interfaces, they have not been specifically targeted at the important area of object database applications. Such applications make use of models that are semantically richer than their relational counterparts in terms of both data structures and application functionality. In general, model-based techniques have not addressed how the information referenced in such applications is manifested within the described models, and is utilised within the generated interface itself. This lack of experience with such systems has led to many model-based projects providing minimal support for certain features that are essential to such data intensive applications, and has prevented object database interface developers in particular from benefiting from model-based techniques. This paper presents the Teallach model-based user interface development environment for object databases, describing the models it supports, the relationships between these models, the tool used to construct interfaces using the models and the generation of Java programs from the declarative models. Distinctive features of Teallach include comprehensive facilities for linking models, a flexible development method, an open architecture, and the generation of running applications based on the models constructed by designers. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Model based user interface development; Object databases; User interfaces to databases

☆ This research is funded by the UK Engineering and Physical Sciences Research Council (EPSRC) -grant number GR/L02692.

* Corresponding author. Tel.: +44-161-275-6139; fax: +44-161-275-6236.
E-mail address: tony.griffiths@cs.man.ac.uk (T. Griffiths).

1. Introduction

In many organisations, a significant proportion of the user interfaces developed are user interfaces to databases (Zloof, 1998). This is reflected in the fact that many commercial interface development packages include facilities tailored for use with databases, and most database vendors supply interface-building tools (e.g. Paddock et al., 1998; Prague et al., 2000). However, such products rarely build upon the most recent research on user interface development environments (such as specifying information flow or automatically generating help or undo facilities), and provide limited support for specifying the dynamics (dialog) of the developed interface. This paper provides an overview of the Teallach project, which has developed a model-based user interface development environment for use with object databases, and in particular those conforming to the ODMG standard (Cattell, 1997).

Model-based user interface development environments (MB-UIDEs) (Paternó, 1999; Szekely, 1995; da Silva, 2000; Griffiths et al., 1998a; Schlungnaum, 1996) seek to describe the functionality of a user interface using a collection of declarative models. In such a context, constructing a user interface involves building and linking a collection of models. In practice, these models often include, but may not be limited to, a domain model, a task model, a dialog model and a presentation model. It can be argued that the model-based approach improves upon current component-based programming environments, for example, by making the interface development process more systematic, and by shielding developers from certain low-level aspects of interface implementation through judicious use of abstract models. The key features of MB-UIDEs are discussed more fully in Section 2. At this point, we note only that MB-UIDEs are the focus of significant attention in the user interface development environment community, and that while model-based techniques have previously been applied to the area of database interfaces, they have not been specifically targeted at the important area of object database applications. For example, GENIUS (Janssen et al., 1993), MacIDA (Petoud and Pigneur, 1989) and TRIDENT (Vanderdonckt and Bodart, 1993; Bodart et al., 1995b) each use an entity-relationship model, and Mecano (Puerta, 1996) and TADEUS (Schlungbaum and Elwert, 1996) both use a generalised object model. Object database applications make use of models that are semantically richer than their relational counterparts in terms of both data structures and application functionality, and because of the reduced distinction between database and application data, (in general) model-based techniques have not considered how the information referenced in such applications is manifested within the described models, and is utilised within the generated interface itself (Griffiths et al., 1998a).

Object databases are now in reasonably widespread use in a range of advanced applications (Chaudhri and Loomis, 1998). These applications are typically characterised by the presence of rich information models and complex processing or analyses, typically referencing large volumes of data. Such applications can therefore be regarded as *data intensive applications*. User interface requirements vary widely from application to application, but many applications use complex custom-built visualisations (e.g. in geographic, scientific or design domains), which indicates a need for an open architecture into which different visual components can be incorporated. Furthermore, many applications must make use of operations associated with database objects, and displays often have to

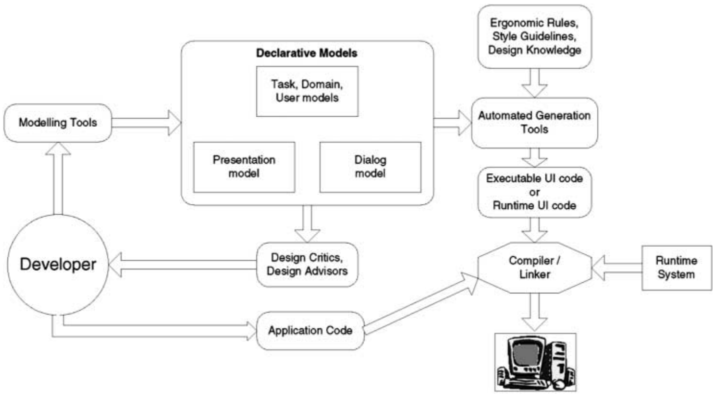

Declarative Models
Task, Domain, User models
Presentation model
Dialog model
Ergonomic Rules, Style Guidelines, Design Knowledge
Automated Generation Tools
Executable UI code or Runtime UI code
Modelling Tools
Developer
Design Critics, Design Advisors
Compiler / Linker
Runtime System
Application Code

Fig. 1. Typical Model-based Development Life-Cycle.

present information that is not directly associated with a small number of persistent objects. This means that interface development environments for object databases must incorporate effective facilities for communication and temporary storage of application data within the interface.

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

The paper is structured as follows. Section 2 provides an introduction to MB-UIDEs, in particular describing and comparing a number of important proposals. Section 3 describes the Teallach models, of which there are three — a domain model, a task model and a presentation model. Section 3 also describes how the models relate to each other, which is important to the functionality of the overall environment, and the design method that this environment supports. Section 5 provides an overview of the flexible design method supported by Teallach, and the application development tools that support the method. Section 6 describes how the models can be executed as Java programs constructed by a code generator. Section 7 illustrates the use of the Teallach models in a case study. Section 8 presents some conclusions.

2. Model-based user interface development environments — an overview

MB-UIDEs can be classified as interface development environments that exhibit three main features, namely they: support the automatic generation of interfaces; utilise declarative methods for specifying interfaces; and adopt a methodology to support the development of the interface. The process of developing a user interface with a MB-UIDE is an iterative process of developing and refining a set of declarative models using graphical editor tools or high-level specification languages. Once developed, these models are transformed according to ergonomic rules and/or style guidelines into an interface specification. This specification is subsequently linked with the underlying application code to generate a running application. This development process is illustrated in Fig. 1. From the designer's viewpoint, the key components of a MBIDE are the declarative models, which store the conceptual representation of the interface. During recent years, the models supported by MBIDEs have increased both in number and in expressiveness, but typically include domain, presentation, task and dialog models. Some MBIDEs (e.g. Trident (Bodart et al., 1995a)) also try to help the designer by providing design critics or advisors. These are tools that analyse models and either suggest improvements or identify inconsistencies or errors in the design.

Early interactive design environments (Buxton et al., 1983; Olsen, 1987) provided programmers with a way of developing user interfaces semi-automatically via the specification of the dialog, or interactive behaviour, of the run-time system. Although clearly superior in some respects to manual coding in a general-purpose language, these systems generated rather crude interfaces, and the designer is left with the substantial job of configuring the system to take into account the nature of the application data and operations, the resources of the graphics or window system, and the abilities of users. Over the past 10 years, one strand of research has focused on enhancing the user interface design environment with additional information organised into components that model design-relevant aspects of the target system and its proposed context of use.

MB-UIDEs may be compared with respect to the number and expressiveness of their models, and the design tools that construct and make use of the model-based information (Paternó, 1999; Szekely, 1995; da Silva, 2000; Griffiths et al., 1998a; Schlungnaum, 1996). Early MB-UIDEs, such as UIDE (Foley et al., 1989) and Humanoid (Szekely et al., 1992), incorporate relatively simple application, dialog and presentation (i.e. user interface component) models, plus a rule-based user interface generator. ADEPT (Johnson et al., 1995) adds a model of user-system tasks, at a higher level of abstraction than dialog, to produce prototype interfaces for evaluation and subsequent refinement. DRIVE (Mitchell et al., 1996) includes a user model that captures capabilities and limitations of the user population for automatic selection of an appropriate interaction technique from several candidates. As well as using sophisticated application, task, dialog and user models, TADEUS (Schlungbaum and Elwert, 1996) partitions presentation specification into abstract and concrete presentation models, representing, respectively, the dialog role and the toolkit-independent appearance and behaviour of interface objects.

In addition to semi-automatic interface generation, a variety of forms of design assistance can be found in MB-UIDEs. Humanoid produces application-related and context-sensitive help and undo/redo facilities. TRIDENT (Bodart et al., 1995a) uses a knowledge base of design guidelines, provide designers with a set of appropriate presentation components plus a human-readable rationale for the selection. TADEUS's dialog model can be translated into a Petri Net for simulation and model checking.

MB-UIDEs have yet to receive widespread acceptance in non-research-based settings, with some researchers (e.g. (Szekely, 1998)) arguing that they are probably best suited to highly domain-specific applications where specialized domain knowledge can be fully exploited. The most common tool for interface construction remains the GUI-builder, which typically holds no information about the components of the system beyond the composition and configurable properties of interaction objects. It has been hypothesised that model-based systems may not yet be expressive enough (Puerta et al., 1994b). Clearly, the models must be able to capture sufficient relevant information to provide useful assistance, and to do so via a specification language and tools, which retain a positive cost/benefit ratio for the developer. Also, generation rules may produce poor initial interfaces via automatic generation. A MB-UIDE must support, or at least accommodate, the construction of user interfaces as sophisticated as those that can be built by other means. Additionally, the tools must not impose an unacceptable method of working on developers (e.g. a rigidly top-down approach that leaves prototype construction until the end of the design process (Griffiths et al., 1998a)).

In the following sections, whenever new Teallach models or features are introduced, they are compared with support for related facilities in other MB-UIDEs. Teallach has many features that are familiar from other MB-UIDEs, but the ways in which its models are related to each other, its flexible design method and tools, its open architecture, and its direct support for code generation, serve to distinguish Teallach from other proposals.

3. The Teallach models

There is as yet no consensus as to what models a MB-UIDE should support (Griffiths et al., 1998a), however, in general, they often include task, domain, application, presentation, dialog and user models. In general, a dialog model allows the dynamics of the target interface to be modelled, in terms of what commands can be invoked, in a presentation-independent manner. During the development of Teallach, we found that much of the information contained in such a dialog model is a refinement of the information already contained in the Teallach task and presentation models. Therefore, rather than create and maintain, a separate (and highly duplicated) dialog model, we instead partition such information between our task and presentation models. This decision also facilitates Teallach's flexible design method, as described in Section 5. A user model identifies and describes a target application's potential user groups in terms of their roles and attributes. MB-UIDEs such as MOBI-D (Puerta and Maulsby, 1997) and ADEPT (Johnson et al., 1995) provide user models that allow their interface specifications to state which user groups can be involved in which tasks. Such specifications result in highly customizable interfaces, but can require extra work on the part of the designer, as they may need to specify different presentations for each presentation or dialog for each identified user group. Teallach's lack of a user model means that its models result in only a single generated interface, with generic applicability.

Teallach has three principal models, *a domain model*, *a task model*, and *a presentation model*, which are described in the following sections. These models, and the relationships between them, form Teallach's central components. Each of Teallach's models defines a view of the information required to generate an interface to a particular application. Each model in effect has expert knowledge about its own view, and defers responsibility for concepts outside its area of specialisation to the other models. The *domain model* describes the underlying application in terms its the data and operations, the *task model* describes

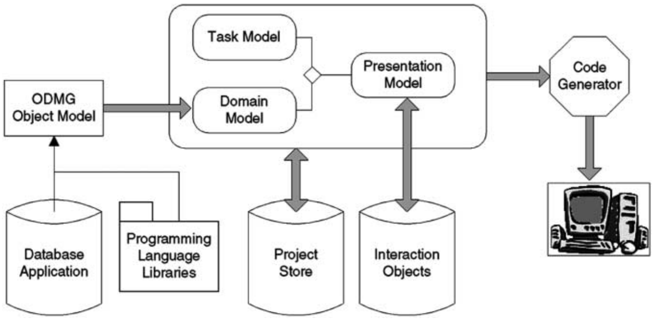


Fig. 2. The Teallach models and architecture.

what users can do with the interface in terms of its dynamics and information processing requirements, and the *presentation model* indicates how the resulting interface will appear. The models have explicit relationships with each other. For example, if the task model needs to reference underlying application functionality, then the domain model provides detailed information as required. If on the other hand a designer wants to customize the visualisation associated with a particular user task, then they use the presentation model to achieve this. The architecture of Teallach is presented in Fig. 2.

The domain model and the presentation model make reference to external resources. In the case of the domain model, these consist of the object database on top of which the application is to be built, and any auxiliary data types to be used in association with the application. In the case of the presentation model, these consist of an existing widget set and any third-party interface components registered with the presentation model. A collection of models constitutes an interface design, and can be stored in a project store. The code generator takes a set of models and generates a Java program that implements the functionality they describe.

3.1. The domain model

The domain model of a MB-UIDE provides a means for the interface under development to make reference to the application for which the interface is being constructed. However, the role of the domain model in different MB-UIDEs varies significantly, reflecting differences in the scope of the MB-UIDEs in question. For example, in Adept (Johnson et al., 1995) the domain model is little more than a list of entity types, and the associations between the domain model and the other models are represented only at a very abstract level. This is because Adept is very much focused on the identification of user requirements, and not on the generation of fully functional interfaces. The position is similar in MOBI-D (Puerta, 1997), in which the MB-UIDE is associated principally with early phases in interface design. In Teallach, because the domain model (Cooper et al., 2000) is the sole way of accessing underlying application and/or object database information, and because complete interface programs must be generated, the domain model is relatively rich and much more closely integrated. Such close integration is not new in model-based systems, for example, UIDE (de Baar et al., 1992), Mecano (Puerta, 1996), and Humaniod (Luo et al., 1993) each exploit tight links to their domain models. Through its knowledge of the ODMG meta-model, Teallach is however able to automatically create its domain model from the information contained in an object database schema. Additionally, the domain model's ability to automatically analyse (introspect) the API of non-database class libraries (including that of any interactors used in the presentation model), and to describe this information to Teallach's other models, provides a degree of sophistication that is not found in other model-based systems.

3.1.1. The ODMG model

In Teallach, the domain model describes external resources using the constructs of the ODMG object database standard (Cattell, 1997). The ODMG model is essentially a mainstream object-oriented data model, and is an extension of the OMG object model.

The basic modelling primitives defined by the ODMG object model are *objects* and *literals*. Objects and literals can be characterised by their *types*, with a particular instantiation of a type being referred to as an *instance*. The ODMG object model partitions its types into three main categories: atomic types, collection types, and structured types. Atomic types are user-defined types, (e.g. `Department`, `Company`, etc.). Collection types represent the well-understood concepts of set, bag, list, array, and dictionary. Instances of collection types are composed of distinct elements of the same type, each of which can be an instance of an atomic type, another collection, or a literal type. Atomic literal types are exemplified by numeric and character types such as oat and string, whereas collection literal types are set, bag, list, array, and dictionary. The ODMG object model supports several structured literals that are used to represent instances of a particular date or time. In addition, users can define their own structured literal types through the `struct` type constructor to represent simple record types.

ODMG objects are characterised by their *state* and their *behaviour*, with an object's state being defined by the values of its *properties*. Two kinds of properties are distinguished by the ODMG object model, namely *attributes* and *relationships*. Attributes are of one object or literal type (e.g. the `Person` object type could have an attribute called name that is of type `string`), whereas relationships are defined between two object types, and may be further categorised as being one-to-one, one-to-many, many-to-one, or many-to-many, according to the number of instances of each type that participate in the relationship. For any relationship, it is possible to traverse the relationship from either of the types participating in the relationship. Thus, for each relationship two traversal paths must be defined, one for each direction of the traversal. For example, if the `Employee` object type declares a one-to-many relationship called `works_for` with the Department type, then `Department` must also declare a relationship called `employs` that is the inverse of `works_for`. An object's behaviour is defined by the set of operations that its type defines. These operations are defined in terms of a set of overloaded operation signatures, which define the type of each argument and any value returned by each operation. The ODMG object model includes a single inheritance mechanism to allow the definition of *generalisation–specialisation* hierarchies.

3.1.2. The domain model in practice

In practice, the Teallach domain model can be used to describe not only information

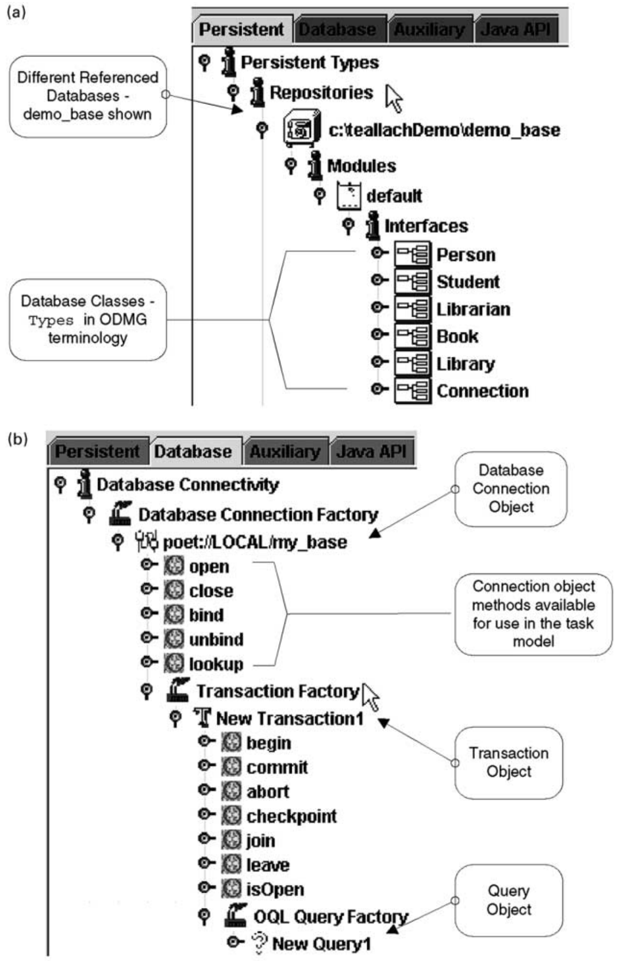


Fig. 3. The domain model tool. (a) Domain Model Persistent Classes; (b) Domain Model Database Classes.

stored in an object database, but also to provide access to auxiliary program classes described using the constructs of the ODMG model, as described below. Overall, the domain model reflects the structure and functionality of the underlying application and, where applicable, database connectivity and interaction. The ODMG object model represents both persistent (database) and transient (application program) types and objects using the same constructs. The Teallach domain model, therefore, uniformly represents both persistent and transient types. There is, therefore, no need to provide separate models for the database types and application types as would have been necessary if we were, for example, using a domain model to a relational data model.

The domain model is essentially a meta-model that realises each of the ODMG concepts as instantiable classes. As such, the domain model contains constructs for describing various ODMG model constructs, such as object classes, their attributes and operations.

For example, the domain model includes a class `dm_Operation` that captures the ODMG specification for operations, including their relationships with multiple `Parameters`, a result `Type` and a list of `Exceptions`.

The principal role of the domain model is to represent underlying application functionality (i.e. database classes and their operations) that are utilised by the interface, as well as providing support for interaction with databases, through support for constructs such as queries and transactions. The Teallach domain model can therefore be seen to represent a particular view of the underlying application as required, and not the entire application as in the application models of systems such as UIDE (Foley and Sukaviriya, 1995) and Humaniod (Luo et al., 1993).

In addition, the domain model represents *auxiliary* data types that may be required to describe transient data vital to the runtime operation of the application and interface. Examples of auxiliary domain information include class libraries that may be needed to manipulate the data input to the system via the user interface before it is used directly in the underlying application. These facilities are required for the runtime operation of the user interface, but are not an inherent part of the application itself. Auxiliary data is also modelled using domain model constructs so that the representation of domain components is independent of their source and persistence.

Fig. 3 shows the domain model tool, by which access is obtained to following four different kinds of information, all described using the ODMG model. These are namely:

Persistent data types stored in an ODMG compliant database. The domain model automatically generates descriptions of these types from the information contained in the database schema through its knowledge of the ODMG metamodel.

Generic database functionality such as querying and transaction management. Since Teallach is specifically targeted at object database applications, the domain model provides facilities to create instances of database query, transaction and connection objects. In the ODMG model, all queries must be associated with a named (and potentially nested) transaction, which itself must be associated with a connection to a particular database. Since several databases could be accessed by a single application, the domain model provides facilities to ensure that designers associate queries with the correct transaction and connection. Fig. 3b shows how a factory metaphor is used to create database-related objects. The database connection factory produces connection objects, which automatically have a subfactory to produce transaction objects, etc.

Auxiliary definitions that have been implemented for use with the application. Such classes are not defined in the database schema and instances of these types will therefore not persist. These types provide arbitrary functionality used within the interface design. For example, an instance of a post-code (zip-code) checker class could be used by an interface to automatically generate address information.

Class libraries supplied as part of the Java API. There are many classes within the ever-expanding Java API that provide functionality that can be exploited within an interface design. For example, the `Integer` class can be used to validate numeric data, and the `String` class can be used to compute substrings of entered data.

The Teallach domain model has been designed to be usable with different object database systems that conform to the ODMG standard, or to allow non ODMG compliant

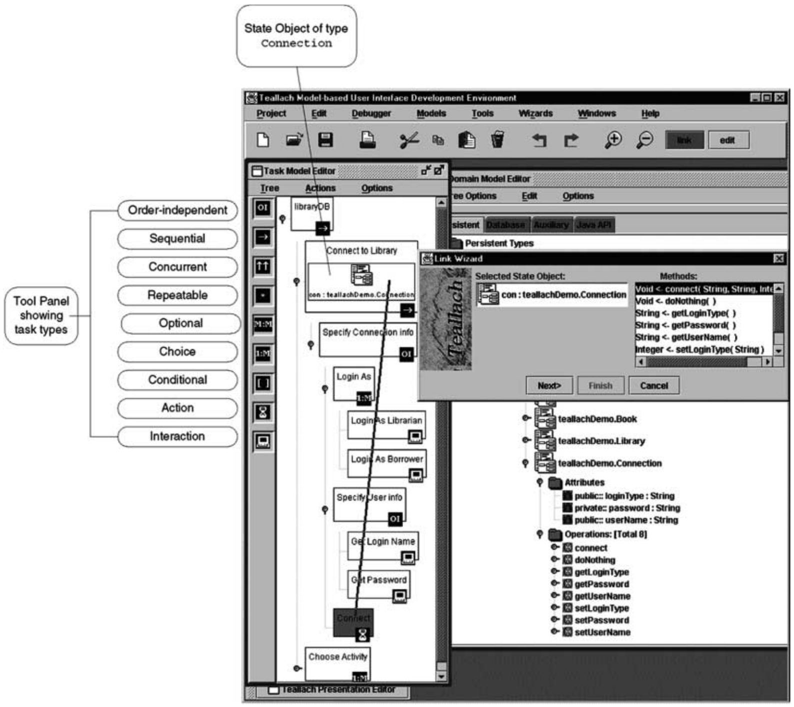


Fig. 4. Task describing connection to library using Teallach tool.

systems to be viewed in terms of ODMG constructs. The Teallach system is currently used in conjunction with the Poet object database (Poet, 2000).

3.2. The task model

Most MB-UIDEs include at least one model for describing what a user can do with an interface. In practice, there seem to be two general schools of thought about what a task model is and where it fits into the model-based development life cycle. One school characterises task modelling as an early, goal oriented, design activity that is subsequently refined in the process of producing an executable prototype (Markopoulos et al., 1992). The other school describes tasks in terms of "user actions including both motor steps and mental steps" (Szekely, 1995), thus focusing on a much lower level dialog between human and computer. The Teallach task model leans towards the latter of these, since its constructs can be uniformly used to specify both high- and lower level task details.

Some model-based systems (i.e. TADEUS (Schlungbaum and Elwert, 1996)), include distinct models for describing user tasks and lower level, interface-specific, dialog. Indeed, an early prototype of Teallach included distinct task and dialog models. However, the dialog model was subsequently dropped because much of the functionality it supported seemed to replicate that provided by existing mechanisms of modern widget sets. Calls to activate this functionality can be accommodated within the task model through its state object mechanism as described in Section 3.2.3.

A key feature of the Teallach task model (Griffiths et al., 1999b,1998b) is that it provides support for modelling both the structure of user tasks and the flow of information between its various models when carrying out the user's tasks. As a result, the task model can be seen as having an important role in describing the relationships between the Teallach models.

3.2.1. Basic task model structure

The task model shares its basic structure with the task models of several MB-UIDEs, such as Adept (Johnson et al., 1995), Mastermind (Szekely et al., 1996) and TADEUS (Elwert and Schlungbaum, 1995). Indeed, it is these notations, together with the results of prototyping several user task models, that provided the stimulus for the current model and its notation. An example of a task model in the Teallach tool is given in Fig. 4. The task model is a goal-oriented task hierarchy, with its leaf nodes (termed *primitive* tasks) representing interaction or action tasks, as described in Section 3.2.2. The temporal relationship between sibling tasks is specified by their parent task. A non-leaf task is known as a *composite* task. The task model provides seven temporal relations for characterising composite tasks, namely:

Sequential. The task's subtasks must be performed in the specified order; all subtasks must be completed before the task's goal is considered achieved.
Order-independent. The task's subtasks may be performed in any order, but all subtasks must be completed before the task's goal is considered achieved.
Optional. Zero or more (including all) of the task's subtasks may be chosen. All chosen subtasks must be completed before the task's goal is considered achieved.
Repeatable. The task's sub-tasks are repeated a specified number of times, or until a condition is satisfied.
Concurrent. The task's subtasks are interleaved; all subtasks must be completed before the task's goal is considered achieved.
Choice. The user must decide which one of the task's sub-tasks is to be performed; the chosen subtask must be completed before the task's goal is considered achieved.
Conditional. There exists a choice between task's sub-tasks that is dependent on a specified condition.

All tasks have the option of being cancellable. Any conditions specified on conditional or repeatable tasks are described using UML's object constraint language (OCL) (Warmer and Kleppe, 1998). OCL is a formal language that is primarily used to express side effect-free constraints on expressions within UML models. In the task model, OCL expressions can optionally reference the properties of state objects (see Section 3.2.3) used within the task model. OCL expressions can be parsed for correctness, and can be compiled out into conditions incorporated into the executable application code.

3.2.2. Primitive task types

The lowest level building blocks of the task model are its primitive tasks, which can be categorised as belonging to one of two categories:

- An *action task*, which corresponds to some low-level activity carried out by the application or user. An action task is therefore optionally associated with a domain model operation. For example, a domain model operation might update the database to reflect the fact that a book is now on loan to a particular borrower. Action tasks may be automatic (are performed behind the scenes within the interface) or manual (require user interaction). For example, an automatic action task could set the date field of a form to the current date whenever the form is opened, whereas a manual action task would be the submitting of a form when completed.
- An *interaction task*, which involves some form of interaction with the user. An interaction task is associated with a presentation model component. For example, an interaction task might obtain a value for a password from a user.

3.2.3. State information and information parameters

A shortcoming of many task models lies in their inability to represent the information that is created, retrieved and passed during the enactment of a task. To overcome this limitation, the Teallach task model allows the passing of information into and out of tasks and the declaration of state associated with tasks.

State information can be declared in any non-primitive task, with the associated task providing its scope — a state object is visible to all direct and indirect subtasks of the composite task in which it is declared. The designer assigns a local name to each state object to assist in the readability of the model. The data type and the properties (attributes) and invokable operations of a state object are automatically ascertained and described using functionality and constructs provided by the domain model. The combination of local name and scope is used to uniquely identify each state object — this information can optionally be supplemented with the state object's data type. A state object can be used, for example, to store data from a database class, an auxiliary class or a programming language API class, reflecting the domain model categories presented in Section 3.2. State objects can also be used to represent instances of presentation model interaction objects, thus providing the ability to update the state of a presentation object, or call one of its operations. An example of a state object is given in Fig. 4, where the *Connect to Library task* has a state object con of type `teallachDemo.Connection`. Information flow is declared by specifying the input and output parameters of a task using state objects using the wizard mechanism described fully in Section 7. Thus, the state of each task is realised by that of its associated state objects (in terms of each state object's properties). In general, the information outputs of a task are linked to state object(s) declared in higher-level tasks. Section 4.1 provides a more detailed discussion of state objects and their use in the Teallach models.

Given the emphasis in Teallach on the development of interfaces to databases, the task model requires a means of creating and invoking database-specific functionality such as sessions, transactions and queries. Since the domain model provides a description of these concepts as ODMG classes, the task model can create state objects that correspond to database sessions, transactions and queries; the required functionality can be invoked through calls to operations on these state objects. An example of the invocation of a query from a task model is provided in Section 6. In addition, several tasks may reference the same state object, manipulating its properties and invoking its operations. For example, a state object representing a database query may require information gathered by several tasks regarding its query parameters and will finally be executed by an action task.

The ODMG model provides facilities to describe the exceptions that an operation can throw at runtime. Since state objects can uniformly represent both domain and presentation artifacts, the task model includes facilities to indicate how the enactment of a task is influenced by such exceptions. For example, Fig. 11 shows that the Search action task is capable of throwing an exception (indicated by the red dot next to the task). The designer has dragged a thread from this dot towards the Search for a Book task to indicate that this task *catches* any exceptions thrown by the Search task. It should be noted that the task that catches the exception must be a parent of the task that throws the exception — this constraint is enforced by the task model tool. An example of an exception thrown by a presentation model object could be a post code checking widget, that throws an exception if the entered post code does not exist. The task model also provides a simple facility to specify what text should be used to inform the user of the generated interface of what has happened if an exception is thrown.

Although several model-based systems (e.g. (Markopoulos et al., 1992; Puerta and Maulsby, 1997)) provide facilities to specify what information is used by a particular task, they do not provide the ability to specify how this information is utilised in the resultant interface. Vanderdonckt (Vanderdonckt et al., 1998) details how data flow techniques can be introduced into user-task modelling. He uses function chaining graphs (FCG) (Petoud and Pigneur, 1989) to specify both the input/output and domain functionality (operations) requirements of each task in terms of domain entities found in an entity-relationship data model. The characteristics of the information conveyed in a FCG is comparable to that captured by Teallach's state objects (i.e. each item of information is characterised by its data type, domain definition, identification, descriptive label and interaction direction (input/output)), although Teallach's state objects can also represent presentation artifacts, whereas it is not clear whether this is so with FCGs. The validation mechanisms used by Vanderdonckt (i.e. consistency, completeness, and reachability) are also to be found in Teallach through the facilities provided by its Wizard mechanism, although the Teallach wizard also type check the input/output parameters of its operations. The notation provided by FCGs, however, provides a more explicit representation of data use within a design than that provided by Teallach. Vanderdonckt recognises that his formalism can result in very large models with an associated usability problem. This problem can also be said of the Teallach models, and techniques to manage the presentation of such information still need to be investigated.

3.2.4. Comparison with other notations

Teallach's basic task modelling notation can be seen to be an extension of those proposed by several other model-based systems (e.g. Johnson et al., 1995; Szekely et al., 1996; Elwert and Schlungbaum, 1995). The ConcurTaskTrees task modelling notation (Paternóet al., 1997, 1999) however deserves special comparison due to the richness of its operators. In common with Teallach, ConcurTaskTrees represents a task model as a hierarchical de-composition of a task into subtasks. However, rather than have its temporal relationships expressed by a super task (as is the case with Teallach and several other notations), ConcurTaskTrees links related subtasks (at the same tree depth) with one from a set of ten operators: namely, inter-leaving, synchronisation, enabling, enabling with information passing, choice, disabling/deactivation, disabling/deactivation with information passing, iteration, finite operation, and optional tasks. The implications of this notational difference is that to realise the same model as that of ConcutTaskTrees, Teallach may need to create a task hierarchy of a greater depth, utilising extra structural nodes to realise the required semantics. Many of the ConcurTaskTrees notation's operators are immediately comparable to those offered by Teallach, however the following differences exist:

- ConcurTaskTrees provides special versions of its operators that either pass or do not pass information, whereas Teallach uses the separate state information mechanism for this purpose. Teallach's close links to its domain and presentation models however allows its task model to explicitly reference the attributes and operations of domain and interface functionality.
- ConcurTaskTrees differentiates between four task types: interaction, application, user, and abstract. In the Teallach task model, these are realised by interaction and action tasks, with the absence of a link from an action task to a domain object inferring a user action task. The ConcurTaskTrees notion makes such task types explicit.
- The semantics of the ConcurTaskTrees disabling/deactivation task types is handled in Teallach by its exception mechanism. While the ConcurTaskTrees notation allows for an abstract specification of disabling/deactivation conditions, Teallach makes use of the actual exceptions that can be thrown by action or interaction (domain) tasks.
- ConcurTaskTrees supports two types of concurrency, namely synchronised (i.e. utilize synchronization points) and interleaved (concurrent with no synchronisation points), whereas Teallach only supports interleaved tasks.

These differences give rise to different representational abilities between the different notations, and therefore an associated difference in the properties of the corresponding interfaces. The use of the ConcurTaskTrees notation within a model-based system to generate dialog and presentation models has been investigated by (Limbourg et al., 2000) with encouraging results arising from the extra semantic possibilities that the ConcurTaskTrees notation gives rise to for interface generation.

3.3. The presentation model

Teallach's presentation model (Gray et al., 1998) is used to describe the appearance and surface behaviour of the generated user interface. It does not specify the dynamics and information processing capabilities of this interface — this is the responsibility of the corresponding task model. There are two levels of abstraction available in the presentation model, namely abstract and concrete. The abstract layer describes an interface in terms of a number of abstract categories, each of which can be implemented using any of a number of interaction objects from the widget set that constitutes the presentation models concrete layer. These abstractions were initially devised by examining the common features of several existing widget sets — primarily those that are provided by Java's Swing widget set.

In some other MB-UIDEs, the presentation model is simply the set of widgets provided by the implementation toolkit; some systems give the designer assistance in selecting an appropriate widget (e.g. Vanderdonckt, 1999). Systems such as Mastermind (Szekely et al., 1992) allow customisation of proto typical widgets. Other systems, such as TADEUS (Elwert and Schlungbaum, 1995), TRIDENT (Vanderdonckt and Bodart, 1993; Bodart et al., 1995a), and ACE (Johnson et al., 1993), have both abstract and concrete presentation models, with ACE's notion of *selectors* as an abstraction mechanism most closely resembling our own. In these systems (and also in Teallach), the entities in the abstract model represent a stage of refinement of the interface, allowing developers to postpone decisions on the actual widgets to be used until later in the design process. It is of note however that Teallach does not preclude designers from freely intermixing both abstract and actual widgets within the same presentation model, as both are first-class objects.

3.3.1. Concrete presentation model layer

The concrete presentation model (CPM) consists of the user-interface components (widgets or interactors) that are used to construct the user interface. Such components are termed *concrete interaction objects* (CIOs). Typical examples of CIOs are buttons, sliders, and menus. In Teallach, the CPM essentially consists of Java's Swing widget set (Topley, 1999). However, in addition to the components provided by Swing, designers can use additional CIOs either created by themselves or by a third party. This enables the use of specialised components associated with the application, for example an interface for use by biologists may require a molecule-viewer widget.

To be usable within Teallach, user-supplied widgets must adhere to the call interface specified by the JavaBeans API. JavaBeans is Java's component model, which defines (amongst other things) a set of conventions to be followed by re-usable components, so that they may be integrated easily into existing systems. This does not mean that all user-supplied widgets used within Teallach must be JavaBeans, or even that they must be written in Java; however, to use a non-JavaBean component, a JavaBean wrapper must be written for it. Writing this wrapper is the responsibility of the developer importing the component into Teallach. Once imported, the new component can be used in the same way as the pre-existing ones.

3.3.2. Abstract presentation model

The abstract presentation model (APM) defines high-level categories that can be used to characterise user interactions without committing designers to specific CIOs. The APM defines a number of categories that are based on the identification of different roles that components may play within a user interface.

The following are the fundamental categories of the APM:

FreeContainer. A FreeContainer can be thought of as a 'top-level window'; strictly, it is a container for other presentation items, that is not itself contained by anything. Typical examples are windows and dialog boxes.

Container. In contrast to a *FreeContainer*, a container represents all containers that are not 'top-level' windows. Containers are typically used to act as a grouping mechanism with associated layout constraints for related interaction objects.

Inputter. An Inputter is used to transfer data from the end-user to the application. A typical example is a text-field, into which the user can type alphanumeric data such as names or salaries.

Display. A Display is used by the application to present data to the user. A typical example is a (read-only) table, which can be used to present tabular data such as employee records to the end-user.

Editor. An Editor allows the end-user to alter existing data; it can be thought of as a combination of Display and Inputter, which first displays an existing data value, and then inputs to the system any alterations made to it. A typical example is an *updatable* table.

Chooser. A Chooser permits the end-user to indicate a constrained choice, by offering several alternatives for selection. A typical example is a pick-list. The Chooser can be thought of as a special kind of `Inputter`, but its usage is frequent and sufficiently distinctive to merit a separate category.

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

Each APM category is characterised by a set of operations, a set of *realisations*, a default realisation and a global customisation, as described below. The term *realisation* is used to describe the representation of an abstract category by a CIO.

3.3.3. Registering and using CIOs within abstract categories

The operations specified by an abstract category are intended to describe, in an abstract way, the intended functionality of the category. Although these operations are specifications rather than executable code, they are described within the Teallach environment in the same way as domain model operations. Each CIO has a set of operations that allow users can to invoke its functionality. When a new CIO is registered with the presentation model, it is the responsibility of the designer to state which of the abstract categories the CIO can be used to realise. As part of this registration process, the designer must specify which of the CIO's operations correspond to the APM's required call interface. For example, the *Display* abstract category defines an operation `setValue(Any input)` that takes as input an object representing the data to be displayed in the generated interface (this object is of undetermined type, since the abstract category may be used for the display of any kind of data). If a molecule-viewer widget is to be registered with the *Display* abstract category, then during the registration process the designer must specify which of the molecular viewer's operations should be used to realise the `setValue` operation. It is these CIO operations that will be used within the executable code.

Teallach comes with a standard set of realisations called the *Default Style*, which specifies which widely used concrete widgets can be used to represent the standard abstract categories. This set can be extended, either by using existing widgets in new ways (e.g. registering a button as an Inputter), or by importing new widgets into the system. It should be noted that the relationship between abstract categories and widgets is many-to-many, i.e. an abstract category will usually contain many different widgets that

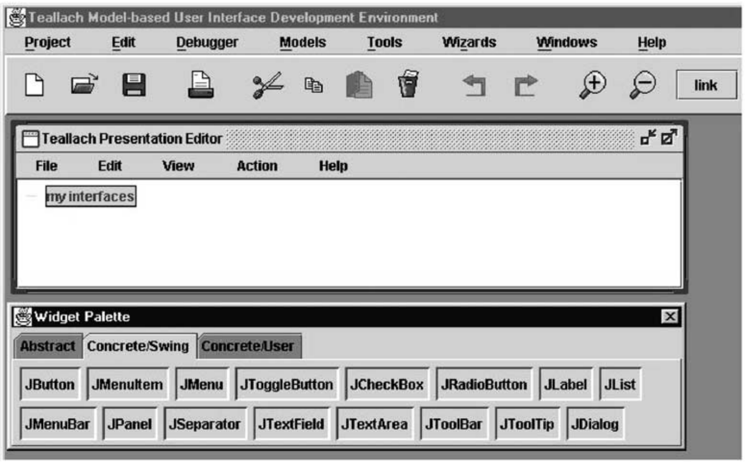

Teallach Model-based User Interface Development Environment
Project Edit Debugger Models Tools Wizards Windows Help
link
Teallach Presentation Editor
File Edit View Action Help
my interfaces
Widget Palette
Abstract Concrete/Swing Concrete/User
JButton JMenuItem JMenu JToggleButton JCheckBox JRadioButton JLabel JList
JMenuBar JPanel JSeparator JTextField JTextArea JToolBar JToolTip JDialog

Fig. 5. The presentation model tool.

can be used to represent it, and any given widget may realise several abstract categories. For example, the *Input* abstract category can be represented either by Swing's `JTextField` component if the user is to enter a single line of text, by a `JTextArea` component if the user is to enter free-format text, or by a `JPasswordField` component if the user is to enter text that must not be echoed. If the `JTextField` is used, then the Inputter's `getValue`() operation will be realised by `JTextField`'s (concrete) method `getText`(); however, if the *Inputter* is represented by a `JPasswordField`, then the

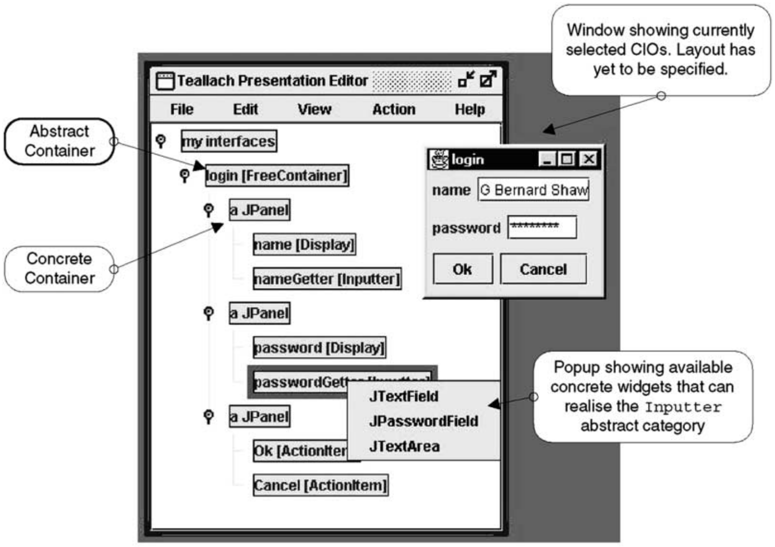


Fig. 6. Construction of a presentation.

same operation should be represented by `JPasswordField`'s (concrete) `getPassword()` method.

A *customisation* indicates how a particular widget or category of widget is to be adapted for use. A global customisation can form part of the definition of any category. For example, it is possible to imagine a 'house style' in which all utilised widgets have a red background. In this case, the definition of Inputter would have a global customisation *backgroundColour* → *red*. It is also possible to apply customisations to individual realisations, for example, so that a specific instance of a *JTextField* could be indicated to use Courier font.

3.3.4. The presentation model in use

This section illustrates the use of Teallach's two-level presentation model for describing a simple login window. Fig. 5 shows two components of the presentation model tool, the *Presentation Editor* and the *Widget Palette*, before construction of the login window has begun. In the presentation editor, the node *my interfaces* represents the fixed root of all presentation models. The widget palette has its *Concrete/Swing* tab selected, allowing the designer to view Swing components available for insertion.

Fig. 6 shows how the login window has been constructed within the presentation editor. The presentation is represented as a hierarchy of components, some of which are abstract and some of which are concrete. For example, a *FreeContainer* (top-level window) has been selected from the widget palette's *Abstract* tab, to represent the login window. The designer can either leave the realisation of this abstract component unspecified, in which case the default will be used, or override the default with one from an offered list of registered realisations. As an example of user selection of a realisation for an abstract category, Fig. 6 shows the offered list of realisations for the password Inputter. The presentation model tool also includes a preview facility that allows the designer to see the concrete representation of the presentation under construction.

4. Interaction between models

In Teallach, a user interface design consists of a set of models and of associations (mappings) between the models (Griffiths et al., 1999a). It has been noted by (Puerta and Eisenstein, 1999) that one of the main challenges faced by model-based systems is how to associate (or in their terms *map*) elements from one model with their related counterpart(s) in other models — henceforth called *the mapping problem*. Early model-based approaches simply hard-wired such mappings into their systems, resulting in interfaces whose characteristics (dialog and presentation) are difficult (if not impossible) to extend or customise. More recent model-based systems (e.g. (Elnaffar and Graham, 1999; Stirewalt, 1999; Eisenstein and Puerta, 1998; Brown et al., 1998), and experimental results in (Limbourg et al., 2000)) have recognised the importance of providing generalised mechanisms for specifying mappings between their models, and therefore benefit from the improved functionality and flexibility such mechanisms provide. A further related issue is the degree to which such mappings can be automatically generated. (Puerta and Eisenstein, 1999) states that there are an intangible number of factors that can affect the choice of mapping, and that the ideal approach to mapping automation " would be one that supported the choices of rational interface design as well as the creativity of artistic interface design" (Puerta and Eisenstein, 1999). The design method used by a model-based system is also intrinsically linked to the system's ability to provide a flexible mapping mechanism. This section will show how Teallach's flexible design method is supported by the possible mappings between its models.

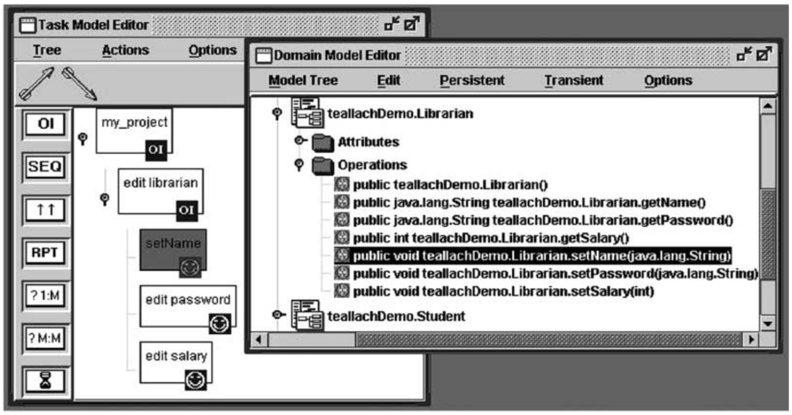
Since Teallach's models must contain sufficient information to allow executable interfaces to be generated from them, it is essential that the relationships between the models can be defined precisely. This section reviews the ways that the three Teallach models can interact. There are two ways of associating elements from different models, namely *linking* and *deriving*. In *linking*, an association is made between existing components in two models. In *deriving*, components in one model are constructed based on components in another; the derive mode also creates links between the newly created component and the source of the derivation. In the modelling environment, linking and deriving are different modelling tasks, expressed using different interaction techniques. The following subsections outline the ways in which components from the different models can be linked and derived. A comparison of Teallach's approach to the mapping problem and those forwarded by other model-based systems concludes this section.

4.1. From domain to task

As stated in Section 3.2.2, a primitive task that invokes a domain model operation is known as an action task. Linking is used to specify which operation is to be invoked on which state object within the task model. This process involves the following steps:

1. Identifying the state object on which the operation is to be invoked.
2. Identifying the operation to be invoked.
3. Specifying the input data to the operation (if any).
4. Optionally, indicating where the result of the operation (if any) is to be stored.

The process of creating a link between an action task and a domain model operation is illustrated in Fig. 4. Here, the designer has created a link between the *Connect* action task and the *con* state object. This initial linking step has identified the state object on which the operation is to be invoked. The process of specifying the remaining information listed above is assisted by the Link Wizard, which is also shown in Fig. 4. The Link Wizard presents the designer with a list of operations that are invokable on the state object, and the designer chooses one. In the case of Fig. 4, the connect operation has been chosen. If the chosen operation requires any input data (through its operands), then the Link Wizard will prompt the designer for where this information is to come from. In the case of the connect operation, this requires three items of data: two strings and an integer. The Link Wizard will list all possible sources of this data (i.e. other state objects or attributes of state objects), and will ensure that only data sources with the correct type and scope are identified. For example, it would be incorrect to suggest a state object contained within the *Choose Activity* task shown in Fig. 4 as, due to the sequential nature of these tasks, any such state object will only be available once the *Connect* task has been completed. If the



```
public teallachDemo.Librarian()
public java.lang.String teallachDemo.Librarian.getName()
public java.lang.String teallachDemo.Librarian.getPassword()
public int teallachDemo.Librarian.getSalary()
public void teallachDemo.Librarian.setName(java.lang.String)
public void teallachDemo.Librarian.setPassword(java.lang.String)
public void teallachDemo.Librarian.setSalary(int)
```

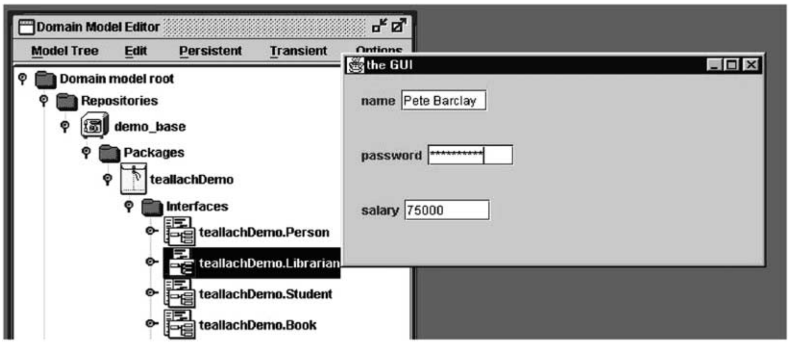
Fig. 7. Task hierarchy created from a domain class.

operation returns a value, the Link Wizard prompts the designer for which state object should be used to store this value. Once again only suitably scoped and typed state objects, or their attributes are suggested. If a designer attempts to link any components, other than those illustrated above (e.g. a non-leaf task and a state object), then the Link Wizard reports this as an error.

Using a domain class as the source of a derive operation with a non-leaf node of the task model as its target creates a new task structure, rooted in a new sub-task *T*. *T* will also contain an automatically generated state object that represents an instance of the domain class. By performing such an operation, the designer is in effect stating that *T*s purpose is to allow editing of the attributes and invocation of the operations on the newly created state object. *T*s subtasks will therefore be realised by interaction and action tasks whose purpose is to edit the attributes, and invoke the operations, of the newly created state object. The process of deriving automatically creates links between related components, asking the designer for information when necessary (i.e. when input data is required in the invocation of an operation). Fig. 7 shows part of the task hierarchy generated by using the *Librarian* class from the domain model to derive a new sub-task (called *edit librarian*) in the task model.

4.2. From presentation to task

Linking a node of the presentation model to a node of the task model specifies that an *instance* of the presentation object is to be used to realise the task in the generated interface. An example could be the linking of a pre-built, custom editor for *Librarian* instances to the task *edit librarian*. In this case, both elements are considered as leaves, as we do not wish to view them at a finer granularity. The designer will therefore create a state object that corresponds to an instance of the presentation model component in the task model and link it using the Link Wizard. Thus both domain and presentation model objects can exist as state in the task model. This means that the same facilities can be used to pass



name Pete Barclay

password **********

salary 75000

Fig. 8. A basic editor for Librarian generated from the domain model.

information to presentation model components as to domain model operations, and that operations can be invoked on presentation model components from within the task model.

Deriving a non-leaf node in the task model from the information contained in a presentation model node generates a new task structure that can be used to describe the dynamics of the presentation object, and creates links between the derived task structure and the original presentation structure. Since the task model cannot ascertain what the dynamics of a presentation are without assistance from the designer, it simply creates an order-independent composite task for each corresponding *FreeContainer* or *Container*. Once the derived task structure has been created, the designer should then specify the required interface dynamics, and data flow, where necessary. For example, using the librarian editor shown in Fig. 8 as the source of a derive operation with the root of the task model as the target would generate the task structure shown in Fig. 7.

4.3. From task or domain to presentation

Linking a leaf of the task model to a leaf of the presentation model has the same effect as creating a link in the opposite direction, as described in Section 4.2.

A task model node can be used to derive the structure of a presentation model node whose purpose is to create a default presentation for the task sub hierarchy. For example, the task model from Fig. 7 can be used to generate the presentation shown in Fig. 8. Such derivation operations apply a set of simple recursive mapping rules to decide which interaction objects should be generated in the presentation as follows:

1. If the task node is a non-leaf task, then generate a FreeContainer abstract interaction object.
2. If the task node is an interaction task, then generate an Editor abstract interaction object.
3. If the task node is an action task, then generate an ActionItem abstract interaction object.

Since the task model does not have knowledge about presentation issues, it delegates choosing the actual interaction objects to the presentation model by mapping to abstract interaction objects. It is the responsibility of the presentation model to decide which concrete interaction objects are chosen to realise the chosen abstract interaction objects according to the particular style that is in use.

Deriving a node in the presentation model from the information contained in a domain model class (e.g. *Librarian*) creates a basic default editor for this class. This editor is constructed by examining the get- and set- methods of the class, and inserting appropriate display or edit fields in the generated presentation object. Fig. 8 shows the result of deriving a presentation for the *Librarian* class. Of course, the user may wish to treat this generated object only as a starting point, and may subsequently modify it.

Teallach maintains information about links between associated components in its three models. It has been noted by (Puerta et al., 1994a) that the problem of maintaining consistency between models is a complex one. Teallach provides a simple mechanism to inform designers if they are about to delete a linked model component. If they choose to proceed then it is the responsibility to re-establish links when modifications are complete. If, however, they are only modifying a component (e.g. if they are changing the abstract category of an AIO), then Teallach prompts the designer to re-establish any links to any referenced operations.

4.4. Comparison of mapping approaches

This section provides a comparison of Teallach's approach to the mapping problem and those forwarded by other model-based systems. MOBI-D (Puerta and Eisenstein, 1999) provides an editor to allow designers to set mappings by dragging an element of any of its models onto the intended target element. For example, dropping a particular domain object onto a user task indicates that the task will use the domain object in performing the target task. The information contained in the set of mappings is utilised by MOBI-D's decision support tool called TIMM (Eisenstein and Puerta, 1998). TIMM provides expert guidance on the possible mappings between model components by consulting a knowledge base of interface design guidelines. For example, TIMM can suggest a set of possible interactors to use for a particular domain object based on the type of the domain object and the known characteristics of the particular interactor. This approach is comparable to that employed by Teallach, in that Teallach also allows designers to associate components from its models with each other. However, in contrast with Teallach, MOBI-D does not provide facilities to derive the structure of one model from the information already contained in another. TIMM's ability to utilise knowledge about its interactors to provide mapping suggestions is a comparable technique to that used by Teallach, with its notion of abstract interaction object categories, although TIMM's mapping rules are significantly more customizable than those of Teallach. It can be seen in Fig. 6 that Teallach's presentation model suggests the possible interactors for the *Input* selected abstract category. As previously stated, the set of CIOs for each of these abstract categories is updatable by the designer. It is not clear from the current literature how MOBI-D exploits domain information associated with a particular task. Teallach on the other hand utilises such information through its state objects to fully specify the information processing in the generated interface. It is also not clear how MOBI-D exploits its models to generate an executable interface.

MDL (Stirewalt, 1999) is a language for binding the user interface models supported by the MASTERMIND (Szekely et al., 1996) model-based system. MDL allows task hierarchies, domain actions and presentations to be constructed, and then linked (bound) together. In terms of the work reported in this paper, MDL allows designers to associate components at different levels of granularity. For example, either a user task can be associated with a complete interface (coarse-grain binding), or alternatively individual parts of a presentation can be associated with a subtask (fine-grain binding). This facility is similar to that forwarded by Teallach, in that a completed presentation model component or imported JavaBean can be associated with a leaf task (coarse-grain binding), or alternatively each task's subtask can be associated with individual interaction objects (abstract or concrete) in a presentation (fine-grain binding).

Vista (Brown et al., 1998) is a prototype tool for examining the links between design artifacts (models), focusing on how such links support co-evolutionary design between groups of designers with differing specialities. Vista's developers argue that while HCI designers may be concerned with an analysis of an application's task specification (i.e. the task model), software engineers may have more interest in lower-level implementation issues (i.e. the domain model). Vista allows developers to associate components from one model with those from another in a similar manner to Teallach. Thus, for example, an operation identified in their domain model can be associated with a user task in their task model, using a hyperlink metaphor to show related model components. Vista's developers recognise that "the links between design representations are complex, due to their differing points of view and differing levels of detail" (Brown et al., 1998). The Wizard mechanism utilised by Teallach to assist designers is one means of simplifying this process. However, since Vista's focus is on collaborative design, its developers focus on mechanisms for aiding communication between designers. To this end, their hyperlink mechanism shows how parts of one design artifact are reflected in and related to others. Although Teallach can facilitate similar development strategies, we do not exploit this facet of the mapping process.

While the above approaches have all been concerned with linking models, Limbourg and his colleagues (Limbourg et al., 2000) have investigated how one model can be derived from another. In particular, they investigated how presentation and dialog models can be systematically generated from a task model that utilises the the rich modelling facilities of the ConcurTaskTrees notation (Paternó o et al., 1997). This derivation results in a decision tree that presents designers with a set of design alternatives based on the attributes and properties of the generated dialog. In contrast tothis approach, Teallach's derivation rules are hard-wired into the system, and therefore there is no comparable notion to the decision trees of (Limbourg et al., 2000). Since Teallach's notion of dialog is captured by its enhanced task model, the requirements for complex derivation rules to generate a corresponding presentation are reduced. In particular, Teallach's use of state (whose properties are fully known and exploitable through its domain model) to represent information flow within the application, results in a much simplified presentation model. While the derivation of presentation/dialog models from task information provides considerable challenges, Teallach has additionally shown how domain $\rightarrow$ task, domain $\rightarrow$ presentation, and presentation $\rightarrow$ task derivations can also be exploited.

5. The Teallach design method and tools

The model based approach to user interface development will only be successful in practice if effective tools are provided to facilitate the creation and association of models, and if the tools that support design methods are consistent with the preferences of designers. Early in the Teallach project, we interviewed a number of interface developers in commercial organisations, and it became evident that different people and organisations had very different approaches to database interface development. For example, one approach was based principally on the mapping of domain model constructs onto screen designs, while another approach involved mocking up screen designs, and filling in the functionality of the interface later. The variety of approaches adopted in industry provided motivation for a flexible development method, and thus tools that could support different ways of working.

It is clear, however, that model-based systems are by no means always associated with flexible design methods. Indeed, many proposals support a single fixed method for developing their component models, frequently stipulating that the task model must be constructed before any other model (Griffiths et al., 1998a). For example, TADEUS (Elwert and Schlungbaum, 1995) provides a systematic approach to the refinement of models, but certainly encourages a particular sequence in the construction of interface designs.

Teallach attempts to circumvent this problem by having minimal constraints on the order in which its three models can be constructed. This flexibility is reflected, among other things, in the lack of restrictions on the order in which links between related model components can be defined. For example, one mode of working could be to create a high-level task model, and then to use the presentation model to articulate each of these high level tasks. Once this process has been completed, the designer must then specify how the presentation model's designs relate to the task model's high level tasks by linking the related components, as described in Section 4. Once the designer is satisfied with their design, they simply press the Generate Code button to start the process of producing executable Java code that corresponds to the semantics of their developed models. If developers wish to change any facets of the generated interface, then they must alter the corresponding models. Although Teallach generates programming code, it is not possible to automatically reflect any changes in such code with corresponding changes in the models.

Teallach provides an integrated toolkit (Barclay et al., 1999) that a designer can use to construct and link the individual models that form the model-based UI design. This toolkit supports Teallach's flexible design method, and in supporting direct manipulation of the models, helps in the prevention and early detection of inconsistencies between models as described in Section 4.3. As shown in Fig. 4, the Teallach toolkit consists of editors for its three models within an overall tool environment providing project management, editing, model linking, and code generation facilities.

As previously discussed, a designer can create links between related model components. In the Teallach tools, this is achieved by switching to *link mode* and drawing a rubber-banded line from an element in the task model to a state object representing the other model construct, to associate the two. This can be seen in Fig. 4, which shows a link being

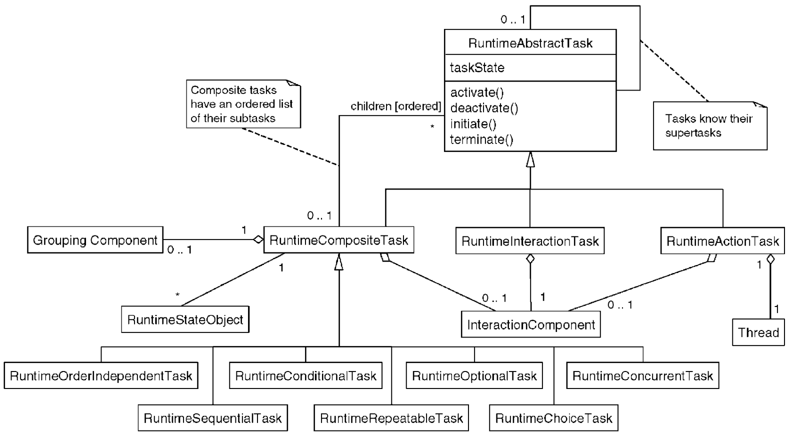


Fig. 9. Class diagram of runtime task type hierarchy.

created between the task model Connect action task and the domain model connect operation defined on the `con: Connect` state object. The Link Wizard guides the designer through this process. Once an operation has been associated with a task, the icon in the task model is updated to re ect this change. In common with Vista (Brown et al., 1998), Teallach currently uses a simple hyperlink metaphor to show associations between linked model components; this allows the designer to jump to an associated component by invoking its *show linked components* operation from a popup menu.

To derive information from one model into another, the designer simply drags a component from one model and drops it at the desired location in the target model (a technique first forwarded by (de Baar et al., 1992)). For example, the designer may construct a partial task hierarchy corresponding to some constructed presentation (or vice versa). Once the new structure has been generated, the relationships between components are maintained through the services provided by the Teallach project store.

6. Running the models

Although many MB-UIDEs 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 are executed directly by an interpreter (Wiecha et al., 1990), whereas in Humanoid (Szekely et al., 1992), TADEUS (Elwert and Schlungbaum, 1995) and FUSE (Lonczewski and Shreifer, 1996) the models are mapped to an alternative representation that is interpreted by a user interface management system (UIMS). There are also a small number of examples where programs are generated in a conventional high-level language, for example, in JANUS (Balzert et al., 1996) and Mastermind (Stirewalt, 1999). The Teallach code generator produces a Java program

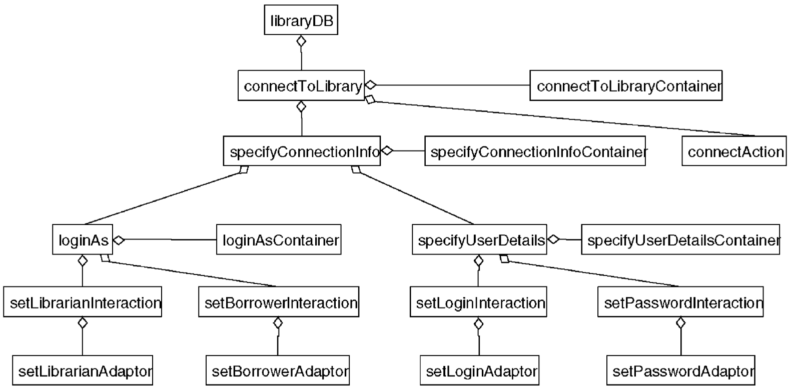


Fig. 10. The generated user interface classes.

that makes use of the model view controller (MVC) design pattern. We have chosen to generate executable code directly to minimise runtime overheads, to maximise portability, and to allow use to be made of existing mainstream widget sets and class libraries. The Teallach code generator is described in (da Silva et al., 2000). The focus in this section is on describing how the Teallach models are represented in the generated code using the MVC pattern.

6.1. Runtime context

The MVC pattern is widely used for the development of user interface programs (Krasner and Pope, 1998). MVC specifies how user interface software can be separated into components, each with a specific function. The *model* components are responsible for handling the state of objects used by the user interface. The *view* components are responsible for the user presentations 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.

The code generator produces as output Java code that exploits the Swing widget set. Swing applications are themselves organised using MVC. In addition to the basic Swing classes, a class library has been developed that provides direct support for most of the functionality provided by the task model. The class library, which is illustrated in Fig. 9 using UML notation (Booch et al., 1999), contains a class hierarchy that mirrors the task types of the Teallach task model.

The tasks of the task model are implemented using subclasses of the class *RuntimeAbstractTask*. More precisely, composite tasks become subclasses of *RuntimeCompositeTask*, action tasks become subclasses of *RuntimeActionTask*, and interaction tasks become subclasses of *RuntimeInteractionTask*. A *RuntimeCompositeTask* can have many subtask classes.

The state information associated with each composite task in the task model are realised by instances of *RuntimeStateObject* classes associated with the corresponding *RuntimeCompositeTask*. As subclasses of *RuntimeAbstractTask*, the task classes implement the operations `activate`(), `deactivate`(), `initiate`() and `terminate`(). Basically, `activate`() enables the interaction of the user with the part of the UI responsible for the activated task. In the opposite way, `deactivate`() disables the interaction. The `terminate`() operation notifies the parent task class that the current task class has finished. The `initiate`() operation returns the UI to the state it had when it was first created.

Still in Fig. 9, *RuntimeActionTask* classes and *RuntimeCompositeTask* classes can be associated with concrete widgets. In this case, the concrete widgets are called *initiators*. Initiators are required to: (1) fire action tasks that are designed to be started on demand; for example, a button that may be associated with the *Connect* action task that initiates the attempt to connect to the database; or (2) fire composite tasks that are subtasks of choice tasks, optional tasks or order independent tasks; for example, a user action should be realised by a widget (i.e. a button) in the container corresponding to the choice task, whose purpose is to activate the optional task.

6.2. The generated code

Fig. 10 presents a class diagram for the user interface code generated for the example application shown in Fig. 4. With the exception of the container (e.g. `specifyConnectionInfoContainer`) and adaptor (e.g. `setLibrarianAdaptor`) classes, the other classes in Fig. 10 are subclasses of *RuntimeAbstractTask* from Fig. 9. For example, the class implementing an action task, namely *connectAction*, is a subclass of *RuntimeActionTask*; the classes implementing interaction tasks (i.e. *setBorrowerInteraction* and *setLoginInteraction*), are subclasses of *RuntimeInteractionTask*; and the classes implementing composite tasks are indirect subclasses of *RuntimeCompositeTask* — for example, *specifyConnectionInfo* is a subclass of *RuntimeOrderIndependentTask*.

6.2.1. The generated interface dynamics

Each generated interface has a single entry point to the generated executable. This starting operation is called `main`. The `main` operation in turn creates and controls instances of classes that represent the subtasks of the `libraryDB` task. Each controlling class calls the `activate`() operation of the class representing its subtask, which in turn calls the `initiate`() operation in the objects representing its direct subtasks. As previously stated, the `activate`() operation enables the task, and the `initiate`() operation initialises the state objects used by the task. When a task is finished, it calls its `deactivate`() operation which in turn calls its *terminate()* operation. The `deactivate`() operation serves to clean up any state information, whereas the `terminate`() operation generates an event to signal its calling task that it has finished. The calling task can then either activate the next subtask (sequential tasks), or can wait for all concurrently executing subtasks to finish. For example, the *libraryDB* class implements the main of the application, which in turn invokes the `activate`() operation of the instance representing the *connectToLibrary* root task. This method in turn invokes the `initiate`() operation in the objects representing the subtasks of *connectToLibrary*, which in turn invoke the `initiate`() operation in the objects representing their subtasks, and so on. As *connectToLibrary* is a *RuntimeSequentialTask* class, it initially only activates the *specifyConnectionInfo* class. When the *specifyConnectionInfo* class finishes normally (without being cancelled), its deactivate() and terminate() operations are invoked. The *terminate()* operation generates a change event that is listened for by the *connectToLibrary* class, so that on detection it can activate its next subtask. In this case, the *connectAction* class is activated. On completion of the *tryConnectAction* class, the *connectToLibrary* class also terminates since it does not have more subtasks.

Where necessary, instances of the *Container* class (e.g. *connectToLibraryContainer*) provide the visualisation for the various *RuntimeCompositeTasks* instantiated in the interface. In terms of MVC, instances of the *Container* class can be seen to act as the view for their *RuntimeCompositeTask*, which acts as the controller of its various subtasks. Each *RuntimeInteractionTask* is associated with a corresponding *Adaptor* class (e.g. the *setLoginInteraction* class has an associated *setLoginAdaptor* class). The adaptor class maintains a reference to both its visualisation (the view) and to the state object (the model) used by the *RuntimeInteractionTask*. The *Adaptor* class functions by listening for changes in the state of either its view or its model, and notifies its controller of any such changes. For example, if the *setLoginInteraction* class is visualised by a text field, then any changes in the state of the text field are propagated to the String state object that had been designated to hold the login name information.

6.3. Extended interaction based on task model semantics

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

Confirmation. Allows the user to indicate that a task has been completed; this is normally depicted using an *OK* button.
Cancellation. Allows the user to exit a task without completing it; this is normally depicted using a *Cancel* button. The semantics of the *Cancel* button depend on the category of the parent task of the composite task class that is been cancelled. For example, if the parent task class is a choice task class, then the parent task class is restarted through the invocation of its activate() operation. In contrast, if the parent task class is a sequential task class, the parent task is finished through the invocation of its deactivate() and terminate() operations.
Change Task. Allows the user to swap from one concurrent task to another; this is normally depicted using a menu or a combo box to be used via a card-stack metaphor. Each of the items referred to by the change task will be labelled by the name of each of the concurrently executing tasks.

7. Teallach in use

Up to this point, the Teallach models have been introduced using examples relating to the login screen of a library application. This section illustrates the construction of an interface that searches for a book in the database using information provided by the user.

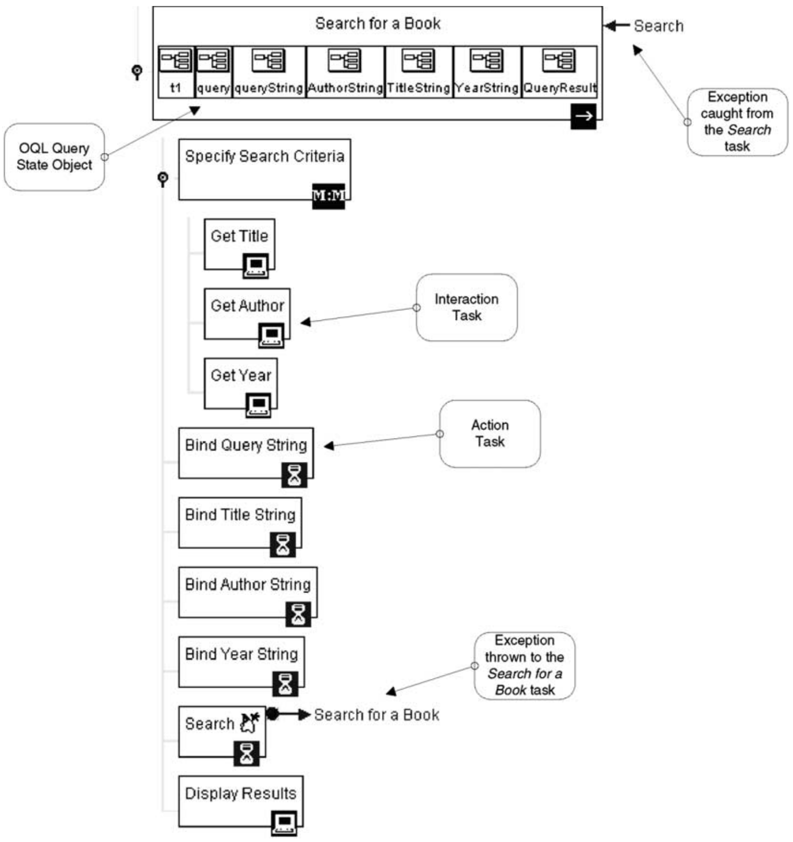


Fig. 11. The task model for searching for a book.

The domain model used when searching for a book is that illustrated in Section 3.2. In particular, Section 3.2 illustrates how the domain model provides a consistent interface to the application classes in the database (e.g. books) and to the facilities provided by the database (e.g. transactions and queries). Section 3.2 shows a single query called `New Query1`, which is to be executed inside a transaction associated with a connection to the my_base object database, which is stored locally and managed by the POET object database management system. As previously shown, Teallach's flexible design method allows the task and presentation models to be developed and inter-related in different orders and using different techniques (Barclay et al., 1999). Here the task model is presented first, in Fig. 11, followed by the associated presentation model.

In addition to specifying the dynamics of the Search for a Book task, the designer has specified the information flow between the various tasks by creating state objects within non-primitive tasks. For example, there is one `OQLQuery` state object and several `String` state objects. These provide temporary storage for the information the user will type into the UI to specify their particular search criteria. More specifically, the

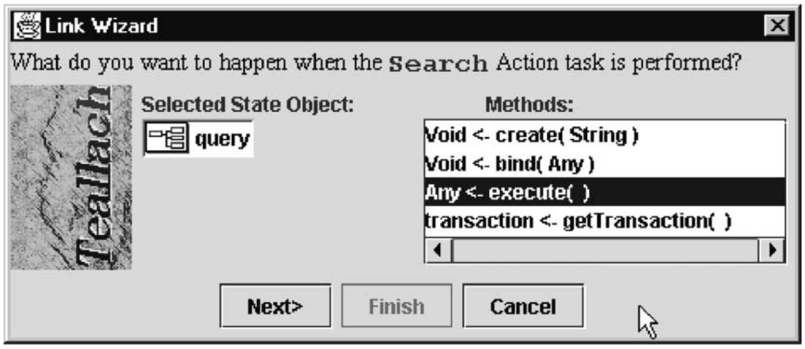


Fig. 12. The link wizard for selecting a method.

AuthorString, TitleString, and YearString state objects provide storage for the information passed into the model's three interaction tasks — Get Title, Get Author, and Get Year by the user of the generated interface.

As well as providing temporary storage for the UI's processed information, state objects allow their behaviour to be invoked. Therefore, in addition to the basic task structure of the Search for a Book task, there are four lower level action tasks that invoke the Bind behaviour of the OQLQuery state object. Thus the book title input by a user of the interface (stored in the TitleString state object), is used as the input parameter of the Bind operation invoked by the Bind Title String action task. It should be noted that the Bind action tasks have been defined to be *automatic*, i.e. they are performed automatically by the generated interface and thus do not require user interaction. It is arguable whether such low-level tasks should be declared within a task model, since they represent the low-level information processing requirements of the interface. Other model-based systems

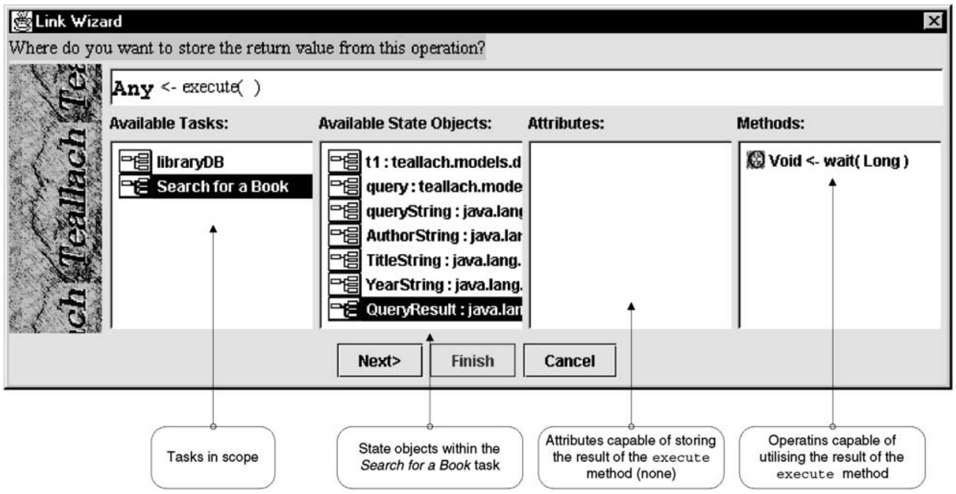
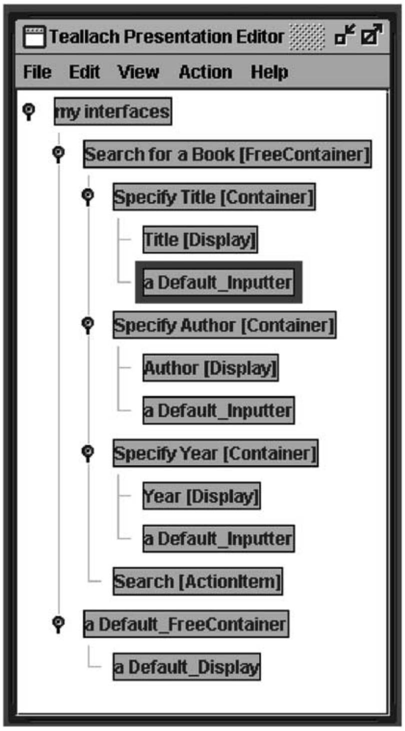


Fig. 13. The link wizard for storing the result of a method.



my interfaces
Search for a Book [FreeContainer]
Specify Title [Container]
Title [Display]
a Default_Inputter
Specify Author [Container]
Author [Display]
a Default_Inputter
Specify Year [Container]
Year [Display]
a Default_Inputter
Search [ActionItem]
a Default_FreeContainer
a Default_Display

Fig. 14. Presentation model for searching for a book.

(e.g. Mastermind (Szekely et al., 1996)), separate such details in their dialog or application models — Teallach's task model, however, combines both task and dialog within the same scope. While this decision may lead to more complex task models, it allows us to view all aspects of the dynamics of the interface within a single environment.

The task model tool uses a wizard to guide the designer through the potentially complex process of invoking a state object's method. Fig. 12 shows the wizard during the process of associating an invocation of the `execute` method on the OQLQuery state object in the `Search` action task. The wizard prompts the designer for the method that is to be invoked on the particular state object, subsequent to which the wizard prompts the designer for the state object that is to be used to store the returned data — in this case the `QueryResult` state object — as shown in Fig. 13. This state object is then used as the input to the `Display Results` interaction task. If the invoked method is capable of raising an exception, then the task model tool will place a red ball icon by the corresponding action task. 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. Fig. 11 shows that the designer has specified that if the Search action task raises an exception at runtime, then control should pass back to the `Search` for a Book task.

A possible presentation model for supporting the tasks in Fig. 11 is given in Fig. 14. As

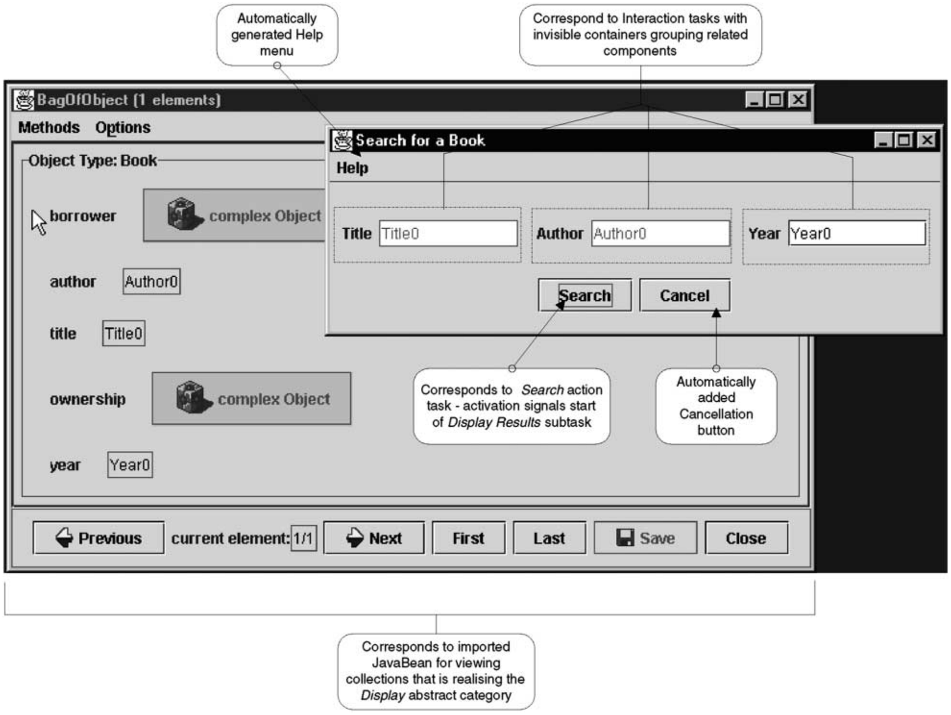

Automatically generated Help menu
Correspond to Interaction tasks with invisible containers grouping related components
BagOfObject (1 elements)
Methods Options
Object Type: Book
borrower
complex Object
author
Author0
title
Title0
ownership
complex Object
year
Year0
Search for a Book
Help
Title
Title0
Author
Author0
Year
Year0
Search
Cancel
Corresponds to Search action task - activation signals start of Display Results subtask
Automatically added Cancellation button
Previous
current element: 1/1
Next
First
Last
Save
Close
Corresponds to imported JavaBean for viewing collections that is realising the Display abstract category

Fig. 15. Generated interface for the Search for a Book task.

described in Section 4, there are different ways in which models can be derived from or linked to each other. The presentation model illustrated in Fig. 14 could either be constructed using the presentation model editor and then linked to the task model, or derived from the task model and then fine-tuned using the presentation model tool. In Fig. 14, the designer has added extra containers around each `Display/Inputter` pair to allow custom grouping widgets to be used. The result of the search is presented within a top level window (the `Default_FreeContainer`), using a `DefaultDisplay` for the collection that is the result of the query. The default display for a collection of objects is a Java Bean for object browsing that has been registered with the presentation model. If no such widget was available, the designer would have had to decompose the Display Results task into a much more complex (and hence lower level) set of subtasks. The system knows that the result of the search action task is to be used as the input to the final display in Fig. 14 because:

1. The result of the `Search` for a Book task has been stored in the state object `QueryResult` using the link wizard, as shown in Fig. 13.
2. The `QueryResult` state object has been indicated to be the input to the `DisplayResults` interaction task through a separate run of the link wizard.

3. The `DisplayResults` task is in turn linked to the final entry in Fig. 14, either as a result of model derivation or linking.

The developer must subsequently repeat the above process for all related presentation and task model components. Where necessary they must also specify the input and output data flow of each task by linking state objects and action/interaction tasks, as also described above. This process allows a developer to specify the dynamics and data flow requirements of otherwise static interface elements, such as could be developed in conventional UIDEs such as Visual Basic, etc. The running interface generated from the models described in this section is shown in Fig. 15.

7.1. Results of initial experiments

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

When presented with this simple case study, an expert developer (i.e. one who is experienced in UI development using Java, and also having experience of the POET ODBMS) implemented the code in 776 lines of code in a few days. Using Teallach, the same interface was developed in a couple of hours with no actual writing of code. Since the generated code utilises the Teallach runtime library, it consisted of only 353 lines of code. Although it is not intended that the code generated by Teallach will be used as *production* code, there are indications that the reduction of application-specific code may well improve the maintainability and testability of the code.

As yet we do not have sufficient evidence to suggest how developers will react to developing user interfaces with Teallach's models and design method. It will be interesting to see both how individual developers, and teams of multi-disciplinary developers (designers, software engineers, etc.) will respond to the proposed method. Experiments such as those performed by Vista's developers (Brown et al., 1998) with such multi-disciplinary teams will be of great interest. This area therefore needs further investigation.

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

8. Summary and concluding remarks

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

- A rich domain model that provides an abstract view of structural and behavioural features of applications in terms of the ODMG object model.
- A hierarchical task modelling language that is fully integrated with the domain model so that not only can user tasks be described, but also the data associated with those tasks.
- A presentation model in which abstract descriptions of displays can be associated with task and domain model concepts, and which is associated with an extensible concrete presentation based on Java Beans and Swing.
- A flexible methodology, in which the models provide abstract facilities for describing the user interface without imposing a prescriptive approach to model construction.
- A design environment that supports the flexible methodology, and in particular the association of components from different models.
- An open architecture, in which both application classes and auxiliary functionality can be made accessible to other Teallach models through facilities provided by the domain model, and in which additional presentation components can be registered with the presentation model using the Java Beans protocol.
- A code generator that produces executable Java Swing programs organised using the model view controller pattern from the Teallach models.

The design of the Teallach system has from the start been motivated by the desire to make the development of user interfaces to object database systems more systematic and more efficient. However, although we were willing to trade some measure of generality in the Teallach system for greater productivity in the database context, we believe that there is very little that is database specific about Teallach. While the domain model is clearly a data model from a database, the ODMG model has much in common with object models from other settings, a fact reflected in the use of the model for describing imported Java class libraries.

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

In a recent survey and critique of the state of user interface software tools (Myers et al., 2000), model-based techniques have suffered criticism due to (a) the unpredictability of the produced interfaces, (b) the limitations of the kind of interfaces produced, (c) the quality of the generated interfaces compared to those produced by traditional programming techniques, and (d) the reduced need for high-level models since interface elements have become more standardized. It is argued that such problems are reflected in the lack of general acceptance of model-based techniques. In the specific case of Teallach these criticisms can be addressed in turn as follows:

(a) The use of an abstract presentation model allows UI developers to specify general categories of interactors for use in their design. Such abstractions allow designers to focus on the general processing requirements of the interface, rather than on specific interface elements. Once a design has progressed, then designers can specify which widget should appear in the generated interface from the set of widgets that fall into the particular abstract category (or they can use the default), safe in the knowledge that the concrete widget used will provide the functionality to realise the interface's processing requirements. Unlike systems such as SELECTVISION (Vanderdonckt, 1999), Teallach does not however provide a means of informing designers of why a particular interface element has been chosen as the default.

(b) While Teallach is specifically targeted at WIMPS-based interfaces, its extensible widget set allows designers to introduce application-specific widgets into their design. This ability is a direct consequence of the standardisation effort noted by (Myers et al., 2000) above. Furthermore, the integrated nature of Teallach's models allows the generated interface to be more than just a series of unrelated screen designs, in that the generated interfaces correspond to the dialog and information processing requirements specified by Teallach's three models.

(c) The interfaces generated by Teallach are intended to assist in prototyping the interface of the final system, and as such these interfaces are not comparable to those produced by specialist programmers. The process of moving from such prototypical designs to a final interface is not an easy one, and remains an open area of research.

(d) Although interface components have become more standardised through the adoption of standards such as Java Beans and ActiveX, such components often encapsulate very complex functionality, but have simple I/O requirements. For example, the object browser bean used in the examples in this paper only requires a single object to be passed to it to allow complex browsing. Such standardised call interfaces allow Teallach's presentation model to register interfaces components within its various abstract interaction categories. The use of standardised call interfaces is therefore a feature to be exploited by Teallach's models, rather than a reason to stop using such models.

The Teallach tool has been developed as a research prototype as a proof of concept, and is as such not a mature system. Readers interesting in experimenting with Teallach in a research setting are advised to contact the first author of this paper.

Acknowledgements

This research is funded by the UK Engineering and Physical Sciences Research Council (EPSRC), whose support we are pleased to acknowledge. We are also grateful for the input of Andrew Dinn, Grant Forrester, Michael Smyth and Adrian West during the project. We would also like to thank the reviewers of the initial draft of this paper for their insights and invaluable comments.

References

de Baar, D., Foley, J., Mullet, K., 1992. Coupling Application Design and User Interface Design. Proceedings of CHI92 Conference. Monterey, May, pp. 259–266.

Balzert, H., Hofmann, F., Kruschinski, V., Niemann, C., 1996. The JANUS application development environment — generating more than the user interface. In: Vanderdonckt, J. (Ed.). Proceedings of Computer Aided Design of User Interfaces (CADUI'96). pp. 183–206.

Barclay, P., Griffiths, T., McKirdy, J., Paton, N., Cooper, R., Kennedy, J., 1999. The Teallach tool: using models for flexible user interface design. In: Vanderdonckt, J., Puerta, A. (Eds.). Proceedings of Computer Aided Design of User Interfaces II (CADUI'99). Kluwer Academic Press, Dordrecht, pp. 139–158.

Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I., Sacre, B., Vanderdonckt, J., 1995. Towards a systematic building of software architectures: the TRIDENT methodological guide. Interactive Systems: Design, Specification and Verification. Springer, Berlin, pp. 77–94.

Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I., Zucchinetti, G., Vanderdonckt, J., 1995. Critical issues in user interface systems engineering. Key Activities for a Development Methodology of Interactive Applications. Springer, Berlin.

Booch, G., Rumbaugh, J., Jacobson, I., 1999. The Unified Modelling Language User Guide. Addison-Wesley, Reading, MA.

Brown, J., Graham, T.C.N., Wright, T., 1998. The Vista environment for the coevolutionary design of user interfaces. Conference proceedings on Human factors in computing systems. ACM Press, Los Angeles, CA, pp. 376–383.

Buxton, W., Lamb, M., Sherman, D., Smith, K., 1983. Towards a comprehensive user interface management system. Comput. Graph. 17 (3), 35–42.

Cattell, R., 1997. The Object Database Standard: 2.0. Morgan Kaufmann, Los Altos, CA.

Chaudhri, A., Loomis, M., 1998. Object Databases in Practice. Prentice-Hall, Englewood Cliffs, NJ.

Cooper, R., McKirdy, J., Griffiths, T., Barclay, P., Paton, N., Gray, P.J.K., Goble, C., 2000. Conceptual modelling for database user interfaces. In: Arisawa, H., Catarci, T. (Eds.). Visual Information Managment, Proceedings of Visual Database Systems, Fukuoka, Japan. , pp. 117–128.

Eisenstein, J., Puerta, A., 1998. TIMM: Exploring Task-Interface Links in MOBI-D. In: CHI98 Workshop on From Task to Dialogue: Task-Based User Interface Design, Los Angeles, April.

Elnaffar, S., Graham, T., 1999. Semi-automatic linking of user interface design artifacts. In: Vanderdonckt, J., Puerta, A. (Eds.). Proceedings of Computer Aided Design of User Interfaces II (CADUI'99). Kluwer Academic Publishers, Dordrecht, pp. 127–138.

Elwert, T., Schlungbaum, T., 1995. Modelling and generation of graphical user interfaces in the TADEUS approach. In: Palanque, P. (Ed.). Design, Specification, and Verification of Interactive Systems (DSVIS'95). Springer, Berlin, pp. 193–208.

Foley, J., Sukaviriya, N., 1995. History, results, and bibliography of the user interface design environment

(UIDE), an early model-based system for user interface design and implementation. In: Paternó, F. (Ed.). Interactive Systems: Design, Specification, and Verification. , pp. 3–10.

Foley, J., Kim, W., Kovacevic, S., Murray, K., 1989. Defining Interfaces at a High Level of Abstraction. IEEE Comput., 25–32.

Gray, P., Cooper, R., Kennedy, J., McKirdy, J., Barclay, P., Griffiths, T., 1998. A Lightweight Presentation Model for Database User Interfaces. In: Proceedings of ERCIM'98.

Griffiths, T., McKirdy, J., Forrester, G., Paton, N., Kennedy, J., Barclay, P., Cooper, R., Goble, C., Gray, P., 1998. Exploiting model-based techniques for user interfaces to databases. In: Ioannidis, Y., Klas, W. (Eds.). Proceedings Visual Database Systems 4. Chapman & Hall, London, pp. 21–46.

Griffiths, T., McKirdy, J., Paton, N., Kennedy, J., Cooper, R., Barclay, P., Goble, C., Gray, P., Smyth, M., West, A., Dinn, A., 1998. An open model based user interface development system: the Teallach approach. In: Markopoulos, P., Johnson, P. (Eds.). Proceedings of Fifth Eurographics Workshop on Design, Specification and Verification of Interactive Systems, (DSV-IS'98). , pp. 32–49.

Griffiths, T., Barclay, P.J., McKirdy, J., Paton, N.W., Gray, P.D., Kennedy, J.B., Cooper, R., Goble, C.A., West, A., Smyth, M., 1999. Teallach: a model-based user interface development environment for object databases. In: Paton, N.W., Griffiths, T. (Eds.). Proceedings of User Interfaces to Data Intensive Systems (UIDIS'99). IEEE Computer Society, pp. 86–96.

Griffiths, T., Paton, N.W., Goble, C.A., West, A.J., 1999. Task modelling for database interface development. In: Bullinger, H.-J., Ziegler, J. (Eds.). Proceedings of Eighth International Conference on Human–Computer Interaction (HCI International), vol. 1. Lawrence Erlbaum, London, pp. 1033–1037.

Janssen, C., Weisbecker, A., Ziegler, J., 1993. Generation user interfaces from data models and dialogue net specifications. In: Proceedings of INTERCHI'93. Addison-Wesley, pp. 418–423

Johnson, J.A., Nardi, B.A., Zarmer, C.L., Miller, J.R., 1993. ACE: building interactive graphical applications. Commun. ACM 36 (4), 40–55.

Johnson, P., Johnson, H., Wilson, S., 1995. Scenario-based design. Rapid Prototyping of User Interfaces Driven by Task Models. Wiley, pp. 209–246.

Krasner, G., Pope, S., 1998. A cookbook for using the model-view-controller interface paradigm in smalltalk. J. Object-Oriented Programming 1 (3), 26–49.

Limbourg, Q., Hadj, B.A.E., Vanderdonckt, J., Keymolen, G., Mbaki, E., 2000. Towards derivation of presentation and dialogue from models: preliminary results. In: Proceedings of DSVIS'2000.

Lonczewski, F., Shreifer, S., 1996. The FUSE system: an integrated user interface design environment. In: Vanderdonckt, J. (Ed.). Proceedings of Computer Aided Design of User Interfaces (CADUI'96). , pp. 37–56.

Luo, P., Szekely, P., Neches, R., 1993. Management of Interface Design in HUMANIOD. In: Proceedings of INTERCHI'93.

Markopoulos, P., Pycock, J., Wilson, S., Johnson, P., 1992. Adept — A task based design environment. In: Proceedings of the 25th Hawaii International Conference on System Sciences. IEEE Computer Society Press, Silver Spring, MD, pp. 587–596.

Mitchell, K., Kennedy, J., Barclay, P., 1996. A Framework for User Interfaces to Databases. In: Proceedings of Advanced Visual Interfaces (AVI'96). ACM Press, New York.

Myers, B., Hudson, S.E., Pausch, R., 2000. Past, present, and future of user interface software tools. ACM Trans. Computer–Human Interaction 7 (1), 3–28.

Olsen, D., 1987. MIKE: the menu interaction kontol environment. ACM Trans. Graph. 5 (4), 318–344.

Paddock, R., Petersen, J.V., Talmage, R., Ranft, E., 1998. Visual Foxpro 6 Enterprise Development. Prima Publishing ISBN: 0761513817.

Paternó, F., 1999. Model-based Design and Evaluation of Interactive Applications. Springer, UK.

Paternó o, F., Mancini, C., Meniconi, S., 1997. ConcurTaskTrees: A Diagramatic Notation for Specifying Task Models. In: Proceedings of IFIP International Conference on Human–Computer Interaction (Interact'97). Chapman & Hall, London, Sydney, July, pp. 362–369.

Petoud, I., Pigneur, Y., 1989. An automatic and visual approach for user interface design. In: Cockton, G. (Ed.). Proceedings of the IFIP TC 2/WG 2.7 Working Conference on Engineering for Human–Computer Interaction (EHCI'89). Elsevier, Amsterdam, pp. 403–419.

Poet, 2000. POET Object Server Suite. JAVA Programmer's Guide.http://www.poet.com.

Prague, C., Irwin, M., Reardon, J., 2000. Microsoft Access 2000 Bible. IDG Books ISBN: 0764534041.

Puerta, A., 1996. The Mecano Project: comprehensive and integrated support for model-based interface development. In: Vanderdonckt, Jean (Ed.). Computer-Aided Design of User Interfaces. Presses Universitaires de Namur, pp. 19–25.

Puerta, A., 1997. A model-based interface development environment. IEEE Software 14 (4), 41–47.

Puerta, A., Eisenstein, J., 1999. Towards a general computational framework for model-based interface development systems. In: International Conference on Intelligent User Interfaces (IUI'99). ACM Press, New York, pp. 171–178.

Puerta, A., Maulsby, D., 1997. Management of interface design knowledge with MOBI-D. In: Proceedings of IUI'97. Orlando, Florida, January, pp. 249–252.

Puerta, A., Eriksson, H., Gennari, J., Musen, M., 1994a. Beyond Data Models for Automated User Interface Generation. In: Proceedings of HCI'94: People and Computers. Glasgow, UK, August, pp. 353–366.

Puerta, A., Eriksson, H., Gennari, J., Musen, M., 1994b. Model-based automated generation of user interfaces. In: Proceedings of National Conference on Artificial Intelligence (AAAI'94).

Schlungnaum, E., 1996, Model-based user interface software tools — current state of declarative models. Graphics, visualization and usability centre, Georgia Institute of Technology, GVU Tech #96-#30.

Schlungbaum, E., Elwert, T., 1996. Automatic user interface generation from declarative models. In: Vanderdonckt, J. (Ed.). Proceedings of Computer Aided Design of User Interfaces (CADUI'96), pp. 3–18.

da Silva, P.P., 2000. User interface declarative models and development environments: a survey. In: Proceedings of Seventh International Workshop On Design, Specification and Verification of Interactive Systems (DSV-IS'00), Springer, Berlin.

da Silva, P.P., Griffiths, T., Paton, N., 2000. Generating user interface code in a model based user interface development environment. In: Proceedings of Advanced Visual Interfaces (AVI'00). ACM Press, New York, pp. 155–160.

Stirewalt, R., 1999. MDL: a language for binding user-interface models. In: Vanderdonckt, J., Puerta, A. (Eds.). Proceedings of the Third International Conference on Computer-Aided Design of User Interfaces. Kluwer Academic Publishers, Dordrecht, pp. 159–170.

Szekely, P., 1995. Retrospective and challenges for model-based interface developments. In: Bodart, F., Vanderdonckt, J. (Eds.). Design, Specification, and Verification of Interactive Systems (DSVIS'95). Springer, Berlin, pp. 1–27.

Szekely, P., 1998. Readings in intelligent user interfaces. Reflections on Beyond Interface Builders: Model-Based Interface Tools. Morgan Kaufmann, Los ALtos, CA.

Szekely, P., Luo, P., Neches, R., 1992. Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In: Proceedings of CHI'92, pp. 507–515.

Szekely, P., Sukaviriya, P., Castells, P., Muhtkumarasamy, J., Salcher, E., 1996. Declarative interface models for user interface construction tools: the MASTERMIND approach. In: Wierse, A., Grinstein, G., Lang, U. (Eds.). Proceedings of Second Workshop on Database Issues for Data Visualization. Engineering For Human–Computer Interaction. Springer, Berlin.

Topley, K., 1999. Core Swing: Advanced Programming. Prentice Hall, Englewood Cliffs, NJ ISBN: 0130832928.

Vanderdonckt, J., 1999. Advice-giving systems for selecting interaction objects. In: Paton, N.W., Griffiths, T. (Eds.). Proceedings of the First International Workshop on User Interfaces to Data Intensive Systems UIDIS'99. IEEE Computer Society Press, Silver SPring, MD, pp. 152–157.

Vanderdonckt, J., Bodart, F., 1993. Encapsulating knowledge for intelligent automatic interaction objects selection. In: Ashlund, S., Mullet, K., Henderson, A., Hollnagel, E., White, T. (Eds.). Proceedings of the Conference on Human Factors in Computing Systems INTERCHI'93 Bridges Between Worlds. ACM Press, London, pp. 424–429.

Vanderdonckt, J., Tarby, J.-C., Derycke, A., 1998. Using data flow diagrams for supporting task models. In: Markopoulos, P., Johnson, P. (Eds.). Supplementary Proceedings of Fifth International Eurographics Workshop on Design, Specification, Verification of Interactive Systems (DSV-IS'98). Eurographics Association, Aire-la-Ville, pp. 1–16.

Warmer, J.B., Kleppe, A.G., 1998. The Object Constraint Language: Precise Modeling With Uml. , Addison-Wesley Object Technology Series. Addison Wesley, Reading, MA ISBN: 0201379406.

Wiecha, C., Bennett, W., Boyles, S., Gould, J., Green, S., 1990. ITS: a tool for rapidly developing interactive applications. ACM Trans. Inform. Syst. 8 (3), 204–236.

Zloof, M., 1998. Selected ingredients in end-user programming. In: Ioannidis, Y., Klas, W. (Eds.). Proceedings Visual Database Systems (VDB4). Chapman & Hall, London.