Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1408–1422
The Journal of Systems and Software
www.elsevier.com/locate/jss

Model-based user interface engineering with design patterns

Seffah Ahmed *, Gaffar Ashraf

Human-Centered Software Engineering Group, Department of Computer Science and Software Engineering, Concordia University, 1455, Maisonneuve West Bvld, Montreal QC, Canada H3G1M

Received 20 June 2006; received in revised form 10 October 2006; accepted 15 October 2006
Available online 8 December 2006

Abstract

The main idea surrounding model-based UI (User Interface) development is to identify useful abstractions that highlight the core aspects and properties of an interactive system and its design. These abstractions are instantiated and iteratively transformed at different level to create a concrete user interface. However, certain limitations prevent UI developers from adopting model-based approaches for UI engineering. One such limitation is the lack of reusability of best design practices and knowledge within such approaches. With a view to fostering reuse in the instantiation and transformation of models, we introduce patterns as building blocks, which can be first used to construct different models and then instantiated into concrete UI artefacts. In particular, we will demonstrate how different kinds of patterns can be used as modules for establishing task, dialog, presentation and layout models. Starting from an outline of the general process of pattern application, an interface for combining patterns and a possible formalization are suggested. The *Task Pattern Wizard*, an XML/XUL-based tool for selecting, adapting and applying patterns to task models, will be presented. In addition, an extended example will illustrate the intimate complicity of several patterns and the proposed model-driven approach.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Model-based user interface; User interface design patterns; Software architecture; Model-driven development; Usability; Task; Presentation and user interface models

1. Introduction

The model-based approach was introduced to support the specification and design of interactive systems at a semantic, conceptual and abstract level as an alternative to dealing with low-level implementation issues earlier on in the development lifecycle (Paternò, 2000). This way, designers can concentrate on important conceptual properties instead of being distracted by the technical and implementation details. As a result, the increasing complexity of the user interface is more easily managed. Moreover, the UI architecture is simplified, hence allowing for better system comprehension and traceability for future maintenance for different context of use.

Unfortunately, model-based methods are rarely used in practice (Trætteberg, 2004). One major reason for this limitation is that creating various models, instantiating and linking them together to create lower level models is a tedious and very time-consuming work, especially when most of the associated activities have to be done manually; tools provide only marginal support. This presents an overhead that is unacceptable in many industrial setups with limited resources, tough competition and short time-to-market. This crippling overhead can be partially attributed to the fact that model-based methods (MOBI-D, 1999; TADEUS, 1998; TERESA, 2004) lack the flexibility of reusing knowledge in building and transforming models. At best, only few approaches offer a form of copy-and-paste reuse. Moreover, many of these reuses involve the "reuser" merely taking a copy of a model component and manually changing it according to the new requirements. No form of consistency with the original solution is maintained (Mens et al., 1998). Copy-and-paste analysis

* Corresponding author.
E-mail address: seffah@cse.concordia.ca (S. Ahmed).
URL: http://hci.cs.concordia.ca (S. Ahmed).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.10.037

and the concept of fragmented design models are clearly inadequate when attempting to integrate reuse in a systematic and retraceable way in the model-based UI development life cycle.

This practical observation motivates the need for a more disciplined form of reuse. We will demonstrate that the reusability problems associated with current model-based approaches can be overcome through patterns. Patterns have been mainly known as proven solutions to well-known problems that occur in different situations. They are usually presented as a vehicle to capture the best practices and facilitate their dissemination. Because patterns are context-sensitive, the solution encapsulated in the pattern can be customized and instantiated to the current context of use before being reused (Alexander, 1997). Nevertheless, in order to be an effective knowledge-capturing tool in model-based approaches, the following issues require more attention:

- A classification of patterns according to models must be established. Such a classification would distinguish between patterns that are building blocks for models and patterns that drive the transformation of models, as well as create a concrete UI.
- A tool support that can assist developers when selecting the proper patterns, instantiating them once selected, as well as when combining them to create a model.

These two aspects are the essence of this paper. After a brief overview of existing model-based approaches, we will introduce how we have been combining model-based approaches and several patterns to build a framework for the development of user interfaces. A clear definition of the various models used and an outline of the UI derivation process are also given. Furthermore, we will suggest how to enhance this framework using patterns as a reuse vehicle. We will demonstrate how HCI patterns can be used as building blocks when constructing and transforming the various models, and list which kind of HCI patterns lend themselves to this use. A brief case study is presented in order to validate and illustrate the applicability of our approach and the proposed list of patterns.

2. Motivations and related work

2.1. Model-based user interface engineering

Model-based UI development has been investigated for more than a decade. In most approaches, model-based UI design is defined as the process of creating and refining models (da Silva, 2000; Vanderdonckt et al., 2003). Many models exist in order to describe the user interface at different levels including task, user, presentation, dialog, and platform models. Until now, no consensus has been reached as to which models are the best for describing UI's (da Silva, 2000) and which model can be instantiated and transformed at each step to create a concrete user interface. Moreover, it is to be noted that instead of automation (JANUS (Balzert, 1996), AME (Märtin, 1996)), most model-based approaches (MOBI-D (MOBI-D, 1999), TERESA (TERESA, 2004)) provide little support in helping developers to interactively define the mappings between various models in the design. Consequently, the mapping between models is done outside the semantics of these models, and mostly manually. This is a tedious and error prone practice that most designers prefer to avoid. We often end up with limited local modeling or no modeling at all.

Constructing and transforming models is a very time consuming task even for small size software. For large systems, it is often prohibitively expensive. It requires different skills and knowledge about the various models. Unfortunately, current approaches for model-based tools offer very limited if any support for reusing model fragments or templates. Rine (Rine and Nada, 2000) defined reuse as the use of existing software artifacts or knowledge to create new software components. Within the scope of model-based UI development, reuse is particularly critical. For example, developing the task model for a medium to large-size system that is highly interactive and providing context-dependent UI's is quite difficult, as shown in Paquette and Schneider (2004). Based on a case study, Paquette observed that task modeling becomes tedious when a task model is specified in sufficient detail. However, the level of detail is crucial if task models are to be used as a starting point for the evolution of the final UI and for preliminary evaluations.

Furthermore, in order to incorporate a great level of details, the various models may grow too large to be easily understood or fully comprehended. The MOBI-D tool suite does not offer any functionality that allows importing model or design fragments. Similarly, TERESA only supports cutting and pasting static task fragments. Important tasks in knowledge reuse such as resolving name or relationship conflicts are not supported. In essence, starting a new project means reinventing the wheel by building and linking all models from scratch. Constraints have to be manually applied and reworked between models, leading sometimes to inconsistencies that are only discovered in later design stages, implementation, or testing. Almost no design reuse from previous works, in terms of model fragments, is possible. Multiple views at various levels of abstraction and granularity should be provided.

2.2. Patterns in models construction and transformation

As will be detailed in this paper, UI design patterns have the potential to provide a solution to the reuse problem while acting as driving artifacts in the development and transformation of models.

Similar to the rest of the software engineering community, the Human Computer Interaction (HCI) design community has been a forum for vigorous discussion on pattern languages for user interface design and usability engineering. The goals of HCI patterns are to share successful HCI design solutions among HCI professionals and to provide a common ground for anyone involved in the design, development, evaluation or use of interactive systems (Borchers, 2001). Several HCI practitioners and interactive application designers have become interested in formulating HCI patterns and pattern languages.

However, in order to facilitate their reuse and applicability, patterns should be presented within a comprehensive framework that supports a structured design process, not just according to the structure of each individual aspect of the application (e.g., page layout, navigation, etc.), which is currently the case for most HCI patterns. This demonstrates the virtue of using model-based design approach in comparison to manual design practices. Gaffar et al. (2005) is one attempt to seemingly integrate patterns in the design process. Based on the UPADE Web Language, the approach aims to demonstrate when a pattern is applicable during the design process, how it can be used, as well as how and why it can or cannot be combined with other related patterns. Developers can exploit pattern relationships and the underlying best practices to devise concrete and effective design solutions.

Similarly, the "Pattern Supported Approach" (PSA) (Granlund and Lafreniere, 1999) addresses patterns not only during the design phase, but throughout the entire software development process. In particular, patterns have been used to describe business domains, processes and tasks to aid in early system definition and conceptual design. In PSA, HCI patterns can be documented according to the development lifecycle. In other words, during system definition and task analysis, it can be determined which HCI patterns are appropriate for the design phase, depending on the context of use. However, the concept of linking patterns together to put up a design is not tackled.

In addition, Molina et al. (2002) found that existing pattern collections focus on design problems, and not on analysis problems. As a result, he proposes the Just-UI framework, which provides a set of conceptual patterns that can be used as building blocks to create UI specifications during analysis. In particular, conceptual patterns are abstract specifications of elemental UI requirements, such as: *how to search*, *how to order*, *what to see* and *what to do*. Molina also recognized that the relatively-informal descriptions of patterns used today are not suitable for tool use. Within the Just-UI framework, a fixed set of patterns has been formalized so they can be processed by the "OliverNova" tool (OliverNova, 2004). Eventually, the JUST-UI framework will use code generation to derive a UI implementation based on the analysis model.

Breedvelt et al. (1997) discusses the idea of using task patterns to foster design knowledge reuse while task modeling. Task patterns encapsulate task templates for common design issues. This means that whenever designers realize that the issue with which they are contending is similar to an existing issue that has already been detected and resolved, they can immediately reuse the previously-developed solution (as captured by the task pattern). More specifically, task patterns are used as templates (or task building blocks) for designing an application's task model. According to Breedvelt, another advantage of using task patterns is that they facilitate reading and interpreting the task specification. Patterns can be employed as placeholders for common, repetitive task fragments. Instead of thinking in terms of tasks, one can think in terms of patterns at a more abstract level. Such an approach renders the task specification more compact and legible.

However, Breedvelt considers task patterns as individual static encapsulations of a (task) design issue in a particular context of use. Concepts for a more advanced form of reuse, including customization and combination, are not presented. Our approach (Gaffar et al., 2005) has highlighted another important aspect of the pattern concept: *pattern combination*. By combining different patterns, developers can utilize pattern relationships and combine them in order to produce an effective design solution. We will consider this principle in Section 5.2 and suggest an interface for combining patterns. As a result, patterns become a more effective vehicle for reuse.

3. PD-MBUI: A pattern-driven and model based user interface framework

3.1. Basic concepts and terminology

PD-MBUI (Pattern-Driven and Model-Based UI) aims to reconcile and unify in a single framework the pattern-driven and model-based approaches, two powerful methods for UI and software engineering as in Fig. 1. It comprises the following components:

1. A set of models including domain, task, user, environment, dialog, presentation and layout models. These models as well as the process of constructing them are detailed in Section 3.2.

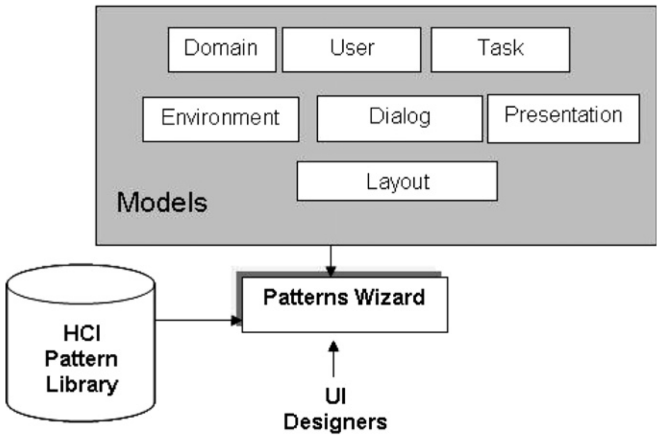


Fig. 1. PD-MBUI framework.

2. A library of HCI patterns that can be used, as building blocks, in this construction and transformations. A taxonomy and examples of patterns we identified are given in Section 4.1.
3. A method of identifying, instantiating and applying patterns during the construction and refinement of these models. This method is summarized in Section 5.2.
4. A tool, the Pattern Wizard, which helps user interface developers in selecting and applying patterns when constructing and transforming the various models to a concrete user interface. This tool aims to combine all the ingredients of the PD-MBUI in a single and integrative framework for the engineering of pattern-driven and model-based UI's.

Within our framework, we will be using the following notions as defined here:

- UI or HCI design patterns as proven solutions to a user problem that occurs in various contexts and projects. We take for granted that the proposed library of patterns and the pattern-driven approach are valid. This aspect has been largely debated by others including our preliminary work (Gaffar et al., 2005; Sinnig, 2004). An example of pattern is *Multi-Value Input Form* (Paternò, 2000). This task pattern provides a solution to the typical user problem of entering a number of related values. These values can be of different data types, such as "date," "string" or "real."
- User interface component or widget such as buttons, windows, dialog boxes are generally defined as object-oriented classes in UI toolkits such as Java swing, etc.
- An artifact is an object that is essential in order for a task to be completed. The state of this artifact is usually changed during the course of task performance. In contrast to an artifact, a tool is merely an object that supports performing a task. Such a tool can be substituted without changing the task's intention (Sinnig, 2004).

Within our framework, several notations have been used including XUL (Extensible User Languages), UML (Unified Modeling Language) as well as CTT (Concurrent Task Tree) for task modeling. We feel it is beyond the scope of this paper to deal with these notations in any substantive detail. Readers unfamiliar with these notations can found more details at http://www.mozilla.org/projects/xul http://www.uml.org and http://giove.cnuce.cnr.it/concurtasktrees.html.

The starting point of our approach is a clear description of the user requirements that include answers to the following questions: who are the users, what are their tasks and in which environment the system will be used. The answers to such questions are essential in deriving and constructing models that reflect the users and their real needs. They are also at the core of the assumption made by designers and engineers regarding the universal properties of users and uses.

3.2. Models used in PD-MBUI

Fig. 2 depicts the models we considered within our approach. We have selected these models based on the fact that they have been largely cited in the scientific literature (Puerta, 1997, 1996, 2002). We spent some effort to define them in such a way that they do not overleap and that the flow of transformation is also clearly stated as detailed in Fig. 1. This is fundamental in order to know precisely which type of pattern is needed and when it applies.

The process of constructing this variety of models distinguishes three major phases.

The starting point of phase I is the domain model. This model encapsulates the important entities of an application domain together with their attributes, methods and relationships (Schlungbaum, 1996). Within the scope of UI development, it defines the objects and functionalities accessed by the user via the interface. Such a model is generally developed using the information collected during the business and functional requirements stage. Two other models are then derived from this model: user and task models.

The user model captures the essence of the user's static and dynamic characteristics. Modeling the user's background knowledge is useful when personalizing the format of the information (e.g. using an appropriate language that is understood by the user). The task model specifies what the user does or wants to do, and why. It describes, at abstract meaning without any knowledge of the tasks that users perform while using the application, as well as how the tasks are interrelated. In simple terms, it captures the user tasks and system behavior with respect to the task set. Beside natural language, notations such as GOMS and CTT are generally used to document task models. The task model is constructed in mutual relationship to the user model, representing the functional roles played by users when accomplishing tasks, as well as their individual perception of the tasks. The user model is also related to the domain since the user may require different views

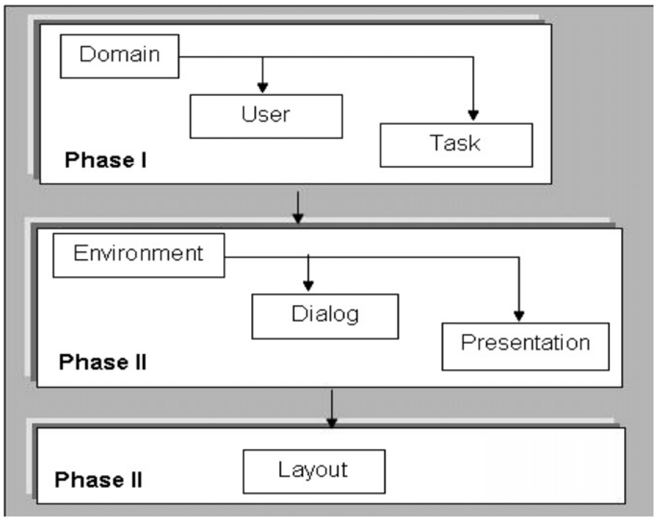


Fig. 2. Models and their relationships in the PD-MBUI Framework.

of the same data while performing a task. Moreover, a relationship must be formed between the domain model and the task model, because objects of the domain model may be needed in the form of artifacts and tools for task accomplishment.

The second stage starts with the development of the environment model. This model specifies the physical and organizational context of interaction (Trætteberg, 2002). For example, in the case of a mobile user application, an environmental model would include variables such as the user's current location, the constraints and characteristics presented by this location, the current time and any trigger conditions specified or implied by virtue of the location type. This model describes also the various computer systems that may run a UI (Prasad, 1996). The platform model describes the physical characteristics of the target platforms, such as the target devices' input and output capabilities. Based on this model and those developed in the first stage, the dialog and the presentation model can be developed.

The dialog model specifies when the end user can invoke functions and interaction media, when the end user can select or specify inputs, and when the computer can query the end user and present information (Puerta, 1997). In particular, this model specifies the user commands, interaction techniques, interface responses and command sequences permitted by the interface during user sessions. The presentation model describes the visual appearance of the user interface (Schlungbaum, 1996). The presentation model exists at two levels of abstraction: the abstract and the concrete presentation model. The former provides an abstract view of a generic interface, which represents a corresponding task and dialog model.

In the last stage, the "Layout Model is realized as a concrete instance of an interface. This model consists of a series of UI components that defines the visual layout of UI and the detailed dialogs for a specific platform and context of use.There may be many concrete instances of layout model that can be derived from a presentation and dialog model.

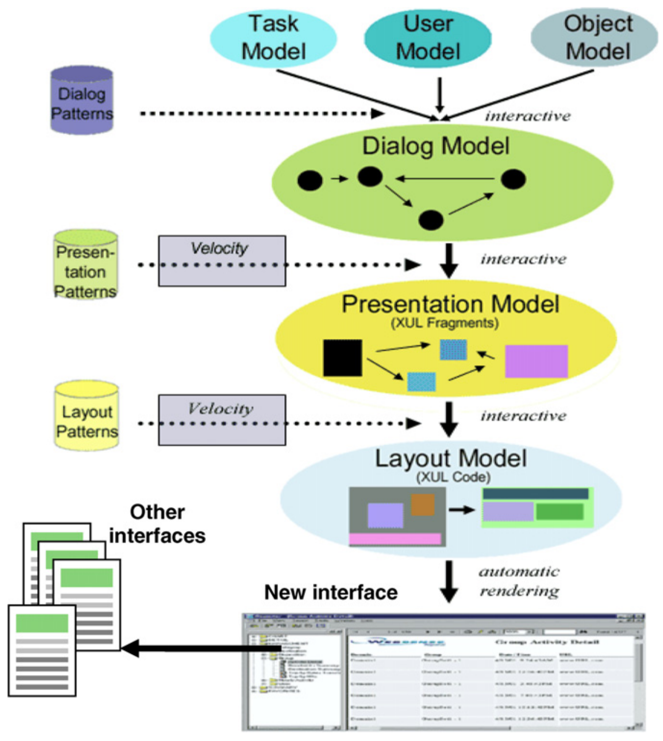


Fig. 3. PD-BMUI framework revisited (putting it all together).

- Patterns for the dialog model are employed to help with grouping the tasks and to suggest sequences between dialog views.
- Presentation patterns are applied to map complex tasks to a predefined set of interaction elements that were identified in the presentation model.
- Layout patterns are utilized to establish certain styles or "floor plans" which are subsequently captured by the layout model.

The following table summarizes some of the patterns we considered.

4. Proposed HCI patterns

4.1. HCI patterns taxonomy and samples

Fig. 3 portrays which types of patterns are used to construct and transform each type of models and how this happen in the model-driven engineering process.

4.2. Patterns as building blocks within a model based methodology

The following are the major types of patterns we considered with our framework:

- Task and feature patterns are used to describe hierarchically structured task fragments. These fragments can then be used as task building blocks for gradually building the envisioned task model.

4.3. Patterns instantiation and application

In this paper, we stated that patterns could be used as building blocks for the construction and transformation of the different models. For the construction of models, the following process, which is part of the PD-MBUI, was proposed to instantiate and apply patterns:

1. *Identification*: A subset M' of the target model M is identified thus: $M' \subset M$. This relationship should reduce the domain size and help focus attention on a smaller, more pertinent subset for the next step.
2. *Selection*: An appropriate pattern P is selected to be applied to M'. By focusing on a subset of the domain, the designer can scan M' more effectively to identify potential areas that could be improved through patterns. This step is highly dependent on the experience and creativity of the designer.

3. *Adaptation*: A pattern is an abstraction that must be instantiated. Therefore, this step has the pattern *P* adjusted according to the context of use, resulting in the pattern instance *S*. In a top-down process, all variable parts are bound to specific values, which yield a concrete instance of the pattern.
4. *Integration*: The pattern instance *S* is integrated into *M′* by connecting it to the other elements in the domain. This may require replacing, updating or otherwise modifying the other objects to produce a seamless piece of design.

Variables are used as placeholders for the context of use. During the process of pattern adaptation, these placeholders are replaced by concrete values representing the particular consequences of the current context of use.

Fig. 4 shows the interface of pattern *A*. The UML notation for parametric classes is used to convey that the pattern assumes two parameters (variables *x* and *y*). In order to instantiate the pattern, both variables must be assigned concrete values. In practical terms, the interface informs the user of the pattern that the values for variables *x* and *y* must be provided in order for the pattern to be used. In the figure, pattern *A* has been instantiated, resulting in Pattern *A Instance*. In addition, UML stereotypes are used to signal the particular type (role) of the pattern.

Most often, patterns are implemented using other patterns, i.e., a pattern can be seen as a composition of several basic patterns that we refer to them as sub-patterns or related patterns. The relationship between patterns can be between patterns related to the same model such a series of patterns related to the construction of a task model. It can be also between patterns related to different models. For example:

- If we consider a task pattern *X* for building a task model A.

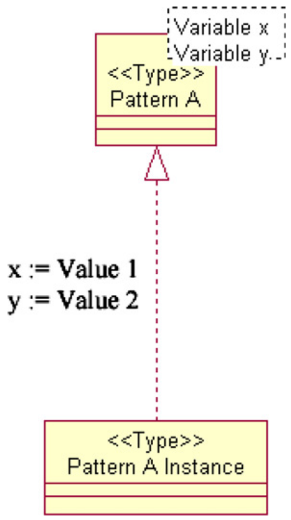


Fig. 4. Interface of a pattern.

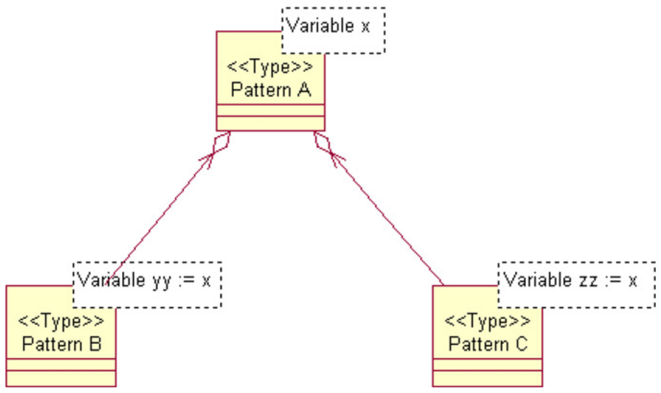


Fig. 5. Pattern aggregation.

- Then the presentation patterns *X* and *Y* apply for the construction of the presentation model B.
- Model B is seen as a transformation of the model A.

The relationship between patterns uses the concept of class aggregation, as presented in Fig. 3. Pattern *A* consists of the sub-patterns *B* and *C*. If we place patterns in this kind of relationship, special attention must be given to the patterns' variables. A variable defined at the super-pattern level can affect the variables used by the sub-patterns. In Fig. 5, variable *x* of pattern *A* affects the variables *yy* and *zz* of sub-patterns *B* and *C*. During the process of pattern adaptation, variables *yy* and *zz* will be bound to the value of *x*. As such, we observe how modifying a high-level pattern can affect all sub-patterns.

5. Examples of models construction using patterns

In previous sections of this paper, it was shown how patterns can generally be applied to models and how they can be aggregated. This section provides an in-depth discussion of how different categories of patterns can be used together when constructing the task, dialog, presentation and layout model.

5.1. Patterns in task modeling

Patterns for the task model describe generic reusable task fragments that can serve to establish the task model. In particular, instances of task patterns (i.e., already customized patterns) can be used as building blocks for the task model. Examples of such patterns for the task model include: *Find* something, *Buy* something, *Search for* something, *Login* to the system or *Fill out* an input form.

A typical example of a task pattern is *Search* (Gaffar et al., 2005). The pattern is suitable for interactive applications that manage considerable amounts of user-accessible data. The user wants to have fast access to a subset of this data.

As a solution, the pattern suggests giving the user the possibility to enter search queries. On the basis of these queries, a subset of the searchable data (i.e., the result set) is calculated and displayed to the user. The *Multi-Value Input Pattern* (Paternò, 2000; Sinnig, 2004) may be used for the query input. After submission, the results of the search are presented to the user and then they can either be browsed (*Browse Pattern* (Sinnig, 2004)) or used as input for refining the search.

Fig. 6 illustrates how the *Search* pattern is composed of the sub-patterns *Multi-Value Input* and *Browse*, as well as of recursive references to itself (*Search*). It also demonstrates how the variables of each pattern are interrelated. The value of the "Object" variable of the *Search Pattern* will be used to assign the "Object" variable of the *Browse* and Sub-*Search Patterns*. In addition, a subset of the "Search" object attributes is used to determine the various "Input Fields" of the *Multi-Value Input Pattern*, which is in turn responsible for capturing the search query. During the adaptation process, variables of each pattern must be resolved in a top-down fashion and replaced by concrete values.

The suggested task structure of the *Search Pattern* is illustrated in Fig. 7. In order to apply and integrate the task structure, the pattern and all its sub-patterns must be instantiated and customized to the current context of use.

The top-down process of pattern adaptation can be greatly assisted by tools such as wizards. A wizard moves through the task pattern tree and prompts the user whenever it encounters an unresolved variable. In Section 7, we introduce the Task Pattern Wizard, a tool that assists the user in selecting, adapting and integrating task and feature patterns.

5.2. Patterns in dialog modeling

Our framework's dialog model is defined by a so-called dialog graph. Formally speaking, the dialog graph consists of a set of vertices (dialog views) and edges (dialog transitions). Creating the dialog graph is a two-step process: first, related tasks are grouped together into dialog views. Second, transitions from one dialog to another, as well as trigger events are defined.

In order to foster establishing the dialog model, we believe that patterns can help with both grouping tasks to dialog views and establishing the transition between the various dialog views.

A typical dialog pattern is the *Recursive Activation Pattern* (Breedvelt et al., 1997). This pattern is used when the user wishes to activate and manipulate several instances of a dialog view. In practical terms, it suggests a dialog structure where, starting from a source dialog, a specific creator task can be used to instantiate a copy of the target dialog view. The pattern is applicable in many modern interfaces where several dialog views of the same type and functionality are concurrently accessible. A typical example of an application scenario is an e-mail program that supports editing several e-mails concurrently during a given session.

In the left pane of Fig. 8, we observe that in order to adapt (instantiate) the pattern, the source dialog view and the corresponding creator task, as well as the target dialog view must all be set. A specific instance of the pattern is shown in the right part of Fig. 8, simulating

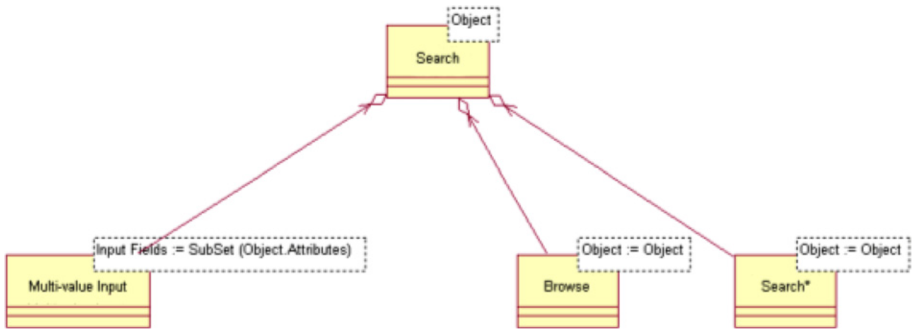


Fig. 6. Interface and composition of the search pattern.

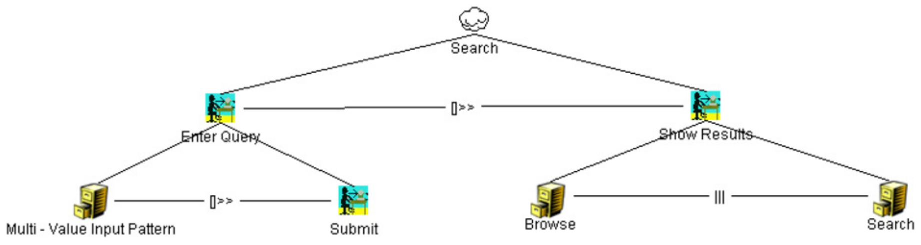


Fig. 7. Structure of the search pattern.

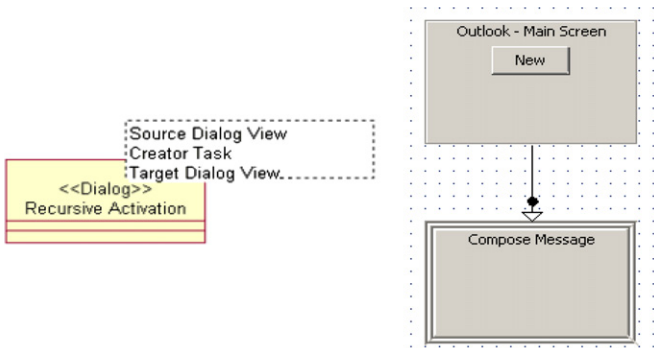


Fig. 8. Interface and "Microsoft Outlook" instance of the recursive activation pattern.

the navigational structure of Microsoft Outlook when composing a new message. For this particular example, the visual notation of a tool called the Dialog Graph Editor (Sinnig, 2004) was used.

5.3. Patterns in presentation modeling

The abstract delineation of user interface is determined in the presentation model through a defined set of abstract UI elements. Examples of such UI elements are Buttons, Lists or more complex aggregated elements such as trees or forms. Note that all interaction elements should be described in an abstract manner without reference to any particular interface components. Likewise, style attributes such as size, font and color remain unset, pending definition by the layout model. Abstraction is key to the success of presentation model as it frees the designers from unneeded details and allows for more efficient reuse on different platforms.

Patterns for the presentation model can be applied when describing the abstract UI elements. However, they can be more effective when applied for defining and mapping complex tasks (such as advanced search) to a predefined set of interaction elements. In this many-to-many interaction, patterns can provide insight into proven solutions ready to reuse.

One illustrative example of a presentation pattern is the *Form Pattern*. It is applicable when the user must provide the application with structural, logically-related information. In Fig. 9, the interface of the *Form Pattern* is presented, indicating that the various Input Fields to be displayed are expected as parameters. It is also shown that the *Unambiguous Format Pattern* can be employed in order to implement the *Form Pattern*.

In particular, the *Unambiguous Format Pattern* is used to prevent the user from entering syntactically-incorrect data. In conjunction with the *Form Pattern*, it determines which interaction elements will be displayed by the input form. XUL code will be produced for the most suitable interaction element, depending on the data type of the desired input as shown in Fig. 10. Three different instances of the *Unambiguous Format Pattern* are presented.

5.4. Patterns in layout management modeling

In this step, the abstract UI elements of the presentation are physically positioned following an overall layout or floor plan, which yields the layout model. Furthermore, the visual appearance of each interaction element is specified by setting fonts, colors and dimensions.

There are two different ways in which patterns can be employed when defining the layout model: (1) by providing a floor plan for the UI and (2) by setting the style attributes of the various widgets of the UI. The proposed solutions and the criteria of selecting between different designs depend – among other factors – on the context of use, nature of the application and satisfaction of the users. Aesthetic and human behavior aspects can complicate the

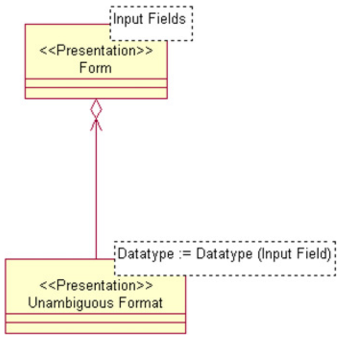


Fig. 9. Interface of the form pattern.

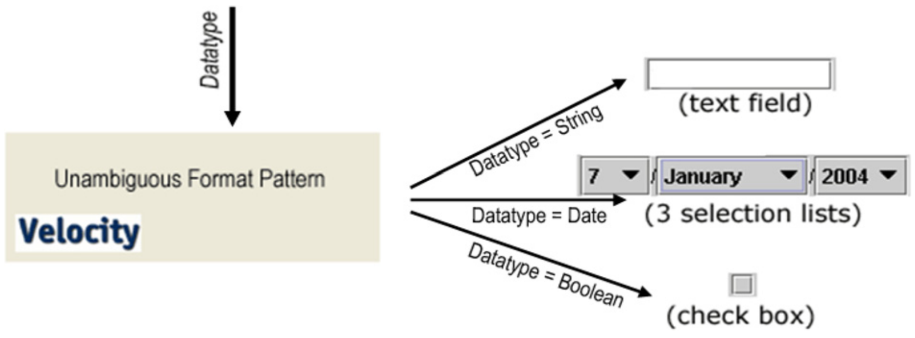


Fig. 10. Unambiguous format pattern with three unique instances.

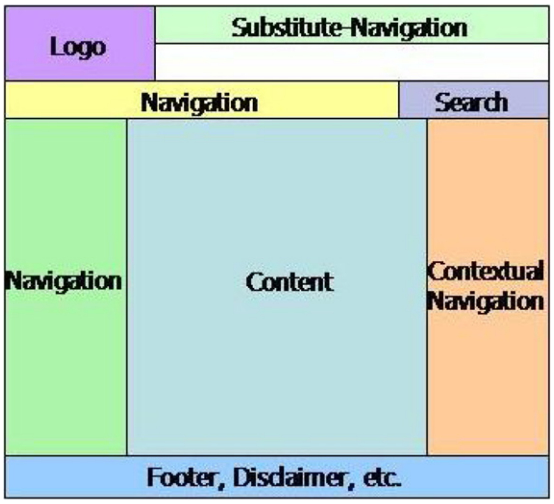


Fig. 11. Floor plan suggested by the portal pattern (Welie, 2004).

design and make the final results unpredictable. Patterns come in handy as shortcuts to analyzing some of these considerations by offering solutions that have been used before with good results.

The layout planning consists of determining the composition of the UI by providing a floor plan. Examples of such patterns are the *Portal Pattern* (Welie, 2004), *Card Stack* (Tidwell, 2004), *Liquid Layout* (Tidwell, 2004) and *Grid Layout* (Welie, 2004). Fig. 11 presents the floor plan suggested by the *Portal Pattern*, which is applicable for web-based UIs.

In the style planning, *Layout Patterns* are beneficial when the style attributes of the various widgets of the UI must be configured. For instance, the *House Style Pattern* suggests maintaining an overall look-and-feel for each page or dialog in order to mediate the impression that all pages share a consistent presentation and appear to belong together.

6. Putting it all together: the pattern wizard tool

Efficiently choosing and applying patterns requires tool support. In response, we have implemented a prototype of a task pattern wizard that is designed to support all phases of pattern application for the task model, ranging from pattern selection to adaptation to integration. After parsing the pattern, the tool guides the user step by step through the pattern adaptation and integration process. In what follows, we provide a brief description of how the Task Pattern Wizard is used. Particular attention is given to outlining how the tool supports each phase of pattern application.

The generic user interface description language, XUL, has been selected as the medium by which patterns and models will be described. This is because XUL provides a clear separation between the high-level user interface definition (in terms of abstract widgets that comprise the UI and that form the syntax and semantics the language) and its visual appearance (the final rendering in terms of layout and *look-and-feel*[1]). XUL is particularly well suited to our approach because, similar to XUL, we also distinguish between the abstract definition of the UI (presentation model) and the actual visual appearance (layout model). Since our presentation model consists of a set of XUL fragments, patterns for the presentation model are used to dynamically generate XUL code. Each presentation pattern has been formulated as a so-called XUL Velocity template. Velocity is a Java-based template engine. The focus of the Velocity Template Language is to describe the generation of any form of mark up code such as XUL.

Furthermore, the various XUL fragments of the presentation model are merged in the layout model, resulting in aggregated XUL code. The loosely connected XUL pieces are nested and associated together. The way these fragments are merged depends on the overall layout of the application. In addition, any style attributes that are not set are bound with concrete values. Because Velocity templates can be used to generate XUL fragments (for the presentation model), they can also serve when aggregating XUL code. Consequently, layout patterns are also formalized as Velocity XUL templates.

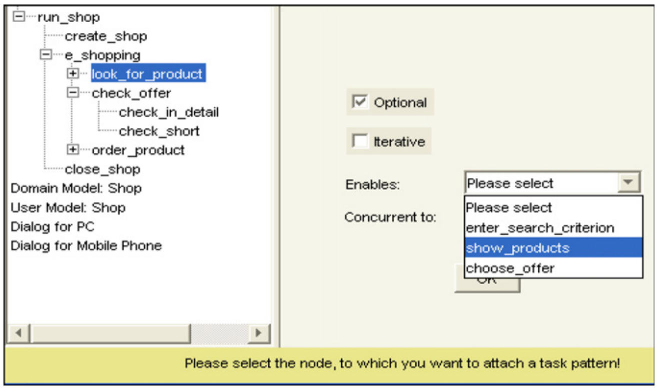
6.1. Identification

In the identification phase, we identify a node of an existing task structure to which a pattern will be applied. After opening the XUL file, the Task Pattern Wizard uses a tree element to display the task structure. Next, the user chooses a node representing a particular task. The user is then prompted to decide how the new task structure, brought in by the pattern, should be integrated into the existing tree. Some characteristics of the new task structure can be integrated as being optional and/or iterative. Optional tasks can be executed, but are not mandatory. Iterative tasks can be repeated more than once. These two characteristics are orthogonal to each other and any combination of them is logically correct. In addition, the new task's temporal relationship with the other tasks can be defined. Fig. 12 is a screenshot taken while identifying a target node from the task model.

6.2. Selection

In the selection phase, an applicable pattern is chosen. In order to perform the selection operation, the Task-Pattern Wizard presents the currently-opened pattern according to the Alexandrian form (Alexander, 1997). Specifically, the pattern is displayed in narrative form: what issue will be solved, when the pattern can be applied (context), how it works (solution) and why it works (rationale).

[1] The layout refers to the arrangement of the user interface real estate. The "look" refers to the rendering of each component; its shape and color, while the "feel" refers to how it behaves when the user interact with it. The "look-and-feel" is often used by java to emphasize how same interfaces look and behave differently on different platforms.



Please select the node, to which you want to attach a task pattern!

Fig. 12. The task pattern Wizard's selection screen.

6.3. Adaptation

Once the appropriate pattern has been chosen, it must be adapted to the current context of use. Each pattern generally contains variables that act as placeholders for context conditions. Different kinds of variables exist, such as "substitution variables" and "process variables."

Substitution variables are simply used as placeholders for certain values (such as the task name). During the pattern adaptation process, the Task Pattern Wizard will prompt the user to enter values for these variables. Then, each occurrence of the substitution variable will be replaced (substituted) with this value in a top-down process.

Process variables are used to describe the structure of the task fragment that will be created by the pattern. For example, entering values in a form is very repetitive. The same basic task (i.e., entering a value) appears over and over again. Each peer task can be distinguished simply by its name and input type. Thus, instead of describing each of these tasks individually, process variables that indicate the number of respective tasks can be used.

After all variables have been defined, a pattern instance that can be integrated into the task model is created.

6.4. Integration

In the integration phase, the pattern instance is incorporated into the current task model. In short, a new branch is added to the task tree or an existing branch is replaced. The new modified task structure can then be saved in XUL format. Within our tool set, XUL serves as a universal exchange format. This approach enables tools such as the XUL Task Simulator and the Dialog Graph Editor to further process the new task model (Sinnig, 2004).

7. An illustrative case study

The management of a hotel is going to be computerized. The hotel's main business is renting out rooms of various types. There is a total of 40 rooms available, priced according to their amenities. The hotel administration needs a tool capable of booking rooms for specific guests. More specifically, the application's main functionality consists of adding a guest to the internal database and booking an available room for a registered guest. Moreover, only certified guests have access to the main functionality of the program. Eventually, the application would be running on WIMP-based systems.

Note that only a simplified version of the hotel management system will be developed. The application and corresponding models will not be tailored to the different platform and user roles. The main purpose of the example is to show that model-based UI design consists of a series of model transformations, in which mappings from the abstract to the concrete models must be specified. Furthermore, it will be shown how patterns are used to establish the various models, as well as to transform one model into another. A summary of all patterns used in this article can be found in Table 1. For a more detailed description, refer to Sinnig (2004).

7.1. The task model

Fig. 13 depicts the coarse-grained task structure of the envisioned hotel management application. Only high-level tasks and their relationships are portrayed. An impression about the overall structure and behavior of the applications is given. The structure provided is relatively unique for a hotel management application; the concrete "realization" of the high-level tasks has been omitted. The Pattern Task symbol is used as a placeholder representing the suppressed task fragments.

A large part of many interactive applications can be developed from a fixed set of reusable components. If we decompose the application far enough, we will encounter these components. In the case of the task model, the more the high-level tasks are decomposed, the easier the reusable task structures (that have been gained or captured from other projects or applications) can be employed. In our case, these reusable task structures are documented in the form of patterns. This approach ensures an even greater degree of reuse, since each pattern can be adapted to the current context of use.

The main characteristics of the envisioned hotel management application, modeled by the task structure of Fig. 13, can be outlined as follows:

Accessing the application's main functionality requires logging in to the system (the login task enables the management task). The key features are "adding a guest" by entering the guest's personal information and "booking a hotel room" for a specific guest. Both tasks can be performed in any order. The booking process consists of four consecutively-performed tasks (related through "Enabling with Information Exchange" operators):

1. Locating an available room.
2. Assigning the room to a guest.
3. Confirming the booking.
4. Printing a confirmation.

Table 1
Pattern summary

Pattern name	Type	Problem
Browse	Task	The user needs to inspect an information set and navigate a linear ordered list of objects such as images or search results
Dialog	Task	The user must be informed about something that is requiring attention. The user must make a decision that will have an impact on further execution of the application, or the user must confirm the execution of an irreversible action
Find	Task	The user needs to find any kind of information provided by the application
Login	Task	The user needs to be authenticated in order to access protected data and/or to perform authorized operations
Multi-Value Input Form (Paternò, 2000)	Task	The user needs to enter a number of related values. These values can be of different data types, such as "date," "string" or "real"
Print Object	Task	The user needs to view the details related to a particular information object
Search	Task	The user needs to extract a subset of data from a pool of information
Wizard (Welie, 2004)	Dialog	The user wants to achieve a single goal, but several consecutive decisions and actions must be carried out before the goal can be achieved
Recursive Activation (Paternò, 2000)	Dialog	The user wants to activate and manipulate several instances of a dialog view
Unambiguous Format	Presentation	The user needs to enter data, but may be unfamiliar with the structure of the information and/or its syntax
Form	Presentation	The user must provide structured textual information to the application. The data to be provided is logically related
House Style (Tidwell, 2004)	Layout	Applications usually consist of several pages/windows. The user should have the impression that it all shares a consistent presentation and appears to belong together

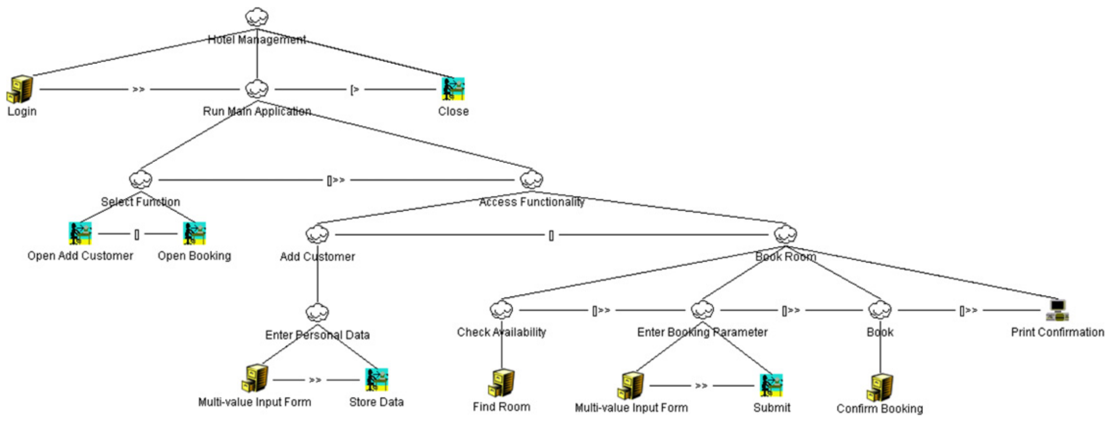


Fig. 13. Course-grained task model of the hotel management application.

As shown in Fig. 13, the *Login*, *Multi-Value Input Form*, *Find* and *Dialog* patterns can be used in order to complete the task model at the lower levels. In the next section, the application of the *Find Pattern* will be described in greater detail.

7.2. Completing the find room task

The *Find Pattern* is essential to completing the "Find Room" task. In contrast to the patterns already used in this example, the *Find Pattern* suggests a number of options rather than providing a task structure. Fig. 14 illustrates how finding an object can be performed by searching, browsing or employing an agent, depending on the pattern.

Within the scope of the hotel management application, the task of finding an available room should only be performed by searching with query parameters. As shown in Fig. 14, the "Information" variable of the *Find Pattern* (in this case, a placeholder for the "Hotel Room" value) is used to assign the "Object" variable of the *Search Pattern*.

The *Search Pattern* suggests a structure in which the search queries are entered, then the search results are displayed. Again, the *Multi-Value Input Form Pattern* is used to model the tasks for entering the search parameters into a form. The following search parameters can be used when searching for an available room: "Arrival Date," "Departure Date," "Non-Smoking," "Double/Single," "Room Type." After submitting the search queries, the search results (i.e., the available hotel rooms) can be manually scanned using the *Browse Pattern* or, based on the search results; a refinement search can be performed by employing the *Search Pattern* recursively. For the scope of this case study, refinement searches are unnecessary, and the search results should only be browsed.

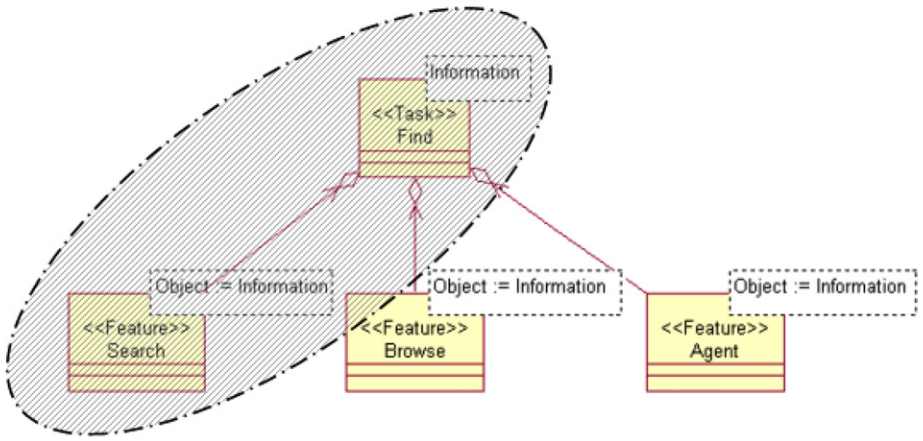


Fig. 14. Interface and structure of the find pattern.

According to the *Browse Pattern,* the list of objects (the hotel rooms) is printed, after which it can be interactively browsed as an option. Details of the hotel room can be viewed by selecting it. The *Print Object Pattern* is used to print out object's properties. It suggests using application tasks to print the object values that can be directly or indirectly derived from the object's attributes. In the case of the hotel management application, the following hotel room attributes should be printed: "Room Number," "Smoking/Non-Smoking," "Double/Single," "Room Type," and "Available Until."

After adapting all patterns to suit the hotel management application, the task structure displayed in Fig. 15 is derived. Note that the "Make Decision" task has been added manually, without pattern support.

A first draft of the envisioned task model can be derived once all patterns have been adapted and instantiated. At this point, first evaluations can be carried out. For instance, the XUL-Task-Simulator (Sinnig, 2004) can be used to simulate and animate possible scenarios. Results of the evaluation indicate that preliminary modifications and improvements of the task model are possible.

7.3. Designing the dialog structure

After establishing the task model for our example application, the dialog models can be interactively derived. In particular, the various tasks are grouped to dialog views, then transitions are defined between the various dialog views. Since the desired target platform of the hotel management application is a WIMP-based system, a dialog view will be subsequently implemented as either a window or a container in a complex window.

When designing the dialog graph for the hotel management application, we designated the login dialog view as both modal and the start-up dialog. After executing Submit, the "Main Menu" dialog will be opened. As such, a sequential transition between both dialog views is defined. From the main menu, either the "Add Guest" or "Search Applicable Room" dialog view can be opened by a

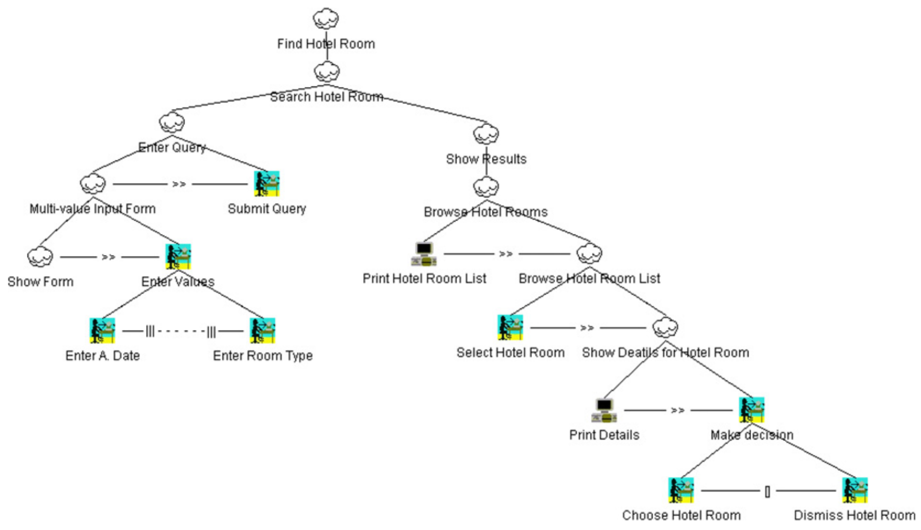


Fig. 15. Concrete task structure delivered by the find pattern.

sequential transition. After completing the "Add Guest" dialog view, the main menu will be re-opened. For this reason, a sequential transition to the "Main Menu," initiated by the "Store Data" task, must be defined.

The application's booking functionality consists of a series of dialog views that must be completed sequentially. The *Wizard* dialog pattern emerges as the best choice for implementation. It suggests a dialog structure where a set of dialog views is arranged sequentially and the "last" task of each dialog view initiates the transition to the following dialog view. Fig. 16 depicts the *Wizard Pattern's* suggested graph structure.

After applying the *Wizard Pattern*, the dialog views "Search Applicable Room," "Browse Results," "Show Details," "Enter Booking Parameters," "Confirm Booking" and "Print Confirmation" are connected by sequential transitions.

However, the sequential structure of the booking process must be slightly modified in order to enable the user to view the details of multiple rooms at the same time. Specifically, this behavior should be modeled using the *Recursive Activation* dialog pattern. This pattern is used when the user wishes to activate and manipulate several instances of a dialog view. In this particular case, the user will be able to activate and access several instances of the "Show Room Details" dialog view. This pattern suggests the following task structure: starting from a source dialog view, a creator task is used to concurrently open several instances of a target dialog view. In our example, the source dialog view is "Browse Rooms" and the "Select Room" task is used to create an instance of the "Show Room Details" dialog view.

A premature exit should be provided to offer the user the possibility to abort the booking transaction. In the hotel management application, this is achieved by the "Confirm Booking" dialog view. At this point, the user can choose whether to proceed with the booking or to abort the transaction. Another sequential transition must therefore be defined: one which is initiated by the "Select Cancel Booking" tasks and leads back to the main menu. The hotel management application's complete dialog graph, as visualized by the Dialog Graph Editor is depicted in Fig. 17.

The next step is to evaluate the defined dialog graph. The dialog graph can be animated using the Dialog Graph Editor to generate a preliminary abstract prototype of the user interface. It is possible to dynamically navigate through the dialog views by executing the corresponding tasks. This abstract prototype simulates the final interface's

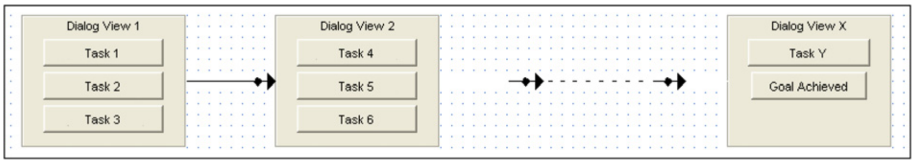


Fig. 16. Graph structure suggested by the Wizard pattern.

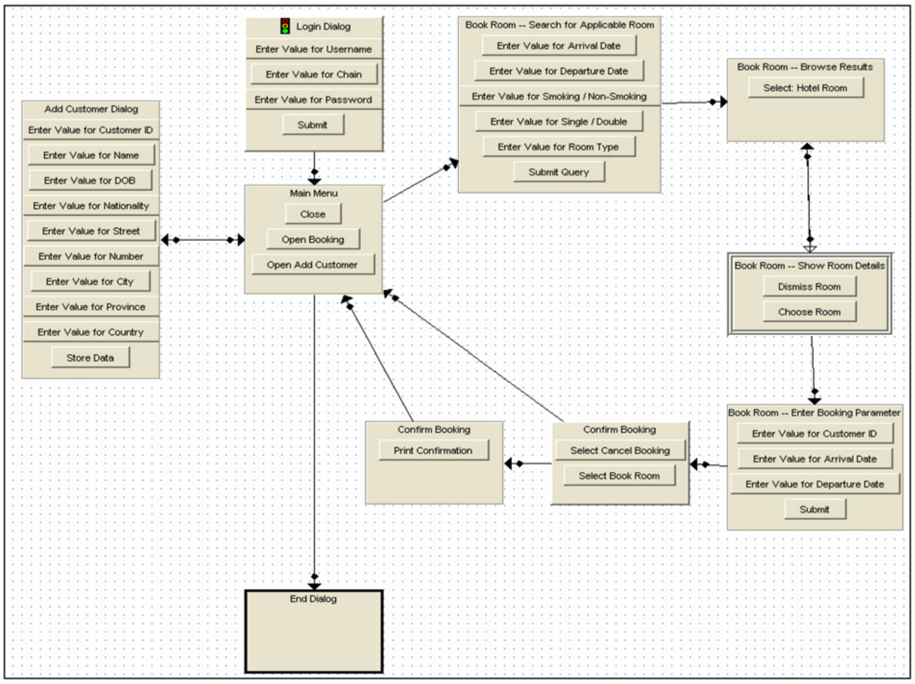


Fig. 17. Dialog graph of the hotel management application.

navigational behavior. It supports communication between users and software developers: design decisions are transparently intuitive to the user, and stakeholders are able to experiment with a dynamic system.

7.4. Defining the presentation and layout model

In order to define the presentation model for our example, the grouped tasks of each dialog view are associated with a set of interaction elements, among them forms, buttons and lists. Style attributes such as size, font and color remain unset and will be defined by the layout model.

A significant part of the user's tasks while using the application revolves around providing structured textual information. This information can usually be split into logically related data chunks. At this point, the *Form Presentation Pattern*, which handles this exact issue, can be applied. It suggests using a form for each related data chunk, populated with the elements needed to enter the data. Moreover, the pattern refers to the *Unambiguous Format Pattern*, in conjunction with which it can be employed.

The purpose of the *Unambiguous Format Pattern* is to prevent the user from entering syntactically incorrect data. Drawing on information from the business object model, it is able to determine the most suitable input element. In other words, depending on the domain of the object to be entered, the instance of the pattern provides input interaction elements chosen in such a way that the user cannot enter syntactically incorrect data.

Fig. 18 shows the windows prototype interfaces rendered from the XUL fragments of the hotel management application's presentation model for the "Login," "Main Menu," "Add Guest" and "Find Room" dialog views. All widgets and UI components are visually arranged according to the default style.

In the layout model, the style attributes that have not yet been defined are set in keeping with the hotel management application's standards. According to the *House Style Pattern* (which is applicable here), colors, fonts and layouts

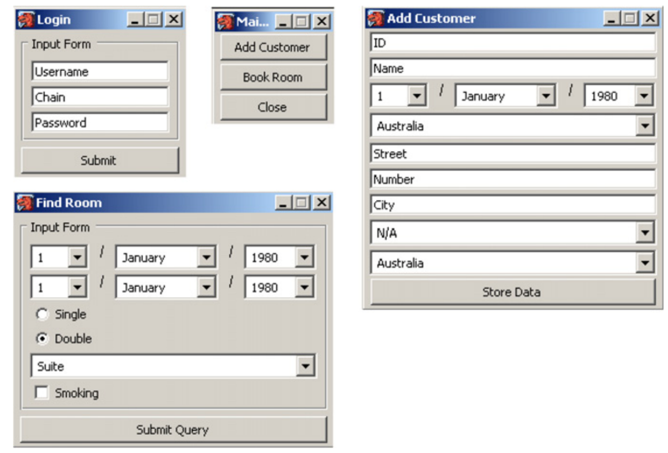


Fig. 18. Screenshots of visualized XUL fragments from the presentation model.

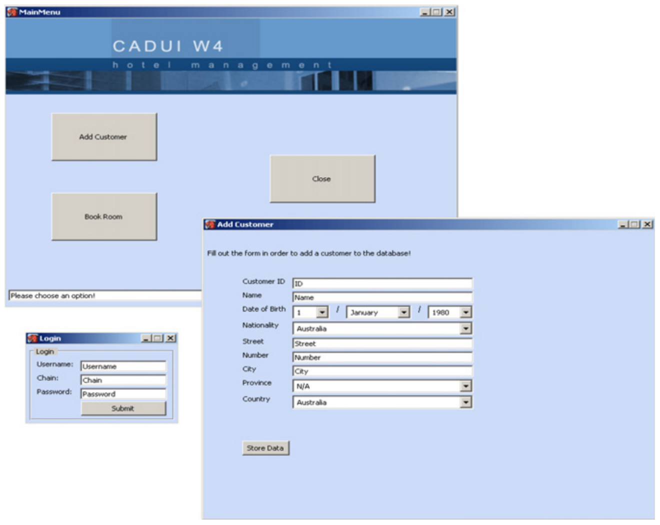


Fig. 19. Screenshots from the hotel management application.

should be chosen so that the user has the impression that all windows of the application share a consistent presentation and appear to belong together. Cascading style sheets have been used to control the visual appearance of the interface. In addition, to assist the user when working with the application, meaningful labels have been provided. The *Labeling Layout Pattern* suggests adding labels for each interaction element. Using the grid format, the labels are aligned to the left of the interaction element.

The layout model determines how the loosely connected XUL fragments are aggregated according to an overall floor plan. In the case of this example, this is fairly straightforward since the UI is not nested and consists of a single container. After establishing the layout model, the aggregated XUL code can be rendered together with the corresponding XUL skins as the final user interface. Fig. 19 shows the final UI rendered on Windows XP platform.

8. Conclusion

In this paper, we demonstrated how patterns could be delivered and applied within be model-based UI development approaches. Within our proposed framework PD-MBUI, patterns were introduced to overcome the lack of reuse in model construction and transformation. This represents one of the major limitations of existing model-based UI development frameworks. In particular, we illustrated how different kinds of patterns can be used as building blocks for the establishment of task, dialog, presentation and layout model. In order to foster reuse, we proposed a general process of pattern application, in which patterns are seen as abstractions that must be instantiated. In addition, we described an interface for combining patterns and a possible model-based formalization. The applicability of the proposed pattern-driven model-based development approach has been demonstrated through a comprehensive case study. Furthermore, we introduced the Task-Pattern Wizard tool for using, selecting, adapting and applying patterns.

A major contribution of this work is the use of patterns to support model reuse in the construction of specific models and their transformations. Traditionally, patterns are encapsulations of a solution to a common problem. In this research, we extended the pattern concept by providing an interface for patterns in order to combine them. In this vein, we proposed the general process of pattern application, in which patterns can be customized for a given context of use. We then transferred the pattern concept to the domain of model-based UI development. In order to foster reuse and avoid reinventing the wheel, we demonstrated how task, dialog, presentation and layout patterns can be used as building blocks when creating the corresponding models, which are the core constituents of our development approach.

In order to demonstrate the applicability of our approach, we developed a UI prototype for a hotel management application. In this elaborate case study patterns were identified and applied for each of the models that were used during development. The main purpose of the example is to show that model-based UI development consists of a series of model transformations, in which mappings from the abstract to the concrete models must be specified and – more importantly – automatically supported by tools.

We are currently expanding the modeling concept into an integrated pattern environment, IPE, which integrate this and other tools into a generalized pattern driven development environment that is independent of platform and programming languages. The transformation between models is automated by the use of XML as a common medium to communicate the modelling semantics between different models. This helps tailor the application and corresponding models to different platform and user roles.

Acknowledgements

This research was supported partially by a grant from the National Science and Engineering Research Consul of Canada. The first author thanks Nicole Li, Homa Javahery, Paul Ashutosh, and especially Mohamed Taleb and Daniel Sinnig all members of the Human-Centered Software Engineering group at Concordia University and who participated actively over 6 years in the development of the approach described in this paper. A special thank goes to Professor Peter Forbrig and his students from Rostosk University who participated in the development of the Task Wizard tool.

References

Alexander, C. et al., 1997. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York.

Balzert, H., 1996. From OOA to GUIs: The JANUS System. Journal of Object-Oriented Programming (February), 43–47.

Borchers, J., 2001. A Pattern Approach to Interaction Design. John Wiley & Sons, New York.

Breedvelt, I., Paternò, F., Severiins, C., 1997. Reusable Structures in Task ModelsProceedings Design, Specification, Verification of Interactive Systems '97. Springer, Granada.

da Silva, P., 2000. User Interface Declarative Models and Development Environments: A Survey in DSV-IS'2000. Springer.

Gaffar, A., Seffah, A., Van der Poll, J., 2005. HCI patterns semantics in XML: a pragmatic approach, HSSE 2005. In: Workshop on Human and Social Factors of Software Engineering, in conjunction with ICSE 2005, the 27th International Conference on Software Engineering, St. Louis, Missouri, USA, May 15–21, proceedings of ACM.

Granlund, A., Lafreniere, D., 1999. PSA: A pattern-supported approach to the user interface design process, July, Scottsdale, Arizona.

Märtin, C., 1996. Software Life Cycle Automation for Interactive Applications: The AME Design Environment. in CADUI'96. Namur University Press, Namur Belgium.

Mens, T., Lucas, C., Steyaert, P., 1998. Supporting Disciplined Reuse and Evolution of UML Models. In UML98: Beyond the Notation. Springer, Mulhouse France.

MOBI-D, 1999. The MOBI-D Interface Development Environment.

Molina, P., Meliá, S., Pastor, O., 2002. JUST-UI: A User Interface Specification Model. In: CADUI 2002, Valenciennes, France.

OliverNova, 2004. CARE Technologies.

Paquette, D., Schneider, K., 2004. Interaction Templates for Constructing User Interfaces from Task Models. In: CADUI 2004. Funchal, Protugal.

Paternò, F., 2000. Model-Based Design and Evaluation of Interactive Applications. Springer.

Prasad, S., 1996. Models for mobile computing agents. ACM Computing Survey 28 (4).

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

Rine, D.C., Nada, N., 2000. An empirical study of a software reuse reference model. Information and Software Technology, 47–65.

Schlungbaum, E., 1996. Model-Based User Interface Software Tools – Current State of Declarative Models. Technical Report 96-30, Graphics, Visualization and Usability Center Georgia Institute of Technology.

Sinnig, D., et al. 2004. Patterns in Model-Based Engineering. In: CADUI 2004, Funchal, Portugal.

TADEUS, 1998. Task-based Development of User interface software.

TERESA, 2004. Transformation Environment for Interactive Systems Representations.

Tidwell, J., 2004. UI Patterns and Techniques.

Trætteberg, H., 2002. Model-based User Interface Design in Computer and Information Sciences. Norwegian University of Science and Technology, Trondheim.

Trætteberg, H., 2004. Integrating Dialog Modeling and Application Development, in Madeira, Portugal. January.

Vanderdonckt, J., Limbourg, Q., Florins, M., 2003. Deriving the Navigational Structure of a User Interface. In: INTERACT 2003. Zurich: IOS.

Welie, M., 2004. Patterns in Interaction Design.