THE COMPLICITY OF PATTERNS AND MODEL-BASED UI DEVELOPMENT

DANIEL SINNIG

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE AT
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

MARCH 2004

Your file Votre référence
ISBN: 0-612-91116-0
Our file Notre référence
ISBN: 0-612-91116-0

ABSTRACT

The Complicity of Patterns and Model-Based UI Development

Daniel Sinnig

The main idea surrounding model-based approaches is to identify useful abstractions that highlight the main aspects of an application's design. Model-based approaches for User Interface (UI) development have the potential to accommodate the increasing complexity of today's interactive applications. However, the mainstream developer has not adopted the model-based approach for creating UIs due to certain limitations. One such limitation is the lack of reusability within such approaches. In order to foster re-use in different contexts of use, patterns are introduced as abstractions, which must be instantiated. In particular it will be demonstrated how different kinds of patterns can be used as building blocks for the establishment of 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 is suggested. A tool for using, selecting, adapting and applying patterns to task models will be presented. In addition, an extended example will illustrate the applicability of this pattern-driven development approach. Within the scope of the example, 13 different patterns relative to the various models have been discovered, formalized and applied.

ACKNOWLEDGEMENTS

The help and support of a number of people made this thesis a reality, and I am grateful to each of them:

Dr. Ahmed Seffah for his guidance and patience throughout my Master's degree. He gave me the opportunity to participate in a number of challenging and interesting projects as a member of the HCSE (Human-Centered Software Engineering) Group at Concordia University.

Homa Javahery, a person who does science with pure, unselfish and honest passion and who sacrificed countless hours of reviewing and brainstorming with me.

Dr. Peter Forbrig, my mentor, who kept an eye on the progress of my work and always was available when I needed his advice.

Andre Lehmann for his input and feedback, in addition to his advice on image creation.

Shirin Sadeghi, Matthias Werner and Svea Eckers for their encouragement and feedback.

I am very grateful for my dear princess Caroline, for her love and patience during my Master's period. She is the reason for me moving to Montreal - A decision, which I have not regretted for any second.

Above all, I am grateful to my family. I thank my parents, grand parents and my sister, Kirstin, for their active support, for believing in me, and for putting up with me!

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CSS:	Cascading Style Sheet
CTT:	ConcurTaskTrees
GUI:	Graphical User Interface
HCI:	Human Computer Interaction
MBA-UI:	Model-Based Approach for UI Engineering
MDA:	Model Driven Architecture
MUI:	Multiple User Interfaces
PIM:	Platform Independent Model
POD:	Pattern Oriented Design
PSA:	Pattern Supported Approach
PSM:	Platform Specific Model
TPML:	Task Pattern Markup Language
UCD:	User Centered Design
UI:	User Interface
UIML:	User Interface Markup Language
UML:	Unified Modeling Language
XBL:	eXtensible Bindings Language
XIML:	eXtensible Interface Markup Language
XML:	eXtensible Markup Language
XUL:	XML User Interface Language

1. Introduction

In this introductory chapter, I will highlight the problem of reusing knowledge within current model-based user interface (UI) development approaches. In addition, I will draw a research roadmap using HCI patterns as a solution. In this vein, HCI patterns act as a vehicle for capturing and disseminating best practices within the context of model-based UI development.

1.1. The Problem of Model-Based UI Development

The model-based approach has been introduced to identify high-level models in order to allow the specification and analysis of interactive systems from more semantic-oriented levels rather than dealing immediately with low-level implementation issues [Paternò 2000]. Thus, designers can focus on important conceptual aspects instead of being distracted by too many implementation details.

As a result, the increasing complexity of interactive applications can be managed more easily. Model-based approaches have the potential to establish the basic foundation for a systematic engineering methodology for UI development [da Silva 2000]. Hence it will be easier to reconstruct the UI development process in order to comprehend the system for later maintenance.

Although model-based UI design is an established field, model-based approaches for UI development are rarely used [Trætteberg 2004]. There are only a few practitioners of model-based UI design methods. The industry and the mainstream developers have not adopted the concept and perception of model-based UI development. In our opinion, one major reason exists for this phenomenon: The lack of reusability.

Unfortunately, the creation of the various models and the process of linking models to each other is a tedious, time-consuming activity. Current model-based frameworks

[TADEUS 1998; MOBI-D 1999; TERESA 2004] lack the flexibility of reusing already modeled solutions. Only few approaches offer a form of copy-and-paste reuse. However, with this kind of reuse, the "reuser" takes a copy of a model component and changes it to new requirements without maintaining any form of consistency with the original component [Mens et al. 1998]. Obviously, copy-and-paste of analysis and design model fragments is not adequate to integrate reuse in a systematic and retraceable way into the model-based UI development life cycle.

From this emerges the need for a more "disciplined" form of reuse. As I will demonstrate in this thesis, patterns can supplement current model-based approaches in order to overcome these problems.

1.2. Patterns as a Solution

A pattern is a three-part rule that expresses a relation between a certain context, a problem, and a solution [Alexander et al. 1977]. As patterns are context-sensitive, the solution encapsulated in the pattern must be instantiated (customized) to the current context of use before it can be reused.

Within the context of HCI, a pattern is a formalized description of a proven concept that expresses non-trivial solutions to a UI design problem. A pattern can be an effective way to transmit experiences about recurrent problems in the Software and UI development domain. Patterns in HCI have been introduced as a tool to capture and disseminate proven design knowledge, and to facilitate the design of more usable systems. Patterns aim to capture and communicate the best practices of user interface design with a focus on the user's experience and the context of use.

In [Erickson 2000], Thomas Erickson proposes using pattern languages as a lingua franca for the design of interactive systems. He discusses the potential of a lingua franca as a way of making communication in design a more "egalitarian process". More importantly than being a communicative tool, pattern languages guide software designers through the

design process, by providing solutions to a set of common design problems. In [Javahery 2003] patterns have been used as a working part of design, and their applicability in different contexts of use has been investigated. The "Pattern Supported Approach" (PSA) [Granlund and Lafreniere 1999] addresses patterns not only during the design phase, but also during the entire software development process. In particular, patterns have been used to describe business domains, processes, and tasks to aid early system definition and conceptual design.

As patterns also inherently promote the reuse of ideas and knowledge, few approaches have considered them as building blocks for the creation of the relevant models. In [Breedvelt et al. 1997; Paternò 2000] the idea of combining patterns with models has been introduced. In particular, patterns have been used to describe reusable task model fragments. At this time, patterns are only used to describe static model fragments. Thus, the reuse is limited to copy-and-paste.

To be an effective tool for capturing the knowledge in model-based approaches, the following issues need to be investigated:

- A classification of patterns according to models needs to be established. Such a classification should distinguish between patterns as building model blocks and patterns for driving the transformations of models.

- Tool support for helping developers select the right patterns, as well as how to apply them.

1.3. Objectives and Scope of this Thesis

The research goals addressed in this thesis are all concerned with the complicity of patterns and models for UI development. The following research objectives will be tackled:

- Enhance the model-based framework with support for knowledge reuse by integrating patterns.

- Recommend a classification for patterns according to models, and in conjunction, propose a set of selected patterns.

- Elaborate an extended example of a UI prototype, which will illustrate the usage of the framework and patterns.

- Demonstrate how tools and technologies like XML can be used for building and evaluating the various models and for applying patterns.

In particular, I will define a common model-based framework for the development of user interfaces. I will provide clear definitions for each model and outline the process of UI derivation. Furthermore, I will suggest how to enhance the framework by integrating the idea of patterns as a vehicle for reuse. More specifically, I will demonstrate how and which patterns can be used as building blocks for the various models. As my model-based framework is based on transformations, I will also show how patterns can help transform one model into another.

Based on the current state of the art, I will further investigate, from a practical standpoint, the applicability of patterns within the context of model-based UI development. I will suggest a classification of patterns according to models and propose improvements for the documentation of patterns that are necessary in order to use and apply them efficiently. In this vein, I will propose a possible formalization and an interface for combining patterns. In order to exemplify their application, I will compile a set of patterns that will be further used to elaborate an extended example.

1.4. Research Methodology

In order to validate and illustrate the applicability of my approach and the proposed list of patterns, I will develop a brief case study of a hotel management application. Step by step, I will show how to establish and transform models using patterns. Starting from a task model, a non-functional UI prototype will be gradually derived.

Throughout my thesis, I will refer to complementary tools, which are useful for establishing and evaluating models. More specifically, I will introduce the Task-Pattern-Wizard, which I have developed in order to apply patterns to task models. It interactively supports the adaptation and integration of task and feature patterns. As a result of this, I will show how the various models can be formalized using derivates of XML-based descriptions.

The major objective of this thesis is to demonstrate how HCI patterns can be combined and customized in order to be vehicles for a more advanced form of reuse. This thesis will illustrate that patterns can be found and formalized for different types of models throughout the UI development process. Due to resource constraints, this approach will not be validated in an industry context and / or via a large empirical study. However, as a preliminary step, I have carried out an extended example to demonstrate how patterns can act as a driving force within model-based UI development approaches. For the scope of this thesis, it is assumed that all proposed patterns are "valid" patterns and are applicable.

1.5. Organization of the Thesis

The organization of the thesis is as follows:

In **Chapter 2** the main features of **"Model-Based UI Development"** will be summarized. Definitions of the most common models will be provided and a review of the most influential model-based frameworks will be given. Eventually, I will discuss the

limitations and advantages of model-based approaches for the development of interactive applications.

Chapter 3 "Patterns in HCI" traces the evolution of patterns and reflects on the current state of the art of patterns and pattern languages. I will show how patterns have been used in the design of interactive applications and why patterns are an ideal vehicle for fostering reuse within model-based development approaches.

Chapter 4 "The Pattern-Driven Model-Based Framework" introduces patterns as a driving force for model-based UI development. An interface for combining patterns will be provided and the general process of pattern application will be introduced. Then, an in-depth description of every phase of my development approach will show in great detail the establishment of each model and the influence of different kinds of patterns. Basically, I will discuss patterns for the task, dialog, presentation and layout model. Additionally the tool Task-Pattern-Wizard is introduced for selecting and applying patterns to task models.

In **Chapter 5 "Illustrative Example: Developing the UI of a Hotel Management Application"**, the practical applicability of my approach will be illustrated by elaborating on an extended example. I will introduce different tools and technologies and I will employ 13 different patterns to develop a non-functional UI prototype of a simplified hotel management application.

Finally, **Chapter 6** summarizes my work and presents future avenues for research in this area.

2. Model-Based UI Development

2.1 Definitions and Advantages of Model-Based UI Development

The main idea of model-based approaches is to identify useful abstractions highlighting the main aspects that should be considered when designing interactive applications [Marucci et al. 2003]. Several models are created and combined to characterize a domain of interest from different perspectives [Forbrig et al. 2003a].

2.1.1. Definition of Model-Based UI Development

Central to all model-based approaches is that all aspects of a user interface design are represented using declarative models. The central component is the Interface Model, which includes different declarative models [Schlungbaum 1996]. A series of declarative models, such as user-task, dialog, and presentation, are interrelated to provide a formal representation of an interface design [Puerta 1997].

In a model-based approach, the UI design is the process of creating and refining the user interface models [da Silva 2000]. In other words, model-based design focuses on finding mapping between the various models [Vanderdonckt et al. 2003b].

Eventually, User Interfaces are generated automatically or semi-automatically from the model descriptions.

2.1.2. Advantages of Model-Based UI Development

Initially it may seem that following a model-based approach for the design of interactive applications may complicate and slow down the development process. However, the benefits to be gained are considerable. In essence, model-based UI development has two major advantages:

- **Design decisions** are made **at conceptual levels** (i.e. designing the envisioned task model). Designers can specify and analyze interactive software applications from a more semantic-oriented level rather than starting immediately to address the implementation level.

- Following a systematic and repeatable development approach makes it **easier to** reconstruct the process in order to **comprehend the system** for later maintenance.

Design Decisions are made at the conceptual level:

Historically, User Interface (UI) development has been treated as a creative design activity rather than a systematic engineering process. However, with the advent of pervasive computing and mobile users, the design and development of UIs has become increasingly complex. Thus, UIs must be aware of dynamically changing contexts and withstand variations of the environment. From this emerges the need for a structured engineering-like development approach. Model-based approaches have the potential to establish the basic foundation for a systematic engineering methodology for UI development.

Model-based specifications serve as a high-level sophisticated model of the application semantics. Within a model-based development approach, requirements and context of use are captured by high-level models. Using these models as a starting point, various models are designed which describe the different facets of the user interface. Thus, first design decisions are made at an abstract level where concrete implementation issues are omitted. The designer can focus on semantically relevant aspects while specifying the interactive application rather than dealing with implementation details.

One example of the increasing design complexity is applications, which provide multiple user interfaces. Multiple user interfaces allow a user to interact using various kinds of computing platforms including traditional desktop PCs, PDAs or mobile phones [Seffah and Javahery 2003]. The user interface can be specified once and then refined for each target device until reaching the implementation level. Model-based approaches do have

the potential to cope with this new challenge. Several models are employed to describe the context (platform, environment, user) and the requirements of the user interface. In order to adapt to different platforms, task models play a key role. Some tasks may only be meaningful on specific platforms. Therefore Paternò [Paternò and Santoro 2002] suggested augmenting tasks with information about the device capabilities, which are necessary for its execution.

Ideally, tools are used to generate the implemented user interface from the corresponding models or to update an existing implementation in order to maintain the consistency to the higher-level specification. As a consequence of using models, which capture semantically meaningful aspects, the increasing complexity of interactive applications becomes more easily manageable.

Comprehend the system for later maintenance:

It is an inevitable fact that software ages over time [Parnas 1994]. One reason for this phenomenon is the dynamic environment that continuously changes. As a result, at a certain point, the software does not meet the user's needs anymore. In addition, the necessary changes and modifications consecutively corrupt the structure of the code and the architecture of the system. Often the changes are done (quick and dirty) at the implementation level only, without updating the high level model and specifications. As a consequence, the software becomes harder to maintain.

Moore [Moore 1996] pointed out that the cost of software maintenance has been steadily increasing over the last two decades, and in the long run, maintenance activities account for between 70 and 80 percent of the software lifecycle today. Nearly 50 percent of the code of the system is devoted to the UI. A large portion of the maintenance tasks consists of reengineering and migrating systems to new platforms and architectures. In this case, the UI code is very likely to change drastically and often needs to be completely replaced.

From this emerges the need for tool support and techniques that help in maintaining the software. In the case of UI development, we envision that the final implementation can be automatically derived from the models that describe the different facets of the UI. In order to maintain or update the UI, modifications can be done at the model level, which is rather abstract and omits implementation details. As a result, the problem space to be comprehended is less complex (see previous section).

In the remainder of this chapter I will describe the main ingredients of model-based UI development: The various models, the concept of model transformation and the different model notations. In addition, I will briefly outline the most influential frameworks and I will discuss the major limitations of model-based approaches.

2.2. The Different Models

Many facets as well as related models exist in order to describe the User Interface. Up to now, there is no agreement on which set of models are the best for describing UIs in a declarative manner [da Silva 2000]. Different model-based approaches use different declarative models [Schlungbaum and Elwert 1996; Puerta 1997; Vanderdonckt et al. 2003a]. Henceforth, in what follows, I will only define the most frequently used models: user task, user, domain, environment, platform, dialog and presentation models. These models have been named differently in different frameworks. In addition, we should note that some of them overlap (as illustrated in Figure 2-1).

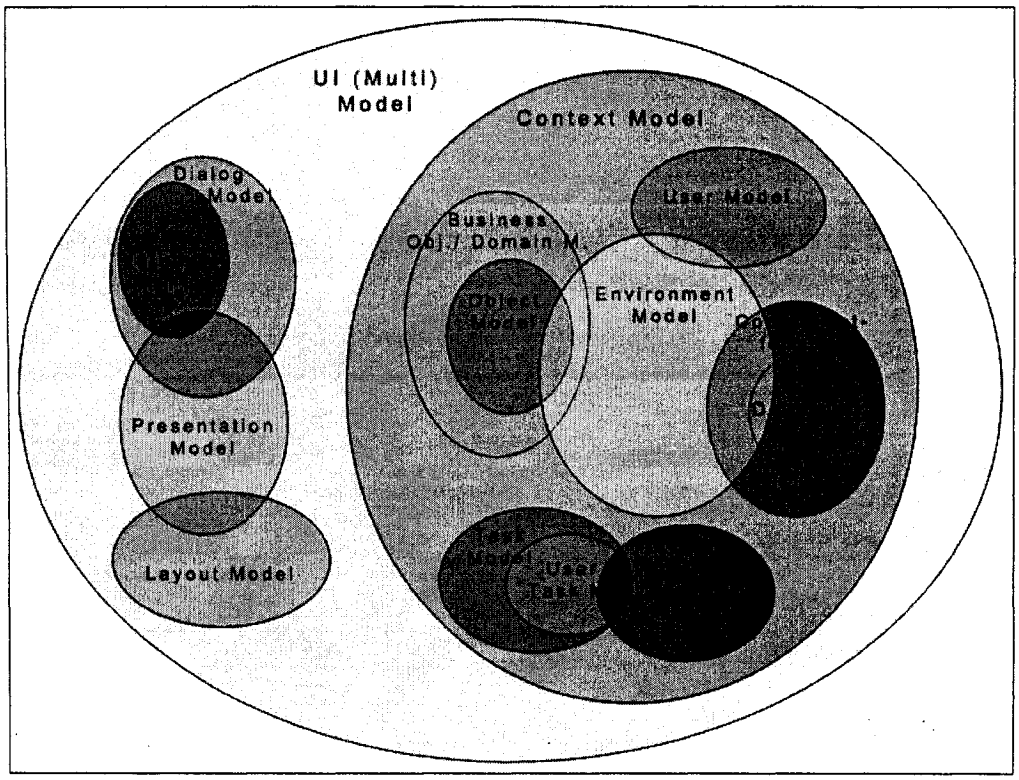

Figure 2-1: The Model Pie

User Task Model:

With respect to the design of interactive systems, "among all these models, the task model has (today) gained much attention and many acceptances in the scientific community to be the model from which (a) development should be initiated" [Forbrig et al. 2003b]. The user task model specifies what the user does, or wants to do, and why. It describes the tasks that users perform using the application, as well as how the tasks are related to each other. In other words, it captures the user task and system's behavior with respect to the task-set. In essence, the system will be viewed through the set of tasks performed by the user, which create input to and output from, the system.

User Model:

The User model captures the essence of the user's static and dynamic characteristics. It is a widely studied field and we adopt a suitable user model in our research. We can usually model the user knowledge or the user preferences [Chin 1986]. Modeling the user background knowledge is useful for personalizing the format of the information (e.g. using an appropriate language understood by the user). Modeling the user preferences is useful for personalizing the content of the interface (e.g. by filtering the results of a database query, or as an aid to a software agent that proactively notifies the user about interesting information). We can further imagine creating a user model for each type or each individual user, or just one user model for the canonical, typical user.

Domain (Object) Model:

The object model encapsulates the important entities of the particular application domain together with their attributes, methods and relationships [Schlungbaum 1996]. In particular, it captures concepts, objects and operations describing the domain. Within the scope of UI development, it defines the objects that the user accesses via the interface.

Environmental Model:

The environmental model specifies the physical and cultural context of interaction [Trætteberg 2002]. It consists of a collection of physical variables that form the input/output of the system. For example, in the case of a mobile user application, the environment model would include such variables as: The current location of the user, the constraints and characteristics of the location, the time, and any trigger conditions specified or implied by the virtue of the location type.

Platform (Device) Model:

A platform model describes the various computer systems that may run a UI [Prasad 1996]. This includes information regarding the constraints placed on the UI by the platform. The platform model describes the physical characteristics of the target platforms. For example, the input and output capabilities of the target devices.

Dialog Model:

The dialog model captures the structure of the "conversation" between human and computer [Trætteberg 2002]. It 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, it specifies the user commands, interaction techniques, interface responses and command sequences that the interface allows during user sessions. It must encompass all static and dynamic information the user needs for the dialog with the machine.

The Presentation Model:

A presentation model describes the visual appearance of the user interface. It describes the constructs that can appear on an end user's display, their layout characteristics, and the visual dependencies among them [Schlungbaum 1996]. The presentation model exists at levels of abstraction: The abstract and the concrete presentation model. The first level provides an abstract view of a generic interface, which represents a corresponding task and dialog model. The second level (within this thesis, called "Layout Model") is realized as a concrete instance of an interface, which can be presented to a user; there may be many concrete instances of an abstract presentation model.

The UI Model (UI Multi Model):

An interface model is a knowledge base that consists of a series of representations (component models) capturing the various facets of the UI. It is constituted out of the various peer models introduced above.

The user interface on the one hand is described by the information captured by the various models but also by the relationships, which can be established between different models. For example Thevenin [Thevenin and Coutaz 1999] suggested decorating the task model with information about the target user, whereas Marucci [Marucci et al. 2003] attaches information about target platform to the tasks. In [Vanderdonckt et al. 2002], the concept of context is partitioned into three models: A user model, platform model and environment model. Based on this, a Multi-Context Task Model is suggested which takes into account all three dimensions of context. It is important to note that this kind of conceptualization is of particular importance for the development of multiple user interfaces [Seffah and Javahery 2003].

2.3. Model Transformations

Forward engineering is the traditional process from high-level abstractions and logical, implementation independent design to the physical implementation of the system [Chikofsky and Cross 1990]. In software engineering, transformational development aims at developing applications by transforming a coarse grained specification to final code through a sequence of small transformations [Limbourg and Vanderdonckt 2004]. Within the scope of Model-based UI design, the development process consists of a series of transformations and mappings between models.

Different models have different levels of abstraction. Early stages of development of the UI focus on general, implementation-independent concepts, which are described at a rather high level or with more abstract models. Later stages of software engineering emphasize implementation details using low-level or less abstract models. Low-level

models must take into account platform-specific attributes, whereas high-level models are rather platform independent. Changes made at the conceptual modeling stage (high-level model) will affect all lower levels [Chikofsky and Cross 1990]. In particular the creation of a low-level model depends on the information modeled by the models at a higher level of abstraction. Therefore it is important to define the dependencies or mappings between models at different levels of abstraction. On one hand, there are purely abstract models, such as the user task (e.g., "get customer's name"), domain or user models. On the other hand, there are concrete models, such as the dialog and presentation models, which reveal details of the interface implementation and visual appearance (i.e. window transitions or the placement of buttons and labels).

In the following, I will discuss two essential transformation strategies in UI design: The transformation from the user task model to the dialog model, and the transformation from the domain model to the presentation model.

Transforming the Task Model to the Dialog Model:

The dialog model is closely connected to the underlying task model. Amongst other things, tasks will be grouped into dialog views and transitions between the various dialog views are defined [Forbrig et al. 2003b]. Mainly, two types of information from the task model constrain and influence the dialog model. On one hand, structural information from the task model, which describes the task–subtask hierarchy, influences the creation of the dialog views. On the other hand, temporal transitions between sub tasks can be used to constrain and derive possible dialog transitions [Sinnig et al. 2004].

Most model-based development approaches define the dialog model though a task model. Information from the task model is exploited, in order to automatically or interactively derive the navigational structure of the application.

In MOBI-D, based on the structural information from the task model, a design assistant allows the designer to specify a preference range on how the tasks should be grouped to

Windows. This can range from grouping all tasks into one "super window" to assigning each task its own window [Puerta and Eisenstein 1999].

In TERESA, structural information, as well as temporal relations, are exploited in order to generate a so-called activation set. This set is then further used to automatically generate the dialog model and parts of the presentation model (grouping of presentation elements).

Transforming the Domain Model to the Presentation Model:

Elements in a domain model possess attributes that are often relevant to presentation element selection. Examples for these attributes are data type, range, minimum and maximum value. During the transformation process, mappings are established which define, for example, what widget should be used to display the value of an integer-type object.

In MOBI-D, the TIMM facility is used to inspect and set domain-to-presentation mappings. By selecting an object in the domain model, developers can view and modify what interactors are appropriate to display and access that particular domain object. Eventually, the interactors of the presentation model will be set according to these predefined mappings.

2.4. Model Notations

In Section 2.3, the most common models have been discussed. These models are present to describe the different facets of the interface. However, different notations can be used to represent the same model. In this section, I will outline and discuss the most influential notations / languages that can be used. In particular, I will summarize the main features of CTT [Paternò 2000] for task modeling and DiaMODL for dialog modeling [Trætteberg 2002]. I will also discuss XUL [XUL 2004a] and UIML [UIML 2003] as mediums for describing the presentation model. Eventually, I will discuss XIML [XIML 2003] as a universal notation for different kinds of models. Using such a standard notation may be

useful in order to describe different user interface models using a common set of constructors.

2.4.1. The ConcurTaskTree Notation

CTT is a well-accepted notation in the UI field and is used for the specification of task models. The notation is supported by the semantics of the formal language LOTOS [ISO_8807 1988]. The notation defines four categories of tasks:

1. **User Tasks:** Tasks performed by the user. Usually, they are cognitive activities like thinking of a strategy to solve a problem. Often, the goal of the envisioned interactive application is to replace these tasks by interactive tasks or application tasks. Thus, normally, the envisioned task model barely contains any user tasks.

2. **Application Tasks:** Tasks performed by the application. For example, presenting results to the user.

3. **Interaction Tasks:** Tasks performed by the user while interacting with the system. For example, entering values into a form.

4. **Abstract Tasks:** Tasks which require complex activities or that could be decomposed into simpler ones.

The graphical representation for each category of tasks is shown in Figure 2-2

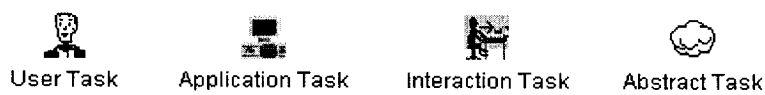

Figure 2-2: Different Task Types

A task can be decomposed into subtasks, which are placed into relations by means of temporal relationships. The ConcurTaskTree notation provides a set of temporal operators based on the semantics of LOTOS. Table 2-1 provides a description of each CTT operator.

Table 2-1 Temporal Relationships of CTT

Temporal Relationship	**Description**
Choice (T1 [] T2)	It is possible to choose from a set of tasks. Once a task is selected, the other tasks in the set are not available until the selected task is completed.
Concurrency – Independent (T1 ||| T2)	Actions belonging to two tasks can be performed in any order.
Concurrency with Information Exchange (T1 |[]| T2)	Two tasks can be executed concurrently but they must synchronize in order to exchange information.
Deactivation (T1 [> T2)	The first task is definitely deactivated once the first action of the second task has been performed.
Enabling (T1 >> T2)	In this case T1 enables T2 when it terminates.
Enabling with Information Exchange (T1 []>> T2)	In this case T1 additionally provides information to T2, in addition to enabling it.
Iteration T*	The task can be executed repetitively. When the task ends it can be executed again.
Iteration – Finite	It is used when designers know in advance how many times a task

(T1 (n))	will be performed.
Optional Tasks ([T])	The task may be executed or not.
Recursion	A sub tree originating from a certain task contains another occurrence of this task.
Suspend-resume (T1 |> T2)	This operator gives T2 the possibility of interrupting T1 and then when T2 is terminated, T1 can be reactivated from the state reached before the interruption.

2.4.2. DiaMODL

DiaMODL is a dialog modeling language based on the Pisa interactor abstraction and UML Statecharts. DiaMODL is capable of modeling the dataflow as well as the behavior of interaction objects. It may be used for documenting the function and structure of concrete UIs, abstract specification of UI functionality, and for systematic exploration of design alternatives [Molina and Trætteberg 2004].

In particular, the concept of interactors is used to describe the functionality and the behavior of interaction objects. An interactor receives and sends information through a set of gates, each consisting of a base and a tip. Figure 2-3 portrays the generic interactor, which has 4 different gates:

1. **Input/send** (is): Input from the user results in information being sent out of the interactor towards the system.

2. **Output/receive** (or): Output from the system is received by the interactor, which is responsible for outputting it to the user.

3. **Input/receive** (ir): User input is received by the interactor for further processing/meditation.

4. **Output/send** (os): Output to the user is sent out of the interactor.

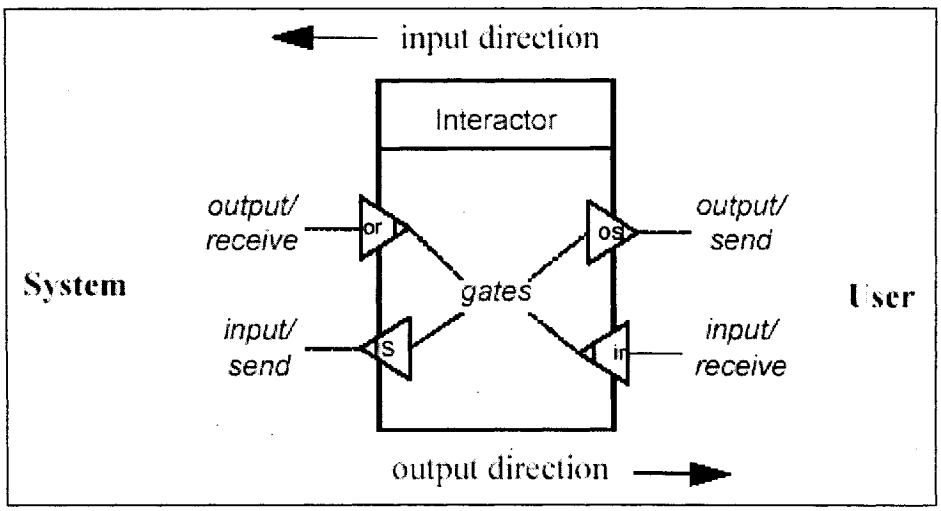

Figure 2-3: Generic Interactor [Trætteberg 2002]

Please note that interactors can be composed and aggregated into networks of connected interactors, resulting in a new interactor.

The activation and deactivation of interactors and the information flow is modeled using UML statecharts. An interactor can be considered as a composite Statechart state, which is entered to activate the interactor and exited to deactivate it. As a composite state, each interactor may be decomposed into a hierarchy of substates, using and- or or-decomposition. Substates can be connected by transitions to control activation and deactivation. In addition, transitions may be labeled with triggering events, conditions that control when it can be triggered, and actions that are performed when the transition is followed [Trætteberg 2002].

2.4.3. User Interface Markup Language (UIML)

UIML is a meta-language that allows the developer to describe the UI in generic terms but also to use style descriptions to map the UI to various target platforms. It was developed to address the need for a uniform UI description language for building multi-platform applications.

A UIML document contains three different parts: a UI description, a peers section that defines mappings from the UIML document to external entities (rendering to the target platform and application logic), and finally a template section that allows the reuse of written elements. The UI description specifies a set of interface elements with which the end user interacts. For each element, a presentation style is defined (e.g. position, font, color) along with its content, possible user input events, and resulting actions.

Eventually, the UI description can be rendered to the corresponding target platform, resulting in a functional UIML that provides a uniform language to describe UIs for different target platforms. However, the creation of UIs for different target platforms, from a single specification is not possible. There is still a need to design separate UIs for each device.

2.4.4. Extensible User Interface Language (XUL)

XUL is an official Mozilla initiative, and provides an XML-based language for describing window layout. The goal of XUL is to build cross platform applications, which are easily portable to all of the operating systems on which Mozilla runs. XUL provides a clear separation between the user interface definition (the various widgets that determine the UI) and its visual appearance (the layout and the "look and feel"). A UI is described as a set of structured interface elements along with a set of predefined attributes. Event handlers and scripts can be defined in order to allow interaction with the user. In order to extend XUL, XBL (eXtensible Bindings Language) [XBL 2004] can be used to define

new elements and XUL widgets. In addition, it is possible to integrate external libraries (i.e. written in C/C++ or JavaScript) using the XPCOM / XPConnect interfaces.

However, XUL has its focus on window-based user interfaces. This focus is also a limitation. At the moment XUL specifications cannot be rendered to multiple user interfaces, including small mobile devices [Souchon and Vanderdonckt 2003].

2.4.5. The eXtensible Interface Markup Language (XIML)

XIML is the follower of MIMIC [Puerta and Maulsby 1997] and provides a way to describe the UI without worrying about its implementation [Souchon and Vanderdonckt 2003]. The goal of XIML is to describe the various abstract (task, domain, user) and concrete (presentation and dialog) aspects of the UI throughout the development lifecycle. In addition, XIML supports the definition of mapping from the abstract to concrete elements [Puerta and Eisenstein 1999].

Figure 2-4 illustrates the basic structure of XIML. Practically, it is a hierarchical organized set of interface elements that are distributed into one or more interface components. Please note that XIML uses the term component instead of model. XIML predefines five basic interface components:

- **Task component**: Captures the business process and/or user tasks that the interface supports
- **Domain component**: Comprises a set of all objects and classes used.
- **User component**: Captures the characteristics of the users that use the application
- **Dialog component**: Specifies the UI interaction
- **Presentation component**: Defines a hierarchy of concrete interaction objects

However, the language does not limit the number and types of components and elements. XIML can be extended in order to accommodate customized or new interface components.

Besides the interface components, a XIML description consists also of attributes and relations. On one hand, an attribute is a feature or property that has a value and belongs to an element. On the other hand, a relation is used in order to link one or more elements together, within the same component or across several components.

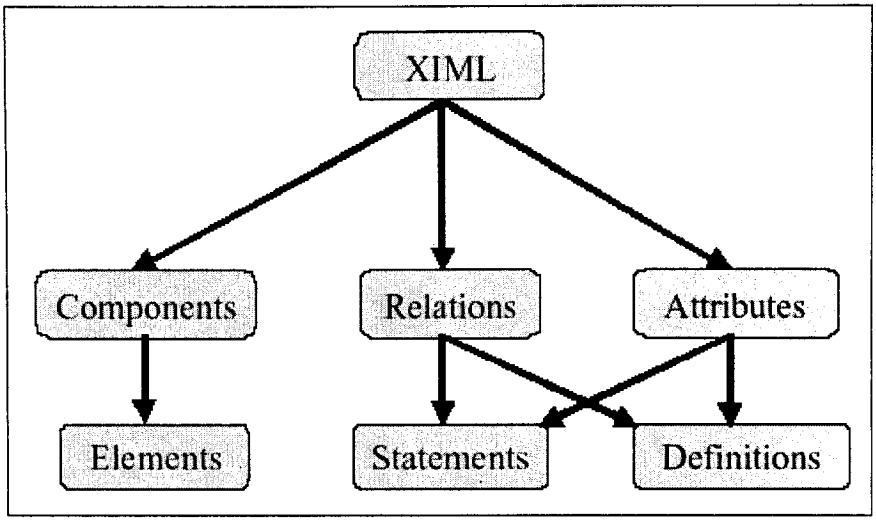

Figure 2-4: The Basic Structure of XIML [XIML 2003]

XIML has mainly been introduced to standardize the representation of the different models in order to act as a universal exchange format and to foster the interoperability of applications. However, currently, there are only a few tools (Vaquita [Bouillon and Vanderdonckt 2002], XIML-Task-Simulator [Forbrig et al. 2003b], Dialog-Graph-Editor [Forbrig et al. 2003b]) which employ or use XIML. Therefore, there are no tools presently available which are capable of rendering a XIML description to a user interface. In addition the extensibility of XIML is also its limitation. By defining new XIML Components, the interoperability cannot be ensured. If portability is needed, XIML Components may need to be limited to the boundaries that are predefined.

2.5. Existing Model-Based Frameworks

Model-based UI development has been investigated for more than a decade. Many groups and individuals have devoted themselves to the development of model-based frameworks. This section gives an overview of the most current and influential approaches. It is to be

noted that instead of automation (JANUS [Balzert 1996], AME [Märtin 1996]), "modern" model-based systems (MOBI-D [MOBI-D 1999], TERESA [TERESA 2004]) provide tools that allow developers to interactively define mappings between the various models.

2.5.1. Model-Based Interface Designer (MOBI-D)

MOBI-D (Model-Based Interface Designer) is a software environment for the design and development of user interfaces from declarative interface models. The development of the MOBI-D modeling framework and the corresponding tool environment is based on the experience gained from developing the MECANO system [Puerta 1996]. While the MECANO system only used a domain model to generate the final user interface, the MOBI-D system supports the user interface developer to design the user interface through specifying task, domain, user, dialog, and presentation models. Internally, the MIMIC modeling language is used to define the various models. MIMIC is the forerunner of XIML and supports the declaration of a so-called design model. This design model describes dynamic relationships between entities of the other declarative models.

Figure 2-5 shows the MOBI-D development cycle. The cycle is of an iterative nature. The interface design begins with the elicitation of the user tasks. Next, the developer takes this description and builds user-task and domain models. The models are then integrated so that domain objects are related to the user tasks for which they are relevant. The decision support mechanisms in MOBI-D use the user-task and domain models to make recommendations for presentation and interaction techniques. These recommendations are displayed to the developer to guide the design and ensure that all task and data elements are embodied in the interface. MOBI-D guides the developer through the selection and layout of interface components, providing for each subtask a choice of suitable components pre-configured for the task data. The resulting interface is tested by the end-user.

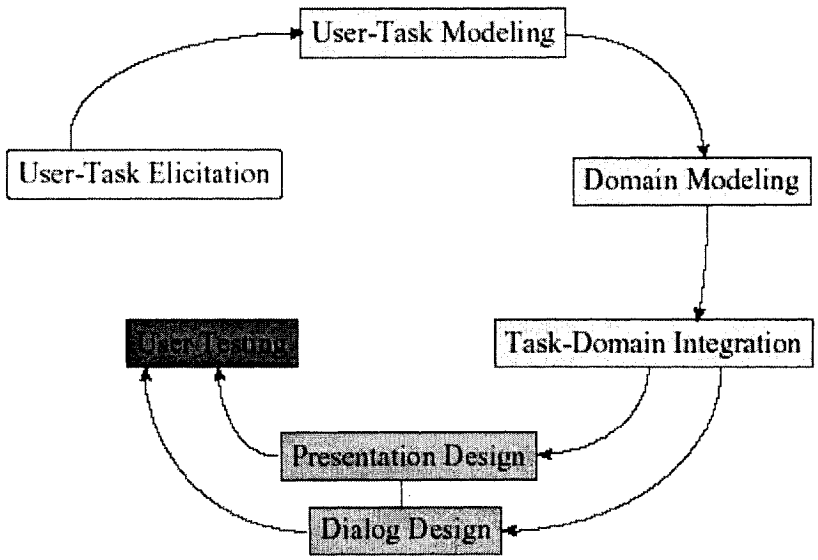

Figure 2-5: The MOBI-D Framework [MOBI-D 1999]

Instead of automatic generation of the UI (like in MECANO) MOBI-D emphasizes the support of interactive UI design through the management of design knowledge. The designer has more control and is more or less guided by the MOBI-D tools set.

2.5.2. TERESA

TERESA is a semi-automatic environment developed to generate the concrete user interface for a specific type of platform based on the task model. It is composed of a number of steps that allows designers to start with an envisioned overall task model of a multiple user interface application [Marucci et al. 2003] and then derive concrete and effective user interfaces for multiple devices. In particular, TERESA distinguishes four main steps:

1. **High-level task modeling of a multi-platform application:** In this phase, a single task model is developed, addressing possible contexts of use and roles involved.

Within the task model, the objects that have to be manipulated to perform tasks are also specified using the CTT notation.

2. **Developing the system task model for the different platforms considered**. Here, the task model is filtered and refined resulting in the system task model for one specific platform.

3. **From system task model to abstract user interface**. The goal of this phase is to obtain an abstract description of the user interface composed of a set of abstract interactors. The interactors are identified through an analysis of the task relations.

4. **User interface generation**. This phase is completely platform-dependent and has to consider the specific properties of the target device: Every interactor is mapped into interaction techniques supported by the particular device (operating system, toolkit, etc.)

The TERESA framework and the corresponding tool set allow designers to provide the results of conceptual analysis in terms of tasks and their relations. In addition the UI generation is supported while taking into account the characteristics of the platform considered. Rather than automatic generation, the UI derivation in TERESA is highly interactive, in which designers have different levels of control over the development process.

2.5.3. AME and JANUS

The AME and JANUS systems are very similar from the point of view of declarative models. Both systems emphasize the automatic generation of the desired user interface from an extended object-oriented domain model. During this automatic generation process, both systems make use of different comprehensive knowledge bases. They do not include any other declarative models. Both systems generate user interface code for

different target systems like C++ source code in order to link it with UI toolkits [Schlungbaum 1996].

2.6. Related Investigations on Model Driven Architecture in Software Engineering

In the last five years, the software engineering community introduced the concept of Model Driven Architecture which we find, in terms of goals, has some similarities with the model-based approach in UI engineering (MBA-UI). In this section we briefly introduce the MDA, while comparing and contrasting it with MBA-UI.

2.6.1. Basic Foundations of MDA

The Object Management Group introduced the Model-Driven Architecture (MDA) initiative as an approach to system specification and interoperability based on the use of formal models [D'Souza 2001; MDA 2004b; MDA 2004a]. The main idea of MDA is to specify business logic in the form of abstract models. These models are then mapped (partly automatically) according to a set of transformation rules to different platforms. The models are usually described by UML in a formalized manner, which can be used as input for tools which perform the transformation process.

The main benefit of MDA is the clear separation of the fundamental logic behind a specification from the specifics of the particular middleware that implements it. In other words, the MDA approach distinguishes between the specifications of the operation of a system from the details of the way that system uses the capabilities of its platform. This architectural separation of concerns constitutes the basic foundation of MDA in order to reach three main goals: portability, interoperability and reusability [MDA 2004b].

The MDA approach is comprised of three main steps:

- Specifying the system independently from the platform that supports it
- Specifying target platforms
- Transforming the system specification into a specification for a particular platform

In what follows I will provide brief descriptions about each phase:

Specifying the system:

In this phase a *platform independent model* (PIM) is established. Usually a formalized UML notation is used to specify the PIM. It describes the system, but does not show details of its use or its platform. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type.

Specifying the platform:

A *platform model* provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides, for use in a platform specific model, concepts representing the different kinds of elements to be used in specifying the use of the platform by an application. The architect will then choose a platform (or several) that enables implementation of the system with the desired architectural qualities.

Transforming the system specification into a specification for a particular platform:

In this phase the platform independent model (PIM) will be transformed into a platform specific model (PSM) according to some mapping rules. In particular, an MDA mapping provides specifications for transformation of a PIM into a PSM for a particular platform. The platform model will determine the nature of the mapping. A mapping may also

include templates, which are parameterized models that specify particular kinds of transformations. These templates are like design patterns, but may include much more specific specifications to guide the transformation. Templates can be used in rules for transforming a pattern of model elements in a model-type mapping into another pattern of model elements.

A *platform specific model* is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform. A PSM may provide more or less detail, depending on its purpose. Eventually, if the PSM provides all the information needed to construct a system and to put it into operation, it is used for the implementation of the system.

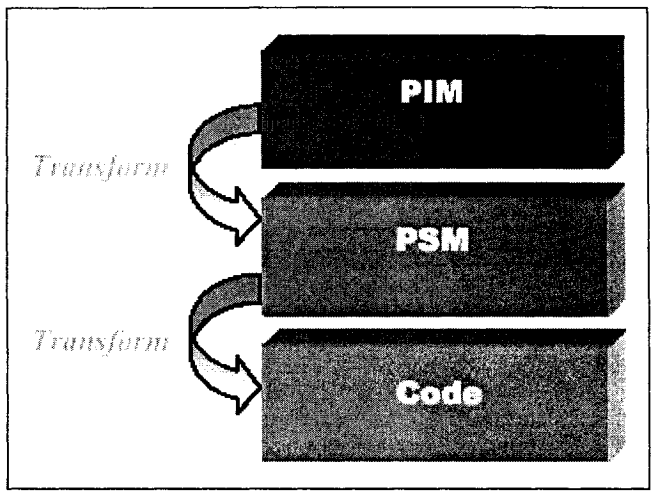

Figure 2-6 MDA Transformations

Figure 2-6 portrays the transformation from a PIM to PSM and eventually the implementation code of the system.

For a very long time, interoperability had been based mostly on CORBA standards and services. Heterogeneous software systems inter-operate at the level of standard component interfaces. The MDA process, on the other hand, places formal system models at the core

of the interoperability problem. What is most significant about this approach is the independence of the system specification from the implementation technology or platform. As illustrated in Figure 2-7, the system definition exists independently of any implementation model and has formal mappings to many possible platform infrastructures such as CORBA PSM, EJB JAVA PSM, and SOAP PSM. After establishing the PSM, it must be implemented on the specific target platform.

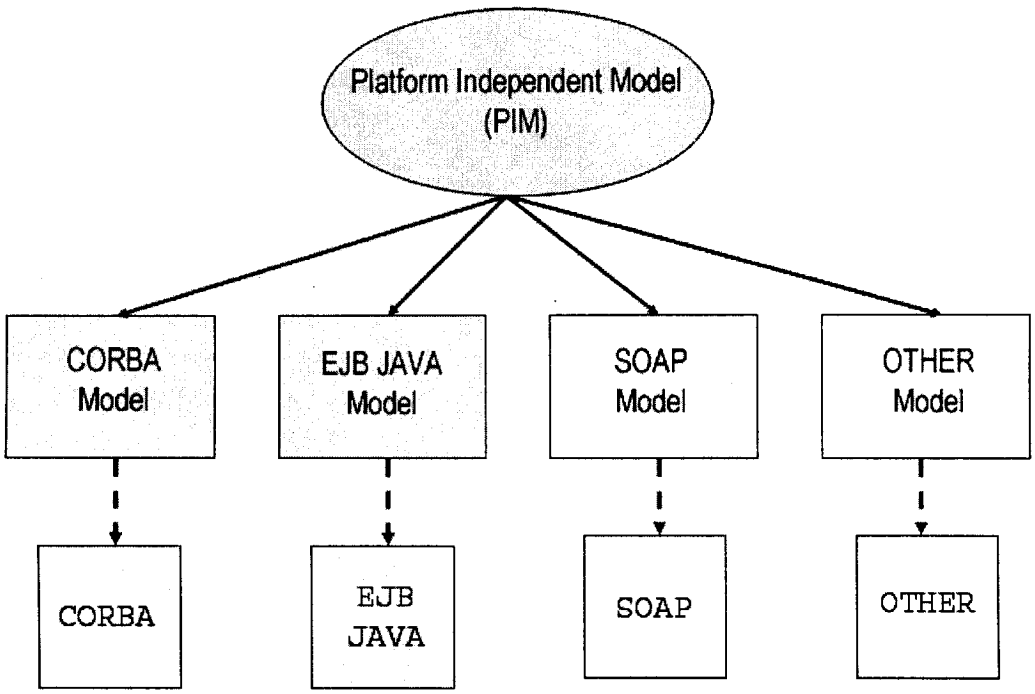

Figure 2-7 PIM, PSM and Implementation

2.6.2. MDA vs. MBA-UI: A Comparison

In the object-oriented community, the MDA is supported by a number of important OMG standards:

- The Unified Modeling Language (UML) to specify the PIM

- Meta Object Facility (MOF) to specify meta models, which can serve as a template for a PIM

- XML Metadata Interchange (XMI) as an XML formalization of a PIM

- Query, View, Transformation (QVT) is a specification to transform one model into another.

These standards define the core infrastructure of the MDA, and have greatly contributed to the current state-of-the-art of systems modeling [MDA 2004a]. These standards for MDA are what XIML and UIML are becoming for model-based approaches (MBA) in UI engineering. However, while the central focus of MBA-UI is usability, the MDA represents a major evolutionary step in the way the OO community, and in particular OMG, defines interoperability standards.

In what follows I will discuss to what extent MDA principles have been adopted for the model-based approach in UI engineering. In addition I will analyze if UML notations are suitable for the specification of interactive applications.

Similarities between MDA and MBA-UI:

In the same manner as MDA distinguishes between platform independent UML models and platform dependent UML models, models involved in the model-based UI development can also be separated. Within the context of multi-platform applications, Marucci [Marucci et al. 2003] suggested the following three steps for the development of interactive applications with Multiple User Interfaces:

1. Developing an overall envisioned task model
2. Developing the system task model for the different platforms
3. Deriving the abstract user interface from the system task model

In what follows I will describe each phase in more detail and I try to draw comparisons to the MDA approach.

Generic envisioned task model:

In this phase, the designer defines the logical activities to be supported and the relationships among them. The most abstract model is the general task model for the problem domain. It is more or less for analysis reasons only. Based on this analysis model, an envisioned task model considering the future support of the interactive application under development will be designed. This model addresses the software support in various contexts of use and for different user roles.

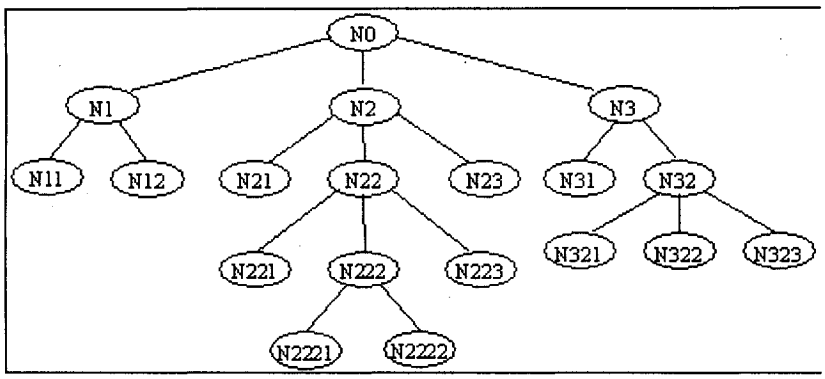

Figure 2-8: Generic Envisioned Task Model [Seffah and Forbrig 2002]

The generic envisioned task model is similar to a platform independent model (PIM) of the MDA approach. It is not tailored to a specific platform or architecture.

System task model:

In the next step, in order to establish the system task model, the generic task models is filtered according to the target platform.

Tasks are attached to devices (more or less stereotypes or roles of devices only). This mapping specifies which type of device supports which task. The restrictions are due to the input/output features of devices. Figure 2-9 portrays the generic task model of Figure 2-8, where the various tasks have been attributed with stereotypes of a device, where "PC" represents a desktop computer, "PT" a palmtop device and "MP" a mobile phone [Seffah and Forbrig 2002].

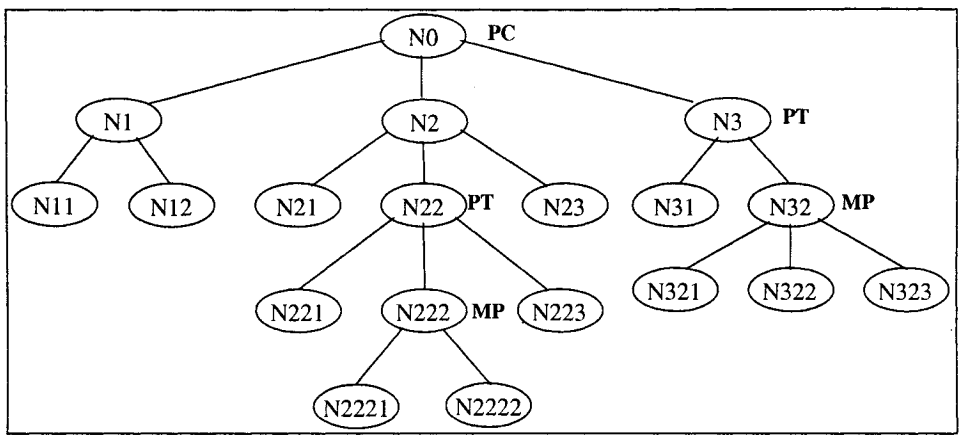

Figure 2-9: Attributed Generic Task Model [Seffah and Forbrig 2002]

In addition, if necessary, the system task model may be further refined for a specific device. In this filter-and-define process, tasks that cannot be supported on a given platform are removed. In other cases, it is necessary to add supplementary details on how a task is decomposed for a specific platform. The left hand side of Figure 2-10 portrays the platform specific task model for the palmtop and the right hand side illustrates the task model for a mobile phone.

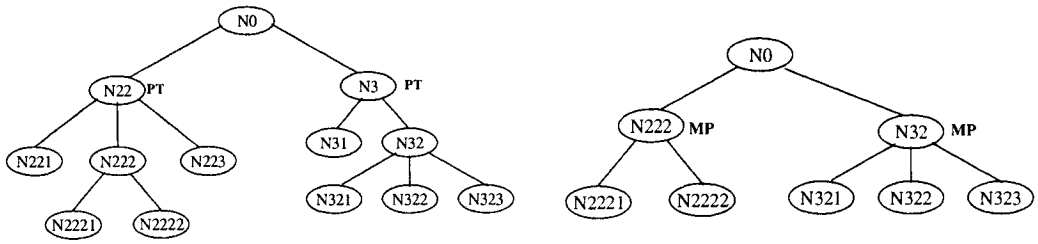

Figure 2-10: Device Dependent Task Models for Palmtop and Mobile Phone

The system task model can be compared to a platform specific model (PSM) of the MDA approach. The model has been tailored to a specific target platform and architecture.

Deriving the abstract user interface:

In the next step, the information of the system task model is used as input for an abstract description of the UI for a specific target platform. Similar to MDA, the platform specific models serve as a basis for the implementation of the application.

In this section I have shown that some of the main principles of MDA are reflected in the domain of interactive application development. However, it remains questionable if concepts for specifying system functionalities and architectures are suitable for specifying the human-computer interaction in terms of the UI. In the next section I will access the suitability of UML for the design of interactive applications.

2.6.3. Suitability of UML for the Design of Interactive Applications

During the development of interactive applications, the specification of the system and the interaction design are often performed in parallel and thus must be coordinated. Therefore a common notation that can be used and understood by both developers and interface designers would foster the integration. This is of particular importance since the interface must be eventually integrated with the rest of the system.

UML consists of a set of notations that have been developed in order to specify and design object-oriented software. Since its beginnings in 1998, it has gradually evolved to become the industry standard. However, UML focuses on the specification of the software system, which is not necessarily suitable for the interaction design of interactive applications. In the following, I will evaluate to what extent UML is suitable and beneficial in the domain of user interface design.

UML consists of a family of notations and models. Among them, class and object diagrams for static domain modelling and use cases, sequence diagrams and activity diagrams, are used for documenting functional requirements. In addition the system's behaviour can be specified using sequence, collaboration, state and activity diagrams.

However within the scope of designing interactive applications, Vanderdonckt suggested nine models that are relevant for the UI domain [Vanderdonckt and Puerta 1999]. It is questionable how easily the various UML models can be used to represent these HCI models and to what extent UML can act as a common ground in order to catalyze and facilitate the communication between interaction designers and system developers [Trætteberg 2002]. In the following, I will discuss how UML supports the documentation and specification of interaction scenarios and task models.

Interaction scenarios and UML Use Cases:

A *use case* as defined in UML consists of a class of sequences of actions that a system performs while interacting with an actor. They describe the behaviour of the system under design. However they do not primarily describe the activities of the user using the interactive system. As described by UML use cases are performed by actors, who can be human users but also systems [Markopoulos and Marijnissen 2000].

Within the scope of interaction design, the term interaction scenario captures primarily the user interaction rather than system behaviour. Thus, use cases for interaction design describe single user interaction rather than modelling system-wide functions. If system

functionality is described, it will be described from the view point of the user [Trætteberg 2002].

Traditionally, (interaction) scenarios within interface design capture the goals and intentions of individual users in a certain context of use. Hence use cases, as defined by UML, are not suitable for identifying and understanding the user's needs, which is the foremost doctrine of the user centred design process.

Task models vs. UML Sequence and Activity diagrams:

Task models describe in a high level manner how activities can be formed to reach the user's goals when interacting with the system. UML Sequence diagrams as well as UML activity diagrams are behaviour oriented and therefore capable to model task sequences [Trætteberg 2002].

UML activity diagrams are useful to specify the activity of different actors and during system specification to model the order of execution of object methods. They have the potential to capture sequencing task information and to specify unordered and concurrent activities. In order to associate actors with activities the diagram can be partitioned into so-called "swim lanes".

Nevertheless activity diagrams do not show clearly the hierarchically nature of task structures and the decomposition of tasks. In contrast, task models are more intuitive for the customer and less technical-oriented. Task models can be animated and the future user can get first impressions of the behaviour of the envisioned interactive system. With the help of tools such as the XIML Task Simulator [Forbrig et al. 2003b], the user can step through possible task scenarios (Pluralistic walkthrough) [Nielsen and Mack 1994] within the scope of the underlying task model. Early evaluations and usability tests are possible.

Sequence diagrams show how actors or objects communicate over the time. Actors can be either users of the systems or system functions. In addition, the lifecycle of objects and the

invocations between their methods can be modelled using sequence diagrams. The communication in sequence diagrams is based on message and information passing. If sequence diagrams were used as visual representations for task modelling, the actors would be the different users who perform the tasks. Message passing would be interpreted as task activation or information flow. However if sequence diagrams are used for task representation, the focus will be shifted from the task structure to the actor's interaction [Trætteberg 2002].

As demonstrated by the generic and specific task model, some principles of MDA can also be detected in the area of model-based UI development. Principles such as "Separation of concerns" have been applied in model based UI development approaches as well.

However, it must be noted that MDA, and therefore UML, have been developed in order to specify the business logic and the architecture of software systems. As demonstrated, UML has its focus on technically oriented design including architectural and implementation issues rather than human centred specification. In addition borrowing the UML notation and shifting its interpretation from system-oriented specification to user-centred design can be a source of confusion and misunderstanding.

2.7. Limitations of the Model-Based Approach for UI Engineering

Model-based user interface development is being investigated and researched for more than a decade [Trætteberg 2004]. Even though its potential and advantages are considerable, none of the model-based frameworks have been adopted by the mainstream developer. The industry does not share the perspective of employing model-based techniques for UI development. On the contrary; the industry is still led by RAD tools (Rapid Application Development), IDE (Integrated Development Environments) and authoring tools (like Macromedia Dreamweaver or Macromedia Flash) [Molina 2004]. This "phenomenon" can be explained in good part by Novak's rule [Novak 2004]:

"Automatic Programming is defined as the synthesis of a program from a specification. If automatic programming is to be useful, the specification must be smaller and easier to write than the program would be if written in a conventional programming language."

In fact, model-based UI development is still challenging since some essential problems related to this "technology" are not completely solved. In the following, I will try to identify some of the particular limitations and obstacles.

2.7.1. Building and Reusing Standard Model Templates

In my opinion, the most crucial limitations are the lack of reuse. Rine [Rine and Nada 2000] defined software reuse as the use of existing software artifacts or knowledge to create new software. It is a key method for significantly improving software quality and productivity. Reuse of knowledge and solutions will reduce the development time (time to market) and development costs. In addition, the software artifact becomes more reliable and predictable.

However, within the domain of model based UI development this aspect has not been sufficiently tackled, yet. Current model-based frameworks offer no, or only very limited, support for re-using model fragments or templates. For example, the MOBI-D tool suite does not offer any functionality that allows importing model or design fragments. TERESA allows only the cut and paste of static task fragments. Resolving name or relation conflicts is not supported. In essence, starting a new project means re-inventing the wheel and building and interlinking all models from scratch. Almost no design reuse from previous works in terms of model fragments is possible.

Within the scope of model-based UI development, the lack of reuse is particularly crucial since the creation of the various models and the process of linking models to each other is a tedious, time-consuming activity.

For example, as shown in [Paquette and Schneider 2004] the development of the task model is quite onerous for medium to large size systems which are highly interactive and provide context-dependent UIs. By elaborating on a Case study, Paquette investigated that task modeling becomes tedious when specifying a task model in sufficient detail. The detailed level is important if task models are used as a starting point for the evolution of the final UI and for early evaluations.

From this emerges the need for a more advanced "disciplined" form of reuse. Obviously copy-and-paste of analysis and design model fragments (as implemented in the TERESA framework) is not enough to integrate reuse in a systematic and retraceable way into the model-based UI development life cycle. Furthermore, aside from the issue of encapsulation, customization and composition of complex model "templates" needs to be addressed.

In particular, it is important to outline a clear classification of models which determine the UI development process. Based on this classification, different kinds of model fragments, which are candidates for reuse, can be identified and extracted. These fragments can then be re-employed in a different context and project as model templates to establish the various models. Within the scope of this thesis, I will show how the concept of patterns (discussed in Chapter 3) can be used in order to describe these model templates.

2.7.2. Complexity of the Model and Lack of Tool Support

A major disadvantage of model-based UI development approaches is the complexity of the models and their notations, which are often hard to learn and use [Myers 1995]. In addition, in order to incorporate a great level of detail, the various models might grow too large to be easily understood and to be entirely comprehended. Different views at different levels of abstraction and granularity on the models should be provided. At this point, the employment of patterns can bring in a more abstract level of model information. Instead of thinking in terms of tasks (task model) or dialog transitions, one can think in terms of patterns at a more course-grained level.

An appropriate environment can help to overcome the complexity limitation, providing features such as graphical editors, assistants and design critics to support UI designers. Unfortunately, there are only a few tools available on the market. Most of them are research projects, and thus of prototypical nature. In addition, there is almost no interoperation between the few tools. In [Molina 2004] the need for a standard in XML with clearly defined semantics, as a base for tool interchange, is postulated.

3. Patterns in HCI

In this chapter, I will provide an overview of the main characteristics of patterns and pattern languages in HCI. Then, based on existing pattern-driven UI development frameworks, I will summarize the main aspects that must be considered in order to effectively integrate patterns as vehicles for reuse into model-based UI development.

3.1. Patterns, Software Patterns, and Patterns in HCI

3.1.1. Tracing the Evolution of Patterns

The architect Christopher Alexander introduced the idea of patterns in the early 1970s. In "A Pattern Language" [Alexander et al. 1977] Alexander compiled 253 solutions or design "patterns" that recur in architecture. Examples of such patterns are SMALL PARKING LOTS (#103) or SIX-FOOT BALCONY (#167).

Ever since Christopher Alexander published his seminal work, his idea of identifying patterns of successful architectural experiences has been translated to other domains and new disciplines. Among others, his idea has been influential in computer science, information structures and organizations [Coplien 1997], music [Borchers 2001] and even oriental carpet studies [Salinagros 2000].

Christopher Alexander's work is having a great impact in computer science. As mature engineering disciplines have handbooks that describe known design solutions, software engineering lacks such common ground. For instance, automobile designers do not design cars using the laws of physics. Instead, they reuse designs, documented in their handbooks. In the domain of software engineering, patterns are one attempt to describe successful solutions to common software problems [Schmidt et al. 1996]. Any design solution that reappears in different contexts can be called a pattern and thus re-used as a unit. To date, patterns are acknowledged as a powerful theoretical framework in order to compose complex applications [Salinagros 2000].

The first software patterns were written by object-oriented developers. They focused on object-oriented design and programming [Gamma et al. 1995] or on object-oriented modeling [Coad 1996]. The book "Design Patterns" by the "Gang of Four" [Gamma et al. 1995] has been widely acknowledged and referenced within the community. Since then, the idea of design patterns has been expanded [Shalloway and Trott 2001] and adopted to concrete programming languages [Alpert 1998; Grand 2002]. However, patterns in the object-oriented world suffer from the same limitation as most object-oriented approaches: they focus more on the internal design aspects of the system, rather than the external facets, which determine in good part how the user interacts with the system.

Therefore, similar to the entire software engineering community, the user interface design community has been a forum for vigorous discussion on pattern languages for user interface design and usability engineering. An HCI pattern is an effective way to transmit experience about recurrent problems in the HCI domain.

3.1.2. Definition of HCI Patterns

HCI patterns focus primarily on solutions to problems related to the design of interactive systems. By providing well-known solutions to UI and usability specific problems and capturing best user experiences, they help with the design of more usable systems [Javahery 2003].

Well written patterns can guide (process patterns) and support (product-oriented patterns) the practice of performing task analysis, writing scenarios, etc. . At its best, patterns embody and articulate crucial empirical user experiences. They have the potential to bring in critical and knowledgeable end users into the design process. Moreover, patterns can act as a common ground in order to transcend professional jargon and to foster and ease the communication between end user and developer.

In other words UI patterns capture the essence of successful solutions to recurring design problems. Correctly applied, they ease and accelerate the development of initial prototypes and assist with design choices. A pattern takes what was previously an art of designing usable software and turns it into a reusable unit.

3.2. HCI Pattern Languages

3.2.1. From Patterns to Pattern Languages

An individual pattern can be valuable for designers, but when patterns are placed into relationships with each other, a more valuable solution can be attained [Welie and Veer 2003]. Moreover, narrowing possibilities is essential for practical design [Salinagros 2000]. The elimination of choices simplifies and directs the design process. Then out of the remaining choices, the most suitable candidate is picked depending on the context of use and application domain. In the field of patterns, a pattern language brings in constraints in the form of interrelations between patterns, which eliminate a large number of possibilities to combine patterns. However it is to be noted that this still allows an infinite number of combinations between patterns and thus possible designs.

Thus, a pattern language is more than just a collection of patterns. A catalogue or collection of patterns can be viewed as a dictionary, which has no rules for flow through the patterns, internal relations or hierarchical structures. In contrast to this, a pattern language contains information regarding when and in what manner patterns can be combined in order to form higher-level patterns.

In order to visualize patterns and their interconnections, graph representations can be used. Patterns may be visualized as nodes in a graph. The pattern language combines the nodes together into an organizational framework. Thus, the graph is connected by edges of different length. Figure 3-1 illustrates the pattern map of the "Experiences" pattern language [Coram and Lee 1998]. A loose collection of patterns is not a pattern language since it lacks connections.

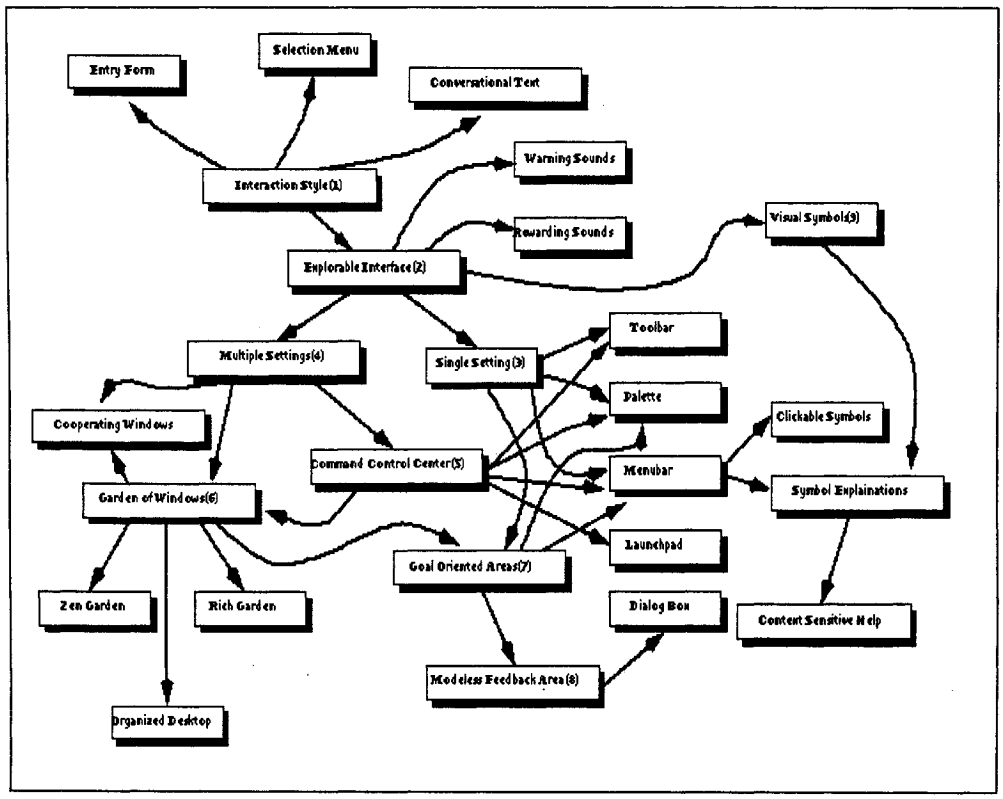

Figure 3-1: Pattern Map of the "Experiences" Pattern Language [Coram and Lee 1998]

The knowledge of a pattern language lies not only in the patterns itself, but can be derived from the connections between patterns. Thus, the rules, by which the patterns are interconnected, are just as important as the patterns themselves. A coherent combination of patterns forms a new high level pattern which contains more information (knowledge) than the sum of information contained in each single pattern. The reason for this phenomenon is that organizational information is not contained in any of the peer patterns.

A pattern language contains super-ordinate and sub-ordinate patterns. Similar to most complex systems, pattern languages contain hierarchical structures. Connections between patterns exist at the same level and across different levels of hierarchy. As a result pattern languages are structured in at least a horizontal and a vertical dimension. The horizontal dimension captures the hierarchical structure of the language, whereas the vertical dimension captures interrelations of patterns at the same hierarchical level.

3.2.2. Influential Pattern Languages in HCI

"The goals of an HCI Pattern Language are to share successful HCI design solutions among HCI professionals, and to provide a common language for HCI design to anyone involved in the design, development, evaluation, or use of interactive systems." [Borchers 2001]

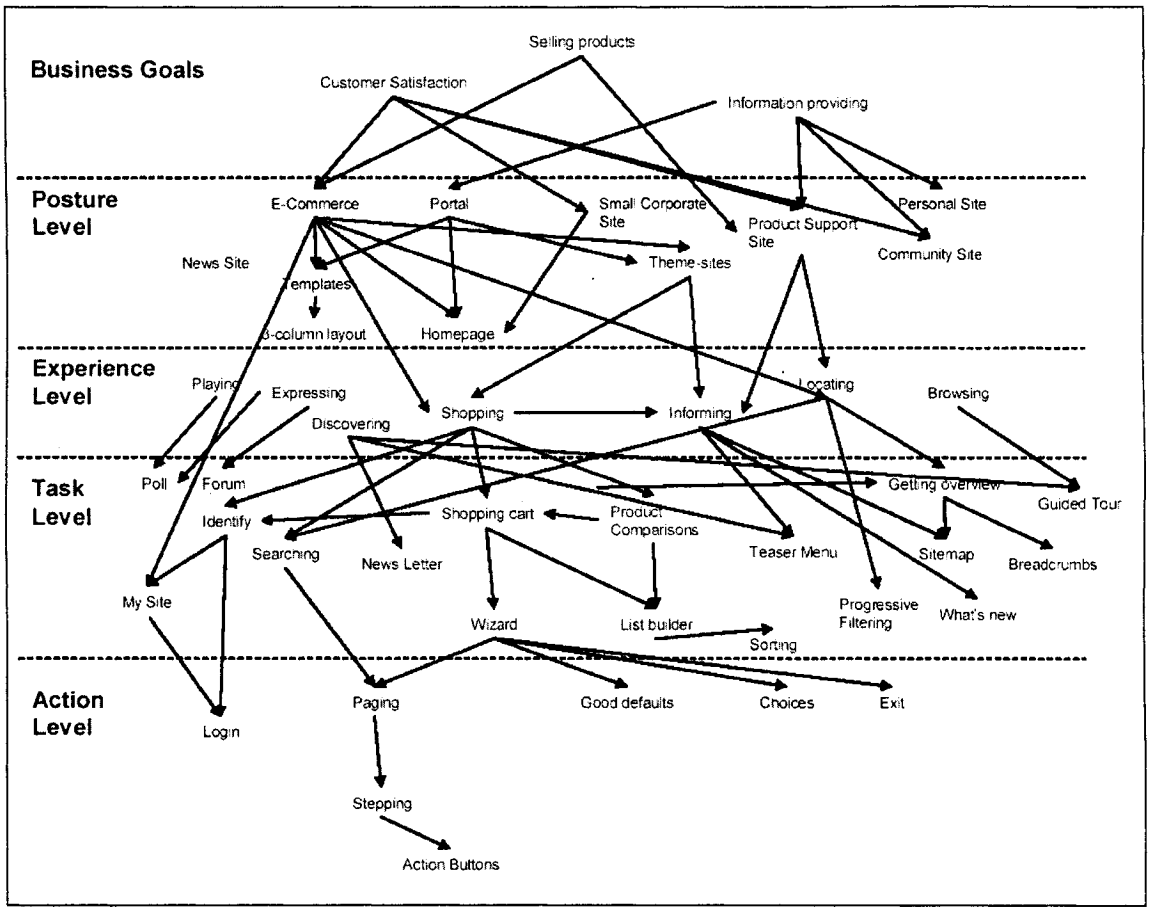

Figure 3-2: Structure of Interaction Design Pattern Language [Welie and Veer 2003]

Recently, several groups of HCI practitioners and interactive application designers have become interested in formulating HCI patterns (pattern languages). A number of concrete

pattern languages for interaction design have been suggested. Van Duyne's "The Design of Sites" [Duyne et al. 2002], Welie's Interaction Design Patterns [Welie 2004], Experiences [Coram and Lee 1998], and Tidwell's UI Patterns and Techniques [Tidwell 2004] play an important role in the field of HCI and interactive application design. In addition, smaller languages such as Laakso's User Interface Design Patterns [Laakso 2003] and the UPADE Web Language [Javahery 2003], developed by the HCSE team, are quite influential.

Currently the pattern language for web site design written by van Duyne is the most comprehensive effort in this field. The authors compiled 90 patterns addressing all aspects of the design of web pages ranging from e-commerce to personal pages. The patterns are arranged in 12 groups which exist at different levels of hierarchy. The highest level is called "Site Genres" which provides a convenient starting point into the language, and enables the user to choose the type of site to be created. Starting from a particular site genre pattern, various lower patterns are referenced. Hence the authors have succeeded not only to provide a starting point into their language, but also to show how patterns on different levels interact with each other.

The Interaction Design pattern language by Welie is organized similarly to van Duyne. Welie also sees web design as a top-down activity. The user enters the language by selecting a "posture" pattern. According to Welie, posture patterns describe a certain overall type or genre of the site. Then the user can follow along the references to lower level patterns which have been placed at so called "Experience", "Task" and "Action" levels. Figure 3-2 depicts the hierarchical nature of Welie's pattern language.

Another example of a pattern language is the UPADE Web Language. UPADE is an acronym for Usability Patterns-Assisted Design Environment and has been developed by Concordia's HCSE team. Each pattern in the UPADE web language provides a proven solution for a common usability and HCI-related problem occurring in a specific context of use for web applications. Figure 3-3 portrays the structure of the language, which consists of a set of patterns grouped into three categories:

- Architectural Patterns
- Structural Patterns
- Navigation Support Patterns

In the UPADE Web Language, patterns are organized at different levels of hierarchy with interrelationships such as super-ordinate and sub-ordinate. In addition, patterns at the same level of hierarchy can also have interrelationships such as competitor and neighboring [Li 2001].

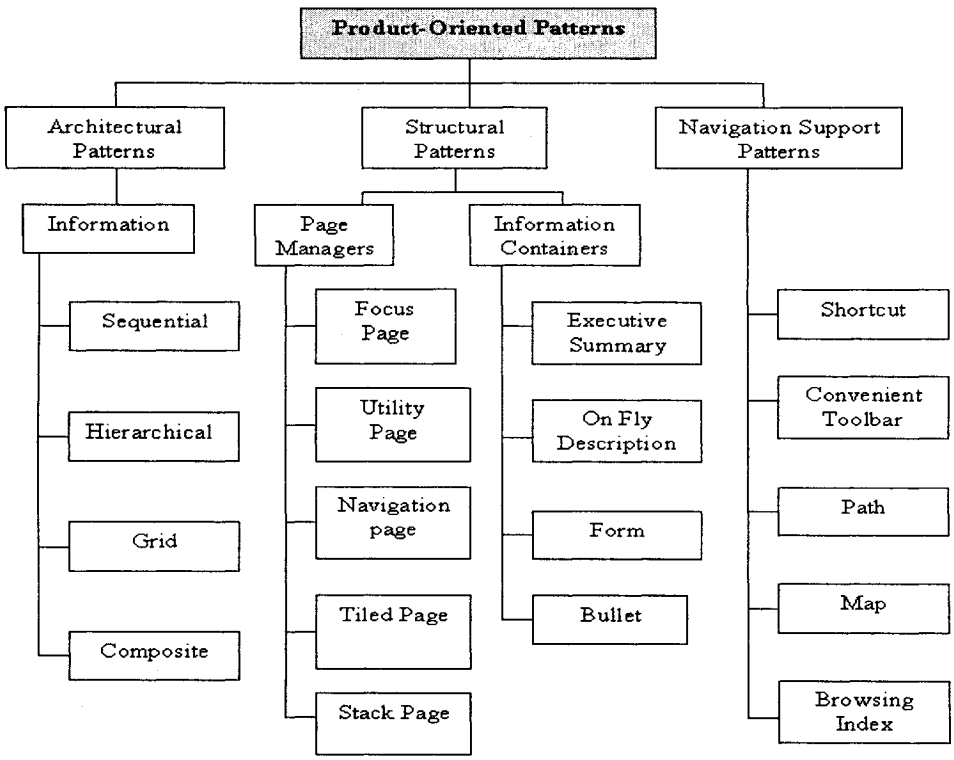

Figure 3-3: The UPADE Web Language [Javahery 2003]

3.3. Frameworks for Integrating Patterns in the Development Process

One of the major limitations that HCI has not yet overcome lies in communication barriers between different individuals involved in the development process. It is difficult for user interface experts to communicate their experience and methods to other design team members [Borchers 2001]. Successful designs require individuals to communicate their concepts and ideas, and to build a common forum for the discussion of already-available design practices. Patterns offer a good way of capturing and transferring this knowledge. They are presented consistently, are easy to read, and provide background reasoning. The format provides information about the problem at hand, the context, a solution, as well as the rationale behind this solution [Granlund and Lafreniere 1999]. In [Erickson 2000], Erickson proposes using pattern languages as a common ground for such communication and dissemination of knowledge. Patterns provide a lingua franca that can be read and understood by all, regardless of background. Such a "common" language can act as a communicative resource to facilitate discussion, presentation, and negotiation for the many different individuals who play a role in designing interactive systems [Javahery 2003]. More importantly than being a communicative tool, pattern languages guide software designers through the design process, by providing solutions to a set of common design problems. In the following, I will outline the role of patterns within some of the most influential development frameworks.

3.3.1. Pattern Oriented Design (POD)

In order to render HCI patterns understandable by novice designers and software engineers who are unfamiliar with UCD and usability engineering, patterns should be presented to developers as part of the design process. Within the scope of the development of web interactive applications, POD as proposed by Seffah and Javahery [Javahery and Seffah 2002], is one attempt to seemingly integrate patterns into the design process. On the basis of the UPADE Web Language, the POD 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 come up with concrete and effective design solutions.

3.3.2. The Pattern Supported Approach (PSA)

The "Pattern Supported Approach" (PSA) [Granlund and Lafreniere 1999] addresses patterns not only during the design phase, but also during the entire software development process. PSA aims to support early system definition and conceptual design through the use of HCI patterns. In particular, patterns have been used to describe business domains, processes, and tasks to aid early system definition and conceptual design. The main idea of PSA is that HCI patterns can be documented according to the development lifecycle. In other words, during system definition and task analysis, depending on the context of use, it can be decided which HCI patterns are appropriate for the design phase. In contrast to POD, the concept of linking patterns together to result in a design is not tackled.

3.3.3. Patterns in UI Reengineering

In [Javahery et al. 2003], patterns are used in reengineering existing systems to different platforms, or Multiple User Interfaces. HCI patterns are used in the process to abstract and re-deploy the UI onto different platforms. In particular, during reverse engineering, patterns are extracted from the interface und used to create a UI multi-model. Next, patterns and design strategies in the platform-independent UI model are analyzed, and transformed if appropriate. Eventually during forward engineering, the UI multi-model can be "instantiated" to different platforms based on constraints and capabilities. The patterns are then implemented on different platforms.

3.3.4. Just-UI Framework

Molina [Molina et al. 2002] has recognized that existing pattern collections focus on design problems, and not on analysis problems. Therefore, 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 mostly informal descriptions of patterns today are not suitable for tool use. Within the Just-UI framework, a fixed set of patterns has been formalized in order to be processable by the "OliverNova" tool [OliverNova 2004]. Eventually, based on the analysis model, the JUST-UI framework uses code generation in order to derive an implementation of the UI.

3.3.5. Task Patterns

Breedvelt [Breedvelt et al. 1997] discusses the idea of using task patterns in order to foster design reuse for task modeling. Task patterns encapsulate task templates for common design problems. Whenever designers realize that the problem they are considering is similar to one problem which has already been found, and solved then they can immediately reuse the solution previously developed (captured by the task pattern). In particular, the task patterns are used as templates (or building task blocks) for designing the task model of an application. According to Breedvelt, an additional advantage of using task patterns is that they help to make the task specification easier to read and interpret. Patterns can be employed as placeholders for common, repetitive task fragments. Instead of thinking in tasks, one can furthermore think in patterns at a more abstract level. This makes the task specification more compact and legible.

However, as suggested by Breedvelt, task patterns are merely static encapsulations of a (task) design problem in a particular context of use. Concepts for a more advanced form of reuse, including customization and combination, are not presented.

3.4. Limitations and Consequences

As outlined in the previous section, patterns have been employed in various research and development frameworks. Different approaches have exploited different aspects of patterns. However, there is no framework that utilizes exhaustively the full potential of patterns. Therefore in this thesis, I will try to unify the most important aspects under one umbrella, in order to establish a pattern concept that is able to cope with the three dimensions of reuse (Encapsulation, Customization and Combination).

Inspired by the approach of Breedvelt, I will use patterns as a medium to describe reusable fragments for the creation of models. The Pattern Supported Approach has demonstrated the applicability of patterns in the abstract and concrete phases of UI development. In addition, patterns have been used to create a UI multi-model containing the abstract and concrete aspects of the UI, within the context of UI reengineering for multiple user interface (MUI) migration. Therefore, in contrast to Breedvelt, I will use patterns not only as building blocks for the task model, but also for the more concrete models of the UI development lifecycle.

The POD approach has highlighted another important aspect of the pattern concept: Pattern combination. By combining different patterns, developers can exploit pattern relationships and combine them in order to come up with an effective design solution. I will consider this principle and suggest an interface for combining patterns. As a result, patterns can be a more effective vehicle for reuse.

Furthermore, in order to select and apply patterns efficiently, tool support is necessary. Unfortunately, the purely narrative nature of most patterns and pattern languages does not support reasoning, decision-making, automatic transformations or model building. In Just-UI, Molina recognized the need for a formalization and tool support for patterns. Within this thesis, I will also jump on this bandwagon and propose a possible formalization of patterns according to their target model. Moreover, I will propose a tool that helps the pattern user in selecting, adopting and integrating task patterns to task models.

In summary, patterns have the potential to inherently promote the reuse of ideas and knowledge. From experiences with implementation reuse, it is known that an effective concept of reuse should include these three different dimensions:

- Encapsulation
- Composition
- Customization

The concept of patterns addresses all three reuse dimensions. Naturally and by definition, patterns are encapsulations of solutions to problems in a specific context of use. In addition, the concept of pattern languages implies that patterns can be combined and nested depending on their relationships. Last but not least, patterns promote solutions, which occur over and over again in different situations. Therefore patterns are not static, and must be customized to the current context of use.

In the next chapter, I will illustrate how patterns can act as a driving force for model-based UI development. An interface for combining patterns will be provided and the general process of pattern application will be introduced. In addition, I will approach a formalization of patterns according to the target models. An in-depth description of every phase of my development approach shows in great detail the establishment of each model and the influence of different kinds of patterns. Basically, I will discuss patterns for the task, dialog, presentation and layout model.

4. The Pattern-Driven Model-Based Framework

In this section, I will introduce our pattern-driven model-based framework for the development of interactive applications. I will specify, define and document different kinds of HCI patterns that can be applicable for the various models. In addition, I will introduce the concept of variables for patterns and I will outline the general process of pattern application on models.

4.1. Overview of the Proposed Framework

Within our pattern-driven model-based approach, a framework consists of a set of models, a method for constructing these models, a set of patterns that can be used in this construction, and tool support. In what follows, I will explain each of these core constituents of the proposed framework

4.1.1. The Basic Models

In model-based UI design methodology, various models are used to describe the relevant aspects of the user interface. As illustrated in chapter 2 and Figure 2-1, many facets exist, as well as related models. Figure 4-1 portrays that our approach focuses on a subset of the proposed models and consists merely of a (envisioned) task model, a user model, a business object model, a dialog model, a presentation model and a layout model. Basically, it starts with domain analysis with the creation of the user, task and business object model, capturing the current situation. Next, in consideration of the support of a future application, envisioned user, task and object models are designed. Then, based on these rather abstract models, the more concrete dialog, presentation and layout models are developed to reveal some implementation details of the UI.

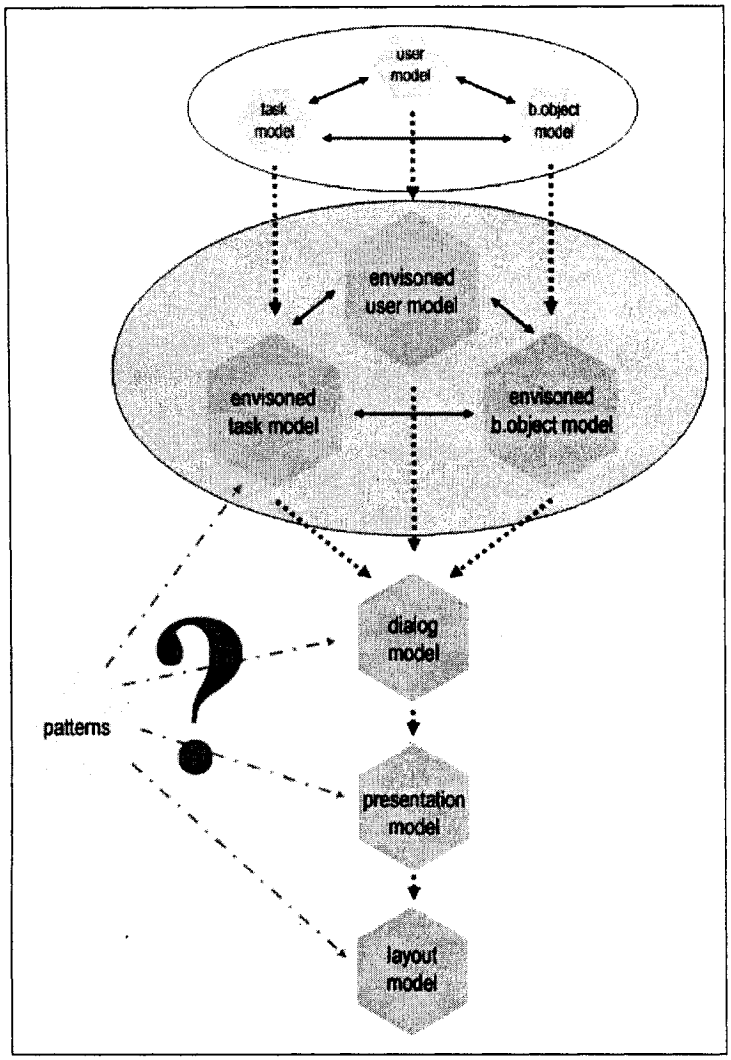

Figure 4-1: Model-Based Development

With respect to the design of interactive systems, "among all these models, the task model has gained much attention and many acceptances in the scientific community to be the model from which development should be initiated" [Vanderdonckt and Puerta 1999]. Therefore, user task analysis resulting in a task model is the starting point of our model-based approach. Possible intentions of the user are captured. In addition, activities of users pursuing a certain goal are described [Paternò 2000]. The user's tasks, as they are currently performed, are elicited. Additionally, models for capturing user characteristics and business objects are developed.

As shown in the above figure, the task, object and user models have many interrelationships. The task model is modeled in mutual relationship to the user model, representing the functional roles users have to play for task accomplishment, as well as their individual perception of the tasks. The user model is also related to the business-object model since the user may require different views of the data while performing a task. Besides, a relationship between the business-object model and the task model is required, since objects of the business-object model may be needed in form of artifacts and tools for task accomplishment. An artifact is an object, which is essential for a task. Without this object, the task cannot be performed. The state of this artifact is usually changed during the course of task performance. In contrast to an artifact, a tool is an object that merely supports performing a task. Such a tool can be substituted without changing the intention of a task [Forbrig et al. 2003b].

From the user-task model evolves the envisioned user-task model describing from the user's point of view, how activities can be performed in order to reach the user's goal while interacting with the application [Paternò 2000]. In other words, the envisioned user-task model captures the user task and system behavior with respect to the task-set, in order to achieve a goal. In essence, the system will be viewed through the set of tasks performed by the user, which creates input to, and output from, the system [Abi-Aad et al. 2003].

The distinction between existing and envisioned task models is necessary because the introduction of a new interactive application leads inevitably to a change of the role of the user and the task that the user will perform. Furthermore, new interactive tasks (the actions the user performs with the new interactive system) must be modeled. At this point, early design decisions regarding the future support of the interactive system, are made. As illustrated by the dark shaded ellipse of Figure 4-1, envisioned user- and business object models are also established.

Model-based design focuses on mapping between the various models [Vanderdonckt et al. 2003b]. Thus at this point, based on the rather abstract task, user and object models; a

dialog, presentation and layout model are derived to reveal some implementation details of the UI.

First, a dialog model is interactively derived from the task, user and object model. It captures the navigational structure of the UI. In particular, the dialog model associates several tasks to dialog views and defines transitions between these dialog views.

Next, in order to develop the presentation model, the tasks of each dialog view are associated in an abstract manner with interaction elements, which are necessary for task accomplishment. Examples of such interaction elements can be simply buttons or text fields. However, complex aggregated "components" such as calendars or forms are also possible. Style attributes such as size, color and position are left open and will be defined by the layout model.

Eventually, the interaction objects are positioned following an overall layout or floor plan resulting in the layout model. Additionally, the visual appearance of each interaction element is specified by setting fonts, colors and dimensions. Finally, based on the information of the various models, a concrete user interface implementation is automatically rendered.

As already insinuated by Figure 4-1 within our approach patterns are used to tackle the establishment of the different models. In the next sections I will describe how and which patterns are applicable for the various models.

4.1.2. Patterns Used

In order to accelerate the development and foster reuse, patterns are employed within our framework. In contrast to Figure 4-1, Figure 4-2 reveals a more detailed view of our proposed framework by focusing on the "design" models. It is illustrated that different kinds of patterns act as building blocks for the establishment of the various models.

In particular, task and feature patterns are used to describe hierarchically-structured task fragments. These fragments can then be used as building task blocks for gradually

building the envisioned task model. Patterns for the dialog model are employed to help in grouping the tasks and to suggest sequences between dialog views. In contrast to this, presentation patterns are applied to map complex tasks to a predefined set of interaction elements defined in the presentation model. Eventually, layout patterns are utilized to set certain styles or "floor plans" which are captured by the layout model.

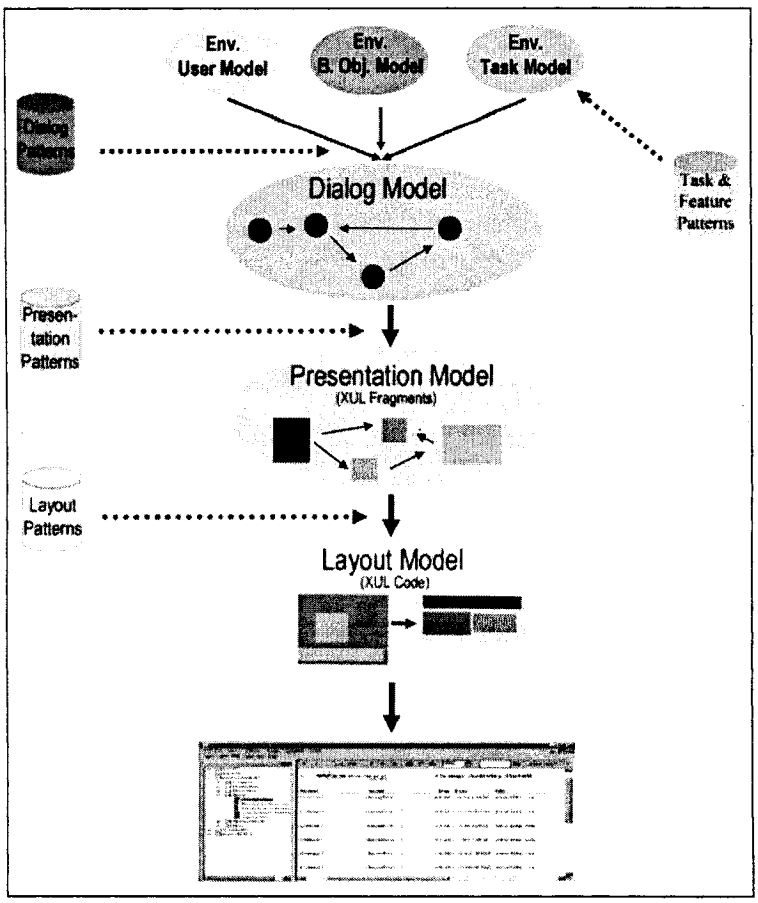

Figure 4-2: Different Patterns Associated with Different Models

In the following sections, I will introduce the general process of pattern application and I will suggest an interface notation for patterns. In addition, I will explain in greater detail each phase of our pattern-driven model-based framework.

4.1.3. The Process of Pattern Application

As motivated in the previous chapters using patterns can be an effective way to transmit experiences about recurrent problems in the software and UI development domain. Therefore a solution should be generic enough to apply to different contexts of use. In other words patterns should be formulated generically enough to withstand variations of context and domain. Before the pattern solution stated in the pattern is really tangible and applicable, it must be adapted to the current context of use. Thus I suggest that patterns contain variables, which can act as placeholders for each particular context of use. In other words, the variables must be bound to concrete values representing the surrounding context.

In the previous section I mentioned that patterns are used as building blocks for different models throughout our model-based development approach. For each model, I propose the following process of the pattern application including four milestones:

1. **Identification**: A subset M' of the target model M is identified: M' $\subseteq$ M. This should reduce the domain size, and help focus the attention on a smaller subset of concern 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 capture potential "spots" that could be improved using some patterns. This step depends strongly on the experience and the creativity of the designer.

3. **Adaptation**: A pattern is an abstraction that must be instantiated. Therefore in this step the pattern P will be adjusted according to the context of use resulting in the pattern instance S. In a top down process all variable parts will be bound to specific values, resulting in a concrete instance of the pattern.

4. **Integration**: The pattern instance S will be integrated into M' by connecting it to the other elements in the domain. This may require replacing, updating or otherwise modifying other objects to produce a seamless piece of design.

It is to be note that ideally all four steps should be supported by tools in order to apply patterns more effectively. In the last section of this chapter, the Task-Pattern-Wizard is introduced, which aims at supporting the process of applying patterns to task models.

4.1.4. Pattern Structure and Interface

Alexander [Alexander et al. 1977] gave all of his patterns a uniform structure and format. Each of Alexander's patterns consisted of the same properties, presented in the same sequence and form. In this thesis, I will follow this principle and document our patterns as follows:

- **Name:** Name of the pattern
- **Problem:** Problem that is tackled by the pattern
- **Context:** Context of use in which the pattern is applicable
- **Solution:** Outline of the solution for the problem
- **Rational:** Reasons why the solution works
- **Interface:** External representation of the pattern
- **Related Pattern:** Other patterns to which the pattern is related

As mentioned in the previous section, variables are used as placeholders for the context of use. During the process of pattern adaptation, these placeholders will be replaced by concrete values representing the particular consequences of the current context of use.

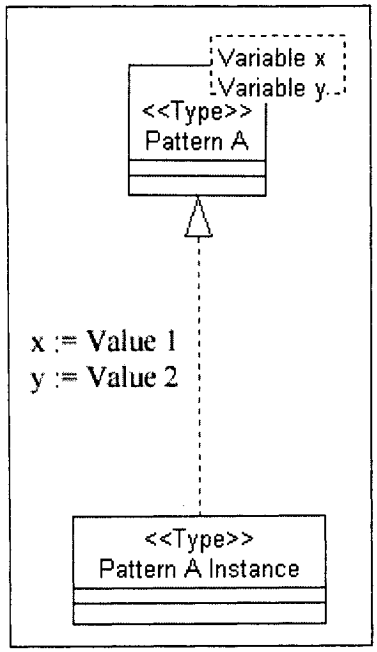

Figure 4-3: Interface of a Pattern

Figure 4-3 shows the interface of pattern *A*. The UML notation for parametric classes is used to visualize that the pattern assumes two parameters (Variable "x", "y"). In order to instantiate the pattern both variables must be assign concrete values. Practically the interface tells the patterns user, that values for variable "x" and "y" must be provided in order to use the pattern. In Figure 4-3 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.

Often one pattern is implemented using other patterns. In other words a pattern can be composed of several sub-patterns. In Figure 4-4 I have visualized this pattern – sub-pattern relationship using the concept of aggregation of classes. The pattern *A* consists of the sub-patterns *B* and *C*. If we place patterns in this kind of relationship, we have to pay special attention to the variables of the pattern. A variable, defined at the super-pattern level can affect the variables used in the sub-patterns. In Figure 4-4, the variable "x" of the pattern A affects the variables "yy" and "zz" of the sub-patterns *C* and *B*. During the

process of pattern adaptation, the variables "yy" and "zz" will be bound with the value of "x". We can see that modification of a high level pattern can affect all sub-patterns.

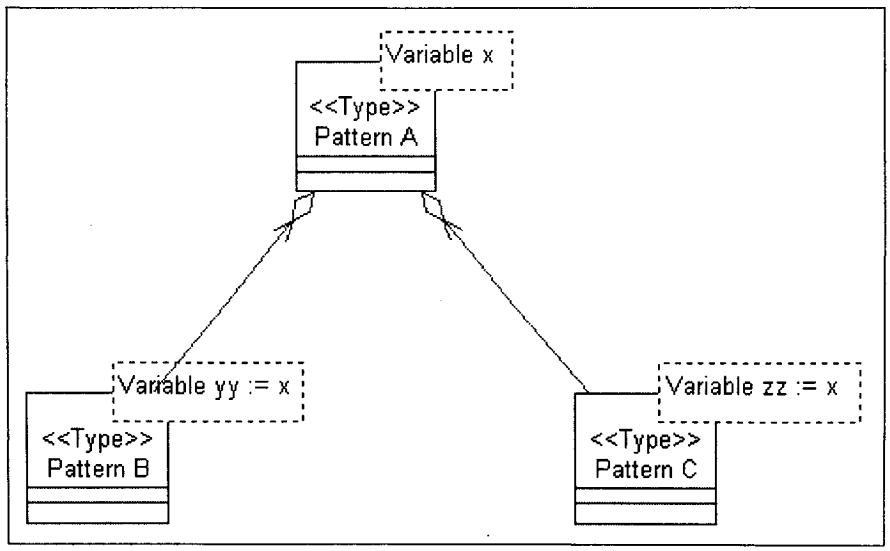

Figure 4-4 Pattern Aggregation

So far, it was shown how patterns are generally applied to models and how they can be aggregated. In the next section, I will discuss in great detail, how particular patterns can be applied on the task, dialog, presentation and layout model.

4.2. Patterns in Task Modeling

In what follows, I will describe how and which patterns are applicable for the envisioned task model. I will also demonstrate how the previously introduced pattern interface can be used to combine and aggregate task and feature patterns.

The envisioned task model describes how activities can be performed to reach the user's goals when interacting with an interactive system [Paternò 2000]. Using task models, designers can develop integrated descriptions of the system from a functional and interactive point of view. Task models typically are hierarchical decompositions of tasks

and subtasks into atomic actions [Vanderdonckt et al. 2002]. Also the relationships between tasks are described in correlation with the execution order or dependencies between peer tasks. The tasks may contain attributes about the importance, the duration of execution and the frequency of use.

4.2.1. Definition and Application of Task and Feature Patterns

In order to speed up the process of establishing the task model and to integrate proven and efficient solution, I suggest using patterns as building blocks. In the following I will explain how patterns for the task model should be written and how they should be applied.

In a subtle manner we distinguish between two kinds of patterns that are applicable for the user-task model: Task Patterns and Feature Patterns.

- **Task Patterns** describe the activities the user has to perform while pursuing a certain goal. The goal description acts as an unambiguous identification for the pattern. In order to compose the pattern as generic and flexible as possible, the goal description should entail at least one variable component. As the variable part of the goal description changes, the content solution part of the pattern will adapt and change accordingly. Task Patterns can be composed out of sub-patterns. These sub-patterns can either be task-patterns or feature-patterns.

- **Feature Patterns**, applied to the user-task model describe the activities the user has to perform using a particular feature of the system. For the purpose of this thesis I define a feature as an aid or a "tool" the user can use in order to fulfill a task. Examples of these features can be *Keyword Search*, *Login* or *Registration*. Feature patterns are identified by the feature description, which should also contain a variable part, to which the realization of the feature (stated in the pattern) will adapt. Feature patterns can comprise other sub-feature patterns.

As mentioned above, the difference between task and feature patterns is subtle, but noticeable. While task patterns concentrate on a specific goal, the same task can be accomplished in different ways using different feature patterns. That is why feature patterns are important as a classification. Similarly, the same feature pattern can be used to accomplish different task patterns. Therefore it is safe to say that there is a many-to-many relationship between the two. To summarize, Task patterns are concerned with the user goals (what we need to do), while feature Patterns are concerned with the system behavior (how we can do it).

In order to clarify the previously introduced concept I will now illustrate some examples. For the sake of simplicity, I will only use simplified versions of patterns in this section.

A typical task a user performs in many different applications is to find something. This can range from finding a book at www.amazon.com to finding a used car at www.cars.com, to even finding a computer in a network environment. All these tasks embody the same basic task and can just be distinguished by the particular "Find" object in the goal description. In order to create a generalized *Find* Pattern we must abstract the particular object we are searching, and replace it with a generic variable. Figure 4-5 gives an impression about the task tree of such a pattern. I will use the notation of CTTE [Paternò 2000].

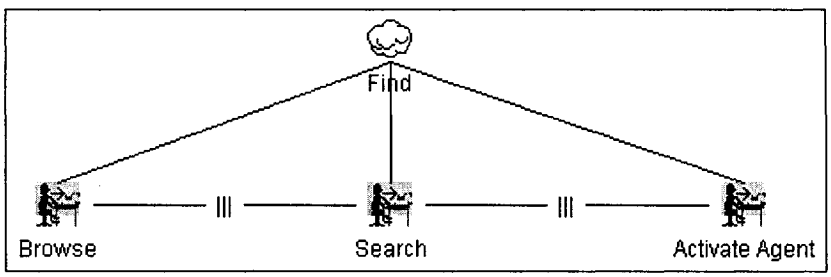

Figure 4-5: Pattern for the Task "Find" Information

Find information can be performed by browsing, searching or using an agent. For more abstract considerations the previously introduced UML notation for parametric classes is used. Figure 4-6 shows the pattern of Figure 4-5 in this way. Details of the task structure

are omitted. Here, the *Find* pattern contains the variable "Information" which is a placeholder for the particular type of information one is trying to find.

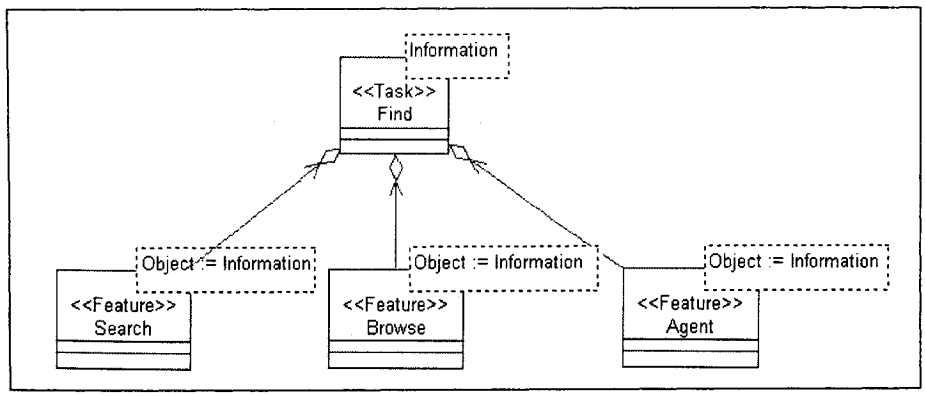

Figure 4-6: The *Find* Pattern and its Sub-Patterns

Figure 4-6 shows that the *Find* pattern is composed of the feature patterns *Browse*, *Search* and *Agent*. It also demonstrates how the variables of each pattern are interrelated. The value of the variable "Information" of the *Find* pattern will be used to assign the "Object" variable in all sub patterns. During the process of adaptation, the variables of each pattern must be resolved top-down and replaced by concrete values.

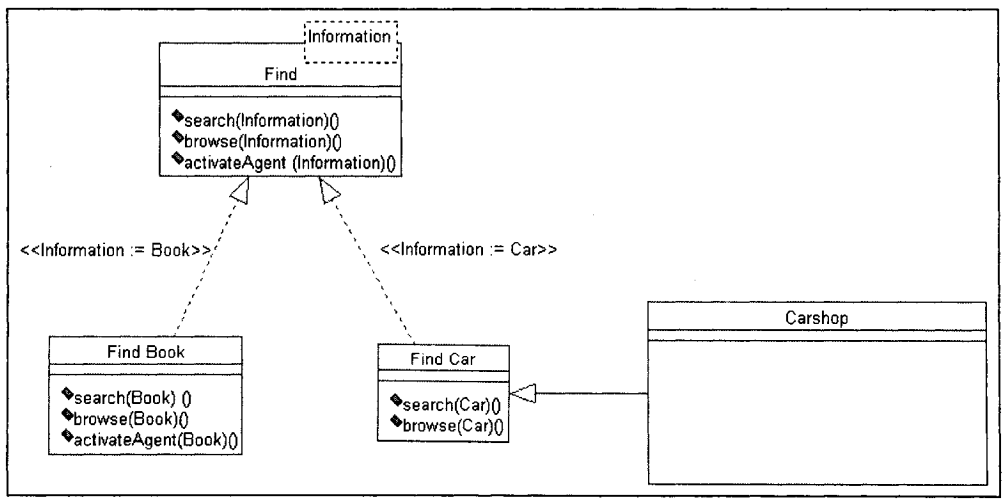

Figure 4-7: The *Find* Pattern and its Instances

In Figure 4-7, we have bound the variable "Information" with the value "Book" to create the pattern instance *Find Book*; and with the value "Car" to create the pattern instance *Find Car*. Please note that with the binding of a concrete value to the variable "Information" in the goal description, the body of the pattern has changed accordingly.

After the pattern adaptation process, the pattern instance can be integrated in an already-existing task model. In Figure 4-7 *Find Car* has been integrated into the Car-shop task model. This process of integrations has been visualized using the inheritance relationship and can be interpreted as: Car-shop has inherited all methods (sub-tasks) from *Find Car*.

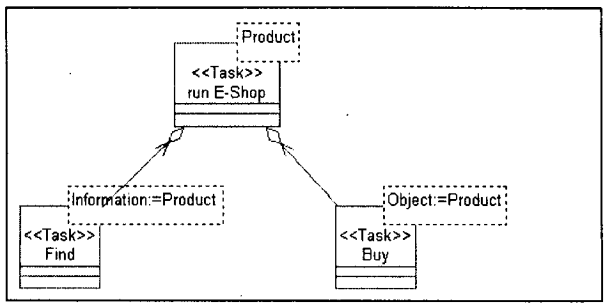

Figure 4-8: The *run E-Shop* Pattern and its Sub-Patterns

As mentioned previously, a pattern can be composed of several sub-patterns. Since task models are naturally hierarchical often task and features patterns are structured hierarchically as well. In Figure Figure 4-8 the pattern *run E-shop* consists of the sub-patterns *Find* and *Buy*. The variable "Product" of the *run E-Shop* pattern affects the variables "Information" and "Object" of the sub-patterns *Find* and *Buy*. During the process of pattern adaptation, the "Information" and "Object" will be bound with the value of "Product".

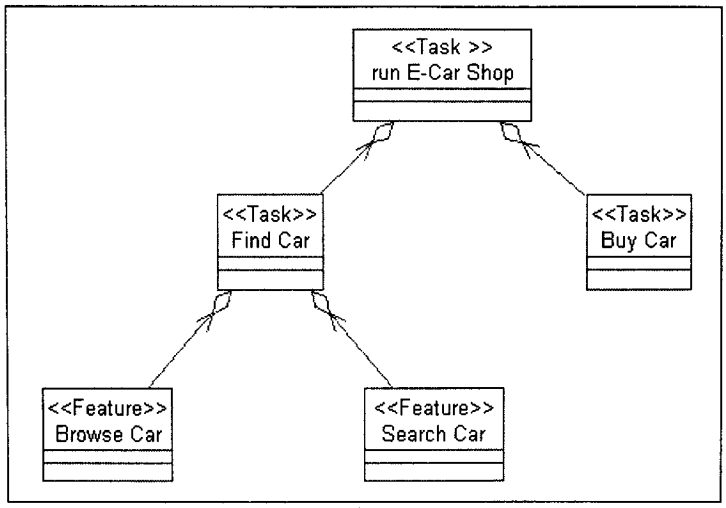

Figure 4-9: The *Car Shop* Pattern Instance

In the case of the example of a Car-Shop displayed in Figure 4-9, the variable Information of the E-Shop pattern has been assign with the value "Car" in order to create the pattern instance *CarShop*. This value has then been passed to the sub-patterns *Buy* and *Find* in order to assign the variables "Information" and "Object" and subsequently to create the instances *Buy Car* and *Find Car*. In the case of *Find Car* the value of the variable "Information" ("Car") is used to automatically assign the variable "Object" in all sub-patterns. Moreover the *Find Car* instance employs only the "Browse" and "Search" branch of the *Find* pattern.

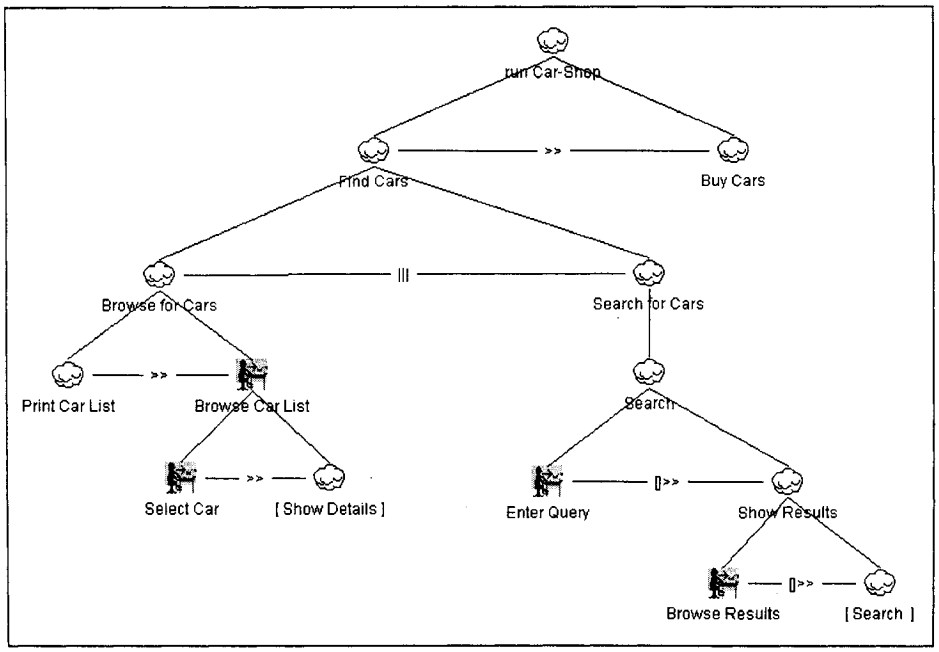

Figure 4-10: Car Shop in CTTE

Eventually after resolving all variables, the *Car Shop* pattern instance will be transformed into a concrete task structure. Figure 4-10 illustrates a cut-out of the modified task model visualized with CTTE.

The top-down process of pattern adaptation can be greatly assisted by tools such as Wizards. A Wizard runs through the task pattern tree and questions the user each time it encounters a variable that has not been resolved yet. In chapter 6, the tool Task-Pattern-Wizard is introduced, which assists the user in selecting, adapting and integrating task and feature patterns.

After elaborately defining the complicity of patterns and the task model, I will now outline how and which patterns are applicable for the more concrete dialog, presentation and layout models. From now on, I will put the main focus on what the different models and the corresponding patterns should encapsulate and how the patterns can be documented or respectively formalized.

4.3. Patterns in Dialog Modeling

The dialog model defines the navigational structure of the user interface. It is a more specific model and can be derived in good part from the more abstract task-, user- and business object models.

There are different strategies to design the dialog model. One possibility is the evolution of the task model to a final user interface. JANUS uses information mainly from the object model. However most approaches are based on tasks. TERESA follows an idea of grouping tasks based on preconditions, which allows an automatic generation of dialog models.

Within our approach, the dialog model is defined by a so-called dialog graph. Formally the dialog graph consists of a set of vertices (dialog views) and edges (dialog transitions). The creation of the dialog graph consists of two steps: First, related tasks are grouped together into dialog views. Secondly, transitions from one dialog to another as well as trigger events are defined.

Please note that finding dialog views and transitions is closely connected to the underlying task model. On the one hand, structural information from the task model, which describes the task–subtask hierarchy can be used to group related tasks into task views. On the other hand, temporal transitions between sub tasks can be used to constrain and derive possible dialog transitions.

We have defined five types of dialog views and two types of transition views. As illustrated on the left hand side of Figure 4-11, we distinguish between a sequential transition and a concurrent transition. In contrast to a sequential transition, a concurrent transition means that both views are still visible.

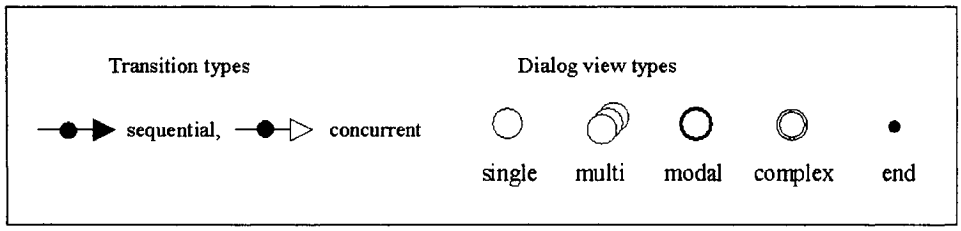

Figure 4-11: Elements of a Dialog Graph

In addition, we distinguish between a single, multi, modal, complex and an end dialog view (portrayed on the right hand side of Figure 4-11). In contrast to the single dialog view, the modal dialog view must be closed first, before any other dialog view can be accessed. In order to define that several instances of the same dialog view are possible, the multi dialog view should be used. The complex dialog view can be composed of other sub dialog views and has its own dialog graph. Eventually, the termination of the application is signaled by the end dialog view [Forbrig et al. 2003b].

In order to foster the establishment of the dialog model, I believe that patterns can first help with the grouping of tasks to dialog views, but also with establishing the transition between the various dialog views.

A typical dialog pattern is the *Recursive Activation* pattern. Originally it was introduced as a task pattern by Breevelt and Paterno [Breedvelt et al. 1997]. However within our framework, we abstracted its essence and re-introduced it as a dialog pattern. The pattern is used when the user wants to activate and manipulate several instances of a dialog view. Practically it suggests a dialog structure where starting from a source dialog, a specific creator task can be used to create an instance of the target dialog view. The pattern is applicable in many modern interfaces where several dialog views of the same type and functionality are accessible concurrently. A typical application scenario example is an e-mail application, which allows the editing of several e-mails concurrently during one session.

In the left part of Figure 4-12, it is illustrated that in order to adapt (instantiate) the pattern the source dialog view and the corresponding creator task and the target dialog view must

be set. A particular instance of the pattern is portrayed by the right part of Figure 4-12 simulating the navigational structure of MS Outlook for composing a new message. For this particular example the visual notation of a tool, called Dialog-Graph-Editor [Forbrig et al. 2003b], was used.

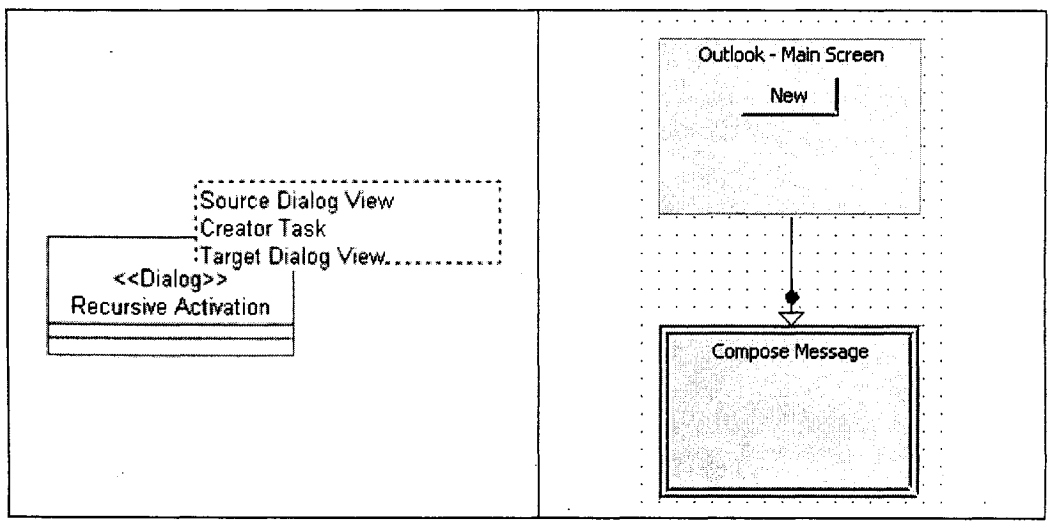

Figure 4-12: Interface and "Outlook" Instance of the *Recursive Activation Pattern*

We have developed the Dialog-Graph-Editor in order to define and animate dialog graphs. Using the application, the user can interactively "walk through" the dialog graph. It is also possible to experiment with different dialog graphs for the same task model. Different versions can oppose each other and the most suitable version is used for establishing the presentation and the layout model. Due to the separation of task and dialog structure, dialog patterns can be brought in independently.

In the future, we hope to extend the Dialog-Graph-Editor in order to support the application of dialog patterns. As the Dialog-Graph-Editor internally processes dialog structures described in XIML, we are currently investigating the possibility of formalizing dialog patterns as XIML fragments.

4.4. Patterns in (Abstract) Presentation Modeling

In the presentation model, a set of abstract UI elements is defined to determine the abstract appearance of the user interface. In particular, the grouped tasks of each dialog view are associated with a set of interaction elements such as forms, buttons and lists. Moreover, some domain objects which will be displayed on the interface are mapped to interaction elements. Please note that all interaction elements should be described in an abstract manner. Style attributes such as size, font, and color remain unset and will be defined by the layout model.

We have chosen the generic user interface description language XUL as a medium to describe the presentation model (and also the layout model). XUL provides a clear separation between the user interface definition (the various widgets that determine the UI) and its visual appearance (the layout and the "look and feel"). XUL is particularly suitable for 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).

Practically, our presentation model consists of a set of XUL fragments. Each fragment represents a single interaction element or a group of interaction objects. For the presentation model, presentation patterns embody building interface object blocks. In practice, instantiations of presentation patterns deliver XUL fragments which again describe user interface objects. Therefore each presentation pattern must have a mechanism to generate variants of XUL code depending on the assignment of their variables.

We have chosen to employ XUL Velocity templates in order to implement the patterns. Velocity is a Java-based template engine. Primarily the Velocity Template Language is meant to provide the easiest, simplest, and cleanest way to incorporate dynamic content in a web page [Velocity 2004]. However, Velocity can also be used to generate any form of mark-up code (including XUL). Within our approach, presentation patterns are used to

dynamically generate XUL code. If the variable parts of the presentation pattern change, conditions, loops and other control structures are used to adapt the template (pattern) accordingly.

One illustrative example of a presentation pattern is the *Form* pattern. It is applicable when the user must provide structural, logically related information to the application. On the left hand side of Figure 4-14, the interface of the *Form* Pattern is displayed showing that the various Input Fields to be displayed are expected as parameters. It is also illustrated that the *Unambiguous Format* pattern can be employed in order to implement the *Form* pattern. In particular, it is used in order to prevent the user from entering syntactically incorrect data. In conjunction with the *Form* pattern, it determines which interaction elements are displayed by the input form. Depending on the data type of the desired input, XUL code for the most suitable interaction element will be produced. Figure 4-13 shows the rendered interaction elements of three different instances of the *Unambiguous Format* pattern.

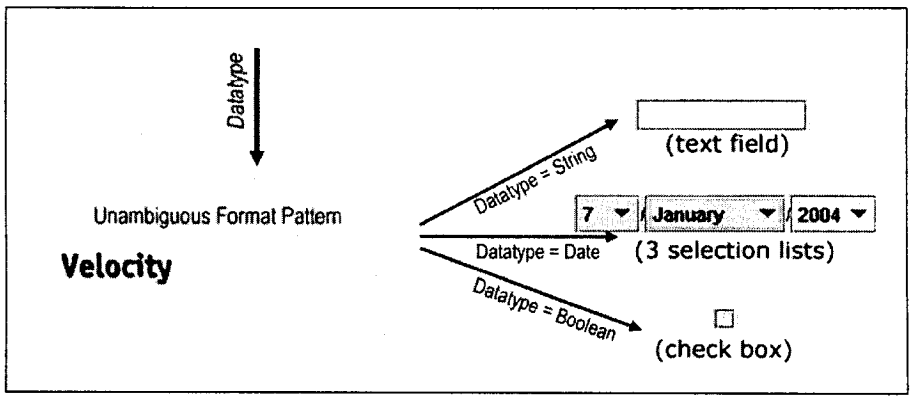

Figure 4-13: The *Unambiguous Format* Pattern and Three Different Instances of It

The right-hand side of Figure 4-14 shows the formalization of the *Form* pattern, which consists of a mixture of XUL and Velocity template code. The variable `$NUMBER` is used to determine the number of iterations of the `#foreach` loop. Within the loop, XUL code for displaying the Input fields is produced. In particular the Velocity template for the *Unambiguous Format* pattern is called in order to generate the XUL code for the

interaction element. The parameter `datatype`, passed to the template, determines which interaction element will be produced.

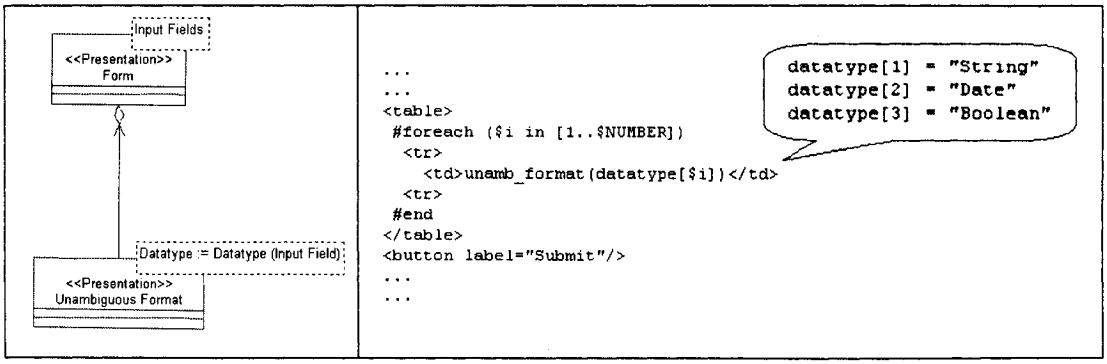

Figure 4-14: Interface and Implementation of the *Form* Pattern

Figure 4-15 portrays the rendered output of the instance of the *Form* pattern. The pattern has been adapted in order to allow three different inputs. One text or string input, one date input and one input of a Boolean value.

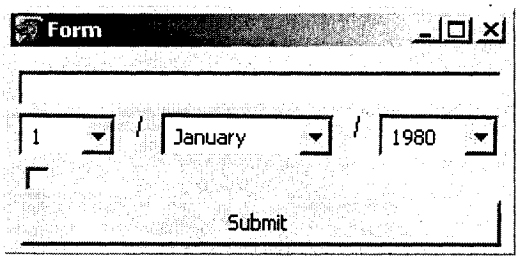

Figure 4-15: Rendered Output of an Instance of *Form* Pattern

Please note that the form of Figure 4-15 has been rendered using the default XUL style. The arrangement and the layout of the interaction elements have not been customized yet. Customizations such as the format of the date will be eventually defined in the Layout Model.

4.5. Patterns in Layout Management Modeling

In the layout model, the various XUL fragments of the presentation model are merged resulting in aggregated XUL code. The loosely connected XUL pieces are nested and associated together. The way these fragments are merged together depends on the overall layout of the entire application. In addition, the style attributes that are unset, are bound with concrete values.

The layout model defines how the loosely connected XUL fragments of the presentation model are aggregated and placed into relations. The merging process is performed according to an overall "floorplan" of the application. The floorplan depends on the type of application or site and its complexity.

Another aspect captured by the layout model is the binding of style attributes with concrete values. Usually a Web site or a GUI consists of several pages or windows. In order to maintain a consistent feeling across them, the same basic layout or floor plan should be kept throughout the entire interface [Tidwell 2004]. Depending on the purpose, the complexity, the in-house style and other attributes, a certain basic layout for the UI is chosen and kept throughout all sub-sites.

In the definition of the Layout Model, patterns can be employed with two different methods: (1) By providing a floor plan for the UI, and (2) By setting the style attributes of the various widgets of the UI.

The first method consists of determining the composition of the UI by providing a floor plan. Examples for such patterns are: Portal Pattern [Welie 2004], Card Stack [Tidwell 2004], Liquid Layout [Tidwell 2004] or Grid Layout [Welie 2004]. For example, Figure 4-16 shows the floor plan suggested by the Portal Pattern, which is applicable for web-based UIs.

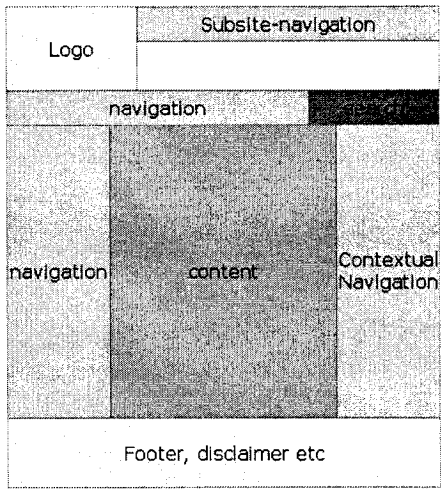

Figure 4-16: Floor Plan Suggested by the *Portal* Pattern [Welie 2004]

Another example is the *Labeling* Pattern, which simply attaches labels to the interaction objects and UI widgets in order to provide extra information about the current window / site and the displayed interaction elements. As Velocity templates can be used to generate XUL fragments (presentation model), they can also be used to aggregate XUL code. Therefore layout patterns are formalized as Velocity XUL templates as well.

In the second method, Layout Patterns are beneficial for setting the style attributes of the various widgets of the UI. 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 "hang together" and look like one thing. Ideally, the instance of the *House Style* Pattern delivers as a so-called XUL Skin. A XUL Skin is a style sheet which contains style information for elements. It was originally designed for HTML elements but can be applied to XUL elements as well, or to any XML for that matter. The style sheet contains information such as the fonts, colors, borders, and size of elements [XUL 2004b]. Applying the same style sheet to all pages / sites of the application will lead to a consistent overall appearance of the entire application.

Eventually, the established layout model (XUL code) is used as input for the automatic generation process in which the concrete interface is generated. Please note that the same layout model can be rendered to different target platforms such as Java Swing and Mozilla /Netscape.

After describing how and which patterns are applicable within our development approach, I will now introduce the tool Task-Pattern-Wizard for selecting and applying task and feature patterns.

4.6. The Task-Pattern-Wizard

In this section I will introduce my tool Task-Pattern-Wizard by providing a brief overview on its functionality and usage. In addition I will reveal some implementation details by describing the Task Pattern Markup Language (TPML) and the internal structure of the application.

4.6.1. Overview

In order to choose and apply patterns efficiently, tool support is necessary. By integrating the idea of patterns into development tools, patterns can be a driving force throughout the entire UI development process. Therefore, I have implemented a prototype of a task pattern wizard, which is aiming to support all phases of pattern application for the task model, ranging from pattern selection to pattern adaptation, to pattern integration. After parsing the pattern, the tool guides the user, step by step through the pattern adaptation and integration process. Figure 4-17 shows a screenshot of the start screen. In what follows, I will provide a brief description of Task-Pattern-Wizard usage. In particular, I will outline how each phase of pattern application is supported by the tool.

Figure 4-17: The Task-Pattern-Wizard

4.6.2. Task-Pattern-Wizard Usage

This section illustrated the use of the Wizard during the process of identifying, selecting, adapting and integrating task and feature patterns.

Identification:

In the identification phase a node of an already existing task structure, to which a pattern will be applied, is identified. The task description must be specified using the XIML notation. After opening the XIML file, the Task-Pattern-Wizard uses a tree element to display the task structure. Next, the user chooses an appropriate node representing a particular task. The user is then asked to decide how the new task structure, brought in by the pattern, should be integrated into the existing tree. The new task structure can either be integrated as an optional or as an iterative element. Optional tasks can be executed, but they are not mandatory. Iterative tasks are performed repetitively. In addition, the temporal relations of the new task with the other tasks can be defined. Figure 4-18 shows a screenshot taken during the identification of a target node from the task model.

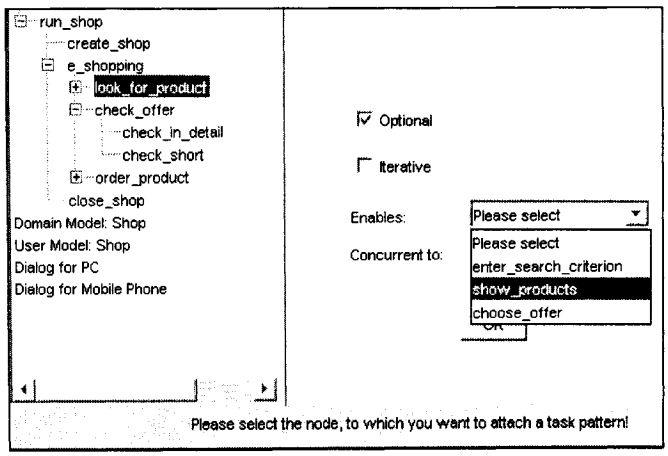

Figure 4-18: Selection screen of the Task-Pattern-Wizard

Selection:

In the selection phase, an applicable pattern is chosen. In order to perform selection, the Task-Pattern-Wizard presents the currently opened pattern according to the Alexandrian form [Alexander et al. 1977]. In particular, it is displayed in narrative form: What problem will be solved, when the pattern can be applied (context), how it works (solution), and why it works (rationale).

Adaptation:

After choosing an appropriate pattern, it must be adapted to the current context of use. Generally each pattern contains variables which 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 process of pattern adaptation, the Task-Pattern-Wizard will question the user to enter values for these variables. Then, in a top-down process, each occurrence of the substitution variable will be replaced (substituted) with this value.

Process variables are used to describe the structure of the task fragment, which will be created by the pattern. For example, entering values into a form is very repetitive. The same basic task (enter a value) appears over and over again. Basically each peer task can

just be distinguished by its name and the kind of input. Thus, instead of describing each of these tasks on its own, process variables that signal the number of respective tasks can be used.

After defining all variables, a pattern instance is created, which can be integrated into the task model.

Integration:

In the integration phase, the pattern instance will be incorporated into the current task model. Basically a new branch will be added to the task tree or an existing branch will be replaced. The new modified task structure can then be saved in XIML format. Within our tool set, XIML serves as a universal exchange format. Thus, tools such as the XIML-Task-Simulator and the Dialog-Graph-Editor can further process the new task model [Forbrig et al. 2003b].

4.6.3. Technical Issues

After briefly outlining the usage of the Task-Pattern-Wizard I will now reveal some technical details.

4.6.3.1. The Task Pattern Markup Language

TPML is an acronym for Task Pattern Markup Language and is graphically displayed in Figure 4-19. I have developed the TPML XML Schema as a notation for task and feature patterns.

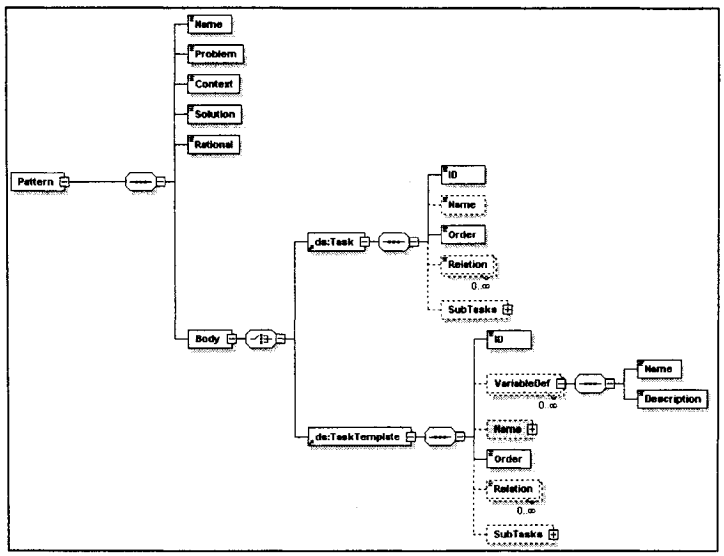

Figure 4-19: Structure of the Task Pattern Markup Language

The TPML Schema consists of the classic elements of patterns like Name, Problem, Context, Solution and Rational. However, these attributes are primarily used only to select an appropriate pattern. The implementation of the pattern has been formalized in the "Body" part. At this point we distinguish between *Task* and *TaskTemplates*. *Tasks* are further decomposed into *SubTasks* and contain no variable parts. Thus they can be adopted 1:1 without further adaptation. On the contrary, *TaskTemplates* are hierarchically structured as well, but also contain variable definitions and variables and must therefore be adapted first.

4.6.3.2. Class Structure

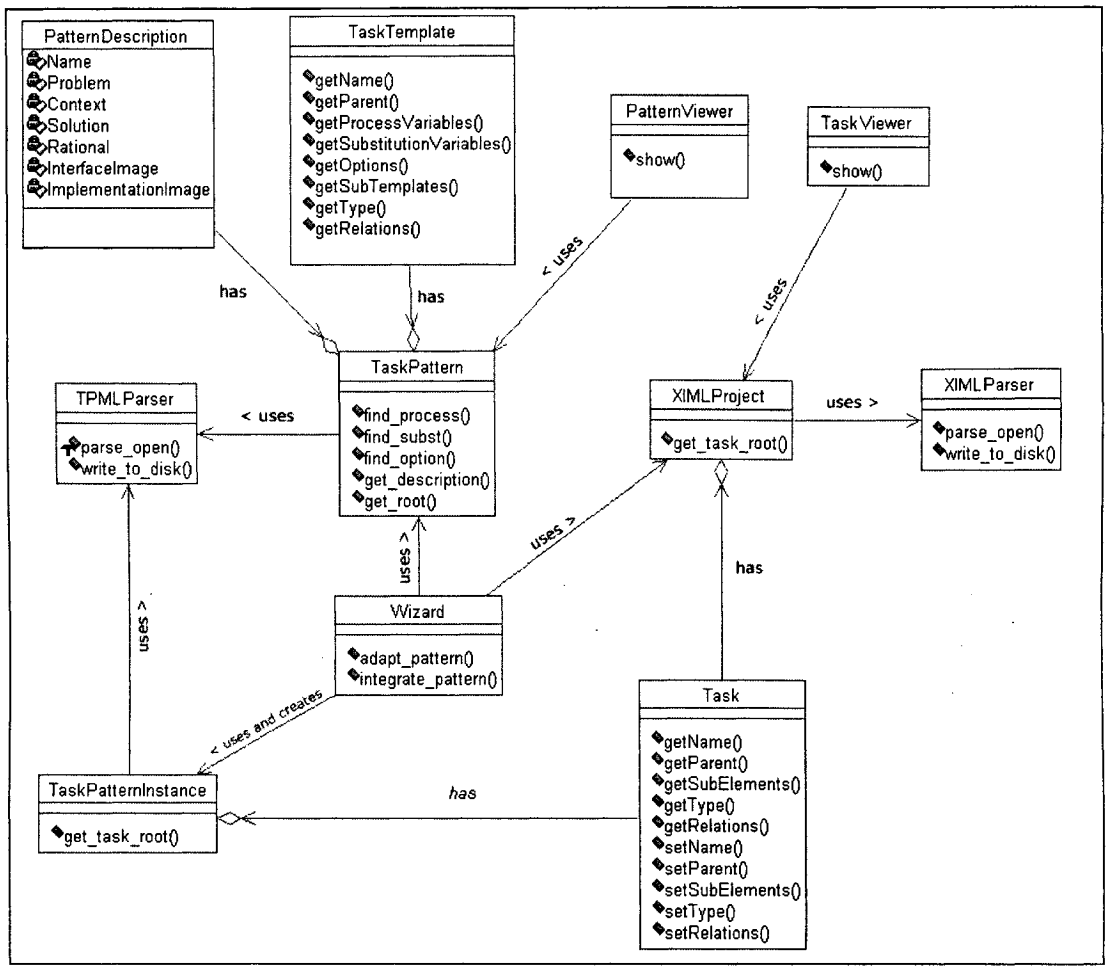

Figure 4-20: Class Structure of Task-Pattern-Wizard

The Task-Pattern-Wizard is a JAVA implementation, which has been developed mainly using the JBuilder 9 IDE. Figure 4-20 shows the basic class structure of the Task-Pattern-Wizard. For the sake of simplicity, only concrete classes and their public attributes and methods are displayed.

At the heart of the application is the *Wizard* class, which is responsible for adapting task patterns (*TaskPattern*) to pattern instances (*TaskPatternInstance*) and for integrating the pattern instances into the existing task model. The task model is specified by the class *XIMLProject*. A task model (*XIMLProject*) can be opened from File and saved to disk

using the methods "parse_open" and "write_to_disk" of the *XIMLParser* class. Physically, task models are stored in XML format using the grammar of XIML. Each XIML Project contains a set of Tasks (*Task*). Starting form the task root ("get_task_root"), it is possible to navigate through the task tree by using the *Task* methods "getSubElements" and "getParent". It also possible to change the task attributes by using the "set...." methods of *Task*.

Task and Feature patterns are specified by the *TaskPattern* class. The patterns are stored in XML format according to the XML Schema of TPML. In order to open a pattern, the "parse_open" method of the *TPMLParser* is used.

The TPMLParser and XIMLParser have been implemented based on the Dom4J libraries [DOM4J 2000]. Dom4J is an open source library for working with XML, XPath and XSLT on the Java platform using the Java Collections Framework with full integration with DOM, SAX and JAXP.

Each task and feature pattern contains a set of Task Templates (*TaskTemplate*). The Wizard class can access the *TaskPattern* class by using its methods "get_root" (delivers the root TaskTemplate), "find_process" (finds process variables), "find_subst" (finds substitution variables) and "find_option" (finds optional paths within the pattern). Each of these methods returns an instance of the *TaskTemplate* from which detailed information about the variables and the encapsulated task structure can be revealed.

After pattern adaptation, a pattern instance (*TaskPatternInstance*) will be created by the wizard. This instance can again be saved to disk by using the "write_to_disk" method of the *TPMLParser* class. Please note that in contrast to a task pattern, a task pattern instance consists of set of Tasks (*Task*).

In order to view task patterns and the task model, the classes *PatternViewer* and *TaskViewer* have been implemented. On the one hand, the task viewer uses *XIMLProject* in order to display the task model. On the other hand, the pattern viewer accesses, via the

TaskPattern class, the pattern description (*PatternDescription*) in order to display the various attributes of a pattern.

After describing the core constituents of our pattern-driven model-based framework, I will exemplify its practical implementation by elaborating an extended example in the next chapter.

5. Illustrative Example: Developing the UI of a Hotel Management Application

In what follows, I will develop a non-functional interface prototype of a simplified hotel management application, in order to illustrate and clarify the core ideas of my approach and its practical relevance. In particular, this brief case study will exemplify the application of patterns in a certain context.

5.1. Requirements

The management of a hotel is going to be computerized. The hotel's main business is renting rooms of different types. There are 40 rooms available with prices depending on their characteristic. The administrative office of the hotel needs a tool capable of booking a room for a specific customer. In particular, the main functionality of the application consists of: Adding a customer to the internal database and booking an available room for a registered customer. Moreover, only certified users should be able to access the main functionality of the program. Eventually the application should be running on WIMP-based systems.

Please note that only a simplified version of a hotel management system will be developed. I am not going to tailor the application and the corresponding models to different platform and user roles. The main purpose of the example is to show that model based UI design consists of a series 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 different patterns are used to establish the various models and to transform one model into another one.

5.2. The Envisioned Task Model

5.2.1. The Extended CTT Notation

In order to visualize the task model, I have used an extended version of the ConcurTaskTree (CTT) notation. In addition to the predefined task types, I have enhanced CTT by adding a fifth task type: The Pattern Task. Figure 5-1 shows the graphical representation of the Pattern Task.

Pattern Tasks: A Task or Feature Pattern is used to describe a set of tasks which can be composed of any previously introduced task types. Generally the pattern must be instantiated in order to derive a concrete task structure.

Figure 5-1: Symbol of the Pattern Task

5.2.2. The Course Grained Task Model

Figure 5-2 depicts the course grained task structure of the envisioned hotel management application. Only high-level tasks and their interrelations are portrayed. An impression about the overall structure and behavior of the applications is given. The shown structure is relatively unique for a hotel management application. It can be hardly re-used or copied from other projects or attempts. Also, the concrete "realization" of the high level tasks is omitted. The Pattern Task symbol is used as a placeholder for the suppressed task fragments.

Many interactive applications can be composed in good part out of a fixed set of re-usable 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 more re-usable task structures, which have been gained or captured from other projects

or applications, can be employed. In our case these re-useable task structures are documented in the form of patterns. This ensures an even higher degree of re-use, since each pattern can be adapted to the current context of use.

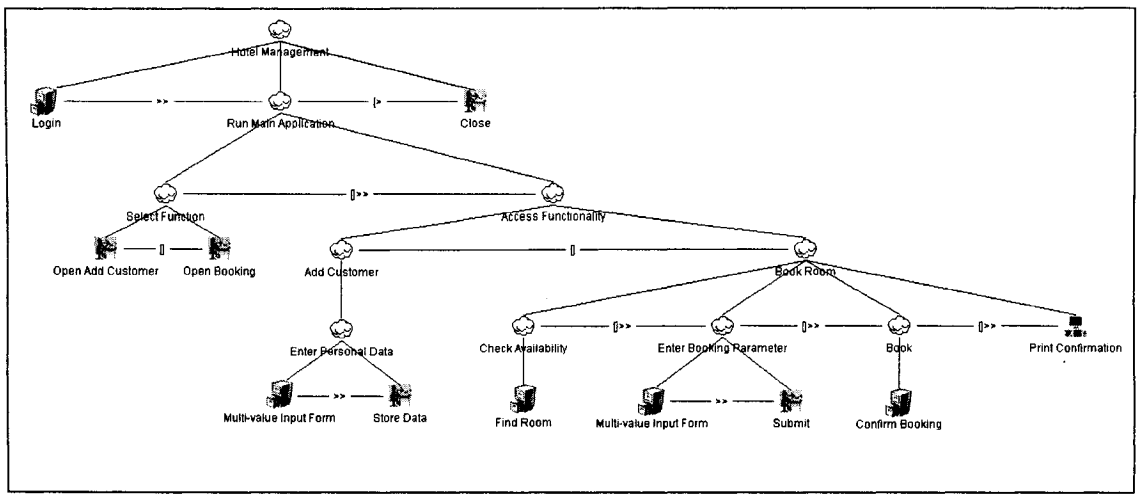

Figure 5-2: Course Grained Task Model of the Hotel Management Application

The main characteristic of the envisioned hotel management application, modeled by the task structure of Figure 5-2, can be outlined as follows:

A Login is necessary in order to access the main functionality of the application (Login task enables management task). The key features are "Adding a customer" by entering its personal coordinates and "booking a hotel room" for a specific customer. Both tasks can be performed in arbitrary order. The booking process consists of four consecutively performed tasks (related through "Enabling with Information Exchange" operators):

- Finding an available room,
- Assigning the room to a customer,
- Confirming the booking, and
- Printing a confirmation

As shown in Figure 5-2 the patterns: *Login*, Multi-*value Input Form*, *Find* and *Dialog* can be used in order to complete the task model at the lower levels. The application of these patterns will be described in the next section.

5.2.3. Completing the Task Model – The Application of Patterns

In the following I will describe how task and feature patterns can be used as building blocks to complete the task model of our example. The identification, adaptation and application of each pattern have been performed interactively by using the Task-Pattern-Wizard. Please note that all used patterns can be found in the appendix. All patterns are documented according to the "Alexandrian" form [Alexander et al. 1977]. In addition, they contain an implementation description for the task model and a description of their interface and composition structure. In some cases the pattern does not resolve the task completely and some tasks remain, which can be further decomposed. In such cases, other patterns may apply or the decomposition must be performed "manually".

Completing the Login Task

As portrayed in Figure 5-3 the *Login Pattern* can be used in order to describe the login task. The variables of the *Login Pattern* are the coordinates, which must be entered in order to start the authentication process. In order to instantiate this pattern, these variables must be assigned with values.

The *Login Pattern* suggests the consecutive execution of the tasks: "Show Login Prompt", "Enter Coordinates", "Submit Coordinates" and "Provide Feedback". The pattern also recommends that the *Multi-value Input Form Pattern* be used in order to realize the "Enter Coordinates" task.

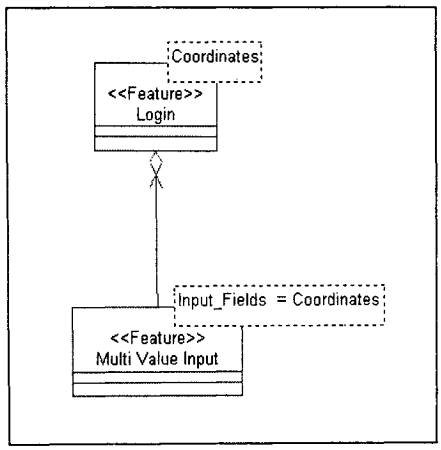

Figure 5-3: Interface and Structure of the *Login* Pattern

The *Multi-value Input Form Pattern* suggests a task structure for entering several values into an input form. As shown in Figure 5-3 the Coordinate variables of the *Login Pattern* are further passed to the *Multi-value Input Form Pattern* in order to determine the input fields of the input form.

If we employ the Task-Pattern-Wizard for the application of the pattern, the user is asked to determine each input field, which is required for the login process. For the sake of the hotel management application the authentication of the login feature will be realized with three values: Username, hotel chain and password. As a consequence, the Task-Pattern-Wizard would deliver the task structure visualized in Figure 5-4.

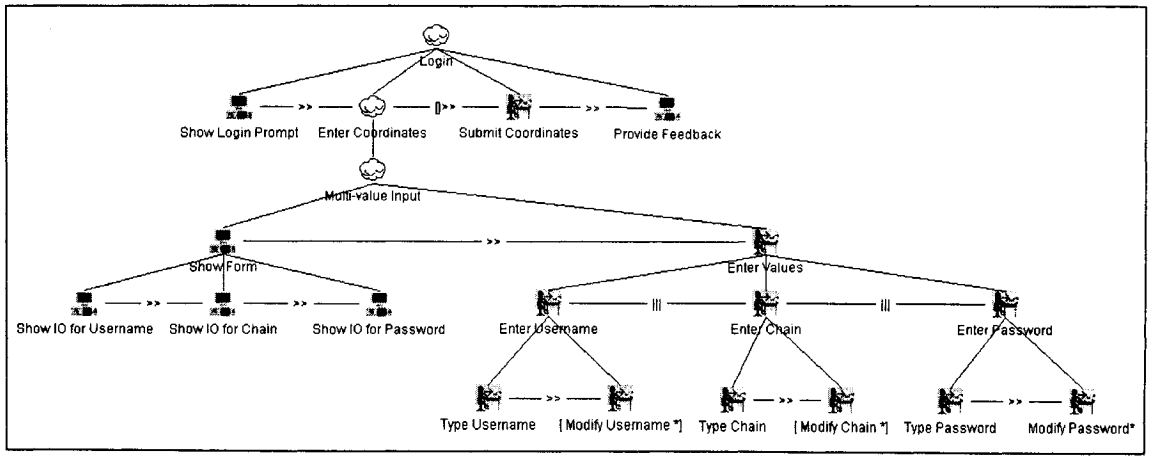

Figure 5-4: Concrete Realization of the *Login* Pattern

Completing the "Enter Personal Data" Task

The "Enter Personal Data" task of our example materializes the input of customer characteristics. Different values of different types are entered into a form. Consequently the *Multi-value Input Form Pattern*, which has been already employed within the Login Pattern, is well suited for this task. The customer characteristics, which must be entered to the hotel management application, are: "Customer ID", "Name", "DOB", "Nationality", "Street", "Street number", "City", "Province", and "Country". After adapting the pattern by using the Task-Pattern-Wizard, a respective task structure will be created, which will be inserted into the hotel management task model.

Completing the Find Room Task

In order to complete the "Find Room" task, the *Find Pattern* is applicable. In contrast to the patterns already used in this example, the *Find Pattern* suggests a number of options rather than providing a task structure. Figure 5-5 shows that according to the pattern, finding an object can be performed either by searching, browsing, or employing an agent.

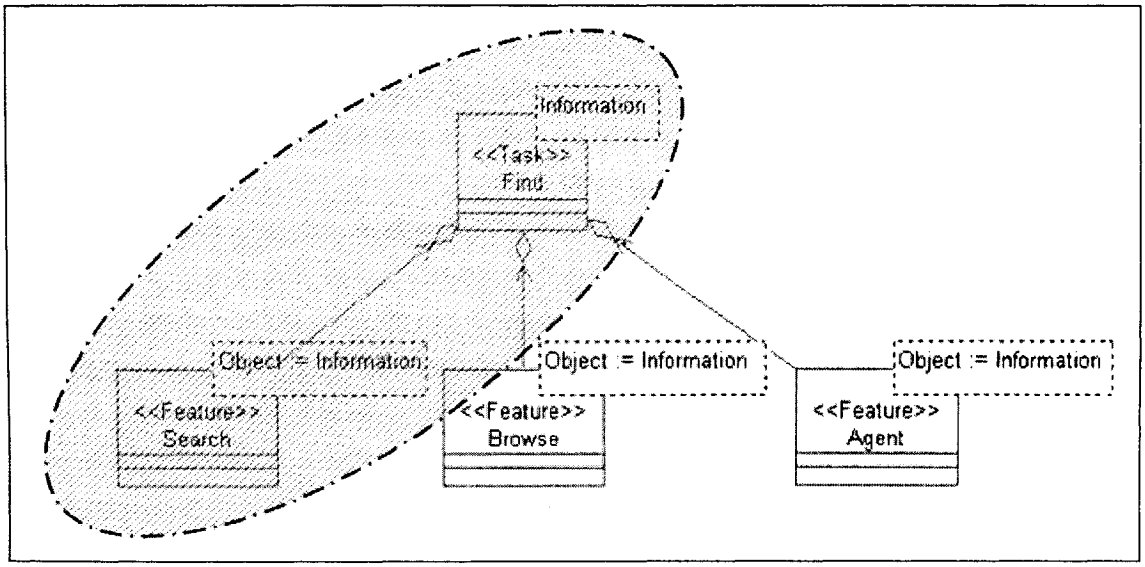

Figure 5-5: Interface and Structure of the *Find* Pattern

Within the hotel management application, the task of finding an available room should only be performed by searching according to some query parameters. As shown in Figure 5-5, the "Information" variable of the *Find Pattern* (in this case placeholder for the value "Hotel Room") is used to assign the "Object" variable of the *Search Pattern.*

The *Search Pattern* suggests a structure in which the search queries are first entered and 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. In order to search for an available room, the following search parameters should be used: "Arrival date", "Departure date", "Non- Smoking", "Double / Single", "Room Type". After submitting the search queries, the search results (in this case the applicable hotel rooms) can be either manually looked through by using the *Browse Pattern* or, on basis of the search results, a refinement search can be performed by employing the *Search Pattern* recursively. For the scope of this case study, a refinement search is not necessary and the search results should only be browsed.

According to the *Browse Pattern,* the list of objects (the hotel rooms) is first printed which can be, in a second step, interactively browsed. Details of the hotel room can be viewed by

selecting it. In order to print out properties about an object, the *Print Object Pattern* can be used. It suggests using application tasks in order to print values about an object which can be directly or indirectly derived from the object attributes. In the case of the hotel management application the following attributes about the hotel room should be printed: "Room Number", "Smoking / Non-Smoking", "Double / Single", "Room Type", and "Available until".

After adapting all patterns to the purposes of the hotel management application, the task structure displayed in Figure 5-6 is derived. Please note that the "Make decision" task has been added manually, without pattern support.

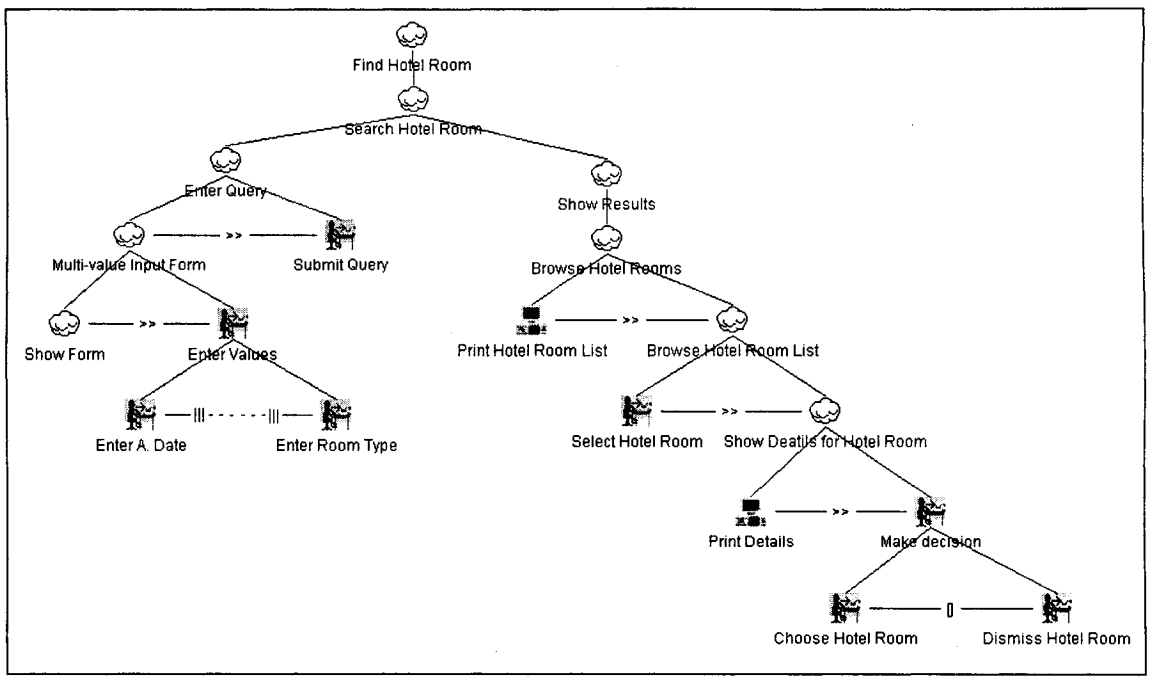

Figure 5-6: Concrete Task Structure Delivered by the *Find* Pattern

Completing the Enter Booking Parameter Task

This task consists of simply entering the ID of the Customer, in order to assign the corresponding hotel room. Again in order to realize this task, the *Multi-value Input Form*

Pattern can be employed. The pattern will be adapted in order to create a task structure for the input of just one value.

Completing the Confirm Booking Task

This task acts as a "safety net" for the user of the hotel management application. It double-checks with the user if the booking of the room should be really executed. Often such a confirmation is implemented as a dialog box, where the user must confirm or cancel the transaction, which is about to be executed. Consequently, the *Dialog Pattern* can be employed at this point. The *Dialog Pattern* suggests a task structure where the user must choose one option out of many. For the purpose of our example, the user can choose between "Confirm Booking" and "Cancel Booking".

Evaluation of the Task Model

Eventually after all patterns have been adapted and instantiated, a first draft of the envisioned task model can be derived. At this point first evaluations can be carried out. For instance the XIML-Task-Simulator can be used in order to simulate and animate possible scenarios. Figure 5-7 shows a screenshot the hotel management task model using the XIML-Task-Simulator. Based on the result of the evaluation, early modifications and improvements of the task model are possible. For example it can be recognized that the Entering the Customer ID (shaded ellipse in Figure 5-7) is not enough in order to book a particular room. Furthermore, it is necessary to pick a time span for the booking of the room. This modification can be simply performed by re-adapting the corresponding *Multi-value Input Form Pattern* by using the Task-Pattern-Wizard.

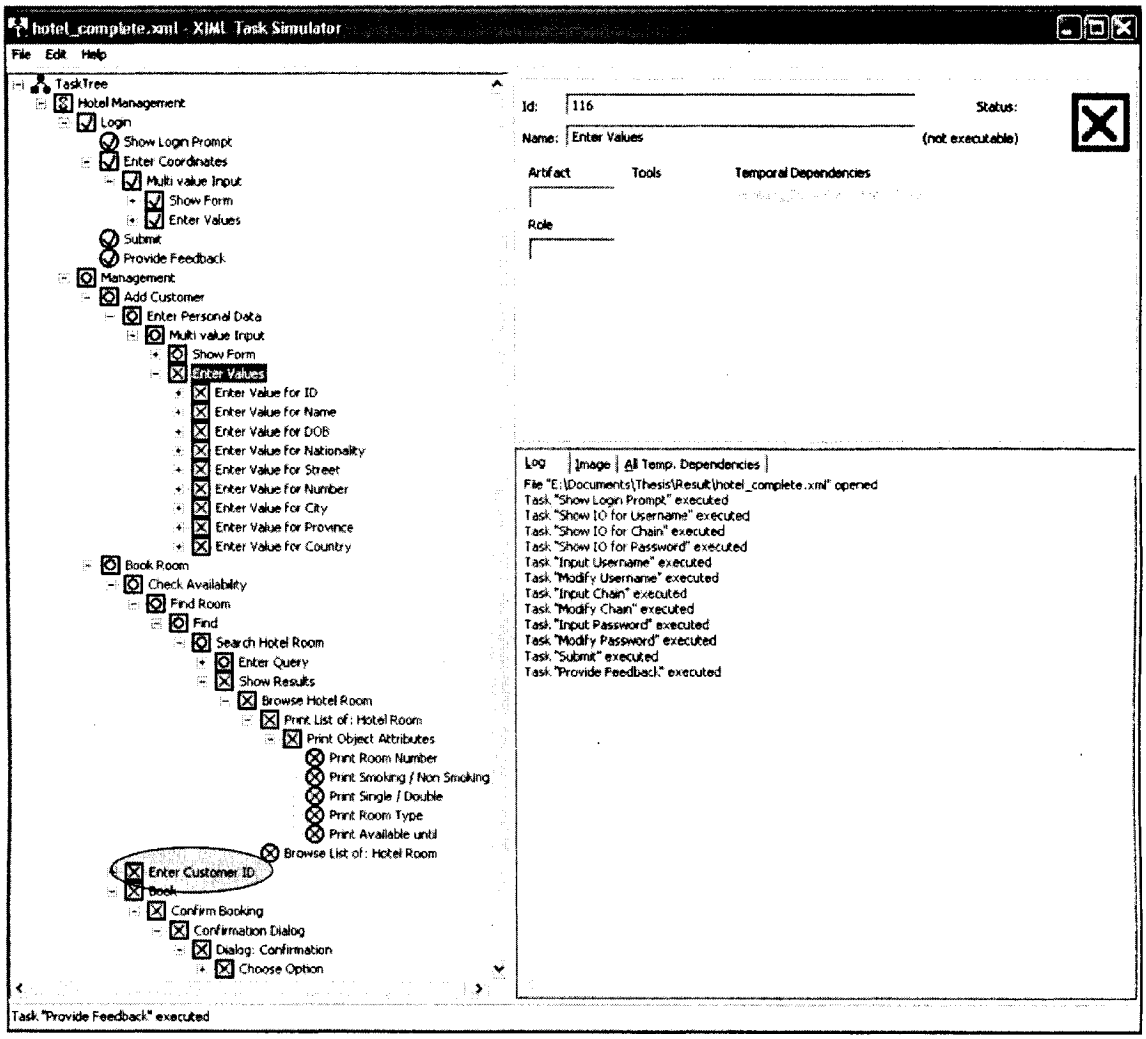

Figure 5-7: Simulation and Animation of the Hotel Management Task Model

5.3. Designing the Dialog Structure

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

5.3.1. Grouping the Tasks

For the hotel management application it is expected that on each dialog view the user must perform at least one interactive task in order to initiate the transition to another dialog view. Thus, in this case, in order to define the various dialog views and their transitions, it is enough to group only the interactive tasks. Using the tool Dialog-Graph-Editor the following task groupings, visualized by Table 5-1, have been made:

Table 5-1: Tasks Grouped to Dialog Views

Dialog View	Tasks
Login	Enter Value for Username
	Enter Value for Chain
	Enter Value for Password
	Submit
Main Menu	Open Add Customer
	Open Booking
	Close
Add Customer	Enter Value for Customer ID
	Enter Value for Name
	Enter Value for DOB
	Enter Value for Nationality
	Enter Value for Street
	Enter Value for Number
	Enter Value for City
	Enter Value for Province
	Enter Value for Country
	Store Data
Search Applicable Room	Enter Value for Arrival Date
	Enter Value for Departure Date
	Enter Value for Smoking / Non - Smoking
	Enter Value for Single / Double
	Enter Value for Type

	Submit Query
Browse Results	Select Hotel Room
Show Room Details	Choose Room
	Dismiss Room
Enter Booking Parameters	Enter Value for Customer ID
	Enter Value for Arrival Date
	Enter Value for Departure Date
	Submit
Confirm Booking	Select Book Room
	Select Cancel Booking
Print Confirmation	Select OK

Please note that if a task is not a leaf (atomic) task all its subtasks are also grouped into the dialog view.

5.3.2. Defining Transitions

After grouping the tasks to dialog views, transitions between the various dialog views must be defined. In addition, tasks must be depicted which initiate the transition from one dialog to another one. Within the scope of our example, three different types of dialog views have been used: Single dialog view, multi dialog view, modal dialog view and end dialog view. The visualization for the various dialog view types used by the Dialog-Graph-Editor are displayed in Figure 5-8. Furthermore, the entry point of the dialog graph must be determined by assigning one of dialog views as the starting point. The start dialog is visually flagged by the "traffic light" symbol (Figure 5-8).

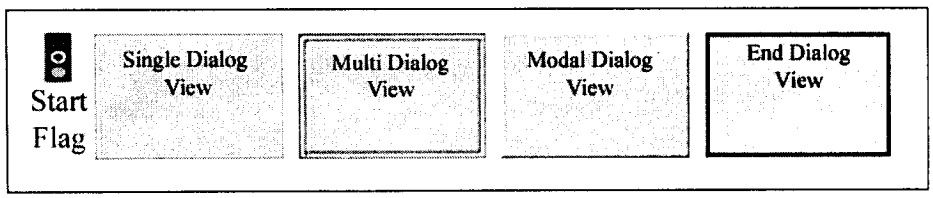

Figure 5-8: Different Types of Dialog Views

In order to design the dialog graph for the hotel management application we assigned the login dialog view to be modal and to be the starting dialog. After executing Submit, the "Main Menu" dialog will be opened. Thus, a sequential transition between both dialog views is defined. From the main menu either the "Add Customer" dialog view or the "Search Applicable Room" dialog view can be opened by a sequential transition. In addition, it is possible to close the entire application from the main menu. After completing the "Add Customer" dialog view, the main menu will be opened again. Consequently a sequential transition to the "Main Menu", initiated by the Store Data task, must be defined.

The Booking functionality of the application consists of a series of dialog views, which must be completed sequentially. At this point, the *Wizard* dialog pattern can be employed. It suggests a dialog structure where a set of dialog views will be arranged sequentially and where the "last" task of each dialog view will initiate the transition to the next dialog view. Figure 5-9 depicts the suggested graph structure of the *Wizard* pattern.

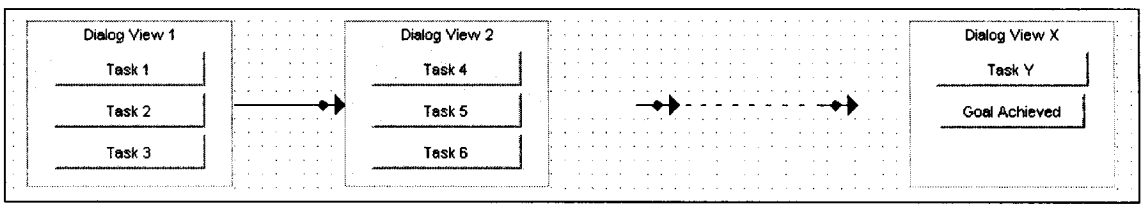

Figure 5-9 Graph Structure Suggested by the *Wizard* Pattern

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, in order to give the user the possibility to see the details of different rooms in parallel, the sequential structure of the booking process must be slightly modified. In particular, the dialog pattern *Recursive Activation* should be used to model this behavior. This pattern is used when the user wants to activate and manipulate several instances of a dialog view. In this case, the user will be able to activate and access several instances of the "Show Room Details" dialog view. In particular, this pattern suggests the following task structure: Starting from a source dialog view, a creator task is used in order 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.

In order to give the user the possibility to abort the booking transaction, a premature exit should be provided. This is realized by the "Confirm Booking" dialog view. At this point, the user can decide whether to book the room or to abort the transaction. Thus, another sequential transition must be defined, which is initiated by the Select Cancel Booking tasks and leads the back to the main menu. The complete dialog graph of the hotel management application is depicted in Figure 5-10.

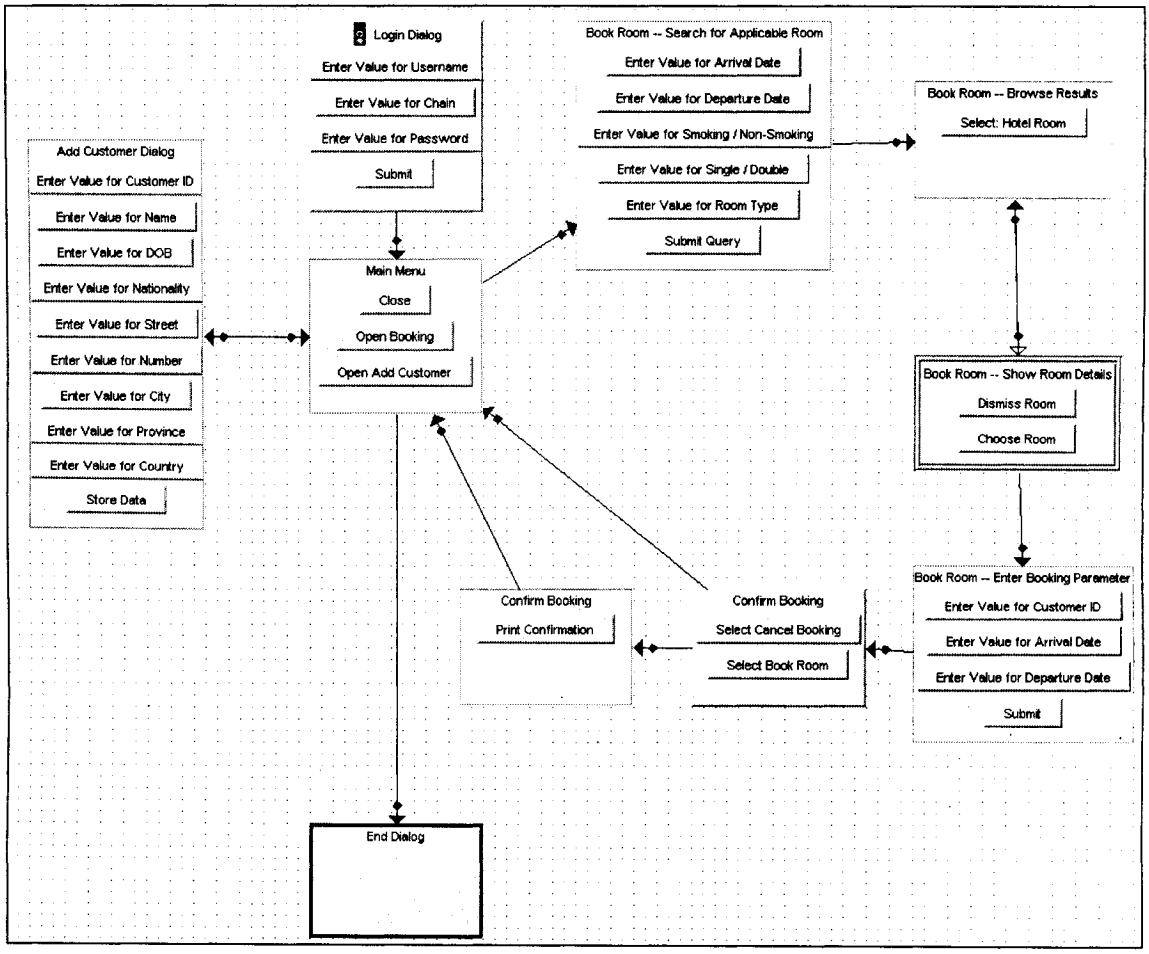

Figure 5-10: Dialog Graph of the Hotel Management Application

Next the defined dialog graph should be evaluated. Using the Dialog-Graph-Editor, it is possible the animate the dialog graph. A first abstract prototype of the user interface will be generated. It is possible to dynamically navigate through the dialog views by executing the corresponding tasks. This abstract prototype simulates the later navigational behavior of the final interface. It supports the communication between users and software developers. First, design decisions are immediately understandable for the user. Stakeholders are able to experiment with a dynamic system. Figure 5-11 shows a walkthrough though the dialog graph generated by the Dialog-Graph-Editor.

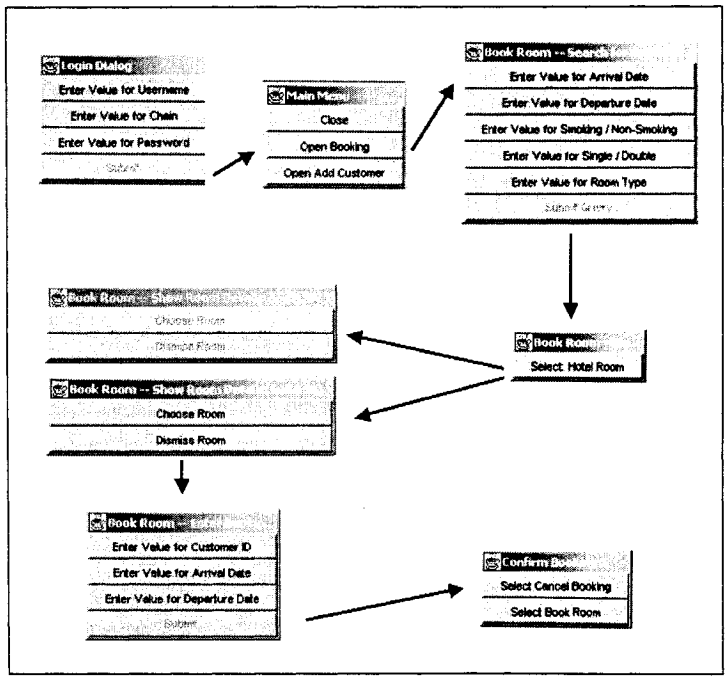

Figure 5-11: Abstract Prototype of Hotel Management Application

5.4. 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 such as forms, buttons and lists. Style attributes such as size, font, and color remain unset and will be defined by the layout model.

A good part of the user's tasks while using the application consists of providing structural textual information. The information can usually be split into data chunks which are logically related. At this point, the *Form* presentation pattern can be applied, which tackles exactly this issue. It suggests to use a form for each related data chunk populated with the necessary elements needed to enter the data. Moreover, the pattern refers to the *Unambiguous Format* pattern, which can be employed in conjunction with it.

The *Unambiguous Format* pattern is used in order to prevent the user from entering syntactically incorrect data. It uses information from the business object model in order to provide 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 which are chosen in a way that the user cannot enter syntactically incorrect data.

Figure 5-12 shows the prototypical windows generated from the XUL fragments of the presentation model of the hotel management application for the dialog views "Login", "Main Menu", "Add Customer" and "Find Room". All widgets and UI components are layed out and arranged according to the default style.

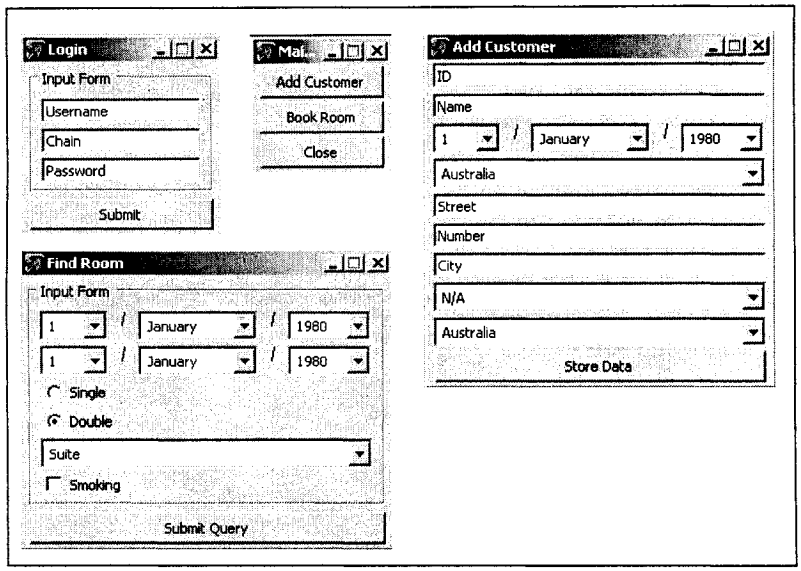

Figure 5-12 Screenshots of Visualized XUL Fragments of the Presentation model

In the layout model, the style attributes that have not been defined yet are set conforming to the standards of the hotel management application. According to the *House Style* Pattern, which is applicable here, colors, fonts, and layouts should be chosen in a way that gives the user the impression that all windows of the application hang together and appear like one thing. Practically, CSS style sheets (also called XUL skins) have been used to determine the visual appearance of the interface. In addition, in order to assist the user in

working with the application, meaningful labels should be provided. The *Labeling* layout pattern suggests employing labels for each interaction element. Using the grid format, the label should be 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 simple since the UI is not nested and consists of only one container. After establishing the layout model, the aggregated XUL code together with the corresponding XUL skins, can be rendered to the final user interface. Figure 5-13 shows the UI rendered on Windows XP.

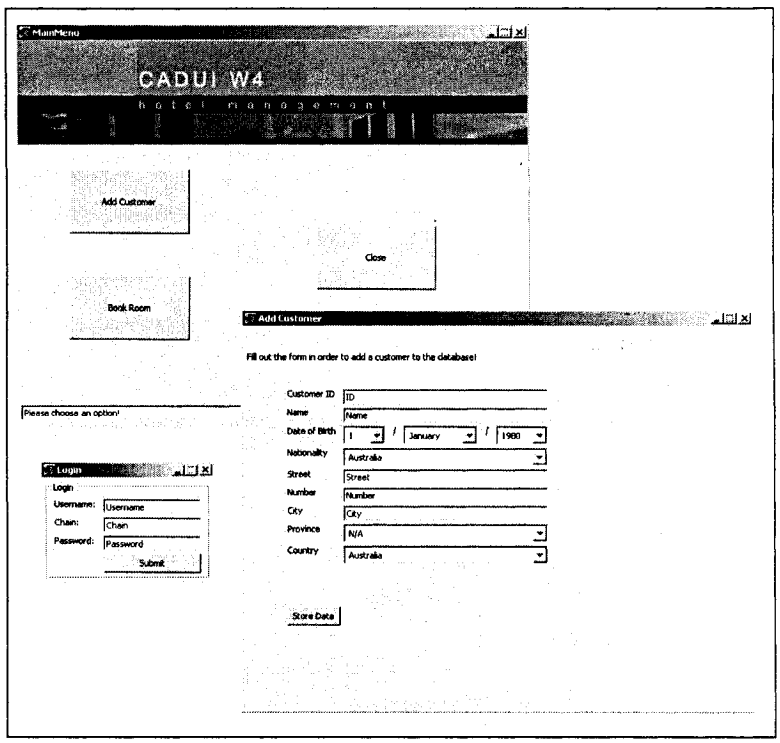

Figure 5-13: Screenshots of the Hotel Management Application

6. Conclusion and Future Investigations

In this thesis, I demonstrated how HCI patterns can be employed to enhance the model-based UI development approach. Within the proposed framework, patterns have been introduced with the aim to overcome the lack of reusability, which is one major limitation of the existing model-based UI development frameworks. In particular, I illustrated how different kinds of patterns can be used as building blocks for the establishment of task [Sinnig et al. 2003], dialog, presentation and layout models [Sinnig et al. 2004]. In order to foster reuse in different contexts of use, I defined the general process of pattern application, in which patterns are abstractions that must be instantiated. In addition, I described an interface for combining patterns and a possible formalization according to models. The applicability of the proposed pattern-driven model-based development approach has been exemplified using a comprehensive case study. A body of 13 different patterns has been compiled, formalized and used in the case study. Furthermore, I developed a tool for using, selecting, adapting and applying patterns to task models [Sinnig et al. 2003].

A major contribution of this research is the use of patterns to fully 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 thesis, I extended the pattern concept: First, by providing an interface for patterns in order to combine them. In this vein, I proposed the general process of pattern application, in which patterns can be customized to the particular context of use. Second, I transferred the pattern concept to the domain of model-based UI development. In order to foster reuse and to prevent reinventing the wheel, I demonstrated how task, dialog, presentation and layout patterns are used as building model blocks for the creation of the corresponding models, which are the core constitutes of my development approach. Within the scope of this thesis, the complicity of patterns and the envisioned task, dialog, presentation and layout models has been investigated. As a future avenue, patterns for other models such as the user model or object model (i.e. Design Patterns [Gamma et al. 1995] may be considered.

Another important contribution of this research work is the implementation approach of my framework. I developed a non-functional UI prototype of a hotel management application. Patterns were found and applied for each of the models used during development. Within the scope of the example, 13 different patterns relative to the various models were discovered, formalized and applied. The main purpose of the example was 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. However, I did not tailor the application and the corresponding models to different platform and user roles. In the future, we plan to develop a functional "realistic" application using the proposed pattern-driven model-based framework.

The last contribution was the development of a tool (Task-Pattern-Wizard) that can be effectively employed for establishing task models and selecting and applying patterns to it. I proposed TPML as a notation for "formalizing" task patterns, which can be processed by the Task-Pattern-Wizard. In addition, I illustrated how the XML derivates, XIML and XUL, can be used to represent the various models. In this vein, I also outlined how the Dialog-Graph-Editor and XIML-Task-Simulator can be used to establish and evaluate the task and dialog models.

Our framework is particularly useful in the context of MUI development [Javahery et al. 2003]. With the advent of the Internet, as well as the emergence of ubiquitous and pervasive computing, interactive systems are increasingly pollinating our everyday life. A plethora of new interaction devices has emerged. Interaction techniques and the size of the applications themselves have become more and more complex. Interactive applications must be aware of dynamically changing contexts and must withstand variations of the environment. In addition, different types of users fulfilling different roles must be accommodated by the system. As a result, a new challenge of allowing the greatest number of people access to interactive applications for the largest number of purposes and in the widest number of contexts has materialized [Paternò 2000]. In order to cope with these new challenges, the full potential and power of model-based approaches must be

exploited. In this thesis, I further enhanced the model-based framework in order to make it more applicable and practical.

My research has answered some questions about the role of patterns in model-based UI development. However, it has also led to some fundamental issues that need to be further investigated. One such future investigation deals with the way patterns should be documented and described. There are basically three types of knowledge encapsulation in patterns. First, there is the static documentation, which describes in a mostly narrative manner, the patterns as a solution to problems in a certain context. Second, a significant amount of knowledge can be extracted from the relationships with which the pattern is engaged. These relationships are context-sensitive. Depending on the current context of use, the pattern might or might not be related to another pattern. Third, the creation of a pattern instance by adapting the pattern to the context of use can be very complex. The pattern should be specified in a formal dynamic fashion. In that way, tools can be used in order to guide the user through the pattern adaptation process. Within this thesis, a first step in that direction has been made by proposing an interface for combining patterns and by approaching a possible formalization. In particular, the task patterns have already been specified in a way which allows for processing by the Task-Pattern-Wizard.

Another issue for future investigation deals with the evaluation and dissemination of patterns. In order to better understand pattern use, MOUDIL, the Montreal Online Usability Pattern Digital Library, is currently being developed [Gaffar et al. 2003]. It is targeted for software developers and usability engineers. MOUDIL is being designed with two major objectives: First, as a service to UI designers and software engineers to share HCI pattern information for UI development. Second, as a research forum for understanding how patterns are really discovered, validated, used and perceived. It is hoped that all this information can be helpful in further developing our understanding of patterns.

References

Abi-Aad, R., D. Sinnig, T. Radhakrishnan and A. Seffah (2003). CoU: Context of Use Model for User Interface Design. In *Proceedings of HCI International 2003*, June 2003, Greece, LEA, pp. 8 - 12.

Alexander, C., S. Ishikawa, M. Silverstein, J. M., F.-K. I. and A. S. (1977). *A Pattern Language: Towns, Buildings, Construction.* New York, Oxford University Press.

Alpert, S. (1998). *The Design Patterns Smalltalk Companion*, Addison-Wesley.

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

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

Bouillon, L. and J. Vanderdonckt (2002). Retargeting of Web Pages to Other Computing Platforms with VAQUITA. In *Proceedings of WCRE'02*, October 2002, Richmond, Virginia.

Breedvelt, I., F. Paternò and C. Severiins (1997). Reusable Structures in Task Models. In *Proceedings of Proceedings Design, Specification, Verification of Interactive Systems '97*, June 1997, Granada, Springer, pp. 251-265.

Chikofsky, E. J. and J. H. Cross (1990). Reverse Engineering and Design Recovery - A Taxonomy. *IEEE Software*, pp.13-17.

Chin, D. (1986). User Modeling in UC, the UNIX Consultant. In *Proceedings of CHI 1986*, pp. 24-28.

Coad, P. (1996). *Object Models: Strategies, Patterns, and Applications*, Prentice Hall.

Coplien, J. O. (1997). *The OrgPatterns website* [Internet]. Available from <http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns> [Accessed November, 2003].

Coram, T. and J. Lee (1998). *Experiences: A Pattern Language for User Interface Design* [Internet]. Available from <http://www.maplefish.com/todd/papers/experiences> [Accessed November, 2003].

da Silva, P. (2000). User Interface Declarative Models and Development Environments: A Survey. In *Proceedings of DSV-IS'2000*, Springer, pp. 207–226.

DOM4J (2000). *Dom4J Project Information* [Internet]. Available from <http://www.dom4j.org/project-info.html> [Accessed January, 2004].

D'Souza, D. (2001). *Model-Driven Architecture and Integration: Opportunities and Challenges, Version 1.1.* [Internet]. Available from <http://www.catalysis.org/publications/papers/2001-mda-reqs-desmond-6.pdf> [Accessed February, 2004].

Duyne, D., J. Landay and J. Hong (2002). *The Design of Sites*, Addison Wesley.

Erickson, T. (2000). Lingua Francas for Design: Sacred Places and Pattern Languages. In *Proceedings of Designing Interactive Systems*, August 17-19, 2000, New York, ACM Press.

Forbrig, P., A. Dittmar and A. Mueller (2003a). Adaptive Task Modelling: From Formal Models to XML Representations. *Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces*, London, Welie, pp. 171-192.

Forbrig, P., A. Dittmar, D. Reichart and D. Sinnig (2003b). User-Centred Design and Abstract Prototypes. In *Proceedings of BIR 2003,* Berlin, SHAKER, pp. 132 - 145.

Gaffar, A., D. Sinnig, H. Javahery and A. Seffah (2003). MOUDIL: A Comprehensive Framework for Disseminating and Sharing HCI Patterns. In *Perspectives on HCI patterns: Concepts and Tools: A Workshop at ACM CHI 2003,* Feb. 2003, Florida, USA.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). *Design Patterns: Elements of Object-Oriented Software*, Addison-Wesley.

Grand, M. (2002). *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, John Wiley & Sons.

Granlund, A. and D. Lafreniere (1999). PSA: A pattern-supported approach to the user interface design process. In *UPA 99: A Workshop at Usability Professionals Association Conf,* July 1999, Scottsdale, AZ.

ISO / IS 8807, Information Process Systems - Open Systems Interconnection - LOTOS- A Formal Description Based on Temporal Ordering of Observational Behaviour, 1988.

Javahery, H. (2003). Pattern-oriented Design for Interactive Systems. Master Thesis in the Department of Computer Science, Concordia University, Montreal.

Javahery, H. and A. Seffah (2002). A Model for Usability Pattern-Oriented Design. In *Proceedings of TAMODIA 2002*, July 18-19 2002, Bucharest, Romania, pp. 104-110.

Javahery, H., A. Seffah, D. Engelberg and D. Sinnig (2003). Migrating User Interfaces between Platforms Using HCI Patterns. *Multiple User Interfaces: Multiple-Devices, Cross-Platform and Context-Awareness*, Wiley.

Laakso, S. (2003). *User Interface Design Patterns* [Internet]. Available from < http://www.cs.helsinki.fi/u/salaakso/patterns/> [Accessed Nov., 2003].

Li, N. (2001). Usability Patterns-Assisted Design for Web User Interfaces. Masters Thesis in the Department of Computer Science, Concordia University, Montreal.

Limbourg, Q. and J. Vanderdonckt (2004). Transformational Development of User Interfaces with Graph Transformations. In *Proceedings of CADUI 04*, 16. -19. Jan 2004, Funcal, Portugal.

Markopoulos, P. and P. Marijnissen (2000). UML as a representation for Interaction Design. In *Proceedings of OZCHI.*

Märtin, C. (1996). Software Life Cycle Automation for Interactive Applications: The AME Design Environment. In *Proceedings of CADUI'96*, June 1996, Namur, Belgium, Namur University Press, pp. 57-76.

Marucci, L., F. Paternó and C. Santoro (2003). Supporting Interactions with Multiple Platforms Through User and Task Models. *Multiple User Interfaces, Cross-Platform Applications and Context-Aware Interfaces*, London, Wiley, pp. 217-238.

MDA (2004a). *OMG Architecture Board MDA Drafting Team, "Model-Driven Architecture: A Technical Perspective"* [Internet]. Available from <ftp://ftp.omg.org/pub/docs/ab/01-02-01.pdf> [Accessed February, 2004].

MDA (2004b). *OMG Model-Driven Architecture Home Page* [Internet]. Available from <http://www.omg.org/mda/index.htm> [Accessed 2004, February].

Mens, T., C. Lucas and P. Steyaert (1998). Supporting Disciplined Reuse and Evolution of UML Models. In *Proceedings of UML98: Beyond the Notation*, June 3-4, 1998, Mulhouse, France, Springer., pp. 378-392.

MOBI-D (1999). *The MOBI-D Interface Development Environment* [Internet]. Available from <http://smi-web.stanford.edu/projects/mecano/mobi-d.htm> [Accessed February, 2004].

Molina, P. (2004). A Review to Model-Based User Interface Development Technology. In *Making Model-based UI Design Practical: Usable and Open Methods and Tools: A Workshop at IUI 2004,* January, 2004, Madeira, Portugal.

Molina, P., S. Meliá and O. Pastor (2002). JUST-UI: A User Interface Specification Model. In *Proceedings of CADUI 2002*, May 15-17, 2002, Valenciennes, France, pp.

Molina, P. and H. Trætteberg (2004). Analysis & Design of Model-based User Interfaces. In *Proceedings of CADUI 2004*, 13.-16. Jan 2004, Funchal, Protugal, pp. 211-222.

Moore, M. (1996). Representation Issues for Reengineering Interactive Systems. *ACM Computing Surveys*, 28/4.

Myers, B. (1995). User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, 2, pp.64-103.

Nielsen, J. and R. Mack (1994). *Usability Inspection Methods*. New York, John Wiley & Sons.

Novak, G. S. (2004). *Automatic Programming* [Internet]. Available from <http://www.cs.utexas.edu/users/novak/index.html> [Accessed January, 2004].

OliverNova (2004). *CARE Technologies* [Internet]. Available from <http://www.care-t.com/> [Accessed January, 2004].

Paquette, D. and K. Schneider (2004). Interaction Templates for Constructing User Interfaces from Task Models. In *Proceedings of CADUI 2004*, 13.-16. Jan, Funchal, Protugal, pp. 223 - 235.

Parnas, D. L. (1994). Software Aging. In *Proceedings of 16th International Conference on Software Engineering*, May 16 - 21 1994, Sorrento Italy, IEEE Press, pp. 279 - 287.

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

Paternò, F. and C. Santoro (2002). One Model, Many Interfaces. In *Proceedings of CADUI'02*, May 15-17, Valenciennes, France.

Prasad, S. (1996). Models for Mobile Computing Agents. *ACM Comput. Surv.*, 28 (4).

Puerta, A. (1996). The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In *Proceedings of CADUI 96,* Namur, Belgium, Presses Universitaires de Namur, pp. 19-25.

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

Puerta, A. and J. Eisenstein (1999). Towards a General Computational Framework for Model-Based Interface Development Systems. In *Proceedings of IUI'99*, 5-8 January 1999, Redondo Beach, CA, ACM Press, New York, pp. 171-178.

Puerta, A. and D. Maulsby (1997). Management of Interface Design Knowledge with MODI-D. In *Proceedings of IUI'97*, January 1997, Orlando, FL, pp. 249-252.

Rine, D. C. and N. Nada (2000). An Empirical Study of a Software Reuse Reference Model. *Information and Software Technology*, pp.47-65.

Salinagros, N. (2000). The Structure of Pattern Languages. *Architectural Research Quarterly*, 4, pp.149-161.

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.*

Schlungbaum, E. and T. Elwert (1996). Automatic User Interface Generation from Declarative Models. In *Proceedings of CADUI 96,* Namur, Belgium, Namur University Press, pp. 3-18.

Schmidt, D., R. Johnson and M. Fayad (1996). Special Issue on Patterns and Pattern Languages. *Communications of ACM*, 39(10).

Seffah, A. and P. Forbrig (2002). Multiple User Interfaces: Towards a Task-Driven and Patterns-Oriented Design Model. In *Proceedings of DSV-IS 2000,* Rostock, Germany, Springer, pp. 118-132.

Seffah, A. and H. Javahery (2003). *Multiple User Interfaces, Cross-Platform Applications and Context-Aware Interfaces*. London, Wiley.

Shalloway, A. and J. Trott (2001). *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley.

Sinnig, D., A. Gaffar, D. Reichart, P. Forbrig and A. Seffah (2004). Patterns in Model - Based Engineering. In *Proceedings of CADUI 2004*, 2004, Funchal, Portugal, pp. 197 - 210.

Sinnig, D., H. Javahery, P. Forbrig and A. Seffah (2003). The Complicity of Model-Based Approaches and Patterns for UI Engineering. In *Proceedings of BIR 2003*, Sept. 2003, Berlin, SHAKER, pp. 120 - 131.

Souchon, N. and J. Vanderdonckt (2003). A Review of XML-compliant User Interface Description Languages. In *Proceedings of DSV-IS 2003,* Funchal, Portugal, pp. 377-391.

TADEUS (1998). *Task-based Development of User interface software* [Internet]. Available from <http://www.icg.informatik.uni-rostock.de/~schlung/TADEUS/> [Accessed February, 2004].

TERESA (2004). *Transformation Environment for Interactive Systems Representations* [Internet]. Available from <http://giove.cnuce.cnr.it/teresa.html> [Accessed February, 2004].

Thevenin, D. and J. Coutaz (1999). Plasticity of User Interfaces: Framework and Research Agenda. In *Proceedings of Interact'99,* Edinburgh, Scotland, pp. 110-117.

Tidwell, J. (2004). *UI Patterns and Techniques* [Internet]. Available from <http://time-tripper.com/uipatterns/index.php> [Accessed January, 2004].

Trætteberg, H. (2002). Model-based User Interface Design. Ph.D. Thesis in the Department of Computer and Information Sciences, Norwegian University of Science and Technology, Trondheim.

Trætteberg, H. (2004). Integrating Dialog Modelling and Application Development. In *Making Model-based UI Design Practical: Usable and Open Methods and Tools: A Workshop at IUI 2004,* January, Madeira, Portugal.

UIML (2003). *The User Interface Markup Language* [Internet]. Available from <http://www.uiml.org> [Accessed November, 2003].

Vanderdonckt, J., E. Furtado, J. Furtado and Q. Limbourg (2003a). Multi-Model and Multi-Level Development of User Interfaces. *Multiple User Interfaces, Cross-Platform Applications and Context-Aware Interfaces*, London, Wiley, pp. 193-216.

Vanderdonckt, J., Q. Limbourg and M. Florins (2003b). Deriving the Navigational Structure of a User Interface. In *Proceedings of INTERACT 2003*, Sept. 2003, Zuerich, IOS, pp. 455-462.

Vanderdonckt, J., Q. Limbourg and N. Souchon (2002). Task Modelling in Multiple Contexts of Use. In *Proceedings of DSV-IS 2002*, 2002, Rostock, Germany, pp. 77-95.

Vanderdonckt, J. and A. Puerta (1999). Introduction to Computer-Aided Design of User Interfaces. In *Proceedings of CADUI'99,* Louvain-la-Neuve, Kluwer Academic.

Velocity (2004). *Velocity User Guide* [Internet]. Available from <http://jakarta.apache.org/velocity/user-guide.html> [Accessed February, 2004].

Welie, M. (2004). *Patterns in Interaction Design,* [Internet]. Available from <http://www.welie.com> [Accessed February, 2004].

Welie, M. and G. Veer (2003). Pattern Languages in Interaction Design : Structure and Organization. In *Proceedings of INTERACT 2003*, September 2003, Zuerich, pp. 527-534.

XBL (2004). *The Extensible Binding Language 1.0* [Internet]. Available from <http://www.mozilla.org/projects/xbl/xbl.html> [Accessed February, 2004].

XIML (2003). *eXtensible Interface Markup Language* [Internet]. Available from <http://www.ximl.org> [Accessed November, 2003].

XUL (2004a). *The XML User Interface Language* [Internet]. Available from <http://www.xulplanet.com/> [Accessed February, 2004].

XUL (2004b). *XUL Tutorial* [Internet]. Available from <http://www.xulplanet.com/tutorials/xultu/> [Accessed February, 2004].

APPENDIX A: Discovered Patterns

Browse Pattern

Problem:

The user needs to inspect a set of information and to navigate a linear ordered list of objects such as images, or search results.

Context:

The pattern is applicable for interactive applications, which provide information to the user that is manually manageable. The user usually does not know what kind of and which information he/she needs. In this case, the browse pattern is more suitable than the search pattern.

Solution:

Give the user the possibility to iterate between the different objects that represent the information for browsing. Each object is represented by the print out of direct or indirect attributes.
The Print object pattern is used in order to print out object attributes. The user must have the possibility to select an object and optionally view the details of the object.

Rational:

Since the user does not know what information he is looking for or how to query for this information, each possible piece of information must be gradually inspected. Thus, the needed information must be discovered by browsing.

Implementation for the Task Model:

Interface:

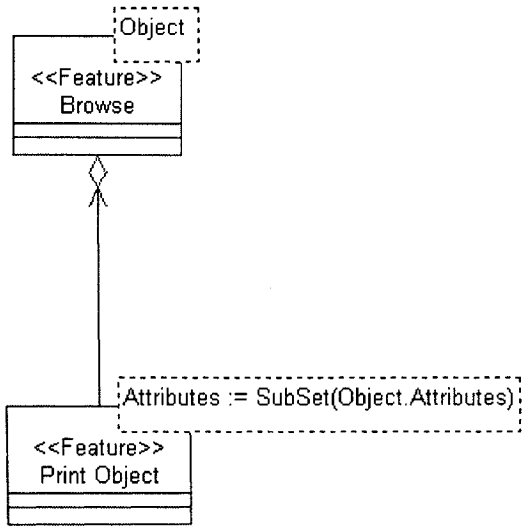

Figure A-1: Interface of *Browse* Pattern

Structure:

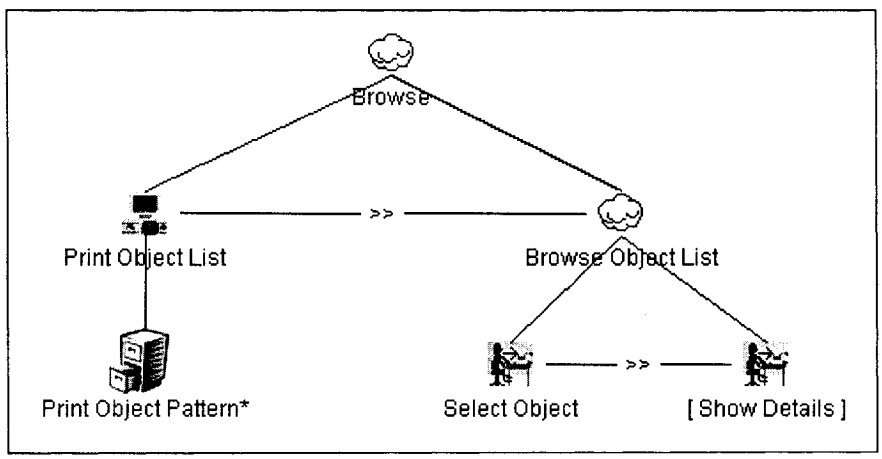

Figure A-2: Structure of *Browse* Pattern

Related Patterns:

Find, Search

Dialog Pattern

Problem:

The user must be alerted about a particular event. The user has to make a decision that is crucial for the further execution of the application or the user must confirm the execution of an irreversible action.

Context:

The pattern is applicable in each interactive application which requires the user's attention for messages that cannot be overlooked, or where a decision is needed where a path out of many is chosen for the further execution of the application. The patterns can also be used as a safety feature for the user in order to prevent the accidental execution of irreversible transactions.

Solution:

A modal dialog box / window can be used to get the user's attention. The program execution will only be continued if the user has confirmed the message or made a decision for the further execution of the program. Different instances of this dialog pattern are possible.

The Message dialog can be implemented in order to "force" the user to read an important message.

The Confirmation dialog is used to double-check of the user really wants to execute an action or to give the user the option to choose a particular path through the application.

Rational:

Pop-up modal dialog windows are an ideal way to attract the user's attention since program execution is stalled until a response to it.

Implementation for the Task Model:

Interface:

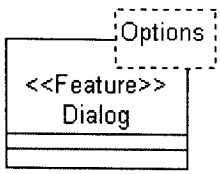

Figure A-3: Interface of *Dialog* Pattern

Structure:

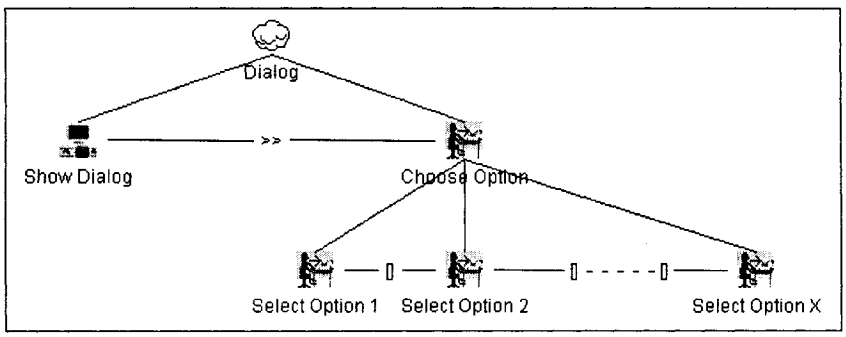

Figure A-4: Structure of *Dialog* Pattern

Find Pattern

Problem:

The user needs to find any kind of information that the application provides

Context:

Due to its generic nature, the pattern is applicable in any application that provides information to the user. The information or data can be of different types and structures.

Solution:

Finding an object can be performed by *Searching*, *Browsing* or by employing an intelligent *Agent*. Furthermore, combinations of the *find* methods are possible.

A *Search* pattern can be used if the data has a coherent structure and the user knows what he/she wants. In other words, the user knows how to query for the desired piece of information.

The *Browse* pattern is applicable when the user does not know how to "ask" (query) for what he/she wants. In this case, manually inspecting and browsing through the data is more suitable.

The *Agent* pattern is generally used when the data pool changes often and the user just wants to be notified if the desired information is found.

Rational:

Information or data exists in different forms and structures. In addition, the user's knowledge about the composition of the data can be very different. Thus depending on the nature of the data, a particular method or a combination of different methods should be employed.

Implementation for the Task Model:

Interface:

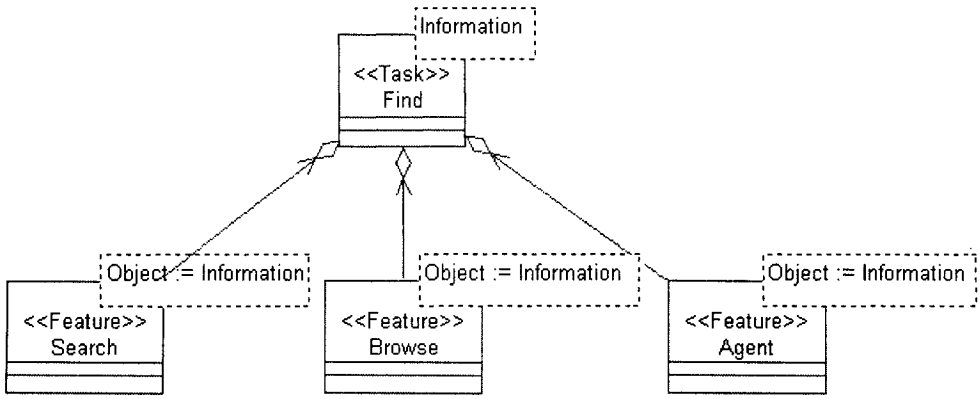

Figure A-5: Interface of *Find* Pattern

Structure:

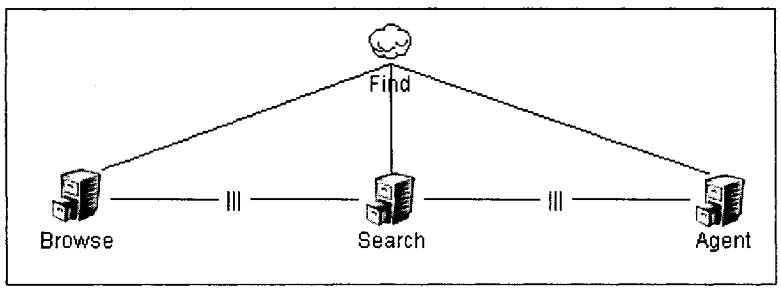

Figure A-6: Structure of *Find* Pattern

Related Patterns:

Browse, Search, Agent

Login Pattern

Problem:

The user needs to identify him/herself in order to access secure or protected data and / or to perform authorized operations.

Context:

The pattern is applicable if the application manages different types of users fulfilling different roles. The login process is also required if the application supports individual user modeling. An example is the individual customization of the appearance of the front end. In addition, the login feature is necessary if the user wants to create his/her own space, which makes it possible to have users enter that information once and use it again for future visits to the site.

Solution:

By displaying a login dialog, the user can identify him/herself. The login dialog should be modal. In addition, the user should login before accessing any personal data or secure features of the application. In order to identify him/herself, the user should enter a combination of different coordinates. After submission, the application provides feedback whether login was successful or not.

Rational:

By using the Login Pattern, the user can uniquely identify him/herself. Therefore the applications can provide customized views to the user and authorize the user to perform operations according to the user's role.

Implementation for the Task Model:

Interface:

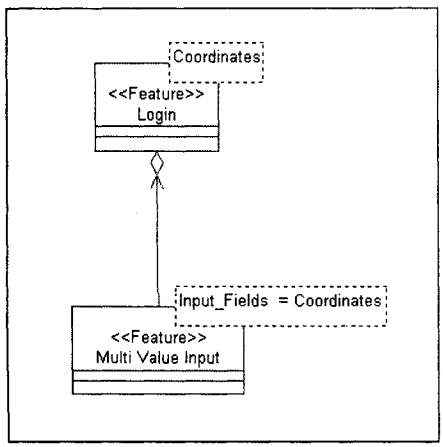

Figure A-7: Interface of *Login* Pattern

Structure:

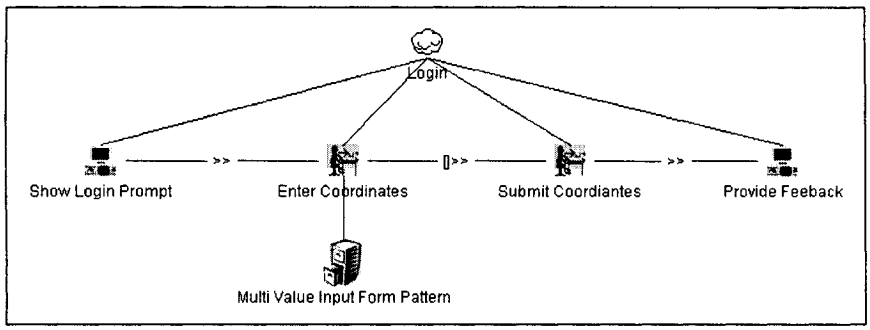

Figure A-8: Structure of *Login* Pattern

Related Patterns:

Multi – value input form

Multi Value Input Form Pattern

(Adapted from [Paternò 2000])

Problem:

The user needs to enter a number of related values. The values can be of different data types, such as "date", "string" or "real".

Context:

The pattern is applicable in an application requiring further written input from the user. The pattern is usually used within a pattern of higher granularity such as the registration pattern or the login pattern.

Solution:

An input form can be used to give the user the possibility to enter the values. The form can contain different input interaction elements such as text fields, lists or other components such as calendars. The multi-value input form pattern should be used in conjunction with the "unambiguous input" presentation pattern in order to avoid inputs in wrong formats. If possible, they should be displayed on one dialog view.

Rational:

The input of values is an interactive activity. In order to enable the user to enter or deliver values to the application, input interaction elements must be used. An input form is a convenient way of grouping related interaction elements.

Implementation for the Task Model:

Interface:

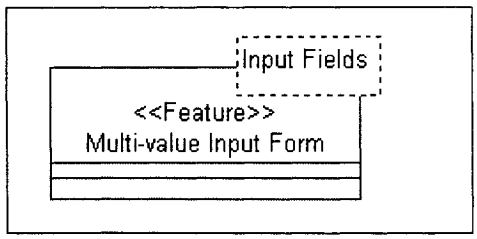

Figure A-9: Interface of *Multi-value Input Form* Pattern

Structure:

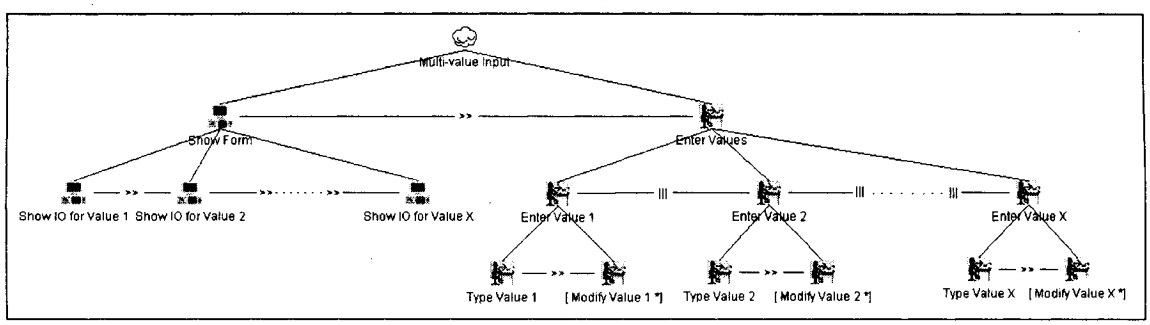

Figure A-10: Structure of *Multi-value Input Form* Pattern

Related Patterns:

Login, Unambiguous Format

Print Object Pattern

Problem:

The user needs to see details about a particular information object.

Context:

The pattern is applicable in each application that shares information with the user about its persistent / non-persistent data. In particular, the pattern is used for information objects which can be easily represented to the user.

Solution:

The object information is derived directly / or indirectly from the attribute values of the object. In addition, return values of methods can be used to gain object information.
All obtained information is then grouped and printed out in a cohesive manner.

Rational:

In order to gain information about the object, it must be inspected. An object is externally represented by its attributes and methods. Thus, object information must be derived from its attributes and methods.

Implementation for the Task Model:

Interface:

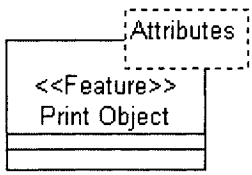

Figure A-11: Interface of *Print Object* Pattern

Structure:

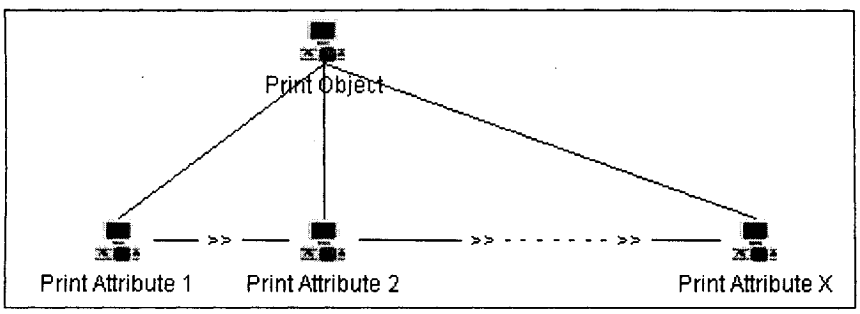

Figure A-12: Structure of *Print Object* Pattern

Search Pattern

Problem:

The user needs to extract a subset of data from a pool of information.

Context:

The pattern is applicable in interactive applications that manage considerable amounts of data, which can be accessed by the user. In most cases, the data can be extracted object instances derived from a common class. The application wants to provide the user with fast access to a subset of these instances. The pattern is not applicable if the user does not know what kind of information he/she needs. In such a case, the "Browse" pattern is more suitable.

Solution:

Give the user the possibility to enter search queries. On the basis of these queries, a subset of the searchable data is calculated and displayed to the user. The Multi-value Input Pattern is used for query input. After submission, the results of the search are presented to the user and can then, in turn, either be browsed or used as input for refining the search.

Rational:

If the user knows, what information he / she wants the search process is the most efficient way to find this data. If the search parameters are well chosen, the desired information can be found in only one step.

Implementation for the Task Model:

Interface:

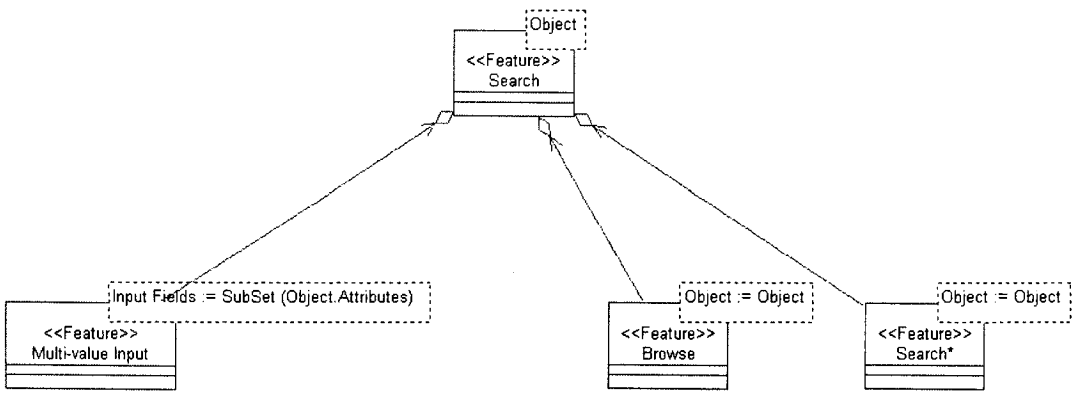

Figure A-13: Interface of *Search* Pattern

Structure:

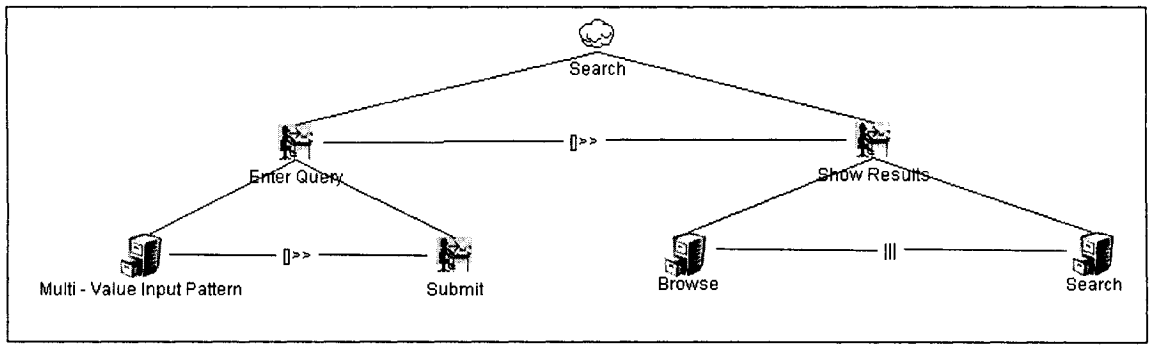

Figure A-14: Structure of *Search* Pattern

Related Patterns:

Find, Browse, Agent

Wizard Pattern

(Adapted from [Welie 2004])

Problem:

The user wants to achieve a single goal but several decisions and actions need to be made consecutively before the goal can be achieved.

Context:

The pattern is applicable whenever the user needs to go through a number of sequential tasks in order to achieve a complex goal.

Solution:

Group the tasks according to their purpose and temporal arrangement into dialog views. Arrange the dialog views sequentially in order to enable the user in reaching the goal consecutively. Enable the last task of each dialog view to act as a trigger event in order to open up the next dialog view.

Rational:

The Wizard is one of the most basic ways to navigate. By going through a number of dialog views the user is guided until he has reached the goal. Due to its linearity, process sequences are easier to understand and less error prone.

Implementation for the Dialog Model:

Interface:

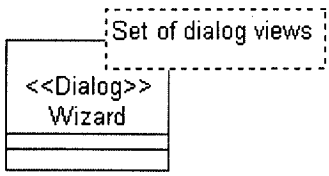

Figure A-15: Interface of *Wizard* Pattern

Structure:

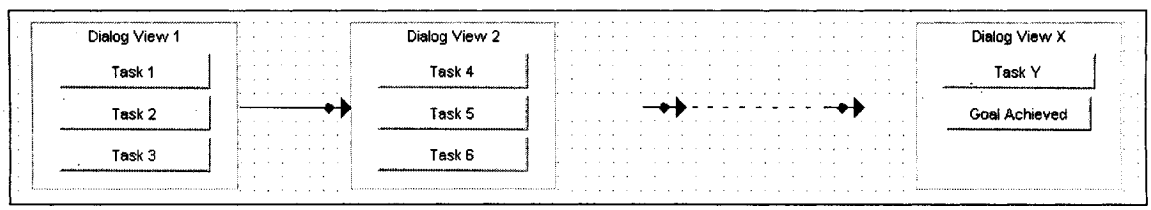

Figure A-16: Structure of *Wizard* Pattern

Recursive Activation Pattern

(Adapted from [Paternò 2000])

Problem:

The user wants to activate and manipulate several instances of a dialog view.

Context:

The pattern is applicable in many modern interfaces where several dialog views of the same type and functionality are accessible concurrently. An example can be a word processor that allows the editing of several documents concurrently during one session.

Solution:

Give the user the possibility of opening several dialog view instances. One source dialog acts as a starting point and a factory for the creation of the instances. In particular, through the execution of a specific task, the instantiation of a new dialog view is initiated. In order to create new instances, the source dialog view must remain active and accessible concurrently with the dialog instances.

Rational:

In order to interactively create instances of a dialog view, a second player is necessary, which acts as an initiator for the creation process. The source dialog view and the instantiated target dialog views must run concurrently.

Implementation for the Dialog Model:

Interface:

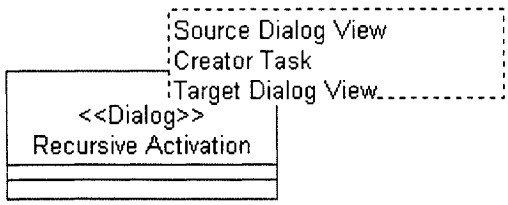

Figure A-17: Interface of *Recursive Activation* Pattern

Structure:

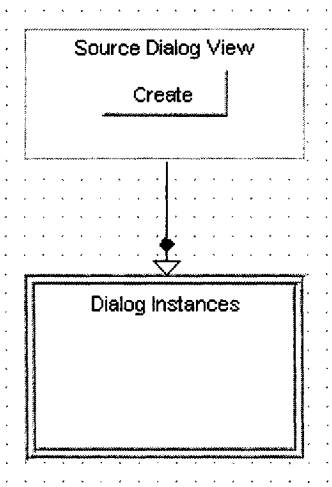

Figure A-18: Structure of *Recursive Activation* Pattern

Unambiguous Format Pattern

(Adapted from [Welie 2004])

Problem:

The user needs to enter data, but may be unfamiliar with the structure of the information and / or the syntax of it.

Context:

This pattern is applicable for all applications which require structural data input from the user on a textual basis. This pattern is especially useful if a novice user, unfamiliar with the demanded input, is interacting with the system.

Solution:

Provide the user with fields for each data element of the structure. The field does not allow incorrect data to be entered. In other words, the input interaction objects are chosen in a way the user cannot submit syntactically incorrect data.

Rational:

The main idea is to avoid entering incorrect data by not making it possible to enter wrong data. By showing the required format, the chances of errors are reduced. However, a resulting consequence is that the user ends up entering multiple values instead of one, which may slow down the performance.

Interface:

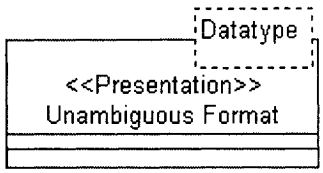

Figure A-19: Interface of *Unambiguous Format* Pattern

Example:

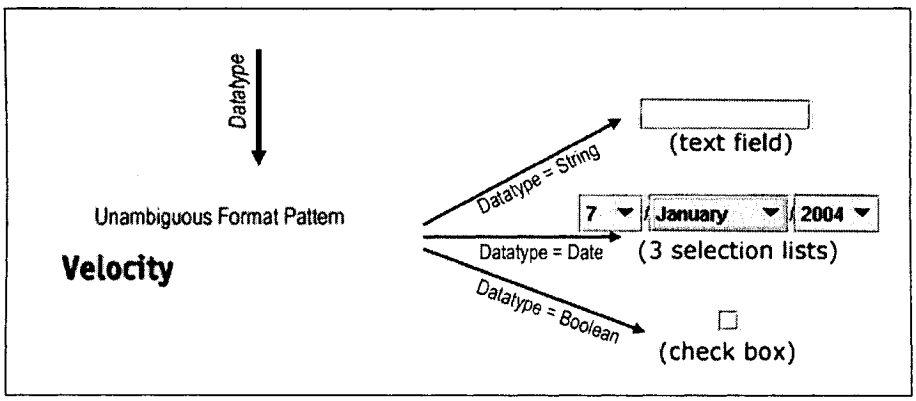

Figure A-20: Example of *Unambiguous Format* Pattern

Related Patterns:

Form

Form Pattern

Problem:

The user must provide structural textual information to the application. The data, to be provided, is logically related.

Context:

The pattern is applicable in any occasion where the user needs to provide structural text – based information to the application. For example when booking a flight, registering, tax calculations or simply logging in.

Solution:

Provide users with a form containing the necessary elements. Forms contain basically a set of input interaction elements and are a means of collecting information. Input interaction elements should be grouped and ordered logically according to the user activities described by the user's task model. Furthermore, it is important to use the right input element for a certain field. This depends on the number of options, single/multiple choice, and the sort of information that is required. Thus the form pattern can be combined with the Unambiguous format pattern.

Rational:

The input of values is an interactive activity. In order to enable the user to enter or deliver values to the application, input interaction elements must be used. The form is an ideal vehicle to structure and group logically-related input elements.

Interface:

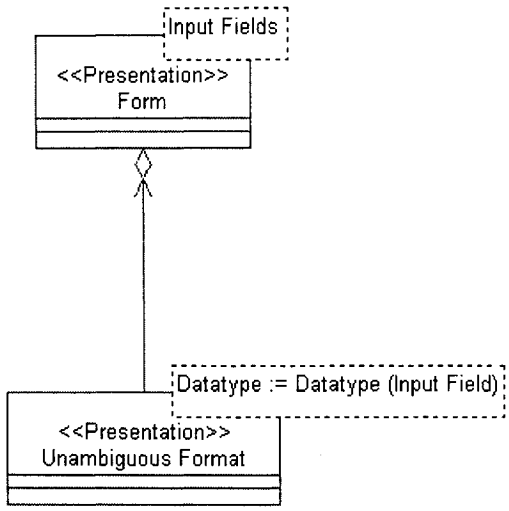

Figure A-21: Interface of *Form* Pattern

Related Patterns:

Unambiguous Format

House Style Pattern

(Adapted from Tidwell's Visual Framework Pattern [Tidwell 2004])

Problem:

Usually your application consists of several pages / windows. The user should have the impression that it all "hangs together" and looks like one thing.

Context:

The pattern is almost always applicable. In particular, if the application consists of several pages or windows.

Solution:

Maintain an overall look-and-feel for each page or dialog. Usually home pages or main windows are "special" and are laid out differently. However, they should still share certain characteristics with the rest of the application pages. The following attributes should be maintained consistently throughout the application:

- Color for background, text, accent color, etc.
- Fonts for titles, subtitles, ordinary text or minor text.
- Writing style and language use.

Rational:

When a UI uses consistent color, font, and layout, and when titles and navigational aids are in the same place every time, users know where they are and where to find things. They are not presented with a new layout each time they switch context from one page or window to another.

Labeling Pattern

Problem:

The user needs extra information about the current windows / site and the displayed interaction elements.

Context:

The pattern is applicable in almost each application, which requires user interaction. Labeling is used to explain the expected interaction of the user.

Solution:

Provide meaningful labels for each interaction element. Using the grid format, the label should be aligned left of the interaction element. Additionally, input hints can be provided in order to give explanations of the desired input.

Rational:

Filling in forms is error-prone and things must be made as clear as possible for users. Using the right labels contributes to the successful completion of the wanted input.

APPENDIX B: The Task-Pattern-Wizard

In what follows I will describe the usage of the various functionalities of the Task-Pattern-Wizard. As portrayed in Figure B-1 the tool features with six core functions:

- Open XIML Task Model
- Open Task Pattern
- Save Task Model
- Save Pattern Instance
- View Task Model
- Apply Pattern

All these functions are accessible via the Main Menu or via the toolbar by using the corresponding Speed Button.

The UI of the Task-Pattern-Wizard is divided into three parts (visualized in Figure B-1):

- The Navigation for accessing the functions
- The Content for using the functions
- The Status Bar for viewing the current process state

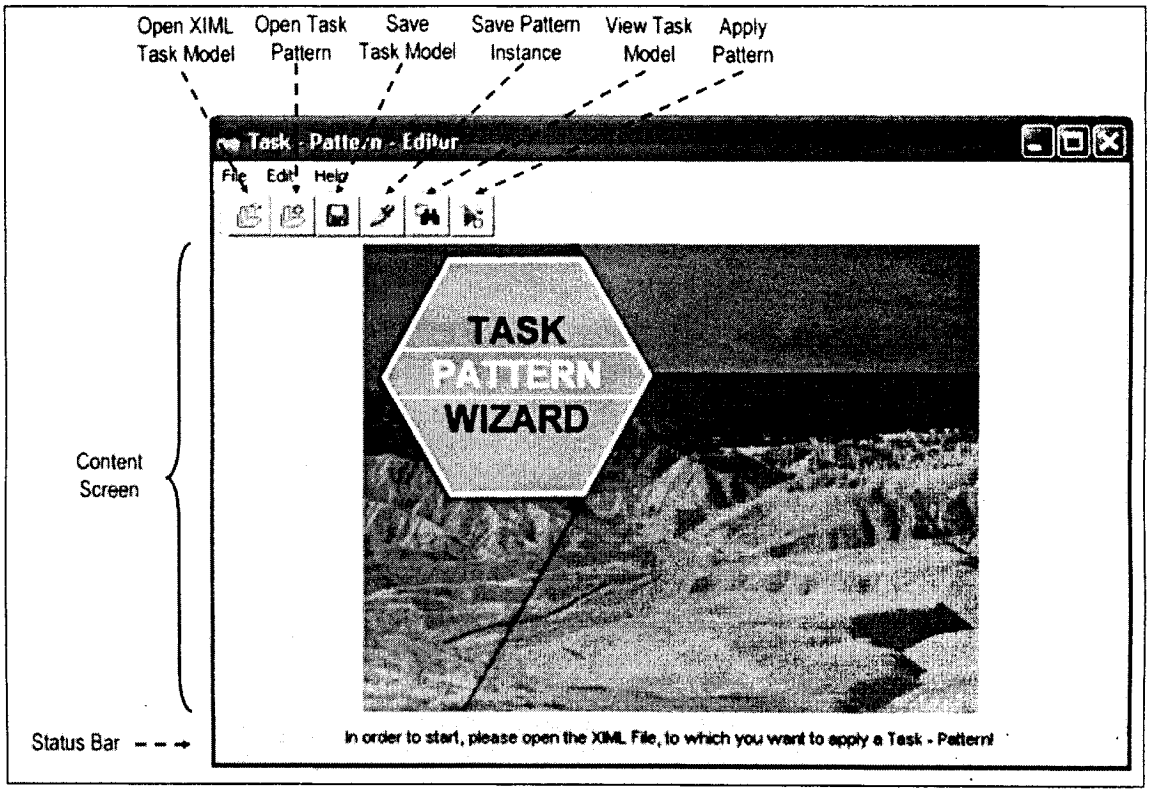

Figure B-1: Functionalities and Layout of the Task-Pattern-Wizard

In the remainder of this section I will describe in detail the usage of each of the functions of the Task-Pattern-Wizard.

Open XIML Task Model:

This function opens the target task model to which the task patterns will be applied. It must be specified according to the XIML specification. If the task model is successfully opened, a confirmation message will be displayed in the status bar portrayed in Figure B-2.

The XIML file has been opened successfully. You may now explore it or open a Task Pattern.

Figure B-2: Status Bar after Opening the XIML Task Model

Open Task Pattern:

After opening the XIML Task Model, a particular task pattern can be opened. All task patterns must be specified according to the TPML specification. TPML is an acronym for Task Pattern Mark-up Language and is described in great detail in APPENDIX C. After successfully opening the task pattern, its attributes will be displayed in the main content window. In addition, a confirmation message will be shown in the status bar. Figure B-3 contains a screenshot of the Task-Pattern-Wizard displaying the attributes of the *Login* Pattern.

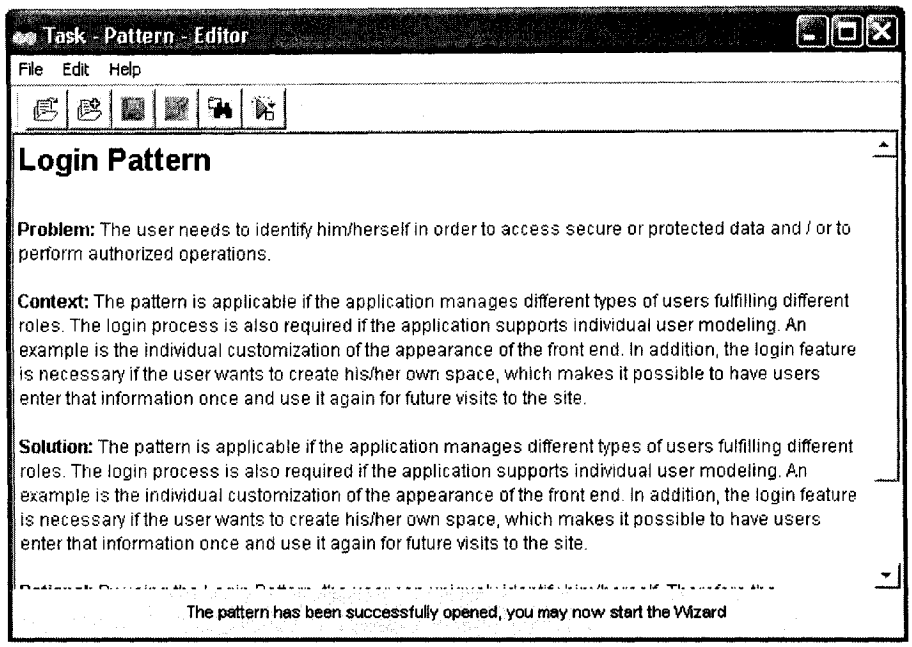

Figure B-3: *Login* Pattern Displayed by the Task-Pattern-Wizard

Apply Pattern:

After opening a target task model and a suitable task pattern, the pattern application function can be accessed. Basically this function is composed of two steps:

- Selecting a target node
- Adapting the pattern

Selecting a target node:

First, in order to apply a task pattern, a target node of the XIML task model must be chosen, to which the pattern will be applied. The user must then specify how the pattern should be integrated into the task model. Figure B-4 displays the corresponding window of the Task-Pattern-Wizard. The screen is divided in two parts. On the left hand side, the task tree is displayed. In order to select a target task node, one element of the task tree must be selected. On the right hand side, different options for integrating the task pattern into the task tree are shown. In particular it will be determined, in which temporal relations the new task fragment will be engaged. It is possible to specify if the execution of the new task fragment is optional or iterative. In addition it must be determined if the execution of the new task fragment enables other tasks or can be executed concurrently.

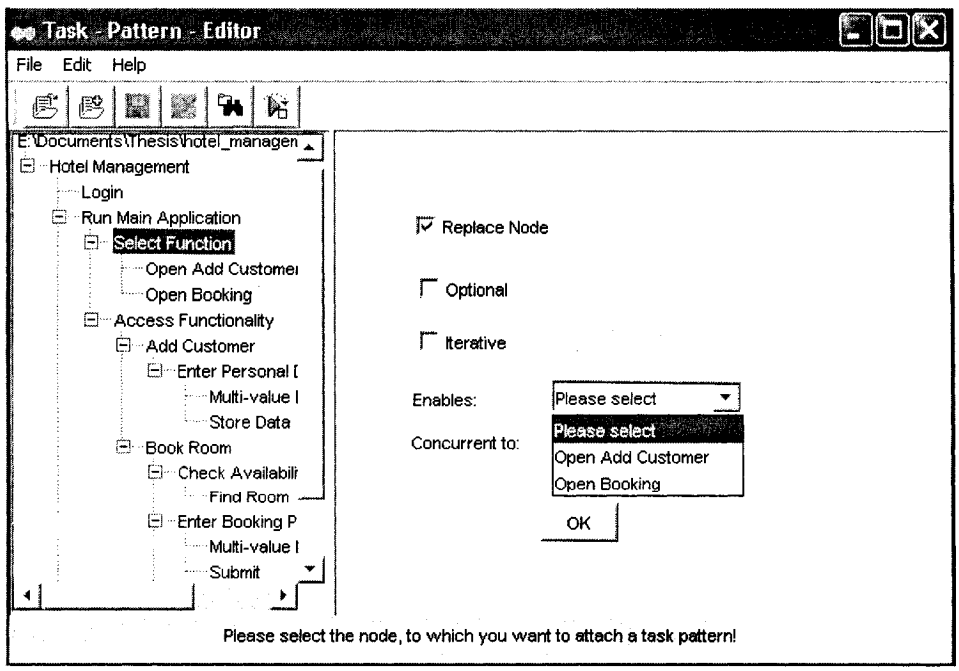

Figure B-4: Selecting a Target Node with the Task-Pattern-Wizard

Adapting the pattern:

After selecting where the pattern should be applied to the task model, the pattern must be adapted to the current context of use. Internally the Task-Pattern-Wizard will run through the pattern tree in order to resolve all variables. Each time an unresolved variable is found, a particular dialog box will be displayed, depending on the variable type. Two different types are possible:

- Process variable found
- Substitution variable found

Process variables are used to describe the structure of the task fragment, which will be created by the pattern. For example, entering values into a form is very repetitive. The same basic task (enter a value) appears over and over again. Each peer task can just be distinguished by its name and type of input. Thus, instead of describing each of these tasks on its own, process variables that signal the number of respective tasks can be used. The corresponding dialog box for a process variable is captured in Figure B-5, where the user is asked to enter the number of repetitive tasks (enumeration).

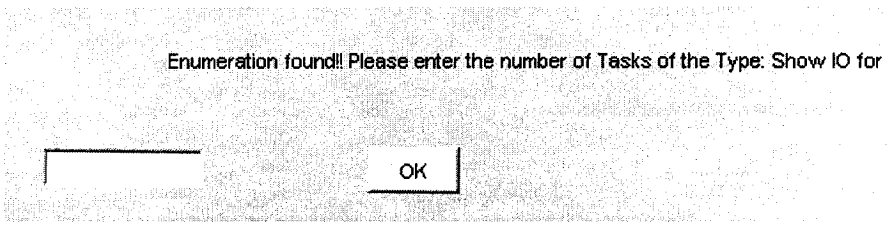

Figure B-5: Dialog Box for Resolving Process Variables

Substitution variables are simply used as placeholders for certain values (such as the task name). Following a top-down process, each occurrence of the substitution variable will be replaced (substituted) with this value. Figure B-6 displays the dialog box for resolving such a substitution variable, where the user is asked to enter the name of a particular task.

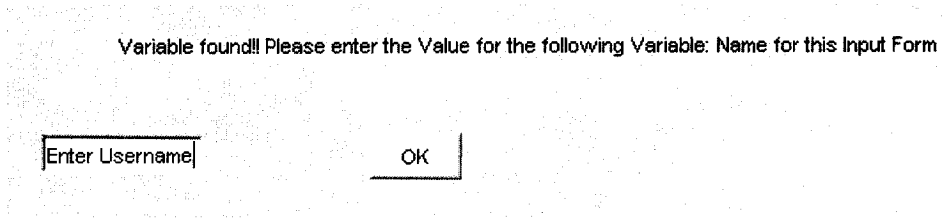

Figure B-6: Dialog Box for Resolving Substitution Variables

After resolving all variables, a pattern instance is created, which will be integrated into the task model. If this process was successful, the Task-Pattern-Wizard displays a confirmation message, as portrayed in Figure B-7.

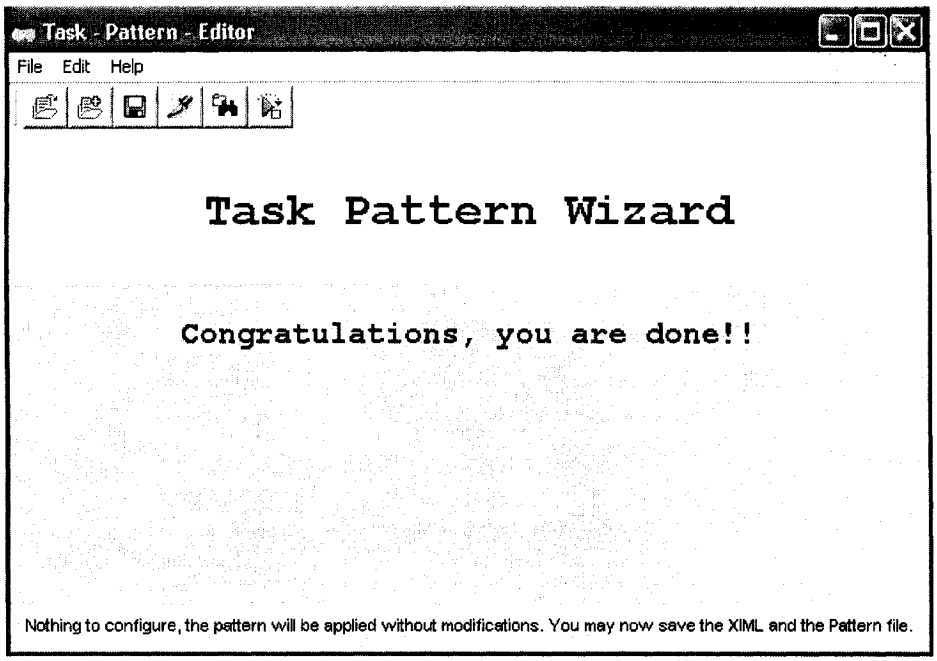

Figure B-7: Confirmation Screen After Integrating the Pattern Instance

Viewing and Saving the Task Model:

In order to verify the correct application of the pattern, the View Task Model function can be used. In a "read-only" fashion, the task model is displayed in a tree structure. Figure B-8 shows a screenshot of the "View Task Model" window.

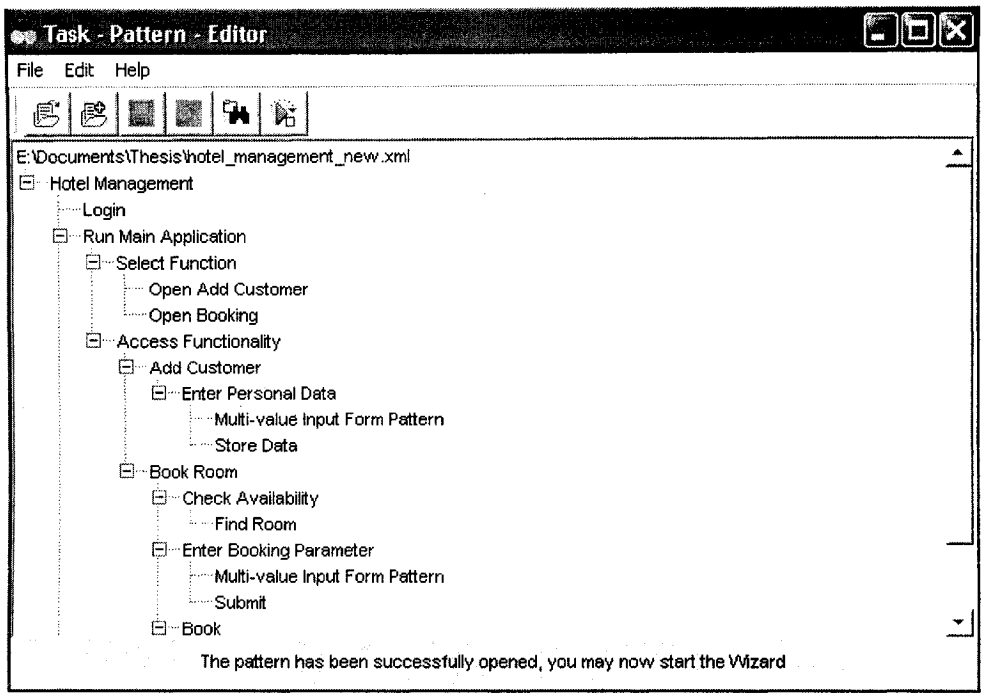

Figure B-8: Screenshot of the View Task Model Window

After verification the "Safe Task Model" function can be used in order to write the task model into an XIML file. If necessary this XIML file can be reloaded by the Task-Pattern-Wizard and other patterns may be applied as well.

Saving the Pattern Instance:

After a pattern has been adapted to the current context of use, the resulting pattern instance can be saved to disk. The pattern instance will be written to file in TPML format. A

pattern instance may be re-used in different contexts of use as a static task fragment for establishing the task model, where no further adaptation is necessary.

APPENDIX C: The Task Pattern Markup Language

In order to document and formalize task patterns, the Task Pattern Markup Language (TPML) has been developed. I proposed an XML Schema for the specification of task patterns. In this appendix, the structure of the TPML XML Schema will be described as well as the semantics of the various tags.

Figure C-1 portrays that TPML basically consists of the classic elements of a pattern (Name, Problem, Context, Solution and Rational) as well as the "Body" part. While the "Body" part contains the formalization of the pattern, the classic elements are more or less narrative descriptors, for selecting an appropriate pattern.

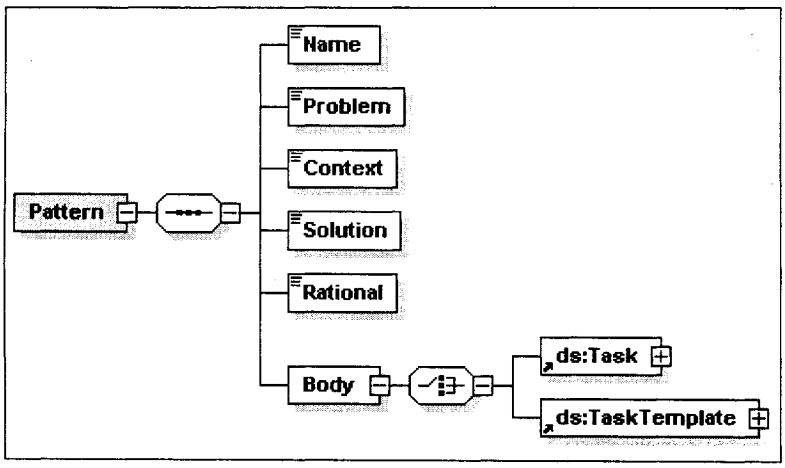

Figure C-1: Basic Structure of TPML

The following semantics are attached to the descriptor elements:

- **Name**: Name of the pattern
- **Problem**: Problem that is tackled by the pattern
- **Context**: Context of use in which the pattern is applicable
- **Solution**: Outline of the solution for the problem
- **Rational**: Reasons why the solution works

In Figure C-2, the Multi-value Input Form Pattern descriptor elements and their content are visualized using the tool XMLSpy.

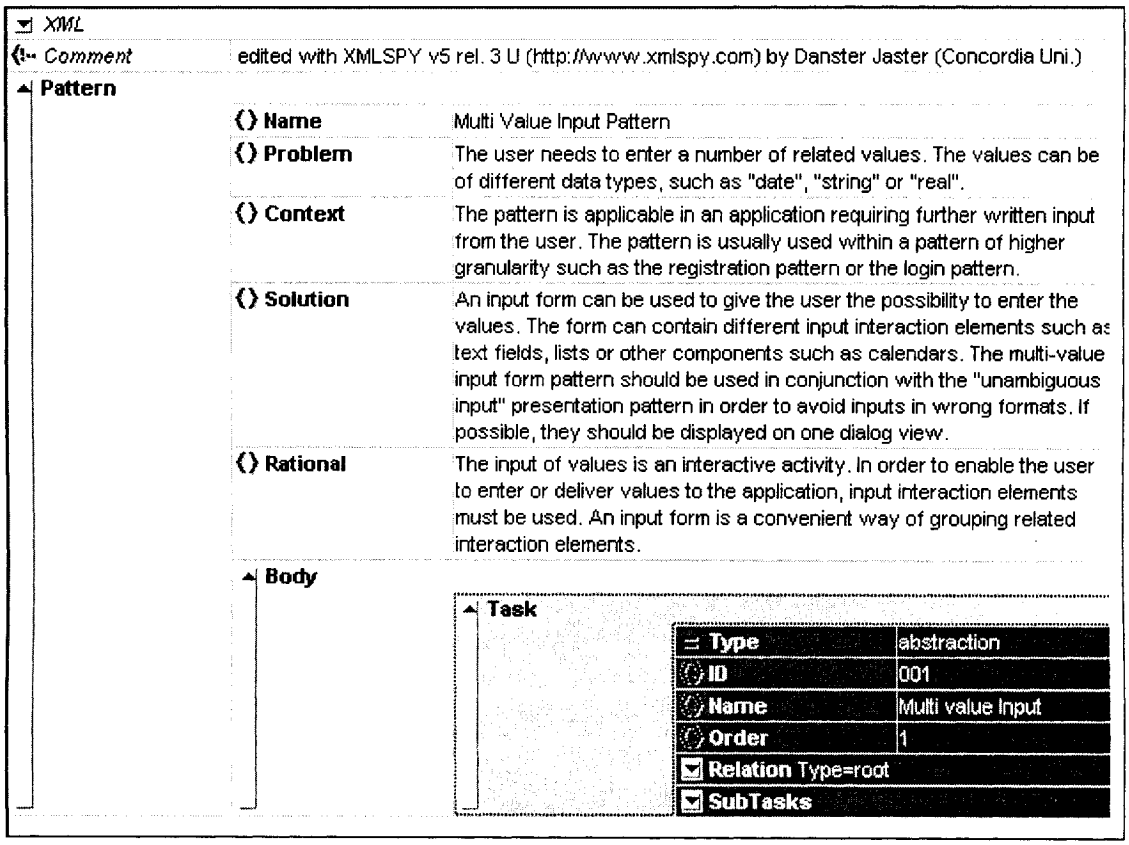

Figure C-2: *Multi-value Input Form* Pattern in XMLSpy

As mentioned before, the descriptor elements are mostly used for selection of an appropriate pattern. Figure C-3 shows a screenshot of the Task-Pattern-Wizard after opening the *Multi-value Input Form Pattern*. Based on these attributes the user can make a decision whether the patterns are appropriate or not.

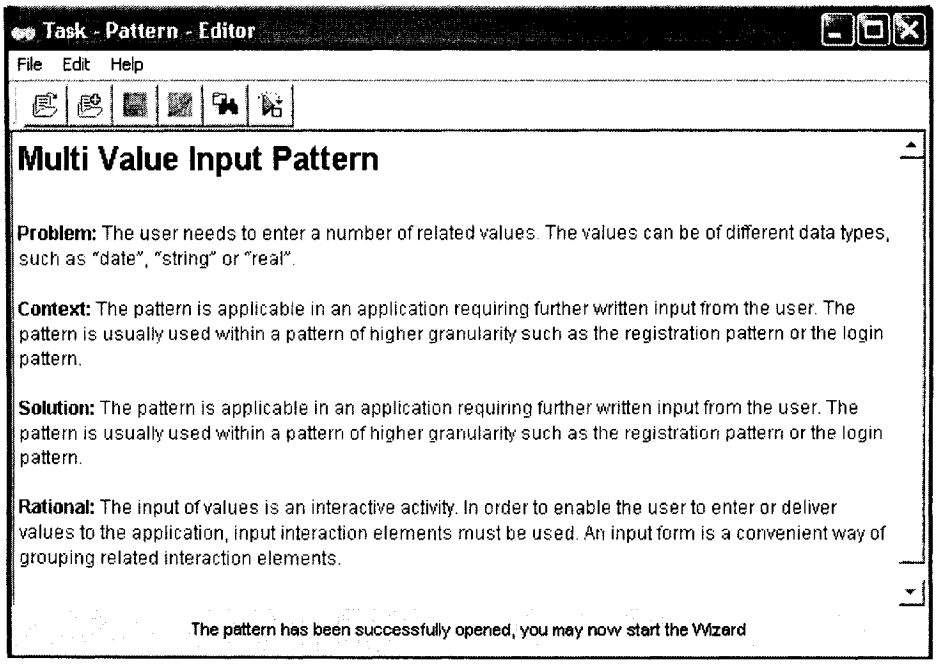

Figure C-3: *Multi-value Input Form* Pattern Displayed by the Task-Pattern-Wizard

As illustrated by Figure C-1, the body part of the pattern consists of two main elements containing its formalization:

- Task
- TaskTemplate

In what follows, I will describe in detail the structure and semantics for both of the complex XML Elements.

The Task Element:

The task element is a complex element used for static task descriptions, which contain no variable parts. Thus, they can be adopted and integrated as they are into an existing task model, without further adaptation. In particular, as visualized by Figure C-4, a task element is composed of:

- ID: Unique Identifier for the task
- Name: Name of the task
- Order: Display order for visualizing the task model
- Relation: Temporal relation to other tasks
- Subtasks: Composition of Task and TaskTemplate elements

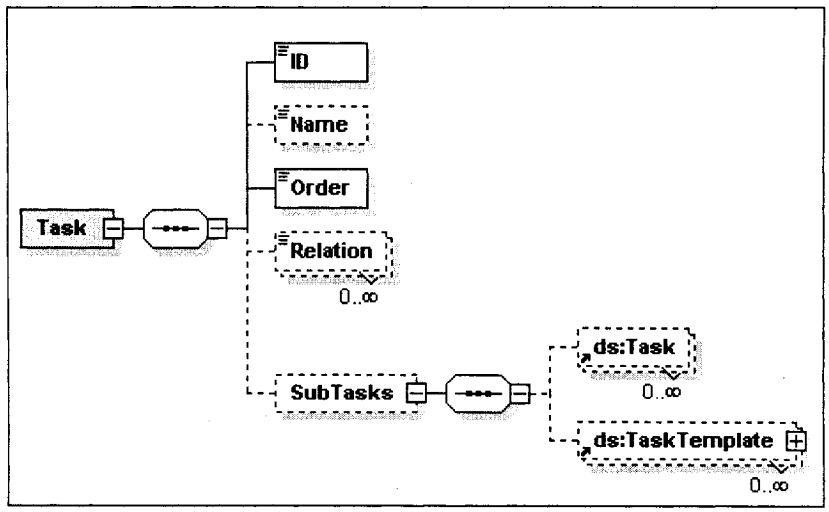

Figure C-4: Structure of the Task Element

It is to be noted that the Task element is hierarchically composed of SubTasks, which in turn, consist of Task and TaskTemplate elements.

The TaskTemplate Element:

Similar to the Task element, the *TaskTemplate* element is hierarchically structured. However, in contrast to the Task, TaskTemplates also contain variable definitions and variables and must therefore be adapted first, before integration into the task model. Figure C-5 shows the composition structure of the TaskTemplate element.

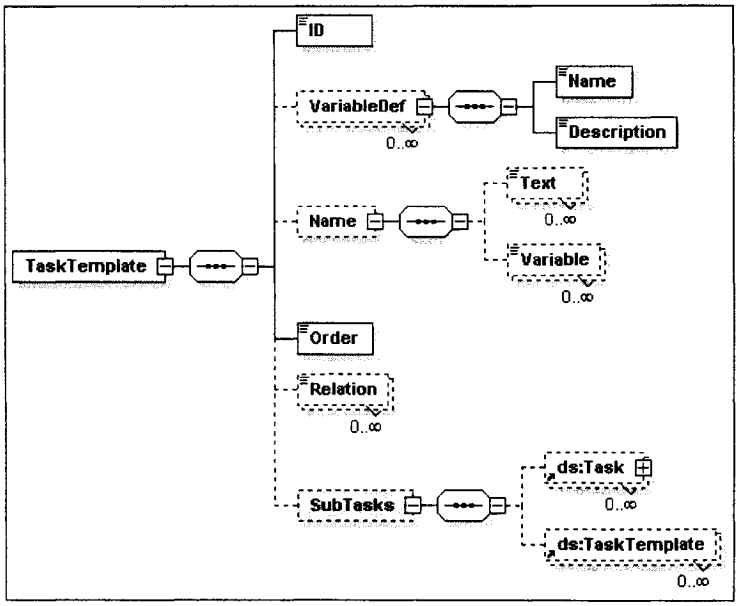

Figure C-5: Structure of the TaskTemplate Element

In addition to the Task element, the TaskTemplate element contains two extra elements:

- VariableDef: Declaration of new variables
- Variable: Use of Variables

Figure C-6 shows the usage of both tags, employed for the definition of the Multi-value Input Form pattern. The substitution variable "object" is used in order to act as a placeholder for the name of the task. During the process of pattern adaptation, the Task-Pattern-Wizard will resolve this variable in order to complete the name of the task.

```
<TaskTemplate Type="application" Category="enumaration" Relation="concurrent">
    <ID>201</ID>
    <VariableDef Type="String">
        <Name>object</Name>
        <Description>Name for this Input Form</Description>
    </VariableDef>
    <Name>
        <Text>Show IO for</Text>
        <Variable>object</Variable>
    </Name>
    <Order>1</Order>
    <Relation Type="root"/>
</TaskTemplate>
```

Figure C-6: Segment of Code from the *Multi-value Input Form* Pattern