Vrije Universiteit Brussel

FACULTY OF SCIENCE
Department of Computer Science

# Transformation of Task Models into Navigation Models in the context of WSDM

Graduation thesis submitted to obtain a License Degree in Applied Computer Science

**Jan Haegeman**

Academic Year 2006-2007

Promotor: Prof. Dr. Olga De Troyer
Advisor: Dr. Sven Casteleyn

Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN
Departement Computerwetenschappen

# Transformatie van Taakmodellen in Navigatiemodellen in de context van WSDM

Thesis ingediend met als doel de graad te behalen van Licentiaat in de Toegepaste Informatica

**Jan Haegeman**

Academiejaar 2006-2007

Promotor: Prof. Dr. Olga De Troyer
Begeleider: Dr. Sven Casteleyn

# Abstract

Since the birth of the World Wide Web in 1990, the need for automated and formulated web design methods has continuously grown. We have arrived at a turning point where we no longer want to code our web applications ourselves, but want them generated. Nowadays, automation plays an increasingly important role in web design. Design methods consist of a certain methodology that describes the systematic steps of the whole design process. A popular way of representing designs is by using models, which describe the interrelated parts of a web application.

The context of this thesis is WSDM, an audience-driven web design method, which takes as its starting point the requirements of the target audience of the web application. WSDM uses models throughout its methodology to represent both requirements and conceptual structure. Task models are used to model the tasks that will allow satisfying the requirements. Navigational models are used to model the conceptual structure of the web application and are based on the task models. These navigational models and the implementation details given in other models allow us to automatically produce the implementation of the web application.

An important step in WSDM is the transformation of task models into navigational models. Although, navigational models can, to a certain extent, be derived automatically from the task models, currently, this still needs to be done manually, while there is no reason not to automate this process. In this dissertation, an algorithm and an implementation are described that fully automate this transformation. Next to this, we look into other web design methods and investigate in what way these methods could be automated and how they describe requirements and conceptual structure. We also propose some design patterns for common task models, like containers used in websites to store items, the typical posting system in community forums, the basic login/logout scheme of websites and a confirmation dialog. These patterns can be considered as a form of automation too, as they help us save time in the design process.

# Samenvatting

Sinds de geboorte van het "World Wide Web" in 1990, is de nood aan geautomatiseerde en geformuleerde web ontwerpmethoden voortdurend blijven groeien. We zijn gearriveerd op een punt van ommekeer waar we niet langer onze webtoepassingen zelf willen coderen, maar dat we ze willen genereren. Tegenwoordig speelt automatisering een belangrijke rol in webontwerp. De ontwerpmethodes bestaan uit een bepaalde methodologie die systematisch de stappen van het gehele ontwerpproces beschrijft. Een populaire manier om ontwerpen voor te stellen is door modellen te gebruiken, die de met elkaar verbonden delen van een webtoepassing beschrijven.

De context van deze thesis is WSDM, een doelpubliek gedreven web ontwerpmethode, die als uitgangspunt de vereisten van het doelpubliek van de webtoepassing neemt. Ook WSDM gebruikt modellen bij zijn methodologie. Taakmodellen worden gebruikt om de taken te modelleren die de vereisten van het doelpubliek ondersteunen. De navigatiemodellen worden gebruikt om de conceptuele structuur van de webtoepassing te modelleren en zijn gebaseerd op de taakmodellen. De navigatiemodellen, samen met implementatiedetails gegeven in andere modellen staan toe om de implementatie van de webtoepassing automatisch te produceren.

Een belangrijke stap in WSDM is de transformatie van taakmodellen in navigatiemodellen. Alhoewel het mogelijk is om de navigatiemodellen tot op zekere hoogte automatisch af te leiden van de taakmodellen wordt dit momenteel nog manueel gedaan, terwijl er geen reden toe is om dit proces niet te automatiseren. In deze verhandeling, worden een algoritme en een implementatie beschreven die deze transformatie automatiseren. Ook kijken we naar andere web ontwerpmethoden en onderzoeken we op welke manier deze methodes zouden kunnen worden geautomatiseerd en hoe zij vereisten en conceptuele structuur voorstellen. Tenslotte, stellen wij ook nog enkele ontwerppatronen voor taakmodellen voor, zoals containers voor items, het typische "posting" systeem van forums, de basis "login/logout" procedure van websites en een confirmatie dialoog. Deze patronen kunnen ook als een vorm van automatisering aanzien worden, omdat ze ons tijd besparen in het ontwerpproces.

# Acknowledgements

I would like to thank my promoter, Prof. Dr. Olga De Troyer, and my advisor, Dr. Sven Casteleyn, for their invaluable help, support, advice, and encouragement. They motivated me throughout the period of the dissertation and helped me improve the quality of this document.

I would also like to thank my parents for giving me the opportunity to study at the Vrije Universiteit Brussel.

Furthermore, I would like to thank my friends and co-students for giving me the proper recreation when I needed it most and when I didn't need it at all.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Introduction

The Internet as we know is growing at an exponentially increasing rate year after year. It has become the most popular source for information in less than fifteen years, which can be considered as a negligible fraction of human history. It is a way of life that is relatively new and unexplored, although we learn more and more every day.

Nowadays, the Internet has become a victim of its own popularity. Vast amounts of information are scattered across the World Wide Web, which makes it difficult to find that piece of information you really need. No more than now has there been a growing need for professionally designed websites that know how to make their information accessible to their users. A well-designed website that offers information that is available to a wide range of users will not only have satisfied users, but will also establish an extensive and global user base.

Web design methods are a way to come to well-designed websites. They describe a fixed set of steps to successfully create a structural design for a web system. The goal of the design methods is to create the design of the web system without thinking about any implementation details. The implementation is completely separated from the design phases. Ultimately, one could automatically generate an entire implementation from a design, which is currently a hot topic in web design research.

To be clear on this, these web design methods are not at all focusing on the esthetic graphical looks of the web system. They describe what information should be available, how it can be reached, and try to make the information as transparent as possible to the user base. In the end, a design clearly outlines the basic navigational structure and the structural links between different parts of the web system. The look and feel of the web system is entirely up to a graphical designer and is fairly independent from the structural design created by a web design method.

## 1.2 Problem Statement

WSDM is such a web design method and is the main topic of this thesis. It consists of 5 phases that describe the purpose, the user characteristics, the conceptual structure, the implementation design and the implementation of the web system respectively. Throughout these 5 phases, it uses models to represent specific design structures, such as tasks that should meet the requirements of the users and models that depict the main course of action one can take when using the web system. These are called task models and navigational models respectively. One step of the design process is to examine these task models and turn them into navigational models. The main problem statement of this thesis can be formulated as follows[1]:

> Currently, WSDM uses Concurrent Task Trees to model the tasks the different users need to be able to perform in a web system. Concurrent Task Trees allow to decompose a particular task in (elementary) subtasks, and denote the temporal relations between the different (sub)tasks. On the basis of these task models, (part of) the navigation structure can be derived. Currently, this process is not automated but must be performed manually by the designer. This is a time-consuming and error-prone activity.
>
> The purpose of this thesis is to provide automatic support for the derivation of navigation structures from task models.

In WSDM, the navigation structure is described in the form of task navigational models. These task navigational models can be combined into one navigational track for an audience class. These tracks can then be combined into a navigational model. We need to automate the conversion of the task models into these task navigational models, to both speed up and simplify the design process.

## 1.3 Motivation

"Automation is the control of production processes by machines with human intervention reduced to a minimum", as Webster's online dictionary states. Automated web design methods are an attractive way of creating web systems. The more steps in the design process can be automated, the less work and time will be needed to design the web system and the more errors can be avoided. While some steps of design involve making decisions on particular design problems, other steps involve going from one design structure to another. The former is an example of something that can't be (fully) automated, while the latter can be automated by creating an algorithm for the transformation and implementing it.

The more automated a web design method is, the more time can be saved by using it and the more it will be used. The time saved will be significant when

[1]taken from `http://wise.vub.ac.be/`

creating designs for large web systems. Automation doesn't only save time, but it also guarantees fewer mistakes in the design process (when correct and stable algorithms are used).

This is what motivates us to create a program that supports the transformation of task models into task navigational models. Automating WSDM can only mean that the design process will become smoother and easier for the designers. While other steps of WSDM need to be done manually and therefore cannot be automated, our contribution is a step in the right direction.

## 1.4 Objectives

The main objective of this thesis is to create a program that automates one particular step of the WSDM process, i.e. transforming task models into task navigational models. Task models and task navigational models are both expressed by means of a graphical annotation. The main idea is to start from the graphical task models and convert them into graphical task navigational models without needing to parse images, which can be quite complex. Therefore, we need a way of presenting our task models in a textual form, so we can convert this format into a model. Finding a solution to this particular issue is a first objective.

The second objective, performing the actual transformation, is the main topic of this thesis. We will need to create an algorithm that performs the transformation and create navigational models out of the task models. Furthermore, we want a tool that implements this algorithm with a user interface to select the task models and adjust the generated navigational models. Because of the great variety in task models, we need to make sure our algorithm works in most (if not all) of the cases.

A third objective is to provide designers using WSDM with a number of predefined task model patterns. In general, design patterns represent solutions to common design problems. Although design patterns are prominently used in OO design, they can also be useful in other design activities, such as in web design. In many web sites, we see the same kind of functionality, e.g., a container for links or products (”shopping basket”), posting a comment in the guestbook, login activities... In WSDM, these activities need to be modeled by means of task models. As they occur that often, it may be more efficient to provide them in the form of task model design patterns. Designers can then benefit from this by using a task model design pattern when possible. This can also be considered as a form of automation as the designer will save time, as he doesn't need to create a task model from scratch.

## 1.5 Thesis Structure

This dissertation consists of 8 chapters, including this introduction. There are two main parts, one part describing the background and related work and one part describing the research done. We will briefly discuss the contents of these chapters.

In chapter 2 we examine WSDM, the Web Semantics Design Method, used in this thesis and supply the reader with the necessary background information on the different phases of this web design method. We describe the task models and navigational models of WSDM in greater detail in the second part of this chapter.

In chapter 3 we consider related work and take a look at other web design methods and investigate how they solve the problem of creating navigational models. We also check for automation in these web design methods, and how they are related to WSDM in general.

In chapter 4 we elaborate on task model design patterns and propose patterns for a confirmation dialog, the basic login/logout scheme, the "post message" system of most community forums and a container type that can store items. We study how these design patterns can help us saving time when creating task models that are related to these patterns.

In chapter 5 we discuss the algorithm that was created to perform the transformation of task models into task navigational models. We perform a thorough analysis of the algorithm, describing every step in full detail.

In chapter 6 we discuss the implementation of the algorithm. This includes a complete overview of the design of this implementation. We take a tour around the WSDM Visio Add-in, which is the transformation tool developed. We look at how it can be used to transform task models.

In chapter 7 we conclude this dissertation with a brief summary of the objectives we accomplished, limitations of the tool, and possible future work that could be done.

# Part I

# Background and Related Work

# Chapter 2

# Web Semantics Design Method

## 2.1 Introduction

In this chapter, we will discuss WSDM, the Web Semantics Design Method [2, 6, 7, 8, 9, 10, 18], developed at the Web & Information System Engineering (WISE) research group of the department of Computer Science of the Vrije Universiteit Brussel. In the first section of this chapter, we will explain the philosophy behind this design method and briefly describe the significant steps in the design process.

Afterwards, we study the models that WSDM employs in its design of web applications. The most prevalent models of WSDM are certainly the task and navigational models. We describe each type of model in the second section of this chapter.

## 2.2 WSDM Overview

WSDM is an audience-driven web design method, which means that it focuses on the potential users of the web system. It models the requirements and needs of these users and builds its design around these requirements. On the Web, most of the users are unknown and can't be interviewed to determine the needs of these users. That's why a designer will have to study the system to be implemented and analyze what classes of users it might possibly have. The audience-driven approach is clearly different from a data-driven or implementation-oriented approach that focuses on how the data of the web site or web application should be presented, rather than how a specific user class should perceive and find information on this web site.

WSDM was introduced in 1998 by De Troyer and Leune [10] under the name "Web Site Design Method". At that time, it was a systematic way of designing kiosk web sites. In general, we can consider two different kinds of web sites: the kiosk type and the application type. A kiosk web site provides information and allows users to navigate through that information. An application web site is a kind of interactive information system where the user interface is formed by a set of web pages. As WSDM evolved into what it is now, it added support for designing application type web sites. It now allows designing traditional application type web sites as well as semantic application type web sites. This is when the method was renamed to "Web Semantics Design Method". As WSDM can now both design web sites and web applications, we will use the term "web system" throughout this chapter to indicate either.

An overview of the different phases of WSDM is shown in figure 2.1. In the first phase, a mission statement for the web system must be defined. The mission statement clearly describes the purpose, the main subject and the target users of the web system. The mission statement is formulated in words (natural language) and should be kept as clear and short as possible.

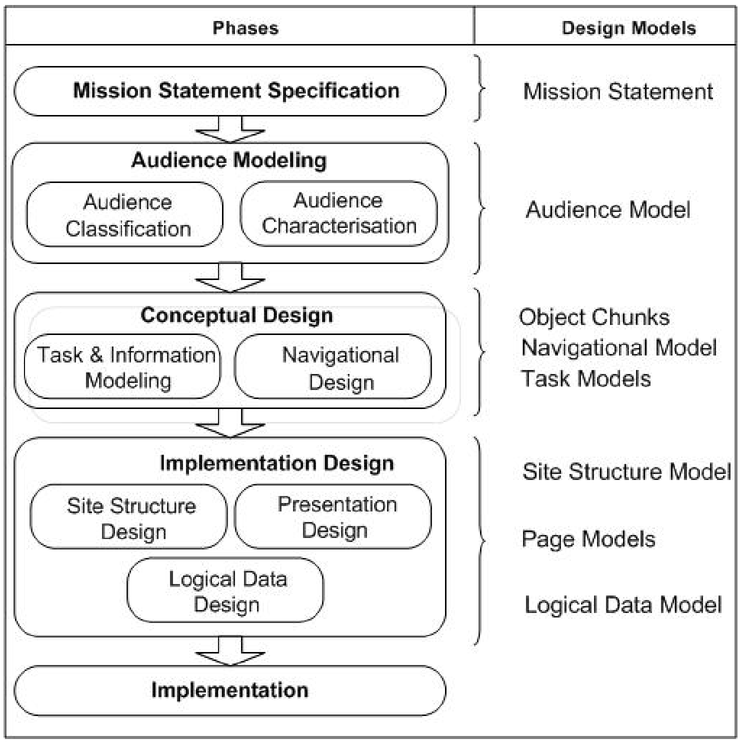


Figure 2.1: WSDM: An overview of the phases

With this mission statement in mind, the designer can identify target audience classes based on the requirements of the users of the web system. Users with need for the same information and functionality will end up being in the same audience class. For each of these audience classes the characteristics are given.

Once the designer has described the audience classes, he creates a conceptual model for the web system. This model abstracts from any implementation details.

From this conceptual structure, clear structural models of how one could navigate through the web system can be developed. For the same conceptual structure, different structural models can be defined. In this phase, we begin to create the "web pages" of the web system, clearly marking what page links to what.

In the last phase of WSDM, we create an implementation from the structural models. The implementation can be generated automatically from these models if the right tool is available.

### 2.2.1 Mission Statement Specification

In this first phase of WSDM, the designer creates a *mission statement* that should answer three questions:

- What is the purpose of the web system?
- What are the target users of the web system?
- What is the subject (the main topics) of the web system?

Without a purpose, it is impossible to define the functionality and information of the web system. If there isn't a purpose for creating the web system, then why create the web system in the first place?

The target users are the users that will actually use the web system and are interested in using it, either voluntary or work-related. They are identified as part of the main purpose of the web system. If the web system is a kiosk web site, then the goal is mainly to provide information to the target users. If the web system is an application type web site, we want the target users to use this system to accomplish certain goals.

The subject is the main theme and is connected with the purpose and target users of the web system. The subject needs to be carefully aligned with the goals. The subject can consist of a series of topics.

The mission statement is expressed in natural language, with a few concise sentences that give an answer to the above questions. Throughout WSDM, the mission statement is the basis for any design decisions that need to be made. The functionality and information of the web system will entirely depend on the purpose, the subject and the target users of the web system.

### 2.2.2 Audience Modeling

In this second phase, a detailed description of the target users of the web system is given. In the first sub-phase of the Audience Modeling, called Audience Classification, the users are divided into *audience classes*. Then, in the second sub-phase Audience Characterization, characteristics are assigned to these audience classes.

The Audience Classification phase classifies users into audience classes. An audience class is a group of users that are equal in needs and information and functional requirements. A typical example is that in a company the employees will have different motives than the CEO and his staff when visiting the company's web site. The employees will be one audience class, while the CEO and his staff will be another. When identifying the audience classes, an *audience class hierarchy* can be constructed. In our example, the employees can be considered as a superclass to midline managers, as midline managers are employees of the company as well.

In the Audience Characterization phase, the crucial characteristics should be specified for each audience class. The typical characteristics used are level of experience, frequency of use, language, educational level, age, income, sex, lifestyle...

The audience classes clearly define what information and functionality should be present in the web system, while the characteristics of each class define how the information and functionality needs to be represented to the user.

### 2.2.3 Conceptual Design

To goal of this phase is to turn the informal requirements of the audience classes into conceptual, high-level, formal descriptions that can be used later in the generation of the implementation of the web system. The Conceptual Design phase has two sub-phases: the Task & Information Modeling phase and the Navigational Design phase.

In the Task & Information Modeling phase, the requirements of the different audience classes should be analyzed. In this phase, the designer will create conceptual descriptions, called *object chunks*, that model the information and functionality needed to satisfy the requirements. We can distinguish two subphases in this sub-phase: the Task Modeling and the Information Modeling. The output of this phase is a set of *task models* and associated *object chunks*.

In the Task Modeling phase, task models are created to describe tasks that were defined to satisfy the requirements of an audience class. A task model is a way of modeling the details of a task; we use the task modeling technique CTT [15, 16, 17] for this. In CTT, tasks are decomposed into subtasks, which are in turn decomposed further on until elementary tasks are obtained. The subtasks of the task model are connected with each other by temporal operators. A temporal operator indicates the temporal relationship between two tasks, essentially the order in which these tasks will be performed.

When a task model is created for each of the requirements of the audience classes, in the Information Modeling phase an object chunk is created for each elementary task of each task model. If the requirement to which the task corresponds is purely informational, then the object chunk will describe this information. A standard conceptual modeling language can be used; WSDM uses OWL for this, as it is a specification language that allows to create domain ontologies. These domain ontologies will help when designing semantically annotated web systems. By describing the object chunks with OWL, it is easily to couple the chunks to an existing or new ontology. When coupled, it is possible to automatically generate the semantic annotations later in the implementation phase.

In the Navigational Design phase, a task navigational model is created for each task model. A task navigational model consists of components connected by links; the object chunks created for each elementary task are connected with these components. It specifies the workflow of the corresponding task. When the task navigational models are created for all tasks of one audience class, they can be combined into a navigational track for that audience class. This navigational track enlists the tasks that the members of an audience class can perform, much like the task navigational model enlists the parts of one task. Finally, a navigational model can be created by composing the navigational tracks into one structure. This means combining all the tasks of all audience classes. The navigational model describes the conceptual structure of the web system.

We will describe the task model and the navigational model in greater detail in sections 2.3.2 and 2.3.3 respectively.

### 2.2.4 Implementation Design

The next phase is the Implementation Design phase, which complements the conceptual design with details needed for the implementation of the web system. The Implementation Design consists of three sub-phases: the Site Structure Design phase, the Presentation Design phase and the Logical Data Design phase.

In the Site Structure Design phase, the navigational model is converted to a site structure model. This model describes how the components from the navigational model are divided amongst the pages of the web system. It is possible that multiple components are put on the same page, while other pages might have only one component. These pages are abstract; they only serve as a tool to provide the necessary information to automatically generate the implementation from them.

In the Presentation Design phase, the look and feel of the web system is specified. As mentioned before, this will depend on the characteristics of the audience classes. E.g., visually impaired people will have special needs when it comes to the layout of the site. The text fonts will have to be large enough and the images used will need to be big enough.

The final sub-phase is the Logical Data Design phase, in which the designer creates a *logical data schema* for the ontology, or conceptual schema. This is similar to the creation of a relational database from an ER schema. This process is normally automatically done with the help of a CASE-tool.

### 2.2.5 Implementation

This is the last phase of WSDM where the actual implementation of the web system is done. The specifics of this implementation is up to a designer to decide, as there weren't any specific implementation details in the previous phases that would make us choose one specific type of implementation. With the right tool, this implementation could be automatically generated from the structural models.

## 2.3 WSDM Conceptual Models

### 2.3.1 Introduction

In this section, we take a closer look at the models that are used in the Conceptual Design phase of WSDM, the task model and the navigational model. These types of models are created in the two sub-phases of the Conceptual Design phase: the Task & Information Modeling phase and the Navigational Design phase. These models define the conceptual structure of the web system being designed.

### 2.3.2 Task model

As mentioned earlier, a task model is a way of modeling the details of a task that satisfies a requirement of an audience class. We use the task modeling technique CTT (ConcurTaskTree)[15, 16, 17] to create a graphical representation of a task model. A task model consists of a tree, with the task to be analyzed as its root node. An example of a task model is shown in figure 2.2[1].

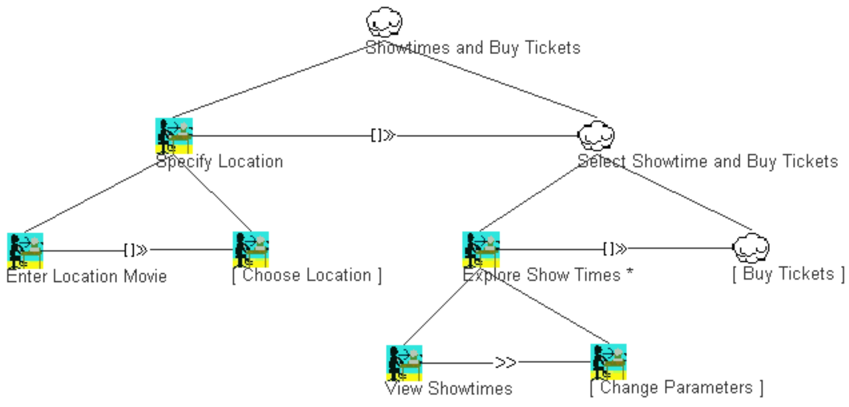


Figure 2.2: CTT: Showtimes and Buy Tickets

In the figure, we see the task *Showtimes and Buy Tickets* being modeled. This task is taken as the root node and is decomposed into subtasks until elementary

[1]this example was taken from the paper on WSDM by De Troyer et. al. [9]

tasks are reached, which are the leaves of the tree. The horizontal lines connecting the subtasks at the same level are the temporal operators, that indicate the temporal relationships between the subtasks.

CTT was developed in the context of Human-Computer Interaction to describe user interaction in the system being designed. In figure 2.3, we distinguish four types of tasks[2]:

- **User tasks:** Tasks that are entirely performed by the user, and does not require any interaction with the system. This usually involves tasks where the user will have to make a decision (a choice) before selecting from a number of options. These types of tasks are mainly for clarification, so the reader of the model knows where the user will have to make decisions.

- **Application tasks:** These are tasks entirely performed by the system, they do not require any user interaction. They indicate that the system is doing some processing work or calculations, before the next task can be performed.

- **Interaction tasks:** Tasks that are performed by users interacting with the system. Here, we have both the user and the system involved in the task. A typical example is when a user quits a process on the system: the user presses a button which makes the system end the process.

- **Abstraction tasks:** These are tasks that are complex and abstract and do not fall under any of the categories above. When an abstraction task appears as an elementary task in the tree, it indicates that the task will be decomposed in another task model. It serves as a reference to this task model.

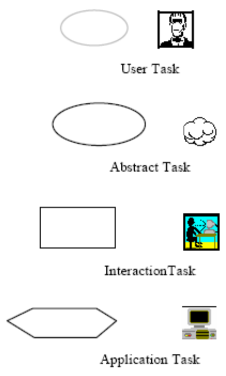


Figure 2.3: CTT: Categories of tasks

[2]taken from the paper on ConcurTaskTrees by Paterno [17]

The temporal operators connecting the sub-tasks indicate temporal relationships. CTT distinguishes 8 temporal relationships:

- **T1 [] T2 - Choice:** A choice temporal relationship signals a choice to be made between task T1 and T2. After making the choice, we continue with the next task. One of the tasks will not be performed.
- **T1 |=| T2 - Order Independency:** Task T1 and T2 can be done in any order and both tasks will have to performed in order to continue.
- **T1 ||| T2 - Concurrent:** Both tasks can be performed concurrently, at the same time.
- **T1 |[]| T2 - Concurrent with information exchange:** Both tasks can be performed concurrently, at the same time, but will need to synchronize by exchanging information. This is also called *synchronization* in earlier versions of CTT.
- **T1 >> T2 - Enabling:** Before we can do task T2, we will first have to do task T1. T1 enables to do T2.
- **T1 []>> T2 - Enabling with information exchange:** Same as Enabling but with exchange of information between the tasks.
- **T1 [> T2 - Disabling:** Also called *deactivation*. The task T1 can be disabled by performing task T2.
- **T1 |> - Suspend/Resume:** We can suspend the T1 task and move directly to task T2. While doing T2, we can also go back and resume task T1.
- **T* - Iteration:** The task T is iterative, it can be repeated as much as needed.
- **T(n) - Finite iteration:** The task T is iterative and can be repeated $n$ times.
- **[T] - Optional task:** Task T is optional and doesn't need to be performed, it is not mandatory.
- **T - Recursion:** The possibility to include in the task specification the task itself. Recursion appears when one of the sub-tasks of a task has the same name as the task itself.

When we use these temporal operators in the task models, there might possibly be some ambiguity in the expressions. When we have *T1 >> T2 [] T3*, it could either be *(T1 >> T2) [] T3* or *T1 >> (T2 [] T3)*. CTT solves this by introducing a priority order among the temporal operators: choice > concurrency > disabling > enabling. Another way of solving the ambiguity is by introducing a new task that has two of these three tasks as its sub-tasks. So we would get *T1 >> T4* where T4 contains *T2 [] T3*. This way, there is no ambiguity. This is clearer than a priority order solution, since the reader of the task model doesn't need to know the priority order to read the task model unambiguously.

WSDM adopted the CTT task modeling technique to create its task models. Even though the same representation is used, there are some differences in specifications to better satisfy the particular requirements of web system design:

- WSDM does not utilize user tasks in its task models. They are not important for making the conceptual structure of the web system.
- CTT enforces a rule that we do not want to enforce in WSDM. When the children tasks of a parent task are of different types (or categories), then the parent task has to be an *abstraction task*. In WSDM, we do not enforce this rule. For example, if a task is an application task in WSDM, then we say that the system is responsible for that task, and its sub-tasks can be of any type.
- All of the temporal operators of CTT are used in WSDM as well. WSDM adds one new temporal operator: **->T<- - Transaction**. The transaction operator signals that a task must be executed as a transaction. If the task is complex and long, and it is not completed entirely, the whole task will not be successful and a roll-back steps in and reverses every change made.
- Finally, the level of detail in the WSDM task models is less than in the original CTT method. This is because *object chunks* are used in the Information Modeling sub-phase to provide more details. They describe the information and functionality needed for an elementary task in the task model, which means we don't need a detailed analysis of every task in the tree. The object chunks will give the necessary details.

The task models in WSDM create a first level of description of the tasks a user can perform in the web system. Each task model describes a sequence of tasks (the process logic or workflow of a task) and the necessary information to derive implementation details later on is kept in the object chunks. A second level of description is given by the object chunks, which conceive a piece of information or functionality for an elementary task in the task model. The object chunks are not a part of the task model itself; they are identified in another part of the conceptual design phase. The task models are created in the Task Modeling sub-phase, while the object chunks are created in the Information Modeling phase.

### 2.3.3 Navigational model

Once a task model is created for all of the tasks of an audience class, the designer can create a *task navigational model* for each task model, which is a translation of the task model into a navigational structure. This model is composed of *nodes* (components) and *links*. The links connect the nodes with each other. Each node can be connected with an object chunk that describes the functionality and information of that node. The nodes can be considered as the basic navigation units of the task navigational model. WSDM distinguish four types of links:

- **Process logic links:** These express the work flow of the nodes. Nodes are connected by process logic links to indicate the order in which the information and functionality corresponding with these nodes needs to be displayed/performed.
- **Structural links:** These signify the conceptual structure of the different parts of the web system.
- **Semantic links:** These are links that indicate semantic relationships. These semantic relationships depend on the reference ontology of the web system.
- **Navigational aid links:** The navigational aid links add (non-mandatory) links to the models to enhance the overall navigation of the web system.

We create a task navigational model for our CTT example in figure 2.2. The resulting model is depicted in figure 2.4.

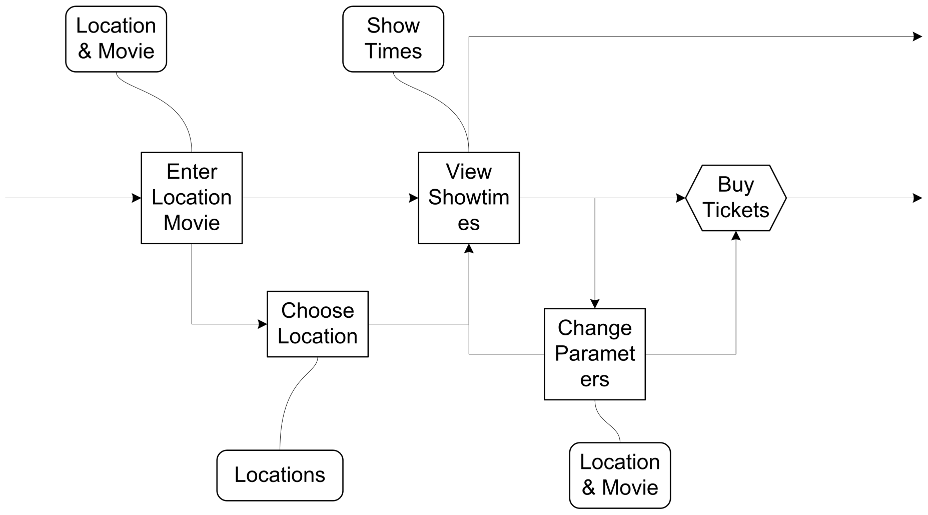


Figure 2.4: Navigational Model: The task navigational model

Note that each elementary task from our task model was converted to a node in this task navigational model. A node is depicted as a rectangle with the name of the task inside of it. An object chunk is depicted as a rounded rectangle with the name of the object chunk inside of it. The process logic links connect the node with each other, while the object chunk connectors connect the nodes with the object chunks.

Once the designer has created a task navigational model for each of the tasks of one audience class, he can create a *navigational track* or *audience track* for that audience class. This navigational track represents all of the tasks an audience class can perform in the web system. These models are of the same structure as the task navigational models. We create a navigational track (figure 2.5 for the audience class Movie Lover that contains the task Show Times and Buy Tickets we modeled in figure 2.4.

When we create a task model that describes how the audience class can choose between the several tasks modeled, we can use the same technique as before to convert this audience class task model into a navigational track, much like we converted a normal task model into a task navigational model. We don't need any object chunks in this stage of the modeling anymore, as there are no elementary tasks to describe.

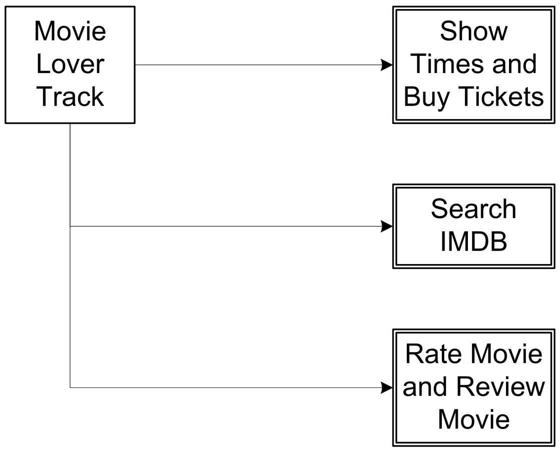


Figure 2.5: Navigational Model: The navigational track

The double-lined rectangles in figure 2.5 are a short-hand notation to indicate that these nodes are expanded further and were decomposed elsewhere in the design. The Movie Lover Track gets connected with the tasks the movie lover can perform through normal process logic links. Here we see a movie lover can choose which action to take; the task model corresponding to this navigational track would contain a choice temporal operator.

After creating a navigational track for each audience class of the web system, the navigational model for the web system can be composed by adding each navigational track into one model by means of the necessary structural links. Then the designer can add semantic links and navigational aid links to further enhance the conceptual structure of the web system.

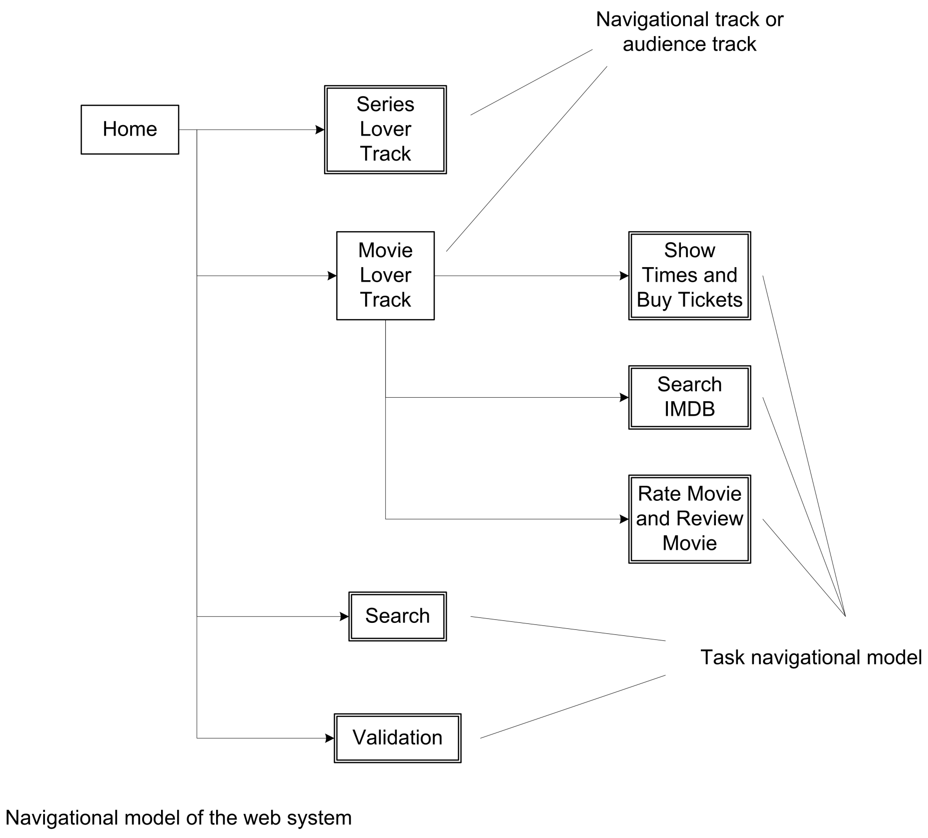

Navigational track or audience track
Home
Series Lover Track
Movie Lover Track
Show Times and Buy Tickets
Search IMDB
Rate Movie and Review Movie
Search
Validation
Task navigational model
Navigational model of the web system

Figure 2.6: Navigational Model: The navigational model

Here we have the navigational model that combines everything in one conceptual structure. We have added a Home node, that indicates the home or start page of the web system. From there a user can be validated, which simply means logging in and logging out. This validation task is described in a task navigational model, which isn't depicted in the figure. We also have added the tracks for Series Lover and Movies Lover, each described into more detail in the navigational tracks and their task navigational models.

Chapter 3

# Alternative Modeling Techniques

## 3.1 Introduction

WSDM is not the only web design method for the "World Wide Web". Although it was one of the first, there are several other methods that have arisen throughout time, one more popular than the other. Typically, these web design methods are quite different from each other, where one is more suitable for a specific situation than the other. In this chapter we discuss some of these web design methods. We describe the basic steps of each and make comparisons with WSDM where possible.

WebML is a model-driven web design method. Unlike WSDM which is audience-driven, WebML focuses more on the data schema and (site view) model schema of the web system, while considering the requirements of the users and the constraints of the environment. It is probably the most well known web design method at this time and is increasingly growing in popularity. WebML is discussed in section 3.2 of this chapter.

OOHDM/SHDM is another model-driven web design method. It clearly separates the conceptual models from the navigation models in its design. OOHDM was one of the first web design methods, while SHDM, a later version of OOHDM, extends it by adding semantic relationships between concepts. OOHDM/SHDM is discussed in section 3.3.

Hera is a methodology that supports the design and engineering of Web Information Systems (WIS). With model-driven techniques, it was specifically designed to cope with information retrieval from different sources on the Web (integration) and use of context (adaptation). It also supports engineering semantically annotated WIS. Hera is discussed in section 3.4.

## 3.2 WebML

### 3.2.1 Introduction

WebML [3, 4], the Web Modeling Language, was created to fulfill the need of companies that were missing a way of formalizing their web development techniques. It was nearly impossible to create and control web systems that were huge and complex of nature. In response to this need, the W3I3 Project (funded by the European Community under the Fourth Framework Program) focuses on "Intelligent Information Infrastructure" for data-intensive web systems. The project initiated a new web modeling language, called WebML, and a corresponding CASE tool, called WebRatio. WebML allows to create high-level, conceptual specifications of complex web systems. WebRatio is a tool that helps in creating the models WebML utilizes; it is centered mainly on the use of WebML. WebRatio enables to create the implementation of the web system automatically by converting the models of WebML into a HTML (Hypertext Markup Language), WML (Wireless Markup Language) or SMIL (Synchronized Multimedia Integration Language) specification.

While WSDM focuses mainly on developing web systems (web sites or web applications), WebML can successfully model the conceptual structure and data information for any data-centric user interface. These need not be Web related, but can be user interfaces that will run on PDAs, WAP phones, and even digital televisions. Because of this, we will talk about developing applications, rather than web systems throughout this section.

WebML is a *model-driven design method.* In its design phases, it uses models to represent the conceptual structures and logical units of the application. It also creates a data model or data schema to represent the data of the application. While it considers the requirements of the users, and groups the users into user classes, it does not base its conceptual structure on these user classes as WSDM does. We can distinguish the following steps in the iterative WebML design process:

- **Requirements specification:** We collect and formalize the essential information about the application domain and wanted functionality, based on the business requirements and environmental constraints. The output of this phase is a simple specification of what the application must do and how it fulfills the requirements of the target users and business requirements.
- **Data Design:** In this phase we organize the conceptual information objects identified in the requirements analysis into a comprehensive graphical data schema (or data model). Data models are a well-known type of model, we can create this data model in any specification we want: UML, ER, ORM, ... These data models will later serve to create the data sources of the application.
- **Hypertext Design:** We create site views based on the requirements specification. Hypertext design operates at the conceptual level, specifying

how units, defined over data objects from the data design, are composed within pages, and how these units and pages are interconnected.

- **Architecture Design:** In this phase we define the hardware, network and software components that make up the architecture on which the application will run. We take into account the constraints and requirements of the application and create a best possible match between the architectural details and the requirements.
- **Implementation:** We implement the software, transforming the conceptual models and data models into an application running on the selected architecture.
- **Testing and Evaluation:** We test and evaluate the software, checking if the software is consistent with the functional and non-functional requirements.
- **Maintenance and Evolution:** We create modifications in later stages of the software's life-cycle to reflect new user-based requirements.

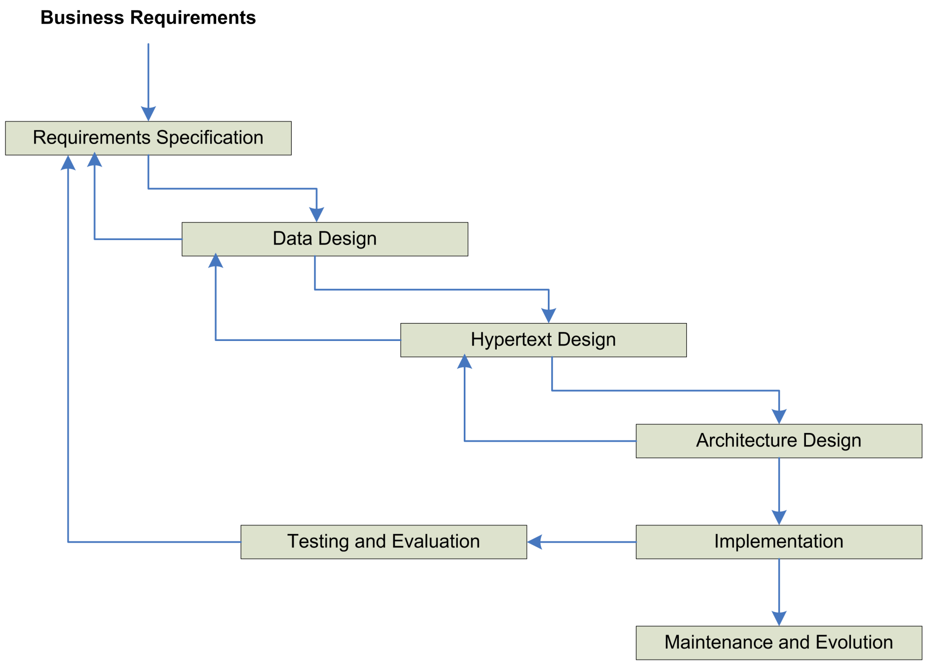


Figure 3.1: WebML: An overview of the phases

### 3.2.2 WebML Overview

WebML is a process, it derives outputs from the inputs it was given. The business requirements and environmental constraints are the inputs of the process.

A deployment architecture, together with the application itself, is the output of the WebML process. We will now discuss every step of the WebML in further detail.

#### 3.2.2.1 Requirements specification

During the requirements specification, we take a look at the business requirements and environmental constraints before building the design of the application. Business requirements are requirements that drive the application's development. They are mostly non-technical and express the long-term goals of the application. They identify the stakeholders' needs. Environmental constraints are the constraints that can affect the construction of the application. These are mainly limitations imposed by the current environment in which the company operates: legacy systems, available time and resources, overall technical skill...

This specification of requirements is fairly unstructured and informal. We use natural language to describe constraints and requirements, and use tables and lists to clarify where needed. We can distinguish two main parts in this phase: the Requirements Collection and the Requirements Analysis.

**3.2.2.1.1 Requirements Collection** During the Requirements Collection, we collect all information we have on the application domain and the requirements. We describe each step of this collection in further detail:

- **Identification of users:** Like in WSDM, we create user classes for each group of users that shows similar characteristics. During the development of the application, we take into account the several user classes. WebML distinguishes amongst several types of users: *internal* and *external* users, *business* and *non-business* users. As in WSDM, we can then create a hierarchy of user classes.
- **Functional requirements:** We address the essential functions that the application should deliver to its users. We can create *use cases* to describe these requirements.
- **Data requirements:** These describe what information should be available in the application. This step describes what knowledge we have and how we want to represent it.
- **Personalization requirements:** If there's need for personalization in the application, we create these type of requirements. Each requirement defines a *user profile*. This profile will help us create a customized application. While WSDM treats each user class differently, this could personalize the user experience on an individual level.
- **Nonfunctional requirements:** These include all the other requirements that are relevant: usability, performance, availability, scalability, security, maintainability.

**3.2.2.1.2 Requirements Analysis** During Requirements Analysis, we formalize the knowledge we have by using different techniques.

- **Group specification:** This is the formalized part of the identification of users. We create a hierarchy of user classes here and formally describe each user class. This is very similar to the Audience Modeling phase of WSDM.
- **Use case specification:** This is the formalized part of the functional requirements specification. We create a use case for each functional requirement. Each use case describes a scenario where a user performs a task with the application to be developed.
- **Data dictionary specification:** This is the formalized part of the data requirements specification. We create a list of the main information objects of the application. We formally describe each information object.
- **Site view specification:** This is the formalized part of the personalization requirements specification. We create a list of *site view maps*, based on the user profiles we created earlier. These sketch the organization of the hypertexts offered to the users.
- **Style guidelines specification:** We create a *style guide* for the application. A style guide describes the rules for presentation of pages: the font type, the colors, the page layout, margins...
- **Acceptance tests specification:** This is the formalized part of the non-functional requirements specification. We create a series of acceptance tests to see if their non-functional requirements are fulfilled.

Now that we've extensively examined the requirements and constraints for the applications, we can create a data schema for the application.

#### 3.2.2.2 Data Design

In the data design phase, we create an ER schema of the information in our application domain. We can extend this ER schema by creating some sub-schemas. Each of these sub-schemas detail an aspect of the requirements specification. We have a *core sub-schema* that details the internal properties of a core concept, an *interconnection sub-schema* that draws the relationships between the core entities, an *access sub-schema* that introduces categories and connects the core entities with these categories and finally a *personalization sub-schema* that maps the users and groups of users on the core entities.

#### 3.2.2.3 Hypertext Design

Hypertext design specifies the site views of the application. Site views have a user-defined name and contain a set of pages (or areas), they package the WebML hypertext of the application. An example of a graphical representation of a site view is shown in figure 3.2.

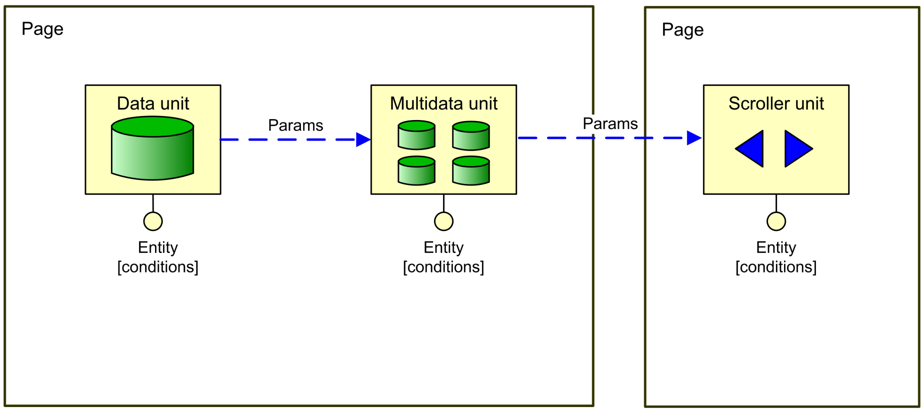


Figure 3.2: WebML: A site view

The hypertext design consists of two phases: the Coarse Design and the Detailed Design. In the site view coarse design, we create a first draft of each site view, by using the data schema and the site view maps created earlier in the WebML process. In the detailed design, we refine the coarse design by revising each draft and adding specific requirement details where needed.

During both of these phases, we will create *hypertext models*. Hypertext models are a way of representing the hypertext of the application. We specify which pages compose the hypertext and which *content units* make up a page. A content unit is logical unit that describes or contains an information object of the data schema. We distinguish four main types of content units:

- **Data units:** These units represent an information object (or *component*). More than one unit can be defined for the same object, to offer alternative points of view. The data units allow us to hide certain attributes of the component. An example is that a student (the entity) gets represented by his name, enrollment number and year (the attributes). Another data unit can be defined on this student that only shows his name and year, this is an alternative point of view created by defining multiple data units for one entity.
- **Multi-data units:** Multi-data units present multiple instances of a component together, by repeating a data unit representation of the component. For our example we create a page that shows multiple students: multiple chunks consisting of a name, a year and enrollment number are shown on the page.
- **Index units:** Index units are much like multi-data units, they allow the presentation of multiple components, based on a common index. This would mean a page that lists the names of each student in our example. We can click on these student names for more information.
- **Scroller units:** These allow browsing of multiple data units. We get to

see the first student in the database on one page. We can click on the Next button in this page to see the next student in line.

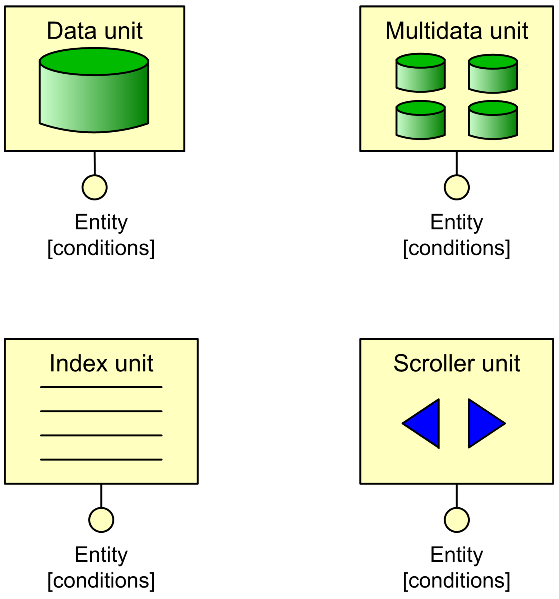


Figure 3.3: WebML: Four main types of content units

In general, we want some of the content units to be displayed together in one structure. To cope with this requirement, WebML defines a *page*. A page is an abstract representation of an interface block, a part of a screen (in HTML, this would be a frame), that can contain multiple content units or other pages. The pages and content units constitute a site view, as shown earlier in figure 3.2.

With pages and contents units we can describe the conceptual hypertext structure of the application. In later stages of modeling, we create a navigational model that connects the pages with each other. The pages can be linked with *contextual links*, meaning some information exchange will happen, and *non-contextual links* that connect pages in a totally free way.

#### 3.2.2.4 Architecture Design

This is the first phase of WebML that doesn't focus on the conceptual structure or requirements of the application, but concentrates on the choice of hardware, network components and software that will make up the application. We need to find a mix of these components that best meets the application requirements.

#### 3.2.2.5 Implementation

In the implementation phase, each entity in our Entity-Relationship diagram from the Data Design becomes a relation table. We choose the type of database and define primary keys on each table. The SQL syntax for creating these tables can be automatically generated from the ER diagram, if this diagram is well-formed.

In a second phase of implementation, we implement the pages, content units and links. Again, we can automatically generate the corresponding JSP or ASP from the hypertext models we created. The mapping is pretty basic and we have all the information needed to perform it. To manage the presentation of the pages, we can create XSL and CSS from the style guides we made earlier.

#### 3.2.2.6 Testing and Evaluation

We test and evaluate the application in three ways:

- **Functional testing:** We test if the application meets all functional requirements. The functional testing can be split into three classical activities: module testing, integration testing and system testing.
- **Usability testing:** We test if the application meets all non-functional requirements.
- **Performance testing:** The throughput and response time of the application is evaluated in various conditions.

#### 3.2.2.7 Maintenance and Evolution

We monitor the performance during the lifecycle of the application and make changes where needed. Changes in requirements are propagated to the design level, changes at the design level are propagated to the implementation level.

## 3.3 OOHDM/SHDM

### 3.3.1 Introduction

OOHDM [5, 20, 21, 22] the Object Oriented Hypermedia Design Method, was one of the first web site design methods, created in 1995, and is built in an object oriented manner. SHDM, Semantic Hypermedia Design Method, is its successor and allows us to successfully develop semantic web systems. Both of these methods use the same methodology. We'll describe SHDM in more detail, since that one contains the OOHDM methodology and enables us to compare it with WSDM more effectively.

SHDM is another *model-driven design method.* The main difference with WSDM is that it doesn't focus on audience classes and characteristics throughout its design. It puts down basic requirements and immediately starts modeling the conceptual structure of the web system. SHDM was specifically created to model web systems, either using JSP, HTML or other languages. It allows automatic generation of the implementation (the HTML and JSP pages) once the design phases have been completed.

SHDM consists of five main phases:

- **Requirements Gathering:** Much like in WebML and WSDM, we gather the requirements and we collect information on the users of the system. We take a look at the application domain and create some basic constraints for the web system where needed. While we identify what role each user plays, we won't divide the users into user classes, like we would in the other two methods.
- **Conceptual Design:** This phase shows some similarity with the Data Design phase of WebML. We create conceptual objects for each object in our application domain and describe the attributes. This conceptual design can be created with UML, since we're describing these concepts in an object-oriented manner. To create constraints, relationships and behaviour in these models, we can use RDF or OWL in SHDM.
- **Navigation Design:** We create relationships (links) between the concepts we described in the conceptual design. The output of this phase is a navigational model that will help us create an abstract interface in the following phase.
- **Abstract Interface Design:** In this phase, we create an abstract interface, that will allow us to create a concrete interface. An abstract interface describes how the objects from the conceptual design will appear to the user of the web system. It also describes how a user will interact with the web system.
- **Implementation:** The conceptual objects are transformed into actual implementation objects. With JSP (Java Server Pages), we can create these objects, as Java is an object-oriented programming language. Once the implementation objects have been created, we can describe the look

and feel of the web system, which is entirely up to a graphic designer to decide.

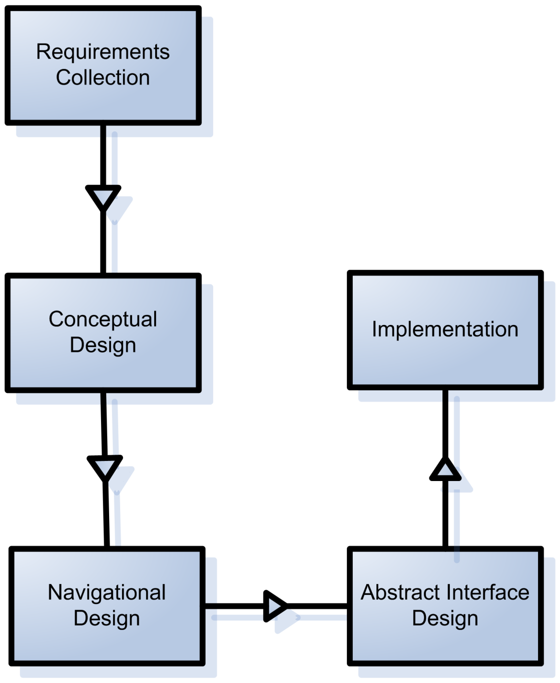


Figure 3.4: OOHDM/SHDM: An overview of the phases

## 3.3.2 OOHDM/SHDM Overview

SHDM, like any other web design method, is an iterative, top-down process. From conceptual, high-level models, that can be corrected or changed iteratively, we create low-level implementation specifications or automatically generate them. We will now describe the five phases of SHDM.

#### 3.3.2.1 Requirements Gathering

The authors on OOHDM/SHDM remain vague on this phase of the web design method. SHDM describes the target users, the role of these users and the basic requirements, restrictions or constraints for the web system. With this information, we can then create constraints in the ontology of the web system.

#### 3.3.2.2 Conceptual Design

During the Conceptual Design, we create a model of the application domain using object-oriented modeling principles. We create a conceptual class for each object of the application domain and create the model by using aggregation, generalization and specialization hierarchies between the classes. We can graphically model this using UML (Unified Modeling Language). To describe these conceptual models textually we use RDF (Resource Description

Framework). To model the constraints and restrictions from the requirements gathering phase we use OWL (Web Ontology Language). In figure 3.5 we see an example of a conceptual model for an academic department[1].

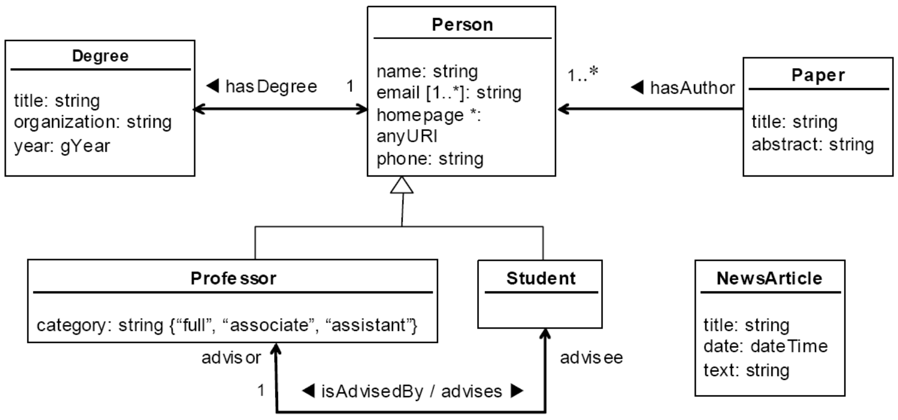


Figure 3.5: OOHDM/SHDM: A conceptual model

Describing the conceptual objects of the web system in this way is similar to the Data Design phase of WebML, where we created a data schema for the application domain. Although both the conceptual model of SHDM and the data schema of WebML describe the name and structure of the concepts, only SHDM is able to describe the behavior and the constraints through use of OWL.

#### 3.3.2.3 Navigational Design

Once we've created the conceptual model, we define *views* over the conceptual objects. We can create multiple views for the same conceptual object. A view describes what objects can be reached by the users, what relations exist between the objects, what sets of objects can be accessed, how these will be accessed and the context of these objects. A view can be described with a *navigational class model*, which is based on the conceptual model we defined earlier. Note the similarity between these views and the *site view maps* of WebML. We create for our conceptual model a navigational class model[2] in figure 3.6.

[1]taken from paper on SHDM by Schwabe et. al. [22]
[2]taken from paper on SHDM by Schwabe et. al. [22]

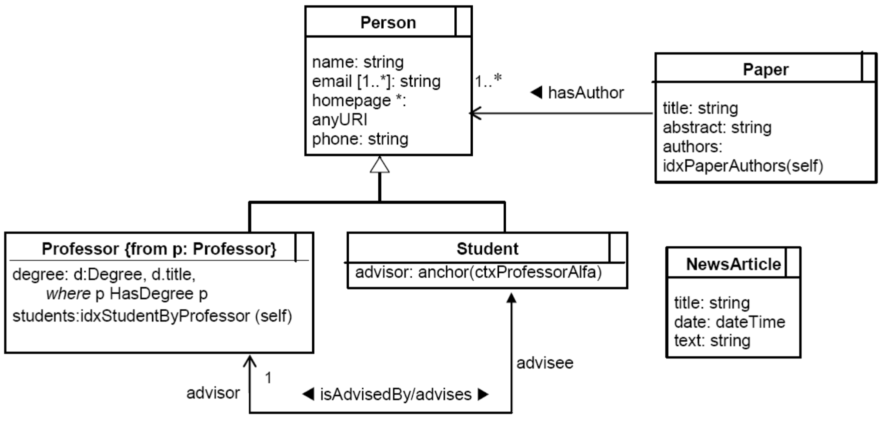


Figure 3.6: OOHDM/SHDM: A navigational model

Next to this navigational class model, we create a navigational context model that defines how the navigation is organized in different *contexts*. Contexts are sets of meaningful navigational objects that describe how concepts can be displayed to the user. Using a Context Definition Card and RQL (RDF Query Language), we can describe and create them.

#### 3.3.2.4 Abstract Interface Design

Once we've created the navigational class model(s), we can make an abstract interface for the web system. This abstract interface will make the functionality and the conceptual objects visible to the user. It is free from any implementation details and only describes how the information is displayed. To create this interface, we use the Abstract Widget Ontology, that gives us the vocabulary to define it. Any interface can be described as a composition of these widgets.

#### 3.3.2.5 Implementation

Once we created the abstract interface, we can then map it onto a concrete interface. We create a *mapping specification* that describes how the abstract widgets will be converted into the real widgets of the user interface. Once we've completed this specification, we can automatically generate concrete interfaces from the abstract user interfaces. These concrete interfaces are normal JSP pages that use Java beans to represent the values of the navigational objects.

If we make use of CSS tags in our mapping specification, so that the abstract widgets will be converted to CSS-tagged objects, it is easier for the graphic designer to define the look and feel of the entire web system.

## 3.4 Hera

### 3.4.1 Introduction

Hera [12, 13, 14, 25] is a model-driven methodology that focuses on the engineering of Web Information Systems (WIS). WISs retrieve information from various sources on the Web and present the information in terms of a hypermedia presentation. At first, these WISs were nothing more than purely informational web sites that were hard-coded in a hypermedia format (e.g. HTML), that included both the presentation and the information. Now, with the growth of available information, there is a need for data-intensive applications that generate their presentation and information, based on the ontologies of the source information and the conceptual model of the application itself.

This shift caused the initiation of Hera, a methodology with an accompanying software suite that supports this generation. Hera divides a WIS into three layers:

- **Semantic Layer:** This layer defines the content that is managed in the WIS. We describe this content with a conceptual model. This layer includes the integration of the information retrieved from various sources on the Web (e.g., information from search agents, databases, wrappers, crawlers, ...) into the conceptual model of the WIS.
- **Application Layer:** This layer defines the navigation view on the data in terms of an application model. This application model shows resemblance to the normal navigational model we discussed in the other web design methods. This layer includes the adaptation of the information retrieved (e.g. personalization, context based on user browse history, ...).
- **Presentation Layer:** This layer describes how the information is presented. A concrete presentation can be generated from the application model.

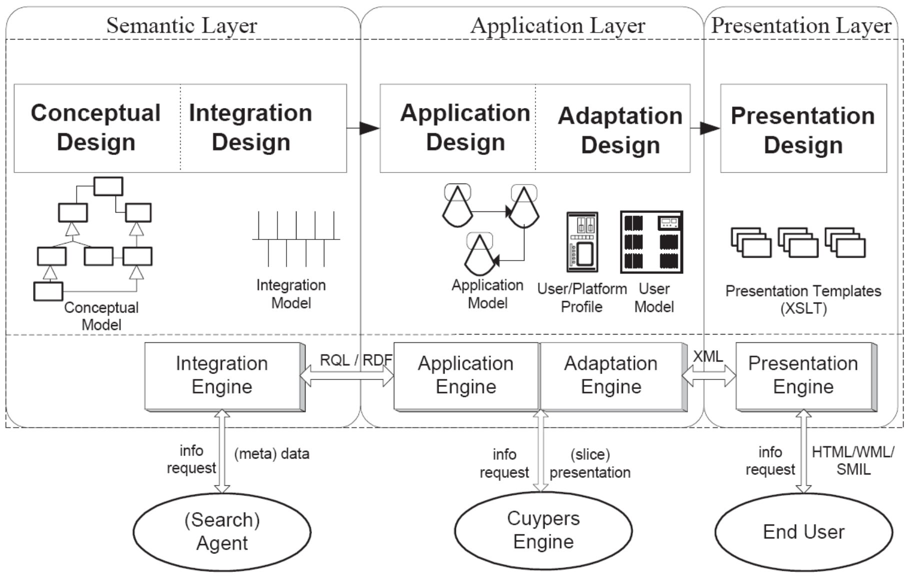


Figure 3.7: Hera: The layers in the Hera methodology

A WIS allows users to perform *informational queries*. The WIS will search for the information requested in its own database and other Web sources and display the query results to the user. We can distinguish two phases in Hera:

- **Integration and Data Retrieval:** The integration sub-phase describes what information is stored in the WIS itself and what information needs to be retrieved from other Web sources. It uses an *integration model* to map the different sources into objects from the conceptual model. To perform this mapping, we take the sources' ontologies, create instances of those ontologies based on the user query and map these instances onto the conceptual model.

  The data retrieval sub-phase handles the reception of the user query. Once the mapping of the *source ontology instance* into concepts has been completed, we create a *conceptual model instance* containing the results (only the necessary concepts) from the query.

- **Presentation generation**: We can then create an *application model instance* from this conceptual model instance while considering possible adaptation. Adaptation changes the application model instance based on context and personalization. Finally, a concrete presentation can be generated.

### 3.4.2 Hera Overview

Most of the Web engineering approaches do not consider integration and adaptation in their methodology as opposed to Hera. Let's take a look at the different phases in Hera and how it copes with integration and adaptation in its design.

#### 3.4.2.1 Integration and Data Retrieval

In this phase, we connect the internal conceptual model with the external information sources of the WIS. For this, we have to map the external source ontologies into concepts of our conceptual model. Figure 3.8 shows how the integration and data retrieval is handled in Hera.

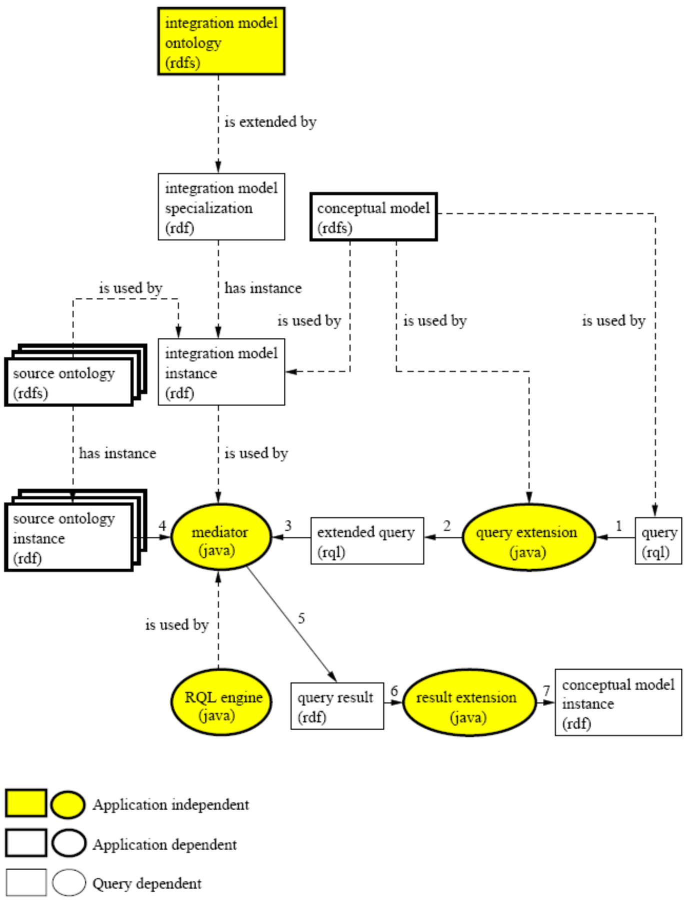


Figure 3.8: Hera: The integration and data retrieval

Let's start from a query, performed by the user of the WIS. The query is transformed into a query the conceptual model can understand, extending it with concepts where needed. This extended query is then delivered to a *mediator*. The mediator is a common *object oriented design pattern* that allows object interaction through its interface, encapsulating the specific details in its structure.

An integration model is already present in the system, before any queries are made. This integration model defines how the external sources will get mapped into concepts. For the query, we create an instance of the model to perform the mapping. Using this instance, together with the source's ontology instance and the conceptual model, we can return a query result. This result can then be translated back into concepts and return a conceptual model instance.

The conceptual model is very similar to the conceptual models we discussed in the other web design methods. It models information objects, their attributes and the relationships between them. A conceptual model can be graphically represented in an ORM-style model. It can be described verbosely in RDF(S).

The external sources of the WIS can be any form of web system. These can be either external databases or information stored in hypermedia format (HTML, WML, SMIL...). Hera assumes that an ontology for each source is available.

The integration model relates concepts of the source ontologies with those from the conceptual model. This process can be done (semi-)automatically through use of parsers with support for dictionaries and lexical matching.

#### 3.4.2.2 Presentation Generation

In this phase, the application model (a type of navigational model) is created from the conceptual model instance, which represents the result of the user query. We take into account the possible adaptation of this application model. Depending on the context or the personalization rules, the application model can be adapted.

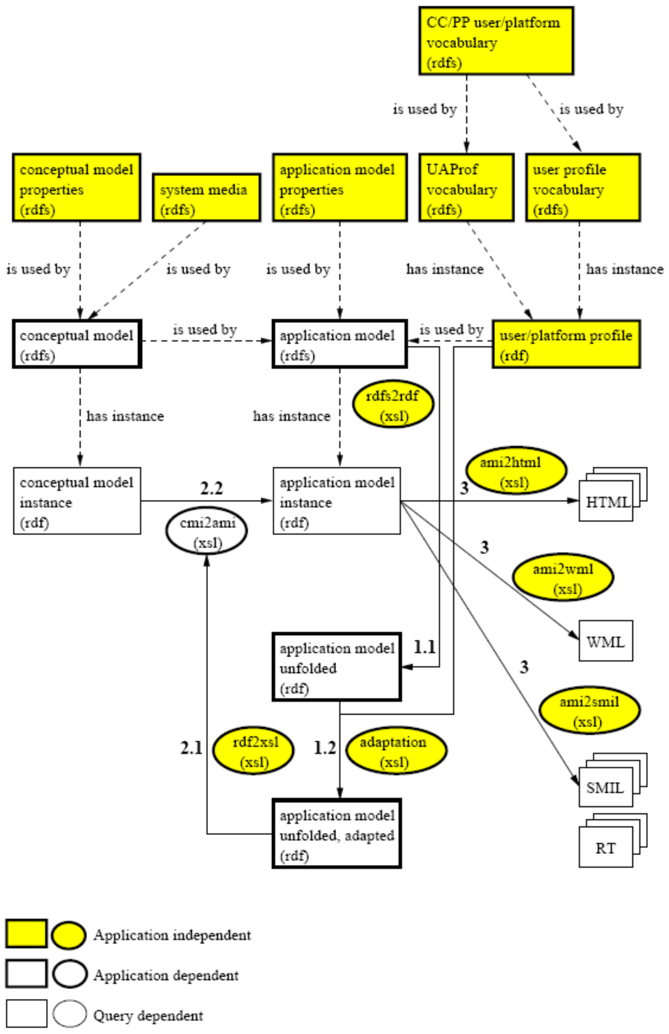


Figure 3.9: Hera: The presentation generation

An application model is made up from *slices* and *slice properties*. A slice is a media content unit, they represent a *media item*. A media item can be a concept from the conceptual model or a concept from an external source ontology. An application model[3] is depicted in figure 3.10.

[3]taken from paper on Hera by Vdovjak et. al. [25]

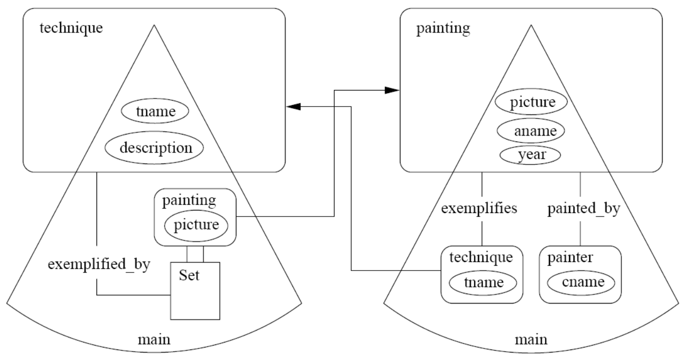


Figure 3.10: Hera: The application model

From this application model, we generate code specific for a user's browser. This code will depend on the type of hypermedia format the browser supports (HTML, WML, SMIL, ...). A media item from our application models can be transformed into a chunk of code that represents it. This transformation can be completely automated. Once all of these media-items are transformed, the WIS will put the code into the result page, where it will then be visible to the user. The code can be generated in such a way that it uses the correct CSS tags of the WIS, so that the results of the query will nicely blend in with the layout and style of the result page.

## 3.5 Comparisons

### 3.5.1 Introduction

In this chapter, we investigate how the conceptual structure of WSDM corresponds to the conceptual structure of the three Web design methods we examined in this chapter. We will enlist the differences in conceptual design between the Web design methods. We then take a look at how the navigational models of WSDM correspond to the navigational models of the three Web design methods we examined: WebML, OOHDM/SHDM and Hera. We will see if these design methods derive navigational structure from their conceptual models as in WSDM and how these steps are or can be automated.

### 3.5.2 Conceptual structure

In this section we examine how the web design methods describe the conceptual structure of the web system. When creating the conceptual model of a web system, we take the requirements of the target users and see what concepts (information objects) are needed to provide the needed functionality later on. Note that this step of the Web design methods cannot be automated. It is up to the designer to decide which concepts constitute the web system or application. We can however support the creation of these models by creating a software suite, corresponding to the web design method.

#### 3.5.2.1 WebML

In WebML, we describe conceptual structure through use of a conceptual model. This is an ER model that identifies concepts, describes the properties of the concepts and signals relationships between the concepts. We create the concepts from the business requirements and environmental constraints. The creation of this ER model happens in the Data Design phase of WebML.

#### 3.5.2.2 OOHDM/SHDM

SHDM uses both a textual and a graphical representation to present the conceptual structure of the web system. As SHDM is object-oriented, the conceptual model is created with UML, a modeling language for object-oriented programming environments. We add a textual OWL representation of each object in the UML to signal possible constraints and restrictions. The creation of this conceptual model is done in the Data Design phase of SHDM.

#### 3.5.2.3 Hera

In the Integration and Data Retrieval phase of Hera, we create the conceptual model of the Web Information System (WIS). Every time a query is handled by the WIS, is then maps the objects from the source ontology into concepts of the conceptual model.

#### 3.5.2.4 WSDM

In WSDM, the conceptual structure is described by the task models, accompanied by the information and functional models or object chunk models. This happens in the Task & Information Modeling sub-phase of the Conceptual Design phase. Instead of using concepts, we define tasks that satisfy the requirements of the target users. These tasks are described by task models and each task interacts with a number of concepts. These concepts are then described in corresponding object chunks. Each object chunk model describes one concept. The collection of all of the object chunk models can be considered as a conceptual model in WSDM.

This is clearly different from the other Web design methods we described. There is no phase in WSDM where the designer writes down all of the concepts of the web system. Instead, we create functionality from the requirements and see which concepts are needed to support this functionality. The functionality is described in the task models, the concepts are described in the object chunk models. The reason for doing this is to support audience-driven design. Linking concepts to tasks, instead of describing all concepts in one conceptual model allows to link the concepts to audience classes.

### 3.5.3 Navigational structure

In this section we examine how the navigational structure is created in the web design methods. Is the navigational structure derived from the conceptual structure of the web system or application? Or is the navigational model created from scratch and totally independent from the conceptual structure? We will answer these questions for each of the web design methods discussed.

#### 3.5.3.1 WebML

In WebML, we create a navigational model in the Hypertext Design phase. A navigational model is actually a view on the conceptual model. This view can contain some of the concepts of the conceptual model and defines how these concepts can be presented to the user. Multiple views can be defined to allow different points of view for different user classes. This step can not be automated; it is up to the designer to decide what views will be defined on the concepts of the conceptual model. WebML does have a tool (WebRatio) that allows creating the views through a graphical user interface and allows linking these views to concepts.

#### 3.5.3.2 OOHDM/SHDM

As in WebML, SHDM creates navigational class models by defining different kinds of views on the conceptual structure of the web system. We create a view by taking the conceptual model, created with UML, and extending it with links that show how the concepts are organized and how they can be presented to the user. The navigational class models are created in the Navigational Design phase. Again, this step cannot be automated. The designer will need to specify how the concepts are links and how they can be represented to the user.

#### 3.5.3.3 Hera

In the Presentation Generation phase of Hera, an application model is created for each concept. Each application model is made up from slices and slice properties. These application models describe how the concepts can be presented to the user. Also, in Hera, these application models will have to be created by the designer.

#### 3.5.3.4 WSDM

In WSDM, the navigational structure is described by navigational models. Navigational models contain nodes (that signal the workflow) and object chunks (that present the information and functionality). The creation of these navigational models happens in the Navigational Design sub-phase of the Conceptual Design phase. Although these can be considered as part of the conceptual structure (as they contain object chunks), these models clearly signify the main navigational structure of the web system as well.

Again, this is clearly different from the other Web design methods we described. Whereas we would create a navigational model on top of the conceptual model in the alternative web design methods, we create a navigational model in WSDM by transforming the task models into a sequential structure. Task models containing concepts (object chunks) are converted to a navigational structure which links these concepts to nodes. In the other web design methods, we expand our conceptual model by defining (multiple) navigational views on it.

We can automate the derivation of navigational models from task models. The difference with the other web design methods is that we already captured some of the navigational structure in our task models, while the conceptual models of the other web design methods do not provide any navigational information. We might seem to be doing twice the work in WSDM, since we describe how tasks are organized in the task models and then describe the workflow of these task in the navigational models. But the task models actually allow us to talk about functionality in a high-level, conceptual manner. When we automate the generation of a navigational structure from these high-level task models, this mechanism actually becomes more powerful than creating the navigational structure from scratch, as in the other web design methods.

# Part II

# Research

# Chapter 4

# Task Model Design Patterns

## 4.1 Introduction

Design patterns are related to a particular problem that often reoccur when designing systems. A design pattern explains how the problem can be solved, in what context this problem may occur and what forces constitute to the need for a solution. The design pattern expresses a relation between problem, context and solution. Whereas guidelines capture some design knowledge into small rules, these rules are often too abstract or simplistic to be used properly. Design patterns solve this by giving concrete examples of how and where the design pattern can be used.

Design pattern can be used for different aspects of design, e.g. in [23, 24] usability design patterns are considered. Here, we propose design patterns for task models. Providing patterns for commonly used situations in task modeling can be considered as a way of further "automating" WSDM. They may help the designer saving time when creating the task models for the web system. When task design patterns are available, the task models wouldn't have to be created from scratch; we could use design patterns and add new tasks to them where needed. New patterns can be derived from existing task models. The use of design patterns increases the reusability of designs of web systems in WSDM.

Before creating the design patterns, we take a look at how design patterns are usually described. For this the following characteristics are used:

- **Problem**: This describes the problem. In the case of task models, this is will be the requirement to be satisfied by the task.
- **Context:** This describes a context in which the design pattern can be used. Only in this context, the design pattern will turn out to be useful for the designer.
- **Forces:** This describes the forces that constitute to the need for the design pattern. These forces describe how the problem may occur.

- **Solution:** This describes the solution to the problem.
- **Examples:** This describes some examples of context in which the design pattern can be used.

In this chapter, we present some task model design patterns. We will start off simple by presenting design patterns for confirmation and validation (sections 4.2 and 4.3). Then we will propose a design pattern for posting a comment or message on a forum or guestbook (section 4.4). Finally, we give the design pattern for a container that can store items. These items can be products, hyperlinks, friends' names... We discuss the container pattern in section 4.5.

## 4.2 Confirmation

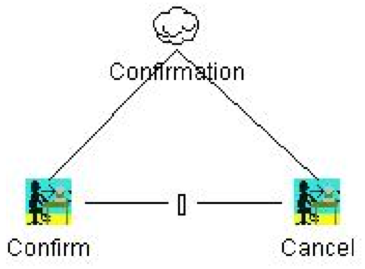


Figure 4.1: CTT: The Confirmation Pattern

| | |
|---|---|
| Problem | The user wants to complete an action. For safety reasons the system needs to know that the user is absolutely sure that he wants to complete the action, because this action might be important and/or irreversible. |
| Context | The Confirmation pattern can be used when a user needs to perform an action that is important to the user or the system. This action usually changes the state of the system in an irreversible way. The action is significant enough to require a final confirmation of the user to complete it. |
| Forces | - The user might be distracted when completing the action.<br>- The user might have chosen to complete the action by accident. |
| Solution | **When the user chooses to complete the action, the system will ask for confirmation whether or not the user really wants to complete the action. This will indicate to the user that the task is important and that the decision of completing it should not be taken lightly.** |
| Examples | - The user wants to save the current file by overwriting another.<br>- The user wants to quit the application while there are open unsaved documents.<br>- The user wants to add an item to the database. |

## 4.3 Validation

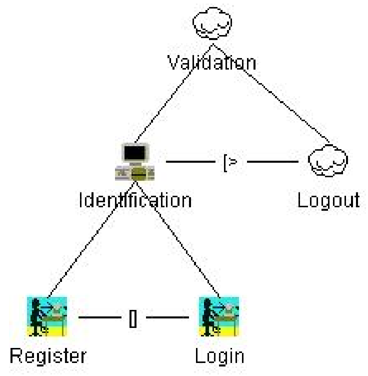


Figure 4.2: CTT: The Validation Pattern

| | |
|---|---|
| Problem | Before any task can be done in the system, the user will need to login, or register if he doesn't have any login details yet. When he wants to leave the system, the user can logout. |
| Context | The Validation pattern can be used in any (web) system that uses a login/logout scheme. |
| Forces | - The user is not registered and wants to perform a task that requires identification.<br>- The user is not logged in and wants to perform a task that requires identification.<br>- The user is logged in and wants to logout. |
| Solution | **The user has the choice of logging in or registering when he is not yet in the system. When he has logged in or registered, the user can then get out of the system by logging out.** |
| Examples | - A user wants to register on a commercial web site.<br>- A user wants to check his online bank account and wants to login to the system. |

## 4.4 Post Message

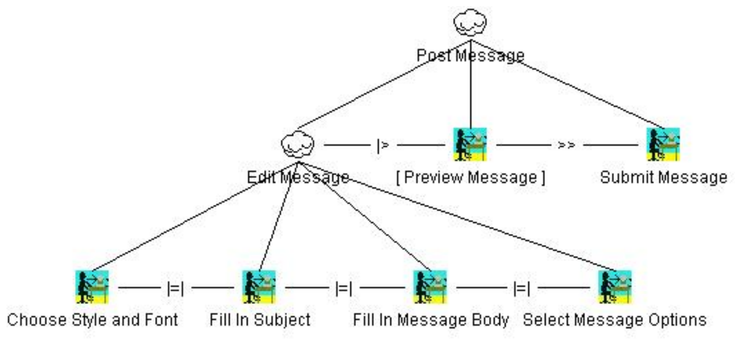


Figure 4.3: CTT: The Post Message Pattern

| | |
|---|---|
| Problem | The user wants to post a message on web site. This message has a subject and a body. |
| Context | The Post Message pattern can be used in any (web) system that uses a message blackboard. This message blackboard can be a guestbook, a topic of a forum, a comment review system... |
| Forces | - The user wants to write a message and post it in the system. |
| Solution | **The users need to create a message first. A message consists of a subject, a message body, the layout of this message body and some various options (whether or not the e-mail of the user is shown, whether or not the message is a poll...). While editing the message, he can stop the editing process and preview the message. When he's done editing the message, he can post the message. A validation pattern can be added to confirm if the user really wants to post the message.** |
| Examples | - A user wants create a new topic on a community forum.<br>- A user want to post a new comment on the guestbook of his friend. |

## 4.5 Container

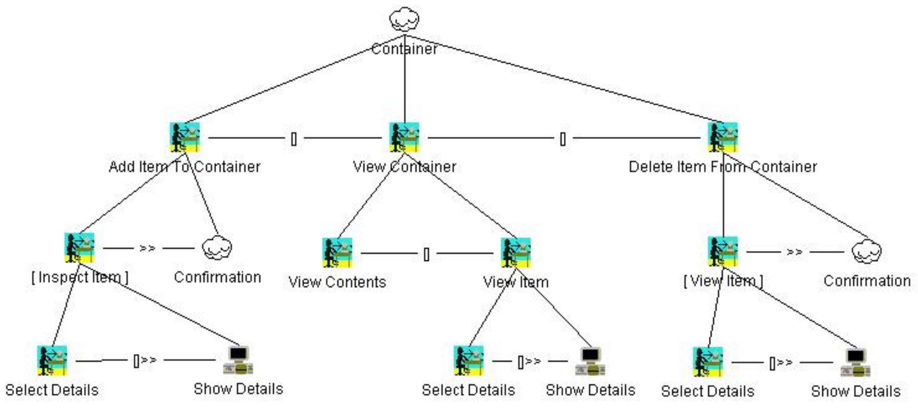


Figure 4.4: CTT: The Container Pattern

| | |
|---|---|
| Problem | A container type is needed in the system. A user should be able to add items to this container, view the contents of the container and delete items from the container. |
| Context | The Container pattern can be used in any (web) system that needs a container type. |
| Forces | - The user needs to keep track of a group of items. |
| Solution | **The user has a choice of adding items, viewing items or deleting items. When adding an item, he can inspect this item to show its details. This happens outside the container. When he wants to view the contents or delete an item, he can view the item. This happens inside the container. A Confirmation pattern was used since adding and deleting items change the state of the system.** |
| Examples | - A user wants to store products in his ”shopping basket” before he checks out.<br>- A user wants to add some friends to his friends list on his personal webpage. |

Chapter 5

# Transforming Task Models into Task Navigational Models

## 5.1 Introduction

As mentioned earlier, the *Conceptual Design* phase of WSDM has two subphases: the *Task & Information Modeling* phase and the *Navigational Design* phase. After creating the task models, we create data objects or object chunks for each of the elementary tasks in the task models. When this is done, we create a task navigational model for each task model. Putting these task navigational models together for an audience class, results in a navigational track for that audience class. We can then create a navigational model out of these navigational tracks.

A major step in this phase is the transformation of the task models into task navigational models. The underlying principle to do this is as follows. Each elementary task has to become a node, while each temporal operator can become a link or multiple links. Currently, this is done manually, although there are certain principles that can be used when transforming these task models. A simple example is that an *interaction task* in a task model will result in a *node* in the task navigational model. Another example is that an *enabling temporal operator* will result in a simple *1-to-1 link*. Using these principles, this step of the conceptual design could be automated.

By applying a set of procedures based on the type of the task or temporal operator, we could automate this whole process. Task models could then be transformed into task navigational models by one click on a button. The ideal situation would be that a task model is created by the user in a graphical environment, saved to disk in a predefined format, then deciphered by the generation tool and finally the resulting task navigational model is displayed on the screen. Once all the task navigational models are generated, it is easy to create the navigational model from them.

In this chapter, we will discuss the transformation algorithm. In section 5.2 we give an overview of the steps of this algorithm. Then we describe how we represent task models in a programmatic environment. We give the basic structure of this representation in section 5.3. Afterwards, we examine the first step of the transformation algorithm in section 5.4 and the second step in 5.5.

## 5.2 Algorithm Overview

Task models represent a hierarchy, where the root task is the most complex one. It is decomposed into a number of sequential sub-tasks, which are in turn decomposed further on. Navigational models represent a sequence, where we have a well-defined start and end of the sequence. Our automation process will have to make a clear distinction between these two representations. It will have to turn the tree of tasks into a sequence of tasks, interleaved by temporal operators to connect them. This is the first step in the transformation process.

Once we have created a sequence of tasks from the task model, we have to look at the types of temporal operators and tasks this sequence contains. Based on the types, we can apply a set of procedures and generate nodes and links accordingly. We also have to examine common task model properties like recursion, iteration, optional behaviour, embedded choice... This is the second and final step of the transformation.

1. Transform the task model into a textual representation.
2. Retrieve the needed information for the transformation.
3. Create a task sequence from the task model tree.
4. Create a navigational model from the task sequence.

Before we elaborate on the transformation of task models, we first will describe how we represent task models in a programmatic environment. We describe the basic structure of this representation. We then describe the two steps of the transformation process in detail.

## 5.3 Task Model Representation

As seen in chapter 2 on WSDM, we know that task models are represented as *ConcurTaskTrees* or *CTTs*. This is a graphical notation that allows us to depict tasks in a tree, with the task we are describing as the root node of this tree. The tree is made up from tasks and temporal operators, the tasks being the nodes of the tree and the temporal operators being the links between nodes on the same level in the tree. To be able to process these graphical task models, an internal representation is needed.

The *ConcurTaskTree Environment* or *CTTE*[1] is a graphical environment in which we can create CTTs. This tool allows saving a CTT as an XML structure. This structure gives us a way of traversing the task models more easily. Therefore we will start from this representation. The details of how the conversion from the graphical representation into the XML representation works, is beyond the scope of this dissertation.

In the XML representation, the root task of the task model tree is the root node of the generated XML document. This root node contains its direct descendant tasks as its child nodes. In turn, each of these child nodes contains its direct descendants as child tasks. A notable fact is that the temporal operators are kept in these nodes as well, they are stored in a field of the task node where they start from. An example of this XML representation can be seen in the section on Implementation on page 80.

We can save this structure in an file, which allows us to traverse it programmatically. A node contains multiple fields that describe the information of the task it represents. Each node contains the following useful information:

- **Identifier:** This field contains the name of the task in the task model.
- **Category:** This field contains the category of the task. In WSDM, a task can be either an interaction task, an abstraction task or an application task.
- **Iterative:** This field contains a boolean which specifies if the task is iterative or not.
- **Optional:** This field contains a boolean which specifies if the task is optional or not.
- **Frequency:** This field contains the frequency of a task.
- **TemporalOperator:** The temporal operator that appears right next to the task on the right. When there is no temporal operator to the right of the task, this field is omitted.
- **Parent:** Specifies the identifier of the parent of the task. If the task is the root task, this field is omitted.
- **SubTask:** This field contains all the child nodes (tasks) of the task at hand. The child tasks are sorted in the way they appear in the task model, from left to right. Each child task contains the same type of information as the parent task. This is how the tree is made up: a task has child tasks, who in turn have child tasks, ...

[1]the ConcurTaskTrees Environment by Paterno et. al. [15]

Other information is stored as well, but this is not important for our algorithm because they are either implicit, redundant or can't be presented in task navigational models:

- **Name:** This field contains a user-defined name of the task that isn't shown in the task model itself. Since we already have an identifier, we cannot and don't have to represent this name in the task navigational model.
- **Frequency:** This field indicates how often a task is performed in a task model. There is no way of presenting this in the task navigational model, unless we expand it so we can.
- **SiblingLeft:** This field contains the identifier of the task on its left. This information is redundant, since we can simply check in the parent of the task what the child task next to it is.
- **SiblingRight:** This field contains the identifier of the task on its right. This information is also redundant.

Finally, there is information that is not available in the original XML representation but that we derive or add, since we need it later on to have the necessary data for the set of rules we want to apply. For each node (task) we deduce or add the following information:

- **Level:** The level of the task in the tree. From the root task which is level 1, we add one level whenever we go down one step in the tree. So the higher we are in the tree, the lower the level of the tasks will be.
- **UniqueID:** We assign an ID to each task to distinguish them throughout our algorithm. Tasks can have the same identifiers. This is necessary to establish recursion, where two tasks will have the same identifiers which signifies the recursion call.
- **Position:** Since we are not using the SiblingLeft and SiblingRight fields, we store the position (index rather) of the child in the list of childs of its parent. That way we can easily ask the parent task what the right sibling and left sibling of the task is.
- **RecursionID:** This is the ID of the task that is the target of the recursion, if recursion appears. We will describe this more thoroughly later, in section 5.5.4.
- **OptionalList:** Instead of storing a boolean whether or not the task is optional, we keep a list of optional behaviour for each task: we store the levels where the task is optional. A task can be optional on many levels. If a task its parent is optional, then that means that task is optional through inheritance. We introduce concepts like *inherited optional behaviour* and *direct optional behaviour* and discuss this further in section 5.5.7.

- **IterativeList:** Instead of storing a boolean whether or not the task is iterative, we keep a list of iterative behaviour for each task: we store the levels where the task is iterative. A task can be iterative on many levels. If a task its parent is iterative, then that means that task is iterative through inheritance. We introduce concepts like *inherited iterative behaviour* and *direct iterative behaviour* and discuss this further in section 5.5.7.

When we traverse the XML tree in an object-oriented environment, we can convert each node into a *Task object*. We give this object the relevant and derived fields we discussed just now, together with the logically derived accessor methods. Besides the standard accessors, there are 10 other methods we need for traversing the *Task object* tree structure.

- **GetSiblingLeft():** As mentioned earlier, the information on siblings can be derived through use of the *parent* and *position* attribute.

```
method GetSiblingLeft()
if parent == null
  then return null
  else if position − 1 < 0
        then return null
        else return parent.GetChildren()[position − 1]
      end if
end if
```

- **GetSiblingRight():** The same applies to this method. The information can be retrieved by using the parent task and the *position* attribute of the Task object.

```
method GetSiblingRight()
if parent == null
  then return null
  else if position + 1 > parent.GetChildren().Length − 1
        then return null
        else return parent.GetChildren()[position + 1]
      end if
end if
```

- **Elementary():** Another straightforward implementation to check if a task is an elementary task or not. A task is elementary when it has no child tasks, so we only have to check if the children list was set or not.

```
method Elementary()
return (children == null);
```

- **LastOfSiblings():** Check if a task is the last of a set of child tasks. Again, this is very easy to check.

```
method LastOfSiblings()
return (GetSiblingRight() == null);
```

- **HasParent():** Check if a task has a parent.

  ```
  method HasParent()
  return (parent! = null);
  ```

- **SetRecursion():** This method checks for recursion for one particular task. Recursion arises when a task has the same identifier as one of its ascendants. The task can then be replaced by that ascendant. Recursion is not the same as iteration, since you can choose when and where in the sub-hierarchy of your task the recursion occurs. After checking if recursion did occur, this method will set the *recursionID* attribute to the *uniqueID* of the ascendant. We will need this information later on when handling recursion in the task models.

  ```
  method SetRecursion()
  Task currTask = parent
  while currTask != null do
          if currTask.identifier == identifier
             then recursionID = currTask.GetLeftMostLeaf().uniqueID
                  break
             else currTask = currTask.GetParent()
          end if
  end do
  ```

- **GetLeftMostLeaf():** A simple method of getting the left-most leaf of a task in the tree.

  ```
  method GetLeftMostLeaf()
  if Elementary()
     then return this
     else return children[0].GetLeftMostLeaf()
  end if
  ```

- **NextOperator():** We want the first operator to the right of the current task. If the temporal operator is not set, that means this task is the last of the siblings. We traverse upwards in the tree to look for the next temporal operator.

  ```
  method NextOperator()
  if temporalOperator == null
     then if parent == null
             then return null
             else return parent.NextOperator()
          end if
     else return temporalOperator
  end if
  ```

- **NextOperatorLevel():** Exactly the same as the previous method, but here we need the level, instead of the name of the temporal operator.

  ```
  method NextOperatorLevel()
  if temporalOperator == null
  ```

```
  then if parent == null
         then return 0
          else return parent.NextOperatorLevel()
       end if
   else return level
end if
```

- **NextTask():** This method forms the basic mechanism in the creation of the sequence. From a leaf task, we go to the next leaf task by using this method. We traverse the task tree from left to right.

```
method NextTask()
if Elementary() ∧ LastOfSiblings() ∧ HasParent()
  then return parent.NextTask()
   else if Elementary() ∧ LastOfSiblings()
         then return null
          else if !HasParent() ∧ LastOfSiblings()
                then return null
                 else if HasParent() ∧ LastOfSiblings()
                       then return parent.NextTask()
                        else
                            Task sibling = GetSiblingRight()
                            if sibling.Elementary()
                              then return sibling
                               else return sibling.GetLeftMostLeaf()
                            end if
                      end if
                end if
       end if
end if
```

The task models are represented as trees composed of *Task objects.* These objects have a list of attributes and methods that allow us to traverse this tree. Methods like NextTask() and NextTemporalOperator() will help us with creating the sequence of tasks. With this background, we can examine the first step of the transformation algorithm in greater detail.

## 5.4 From Task Models to Task Sequences

To create task navigational models, we need to take all the elementary tasks of a task model and create a sequence out of them. A sequence is literally an order of succession, a following of one thing after another. While trees are made up from a hierarchy, we will need to parse this tree and list the elementary tasks in such a way that we can later easily create the task navigational model out of them.

We create this sequence by listing the elementary tasks from left to right, interleaved by the temporal operators that connect them. This can be obtained by a depth-first traversal technique. We start at the left-most leaf of the tree and traverse each leaf of the tree, adding them to the sequence. To clarify, we use the example in figure 5.1.

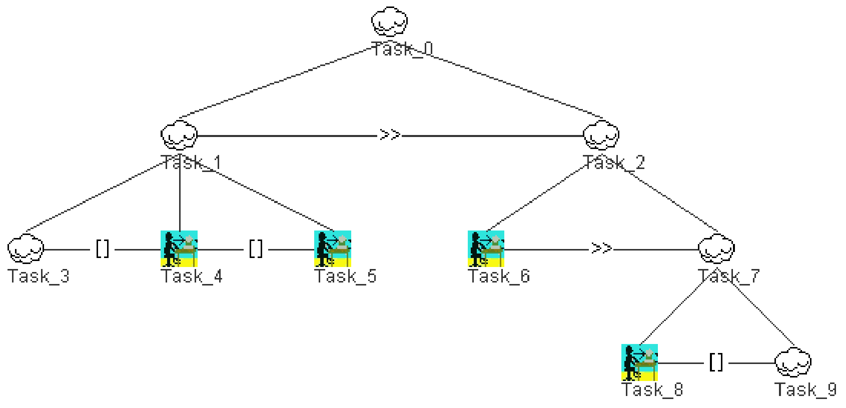


Figure 5.1: CTT: A sequence of tasks

There, we see a task model that has 6 elementary tasks: 4 interaction tasks and 2 abstraction tasks. There are 5 temporal operators that connect these elementary tasks: 3 choice temporal operators and 2 enabling temporal operators. The sequence of this tree is created by parsing the tree, starting at the left-most leaf, *Task_3*. We then traverse the tree looking for the next leaf in line, using depth-first traversal. Our sequence would look like the one in table 5.1.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Task_3 | [] | Task_4 | [] | Task_5 | >> | Task_6 | >> | Task_8 | [] | Task_9 |

Table 5.1: An example of a task sequence

Although this sequence is constructed correctly, we lose some of the information contained in the tree structure. This is why we introduced derived attributes like *level*, *optionalList*, *iterativeList* and *recursionID* to tackle problems like *choice chains*, *optional and iterative behaviour* and *recursion* respectively. We store this derived information in this sequence.

In this step of the algorithm we also consider temporal operators as separate objects. They are no longer considered as part of a task, they need to be treated independently. We introduce three new types of objects: *TaskModelObject*, *TemporalOperator* and *CttTask*. A *CttTask* stores all the useful information from the task, a *TemporalOperator* stores the level and type of the temporal operator. *CttTask* and *TemporalOperator* inherit common behaviour from the abstract object *TaskModelObject*. *TaskModelObject* allows us to store the tasks and temporal operators in the same list in the object-oriented environment. In the code below we show how we create the task sequence from the tree of Task objects.

```
method GenerateTaskSequence(Task task)

CttTask cttTask = null
TemporalOperator tmpOperator = null
task.SetRecursion()

string ident = task.GetIdentifier()
List < int > iter = task.GetIterative()
List < int > opt = task.GetOptional()
int level = task.GetLevel()
int id = task.GetUniqueID()
int rec = task.GetRecursionID()

string tmpType = task.NextOperator()
int tmpLevel = task.NextOperatorLevel()

switch task.GetCategory()
  case "UserTask" :
    cttTask = new UserTask(ident, iter, opt, level, id, rec)
  case "ApplicationTask" :
    cttTask = new ApplicationTask(ident, iter, opt, level, id, rec)
  case "InteractionTask" :
    cttTask = new InteractionTask(ident, iter, opt, level, id, rec)
  case "AbstractionTask" :
    cttTask = new AbstractionTask(ident, iter, opt, level, id, rec)
end switch

switch tmpType
  case "Concurrent" :
    tmpOperator = new TemporalConcurrent(tmpLevel)
  case "ConcurrentInfo" :
    tmpOperator = new TemporalConcurrentInfo(tmpLevel)
  case "Choice" :
    tmpOperator = new TemporalChoice(tmpLevel)
  case "OrderIndependent" :
    tmpOperator = new TemporalOrderIndependent(tmpLevel)
  case "Deactivation" :
    tmpOperator = new TemporalDeactivation(tmpLevel)
  case "SuspendResume" :
    tmpOperator = new TemporalSuspendResume(tmpLevel)
  case "Enabling" :
    tmpOperator = new TemporalEnabling(tmpLevel)
  case "EnablingInfo" :
    tmpOperator = new TemporalEnablingInfo(tmpLevel)
end switch

taskSequence.Add(cttTask)
if tmpOperator != null
  then taskSequence.Add(temporalOperator)
end if
```

```
63: GenerateTaskSequence(task.NextTask())
```

- **Line 1:** the algorithm is defined as a method which takes a Task as input argument. The first time we call this method, this Task will be the leftmost leaf of the Task tree. This way, we will process the tasks from left to right, recursively.
- **Line 3 and 4:** We declare the two new object variables. They are initially set to **null**.
- **Line 5:** We still haven't checked for recursion, so we do this here. This method looks for possible recursion and sets the *recursionID* to the *uniqueID* of the task it refers to in the recursion.
- **Line 7 to 12:** We introduce a new set of variables that store the information needed of the task. We need this information to put it into the sequence. The *identifier* and *uniqueID* help us identify each task in the sequece. The *iterativeList*, *optionalList* and *recursionID* help us with dealing with iteration, optional behaviour and recursion respectively. The level gives us information on the tree structure; we will still need this information.
- **Line 14 and 15:** We get the type of the temporal operator that is to the right of the task. If there is no temporal operator to the right of the task in the tree, this will look in the lower or higher in the tree for the next temporal operator. We also get the level of this temporal operator; we need this information to process the sequence later on.
- **Line 17 to 30:** The objects *UserTask*, *ApplicationTask*, *InteractionTask* and *AbstractionTask* inherit the behaviour of *CttTask*. They each store the information in their attribute structure. When breaking down the task model tree, we only want the abstraction and interaction elementary tasks. Note that we don't really need the application tasks and user tasks, as these are not represented as nodes in the navigational models. We will get rid of these in **line 61** where we add the *CttTask* to the sequence only if this *CttTask* is an interaction or abstraction task. More on this is given in section 5.5.5.
- **Line 32 to 57:** The objects *TemporalConcurrent*, *TemporalConcurrentInfo*, *TemporalChoice*, *TemporalOrderIndepedent*, *TemporalDeactivation*, *TemporalSuspendResume*, *TemporalEnabling* and *TemporalEnablingInfo* inherit the behaviour of *TemporalOperator*. They only store the level where the temporal operator was found. There will be temporal operators that have to be replaced since they can't be presented on a task navigational model, concurrency for example. More on this is given in section 5.5.6.
- **Line 59 to 62:** We add the task to the sequence, then we add the temporal operator to the sequence.
- **Line 63:** This is a form of *tail recursion* where we further expand the sequence with the next task in line.

Now that we know how the task sequence is constructed and what information is stored in it, we can examine the second step of the algorithm: the generation of nodes and links for the task navigational model. This generation is based on a simple set of procedures that are based on the type of the task or the temporal operator.

## 5.5 From Task Sequences to Task Navigational Models

### 5.5.1 Introduction

In this step of the transformation algorithm, we will go over the sequence and generate nodes and links based on the task or temporal operator we encounter. We will explain the basic algorithmic behaviour step by step, covering all the details and clarifying each procedure with an example. Throughout this section, we will use the same example: a task model that describes a search task where a query is used to search and where you can add filters to filter results from the search. This task model is depicted in figure 5.2.

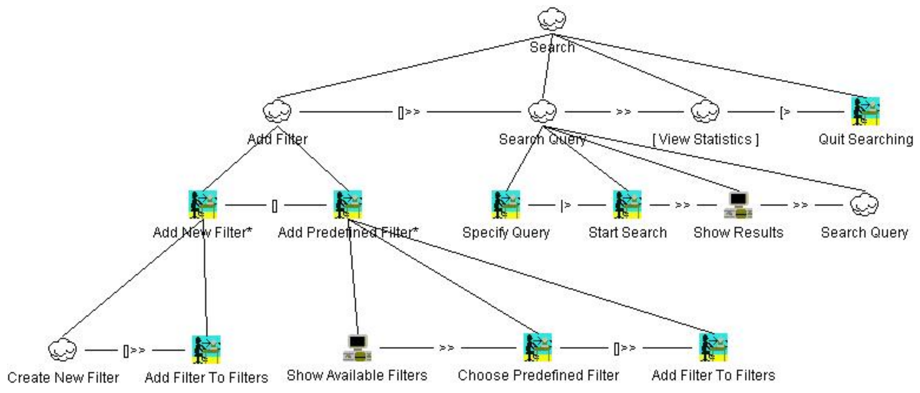


Figure 5.2: CTT: Query searching with filters

Before we elaborate on the principles of the algorithm, we need some background knowledge of how nodes and links are constructed in our object-oriented environment. We discuss this briefly in section 5.5.2. Afterwards, we outline the basic structure of the second step of this algorithm. We can then examine recursion and what problems we face when dealing with recursion. After this, we describe how tasks are converted to nodes and take a look at the temporal operators and how they help us generate links for the task navigational model. Finally, we explain how we deal with optional and iterative behaviour in the task sequence.

### 5.5.2 Nodes and Links

Nodes and links don't store a lot of information. Nodes only need to know its identifier, and whether it is an external node or a normal one. Links only need a source and a target node. Again, we create objects for these in our object-oriented environment. We introduce three new objects: *Drawable*, *Node* and *Link*. Link and Node inherit common behaviour from the abstract object *Drawable*. The Drawable object allows us to treat links and nodes as equals, storing them in the same structure. In our case, this structure is a list. Note the similarity with TaskModelObject, CttTask and TemporalOperator.

These Nodes and Links act as placeholders for the real visible nodes and links of a navigational model. Nodes and Links are objects that store the necessary information to later draw actual nodes and links on an empty drawing page. The drawing is done after this second step of the algorithm, when we go over the list of drawables and create a graphical component for each Drawable in this list.

A Node stores three things that it needs to know. We create a node with the constructor *Node(index, external, id)*.

- **ID:** The *uniqueID* gets stored in this attribute field. This is needed to successfully identify a task, since some tasks can have the same name.
- **Index:** This is the index of the task in the sequence that the node refers to. When it needs information like the *identifier* of the task, it can retrieve it from the sequence using this index.
- **External:** A boolean value which indicates whether or not the node is external. This is only stored for our convenience, because we could retrieve this information from the task in the sequence as well.

A Link stores two things. We create a link with the constructor *Link(index, target)*.

- **Index:** This is the index of the task in the sequence that is the source of the link.
- **Target:** This is the index of the task in the sequence that is the target of the link.

This construction might seem illogical at first sight. One could ask the question: Why not store all the needed information in the Nodes and Links, instead of referring to the sequence? Also, why not store the links in the nodes and have the links refer to nodes instead of indices? This would seem to make more sense, but we simply cannot do this, because some of the links are created before the nodes even exist. There are no nodes to refer to and no nodes to store the links in at that point. This will become clear in section 5.5.4 on recursion.

### 5.5.3 Structure

Before we process the tasks and temporal operators from the sequence, we need to do some preliminary work. Before we start traversing the sequence, we add a *START* and *END* task to it. This is useful for our algorithm, but also useful in our task navigational model. It will indicate the start and end of the task navigational model, clearly marking what task(s) you start with and what task(s) you end with. We give these tasks the level 0, a level that's lower than anything else in the task model tree.

The *START* task will get added in the beginning of our task sequence, together with an enabling temporal operator to connect it with the first task in the sequence. The *END* task will get added at the end of our task sequence, preceded by another enabling temporal operator. Both of these temporal operators' level is also set to 0. This start and end mechanism helps us in adding clear boundaries to the sequence. When we create the task sequence for our example in figure 5.2 and add these boundaries, we get a sequence like the one listed in table 5.2.

| **Name** | START | >> | Create New Filter | []>> | Add Filter To... | [] | Choose Predefined... | []>> | Add Filter... | []>> | Specify Query | \|> | Start Search | []>> | View Statistics | [> | Quit Searching | >> | END |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Level** | 0 | 0 | IV | IV | IV | III | IV | IV | IV | II | III | III | III | II | II | II | II | 0 | 0 |
| **Iterative** | | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | | | | | |
| **Optional** | | | 2 | | 2 | | 2 | | 2 | | | | | | 2 | | | | |

Table 5.2: The task sequence of Query Searching

After adding the start and end to the sequence, we will traverse it twice. The first traversal is to deal with recursion, the second is to deal with the rest. In the first traversal, the sequence might get modified, we might need to delete some tasks and temporal operators from the sequence. The sequence of table 5.2 is the sequence we would get after the first traversal. For every task, we look at its type. Based on this type, we create a node and store it in a list of drawables (a list that accepts objects of the type Drawable). For every temporal operator, we perform a procedure based on its type and add the resulting link(s) to that same list of drawables. These procedures can generate links that are already in that list, but we don't mind since these doubles will get removed before being drawn in the task navigational model.

```
for index = 0 to taskSequence.Length do
    current = taskSequence[index]
    ...
    we handle the recursion in the first traversal
    ...
end for
for index = 0 to taskSequence.Length do
    current = taskSequence[index]
    ...
    we handle everything else in the second traversal
    ...
end for
```

Note that throughout the following sections, we make use of two lists. One containing the sequence, named *taskSequence*, the other containing the generated links and nodes, named *drawables*, which is empty initially. We can now take a look at how tasks are converted to nodes.

### 5.5.4 Recursion

Recursion occurs when a task has the same name as one of its ascendants. Recursion only generates one link to indicate the recursive call. The task making such a call, the descendant with the same name, will never be a part of the task navigational model. This is because the task only indicates a recursive call and has no other purpose than that. So we have to delete this task in the first traversal of the task sequence, otherwise it will be treated as a normal task. Due to the very nature of the algorithm (working with indices to make the nodes and links), we have to be very careful when deleting elements from the sequence. That's why we need to check all the links we generated for possible shifts when deleting this task.

The deletion of tasks (and the corresponding temporal operators) is the reason why we handle recursion in a separate traversal of the sequence. Recursion gets processed in the first traversal, because the only links that could have been generated at that point, are links that indicate recursion. Generally, we would only need to check a few links this way, which decreases the cost in performance.

Recall that we checked for recursion in the first step of the algorithm, before we created the sequence. In the first traversal of the sequence we still have the recursive tasks in the sequence. When a task's *recursionID* is set (the variable *rec* in the code), it is set to the ID of the task that will be the first in line after making the recursive call. We take a look at how recursion is treated by the algorithm.

```
for index = 0 to taskSequence.Length do
    current = taskSequence[index]
    recursive = −1

    if current.rec >= 0
      then for i = 0 to taskSequence.Length do
                if taskSequence[i].id == current.rec
                  then recursive = i
                       break
                end if
            end for

            if recursive == 2
              then drawables.Add(new Link(index − 2, 0))
              else drawables.Add(new Link(index − 2, recursive))
            end if

            foreach drawable in drawables
                    if drawable.target > index
                      then drawable.target -=2
                    end if
                    if drawable.index > index
                      then drawable.index -=2
                    end if
```

```
25:             end foreach
26:
27:             taskSequence.RemoveAt(index)
28:             taskSequence.RemoveAt(index − 1)
29:     end if
31: end for
```

- **Line 1:** We start the first traversal of the sequence here.
- **Line 2 to 3:** We introduce a variable *recursive* that will store the index of the first task in line after the recursive call would have been made. We will use the found index to generate a link.
- **Line 5 to 11:** If the *recursionID* is set, we find the index of the task which ID corresponds to this recursionID.
- **Line 13 to 16:** In the case that the recursive call goes back to the first task in the sequence, we rather want it to go back to the START task, instead of the first real task in the sequence. This is usually the case when recursion on the root task of the task model occurs. Else, we create a link from the current task (the task before the recursive one, since that one will get deleted) to the task we found.
- **Line 18 to 25:** We delete the recursion task from the sequence and the temporal operator next to it. We might need to shift some link indices, we do this here.
- **Line 27 to 28:** We remove the task and its temporal operator from the sequence.

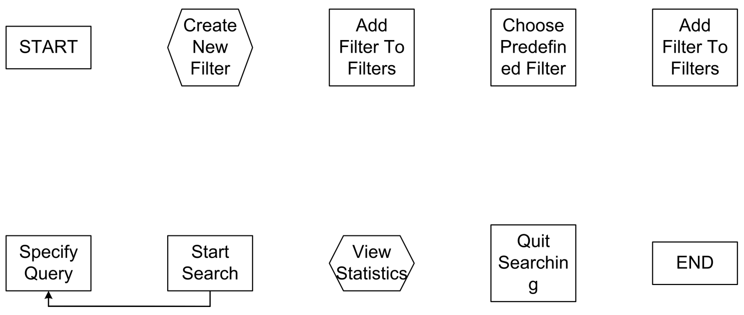


Figure 5.3: Algorithm: Recursion

**Example:** For our example in figure 5.2, there is one recursive task. Search Query appears twice in the task model and is the recursive task here, since one of the two is the ascendant of the other. The descendant is the last of the siblings and gets removed from the task sequence once the link has been generated. Since Show Results is an application task and application tasks didn't get added to the sequence, we take Start Search as the end of our Search Query task. We add a link from Start Search to Specify Query, which is the begin of the Search Query task. At this point, no Nodes exist yet, so we create a Link from indices that will later become Nodes.

### 5.5.5 Tasks

Elementary tasks are converted into nodes in the second traversal of the task sequence. An elementary interaction task will generate a node. An elementary abstraction task means that this task is explained in another task model, it is a form of referring to this other task model. In a task navigational model, we represent this by means of an external node. Every abstraction task of our sequence (which is only there if it is an elementary task) will generate an external node.

```
if current is InteractionTask $\vee$ current is AbstractionTask
  then
      ...
      we handle iterative and optional behaviour here
      ...
      if current is AbstractionTask
        then drawables.Add(new Node(index, true, current.GetUniqueID()))
        else drawables.Add(new Node(index, false, current.GetUniqueID()))
      end if
end if
```

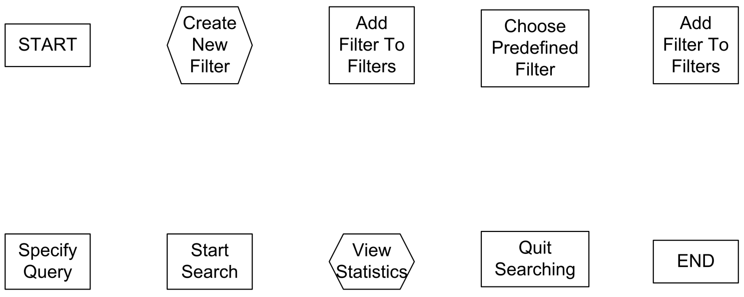


Figure 5.4: Algorithm: Tasks

**Example:** For our example in figure 5.2 we create 8 nodes (2 from START and END) and 2 external nodes. Notice that we are at the second traversal of the task sequence, which means the elementary task Search Query isn't in the task sequence. Also, the application tasks in the task model were never added to the task sequence, so they will not be depicted in the task navigational model as well.

### 5.5.6 Temporal Operators

#### 5.5.6.1 Enabling and Enabling with information exchange

When we come across an enabling temporal operator, we add a link from the node representing the task on its left to the node representing the task on its right. This is the simplest form of temporal operator.

```
if current is TemporalEnabling $\vee$ current is TemporalEnablingInfo
```

```
2:    then drawables.Add(new Link(index − 1, index + 1))
3: end if
```

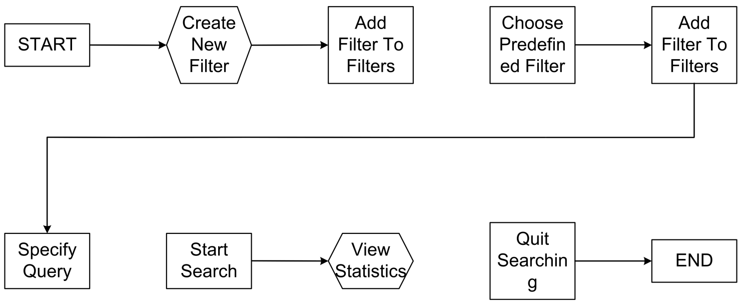


Figure 5.5: Algorithm: Enabling temporal operator

**Example:** We draw 6 links for the 6 enabling temporal operators in the sequence.

### 5.5.6.2 Deactivation

Although deactivation has another meaning than an enabling temporal operator in a task model tree, it will generate into the same thing in the task navigational model, because we cannot represent it otherwise.

```
1: if current is TemporalDeactivation
2:    then drawables.Add(new Link(index − 1, index + 1))
3: end if
```

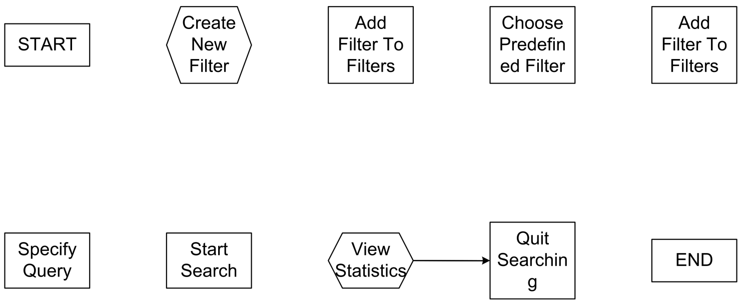


Figure 5.6: Algorithm: Deactivation temporal operator

**Example:** We draw 1 link for the only deactivation temporal operator in the sequence.

### 5.5.6.3 Suspend/Resume

The suspend/resume temporal operator can be displayed on a task navigational model by adding two links to it: one that is generated for the 'suspend' part from the second task to the first, and one that indicates the 'resume' part from the first task to the second.

```
1: if current is TemporalSuspendResume
2:    then drawables.Add(new Link(index − 1, index + 1))
3:         drawables.Add(new Link(index + 1, index − 1))
4: end if
```

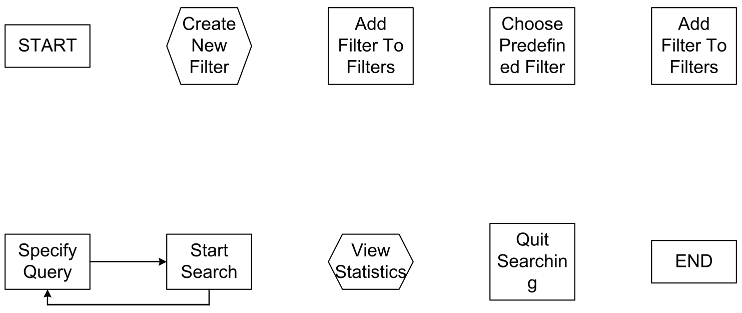


Figure 5.7: Algorithm: Suspend/Resume temporal operator

**Example:** We draw 2 links for the only suspend/resume temporal operator in the sequence. Note that we already had a link going from Start Search to Search Query generated by recursion. The link will end up in the list of *drawables* twice and the double will be removed before drawing them.

#### 5.5.6.4 Choice and Order Independent

Choice and order independency are the most complex temporal operators. They are part of the reason we added the levels to the sequence. Using these levels, we know when we run into *choice chains*. A choice chain is a sequence of tasks that is part of a choice. This sequence consists of tasks of the same level in the task tree that are part of the choice on a lower level (higher in the tree). This signifies that the user can choose between one chain of tasks and other chains of tasks. When this chain is made up by only one task, we have the choice temporal operator in its simplest form.

Using the levels of tasks, we can determine when the choice chains end. From the choice temporal operator we traverse the sequence to the right and to the left. Every task that is higher level (lower in the tree) than this choice operator is part of the choice, and part of the left or right choice chain. The choice chain ends if we find an operator which level is lower or equal to our choice operator (which means at the same level in the tree or higher in the tree than the choice temporal operator).

Once we identified the right and left choice chain, we can connect these choice chains with a *start-choice* task and *end-choice* task. The start-choice task is the task where the user has to make the choice between the two choice chains.

The end-choice task is the task where the user will end up after making a choice between the two choice chains. So we connect the start-choice task with the beginning of each of the choice chains and connect the end of each of the choice chains with the end-choice task.

Order independency is very similar to choice. You can choose what task to begin with, but you will need to do the other tasks of the order independency as well. It can be seen as a choice temporal operator where we add links between the choices, indicating you can go to the second choice when finishing the first one and the other way around. We add a link from the end of the first choice chain to the beginning of the second choice chain, and a link from the end of the second choice chain to the beginning of the first choice chain.

```
if current is TemporalChoice $\vee$ current is TemporalOrderIndependent
  then startOfChain1 $= -1$
       endOfChain1 $=$ index $- 1$
       startOfChain2 $=$ index $+ 1$
       endOfChain2 $= -1$
       startOfChoice $= -1$
       endOfChoice $= -1$

       for i $=$ index to $0$ do
          if taskSequence[i] is TemporalOperator
            then if taskSequence[i].level <= current.level
                   then startOfChain1 $=$ i $+ 1$
                        break
                 end if
          end if
       end for
       for i $=$ index to taskSequence.Length do
          if taskSequence[i] is TemporalOperator
            then if taskSequence[i].level <= current.level
                   then endOfChain2 $=$ i $- 1$
                        break
                 end if
          end if
       end for
       for i $=$ index to $0$
          if taskSequence[i] is TemporalEnabling
             $\vee$ taskSequence[i] is TemporalEnablingInfo
             $\vee$ taskSequence[i] is TemporalSuspendResume
             $\vee$ taskSequence[i] is TemporalDeactivation
            then if taskSequence[i].level <= current.level
                   then startOfChoice $=$ i $- 1$
                        break
                 end if
          end if
       end for
       for i $=$ index to taskSequence.Length do
```

```
37:             **if** taskSequence[i] **is** TemporalEnabling
38:                 $\lor$ taskSequence[i] **is** TemporalEnablingInfo
39:                 $\lor$ taskSequence[i] **is** TemporalSuspendResume
40:                 $\lor$ taskSequence[i] **is** TemporalDeactivation
41:                 **then if** taskSequence[i].level <= current.level
42:                         **then** endOfChoice = i + 1
43:                             **break**
44:                     **end if**
45:             **end if**
46:         **end for**
47:
48:         drawables.Add(**new** Link(startOfChoice, startOfChain1))
49:         drawables.Add(**new** Link(startOfChoice, startOfChain2))
50:         drawables.Add(**new** Link(endOfChain1, endOfChoice))
51:         drawables.Add(**new** Link(endOfChain2, endOfChoice))
52:
53:         **if** current **is** TemporalOrderIndependent
54:           **then** drawables.Add(**new** Link(endOfChain1, startOfChain2))
55:                drawables.Add(**new** Link(endOfChain2, startOfChain1))
56:         **end if**
57: **end if**
```

- **Line 2 to 7:** We initialize a set of variables that will store the tasks that need links between them:
  - *startOfChain1:* The first task of the first choice chain.
  - *endOfChain1:* The last task of the first choice chain. This is the task next to the choice temporal operator we're processing.
  - *startOfChain2:* The first task of the second choice chain. This is the other task next to the choice temporal operator.
  - *endOfChain2:* The last task of the second choice chain.
  - *startOfChoice:* The task where we make the choice between the two chains.
  - *endOfChoice:* The task we go to when the choice has been made.
- **Line 9 to 24:** We find the start of the first choice chain and the end of the second choice chain here. Our chains end when our level gets smaller, which means we go up in the tree.
- **Line 25 to 46:** We find the start and end of the choice. We do this by taking the first non-choice and non-order independent temporal operators, checking their level and taking the task next to that temporal operator.
- **Line 48 to 56:** We add the links to the drawables list. When the temporal operator is order independent, we add two links that connect the choice chains with each other.

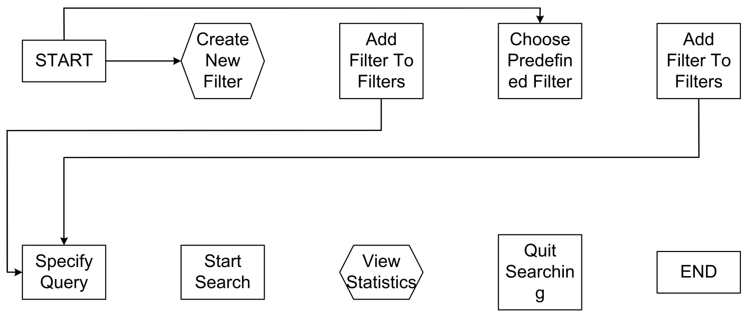


Figure 5.8: Algorithm: Choice temporal operator

**Example:** For our example in figure 5.2 we get 4 links generated by the choice temporal operator. START is the start of the choice and Specify Query is taken as the end of the choice. We have two choice chains as well. The first is Create New Filter - Add Filter To Filters and the second is Choose Predefined Filter - Add Filter To Filters. We do not have to connect the elements of each chain in this step, the temporal operators between them will generate the necessary links to do so.

#### 5.5.6.5 Concurrent and Concurrent with information exchange

Since we are unable to represent concurrency in task navigational models, we process them as if they were order independent temporal operators. We could add new types of links to the task navigational models that can represent concurrency, but this is beyond the scope of this dissertation.

### 5.5.7 Iterative and Optional Behaviour

Code needs reviewing.

```
int startChain = $-1$
int endChain = $-1$

**foreach** optLevel **in** currentShape.optionalList
        **if** optLevel <= currentShape.level
          **for** i = index **to** 0 **do**
              **if** !visitables[i].optionalList.Contains(optLevel)
                **then** startChain = i
                     **break**
              **end if**
          **end for**

          **for** i = index **to** taskSequence.Length **do**
              **if** !visitables[i].optionalList.Contains(optLevel)
                **then** endChain = i
                     **break**
              **end if**
          end for

          drawables.Add(new Link(startChain, endChain))
        end foreach
```

# Chapter 6

# Implementation

## 6.1 Introduction

The primary objective of this dissertation was to provide automatic support for the transformation of task models into navigational models. In chapter 5, we have defined an algorithm for this. In this chapter we describe an implementation of this algorithm.

The implementation of the algorithm is called the WSDM Visio Add-In, and is implemented as an extension to the Visio 2003 environment of Microsoft. Visio 2003 allows us to create diagrams of all sorts using *stencils* and *templates*. A stencil is a collection of shapes (graphical components) that compose a Visio diagram. A template represents a *drawing page* and contains a number of different stencils. On a template, we can draw diagrams by moving shapes from the stencils onto the drawing page (drag and drop). The Add-In software package includes a WSDM template that contains two stencils. It also adds some extra functionality to this template.

A Visio Add-In is a .COM Add-In, a component we add as an extension to Visio. Events coming from Visio are intercepted by this Add-In. An event contains a tag that describes the type of event and a context, that contains the current shape, current page, document and application information of Visio. The Add-In receiving this event will have all the information needed to perform an action based on the tag and context of this event. For example, when the Add-In receives a *getProperties* event from Visio, the Add-In will respond by creating a message box containing the properties of the shape for which the event was generated.

The WSDM Visio Add-In was initiated by Nicolai Roovers [19]. He created the WSDM template and added two stencils to this template to allow users to draw *navigational models* and *information and functional models*. Furthermore, the functionality of this template allows you to generate an OWL (textual) representation of your diagrams.

For our implementation, we took this WSDM Visio Add-In and extended it with more functionality. From the WSDM template, we can now select the functionality to import a task model. This task model, created in an external environment, can be read by the Add-In, parsed, and converted into a navigational model. This navigational model is then drawn automatically on the drawing page of the WSDM template. From this generated navigational model, we can then create an OWL textual representation with Roovers' conversion functionality.

## 6.2 Design

We will first discuss how the WSDM Visio Add-In handles the events it receives from Visio. We will then look at how the Add-In handles the *importTaskModel event*, that allows us to import task models and generate the corresponding navigational models in Visio.

### 6.2.1 Event Handler

The events are handled by two classes, *EventSink* and *EventHandler*. The EventSink class receives all the events that come from Visio. Every event contains a context string, describing the shape, page, document and application itself, as mentioned earlier. There are three events that will get processed by the EventSink and will be handled by the functionality of the WSDM Visio Add-In:

- **setProperties:** When we select a shape and request its properties using its dropdown menu, this type of event will be generated in Visio and received by the EventSink. The appropriate user interface window will be shown for the shape.

- **generateOWL:** When we select the drawing page itself and select the functionality to generate OWL code from the dropdown menu, the EventSink will receive this type of event. This will result in the generation of the OWL code and a save file dialog will be provided to store the OWL file on disk.

- **importTaskModel:** When we select the drawing page itself and select the functionality to import a task model from the dropdown menu, the EventSink will receive this type of event. An open file dialog will appear and the user is asked to select the task model file.

### 6.2.2 XML Traverser

When we receive an *importTaskModel event*, the program will ask for a task model file. This file has to contain an XML representation of a task model. Once we have opened the file, we create a document from the XML code contained in that file. This document will be traversed by the program and for each node of this document we can generate a Task object. This Task object is the same object we defined in section 5.3 on page 59. The XML of a task model looks like this:

```
<?xml version= '1.0'?>
<!DOCTYPE TaskModel PUBLIC "http://giove.cnuce.cnr.it/CTTDTD.dtd" "..\CTTDTD.dtd">

<TaskModel NameTaskModelID="C:\...\Examples\example1.xml">
<Task Identifier="Showtimes and Buy Tickets" Category="Abstraction Task"
      Iterative="false" Optional="false" PartOfCooperation="false"
      Frequency="null">
  <Name> name </Name>
  <Type>   </Type>
  <Description>   </Description>
  <Precondition>   </Precondition>
  <TimePerformance>
   <Max>  </Max>
   <Min>  </Min>
   <Average>  </Average>
  </TimePerformance>
  <Object name="" class="" type="" access_mode="" cardinality="">
  <InputAction Description="null" From="null"/>
  <OutputAction Description="null" To="null"/>
  </Object>
  <Object name="" class="" type="" access_mode="" cardinality="">
  <InputAction Description="null" From="null"/>
  <OutputAction Description="null" To="null"/>
  </Object>
  <Object name="" class="" type="" access_mode="" cardinality="">
  <InputAction Description="null" From="null"/>
  <OutputAction Description="null" To="null"/>
  </Object>
  <Object name="" class="" type="" access_mode="" cardinality="">
  <InputAction Description="null" From="null"/>
  <OutputAction Description="null" To="null"/>
  </Object>
  <Object name="" class="" type="" access_mode="" cardinality="">
  <InputAction Description="null" From="null"/>
  <OutputAction Description="null" To="null"/>
  </Object>
  <SubTask>
    <Task Identifier="Specify Location" Category="Application Task"
          Iterative="false" Optional="false" PartOfCooperation="false"
          Frequency=" ">
      <Name> name </Name>
      <Type>   </Type>
      ...
```

The Task objects are nested in the same way as the nodes of the XML document. The tree structure is maintained and the root Task contains all the other tasks of the task model. We can now traverse this tree of Task objects and create a task sequence out of it.

### 6.2.3 Task Sequence Generator

This class will create a task sequence out of the Task objects tree. How this is done is described in section 5.4 on page 61. This task sequence is stored in a simple List. We add the different kinds of tasks and temporal operators to this list. This List contains Visitables; a Visitable can be either a CttTask or a TemporalOperator. A Visitable corresponds to the TaskModelObject we used in the algorithm in section 5.4 on page 62. These Visitables will then be "visited" by the Process Task Sequence visitor to generate the nodes and links of the navigational model.

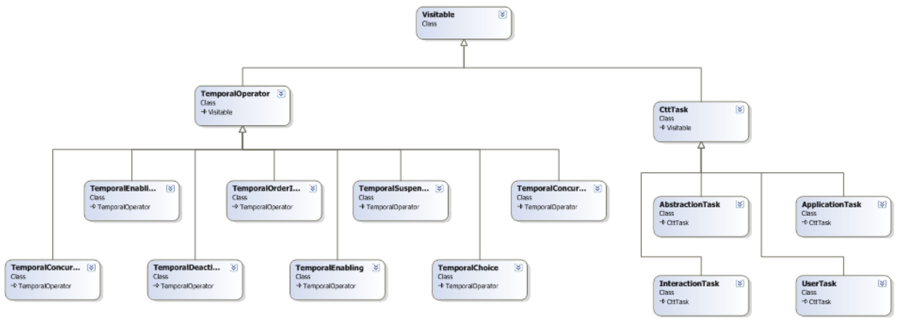


Figure 6.1: Implementation: The visitables

### 6.2.4 Task Sequence Processor

This visitor will create nodes and links and stores these in a List as well. This List stores Drawables, which can either be a Link or a Node. This Drawable stores all the information needed to draw a Link or a Node on the screen. What information was stored and how the task sequence gets traversed by the algorithm is discussed in section 5.5 on page 66. A Drawable is nothing more than a placeholder that stores information for a shape on the drawing page. The actual drawing of the shape is handled in the next class, Shape Placer.

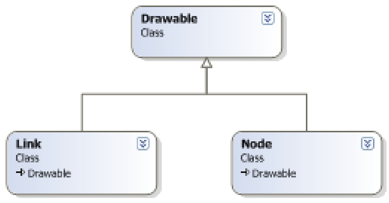


Figure 6.2: Implementation: The drawables

### 6.2.5 Shape Placer

When we created a list of Nodes and Links based on rules of our algorithm in chapter 5, we can start drawing these Nodes and Links. Starting with the nodes first, we draw each drawable on our drawing page of the open document of Visio. Once all the nodes are on the drawing page, we create the links between them, removing any double links that may have been generated by our algorithm.

The resulting navigational model is shown on the drawing page of our WSDM template. The event was handled successfully and the Event Handler can now receive a new event to process. The conversion of a task model that contains 10 to 20 tasks takes about 1 to 2 seconds in real time.

## 6.3 WSDM Visio Add-In Specifications

We now describe on what platform the Add-In can be installed, how the installation is performed and how we can use the Add-In to create a navigational model from a task model. We clarify this with an example and some screenshots.

### 6.3.1 Platform

The program is a self-installing executable (Windows Installer MSI file[1]). The program was developed in Visual Studio 2005 Professional[2] on the Windows XP Professional operating system. The program is a .COM Add-in for Microsoft Office Visio 2003. To run the program, the following software is required:

- Microsoft Windows 2000/XP or later
- Microsoft Office Visio 2003 or later
- Microsoft .NET Framework 2.0 or later
- CTTE version 2.3 or version 1.5.9b

### 6.3.2 Installation

The installation is simple and straightforward.

1. First of all, make sure you meet the requirements.
2. Install the Visio add-in using the MSI installer (the *WSDMVisioAddin.msi* file). When you do not have the latest update of .NET Framework 2.0, the installer might ask to install it first, linking you to the corresponding website. Download and install the update and then install the add-in.

### 6.3.3 Guide

Before we can start with the add-in, we need to create the task models in a separate tool. Once we have made these task models and saved them in the correct format, we can generate the corresponding navigational models from them in the Visio environment. After generating the models, we can modify the models, add object chunks, change link details...

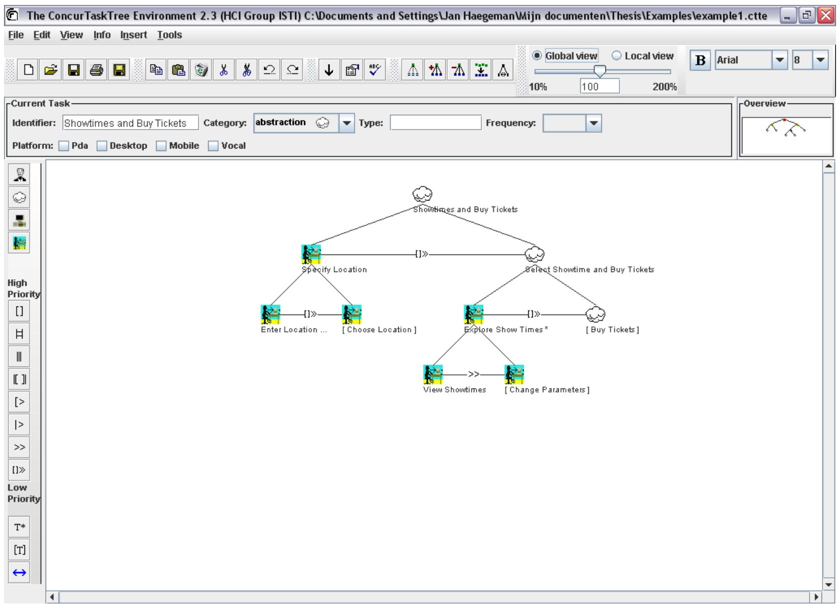
#### 6.3.3.1 Creating the task models

To create task models, we use CTTE[3] [15], which is a program to develop ConcurTaskTrees through a graphical user interface. It is fairly simple to create CTTs using this program.

[1]see `http://support.microsoft.com/kb/893803` for more information

[2]see `http://msdn2.microsoft.com/en-us/vstudio/aa718668.aspx` for more information

[3]the ConcurTaskTrees Environment by Paterno et. al., you can download the needed software at `http://giove.cnuce.cnr.it/ctte.html`



The ConcurTaskTree Environment 2.3 (HCI Group ISTI) C:\Documents and Settings\Jan Haegeman\Mijn documenten\Thesis\Examples\example1.ctte

Current Task

Identifier: Showtimes and Buy Tickets Category: abstraction Type: Frequency:

Platform: Pda Desktop Mobile Vocal

Figure 6.3: The ConcurTaskTree Environment 2.3.

The add-in created, supports two versions of CTTE, version 2.3 and version 1.5.9b. The main reason we did this is because the two versions are very different from each other and we leave it up to the user to decide which version he prefers. Currently, version 2.3 is the latest version of CTTE. It provides the best support for placing, arranging and justifying your task model trees. It also supports automatic enforcing of the CTT rules, which isn't something we want in our case because WSDM uses slightly different rules. When you make the parent of two tasks an *interaction task*, then the two subtasks will automatically change into *interaction tasks* as well. Recall from the chapter on WSDM that we do not want to enforce this rule, but when using version 2.3 we have no choice but to do so. In version 1.5.9b this rule isn't strictly enforced, so you can make CTTs that correspond to the CTTs we would make in WSDM. On the other hand, this earlier version doesn't have all of the fancy features of arranging and justifying task model trees as in version 2.3.

When you have finished making your task model, you need to export it to an XML file, which contains all the details of your task model tree. You can export any task model to an XML file by clicking *File - Save CTT as XML*. Save the XML file, as you will need it to generate the navigational model. Let's take the CTT in figure 6.4 as an example[4] throughout this chapter.

[4]this example was taken from the paper on WSDM by De Troyer et. al. [9]

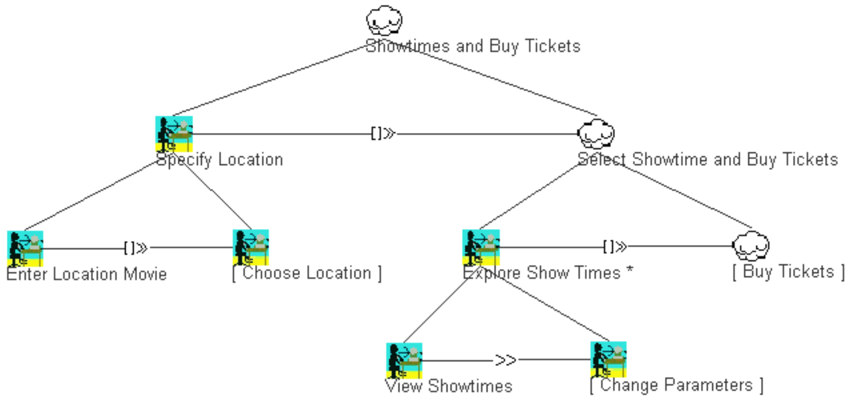


Figure 6.4: CTT: Showtimes and Buy Tickets

In both versions, there is a bug in the CTTE that prevents you from using the "&" character in the generated XML. Task models with this symbol in one or more of their subtasks will generate a faulty XML file. Remember not to use this symbol when you want to generate navigational models from the XML file using the add-in.

#### 6.3.3.2 Generating the navigational models

After installing the WSDM Visio Add-in, you can use it by starting Visio 2003. When in Visio you should be able to select the *Web Semantics Design* template from the category *WSDM* in the *Choose Drawing Type...* menu. The *WSDM* category blends in with the other categories as shown in figure 6.5.

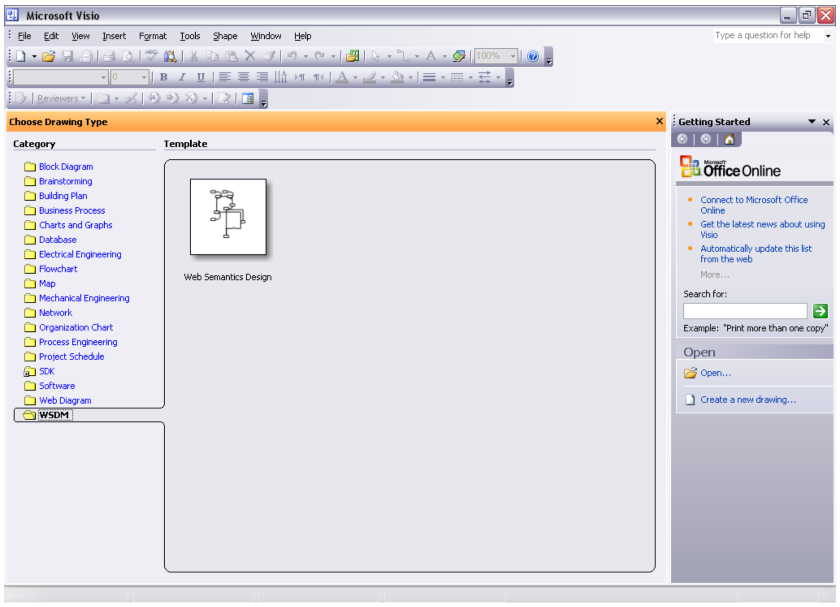


Figure 6.5: Visio's startup screen.

Selecting the template will give you an empty document on which you can add shapes. You can choose between stencils to either create navigational models or object chunk models. These stencils were made by Nicolai Roovers [19] for the WSDM environment. When you want to import a task model and generate a navigational model, you right-click the empty document and select *Import Task Model...* from the dropdown menu as illustrated in figure 6.6.

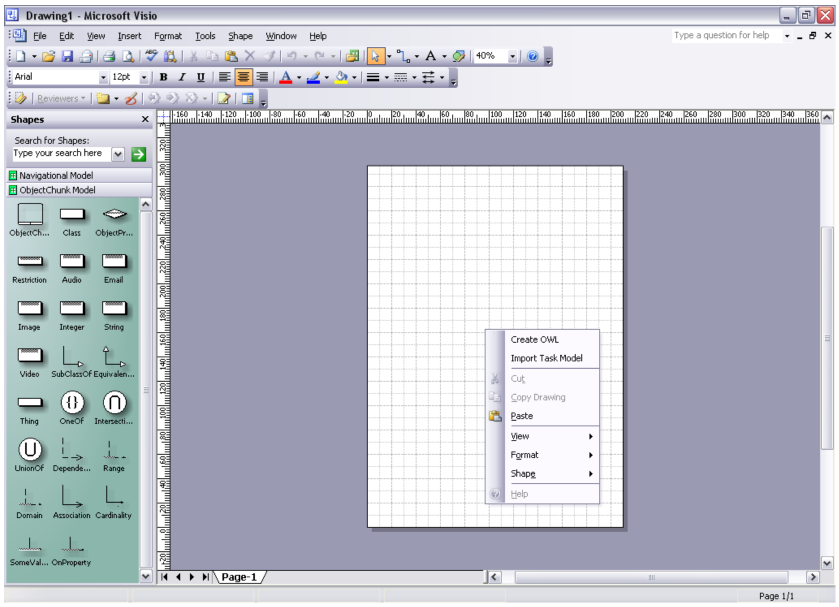


Figure 6.6: Importing a task model.

Selecting this option will give you an open file dialog that asks for an XML file. Select the XML file that represents your task model created in CTTE and the add-in will start to analyze the XML, traverse the task model tree, generate the corresponding navigation structure and place the model on screen in a matter of seconds. After rearranging the nodes and links on the screen manually, we get the navigational model as shown in figure 6.7.

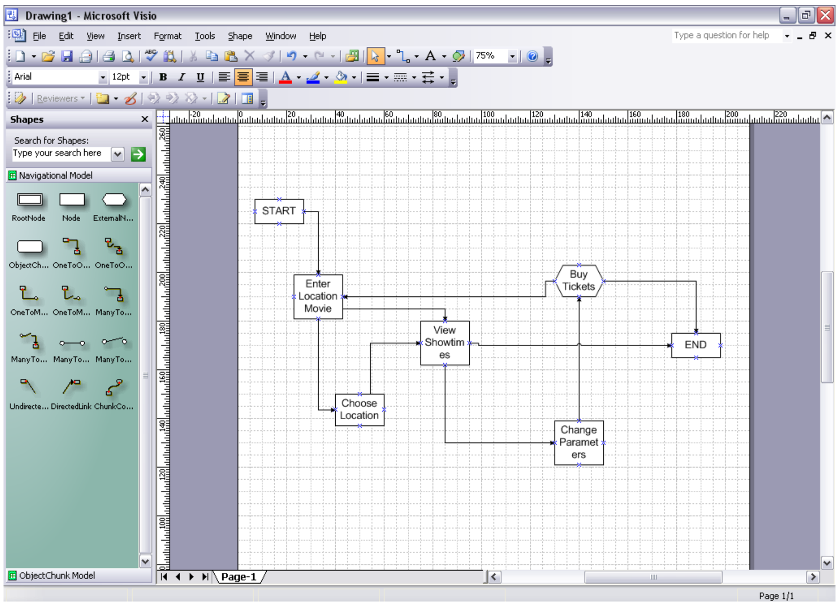


Figure 6.7: A generated navigational model.

In the model we see that every elementary interaction task of the task model was transformed into a node and every elementary abstraction task was transformed into an external node. The proper links are generated for the enabling temporal operators and links for the iteration and optional tasks are added as well. Now it's up to the user to add object chunks to this, change links to conditional links where needed, and add parameters to the links based on the parameters given in the object chunks.

# Chapter 7

# Conclusions

## 7.1 Introduction

To conclude this dissertation, we give a brief summary of the main topics of the background work, the related work and the research that was done. We then give an overview of the achievements and discuss possible future work that is closely related to the subject of this thesis.

The main goal of this thesis was to create a transformation algorithm that can produce navigational models out of task models in the context of WSDM. WSDM, an audience-driven web design method, makes use of these types of models to derive the navigational structure of a web system. Automation is an effective way of making a web design method more attractive to the designer. The implementation of the transformation algorithm is a first step into automating parts of the design methodology of WSDM.

In section 7.2 we give a summary of the chapters of this dissertation. In section 7.3 we list the achievements. Section 7.4 discusses possible future work in the context of the subject of this thesis.

## 7.2 Summary

For the background and related work of this dissertation, we discussed various web design methods. In chapter 2, we introduced WSDM, the Web Semantics Design Method, which was the main context of this thesis. We listed all the steps of the design methodology and examined the task models and navigational models of WSDM in greater detail. In chapter 3, we discussed the Web design methods WebML, OOHDM/SHDM and Hera respectively. These model-driven design methods all show resemblances to WSDM, we gave a brief introduction to each of these.

For the research part of this dissertation, we introduced the notion of task model patterns in chapter 4. We discussed how patterns for task models could be specified and gave some examples. In chapter 5 we described the transformation algorithm we created for the generation of navigational models from task models.

We listed every step of this algorithm and clarified them with examples. In chapter 6, we described an implementation of this transformation algorithm and how this implementation allowed us to perform the transformation in Visio with the WSDM Visio Add-In. Finally, in this chapter we described the results of our work, the achievements and the future work.

## 7.3 Achievements

In this section, we sum up the work achieved during the course of this dissertation:

- **Task model patterns:** We defined the notion of task model patterns, which are a form of design patterns for task models. Task models are a part of WSDM and are created using the CTT task model technique. These task models describe the tasks that allow satisfying the requirements of the target users of a web system. We proposed a method to write down patterns in the task models and created four patterns using this method: confirmation pattern, the validation pattern (login/lougout), container pattern and post message pattern.

- **Transformation algorithm:** We defined an algorithm that enables us to create task navigational models from task models. This algorithm was explained step by step and allowed us to create an extension to the existing WSDM Visio Add-In of Roovers [19] to support this transformation algorithm.

- **Extension to WSDM Visio Add-In:** We took the existing WSDM Visio Add-In and extended it so it would allow the transformation of task models into navigational models. Task models created with the CTT modeling technique can now be transformed automatically into navigational models with the help of the Visio Add-In.

## 7.4 Future work

At present, there is no real WSDM software suite with which a user can create all the models of the WSDM methodology. With the work from Roovers and the work done during the thesis, we are now able to create navigational models, object chunk models and generate navigational models from task models created in CTTE, a graphical environment to create task models. There are a few things we could do in the future:

- **Expanding the WSDM Visio Add-in:** We could create a *WSDM Software Suite* for Visio that includes all the models of WSDM, allows (semi-)automatic transformations between models and full generation of OWL code from these models. Also, the user interface (of the .COM Add-In) could be expanded to allow more kinds of input from the user. For now, the user interface is basic and allows only the most common actions.

- **More tests and evaluation:** The algorithm was tested in basic circumstances. To be certain that the algorithm yields correct results in all cases, more testing and evaluation is needed. Task models come in a great variety, there might be special cases that were overlooked during the design of the algorithm.

- **Addition of new task model patterns:** Now that we formulated a way of writing down task model patterns, we could add new task patterns to the list. The more task model patterns we create, the more likely a designer will be able to use one of them. This can only have a positive effect on the development of task models in WSDM.

# Bibliography

[1] A. Bongio, S. Ceri, P. Fraternali, and A. Maurino. Modeling Data Entry and Operations in WebML. In *Lecture Notes in Computer Science*, volume 1997, pages 201–208, 2001.

[2] S. Casteleyn and O. De Troyer. Structuring Web Sites using Audience Class Hierarchies. In *Conceptual Modeling for New Information Systems Technologies, ER 2001 Workshops, HUMACS, DASWIS, ECOMO and DAMA, Lecture Notes in Computer Science*, volume 2465. Spinger-Verlag, 2001.

[3] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. In *Computer Networks (Amsterdam, Netherlands: 1999)*, volume 33, pages 137–157, 2000.

[4] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Elsevier Science, 2003. ISBN: 1-55860-843-5.

[5] S. de Moura and D. Schwabe. Interface Development for Hypermedia Applications in the Semantic Web. In *LA-WEBMEDIA '04: Proceedings of the WebMedia & LA-Web 2004 Joint Conference 10th Brazilian Symposium on Multimedia and the Web 2nd Latin American Web Congress*, pages 106–113. IEEE Computer Society, 2004. ISBN: 0-7695-2237-8.

[6] O. De Troyer. Audience-driven web design. In *Information modelling in the new millennium*. IDEA GroupPublishing, 2001. ISBN: 1-898289-77-2.

[7] O. De Troyer and S. Casteleyn. The Conference Review System with WSDM. In *First International Workshop on Web-Oriented Software Technology, IWWOST01*, 2001.

[8] O. De Troyer and S. Casteleyn. Modeling Complex Processes for Web Applications using WSDM. In *Proceedings of the Third International Workshop on Web-Oriented Software Technologies, IWWOST2003*, 2003.

[9] O. De Troyer, S. Casteleyn, and P. Plessers. WSDM: Web Semantics Design Method. In ..., 2007.

[10] O. De Troyer and C. Leune. WSDM: A User-Centered Design Method for Web Sites. In *Computer Networks and ISDN Systems, Proceedings of the 7th International World Wide Web Conference*, pages 85–94. Elsevier Science, 1998.

[11] S. Espana, J. Panach, I Pederiva, and O. Pastor. Towards a Holistic Conceptual Modelling-Based Software Development Process. In *Conceptual Modeling - ER 2006*, pages 437–450, Heidelberg, 2006. Springer-Verlag.

[12] F. Frasincar, P. Barna, G. Houben, and Fiala Z. Adaptation and Reuse in Designing Web Information Systems. In *International Conference on Information Technology: Coding and Computing (ITCC'04)*, pages 387–391. IEEE Computer Society, 2004.

[13] F. Frasincar and G. Houben. Hypermedia Presentation Adaptation on the Semantic Web. In *Adaptive Hypermedia and Adaptive Web-Based Systems, Second International Conference, AH 2002, Lecture Notes in Computer Science*, volume 2437, pages 133–142. Spinger-Verlag, 2002. ISBN: 3-540-43737-1.

[14] F. Frasincar, G. Houben, and Vdovjak R. Specification Framework for Engineering Adaptive Web Applications. In *The Eleventh International World Wide Web Conference, Web Engineering Track (CDROM Proceedings)*, 2002.

[15] G. Mori, F. Paterno, and C. Santoro. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. In *IEEE Transactions on Software Engineering*, volume 28, 2002.

[16] F. Paterno. *Model-based design and evaluation of interactive applications.* Springer-Verlag, 2000. ISBN: 1-85233-155-0.

[17] F. Paterno, C. Mancini, and Meniconi S. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In S. Howard, J. Hammond, and G. Lindgaard, editors, *Human-Computer Interaction*, pages 362–369. Chapman and Hall, 1997.

[18] P. Plessers, S. Casteleyn, Y. Yesilada, O. De Troyer, R. Stevens, S. Harper, and C. Goble. Accessibility: A Web Engineering Approach. In *Proceedings of the 14th International World Wide Web Conference, WWW2005*, pages 353–362. ACM Press, 2005. ISBN: 1-59593-046-9.

[19] N. Roovers. WSDM: OWL Generatie voor Grafisch Model. WISE internal apprenticeship report, Vrije Universiteit Brussel, 2006.

[20] D. Schwabe and G. Rossi. The Object-Oriented Hypermedia Design Model. *Commun. ACM*, 38(8):45–46, 1995.

[21] D. Schwabe and G. Rossi. An Object Oriented Approach to Web-Based Applications Design. *Theory and Practice of Object Systems*, 4(4):207–225, 1998.

[22] D. Schwabe, G. Szundy, S. de Moura, and F. Lima. Design and Implementation of Semantic Web Applications. In *WWW 2004 Workshop on Application Design, Development and Implementation Issues in the Semantic Web*, 2004.

[23] M. Van Welie. Integrated Representations for Task Modeling. In *10th European Conference on Cognitive Ergonomics (ECCE'10)*, Linkoping, Sweden, 2000.

[24] M. Van Welie, G. Van der Veer, and A. Eliens. Patterns as Tools for User Interface Design. In *Workshop on Tools for Working With Guidelines*, Biarritz, France, 2000.

[25] R. Vdovjak, F. Frasincar, G. Houben, and P. Barna. Engineering Semantic Web Information Systems in Hera. *Journal of Web Engineering*, 2(12):3–26, 2003.