INSTITUT FÜR INFORMATIK

DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Diplomarbeit

MagicDraw-Plugin zur Modellierung und Generierung von Web-Anwendungen

Petar Blagoev

blagoev@cip.ifi.lmu.de

Aufgabensteller: Prof. Dr. Alexander Knapp
Betreuer: Dr. Nora Koch

Abgabetermin: 9.11.2007

Acknowledgments:

I would like to sincerely thank Prof. Dr. Alexander Knapp for coaching me and supporting me in writing this diploma thesis.
I would also like to express a special gratitude to my supervisor Mrs. Dr. Nora Koch for her help and support.
Finally, a very special thank-you goes to Petya, for her continuous support throughout the writing of my thesis.

Zusammenfassung:

In dieser Arbeit wurde ein UWE Plug-in (eine Erweiterung) für das MagicDraw Program entwickelt. Dieses Plug-in implementiert die UWE Methode, die einen systematischen Ansatz für die Entwicklung von Webanwendungen darstellt.

UWE beruht auf einer konservativen Erweiterung des UML und umfasst das getrennte Modellieren der Konzept-, Navigation- und Präsentationsschicht einer Webanwendung. Dieses Plug-in ist für die MagicDraw Software mittels der bereitgestellten Open API Schnittstelle implementiert worden.

MagicDraw integriert völlig das UML2 Metamodel und ermöglicht es, seine grundlegenden modellierenden Funktionalitäten mit Hilfe von Drittentwicklern zu erweitern. Außerdem es ist ein Modellierungs- und CASE-Werkzeug für die Visualisierung von UML Diagrammen, das viele verschiedene Entwicklungs-Mechanismen für zahlreiche objektorientierte Programmiersprachen, Modellierung von Datenbank Schemata, rückwärts Entwicklung etc. ermöglicht.

Die Implementierung von einem UWE Plug-in für ein so populäres Modellierungs- und Entwicklungswerkezeug wie MagicDraw bietet einen reibungslosen Einstieg für alle, die sich mit dem UWE-Ansatz beschäftigen möchten.

Abstract:

In this work an UWE plug-in (an extension) was developed for the MagicDraw software. This plug-in implements the UWE methodology which provides a systematic approach for the development of Web applications.

UWE is based on a conservative extension of the UML and comprises of the separate modelling of the conceptual, navigational and presentational aspect of Web applications. This plug-in is implemented for the MagicDraw software using its Open API developing interface.

MagicDraw fully integrates the UML2 metamodel and gives a great opportunity to enhance its basic modelling functionalities with third party plug-ins. Furthermore it is a visual UML modelling and CASE tool and provides versatile development mechanisms for different kinds of Object Oriented programming languages, databases schema modelling, reverse engineering facilities and more.

Implementing a plug-in for such a popular modelling and developing tool as MagicDraw, makes it very easy for everyone, who is interested in using the UWE modelling approach, to work with it.

„Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.“

München, den 9.11.2007

Petar Blagoev

Contents

Table Index

Figure Index

1 Introduction

Web applications are getting more important in our daily use of information technologies than ever before. Starting with static Web pages in the past, through dynamic presentation of the page content afterwards and nowadays the attempt to use Web 2.0 [1] techniques everywhere in the Internet had transformed the Web into a platform for more and more complex and popular Web applications. Because of the rapidly growing new Web techniques and the complexity of the Web applications, design and modelling tools supporting the Web applications engineering are in demand.

The objective of this thesis is to develop a Computer-Aided Software Engineering (CASE) tool supporting the UML-based Web Engineering (UWE) [2] methodology. The Web engineering field is rich in design methods supporting the task of designing Web applications. One of the usability requirements to such methods is to provide tool support for the model-driven design and generation of Web applications.[3] The well known standard used for modelling is the Unified Modelling Language.[@1]

The design methodology of UWE is based on a metamodel that is defined as a lightweight extension of the UML metamodel in form of a profile. Furthermore UWE tends to use of standards in the systematic design followed by a semi-automatic generation of Web applications. The developed UWE CASE tool in this work implies the employment of the UWE methodology.

The UWE CASE tool developed in this thesis was built as an extension of MagicDraw CASE tool.[@5] The main advantage of using an already existing CASE tool is the fact that such a tool supports already existing modelling standards and some of the Web engineering modelling methods. Besides that, MagicDraw provides a UML profile support which gives the opportunity to map the UWE metamodel into such a profile and easy integrate it into the modelling process. The developed tool has extended MagicDraw to support the UWE methodology. The tool provides tailored visual editors for an UWE model for modelling of Web applications. Furthermore it supports semi-automated transformations that are defined in the UWE development process. The developed tool is implemented as a Plug-in for MagicDraw and it fully integrates the UWE metamode l[4].

The UWE approach is already supported by ArgoUWE [3] CASE tool which is an extension of the open source ArgoUML [@6] modelling tool. Because ArgoUWE is supporting only UML version 1.5 the goal was to develop a new UWE CASE tool to support the new UML version 2.0 and above. That's way the modelling tool used in our case as basis had to satisfy this requirement. Another point for implementing a new UWE CASE tool is to take advantage on the use of an already established modelling tool with great functionalities for modelling and designing of Web and Object Orientated applications. The support of different interchange formats such as XMI [@7] was also from importance. We chose the MagicDraw for CASE tool to be extended, because it fulfils all specified criteria, providing an easy way to define a UWE profile and integrating an extension as a Plug-in through provided Open API interface.

The structure of this thesis is the following: chapter two provides an overview of the UWE metamodel and approach and how UWE methodology supports the Web application development demonstrated by a small example. Chapter three describes the UWE CASE tool relevant requirements. Chapters four to six are the core of this work and describe the design decisions that have been made during the implementation, the implementation of the tool itself pointing on interesting problems and solutions, and a step by step modelling example using the newly developed tool. Finally, in the last chapter seven some concluding marks and the future work is outlined.

2 UWE

The UML-based Web Engineering (UWE) is a software engineering approach for the development of Web applications that has been continuously extended since 1999.[5][6] UWE supports the development of Web applications with special focus on systematization.[7]

UWE is a methodology for the development of Web systems. The two key aspects that distinguish UWE from other approaches are its reliance on standards and its support by an open source tool [@8].

UWE offers Web-domain specific aids for the three pillars of software engineering:

- visual notation
- process
- tool support

Furthermore UWE is defined as a model-driven development (MDD) [8] process, i.e. models are not built in isolation, but they are rather the basis for both the model-to-model and model-to-code transformations. UWE provides tool support for the design of models, model consistency checks and semi-automatic generation of Web systems.[9]

Thus the UWE components, such as: metamodel definition, profile definition, transformations definitions, constraints definitions and more, are defined in many previous works (see UWE home page for publications [@2]) and are beyond the scope of this thesis. Summarized overview of them will be given in the following sections of this chapter.

2.1 UWE Overview

While UML is the standard of specifying, modelling, visualizing and designing any kind of Objects Oriented applications there is no such a standard for the modelling of Web applications. UML and other established modelling languages do not support such kind of modelling and UWE is trying to fill this gap. UWE defines such missing elements in UML as: menu, index, image, text field, navigation paths between different Web sites, etc.

The UWE approach provides a domain specific development process, a notation and a tool support for the engineering of Web applications. The notation proposed for the analysis and the design of Web applications is defined as a "lightweight" extension of the UML- as a so-called profile.

The UWE profile is based on the extension mechanisms defined by the UML itself. The advantage of using established notation is obvious as Web applications can be designed using existing UML CASE tools and the extension has no impact on the

interchange formats.

UWE uses “pure” UML notation and UML diagram types for the analysis and the design of Web applications whenever possible, i.e. without extensions of any type. For the Web specific features, such as nodes and links of the hypertext structure, the UWE profile includes stereotypes, tagged values and constraints defined for the modelling elements.

The UWE extension of UML covers content, navigation, and presentation and Web process aspects. For each aspect a diagram is built following the guidelines provided by an UWE method for the systematic construction of models. For example, a navigation model consists of navigation classes, links and a set of indexes, guided tours and queries.

The UWE design approach for Web business processes consists of the introductions of specific process classes that are part of a separate process model with a defined interface to the navigation model. In order to model adaptive features of Web applications in a non-invasive way, UWE uses techniques of aspect-oriented modelling (AOM) [@3]. Following the separation of concerns principle UWE proposes to build an adaptive model for personalized or context-dependant systems and weave the models afterwards. [13]

2.2 UWE Metamodel

Metamodelling is the core of a model-driven process and plays a fundamental role in the CASE tool construction. A metamodel is a precise definition of the elements of a modelling language, their relationships and well-formedness rules needed for creating syntactically correct models. Metamodels are essential for the definition of model transformations and semi-automatic code generation.

The UWE Metamodel is designed as a conservative extension of the UML 2.0 metamodel. Thus none of the modelling elements of the UML metamodel are modified. All new UWE modelling elements are related by inheritance to at least one modelling element of the UML metamodel. Additional features and relationships for those new elements are defined in the UWE metamodel. Analogously to the well-formedness rules in the UML, OCL constraints are used in UWE to specify the additional static semantic of the newly defined elements.

Furthermore UWE is compatible with the MOF (Meta Object Facility) interchange metamodel, which can take an advantage of using metamodelling tools based on the corresponding XML interchange format XMI (XML Metadata Interchange).

The resulting UWE metamodel could be mapped to a UML profile, so called profileable. In this way standard UML CASE tools with a support for UML profiles or the UML extension mechanism, i.e. stereotypes, tagged values and OCL constraints, can be used to create UWE models of a Web application. If such CASE tools were designed to be further extended, the UWE metamodel can be integrated to them by creating a UWE profile.

2.2.1 UWE Metamodel Package Structure

The UWE extension of the UML metamodel is created by adding two top-level packages *Core* and *Adaptivity* to the UML as shown in Figure 1. The separation of concerns of Web applications is reflected by the package structure of *Core*. The adaptation cross-cutting is reflected by the packages dependency of *Adaptivity* on *Core*. [9] The *Core* package contains all elements needed to model non-adaptive Web

applications with UWE- such as: a Content*, a Navigation, a Presentation* and a *Process* model. At least one type of UML diagrams is proposed by UWE for the visualization of each model.

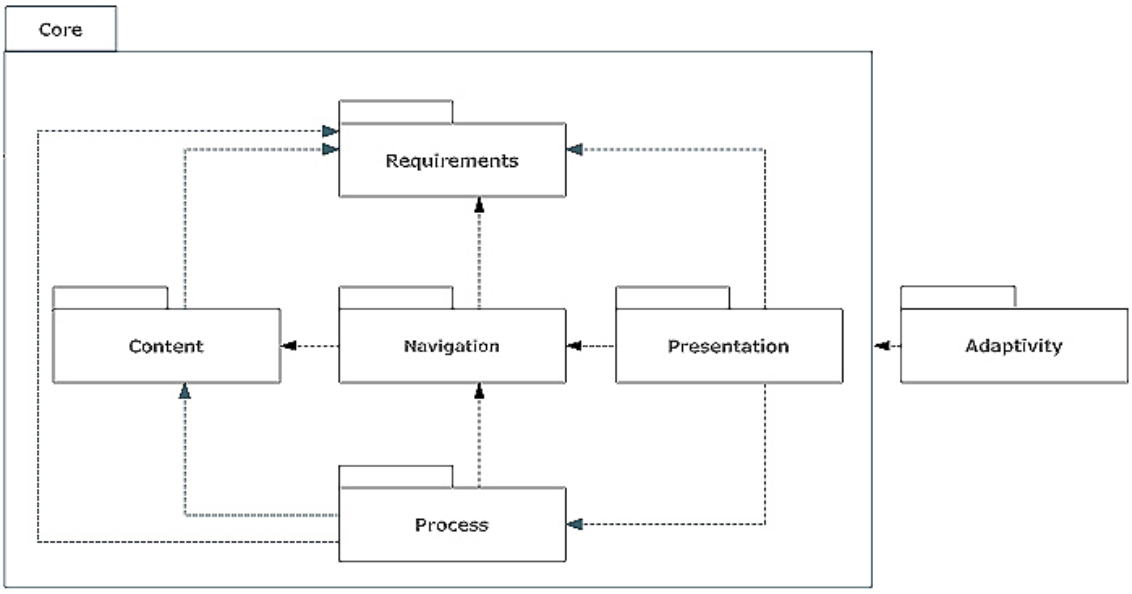

Figure 1 UWE Metamodel Overview

All UWE models: *Content, Navigation, Presentation*, etc. are subclasses of the UML *Core* class *Model*.

Figure 2 shows the UWE metamodel for *Navigation* with a relationship between the abstract classes *Node* and *Link*, which are the basic elements in the *Navigation* model. Their subclasses are *NavigationClass* and *NavigationLink* respectively. Because the *NavigationClass* is a *Node* it can be directly reached from all other nodes of the application with the *isLandmark* attribute. Furthermore, the *NavigationClass* consists of *NavigationProperties* (derived from the UML *Core* element *Property*).
Figure 2 also illustrates how access primitive classes, such as *Index*, are aggregated to a *Node*. Note that *Menu* is a specialization of the *NavigationClass*.

The *Presentation* package of the UWE metamodel can be described analogously to the *Navigation* package. Figure 3 shows the UWE metamodel of *Presentation*. Obviously, the *PresentationClass* is a specialization of the abstract *PresentationElement* class. All owned class attributes by a *PresentationClass* element are *PresentationProperties.*

The *PresentationGroup* and *Page are* Subclasses of the *PresentatonClass*. The user interface element abstract class *UIElement* is specialized through a plenty of UI interaction elements like *Button, TextInput, Image, Form*, etc. A closer detail of the UWE metamodel's UI elements are shown in Figure 4. Besides that, the last figure shows also that *Form* and *AnchoredCollecten* classes are specializations of the *UIContainer* which can have one or more user interface elements like *TextInput* or *Image*.

As we can, see the Presentation package of the UWE metamodel specifies all required user interface unique elements of a Web application.

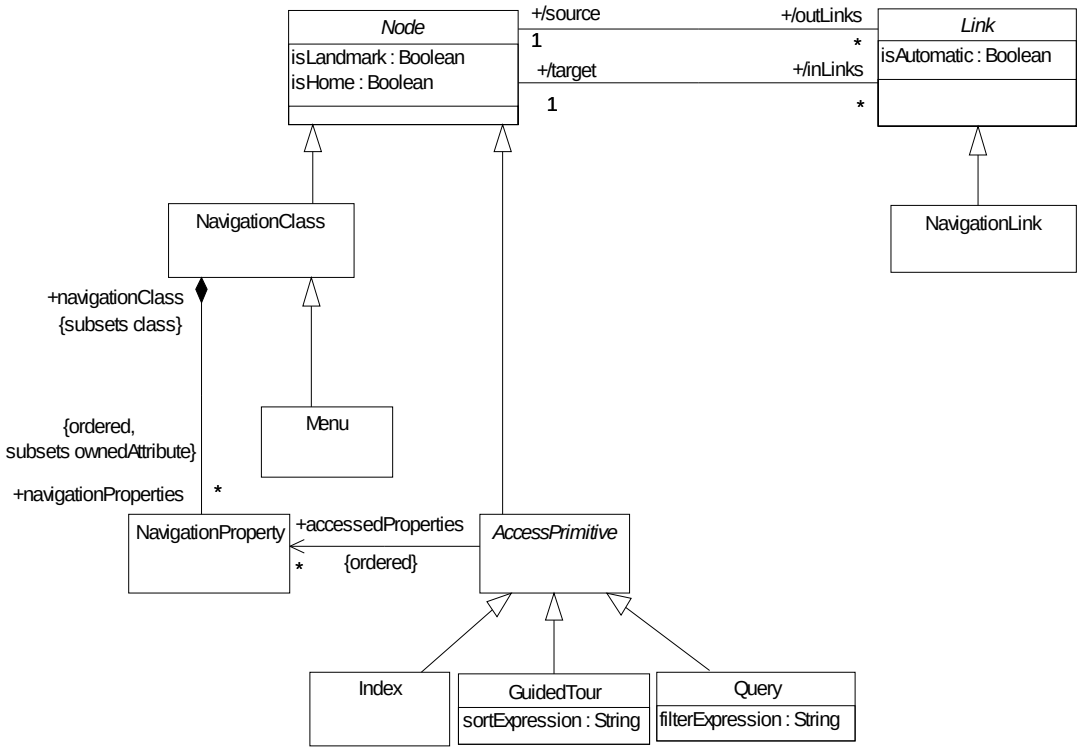

Figure 2 UWE Navigation Metamodel

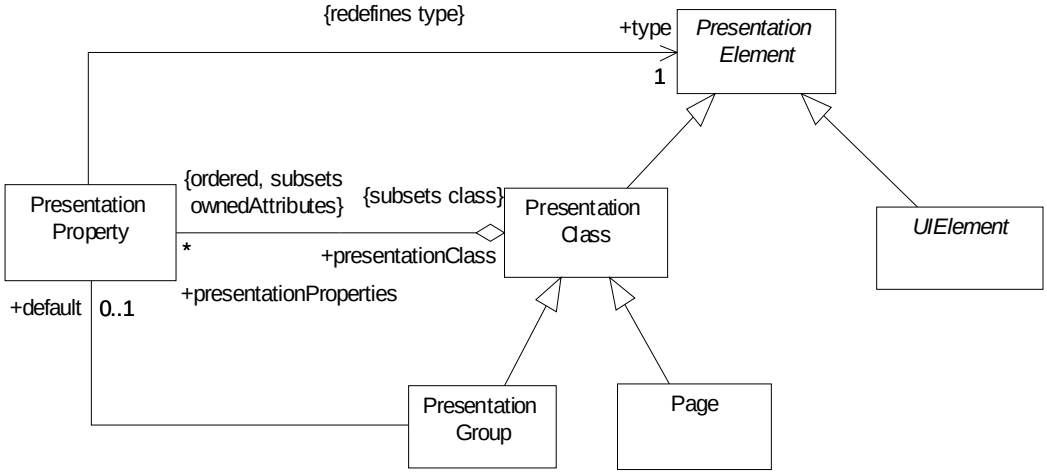

Figure 3 UWE Presentation Metamodel

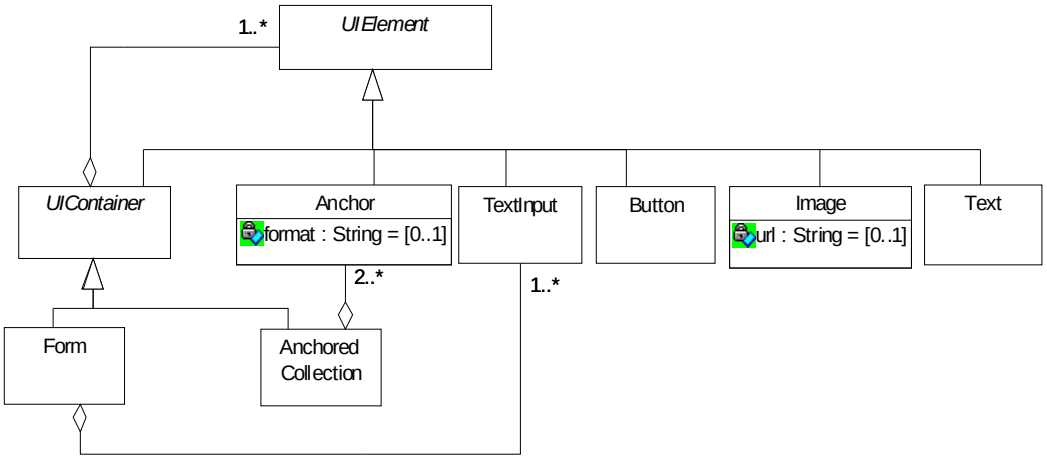

Figure 4 UWE UI Elements Metamodel

The association between *Navigation* and *Presentation* modelling elements can be understood by the diagram shown in Figure 5. These relationships specified in the previous diagram are needed later on for enabling semi-automated model transformations, in particular from *Navigation* diagram into *Presentation* diagram. More about transformation will be explained in the following chapters.

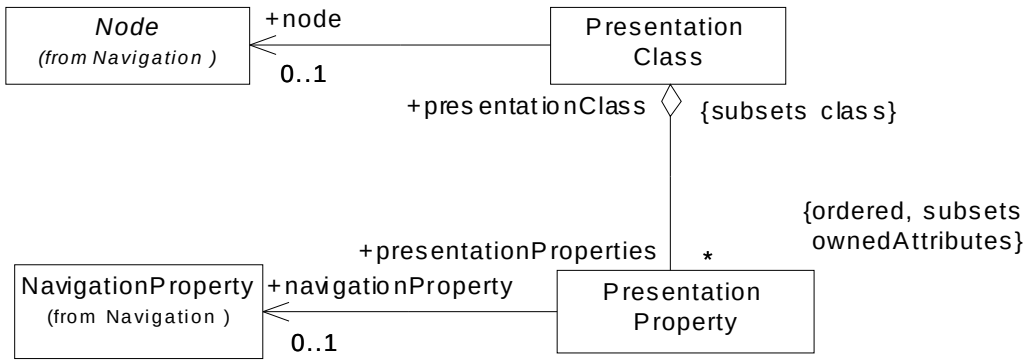

Figure 5 Relationships of Presentation Elements

There are two more UWE metamodel packages, used by now as a stub, reflecting the fact that the modeller can use all UML features while designing their models.

The first one is the *Content* package. The model of which can contain all UML base elements whereas no further specialisation of those elements is needed as shown in Figure 6.

The second and the last package of the UWE metamodel is the *Process*. This package has two important specialisations of the *Node* and *Link* classes from the *Navigation* package. These are the *ProcessCalss* and *ProcessLink*. Again all own attributes of *ProcessClass* elements are *ProcessProperties*. Figure 7 details the *Process* package.

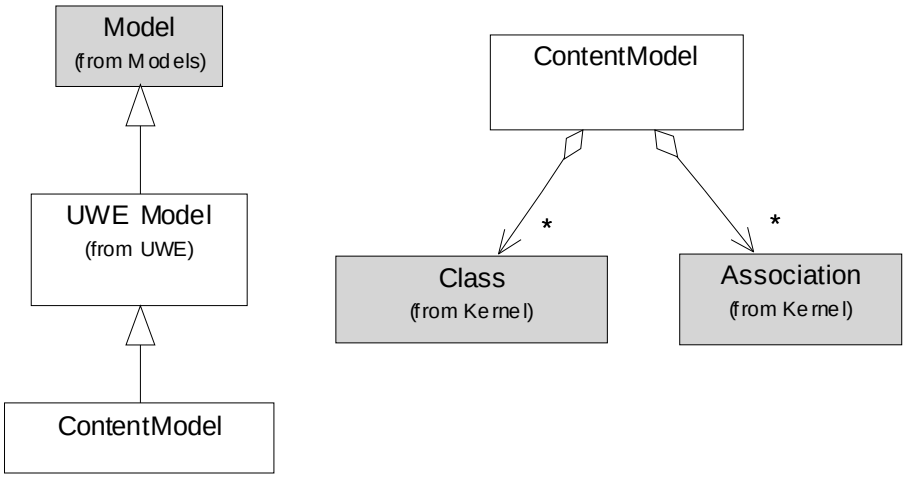

Figure 6 UWE Content Metamodel

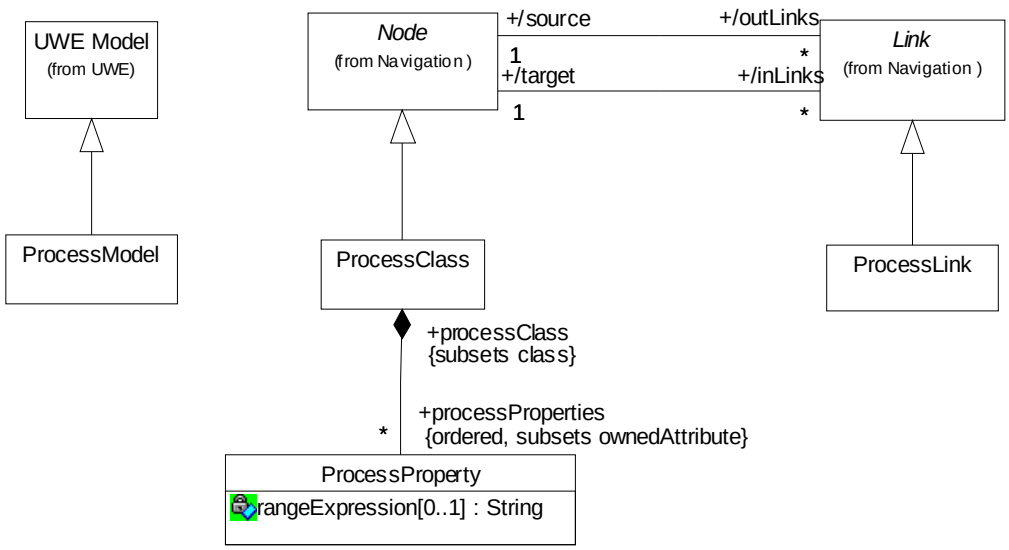

Figure 7 UWE Process Metamodel

2.2.2 Consistency Rules

Following the UML, UWE uses OCL to state more precisely the static semantics of UWE's new metamodel elements as well as the dependencies of metamodel elements both within a single metamodel package and between packages.

The following example shown in Figure 8 is example of the OCL constraints that are a part of the UWE metamodel. This constraint is specifying stereotype *Dependency* for the association between *AccessPrimiteve* and *NavigationProperty* (see Figure 2) and the association between *NavigationClass* and *Menu*. Were the end of the relationship are denote true client and supplier. [9]

```
context: Dependency
inv: self.stereotypes ->
        includes("Primitive2Property") implies
    (self.client.stereotypes ->
            includes("AccessPrimitive") and
     self.supplier.stereotypes ->
            includes("NavigationProperty"))
```

Figure 8 Example of OCL Constraint

2.3 UWE Profile

The UWE metamodel is the basis for the UWE profile. The UML profile of UWE is using UML extending mechanisms, called light weighted profile. Basically these UML mechanisms can define new profiles by using a custom stereotypes, tagged values, constraints for specializing UML and associations. Never the less such a profile can be standardized by the Object Management Group (OMG).[@4]

The UWE profile is used for modeling of Web applications. In UWE, generally model elements, especially Classes and Associations, are extended in e.g. via stereotyping. Moreover, UWE profile is separated into following stereotypes groups: *Content*, *Navigation*, *Presentation*, *Process* and *Web Requirements Engineering (WebRE)*. These groups are corresponding to the UWE metamodel package structure.

To map the UWE metamodel to UWE profile the following systematic rules were applied: [10]

- classes to elements with stereotypes
- attributes to tagged values
- inheritance to inheritance among stereotypes repeated
  mapping of attributes and associations
- associations to tagged values or associations (for classifiers)

2.4 UWE Development Process

The process of software engineering and in particular the development of Web applications is being continuously optimized in coherency of ever changing technology and user requirements. That's why models designed in any phase of the development process have to be easily adaptable to these requirements. UWE implements a Model-Driven Development (MDD) process on the basis of the separation of concerns in the early phases of development process. This UWE development process is based on the construction of models and model transformations.

MDD approve the use of models for every phase of Web engineering and even more, it stresses on the need of transformations in each of these phases of development process. UWE development process is driven by the separate modeling of concerns describing the different views of the same Web application during this process. The different concerns can be: content, navigation, structure, and presentation and their models are built at different phases (requirements engineering, analysis, design,

implementation) of the development process.[9]

The UWE development process can be easily described as: model-transformation-model-transformation-code, whereas the amount of models can be various.

In the following section some of the main model views of the development process will be shown on a case study.

2.5 UWE by Case Study

The following case study describes briefly Web development with the UWE approach. Whereas the models of design concerns of different views of an e-shop are presented. Only some of the basis diagram views are shown here, for whole case study model please refer to the following work reference [10].

In the UWE analysing process the first what the modeller should concern are the functionality requirements of the Web application. These Web application functionalities are described using a UML use case diagrams. In the Figure 9 two actors are shown: non registered user and registered (customer) user of the e-shop Web application. UWE distinguishes between normal and navigation use case. Navigation use cases are used when modelling typical user behaviour when interacting with a Web application, such as browsing through the application content, searching information by keywords, etc.

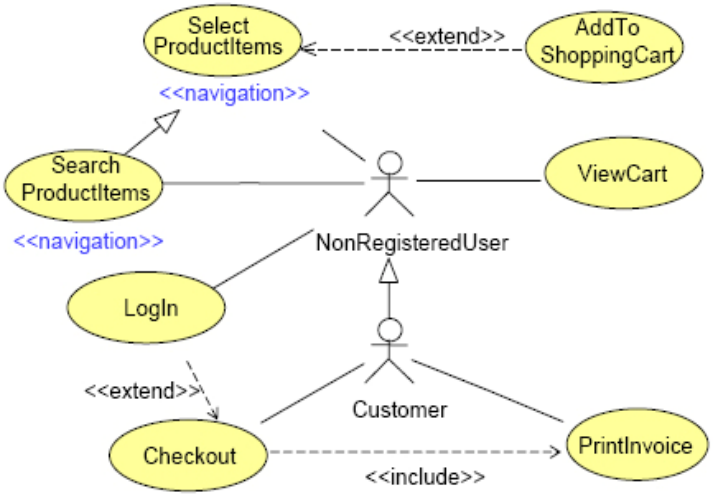

Figure 9 Navigation Use Case Diagram

The next step of the modelling process could be refining of the requirements done in the use case diagram before. Especially all business logic processes should be refined using activity diagrams. In particular UML activity diagrams are used for visual presentation of workflows. For example such workflows could be registering of new user, buying a product, etc. In general the level of refinement of the requirement specifications depends on the application project complexity and risk.[9]

After specifying the requirements and refining it with business process flows the next step is to create the content model. The main goal there is to model the problem domain and to separate content from navigation (hypertext) structure and presentation. Furthermore a UML class diagrams are used for the structure and UML sequence diagrams or state charts are used for the behaviour. Figure 10 shows such a structure content diagram, whereas classes are used to represent units of textual information and multimedia elements. Associations and aggregations are used to show

relationships between classes and inheritances are used to specify their hierarchies.

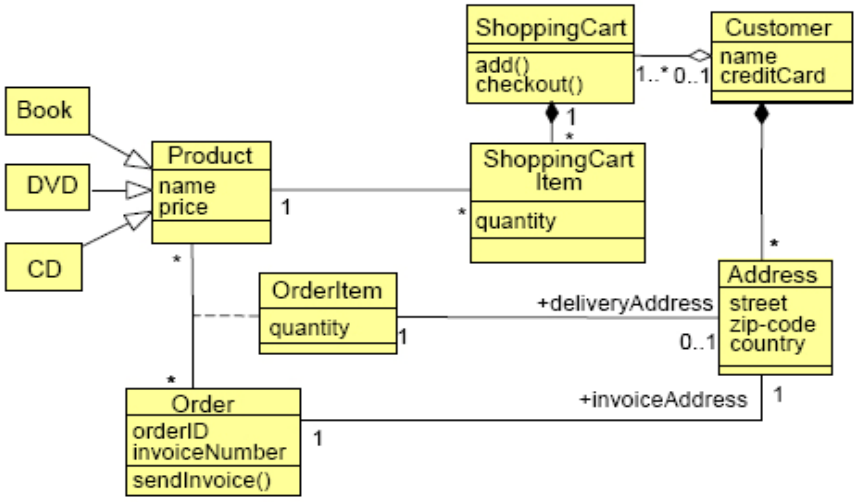

Figure 10 Content Diagram – Structure

To represent nodes and links of the hypertext structure of the Web application a Navigation model has to be created. Furthermore navigation path are also presented into this type of UWE diagram. The goal there is to avoid disorientation and cognitive overload.

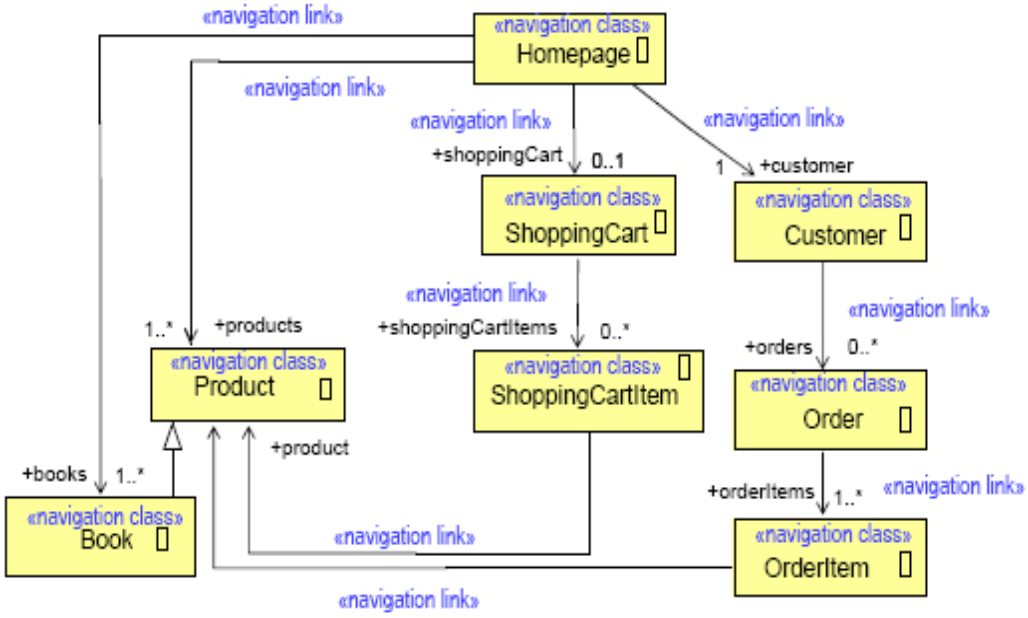

Figure 11 Navigation Diagram

Figure 11 shows such a Navigation diagram. Once again also for the navigation model class diagrams are used. Furthermore there are *Navigation Classes*, which specify the hypertext nodes. These nodes could be visited by the user through browsing and should become the same name as their mapped node. Association between navigation classes are *Navigation Links*. Such a link specifies that the target navigation (node) object is accessed by navigation from the source navigation object.

Navigation classes should be enhanced with additional navigation elements through the modelling process. In Figure 12 the same navigation diagram is enhanced by the following access primitives: *Menus* (MainMenu, AccountInfo), *Queries* (SearchProducts), *Indexes* (BookRcommendationm, SelectedResults, OrderList, etc.), *and Quided Tours* (not included).

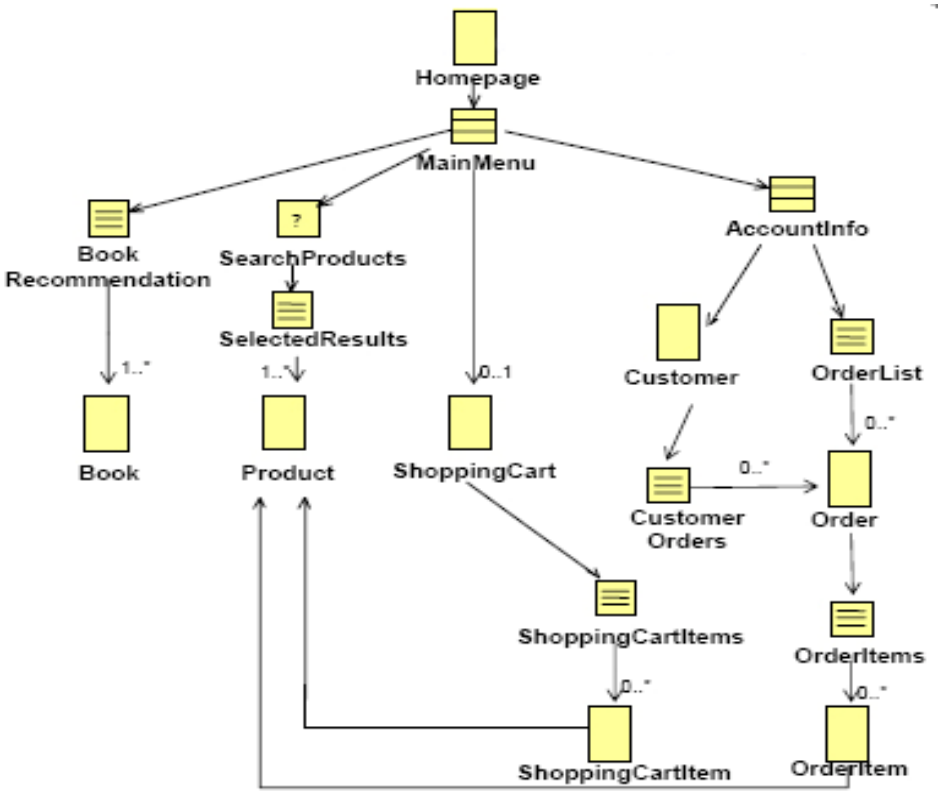

Figure 12 Navigation Diagram - Enhanced by Access Structures

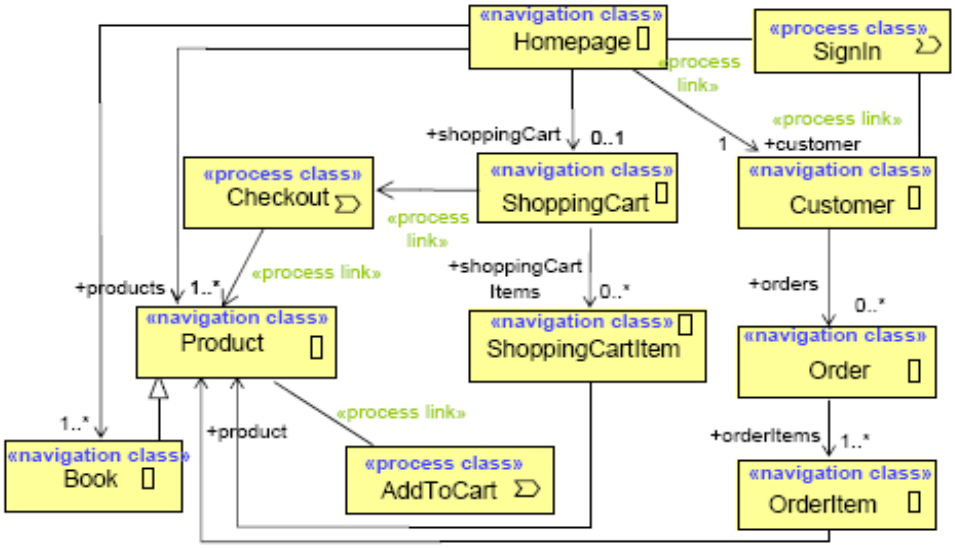

Figure 13 Process Diagram

Next step in the UWE modelling process is to model the process of the workflow driven Web application. First step at this place is to define the process classes. These classes are all non-navigation use cases from the first diagram. After that the process structure model should be constructed. Then all process classes should be integrated into the navigation structure model. Besides that UML activity diagrams should be used for the description of the process flow and object flow of the application. Figure 13 shows the process classes integration into the navigation diagram. The navigation diagram is at this place without the access primitives defined above. This figure above shows also process classes, such as*: SignIn, CheckOut, AddToCart*, etc. which were integrated into the existing navigation diagram. Process classes are such type of classes which instances are needed by the user during process execution. In turn the process links are associations between navigation classes and process classes. Furthermore they indicate the entry and exit points of the process within the navigation structure.

At the end of the UWE modelling process a presentation of the structure and the behaviour of the user interface should be modelled. This is done by creating of presentation diagrams, as shown in Figure 14. Furthermore pages as a hierarchical composition of presentation elements have to be defined. Again class diagrams (in UML container notation) are used for the structure of the presentation and sequence diagrams for the behaviour. Further presentation elements are defined and used in the presentation model. For instance the presentation group *Page* contains all elements that will be presented together on the screen as response to one request. In turn Presentation class consists of set of interface elements which are representing the logic unit of the presentation. All other user interface elements such as: *Anchor, Text, Image, Button*; are also defined in the presentation model.

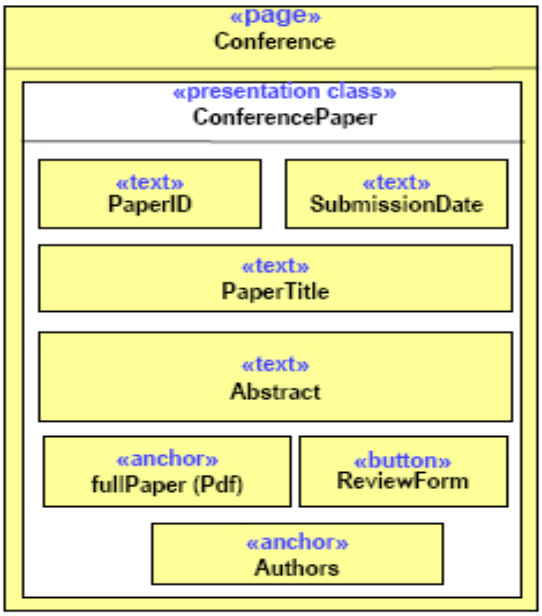

Figure 14 Presentation Diagram

3 UWE CASE Tool Requirements

As mentioned in the beginning of this diploma thesis new Computer-Aided Software Engineering (CASE) tools for supporting the Web development process are still being developed or enhanced to satisfy the new Web techniques and overall requirements. But there is no support of modelling tools yet, like for example for the standard modelling language UML. It is so primarily, because there is no a widely established modelling language for Web engineering like for example UML for Object Oriented development.

CASE tools have to be built on a precisely specified metamodel of the modelling construct used in the design activities.[4] These tools have to provide sufficient flexibility in case modelling requirements change in the future. Tool-supported design and model-based system generation has become essential in the development process of Web systems due to the need of rapid production of new Web presences and Web applications.

UWE is trying to address Web software developers and designers through its build-up models over UML. Furthermore UWE uses at least one UML diagram for every model view of the Web application and there are no new visualization elements defined. Further it means also, that established UML techniques can be used in the UWE modelling process.

The distinguishing feature of the UWE is its UML compliance since the model elements of the UWE are defined in terms of a UML profile and as an extension of the UML metamodel. [14] In particular UWE uses the following UML diagram types:

- use case diagrams
- class diagrams
- state and activity diagrams

Based on the UWE profile mapping definitions specified in the previous chapter UWE uses also the following UML modelling elements:

- stereotypes
- tagged values
- classes
- associations

From the list above it is obvious that the UWE CASE tool will have to cover all these UML modelling functionalities. Furthermore the tool has to support also the UWE development process. In particular it means besides the diagram support also the model to model transformations, model to code transformation, semi-automated elements insertion, etc have to be supported.

Moreover the newly created UWE CASE tool has to be in compliance with the already

existing UWE CASE tools. Such an existing tool is e.g. ArgoUWE [11]. ArgoUWE already supports the UWE methodology and was designed as an extension of the open source CASE tool ArgoUML. ArgoUWE is compatible only with UML version 1.5. However it has been used as a comparing tool during the development of the new UWE CASE tool.

Based on the conclusions made previously, a UWE CASE tool should build up upon UML CASE tool functionalities, such as modelling of use case diagrams, class diagrams, using of stereotypes, etc. Besides these “base” UML modelling functionalities, all further UWE-specific modelling techniques will be newly integrated.

Based on these conclusions the easiest way to develop a new UWE CASE tool is to use an already existing UML CASE tool and to enhance it. Some important aspects have to be considered while selecting the UML tool to be enhanced. In particular such a tool has to support the UML profiling so that a UWE profile can be created and integrated into the tool. It is also important to consider how such a tool can be extended as well as whether, the tool provides a good documentation and powerful plug-in support through an Open API. From importance are also already integrated tool functionalities that may be used in UWE.

The UWE extension of such a tool will have to build upon and enhance its UML modelling techniques to integrate UWE. Normally the CASE tool extension mechanism is based on the UML extension mechanism, which is used to define stereotypes that are utilized for the representation of Web constructs such as nodes and links. In addition, tag definitions and constraints written in OCL (Object Constraint Language) can be integrated.

3.1 Usability Requirements

In order to design a good and usable modelling tool, we need criteria for determining what usable means and we have to prioritize which of those criteria are essential for the UWE CASE tool. Some general important usability criteria are:

- flexibility and efficiency of use
- learn ability
- model generation
- (semi-) automated mechanisms
- consistency and standard
- user control and freedom
- aesthetic and minimalist design
- help and documentation

In following those criteria will be discussed which are most important for the UWE CASE tool usability.

We shall start with the highly important requirements of good usable software: flexibility and efficiency of use.

As the UWE CASE tool builds on UML functionalities and techniques provided by the hosting tool, the main aim is to make the modelling of Web processes easer and at the same time more efficient. There is a substantial risk potential to make the modelling tool too complex and difficult to use, when implementing new functionalities. Therefore the selection of the hosting UML CASE tool determines how the UWE plug-in will be implemented to achieve flexibility and high efficiency during the modelling process. In

general the UWE extension has to merge with the hosting tool and shall be intuitive to use.

Another requirement is the learn ability of the UWE CASE tool. Of course this is again connected with the fact that the developed tool enhances an already existing modelling tool. Besides that the UWE CASE tool has to be designed in such a manner that it is easy to learn and work with.

The standards and consistency requirements are of importance as the UWE CASE tool has to support UML and the UWE metamodel in the form of a UWE profile. The developed tool shall be able to check the created UWE models for consistency with the defined rules (those can be OCL rules).

The model (and also the code) generation and semi-automated mechanisms are crucial making the CASE tool usable. Not only because the model generation is a part of the UWE development process (see MDD in section 2.4), but also because of the goal to rise the efficiency during the modelling process. A semi-automated mechanism can be for instance an insertion of elements into a UWE diagram.

Finally, the last three criteria listed above denote that the user has to be able to control every step of the modelling process and shall, have the freedom to design any kind of diagrams. The use shall also be well informed about which modelling functionalities the CASE tool supports.

All these criteria and requirements are in a smaller or greater extent for creating a good usable UWE CASE tool.

3.2 GUI requirements

The Graphical User Interface (GUI) of a UML CASE tool consists of various diagrams, UML presentation elements, dialogs, frames and windows, message boxes, different menus, toolbars, etc. All these visualization building blocks are generally components of modelling tools and most of them will be used when creating a new UWE CASE tool. Since UWE enhances the UML metamodel without specifying new visualization elements no special modelling GUI elements have to be created.

UWE models consist of UML diagrams (see the beginning of this chapter) containing UWE modelling elements. These elements are in turn UML classes or associations which can have a specified UWE stereotype. Because the UWE CASE tool in this case extends an existing UML modelling tool, all UML visualization elements are part of the software, otherwise will have to be created and integrated into the tool.

The new GUI components of the UWE CASE tool or in the case of this thesis extension (plug-in) are simply several types of menus and menu items or buttons representing UWE modelling actions. Through these menus and actions the user interacts with the UWE CASE tool. Such a menu items could be an action to execute a transformation, to create a new UWE diagram, to draw a UWE element, to insert a UWE element, and etc. Therefore it is important to decide where the menus (or buttons) are placed, how the actions are grouped, and what types of menus are used. All these design decisions will be discussed in chapter 4.

Interaction between the user and the modelling tool can be achieved also through dialogs. Such a dialog can appear when the tool wants to signalize an error, or just to inform the user after an action was fired. The UWE CASE tool should provide such dialogs with meaningful error or message declaration where necessary.

In this case, when a UWE CASE tool is developed as a plug-in enhancing an already existing software, it might be necessary to create own windows or frames containing

such an information (elements, files, etc.), which cannot be accomplish through the Open API of the hosting tool.

All further UWE CASE tool GUI requirements should be the same as any another CASE tool.

3.3 Functionality requirements

The UWE CASE tool has to fulfil all modelling functionalities of the UWE approach. As stated in the chapters before, UWE extends the UML 2.0 metamodel, so UWE CASE tool shall support this version of UML. Furthermore the tool should have a UML general graph editing framework, as well as featuring extendable module (profile) architecture. Other fundamental UML modelling functionalities have to be part of the UWE modelling tool.

Since the tool developed for this diploma thesis is not a standalone software, both the hosting tool and the UWE extension (plug-in) will be considered. When combining both of them, the whole domain of UWE CASE tool functionality requirements should be achieved.

First of all we have to be able to specify a UWE profile (module) and afterwards to load it in any UWE project. UWE profile is a UML lightweight extension of UML and it's an important part of the UWE CASE tool, as it specifies all UWE elements needed for further modelling of the Web applications. For more details of what kind of elements UWE provides, please refer to section 2.2 where the UWE metamodel is described.

In general there are the following main functionality groups in the UWE development process:

- Creating and modelling of UWE diagrams.
- Transformations
- Semi-automatic insertion of elements
- Verification of the model trueness
- Code generation

We shall take a closer look at which further functionalities are included in each of those groups.

Each UWE CASE tool has to be able to create UWE models. Normally an UWE model consists of minimum one UML diagram, as stated in chapter two. The user of the tool shall be able to draw *UWE Use Case*, *Content*, *and Navigation*, *Presentation*, and *Process* diagrams during the development process. The user shall also be able to create (UWE) elements into these diagrams. It shall also be possible to assign stereotypes to those elements, in particularly stereotypes defined trough the UWE profile mentioned above. The user shall be able also to manipulate and modify these elements and diagrams. It shall be possible to create (UWE) associations between the classes. Moreover all UML diagram and element functionalities shall be a part of the UWE CASE tool. In our case these UML modelling functionalities are provided by the hosting UML CASE tool.

Beyond the modelling functionalities provided by the hosting tool a default assignment of stereotypes shall be possible. In particular, UWE stereotypes shall be assigned to classes and associations depending on the diagram type and other specified rules.

The next functionality group comprises the transformations, which can be divided into:

- transforming Content diagram to Navigation diagram
- transforming Navigation diagram to Presentation diagram

The main difference between these two transformations is the type of elements that will be transformed during the transformation process. This depends on the source diagram type, the stereotype of the explored elements from the source diagram, and the type of the executed transformation. These diagram transformations are strictly defined by the UWE metamodel.[12] It is necessary to logically examine, which elements (classes and associations) with defined stereotypes from one diagram type can be transformed into new elements in the new diagram type with a proper stereotype. However, these transformations are a semi-automatic help feature provided by the UWE CASE tool and the user will have to complete or to adjust the newly created diagram.

Next functionality group is the insertion of some UWE elements (UML classes with assigned UWE stereotype). This feature is also a UWE specific functionality during the UWE development process so it has to be provided in the CASE tool. The user shall be able to insert UWE elements, *Query* or *Index,* between two other elements considering UWE metamodel rules.

The last group of functionalities is the UWE model consistency check. By default the UWE CASE tool shall allow the user to draw and create any type of diagrams with diverse types of elements in each of them. During the application modelling process inaccuracy may occur - either unwished or on purpose. The system shall be able to automatically check for such model inaccuracies after the modelling process is finalised, or on action taken from the user. To find out what the modelling inaccuracies in designed models are, UWE model consistency check functionality in the UEW CASE tool is required.

Finally the last functionality of the UWE CASE tool shall be the automatic code generation which is also a part of the UWE development process. This feature is corresponding with the previous function, the model consistency check. Only if a model is consistent, can an automated code generation be executed. The UWE CASE tool shall support code generation to Java (e.g. Java Server Pages) or other developing language for Web applications. [13]

4 Design Decisions for the UWE CASE Tool

This chapter provides an overview of what design decisions were made during the implementation of the UWE CASE tool. As already stated, the tool developed during this work is an extension (plug-in) of the already existing UML CASE tool. At the beginning of this chapter an overview of the general strategy for designing a GUI of a plug-in will be given. Later concrete GUI decisions that were made will be described based on examples.

To integrate a plug-in into existing software, some design decisions on where to place the GUI elements are required. These GUI elements have to match as good as possible the already existing GUI elements structure of the hosting system such as: Menu, submenu, toolbar menu, context menu, buttons, and etc. If we would just place one extra custom menu for our extension to collect all menu actions needed for the plug-in, the intuition that the user has learned while using the hosting software would be lost for the newly integrated plug-in. Furthermore the modeller would be slowed down in the modelling process when using combined features of the CASE tool including the new plug-in functionalities.

Before starting with the analysis, designing and developing a new UWE CASE tool as an extension, an already existing UML CASE tool has to be selected. UML has been a fundamental part of the software development process since many years, so there is a big variety of CASE tools supporting UML and in particular UML 2.0. The use of the UML CASE tool MagicDraw as a hosting software was predetermined in this work, so there was no need to specify criteria and requirements for selecting a UML CASE tool.

Based on the requirements and conclusions made in chapter three, we know that the UWE CASE tool in general does not have any specific visualization components. Furthermore the UWE functions can be achieved through GUI elements such as menus, buttons, dialogs, etc. Because the GUI of software should be independent of the used technology, platform and implementing language, at this point it is not important how exactly the hosting tool is implemented. The only matter that is essential at this point is the GUI design, meaning the “look and feel” of the hosting tool. After analysing the hosting tool design, the UWE GUI components can be designed whereas the requirements specified in the previous chapter are considered. Besides that, the UWE plug-in should be effectively integrated into the GUI of the hosting tool.

In the following sections a brief overview of the hosting tool will be given and some important examples of UWE plug-in GUI design decisions will be outlined.

4.1 MagicDraw

This section provides a brief overview of the MagicDraw software and its featured functionalities.

Some basic facts about MagicDraw shall be outlined first. MagicDraw is a product of

No Magic Company, which was founded in July 1995 in Lithuania. The first version of MagicDraw UML was released in July 1998. Since then MagicDraw has won several prestigious software awards. For further information of the No Magic Company and history of MagicDraw software please visit the following link [@10].

MagicDraw is a visual UML modelling and CASE tool with teamwork support. This tool supports many modelling and developing standards, such as UML, support for J2EE, C#, C++ and other techniques, as well as database scheme modelling and reverse engineering facilities.[@9] This tool can be used on different platforms like Windows, Linux, or MacOS.

MagicDraw can be used as a modelling tool for Object Oriented (OO) languages and databases based on the UML. Its UML diagram creating, editing and manipulating tools are easy to learn and intuitive to use. It also provides a plug-in interface through an OpenAPI for third party software and profiling features for the integration of new UML profiles and modules (see previous chapters on UWE profile). This plug-in interface allows adding new menus, buttons and other elements into the GUI of the software. Besides that, new functionalities can be implemented as needed for the plug-in and integrated into MagicDraw. More about its OpenAPI will be discussed in section 5.2.

MagicDraw is being continuously developed and upgraded. The version used in this work is 12.0.

4.2 Design Decisions

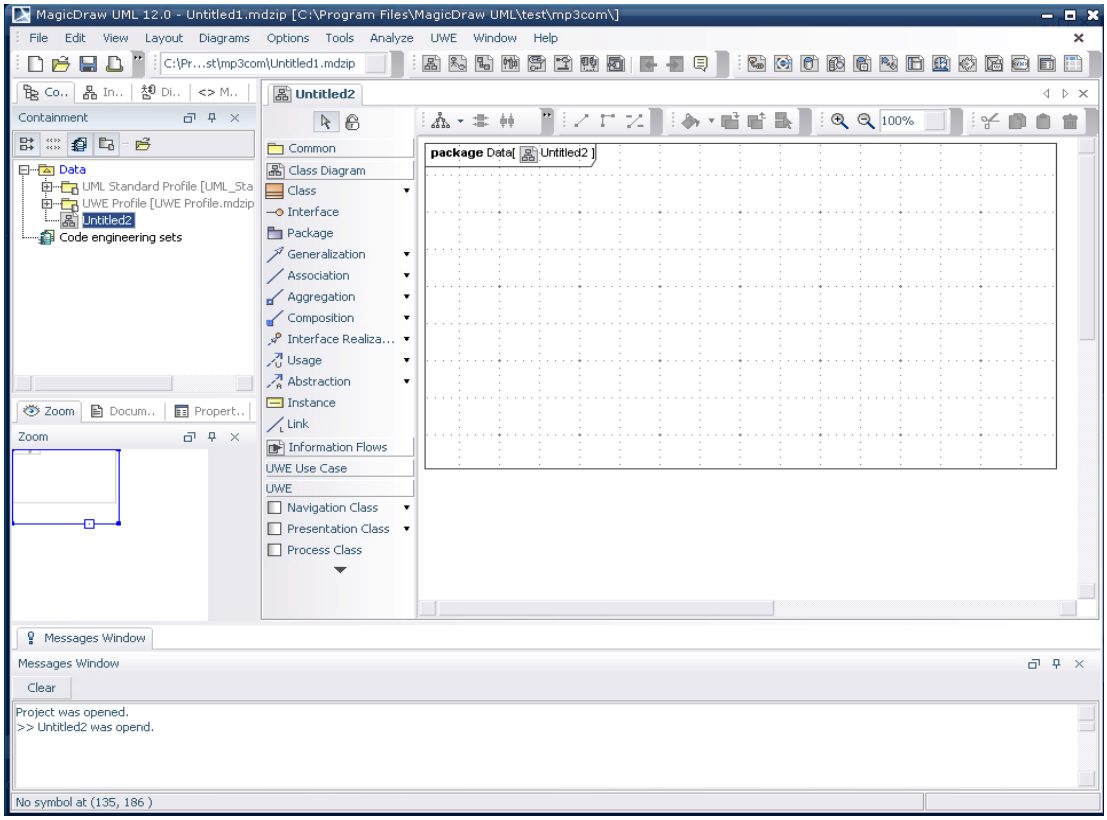

Figure 15 MagicDraw Overview

This section describes the most important design decisions that were made during the development process of the UWE CASE tool in the form of a plug-in for MagicDaraw UML. The Plug-in is designed for MagicDraw version 12.0 and above. To be able to

implement a usable UWE extension into the GUI of MagicDraw first we had to get familiar with the structure and design of its existing GUI.

The main window of MagicDaraw is separated into main menu, main toolbar, side elements package browser and custom diagram toolbar. The main menu consists of submenus and menu items. The main toolbar consist of buttons and a view dropdowns. The diagram toolbar depends on the diagram type, so it can contain different kinds of toolbar actions (menu items or buttons) for each type of a UML diagram. There is a message output window on the bottom of the main window. Actually the “look and feel” of the program can be customized according to the preferences of the user. The user can specify what kind of menu toolbar actions shall be displayed, what kind of diagram actions, etc. For further information on how to do this, please refer to the MagicDraw manual included in every installation of the product.

Figure 15 shows an overview of MagicDraw. All GUI parts discussed above can be viewed. The main menu is on the top, below it there is the main tool bar, on the left hand side - the containment browser and in the middle - the diagram editing window with the diagram tool bar on its left. Finally the messages window is on the bottom.

4.2.1 UWE Profile and Template

MagicDraw normally works with so called profiles. In those profiles all metamodel elements and stereotypes used for modelling are specified. The default MagicDraw profile is the UML Standard Profile. This profile is automatically loaded to every kind of project that the user can create. UML Standard Profile contains stereotypes that are necessary for working with various parts of MagicDraw, primitive data types, and constraints, and UML 2.0 metamodel elements. The following data types are specified in Magic Draw: Boolean, byte, char, date, double, float, int, Integer, long, short, void, and String.

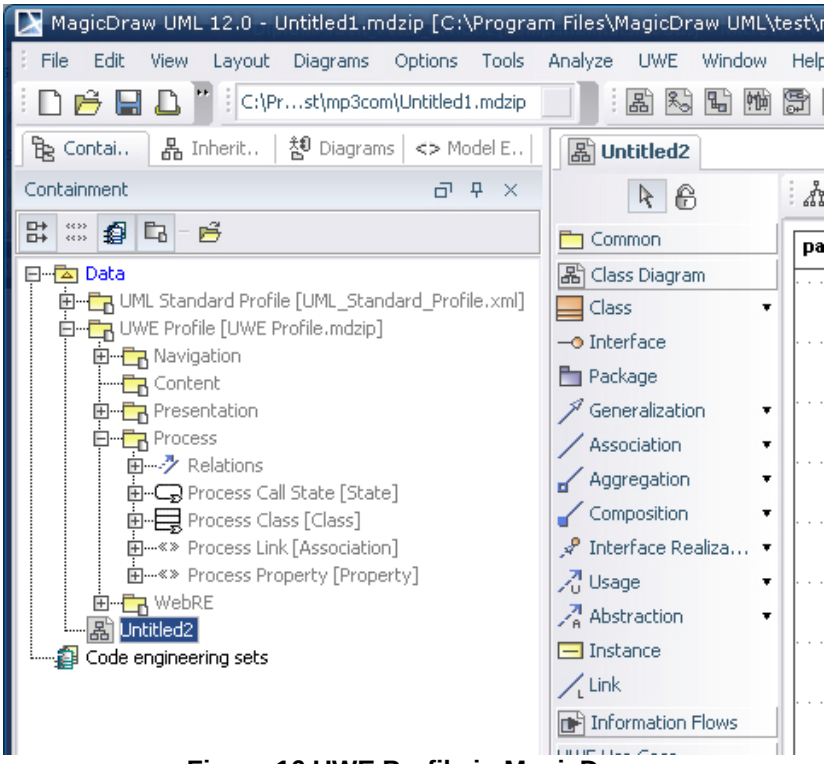

Figure 16 UWE Profile in MagicDraw

To integrate the UWE metamodel through mapped stereotypes into a modelling project, a UWE Profile has been created. Like the Standard UML Profile, all the necessary UWE stereotypes are defined in this profile. The UWE Profile is divided into five profile packages: *Content, Navigation, Presentation, Process, WebRe*. Each of these packages contains the stereotypes and relations defined in the UWE metamodel (see chapter 2.2). For example, Navigation contains the following stereotypes: *Access Primitive, External Ling, Navigation Class, Guided Tour, External Node*, etc. Figure 16 shows the crated UWE Profile and its structure with the packages outlined above. The UWE profile is by default visible in the containment browser on the left hand side, when loaded.

4.2.2 UWE Main Menu

After starting MagicDraw and before creating a new project, only the main menu and toolbar are visible to the user. There is also a UWE menu item in the main menu called *UWE*. As you can see in Figure 17 this menu item is between the *Analyze* and *Window* main menus.

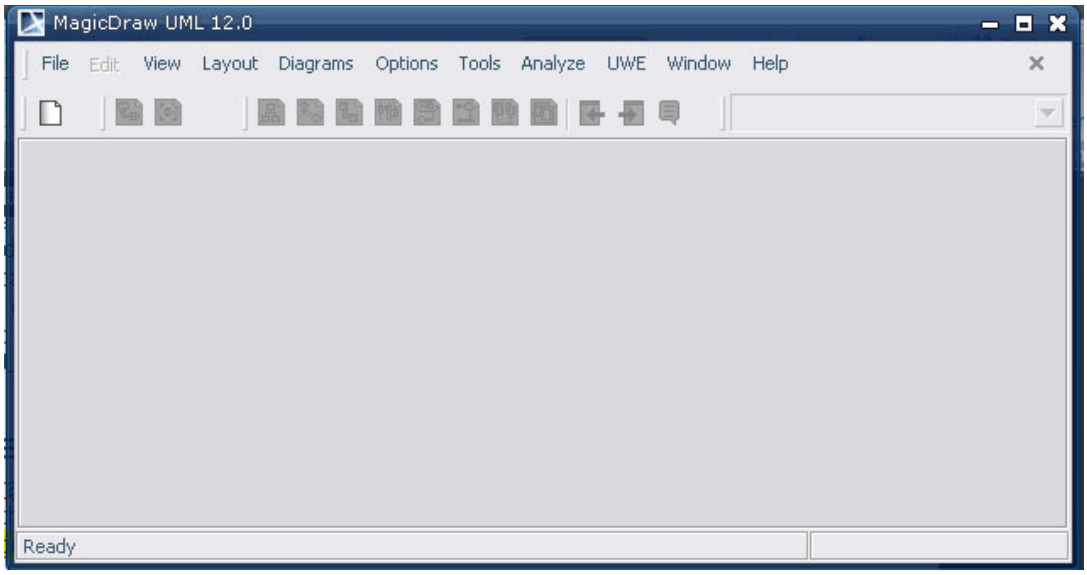

Figure 17 MagicDraw with integrated UWE

This *UWE* main menu is one of the starting points - the basis, where all UWE relevant main functionalities are integrated in the form of submenus. It contains *Diagrams*, *Transformation* and *About UWE* submenu items as shown on Figure 18.

The *Transformation* submenu item is unavailable by now, because there is no new project created yet. The Open API of MagicDraw gives us the opportunity to enable or disable almost any kind of action (like main menus, submenus, diagram actions, etc.) depending on predefined MagicDraw states (like new project was created, or class diagram was opened, etc.). In the next chapter further details about the specific implementation will be given. For now it is interesting that through enabling and disabling GUI elements we can easily navigate and accompany the user step by step in creating a UWE project and model.

As displayed in the previous figure it is obvious that the main menu item *UWE* comes out between the remaining generic MagicDraw main menus. There is no other plug-in menu in this top menu navigation. First design decision was to integrate UWE GUI elements only in the appropriate submenus, for example UWE diagrams under the *Diagrams* main menu (see the following description about the diagrams). Later, when

the UWE diagram functionalities were implemented the next step was to implement the UWE transformations and integrate those into the GUI. At this point it was decided to put both the transformation and diagram actions into this UWE main menu. That allows the user to receive a quick overview of the main UWE functionalities at a first sight. The user can execute those main UWE functionalities from here. Secondly, we wanted to have a compact UWE starting point including the main features of the plug-in. This UWE main menu can be later on extended by other items (actions), like code generation, help, automatic update, and so on.

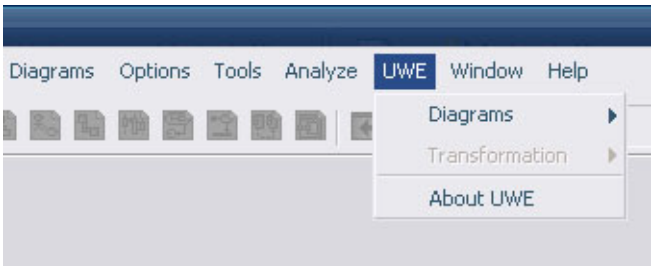

Figure 18 UWE Main Menu Submenus

It was already mentioned that the UWE main menu is one of the starting points for modelling with the UWE Plug-in. The second one is more integrated into the default MagicDraw main menu structure.

When taking a closer look at Figure 12, one can see that the item name *Diagrams* appears twice. There is a default main menu item and another one, as a submenu of the *UWE* main menu item both called *Diagrams.*

That is reasonable, since various types of diagrams are integrated in any UML modelling and CASE tool. Therefore to enhance the usability and the intuition, that a user might have built using the MagicDraw products, we decided to integrate all UWE specific diagrams also under the default *Diagrams* main menu item of MagicDraw. For that reason a *UWE Diagrams* submenu is included into the default *Diagrams* containing the same diagram actions as under the *Diagrams* submenu of the *UWE* main menu. This is visible on Figure 19. These UWE Plug-in design decisions are a good example of how the plug-in is effectively integrated into the hosting program.

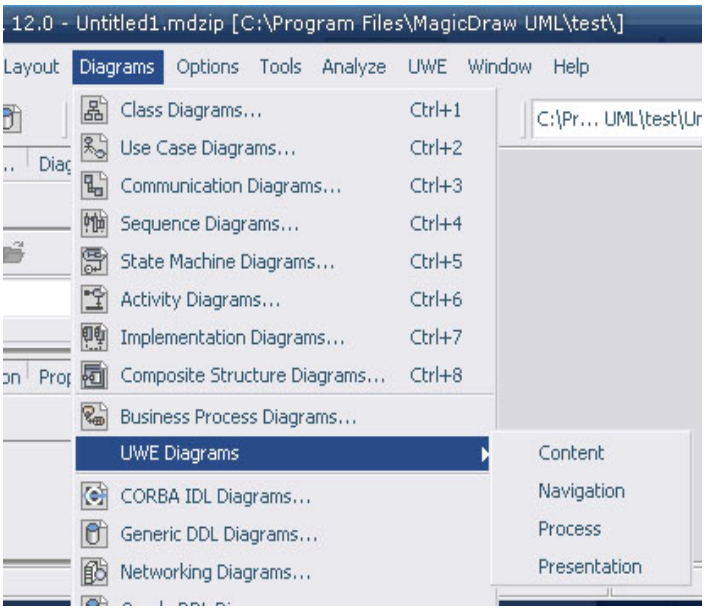

Figure 19 UWE under Diagrams

One can admit that only one menu item is enough for the same group of functionalities or actions. But the example above gives more flexibility and conformance in using the UWE Plug-in. On the one side there are all new plug-in actions and functions grouped in one UWE menu item and on the other side, important functionalities are integrated deeper in the existing MagicDraw GUI structure so the user can still use his old habits of working with MagicDraw. For example, the user can look for any diagrams under the default MagicDraw *Diagrams* main menu.

The UWE main menu encompasses another important submenu and this is the *Transformations* item. As shown on Figure 20 there are two, already mentioned UWE transformations. The first one is *Content2Navigation* and the second one is *Navigation2Presentation*. Opposite to the diagram actions described above, the UWE transformations are placed only at this submenu in the GUI of the tool.

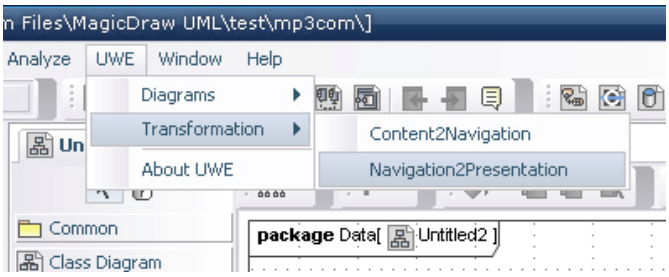

Figure 20 UWE Transformation Actions

For further consideration of implementing the UWE code generation in the future, maybe the UWE main menu will be the right place to integrate this functionality. Decisions have to be made whether the code generation belongs to the transformations or a new submenu, for example *Code Generation* will be created.

4.2.3 UWE Diagram Toolbar Menu

Next group of GUI UWE menu items and actions are created into the diagram toolbar.

As the name says, this toolbar is shown only when a diagram is opened. By now the UWE menu items (actions) integrated into this toolbar are visible every time a diagram of type class diagram is opened and not only when one of the UWE diagrams (*Content, Navigation, Process,* and *Presentation*) is opened.

The UWE Plug-in creates its own diagram toolbar menu group called *UWE*. The UWE actions included into this group are in turn grouped into three groups depending on the UWE package they belong to. The first group includes actions from *Navigation* package; the second includes actions from *Presentation* package, and the last one includes action from *Process* package. Figure 21 shows the *UWE* item diagram toolbar structure.

Because these functionalities are explicit diagram actions, putting them all in the already existing UWE main menu will be confusing for the users of the tool. Furthermore it will destroy the systematic GUI design of the hosting tool.

Alternatively we tried to put all these actions under already existing toolbar menu items, more precisely into the toolbar menu item *Class* and *Association* from the *Class Diagram* group (see Figure 21). The reason behind is that all UWE actions create UML elements (classes or associations) with assigned UWE stereotype. So logically classes should be integrated into *Class* and associations should be integrated into *Association* (compare with the decisions made by the transformations).

After implementing this first solution, it was realised that the UWE diagram actions should be separated from the default UML modelling items. The main reason for this decision is that one more time we wanted all UWE diagram actions to be grouped under one UWE menu and then separated depending on their definitions. Furthermore after using the first solution, we found out that it is not so intuitive to look for UWE actions under the default UML *Class Diagram* group.

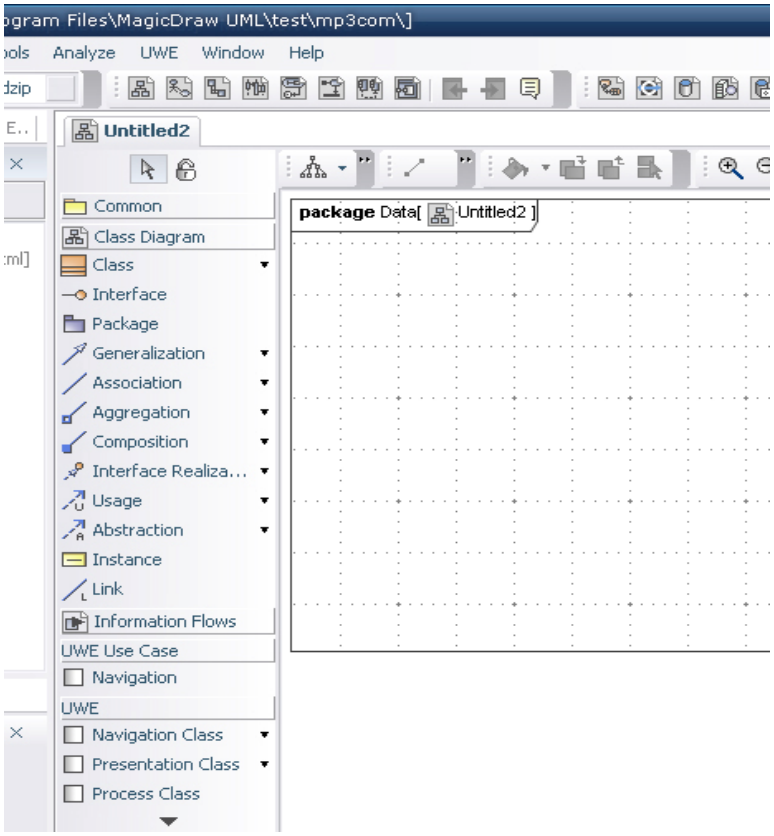

Figure 21 UWE Diagram Toolbar Actions

Another UWE diagram toolbar menu that is integrated into MagicDraw with the UWE Plug-in is the *UWE Use Case* diagram toolbar group shown as well in Figure 21. This diagram menu is displayed when a class diagram is opened as well as when a use case diagram is opened. By now this group consist of only one UWE use case action called *Navigation*. Because this action is relevant only for the UWE use case diagrams we decided to separate it from the rest of UWE diagram (class diagram) actions.

4.2.4 UWE Diagram Context Menu

The semi-automated insertion of UWE elements functionality is placed into the diagram context menu of a diagram. As shown on Figure 41 there is again a menu group called *UWE* and two actions: *Insert Query* and *Insert Index*. Because the user has to click on an association for selecting it before one of these actions can be executed, the best solution was to create diagram context menu items for this type of actions. The user has simply to click once with the left button, the association is selected, then once with the right button and has to choose the desired context action. There is no need to move the mouse pointer to the diagram toolbar or main menu.

5 Implementation

The implementation of the UWE CASE tool as an extension (plug-in) for MagicDraw will be discussed in this chapter.

Although MagicDraw is a close-source application the UWE extension is implemented as Plug-in into it using its Open API interface. This interface is written in Java just like MagicDraw. The Open API provides also the opportunity to write and develop Plug-ins also in JPython [@11] scripting language.

JPython is actually a Python written in Java, and seamlessly integrated with Java. It thus allows to run Python on every Java platform [@12]. This possibility was not used in the UWE Plug-in, which became a pure Java application.

The UWE application (Plug-in) was implemented from scratch without using any existing UWE products or code. Basically only the GUI functionalities of the already existing ArgoUWE were inspected to better understand the typology and structure of the UWE approach. ArgoUWE is an extension for the open source tool ArgoUML for the UWE approach. More information about ArgoUWE can be found in the following reference [@13].

Because the Plug-in was implemented not only as a prototype of a UWE CASE tool, it should be capable to do all steps of modelling described in previous chapters and some more features which came out during the development process. Some of them are the following:

- Loading of the UWE Profile to every UML project
- Creating a new project through UWE Template file
- Integration of the Plug-in in the GUI of MagicDraw
- Be able to create any of the four types of UWE diagrams and automatically to load the UWE Profile if it hasn't done yet
- Default assignment of model elements stereotypes on the basis of defined rules
- Configuration of some stereotypes values through application properties file
- Transformations form one diagram type to another on the basis of defined rules
- Consistency checking of the created UWE model

5.1 Writing a plug-in

A plug-in is a piece of software that extends an existing software application or platform in numerous of ways. The main purpose of a plug-in is to add new functionality and features to some existing application, whereas there might be some limited ability to remove already existing functionalities for some reason or another.

In our case the goal is only to add new functionalities while fully using the default spectrum of functionalities provided by MagicDaraw modelling tool. Creating a plug-in is the only possibility how to enhance closed-source software with new features that the user wishes for. The plug-in developer has the power to suit that software to his needs limited only by the Open API interface provided by the existing program.

Before starting to create a new plug-in the first thing to do is to search various plug-in repositories or Internet sites and see whether there already exists a plug-in for the software that suits the project's needs. Of course in our case there is no such a plug-in for MagicDraw that implements the UWE approach.

Next step is to get familiar with the plug-in interface of the hosting program. This is the so called Open Application Programming Interface (API) which is provided by any extendible software. It assumes that the user is also familiar with the programming language of the Open API. In our case the programming language is Java. In some cases the documentation of the interface can be very accurate. In other cases the examples provided with the software or other open-source plug-ins for that program are even more useful.

The first task in creating a plug-in is to analyse what the plug-in will do, and create a unique name for this plug-in. Existing plug-ins and other places (repositories, internet sites) need to be checked to verify that the name is unique. Most common way to choose a plug-in name is to use names that somehow describe what the plug-in does. For instance a time related plug-in would probably have the word "time" in the name. The name given to the UWE Plug-in is "*uweMDPlugin*". The name means "UWE MagicDraw Plug-in". More about the plug-in structure and Open API for MagicDraw will be discussed in the next chapters.

5.2 MagicDraw Open API

Plug-ins are the only way to change the functionality of MagicDraw. The plug-in typically creates some GUI elements and adds them to the MagiDraw graphical user interface. The plug-in is capable of reacting to user interaction - "listening" for some changes in the project - without using GUI elements and thus reacting to the user behaviour only.

A MagicDraw plug-in has to contain the following resources: [15]

- Plug-in directory
- Compiled Java files, packaged into jar file
- Plug-in descriptor file
- Optional files used by the plug-in

The first file contains information on where to find all necessary MacigDraw OpenAPI documentation files. All needed files are located in *<MagicDraw installation directory>/openapi/docs*. One of the files is the "MagicDraw OpenAPI UserGuide.pdf". This file is the documentation and user guide of the OpenAPI interface of MagicDraw. Here the user can find step by step examples and information on how to write a plug-in, what can be integrated into a plug-in, etc.

The second file is generated JavaDoc file with details of classes, attributes and operations. Although the Open API contains a huge number of classes the most of them are very poorly commented in the JavaDoc. Basically there are only the names of the classes and their constructors and methods with their attribute names, but the description what exactly the class or method is doing is absent. The user can only

guess the functionality by their names and inherited abstract classes and interfaces.

Every installation of MagicDraw provides also a set of samples of the functionalities of the Open API interface. These samples can be found in *<MagicDraw installation directory>/openapi/samples*. Sometimes samples are the best way to find out how to use some Open API, especially when no sufficient documentation is available.

The last possibility to figure out a solution to an issue related to the implementation is to write directly to the creators of the application. Generally the creators run various types of internet forums where groups of plug-in developers can exchange knowledge among each other. You can find all necessary information about the forums on the MagicDraw home page at [@5].

The steps above describe the main resources of information about plug-in implementation. In some cases (depending on how good the documentation of the OpenAPI interface is) the plug-in developer has to anticipate increased time expenditure for the implementation process. The MagicDraw OpenAPI documentation itself describes only insufficiently its functionality, thus many problems have been solved only after contacting the creators in the internet forum.

There is one more step to be performed before starting with coding: the necessary Open API classes have to be loaded into the plug-in project have to be specified. These classes are packed in jar files which have to be imported into the plug-in project into the integrated development environment (IDE). Here are the jars needed for implementation of a plug-in: [15]

- <MagicDraw installation directory>/lib/md.jar
- <MagicDraw installation directory>/lib/uml2.jar
- <MagicDraw installation directory>/lib/javax_jmi-1_0-fr.jar
- <MagicDraw installation directory>/lib/cmof14.jar
- <MagicDraw installation directory>/lib/y.jar

5.3 UWE Plug-in Design and Architecture

This section will describe the main software architecture and design basic essentials of the UWE Plug-in. First of all we have to understand how plug-ins in MagicDraw works. As shown in Figure 22 MagicDraw on every start-up scans the plug-ins directory and searches there for subdirectories with the following rules:

- If subdirectory contains plug-in descriptor file, than the Plug-ins Manager reads the descriptor file
- If requirement specified in descriptor file is fulfilled, plug-ins Manager loads specified class from the given jar file. Specified class must be derived from *com.nomagic.magicdraw.plugins.Plugin* class. At this moment the *init()* method of loaded class is called.

The method *init()* from the plug-in called while initializing the application should add GUI components using actions architecture or do other activities and return from the method. In the same plug-in subclass derived from *Plugin* there is one more method that has to be overwritten: the method *close*(). This method in turn is called from the Plug-ins Manager on MagicDraw shutdown. Furthermore the *close*() method has to return true, if the plug-in is ready to close or false if not and the shutdown process will be cancelled.

Well so easy can be plug-in creating, let's see how are these methods implemented in

the UWE Plug-in and which further files and steps are necessary before running MagicDraw with it.

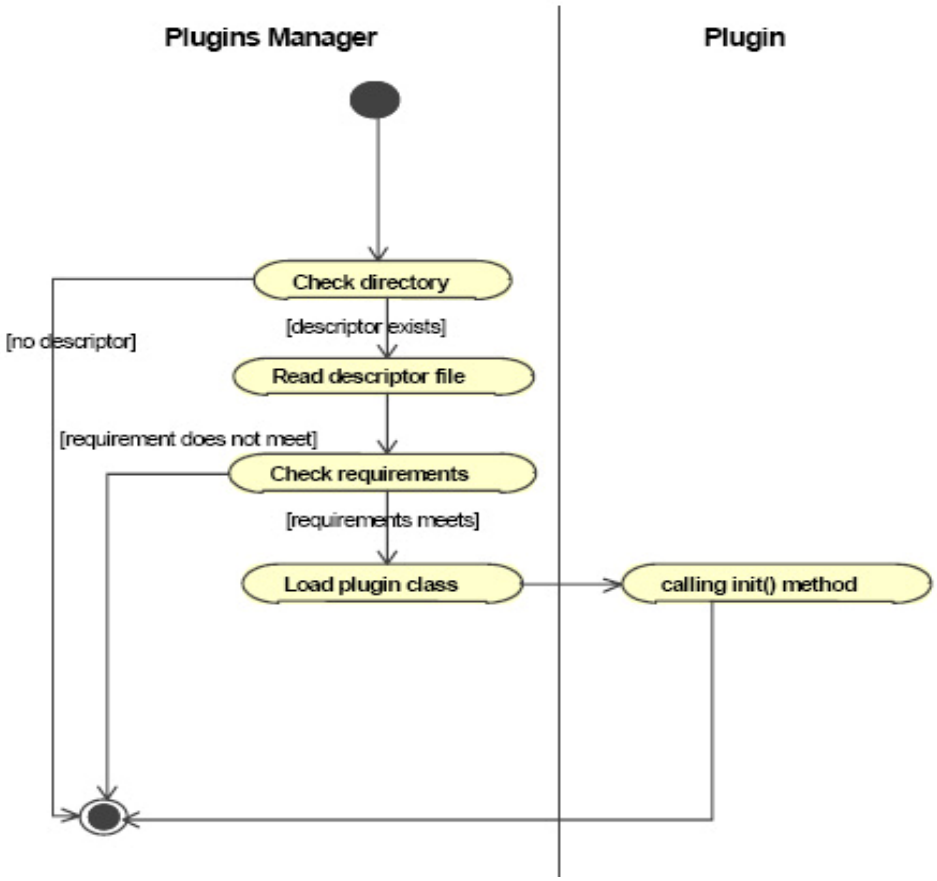

Figure 22 Plug-ins Manager Process Flow

As stated at the beginning of this section the Plug-ins Manager starts searching for a descriptor file in the plug-in directory. Such a descriptor file is a proper XML (Extensible Markup Language) [@15] file, defining some information about the plug-in itself. The descriptor file is called *plugin.xml* for any plug-in and contains properties of the plug-in. Descriptor file should contain only one "*plugin*" element definition. All names of the elements and their attributes are defined in the *MagicDraw OpenApi UserGuide*.

The UWE Plug-in directory is called as the plug-in itself: *<MagicDraw installation directory>/plugins/uweMDPlugin.*

The directory contains two files: the *plugin*.xml and the *uweMDPlugin*.jar. Let's look at the descriptor file *plugin.xml* in Figure 23. Besides the elements that describe the name, version, and version of the used Open API there are two important attributes giving the names of the plug-in library: *uweMDPlugin.jar* and of the main plug-in class: *de.lmu.ifi.pst.plugin.uwe.manager.PluginManager*. It means that MagicDraw at start-up will look for the *PluginManager* into the library *uweMDPlugin*.jar. The PluginManager class is a subclass of represented above *Plugin* class and implements the both methods: *init()* and *close();*

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
        id="uwe"
        name="UWE Project"
        version="1.0b"
        provider-name="LMU – Institute for Informatics"
        class="de.lmu.ifi.pst.md.plugin.uwe.manager.PluginManager">
        <requires>
          <api version="1.0"/>
        </requires>
        <runtime>
          <library name="uweMDPlugin.jar"/>
        </runtime>
</plugin>
```

Figure 23 plugin.xml

The *PluginManager* class is the main class that constructs the UWE Plug-in. In its *init()* method all configurations, GUI actions elements, and other functionalities are created which later are called from the MagicDraw user interface trough the user interaction.

Let's see a small code snippet of *init()* method. Figure 24 displays the beginning of the method and how action elements are added to the GUI of MagicDraw. First obvious thing that could be noticed is the *ActionsConfiguratorsManager* object called in this case *manager*. The ActionsConfiguratorsManager class is a part of the Open API. It is a singleton class for adding or removing configurations of actions managers in MagicDraw application.

```
…
public void init() {
                …
   //initialize project listener
   projectListener = new ProjectListener(projectsManager);
   Application.getInstance().addProjectEventListener(projectListener);
   ActionsConfiguratorsManager manager =   ActionsConfiguratorsManager.getInstance();

    // adding submenu
   manager.addMainMenuConfigurator(new MenuConfigurator(getSubmenuActions()));
   // adding actions with separator
   manager.addMainMenuConfigurator(new MenuConfigurator(getSeparatedActions()));
   // add browser item
   manager.addContainmentBrowserContextConfigurator(new BrowserMenuItem());
   // add submenu to Diagrams main menu
   manager.addMainMenuConfigurator(new
DiagramsMenuConfigurator(getDiagramsMenuAction()));
   // add class diagram toolbar actions
   manager.addDiagramToolbarConfigurator(DiagramType.UML_CLASS_DIAGRAM,     new
ClassDiagramToolbarConfigurator(getDiagramToolbarProcessActions()));

…
```

Figure 24 PluginManager init() Method

At this place I would like to explain in a little bit more detail the structure of the MagicDraw Open API architecture, before continuing with the explanation of the code fragment.

As written before, MagicDraw is implemented in the Java programming language using the Swing toolkit for its GUI. However the MagicDraw Open API provides its own actions mechanism to add new functionality to the application and the way to invoke them trough interaction with the GUI. The exact MagicDraw actions hierarchy can be viewed in Figure 25.

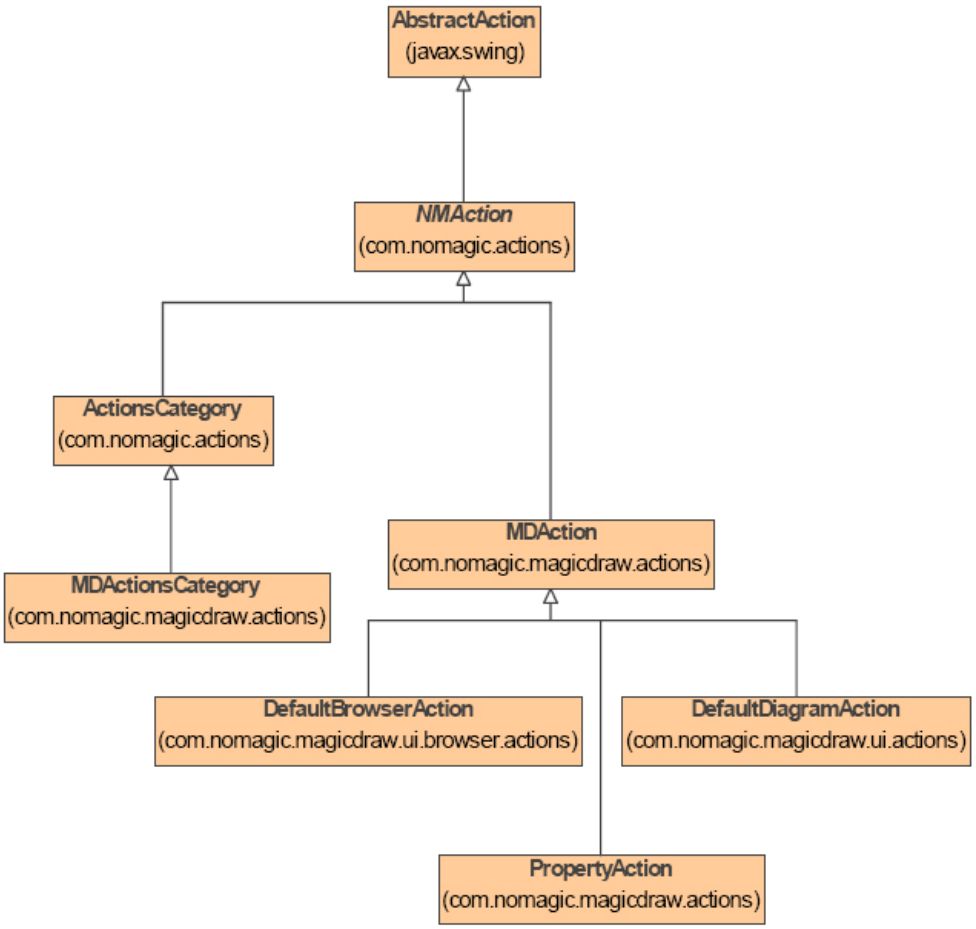

Figure 25 MD Actions Hierarchy

The first step in adding new functionality is to create a custom action class. This action class comprises the functionality which will be later fired from the interaction of the user with the application. Furthermore the class must be a subclass of the *MDAction* class from the Open API. There are already few predefined action classes for different purposes, so one can use them as needed. Every action class has also to describe its properties like: *id, name, shortcut, icon*, etc.

Every action must be added to some actions category. This step can be seen as the second level of the action implementation. The *ActionsCategory* class is used to group the actions. It can be also represented as a separator or submenu.

Categories in turn are added into actions manger using the *ActionsManager* class. This can be explained as some kind of container holding the groups of actions. There are different kinds of *ActionsManager*, each of them representing one GUI element: menu

bar, context menu or toolbar.

Table 1 explains how MagicDraw classes map into GUI elements.

	Action	**Category**	**Actions Manager**
Context Manu	Menu Item	Submenu	Context Menu
Toolbar	Button	One Toolbar	All Toolbars
Menu	Menu Item	Menu	Menu Bar

Table 1 Mapping of MagicDraw Classes to GUI Elements

All actions included in actions managers are configured by numerous Configurators. These Configurators are responsible for adding or removing an action. Furthermore they can be added only into some strictly defined place and positioned between other actions. The Open API provides three types of Configurators:

- *AMConfigurator* – for general purpose. Used for menus, toolbars, browser, and diagrams shortcuts.
- *BrowserContextAMConfigurator* – for configuring managers for browser context menu.
- *DiagramContextAMConfigurator* – for configuring context menus in diagrams.

At this place it is very important to notice, that actions managers for the main menu and all toolbars are created and configured once on the start-up. Thus actions later can be only disabled but not removed. Context menus in turn are created in every invoking. Therefore actions managers are created every time and actions can be added and removed any time.

After this brief introduction to adding new functionalities into MagicDraw, let us have a look one more time at the example in Figure 24. I think the code shown there is now more readable and some elements discussed before could be found.

Beholding the first actions that are added as submenu actions, exactly the steps described above could be seen. The method *getSubmenuActions*() returns all submenu action objects. These are passed to the *MenuConfigurator* and in turn the Configurator is passed to the actions manager and created at the proper place. The same scenario is applied also for all other UWE actions for different types of menus. As already discussed above, each action is added to an actions category. This step is not shown at this code example but it is done in the *getSubmenuActions*() method. Figure 26 shows this method where an *ActionsCategory* is created: *category*. The next move is to add all newly created actions into this category. At the end, the method returns the category with all actions in it.

This example shows what is necessary to be done in order to add and to register new actions to MagicDraw. Of course all other functionalities are specified in every type of action classes.

```
/**
 * Creates action which is submenu (when represented in menu).
 * Separator is added for group of actions in actions category.
 */
private NMAction getSubmenuActions(){
    ActionsCategory category = new
ActionsCategory(GlobalConstants.DIAGRAMS_SUB_MENU_NAME,
GlobalConstants.DIAGRAMS_SUB_MENU_NAME, null,
ActionsGroups.PROJECT_OPENED_RELATED);
    // this call makes submenu.
    category.setNested(true);
    category.addAction(new DiagramAction(null, GlobalConstants.CONTENT_DIAGRAM,
projectListener));
    category.addAction(new DiagramAction(null,
GlobalConstants.NAVIGATION_DIAGRAM, projectListener));
    category.addAction(new DiagramAction(null, GlobalConstants.PROCESS_DIAGRAM,
projectListener));
    category.addAction(new DiagramAction(null,
GlobalConstants.PRESENTATION_DIAGRAM, projectListener));
    return category;
}
```

Figure 26 getSubmenuActions() Method

For more details of all named Open API classes above please refer to the Open API Java Doc included with every installation of MagicDraw.

5.3.1 The UWE Profile and Template

After knowing how to define new actions we will have to be able also to assign the UWE defined stereotypes to model elements. As mentioned in this document at many places, we need a UWE Profile where all UWE metamodel elements (stereotypes) are defined. In MagicDraw there is a predefined approach how to add a custom stereotype. Even more, when once a profile is created, it can be reused in any MagicDraw project. The UWE plug-in provides such a profile file, which is installed during the installation process.

The UWE Profile is actually a standalone file that can be exported as a "*Packed MagicDraw File Format*" (*.mdzip), "*MagicDraw File Format*" (*.mdxml), or to XML file format (*.xml). Once creating the UWE Profile users and modellers can redistribute and reuse this profile in any MagicDraw program (depending on the MagicDraw version from which it was exported). All default and custom profiles are stored in the profiles directory of the MagicDraw root directory.

As stated before, once created a profile can be used in any modelling project. By modelling with the UWE approach the UWE Profile has to be loaded into the project. This can be done manually or the UWE Plug-in will recognize that the user is starting a new UWE diagram and the UWE Profile will be loaded automatically.

As a matter of fact there are two possibilities to load the UWE Profile into the project. The bottom line is that both have also a slight design difference effect. The first method is just to start a new standard project and then to create any of the UWE specific diagrams. At this moment the UWE Profile will be loaded into the project.

The second method is to start a new project from a template and to choose the *UWE* template from the available templates menu. At this moment the new UWE project will be created with the UWE Profile loaded in it and additional four data packages in the data root directory. As shown in Figure 27 these packages are: *Content, Navigation, Presentation and Process.*

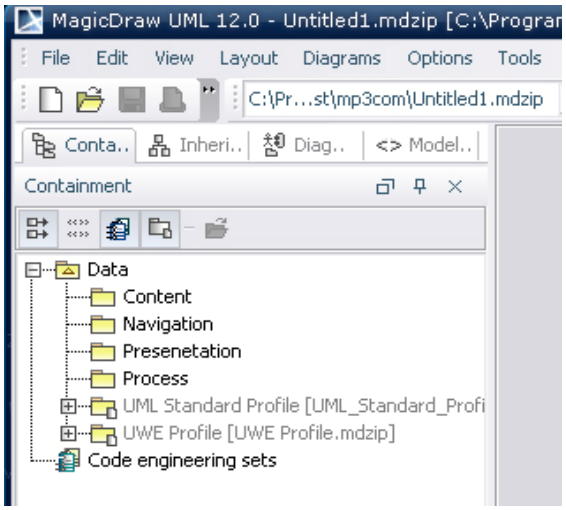

Figure 27 Loading of the UWE Profile through UWE Template

The creation of those predefined four model packages was made to show the difference between starting a UWE template and just loading the UWE Profile in some project. More about designing and modelling with the UWE Plug-in will be discussed in chapter 6.

The following sections describe how such a profile is created and what specifics shall be considered when creating a profile in MagicDraw.

5.3.1.1 Creating a Profile

To define a profile in MagicDraw a new default project has to be created first. After the new project is opened, the model element browser tree window is displayed on the left side. As default in every project there is automatically the UML Standard Profile loaded and included into the project. Now a new package has to be created (to see how to create a package, please refer to the *MagicDaraw User Guide*) and a name has to be given (equals the name of the new profile). At this point a package has been created where all UWE stereotype elements will be added. Let us say this is the root of the new profile.

To be more precise the UWE Profile has five packages: *Content*, *Navigation*, *Presentation*, *Process* and *WebRE*. They build the same structure as described in the UWE metamodel before. At this place the proper UWE stereotypes have to be created and added into those packages. To do so a click with the right mouse button on the package name has to be done and *New Element* from the context menu window has to be chosen. After that the type of the new element has to be chosen, in this case it is a stereotype. Now the name, the metaclass, and other attributes of this newly created stereotype can be set. In the same manner all UWE profile packages are filled with the necessary UWE stereotypes. For instance there are the following stereotypes in the *Navigation* package: *Navigation Class, Index, Menu, Navigation Link, Navigation Property, Node*, etc. In the *Presentation* package in turn there are: *Presentation Class,*

Form, Page, Image, Anchor, Text, Text Input, etc.

We have now all required stereotypes and packages for the UWE profile but sill they remain in a normal MagicDraw project. There are further actions to be performed before the UWE profile is ready to use.

These newly created UWE stereotypes have to be shared in order to be available for usage in any MagicDraw project (after loading the UWE profile). This means, that the stereotype elements form the shared profile or module can be used in any other project. To do this the UWE root package has to be chosen and clicked with the right mouse button. From the opened context menu, it has to be scrolled down till the *Modules* menu item and clicked on *Share Packages* inside it. Now the root UWE package has to be selected and finally OK has to be clicked to proceed.

At this moment there is only one last step to be made. The project has to be exported as a *Modul*, and saved. This means, that this newly created file could be loaded as a profile in any MagicDraw project in the future. (Please refer to the *MagicDraw User Guide* to see how to export a project.) While exporting the module, also the name of the profile file has to be specified, in our case the name is: *UWE Profile*. Depending on the file format settings a profile file with the specified name is created. In our case it is called "*UWE Profile.mdzip*". Last thing to do now is to move the UWE profile file into the profiles directory of MagicDraw. After restarting MagicDraw the created UWE profile can now be loaded into any project through the *Use Module* submenu of the *File* main menu.

Because a UWE Profile is provided with the installation of the UWE Plug-in the user of does not have to perform the steps described above. The pug-in is "clever" enough, so it will assist the user when a new project is created. The plug-in is implemented to check automatically if the UWE profile is already loaded when the user is trying to create one of the UWE's diagrams. If the profile is not yet loaded, it will be automatically loaded into the project. This step is necessary because all defined UWE stereotypes have to be defined (loaded) before starting to draw some classes and links into the UWE diagrams. This mechanism is implemented through the *loadUweProfile*() method of the *ProjectListener* class. Figure 28 shows a code fragment of this method. As can be seen the *MountTable class* is responsible for manipulation of module (profile) files in MagicDraw and the *ProjectDescriptor* represents the profile file.

```
/**
 * loads the uwe profile file from the profile directory
 * and returns true if successful
 */
public boolean loadUweProfile(){
    ProjectDescriptor module =
ProjectDescriptorsFactory.createProjectDescriptor(moduleProfileFile.toURI());
      MountTable mountTable = activeProject.getMountTable();
      boolean done = false;
      synchronized (this) {
          try{
          ModuleDescriptor moduleDescriptor = mountTable.mountModule(module);
          moduleDescriptor.setEditable(false);
          mountTable.loadModule(moduleDescriptor, new SimpleProgressStatus());
          mountTable.importModule(moduleDescriptor);
          done = true;
…
```

Figure 28 loadUWEProfile() Method

Even more the UWE profile file name can be set in the *uwePlugin.properties* file in the plug-in sources through the *uweProfileName* property. This can be useful if there are more than one defined profiles and the user would to be able to choose which one to use in his project.

The UWE Profile is mandatory for the UWE Plug-in to work properly because of the references of stereotypes defined in the profile. If the profile is damaged or missing, it has to be reinstalled or a new one has to be created.

There is one more possibility how to load all necessary UWE stereotypes (included in the profile) into a MagicDraw project. This topic will be described in the next section.

5.3.2 The UWE Template

MagicDraw allows also for creating a new project from an already existing template. The newly created project will contain specific model elements and stereotypes. Because the UWE plug-in has to support also this feature of the application, a UWE template is necessary.

Let us see how a template is created. Normally there are almost the same steps needed as for creating a profile. The only difference is that the project that contains all defined stereotypes has to be exported as a *Template* and not as a *Module* like when creating a profile. After the template is exported new files are created in a directory with the template name that was given during the exporting process. In this case the name of the template directory is *UWE*. This directory has to be moved into the *templates* folder of MagicDraw. After restarting MagicDraw the user can choose to create a project from template, and the UWE template will be listed in the templates tree browser under the name *UWE*.

In our case there is no need to define the UWE stereotypes one more time. Creating a UWE template as described above will create only a second type of UWE profile containing exactly the same stereotypes like those the *UWE Profile* already contains. In this plug-in *UWE Template* has the same functionality as the *UWE Profile*. Furthermore it means that there has to be a way how to define the profile once and use it in the both cases. Let us see how this is achieved.

As the aim was to have the same functionalities like when importing the UWE profile, a decision was made that the template has to load automatically the same UWE profile file defined in the previous section. To achieve this, a tricky approach was needed for modifying the exported UWE template file.

After exporting the template (without defining any stereotypes) a file called *UWE.mdzip* is created. Its file extension is actually a MagicDraw XML file packed through the zip method. To be able to modify the packed file, *UWE.mdzip* has to be extracted (unzipped). Newly extracted file can be opened with any text or XML editor, because the file is a regular XML file. The next step is to add an entry that refers to the UWE profile file in the profiles directory of MagicDraw. This entry is shown in Figure 29.

As shown on the previous figure there is an element called *module* which is exactly referencing to the UWE profile file. After adding this element to the existing file definitions, the file has to be saved and packed (zipped) again. Finally the file has to be renamed back into *UWE.mdzip*. After these modifications are made, when starting MagicDraw again and choosing to create new project from the *UWE* template the *UWE Profile* will be loaded automatically.

```
...
<xmi:Extension xmi:Extender="MagicDraw UML 12.0" xmi:ExtenderID="MagicDraw UML
12.0">
    <shareTable/>
    <mountTable>
        <module resource="file:/D:/MD_115/profiles/UML_Standard_Profile.xml"
autoloadType="ALWAYS_LOAD" readOnly="true" loadIndex="false" requiredVersion="-1">
            <mount mountPoint="magicdraw_uml_standard_profile_v_0001"
mountedOn="eee_1045467100313_135436_1"/>
        </module>
        <module resource="file:/D:/MD_115/profiles/UWE%20Profile.mdzip"
autoloadType="ALWAYS_LOAD" readOnly="true" loadIndex="false" requiredVersion="-1">
            <mount mountPoint="uwe_profile_v_0001"
mountedOn="_12_0_6610220_1173265770454_852159_250"/>
        </module>
    </mountTable>
</xmi:Extension>

..
```

Figure 29 Reference of the UWE Profile in *UWE.mdzip*

There is one more specific thing of using the UWE template stated already in the previous sections. After a project is created from the UWE template, the following four packages will be automatically created in that project: *Content*, *Navigation*, *Presentation* and *Process*.

The user can now load the *UWE Profile* in all possible ways supported by MagicDraw. Furthermore this gives flexibility by using the UWE approach, as the modeller can either start a UWE project from scratch (using the UWE template) or load the *UWE Profile* only when it is really needed (by loading the UWE module (profile) or just opening a UWE diagram).

Please note that it is not possible to have only the UWE template without the UWE profile file, because the template is referencing to the profile file, and will throw an exception if the file is missing. Besides that, the UWE plug-in will also not work without the UWE profile. So if the UWE profile file gets corrupt or is missing it has to be restored (reinstalled) or created a new one. Furthermore if the name of the profile is changed, the maintainer of the plug-in has to adjust manually the template file specified above, otherwise the application would not be able to find the referenced profile file and an exception will be thrown.

5.3.3 The UWE Core System and Main Classes

The last two sections described how the UWE stereotypes are created. Actually the UWE profiles as well as the template are separated definitions files that are copied in the proper target directories during the installation of the UWE Plug-in.

All further files with all the necessary functionalities of the plug-in are packed in the UWE jar file in the plug-ins directory of MagicDraw. This section describes the main classes and methods of the plug-in.

Section 5.3 described how a plug-in is initialized and which methods are called during this process. Since then we know that the *PluginManager* class is the starting point where all GUI elements are registered in the MagicDraw application.

To understand better the interaction between the UWE plug-in on the one side and the Open Application Programming Interface of MagicDraw on the other, let us take a look at Figure 30. The UWE Plug-in is designed to fit best to the architecture of the Open API it means that there are three big component sections of the plug-in:

- *Manager* – consisting of plug-in main creator class: *PluginManager*
- *Listeners* – consisting of all kind of application listeners
- *Configurators and Actions* – consisting of all kind of Configurators and actions (GUI elements) and secondary classes needed for some specified actions such as transforming.

A *Manager* component member, *de.lmu.ifi.pst.md.plugin.uwe.manager.PluginManager* is straightforward subclass of *com.nomagic.magicdraw.plugins.Plugin*. The *PluginManager init()* method is the first plug-in method invoked from the application while initializing the MagicDraw application. As said before, at this place all GUI objects of the UWE plug-in are created and registered in MagicDraw through its Open API.

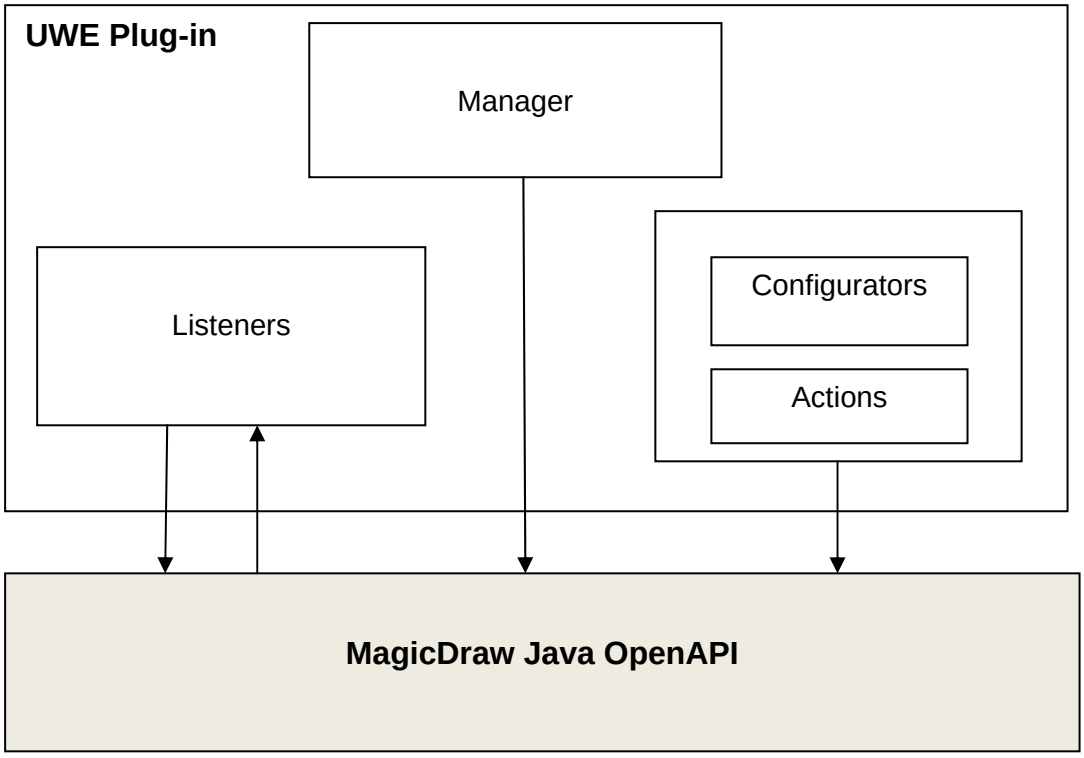

Figure 30 UWE Plug-in Architecture Overview

Members of the Listeners component group are the *ProjectListener* and the *ProjectEventChangeListener* classes. Both of them are in the same plug-in package *de.lmu.ifi.pst.md.plugin.uwe.core*. ProjectListener class is a straightforward subclass of *com.nomagic.magicdraw.core.project.ProjectEventListenerAdapter* class. Furthermore this class defines what actions are executed when project relevant actions are fired. Such actions can be: project opened, project closed, project saved, project deactivated, project replaced. This class is also responsible for the loading mechanism of the UWE Profile file. The method *checkIfloaded()* returns *true* if the profile is already loaded.

The second member of this group is the *ProjectEventChangeListener* class which implements the *java.beans.PropertyChangeListener* class. This class is the main event listener of the plug-in. Every time when a new project is opened an object of this class is instantiated through the *projectOpened()* method of the *ProjectListener*. All effects of model and diagram manipulations in the application are observed exactly from this class. Furthermore all default stereotypes of classes and associations are also set here. If we can apply the Model View Controller paradigm, this will be exactly the controller of the plug-in.

The last and the biggest components group contain all Configurators and actions of the plug-in. There are the following Configurator types:

- *MenuConfigurator* – creates the UWE main menu
- *DiagramsMenuConfigurator* - creates the Diagrams main menu
- *DiagramTransformersConfigurator* – creates the Transformations submenu of the UWE main menu
- *ClassDiagramToolbarClassConfigurator* – creates the UWE diagrams toolbar menu for the class element stereotypes
- *ClassDiagramToolbarAssociationConfigurator* – creates the UWE diagrams toolbar menu for the association element stereotypes
- *UseCaseToolbarConfigurator* – creates the UWE Use Case diagram toolbar menu

Every class of these configruators implements or extends a different type of MagicDraw configurator classes, depending on where exactly in the GUI the configuration will be placed.

There are also different types of UWE action classes, some of the most important are:

- *DiagramAction* – action for creating a UWE diagram, there are four different types of diagrams
- *DrawClassAction* – creates a class instance into the UWE diagram
- *DrawAssociationAction* – creates an association instance between two UML classes
- *TransformatorAction* – transforms elements from one diagram type into another

This components separation of the plug-in classes is also logically underlined with the package structure of the plug-in. The UWE plug-in contains the following Java packages: *manager, core, actions, configurators, transformation, and properties*. In the previous class examples there were mentioned classes from all these packages except for the *properties* package.

In this package there are two important classes, first of them is the *GlobalConstants* class. All plug-in constants with their values are defined in this class. The constants placed in this class are reused in more than one class of the plug-in, so it is more efficient and more accurate to place them in this class. Such a constant can be a UWE main menu name, a UWE diagram action, the UWE profile name and so on.

The second class in the *properties* package is *DefaultProperties* class. This class is responsible for reading the values of property attributes from the *uwePlugin.properties* file. The main properties defined there are: the UWE profile file name, the name of the default process association stereotype, and the name of the default navigation association stereotype. The values of these properties could be changed according to the comments written in the properties file itself.

Generally the goal of designing and implementing the plug-in was to design the architecture to be easy to maintain and to extend in the future.

5.3.3.1 UWE Plug-in Actions

One more topic in the UWE plug-in implementation is the use of action classes. Open API of MagicDraw predefines all possible actions that can be used in the application. As written in the previous chapters, all actions used in MagicDraw have to be a subclass of *MDAction* class. This declaration is not valid if we want to add an action into the toolbar menu of any type diagram. For this purpose we have to implement a custom class extending the *DefaultDiagramStateAction* which in turn extends *MDStateAction* class. Let us take a look at Figure 31. It is obvious that both, MDStateAciton and MDAction are subclasses of NMAction. Thus both of them have the same functionality base but are specified for a different disposal in MagicDraw. Because the actions used from the diagram toolbar menu are state actions we cannot use the “normal” *MDAction* to place our custom action somewhere in the toolbar, especially if several actions need to be grouped into one action name.

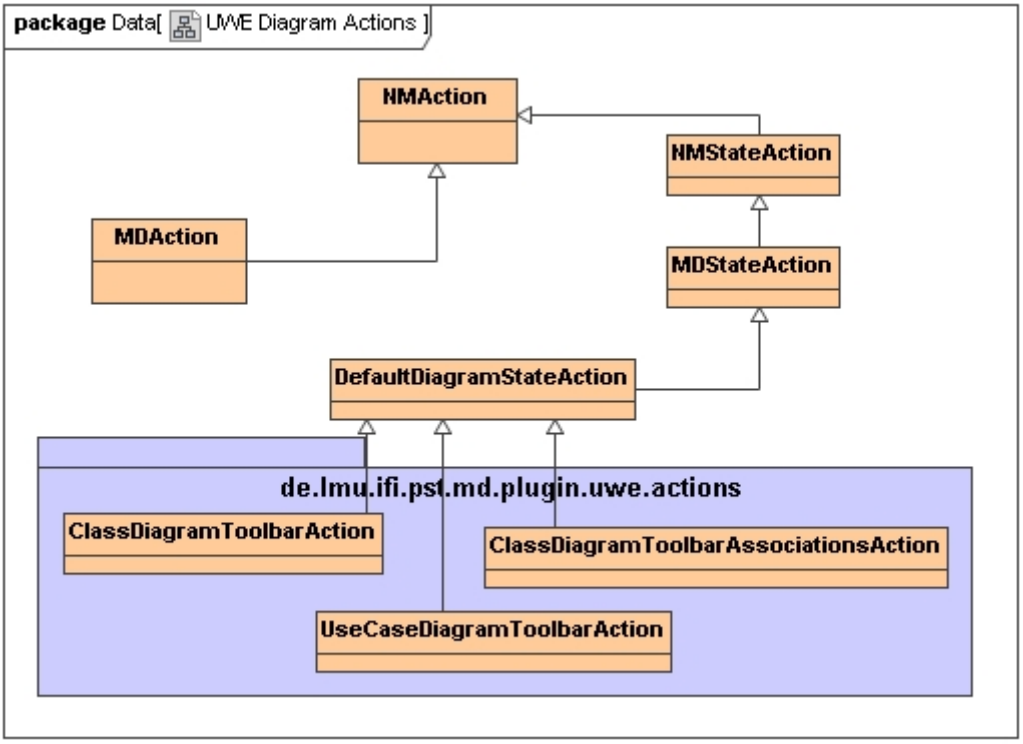

Figure 31 UWE Diagram Actions

A state action means that the action returns also some Boolean state, true if selected and false otherwise. Moreover in the figure above all UWE diagram actions can be seen. Whereas *ClassDiagramToolbarAction* is used for UWE class elements, *ClassDiagramToolbarAssociationsActions* is used for UWE association elements, and *UseCaseDiagramToolbarActions* class is used for the UWE use case element. More about the UWE diagram actions in the plug-in will be discussed in section 5.4.1.3 of this chapter.

5.3.3.2 UWE Plug-in Transformation

One of the most important issues of the UWE Plug-in is its semi-automated transformation mechanism. There are two types of transformation implemented in the plug-in depending on their specification in the metamodel and these are:

- Content diagram elements to navigation diagram elements,
- Navigation diagram elements to presentation diagram elements.

Both transformations are defined in the *DiagramTransformator* class of the package *de.lmu.ifi.pst.md.plugin.uwe.transformation*.

In order to illustrate how this class transforms the elements first I would like to explain how the UML elements are specified in MagicDraw. There are two kinds of elements:

- UML model elements, called model elements
- Presentation elements, called element views.

Usually there is an element view for each model element, displaying that element on a diagram. Actually the view element is a graphical element shown on the screen to visualize the model element. However, not every presentation element (view) must have a model element. Such example can be the presentation element *Text Note.*

It is complicated to find out which presentation element belongs to which model element, because they are associated only in one direction. It means that it could be found out which model element belongs to a view but not vice versa. To do so, the *getElement()* method of any presentation element should be called.

The implementation of the UWE transformation collects all presentation elements of the source diagram and then identifies their model elements. After that it creates new model elements applying the converted stereotypes of the source model elements. Some transformation rules are applied when choosing which elements, from which source diagram can be transformed and displayed in the target diagram. Furthermore there are some elements that are not transformed but only copied into the target diagram depending on the source diagram type and their own stereotype.

Maybe the most interesting and complicated part of used transformation algorithm is the moment when an association has to be transformed and connected to its new transformed supplier and client. The problem here is how to find out which new classes in the target diagram are the correct supplier and client.

First we need three hash maps to store the information that will be needed later. In the first one: *transformedClassElements* all pairs of old (source) element and transformed (target) element during the transformation process are stored, whereas the key is always the old element from the source diagram. The second hash map: *path2origSuplier* stores the path element of the source (original) association and it supplier. The key is the path element. The final map: *path2origClient* is analogous to previous map and stores the original path element with its client. Again the key is the path element.

Now the whole needed information is stored and the class and association elements can be transformed. After the classes are transformed (new class elements are created in the target diagram), comes the turn of their association. A new (transformed) association is created, whereas its presentation element (the path element of the new association) can be created only when a supplier and client exists and can be assigned.

At this point filled up with information hash maps above have to be used. Simply the source association will be used to find out the source supplier and client from *path2origSuplier* and *path2origClient.* Next the transformed (new created in the target

diagram) class elements will be holed from the third hash map, *transformedClassElements* using previous got source supplier and client as a key. Finally the transformed supplier and client are assigned to the new created (transformed) association. Shown in Figure 32 code fragment is a part of the whole algorithm described above. This part is the final part when the new association and its path element: *newAssPath* with supplier and target is created.

```
…
    Relationship relationship = newAss;
    logger.debug("relationship: " + relationship);
    PathElement pathEl = (PathElement) origAss;
    logger.debug("pathEl: " + pathEl);
    // get the new client
    PresentationElement origClient = path2origClient.get(pathEl);
    logger.debug("origClient: " + origClient);
    PresentationElement newClient = transformedClassElements.get(origClient);
    logger.debug("newClient: " + newClient);
    //get the new suplier
    PresentationElement origSuplier =  path2origSuplier.get(pathEl);
    logger.debug("origSuplier: " + origSuplier);
    PresentationElement newSuplier = transformedClassElements.get(origSuplier);
    logger.debug("newSuplier: " + newSuplier);
  //one of them can be null due of the restrictions made in isElemenent4transformation!
    if(relationship != null && origClient != null && newClient != null && origSuplier
!= null && newSuplier != null){
        //set the association client
        ModelHelper.setClientElement(relationship, newClient.getElement());
        //set the association suplier
        ModelHelper.setSupplierElement(relationship, newSuplier.getElement());
        //create new path element for the new association with its client and suplier
        PathElement newAssPath = presentationElementsManager.createPathElement(
relationship, newClient, newSuplier);
…
```

Figure 32 Transformation Implementation

5.4 Problems During the Development

As in any software development there are some smaller and bigger problems that occur during the implementation process. In this case, while implementing the UWE Plug-in the main problem was the insufficient or even missing documentation of a large number of classes and methods in the Open API documentation provided with MagicDraw. Thus after the quick start into the Open API and plug-in environment, bigger problems have appeared, when solving complex implementation tasks.

The first way how to find a solution on such undocumented functionalities or approaches of the Open API is to inspect the already existing code. Generally with the MagicDraw installation there are several plug-in samples for almost all scopes of the plug-in installation. For example, there are samples showing how to add different types of actions into MagicDraw, also how to draw shape elements such as classes or text boxes, how to use configurators and which of them for what purpose, how to use

selection actions, and so on.

Unfortunately as there are only few of those examples provided by default, the next possible way how to move forward is to use online MagicDraw Open API forums and newsgroups. The official newsgroups that are provided by the Magicdraw developing company can be found on the MagicDraw homepage: [@14]. The most important newsgroup for implementing a MagicDraw plug-in is called "*MagicDraw OpenAPI related questions and discussions*". The most helpful information for solving the implementation questions and problems was found exactly in this newsgroup. The other side of the coin is that the process of resolving a problem can cost a lot of time till some other developer gives more information or even resolves the issue.

5.4.1 Concrete Examples of Implementation Problems

In this section an overview of the main problems and their solutions are described. Of course some of their solutions are "elementary", while others are more complex, but each of those was quite a problem depending on the implementation phase of its appearance.

5.4.1.1 Understanding the Module Loading Mechanism

After creating the UWE Profile actually the implementation phase of the plug-in started. The first step was to get reassured that the newly created profile can be used and also to find out if exactly the UWE Profile is loaded into a MagicDraw project. The loading of the profile is a very important step of the plug-in, because if the UWE Profile cannot be loaded all other functionalities of the plug-in cannot be used.

```
/**
 * loads the uwe profile file from the profile directory
 */
public boolean loadUweProfile(){
    ProjectDescriptor module =
ProjectDescriptorsFactory.createProjectDescriptor(moduleProfileFile.toURI());
    logger.debug("module: " + module);
    MountTable mountTable = activeProject.getMountTable();
    logger.debug("mountTable: " + mountTable);
    boolean done = false;
    synchronized (this) {
        try{
            ModuleDescriptor moduleDescriptor = mountTable.mountModule(module);
            moduleDescriptor.setEditable(false);
            mountTable.loadModule(moduleDescriptor, new SimpleProgressStatus());
            mountTable.importModule(moduleDescriptor);
           done = true;
            loaded = true;
        }
…
```

Figure 33 ProjectListener *loadUweProfile*()

To load a profile into an opened project, first a *ProjectDescriptor* object has to be created using the URI of the profile file (*module*). Figure 33 shows a code fragment of

the *loadUweProfile()* method of the *ProjectListener* class. After the *module* is created, it has to be mounted to the *MountTable* of the opened project. By doing so, a new *ModuleDescriptor* object including the UWE profile file is created and can be loaded and imported into the active project in the application.

The mechanism of loading and collecting all loaded modules (profiles) gives the flexibility to check at any place in the plug-in whether the UWE profile is already loaded or not. Based on the UWE diagram the plug-in can automatically check whether the required profile file is loaded and to load it in case it is not yet loaded. Furthermore this feature prevents from exceptions and the user is not expected to load the UWE profile every time prior opening a diagram.

5.4.1.2 Hiding of the Class and Association Stereotypes

Another main issue of the implementation was the requirement to add default stereotypes to all created UWE elements on the diagram. In addition these default stereotypes have to be hidden. That means that the stereotypes are assigned to the model and presentation elements but are not shown in the GUI with the typical <<>> stereotypes brackets. Normally MagicDraw shows all stereotypes by default and the user can switch them off by setting the flag "*Show Stereotypes*" in the *Properties* window of every presentation element on a diagram. To open the Properties window of an element just select the element and press the combination of *Alt* and *Enter* keys.

MagicDraw Open API provides the opportunity to set any kind of predefined property of an element to some suitable value. The comprehensive set of properties is shown in Figure 34. Every property has two attributes: *id* and *value*. The property id identifies a specific property in the properties set. Every specific property has a *value* of specific type. For example value of *BooleanProperty* is *java.lang.Boolean*. The collections of properties are grouped by *PropertyManagers*. [16]

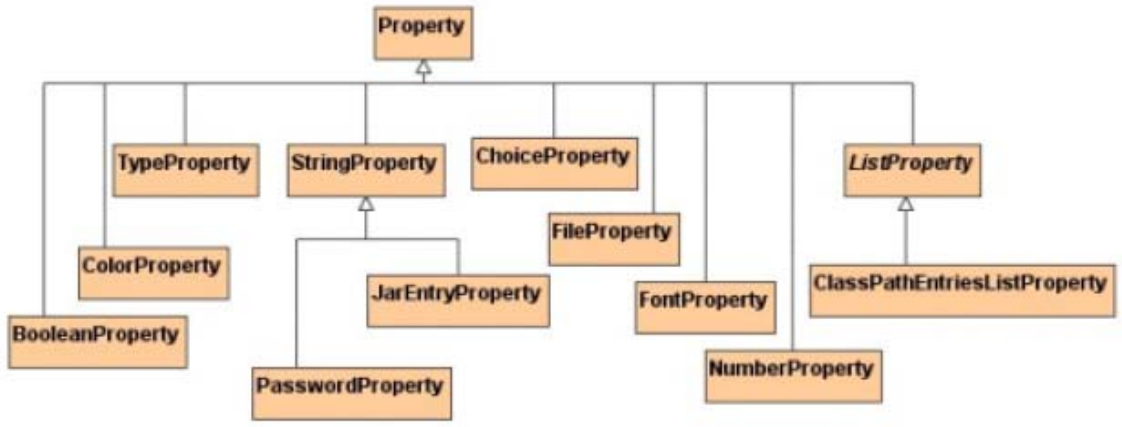

Figure 34 Properties Set

To achieve the task specified above the UWE Plug-in has to cover the stereotype manipulation on several places where presentation elements are created, modified of inspected. The first place is when creating a UWE element from the UWE Menu actions. These elements are with predefined specific UWE stereotypes which are corresponding to those stereotypes defined in the UWE Profile. Creating such a element can be done by using one of the following classes:

- de.lmu.ifi.pst.md.plugin.uwe.*actions*.*DrawClassAction*
- *de.lmu.ifi.pst.md.plugin.uwe.actions.DrawAssociationAction*

These classes are responsible for creating a shape element for custom UWE class elements (such as Navigation Class, Query, Index, etc) or path element for custom UWE association elements (such as Navigation Link, External Link, and Process Link). After that specific model element properties are set such as proper stereotype, properties to show class attributes and operations for classes, properties to set a navigable property for associations, etc. Once created, the element is immediately inspected by a *ProjectEventChangeListener* object. In its overwritten method *propertyChange(PropertyChangeEvent event)* all stereotypes of the newly created presentation element are hidden. Furthermore in this method also the default stereotypes of all presentation elements on any UWE diagram are created and hidden.

These above actions are taken when the event "*Selection Changed*" is passed. To minimize the overload of objects and to be able to show again switched off stereotypes references of once inspected objects are stored in collections. Next time when the same object of a presentation element is inspected no more actions (such as set default stereotype, hide stereotype, set navigable, etc) will be taken on this object. The concept behind this implementation is to permit the user to change the elements properties at any time after the default values were set.

There is one more place where element properties have to be set and this is when a transformation is invoked. In the *tranformOriginal(..)* method in the *de.lmu.ifi.pst.md.plugin.uwe.transformation.DiagramTransformator* class.

In Figure 35 a code fragment shows how the "*Show Stereotypes*" property of an association path element is set. Other properties are set in the same way.

```
…
//hide stereotypes of the presentation element
PropertyManager propertyManager = new PropertyManager();
propertyManager.addProperty(new BooleanProperty(PropertyID.SHOW_STEREOTYPE,
false));
presentationElementsManager.setPresentationElementProperties(newAssPath,
propertyManager);

…
```

Figure 35 Hide Stereotypes Property

At this point it is important to state that to be able to hide also the role stereotypes of an association element, first all its own presentation elements have to be collected. Then the hide stereotypes property has to be applied on its *RoleView* child presentation elements. How to retrieve all child presentation elements of an association is shown in Figure 36.

```
…
//need to hide also the role view
List<PresentationElement> childPresElements = assPresEl.getPresentationElements();
for(PresentationElement childPres: childPresElements){
    if(childPres instanceof RoleView){
        hideAllStereotypes(childPres);
    }
}
…
```

Figure 36 Hide Rowe View Stereotype

5.4.1.3 Usage of the State Actions

The most used type of custom action class in the UWE Plug-in is the *de.lmu.ifi.pst.md.plugin.uwe.actions.ClassDiagramToolbarAction*. This action class is a subclass of *DefaultDiagramStateAction* which in turn is a sub-subclass of *NMAction*. Unfortunately there is no information about the state actions in the documentation provided by the MagicDraw software. [15] Therefore the only resource to find some information about them is the Javadoc of the MagicDraw Open API.[17] In general state actions are used in MagicDraw for changing some Boolean state. In our case all *ClassDiagramToolbarActions* are packed together in view action groups. These groups are respectively corresponding to the defined packages in UWE metamodel:

- Navigation actions group
- Presentation actions group
- Process actions group
- UWE Use Case actions group

Furthermore these groups are added into the diagram toolbar menu, and actually all actions used in the diagram toolbar menus are such state actions. As already stated, a state action returns a true or false, depending on whether an action is selected or not. This behaviour of the actions is the reason why also the UWE custom diagram toolbar actions are structured in groups. Thus only one action of each group can be selected at the same time. Furthermore the action groups can have only actions in their first level submenu, it means there is no possibility to add another subgroup consisting of other kind of actions into the already existing state action group. Such nested subgroups (submenus) for example can be created in the main menu (see UWE main menu). Figure 37 shows such UWE state actions grouped in the *Navigation* action group.

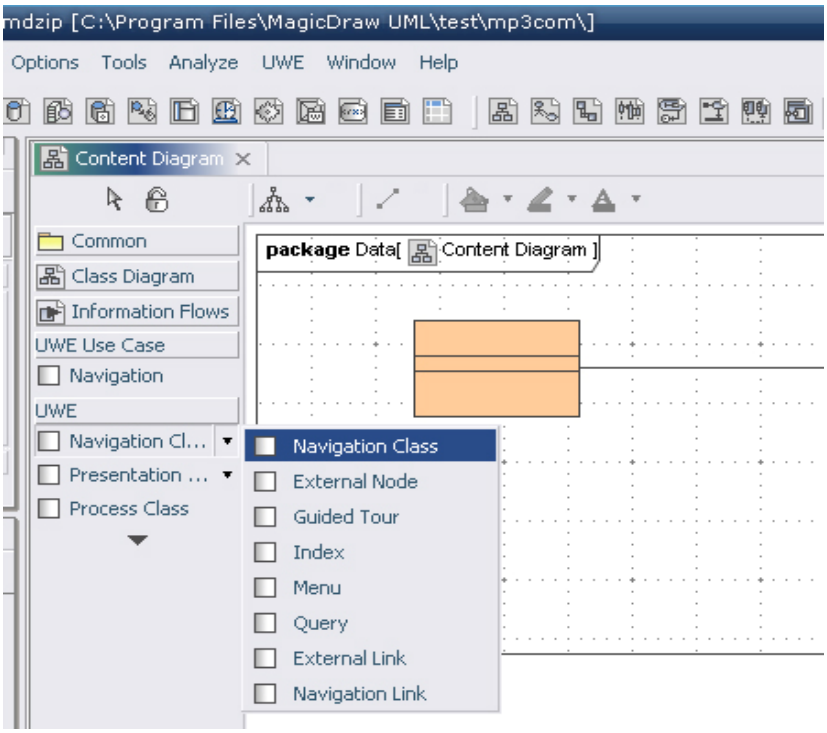

Figure 37 UWE State Actions

Actually the problem is not in the implementation of state actions itself, but to know and understand where what actions can be used. Furthermore it is crucial, to implement

such type of GUI elements (accurate actions) that make the Plug-in usable and efficient. The code fragment in Figure 38 shows how the described actions above are implemented in the *configure()* method of *ClassDiagramToolbarClassConfigurator* class. This configurator is created from the *PluginManagerClass* with collection of *ClassDiagramToolbarActions* stored in the *actions* class variable. As shown in the previous figure, first the UWE diagram toolbar configuration is created in case it does not exist yet.

```
//check if there is allready uwe toolbar menu existing if not create it
DiagramInnerToolbarConfiguration category =  (DiagramInnerToolbarConfiguration)
mngr.getCategory("UWE");
if(category== null){
    category= new DiagramInnerToolbarConfiguration("UWE", null, "UWE", true);
    category.setNested(true);
    mngr.addCategory(category);
}

ActionsCategory actionsCategory= new ActionsCategory("","");
actionsCategory.setNested(true);
category.addAction(actionsCategory, 0);

// add the actions
Iterator it = actions.iterator();
while(it.hasNext()){
    DefaultDiagramStateAction action = (DefaultDiagramStateAction)it.next();
    actionsCategory.addAction(action);
    logger.debug("action added: " + action);
}
```

Figure 38 State Actions Implementation

Then an actions category with no name is created and added to the newly created UWE configuration. This no name actions category is exactly the separation in groups discussed above, because for every diagram toolbar action group created in the *PluginManager* the *ClassDiagramToolbarClassConfigurator* is called, or respectively its *configure()* method.

Final step implemented in this code example is adding the concrete custom state actions objects into the no name *ActionGroup* passed from the *PluginManager* for each group done by iteration through all passed actions.

5.4.1.4 Transformation Package Browser

One of the features of the UWE Plug-in is that after transformation is executed a data and elements package browser window is shown, so the user can choose where the new transformed elements will be stored. This browser view is displayed in Figure 39. After selecting the target package the user will confirm with clicking on the “Select” button and all new created elements will be moved into the selected package. Of course only folders (packages) can be selected as a target, otherwise an error message will be displayed on the screen.

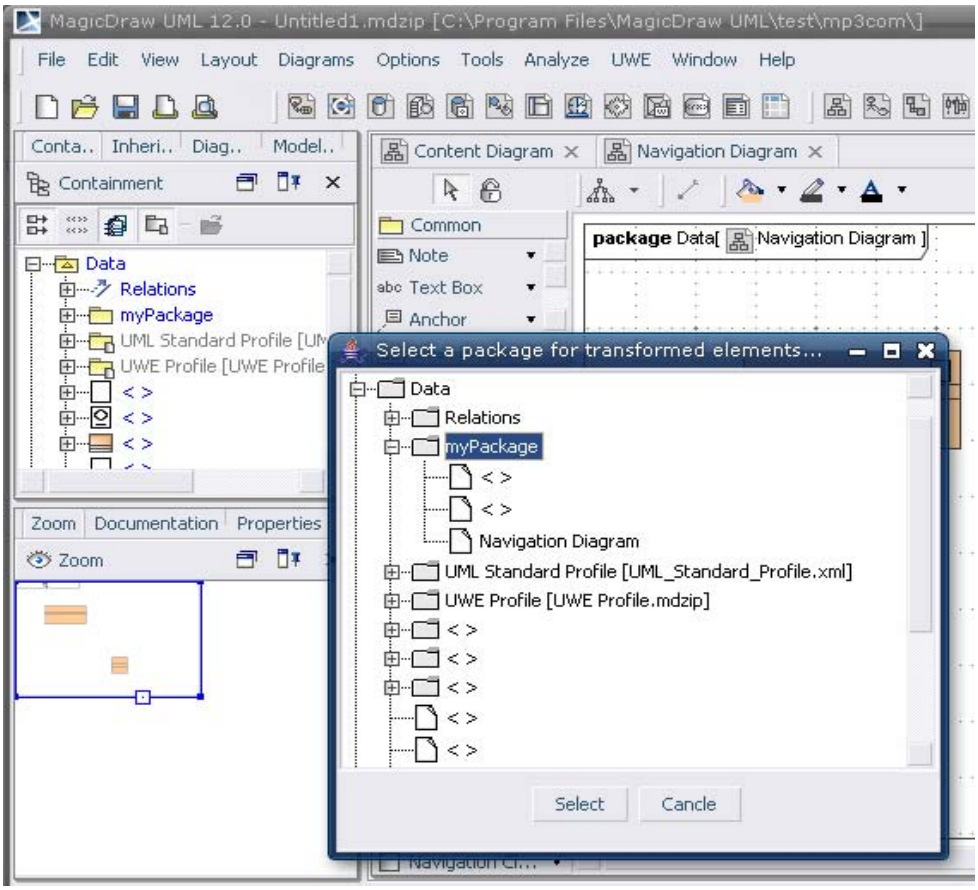

Figure 39 Elements Package Browser

```
private void setUpTheTreeBrowser(){
    //get the model browser
    Browser browser = Application.getInstance().getMainFrame().getBrowser();
    //get the active browser tree
    Tree activeTree = browser.getActiveTree();
    logger.debug("activeTree: " + activeTree);
    //Create a tree that allows one selection at a time.
    DefaultMutableTreeNode top = activeTree.getRootNode();
    tree = new JTree(top);
    tree.getSelectionModel().setSelectionMode (TreeSelectionModel.
SINGLE_TREE_SELECTION);
    logger.debug("tree:  " + tree);
   //Listen for when the selection changes.
   tree.addTreeSelectionListener(this);
   //Create the scroll pane and add the tree to it.
  JScrollPane treeView = new JScrollPane(tree);
  …
```

Figure 40 Elements Browser Window Implementation

The implementation is done in the *de.lmu.ifi.pst.md.plugin.uwe.actions. TransformatorAction* class. To display the actual structure of the elements tree as it is shown on the default containment tree window of MagicDraw the application active tree

of the *Browser* object has to be called. After that a *DefaultMutableTreeNode* is created and added into the custom tree object. Figure 40 shows this above implementation. Furthermore the class *TransformatorAction* implements the following listeners: *TreeSelectionListener, ActionListener*. That means that both of them have listener methods that have to be overwritten. In the first one, *valueChanged(TreeSelectionEvent event)* the actual selected node from the displayed elements tree is set. The second method, *actionPerformed(ActionEvent event)* is called when a button is clicked. Depending on the selected node on the tree before, all new transformed elements are moved into the package. If the selected node is not a package element a proper exception is thrown.

5.4.1.5 Inserting of Query and Index

Another important feature is the possibility to insert a *Query* or *Index* UWE element into an existing diagram considering some UWE metamodel rules. These functions are implemented into the diagram context menu under the menu item *UWE* shown in Figure 41. This item is active only when an element from the active diagram is selected, otherwise the user can not click on it. To use the automatic element insertion, an association of type Composition has to be selected first. After that either *Insert Query* or *Index* can be clicked and the selected association will be split into two parts and the new element will be added between them. The implementation of this functionality is made in the *DiagramContextAction* class. Objects of this class are created in the *PluginManager* while initializing the Plug-in. *DiagramContextAction* class extends the *DefaultDiagramAction* class and implements its *actionPerformed()* method.

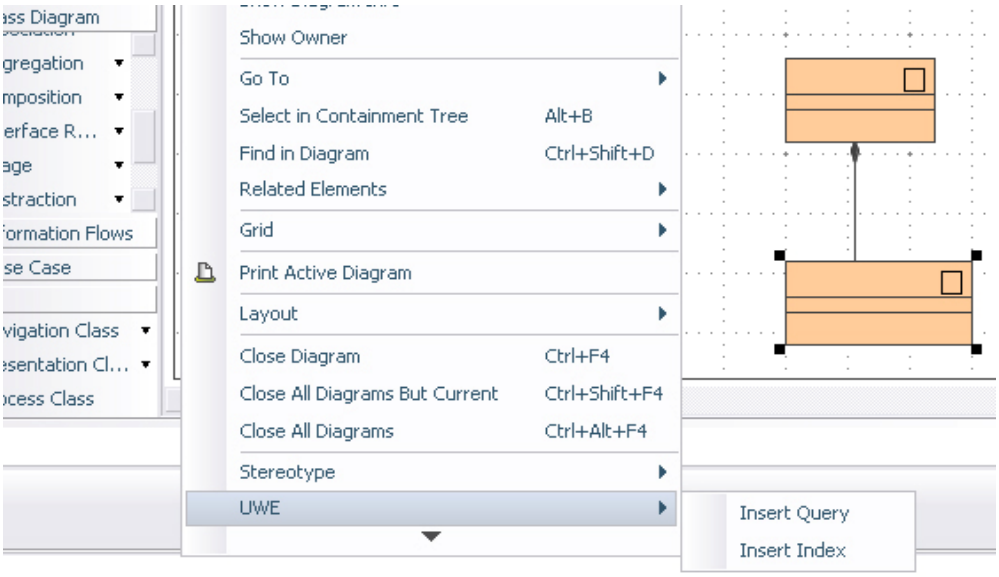

Figure 41 UWE Diagram Context Menu

This method is called every time when the actions *Insert Query* or *Index* are clicked. This method also contains all tests whether an element can be inserted in place of the actually selected presentation element from the diagram. After the test is performed the method *insertElement(…)* is called and the new model element with its presentation element is created and added into the actual diagram in the proper place.

Difficult moment in the implementation of this functionality was to place the newly inserted element in the proper place between the already existing classes of the split association. The current solution calculates the middle point of the association and

inserts exactly there the new element. The problem is that if the association is not long enough the newly inserted element would overlap with the already existing one. To avoid that, further optimization of the used algorithm is necessary. The described algorithm is shown in Figure 42. The whole implementation will not be described here due to its large size. Please inspect the *DiagramContextAction* class implementation for more details.

```
//set the proper display location in the niew diagram
Rectangle bounds = origSuplier.getBounds();
int x = 0;
if(origClient.getMiddlePointX() > origSuplier.getMiddlePointX()){
     x = origClient.getMiddlePointX() - ((origClient.getMiddlePointX()-origSuplier.
getMiddlePointX())/2);
}
else{
     x = origSuplier.getMiddlePointX() - ((origSuplier.getMiddlePointX() - origClient.
getMiddlePointX())/2);
}
bounds.x = x;
int y = 0;
if(origClient.getMiddlePointY() > origSuplier.getMiddlePointY()){
    y = origClient.getMiddlePointY() - ((origClient.getMiddlePointY()-origSuplier.
getMiddlePointY())/2);
}
else{
    y = origSuplier.getMiddlePointY() - ((origSuplier.getMiddlePointY() - origClient.
getMiddlePointY())/2);
}
bounds.y = y;
presentationElementsManager.reshapeShapeElement(newShape, bounds);
```

Figure 42 Computation of the Place of Insertion

6 Modelling with the UWE-Plug-in

This chapter describes how to install and use the UWE Plug-in for MagicDraw. After the installation instructions in the first section of this chapter a detailed modelling example with the Plug-in is discussed in the next sections. This example is based on an existing online web application. This example is structured in dedicated sections for each diagram type so that a better readability is ensured.

6.1 Preparations before Modelling

This section describes all necessary steps for using the UWE Plug-in in a project of MagicDraw. In general some knowledge of using the MagicDraw software is expected; otherwise a starting source of useful information can be the MagicDraw *User Manual* document. [18]

After successful installation of the Plug-in described in the previous section MagicDraw can be launched in the usual way. Now the UWE modelling functionalities are implemented in the core software and the UWE main menu is shown between other main menu items. To use these functionalities the UWE Profile has to be loaded into the opened project. This can be achieved through three different approaches:

- Starting a new project from a template and choosing the UWE template
- Starting a new default project, clicking on *"Use Module…"* from the "*File*" main menu and selecting the UWE Profile file from *profiles* directory
- Starting a new default project and clicking on one of the UWE diagram types from the "*UWE*" main menu. The UWE Profile will be loaded automatically into the opened project

Now the MagicDraw environment is ready to work with all defined UWE Stereotypes from the UWE Profile and these UWE elements can be referenced in any diagram from the project.

In the next section a concrete step by step example will be examined.

6.2 Modelling by Example

In this section a step by step example of modelling with the UWE plug-in will be described. The popular web application www.mp3.com is used as a source for the example. The page is simple and intuitive to use, its purpose is to bring easily songs from different singers to the user. The user can search songs by the song's title,

singer's name or album's title. Furthermore the user can view singers and songs grouped by their music genres. Songs, albums and singers can be also provided with comments by other users. The user than can read these comments (called reviews), can rate the songs, albums and signers, can vote for all these items, etc. Furthermore users who register via email and password can download songs or whole albums for free, or can buy some credit and download any kind of music and also video content from this page. There are other additional functionalities such as video clips, daily news and charts, podcasts, forums, photos of singers and bands, etc. that are not included in the example bellow.

Let us now focus on how such a website is modelled by the UWE approach starting with the UWE use case. As the functionalities of this webpage are numerous and complex, the example concentrates only on the main user and music content functionalities.

6.2.1 Use Case Diagram

In addition to the UML features, UWE distinguishes between three types of use cases: navigation, process and personalized use cases. In our example in Figure 43 Navigation use case diagram is shown. Navigation UWE use case diagram models typical user behaviour when interacting with the Web application.[19] Use cases can be e.g. searching content on the page, browsing through the Web page, registering and filling out forms, downloading some files, etc.

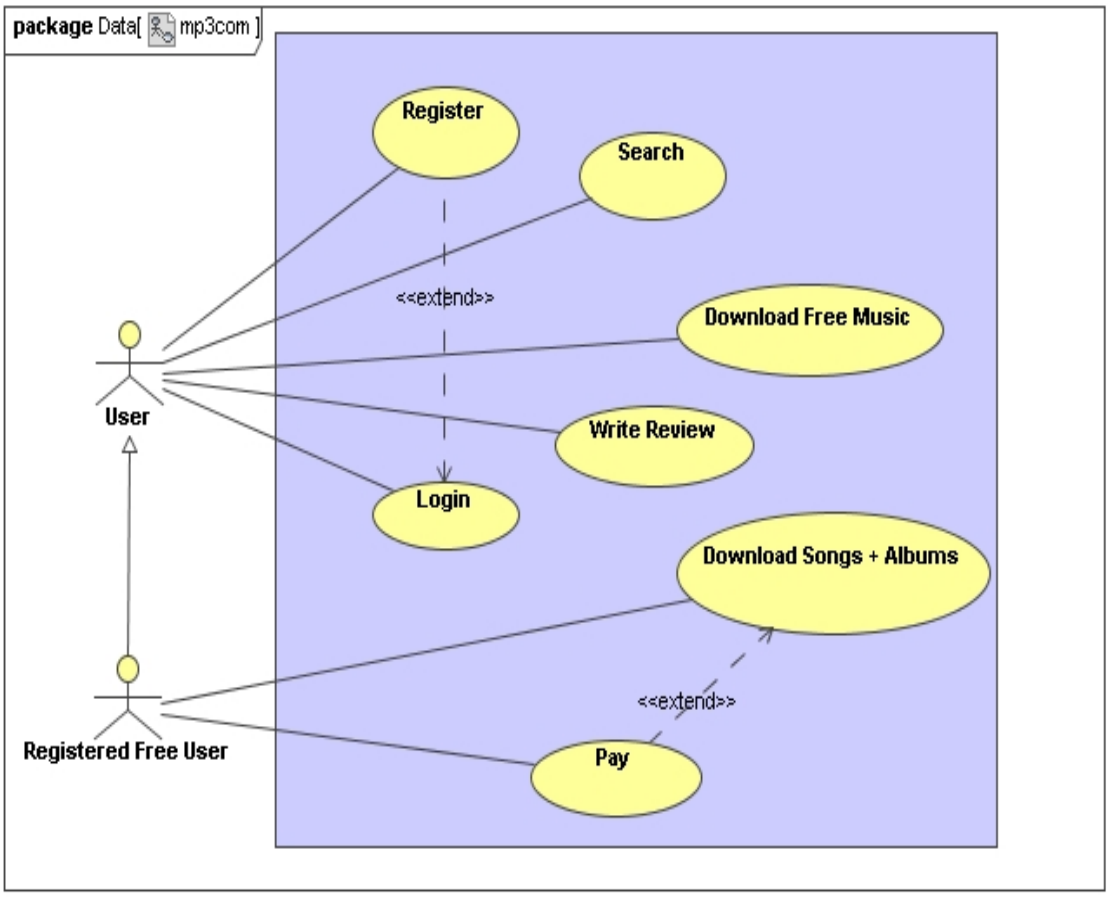

Figure 43 Navigation Use Case Diagram of the Example

To draw a UWE use case diagram a standard MagicDraw use case diagram has to be created. It can be started from the "*Diagrams*" main menu. Besides the default use case diagram toolbar menu items also the Plug-in "*UWE Use Case*" menu item is available from the diagram toolbar. This new menu item includes new UWE action called *Navigation*. Selecting the *Navigation* action from there, the user can draw use cases with assigned stereotype of type *Navigation*. The UWE Profile has to be loaded to assign the stereotype; otherwise a message will be displayed on the message window prompting the user to load the profile.

In Figure 43 – the Navigation use case model - also two actors on the diagram are displayed. The first one is the *User* actor, which represents a normal user browsing the www.mp3.com webpage. The second actor, *Registered Free User* extends the *User* actor and it represents a registered user of the page. There is one more type of user actor, a user that is registered and has loaded some money credit on his account. This last actor is unaccounted for the example discussed here. In this figure there are also all main use cases for both actors. *Registered Free User* inherits all use cases from the extended *User* actor and has some additional ones such as: *Pay*, *Download Songs and Albums*. The both actors can search for content on the webpage, can register or login when already registered, can write reviews of a song, singer or album, etc.

The next step of modelling by the UWE approach can be a more detailed use case diagram. This type of diagram will include notes with comments, more detailed use cases and relations. It depends on the complexity of the web application and project risk if some more detailed diagrams will be drawn. Very often a requirement specification based on use cases is not enough.[19]

6.2.2 Content Diagram

The aim of the content model is to provide a visual specification of the domain relevant information for the Web system that mainly comprises the content of the Web application. Figure 44 shows the content diagram of the example. Content diagram is graphically represented as UML class diagram.[20]

No UWE stereotypes are defined for the Content diagram elements, but the user can use any available stereotypes if he wants to point out a detail of the diagram. It shall be admitted, that only relevant elements can be transformed from *Contend* diagram into *Navigation* diagram when the semi-transformation action is launched. In particular only elements with no stereotype at all or with stereotypes *Navigation Class, External Link and Navigation Link* will be transformed automatically.

The example of the *Content* model diagram shows that an Artist has zero or more Videos, Photos or Albums. In turn an Album has Songs and both Albums and Songs can have Reviews.

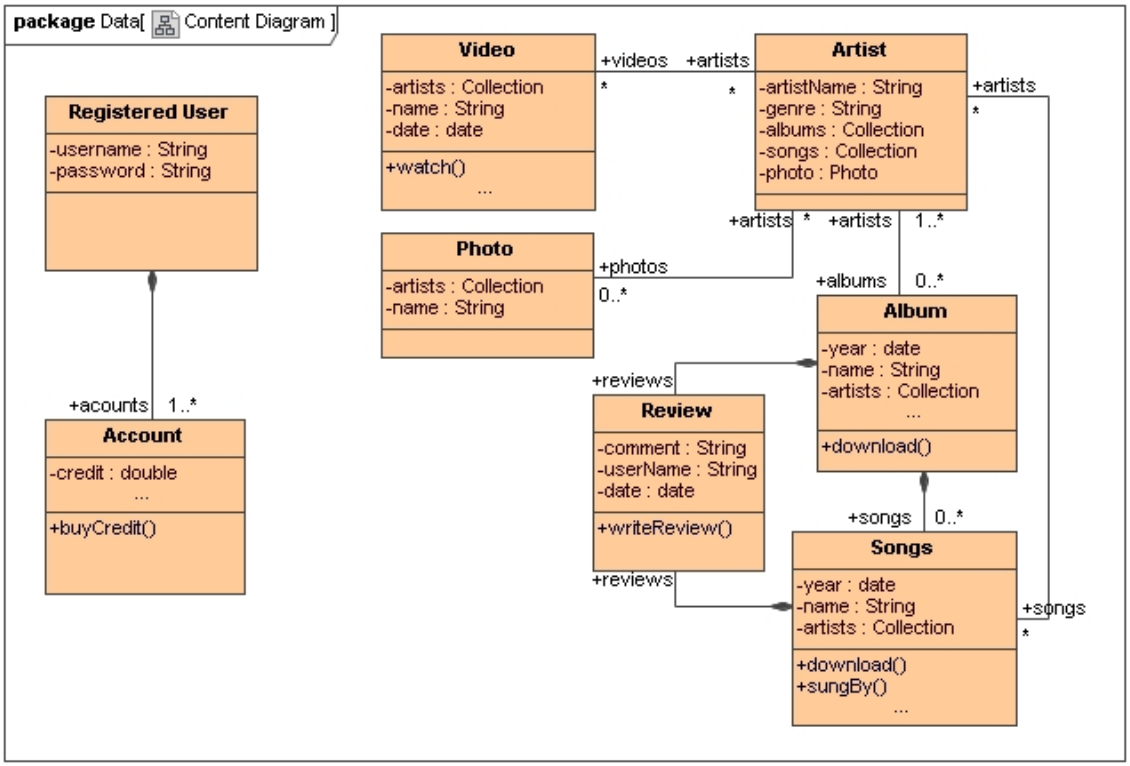

Figure 44 Content Diagram of the Example

6.2.3 Navigation Diagram

Based on the requirement analysis (use case diagram and other diagrams if necessary) and the content modelling, the *navigation structure* of a Web application is modelled. *Navigation* classes represent navigable nodes of the hypertext structure. *Navigation Links* showing direct links between Navigation classes and alternative navigation paths are handled by *Menus*. Access primitives are used to reach multiple instances of *Navigation Class: Index* or *Guided Tour,* or to select items: *Query*.[21]

All elements with assigned navigation diagram stereotypes are integrated into the MagicDraw diagram toolbar menu and can be directly used from there as shown in Figure 37 earlier. For producing a *Navigation* diagram from an already existing *Content* diagram the integrated semi-automated UWE Content-to-Navigation transformation can be used. The action for starting this transformation is located in the *UWE* main menu under the *Transformation* submenu item and is called *Content2Navigation*. After clicking on this action a new *Navigation* diagram from the actual Content model is created. All classes from the source *Content* model will be copied into the newly created *Navigation* diagram and will be automatically converted into *Navigation* Classes depending on their original stereotype.

The next step is to add *Menus*, *Indexes* and *Queries* into the navigation model. *Index* and *Queries* can be easily inserted by selecting a composite association between two classes and clicking with the right mouse button. From the displayed context menu the *UWE* item shall be chosen and the type of insertion shall be selected. After that the newly chosen element will be inserted between these two classes. See section 5.4.1.5 for more detailed information about the automated element insertion.

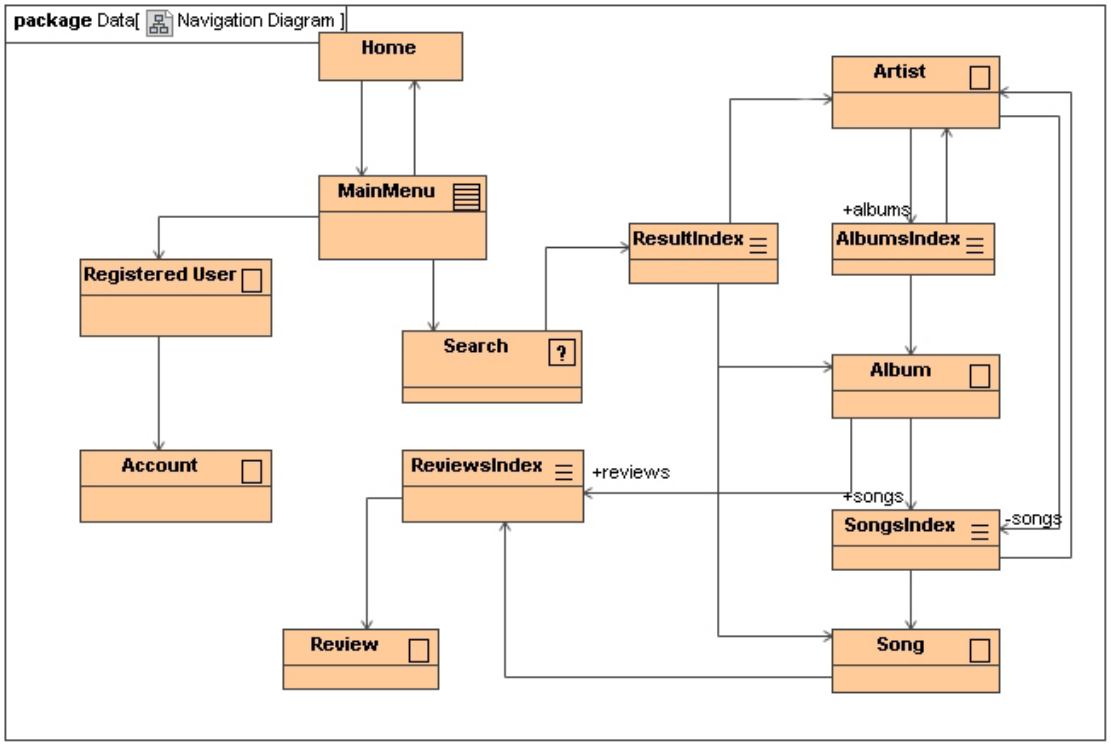

Figure 45 Navigation Diagram of the Example

There are numerous new navigation elements added in the *Navigation* diagram example shown in Figure 45 compared to the previous diagram of type *Content*. The most new elements are *Indexes* representing multiple instances of a *Navigation Class* such as: *AlbumsIndex* to *Album* or *SongsIndex* to *Song*. There is also the *MainMenu* of type *Menu* representing the navigation paths of the main menu of the webpage. There is also a *Query* element called *Search*, which represents the search field for any type of media content on this webpage.

There is also one very important element of type *Process Class* shown on the diagram called *Login*. In general, in web applications that contain business logic the business process must be integrated into the navigation structure. The entry and exit points of the business processes are modelled by *Process Class*.[21]

6.2.4 Navigation Diagram with Integrated Processes

In the next step, the navigation structure can now be extended with *Process* classes which represent the entry and exit points of business processes. These *Process* classes are derived from the non-navigational use cases.[21]

In the *Process* diagram shown in Figure 46 the business processes *Register*, *CreateProfile, Login, ManageMyFavorits, BuyIt* have been added. The integration of these classes into the navigation model is done via the main menu (*MainMenu*) which provides links to *Register, Login*, etc. The user can only create a profile if he has been registered and the same user can manage his favourites (songs, videos, photos, etc.) only after logging in first. Finally a user can buy songs or all albums by the process *BuyIt*.

Like the previous diagram type elements, all process elements can be accessed from the diagram toolbar menu called *UWE.*

A single Navigation structure and Process diagram for the whole Web application would inevitably lead to cognitive overload. Different views to the navigation structure with integration of related business processes should be produced. In turn each *Process* class of navigation model is refined into a process model.[22]

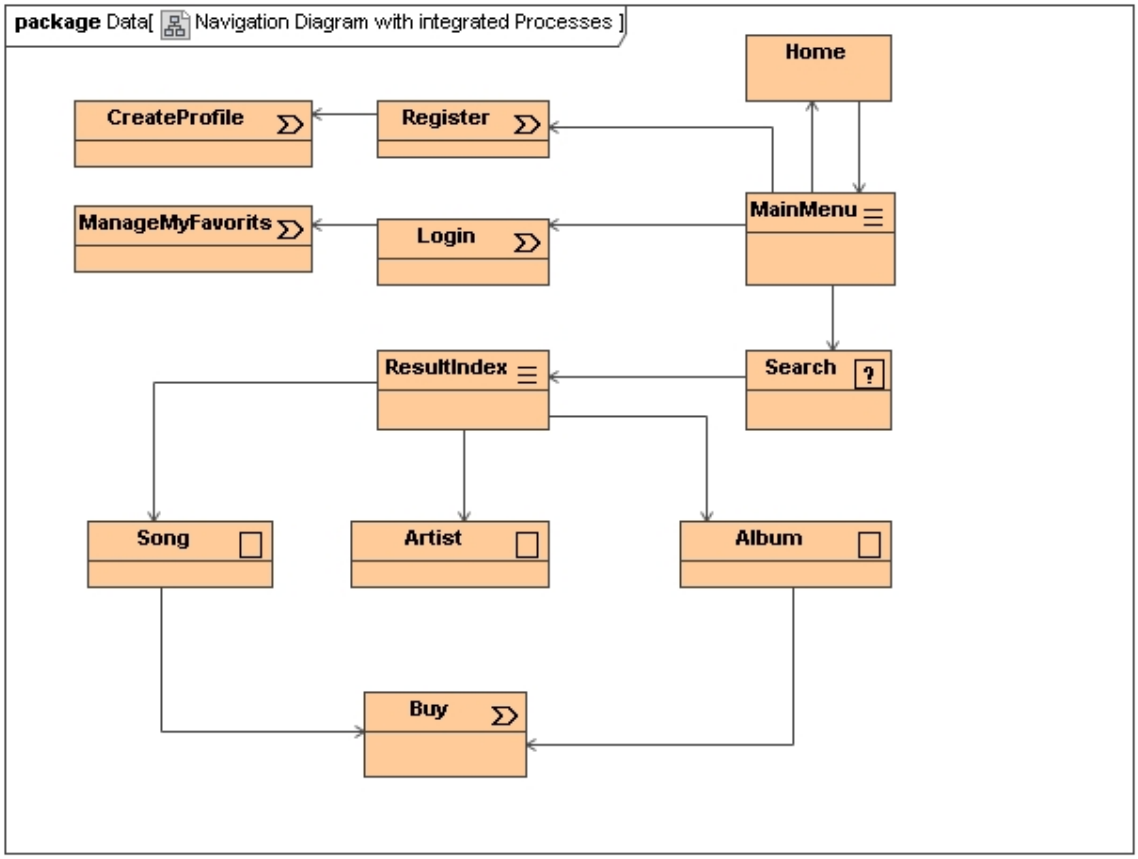

Figure 46 Process Integration into Navigation Diagram

6.2.5 Presentation Diagram

The presentation model provides an abstract view on the user interface (UI) of a Web application. It describes the basic structure of the UI, i.e. which UI elements are used to present the navigation nodes. Such UI elements can be: text, images, anchors, forms. The relationship between presentation classes and UI elements is of type composition.[23] Furthermore each attribute of navigation class is represented with an appropriate UI element. For example, a text element is used for the title attribute and an image element is used for the photo attribute.

The building process of a *Presentation* diagram from a *Navigation* diagram is similar to building a *Navigation* diagram from a *Content* diagram. It is triggered by choosing the main menu item *UWE,* then *Transformation* and finally the *Navigation2Presentation* action. All the *Navigation Classes* are copied and converted into the new *Presentation* diagram and their stereotypes are set to *Presentation Class*.

In the example described in this chapter, only one *Navigation* class will be represented with *Presentation* model, the rest of them are produced analogously. In Figure 47 the

Artist Navigation class is converted into its *Presentation* model. All class attributes now are represented with their own presentation elements and all of them are connected with association of type composition with their own presentation class, in our case the *Artist* class

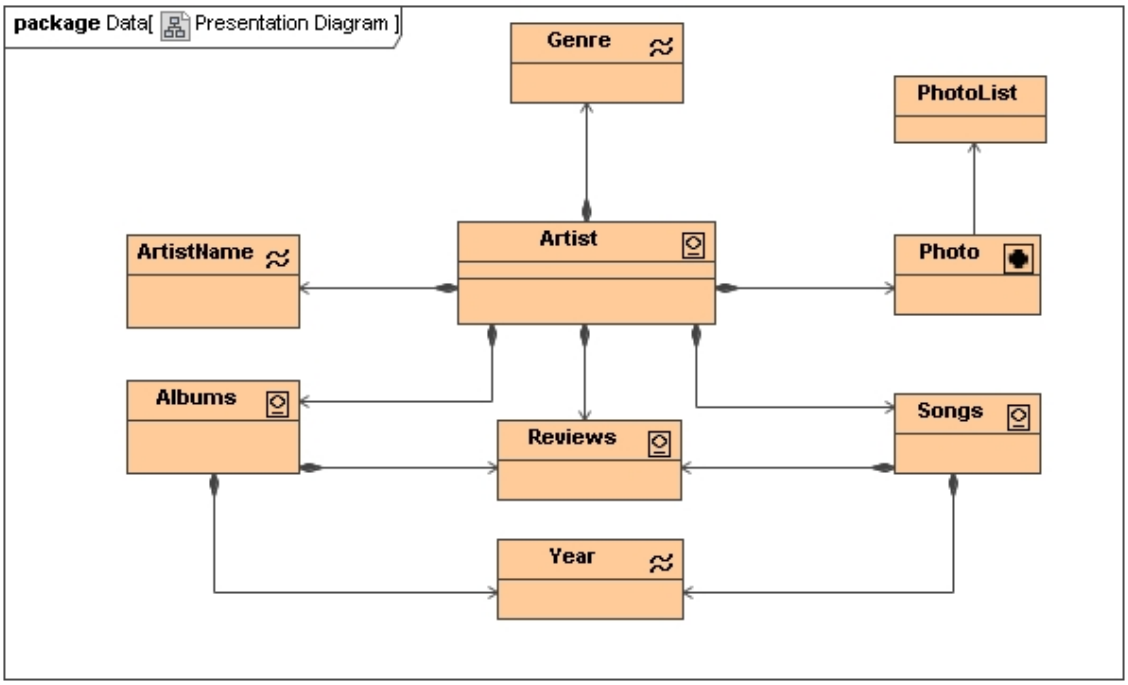

Figure 47 Presentation Model of Artist Navigation Node

6.2.6 Transformations

As described in the sections before the UWE Plug-in has a semi-automated transformation functionality. There are two types of transformations: from *Content* diagram into *Navigation* diagram (Content2Navigagtion action), and from *Navigation* Diagram into *Presentation* diagram (Navigation2Presentation action). The both actions are shown in Figure 48. Their implementation and functionalities were already discussed in section 5.3.3.2. This section describes when these transformations are beneficial to use in the UWE modelling process.

The UWE Plug-in is designed to affect minimally the work of the user during modelling. Furthermore the user is allowed to draw any kind of diagrams with any kind of elements. The UWE transformations are mainly usable when diagrams with a large amount of UWE elements are created, thus the process of producing the next UWE model will be much faster than drawing it manually. Here is to admit, that not all of the elements from the source diagram will be transformed into the new target diagram. All elements of the source diagram (the active diagram from where the transformation is launched) are examined during the transformation based on the rules shown on Table 2. All elements with assigned stereotypes different than those shown on Table 2 will not be considered and will remain only in the source diagram. For every considered element a new element will be created in the target diagram. The name of newly created element will be the same as the source element only its stereotype will be transformed.

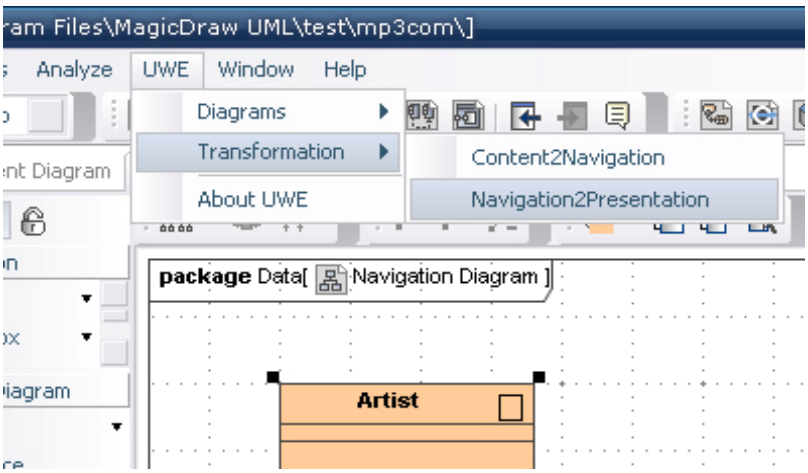

Figure 48 Transformation Actions

The association type of transformed association will be preserved, so if an association of type composition or aggregation is transformed the new association element in the target diagram will be of the same type. Furthermore the opposite end of the newly created association will be set to navigable.

	Content to Navigation	**Navigation to Presentation**
Stereotype of the elements that will be considered for transformation	- Without assigned Stereotype - Navigation Class - Navigation Link - External Link	- Navigation Class - Navigation Link - External Link - Presentation Class

Table 2 Transformation Rules

6.2.7 Exporting the Model

MagicDraw allows by default to export all created diagrams and models from a project. Selected elements, diagrams or the whole project can be exported in several formats.

Perhaps the most usable model exporting file format is the XML Metadata Interchange (XMI). MagicDraw allows model export to EMF based on UML 2.0 compatible XMI. It allows interchange of UML 2.0 models with popular Model Driven Architecture (MDA) tools such as AndroMDA, OpenArchitectureWare, and others.[@5]

Another way to export created diagrams in MagicDraw is to export them into an image files. There are several file formats that can be chosen for such an exported image e.g. *jpeg, png, svg, eps, emf* etc.

7 Future Work and Conclusion

7.1 Unresolved Issues

This section describes the main issues that have not been implemented yet in the current version 1.0b of the UWE Plug-in.

One of the important unimplemented tasks is the UWE model consistency check. This functionality has to check automatically the actual UWE diagram elements for model consistency with selected UWE metamodel rules. For activation of this checking mechanism a menu item action has to be placed on an adequate place into the MagicDraw GUI. After this action is activated, all elements and their relationships from the active diagram will be collected and inspected for consistency. If some of the elements are not correct, they should be displayed in red bounds and a meaningful message will be shown in the message window.

Another complex functionality to be implemented is the automatic code generation. After the modelling of a Web application is finished, the user should be able to launch an automatic code generation. It shall be examined what kind of technology shall be used. As the Plug-in itself is developed with Java implementing language, perhaps the most suitable technology to use is to combine Java Server Pages (jsp) with other useful frameworks.

Another interesting issue is the code refactoring and optimization. Here is to focus on the *ProjectEventChangeListener* class, particularly on the default stereotype assignment and how they are hidden from the diagram. The problem here is that these steps are made in several places and cannot be joined to one place, due to the session and events listener mechanisms of MagicDraw. The aim should be to find a point when to assign the default stereotypes and to make them invisible in the diagram and thus reduce the implementation complexity and improve the performance.

A list of resolved and unresolved implementation tasks can be found in the *ToDo* class of the UWE Plug-in sources.

7.2 Conclusion

The purpose of this diploma thesis was to develop a functional extension for the computer aided design and engineering of Web applications using UML-based Web Engineering (UWE) methodology. The UWE Plug-in is built as a flexible extension of the MagicDraw software based on the OpenAPI interface provided by MagicDraw version 12.0. The focus of this diploma thesis is to show how the UWE Plug-in was designed and implemented on one side to fulfil the UWE approach requirements, and on the other side to be integrated effectively into the MagicDraw environment.

The first chapter of the thesis presented the basics of the Web engineering and introduced the UWE methodology. Besides, already existing tools supporting the UWE

approach were presented and the main goals of the UWE Plug-in were briefly described.

A detailed overview of the UWE methodology was provided in chapter two. Afterwards, UWE CASE tool requirements were defined in chapter three, whereas a special attention was paid to the usability, GUI, and functionality requirements.

The last three chapters are the core part of this diploma thesis. This main part begins with the UWE Plug-in design decisions in chapter four, continues with the implementation of the Plug-in in chapter five, and finishes with the example using the UWE Plug-in in chapter six. In chapters four and five the aim was to stress the most important goals and to point out main issues and their solutions. Finally the Plug-in functionalities were explained in chapter six.

The UWE Plug-in for MagicDraw developed during this diploma thesis has built a significant foundation stone for the integration of UWE methodology into MagicDraw. However, further functionalities need to be implemented considering the UWE approach to complete the Plug-in. Those are briefly outlined in sub-chapter 7.1.

Furthermore the UWE approach is continuously evolving so the UWE Plug-in has to be continuously adapted and maintained.

Nevertheless the developed UWE Plug-in is ready to use.

References:

[1] Gottfried Vossen. *Unleashing Web 2.0: From Concepts to Creativity*. Morgan Kaufmann Publishers, ISBN: 978-0-12-374034-2, 2007

[2] Nora Koch and Andreas Kraus. *The expressive Power of UML-based Web Engineering*. Second Int. Worskhop on Web-oriented Software Technology (IWWOST´02), May 2002.

[3] Alexander Knapp, Nora Koch, Flavia Moser and Gefei Zhang. *ArgoUWE: A Case Tool for Web Applications*. First Int. Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE´03), Sept. 2003.

[4] Nora Koch and Andreas Kraus. *Towards a Common Metamodel for the Design of Web Applications*. Third Int. Conference on Web Engineering (ICWE´03), LNCS 2722, ©Springer Verlag,July 2003.

[5] Hubert Baumeister, Nora Koch, and Luis Mandel*. Toward a UML extension for hypermedia design*. In *UML´99 The Unified Modeling Language - Beyond the Standard, LNCS 1723*, Fort Collins,USA, 614-629, Springer Verlag, October 1999.

[6] Nora Koch and Andreas Kraus. *Towards a Common Metamodel for the Development of Web Applications*. In Juan Manuel Cueva Lovelle, Bernardo Martín González Rodríguez, Luis Joyanes Aguilar, José Emilio Labra Gayo, and María del Puerto Paule Ruíz, editors, *Proc. 3rd Int. Conf. Web Engineering (ICWE 2003)*, volume 2722 of *LNCS*, pages 497-506. Springer Verlag, 2003.

[7] Nora Koch and Andreas Kraus. *The expressive Power of UML-based Web Engineering*. In D. Schwabe, O. Pastor, G. Rossi, and L. Olsina, editors, *Second International Workshop on Web-oriented Software Technology (IWWOST02)*. CYTED, 105-119, June 2002.

[8] Sami Beydeda, Matthias Book, Volker Gruhn. *Model-Driven Software Development*. ISBN 354025613X, Springer Verlag, 2005

[9] Nora Koch, Alexander Knapp, Gefei Zhang, and Hubert Baumeister. *UML-Based Web Engineering: An Approach Based on Standards.* In Luis Olsina, Oscar Pastor, Gustavo Rossi, and Daniel Schwabe, editors, *Web Engineering: Modelling and Implementing Web Applications*, volume 12 of *Human-Computer Interaction Series*, Chapter 7. Springer-Verlag, 2007. To appear.

[10] Nora Koch, joint work with Gefei Zhang, Martin Wirsing, Andreas Kraus, Alexander Knapp, Rolf Hennicker, Hubert Baumeister. UML-Based Web Engineering. Presentation. Web Engineering Group, Ludwig-Maximilians-Universität München. Sevilla, 21.4.2005

[11] Alexander Knapp, Nora Koch, Flavia Moser and Gefei Zhang: *ArgoUWE: A Case Tool for Web Applications*. Ludwig-Maximilians-Universität München, Germany

[12] Alexander Knapp and Gefei Zhang. *Model Transformations for Integrating and Validating Web Applications Models*. In Heinrich C. Mayr and Ruth Breu, editors, *Proc. Modellierung 2006 (MOD'06)*, volume P-82 of *Lect. Notes Informatics*, pages 115-128. Gesellschaft für Informatik, 2006.

[13] Andreas Kraus, Alexander Knapp and Nora Koch. *Model-Driven Generation of Web Applications in UWE*. Proc. of 3rd International Workshop on Model-Driven Web Engineering, MDWE 2007, Como, Italy, July 17, 2007

[14] Koch, Nora and Kraus, Andreas (2003). *Towards a Common Metamodel for the Development of Web Applications*. In Lovelle, Juan Manuel Cueva, Rodriguez, Bernardo Martin Gonzalez, Aguilar, Luis Joyanes, Gayo, Jose Emilio Labra, and del Puerto Paule

Ruiz, Maria, editors, Proc. Int. Conf. Web Engineering (ICWE'03), volume 2722 of Lect. Notes Comp. Sci., pages 495-506. Springer, Berlin.

[15] No Magic, Inc. *Magic Draw OpenAPI UserGuide*. Magic Draw UML Version: 12.0, Documentation provided with the installation of MagicDraw software. 2006

[16] No Magic, Inc. Magic *Draw OpenAPI UserGuide*. Magic Draw UML Version: 12.0. Chapter 8, Properties. 2006

[17] No Magic, Inc., *Magic Draw OpenAPI Javadoc*. Magic Draw UML Version: 12.0, Javadoc provided with the installation of MagicDraw software. 2006

[18] No Magic, Inc., MagicDraw UserManual, Version 12.0, December 2006, Document provided with the installation of MagicDraw software. 2006

[19] Nora Koch, Alexander Knapp, Gefei Zhang, Hubert Baumeister: UML-Based Web Engineering, An Approach Based on Standards. In Luis Olsina, Oscar Pastor, Gustavo Rossi, and Daniel Schwabe, editors, *Web Engineering: Modelling and Implementing Web Applications*, volume 12 of *Human-Computer Interaction Series*, , Chapter 7, Section 2.1 Starting with Requirements Specification. Springer-Verlag, 2007.

[20] Nora Koch, Alexander Knapp, Gefei Zhang, Hubert Baumeister: UML-Based Web Engineering, An Approach Based on Standards. In Luis Olsina, Oscar Pastor, Gustavo Rossi, and Daniel Schwabe, editors, *Web Engineering: Modelling and Implementing Web Applications*, volume 12 of *Human-Computer Interaction Series*, , Chapter 7, Section 2.2 Defining the Content. Springer-Verlag, 2007.

[21] Nora Koch, Alexander Knapp, Gefei Zhang, Hubert Baumeister: UML-Based Web Engineering, An Approach Based on Standards. In Luis Olsina, Oscar Pastor, Gustavo Rossi, and Daniel Schwabe, editors, *Web Engineering: Modelling and Implementing Web Applications*, volume 12 of *Human-Computer Interaction Series*, , Chapter 7, Section 2.3 Laying Down the Navigation Structure, 2007.

[22] Nora Koch, Alexander Knapp, Gefei Zhang, Hubert Baumeister: UML-Based Web Engineering, An Approach Based on Standards. In Luis Olsina, Oscar Pastor, Gustavo Rossi, and Daniel Schwabe, editors, *Web Engineering: Modelling and Implementing Web Applications*, volume 12 of *Human-Computer Interaction Series*, , Chapter 7, Section 2.4 Refining the Process, 2007.

[23] Nora Koch, Alexander Knapp, Gefei Zhang, Hubert Baumeister: UML-Based Web Engineering, An Approach Based on Standards. In Luis Olsina, Oscar Pastor, Gustavo Rossi, and Daniel Schwabe, editors, *Web Engineering: Modelling and Implementing Web Applications*, volume 12 of *Human-Computer Interaction Series*, , Chapter 7, Section 2.5 Sketching the Presentation, 2007.

Web References:

[@1] Unified Modelling Language UML home page, http://www.uml.org/, 30.09.2007

[@2] UWE Approach project home page, http://www.pst.informatik.uni-muenchen.de/projekte/uwe/uwe.shtml 09.09.2007

[@3] Aspect Oriented Modelling home page, http://www.aspect-modeling.org/, 30.10.2007

[@4] OMG home page, http://www.omg.org/, 1.11.2007

[@5] MagicDraw home page, http://www.magicdraw.com/, 20.09.2007

[@6] ArgoUML home page, http://argouml.tigris.org/ , 25.09.2007

[@7] XML Metadata Interchange (XMI), http://www.omg.org/technology/documents/formal/xmi.htm, 25.09.2007

[@8] ArgoUWE project home page, http://www.pst.ifi.lmu.de/projekte/uwe/argouwe.shtml, 30.09.2007

[@9] MagicDraw, What is MagicDraw, http://www.magicdraw.com/main.php?ts=navig&NMSESSID=9f6735dacd552437406a88ad449d981d&cmd_show=1&menu=what_is&NMSESSID=9f6735dacd552437406a88ad449d981d, 30.09.2007

[@10] No Magic, http://www.nomagic.com/dispatcher.php , 30.09.2007

[@11] Jython (JPython) project home, http://www.jython.org/Project/index.html, 1.10.2007

[@12] Java project home, http://java.sun.com/, 1.10.2007

[@13] ArgoUWE project home, http://www.pst.informatik.uni-muenchen.de/projekte/uwe/argouwe.shtml, 1.10.2007

[@14] MagicDraw Newsgroups, http://www.magicdraw.com/main.php?ts=navig&NMSESSID=73b19c78d15e30ee67ceb737f8ac9fef&cmd_show=1&menu=newsgroups&NMSESSID=73b19c78d15e30ee67ceb737f8ac9fef, 5.10.2007

[@15] XML project home, http://www.w3.org/XML/, 30.10.2007

[@16] IzPack Java Installer, http://izpack.org/, 20.10.2007

Appendix

Installation of the UWE Plug-in

To be able to work with the UWE Plug-in all it components have to be installed to an existing MagicDraw application. Exactly to say, there are a couple of Plug-in files that have to be copied in the proper MagicDraw directories. One possibility to copy the necessary Plug-in files is to do it manually and to copy each of the files to the target MagicDraw directory from existing Plug-in binaries. This approach is very insufficient, because the user doesn't has to know anything about the MagicDraw plug-in file structure and it will cause only unnecessary overhead for installing the UWE Plug-in. For that reason an automatic installer is used. At the end only one file with *jar* extension is produced which comprises all necessary plug-in files. The installation of the UWE Plug-in can be done either by double clicking on this file or true command line prompt. Furthermore to be able to install the Plug-in only a Java runtime environment is required.

Used open source application installer is called *IzPack*.[@16] *IzPack* is installer generator for Java platform. It is designed around modularity and flexibility and is configurable through xml based configuration files called *install.xml*. In this file it could be defined how the installer will looks like, what components from the application will be installed by default and what components (for example source codes, etc.) the user can choose to install optional. After configuring and compiling the installer a single installation jar file is created. This installation file can be launched through double click or from the shell using the java command. For understanding how to configure and compile custom application installer please refer to the documentation file provided with *IzPack* or check the homepage. In general *IzPack* is listed under the terms of liberal open source Apache Software Licence version 2.0.

The UWE installation file is called *UWE_MD_Plugin_Installer.jar*. To be able to launch the installation process a Java Virtual Machine has to be already installed on the operating system. It is also necessary to install MagicDraw before launching the Plug-in installation, because the user will be prompt to choose the MagicDraw installation directory during the installation process. The Plug-in was made and tested for MagicDraw version 12.0. It should work properly with all later versions of the software. After launching the installation file just follow the instructions displayed on the screen. Once the Plug-in is installed it can be also easy uninstalled by clicking on the *uninstallUWEPlugin.jar* created in the MagicDraw directory after the installation process is finished. In Figure 49 one of the dialogs during the installation process is shown. Concrete at this step the user can choose if some do not required program files will be also installed or not. The *"Base"* package includes all necessary Plug-in files to work with it and can't be deselected.

Figure 49 UWE Installer - Packages Dialog

Directories and Files Created during the Installation

This section describes which files and directories are created during the installation process of the MagicDraw Plug-in.

During the installation process all necessary files are copied into appropriate target MagicDraw directory and subdirectories. This directory structure has to retain unchanged, otherwise the Plug-in or some of it parts will not work correctly. The files and their structure are included in the installation jar file and from there are copied into the following directories:

- MAGIC_DRAW_HOME/plugins/uweMDPlugin
- MAGIC_DRAW_HOME/profiles
- MAGIC_DRAW_HOME/templates/UWE
- MAGIC_DRAW_HOME/Uninstaller

The files copied into the specified directories above are included in Table 3. Some of the UWE custom directories are created during the installation if they do not exist yet. Otherwise only the UWE files are overwritten and all other already existing files will remain unchanged in the target directories. After launching the *uninstallUWEPlugin.jar* all described UWE files in the previous table will be removed and the previous MagicDraw state will be restored. The optional subfolders *doc* and *src* in the *uweMDPlugin* directory are filled only if their checkboxes during the installation dialog shown in Figure 49 are checked.

Directory Name	Included Files and Subdirectories
plugins/uweMDPlugin	plugin.xml, uweMDPlugin.jar, Licence.txt, (optional subfolders: doc, src)
profiles	UWE Profile.mdr, UWE Profile.mdzip, UWE Profile.xml
templates/*UWE*	description.html, UWE.html, UWE.mdr, UWE.mdzip
Uninstaller	uninstallUWEPlugin.jar

Table 3 UWE Plug-in Files and Directories

The UWE Plug-in installation file can be downloaded from the following URL:

www.cip.ifi.lmu.de/~blagoev/UWE_MD_Plugin_1.0b_Installer.jar