Masterarbeit

Reformation of a MDSD tool chain by applying modern evaluation methods for solution comparison

Robert Wenger, BSc

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Begutachter: Dipl.-Ing. Dr. techn. Christian Kreiner
Betreuer: Dipl.-Ing. Dr. techn. Michael Thonhauser

Graz, im Mai 2013

Kurzfassung

Verschiedene Techniken der modernen Softwareentwicklung erhöhen sogenannte „Nichtfunktionelle“ Anforderungen von Software Produkten. Bei diesen Anforderungen handelt es sich unter anderem um leichte Erweiterbarkeit, leichte Anpassbarkeit und leichte Wartbarkeit. Eine Methodik, diese Punkte zu erfüllen, ist die Modellgetriebene Softwareentwicklung. Es existieren sehr viele Ansätze und fertige Teillösungen für Umsetzungen in der Praxis, jedoch noch kein vollständiges allgemeines Paket.

Die Firma *Salomon Automation* mit Sitz in Friesach bei Graz implementierte vor einiger Zeit eine Kette von Software-Werkzeugen, mit deren Hilfe es gelang, Teile ihres Produktes *WAMAS* automatisch zu generieren. Dabei werden sowohl Code (verschiedene Java-Versionen), als auch Definitionen für z.B. Hibernate erzeugt. Zusätzlich wird eine vollständige Dokumentation aller Modelle und deren Zusammenhänge mitgeneriert. Somit können Anpassungen und Erweiterungen in der Funktionalität des Programms über Modelländerungen umgesetzt werden. Der teilautomatisierte Prozess besitzt jedoch einiges an Verbesserungspotential. Er soll durch Adaptionen der Generatoren, eine modernere Modelldefinitonssprache und zusätzliche unterstützende Funktionalitäten, wie Team-Unterstützung bei Modelldateiänderungen, auf aktuellen Stand gebracht werden.

Ziel dieser Arbeit ist zuerst die Aufnahme des aktuellen Standes, inklusive aller Arbeitsschritte. Die sich daraus ergebenden Problematiken und Verbesserungsmöglichkeiten werden anschließend mit Hilfe von Methodiken zur Findung von klar definierten Anforderungen untersucht. Die gefundenen Punkte stellen einerseits ein Anforderungsprofil dar, mit dessen Hilfe bereits potentielle Kandidaten einer Lösung gefunden werden können. Die Methodik zur Findung dieser „Knock-Out-Kriterien“ liefert andererseits auch noch Bewertungspunkte, mit denen die unterschiedlichen Kandidaten untereinander verglichen werden können. Diese Arbeit beinhaltet ebenfalls die Beschreibung dieser Methodik zur formalen Findung der bestmöglichen Variante, passend zu den Anforderungen dieser Aufgabenstellung.

Abstract

Modern software development contains different technologies for improving so called non-functional requirements of software products. Some examples for this points are expandability, adaptability and maintainability. One possible way to implement a process for improving those points is called "Model Driven Software Development" (*MDSD*), although there is no complete tool chain for implementing *MDSD* in a general way. Fortunately many tools exist, which are able to automate some parts of the whole process.

Some time ago the company of *Salomon Automation* located in Friesach near Graz implemented a working tool chain to generate parts of their main software product *WAMAS* in an automatic way. As an output, this tool chain produces software code (different Java versions) as well as definition files e.g. like Hibernate files. Additionally it generates a full documentation of all models and their context. Therefore it is possible to realize adaption or modification of the program's functionality by simply changing models. Nevertheless, this semiautomatic process has a high potential of improvement. By adapting the generators, implementing a modern model definition language and adding features for team support the current tool chain should be brought up to date.

The first goal of this thesis is the documentation of the current working processes containing all necessary steps. The resulting problems and points of improvement then act as an input for a following requirement finding process. On the one hand the found points are a profile for possible candidates for a solution. On the other hand the method of finding those points also delivers criteria for comparing the candidates among themselves. The second part of this thesis contains a description of this methods for formally finding the best solution to this problem definition.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....

date (signature)

Credits

This master thesis has been carried out at the Institute for Technical Informatics, Graz University of Technology. At this point, I would like to thank my supervisors Christian Kreiner and Michael Thonhauser for their time and great support. Their advices and support helped me staying on the right track during the whole work. Especially, I want to thank Manuela: without her support, confidence and love this thesis would not have been possible.

Graz, in May 2013 Robert Wenger

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and objectives

The company of *Salomon Automation*, located in Friesach near to Graz, creates software and automation systems for intralogistic issues. Their main software product is called *WAMAS*®, which is short for Warehouse Management System. It uses self-contained modular parts of software in its system, i.e. a database as a reference to the real world's warehouse. A more efficient way to adapt these parts for each customer is to automatically generate the code out of description models, rather to implement it manually. To accomplish this task, a tool chain has been developed, which takes UML 1.4 class and state diagrams as inputs and delivers program code and documentation files of the customized parts as output.

The first main goal of this document is a description of the current system, defining the whole workflow. All necessary steps for creating and modifying model files as well as managing the product WAMAS with the customer projects should be documented. The tool chain currently consists of workarounds and inefficient process parts, which have a high potential of improvement. Also some features can be considered, which should support all development teams working on the product and the projects. Therefore the collected information should provide a base for a further requirements finding process for a new tool chain.

A requirements finding process should be defined as a second goal of this thesis. The intention is to find methods for an objective description of the problem to point out the important issues and define them as requirements for a new system. Based on that an evaluation method should be described for comparing different candidates for a solution.

As a third goal, new systems and different approaches to a new tool chain should be tested and described. The results of these tests should contain descriptions of all relevant properties for a comparison, like included functionality, effort for implementation and prices. As a result this document should provide all relevant information for a later decision making process targeting a definition of a new tool chain.

1.2 Outline

Chapter 2 is dedicated to related work and theoretical information, concerning this master's thesis issue. In the first part of the related work, an introduction is given to *Model Driven Software Development* (MDSD), the idea of *Repository Systems* including a brief description of how they work is shown and also some problems concerning a merge of model files are mentioned. These points are important to understand the issues belonging to the current tool chain.

The second part of the related work is dedicated to methods for finding requirements, aiming the problems of the current situation. It also shows up methods for evaluating the results of possible solutions.

The next Chapter 3 describes the current system of the tool chain in a very detailed way. It should give an overview of the current situation, used technologies, used software and file formats. It results in a workflow which is shown as obviously improvable. The optimization of it is probably the main criteria for a new system to be called "better". Therefore it is important to have a very detailed description of the current one.

To this point this thesis will have dealt with theoretical work and a description of the current environment. Chapter 4 shows the first practical work. It describes the chosen methods for finding requirements and evaluating possible solutions. It also shows the practical implementation based on the current situation at *Salomon Automation*.

The following Chapter 5 is giving the detailed information over the currently on the market available technologies. Additionally, the different approaches are evaluated and compared to each other.

Chapter 2

Related work

This chapter is dedicated to technologies used by the current tool chain. It also mentions and explains technologies, which are related to new possible solutions.

2.1 Related technologies of the current and future tool chains

2.1.1 Model driven software development

Modern software development has a huge set of requirements to fulfill. Not only the production of a software is in focus anymore, but also modularity, ability for fast adoptions and changes, documentation and many more properties have grown in importance. Constantly changing implementation technologies also play a major role.

To achieve this goal, a well known concept has been reactivated: the usage of models. The resulting development concept is called *Model driven software development* (*MDSD*). Figure 2.1 shows a graphical description of the idea behind MDSD.

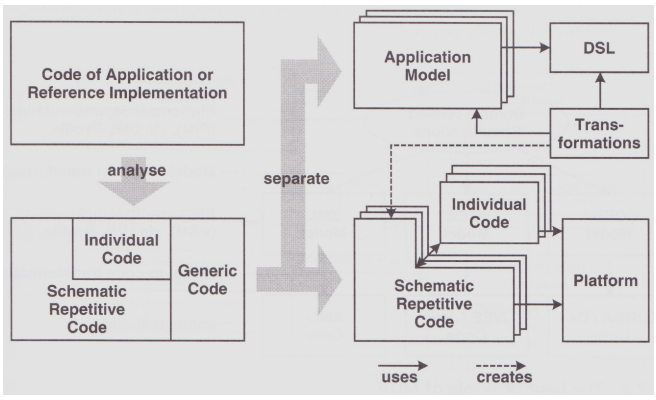

Figure 2.1: The basic idea of MDSD [Lad03]

According to [SV06], MDSD heads out to achieve following goals:

- *Development speed* by using code generation from formal models.

- *Software quality* by using transformations and formally-defined modeling languages. The software architecture will recur uniformly in the implementation.
- By using transformation steps and generators, a so called *separation of concerns* [Lad03] is automatically applied, as aspects, which cannot be easily implemented in one single module, can be specified through the language as one single information.
- A higher level of *reusability* can be achieved, as it is easier to adapt models and maybe their generators for a new purpose, than adapting the produced code.
- The language of the model description can also be domain specific. This allows an *improved manageability of complexity through abstraction.*
- Usually models can be transformed between different languages and systems. Also generators for different platforms and systems can use one and the same model as inputs. Thus, *interoperability* and *portability* increases significantly.

In the year 2000 an association called *Object Management Group* (OMG) introduced an approach to MDSD called *Model Driven Architecture* (MDA) [StOSSG00]. To work out this approach the OMG saw the necessity of defining standards, like the *Unified Modeling Language* (*UML*) or the *MetaObject Facility* (*MOF*). Especially the UML standard in the version of 2.0 or higher is a well known and popular language to describe technical problems and situations. Figure 2.2 shows an example of a UML2 *Sequence Diagram*, which describes the flow and interactions of different modules of a software startup code.

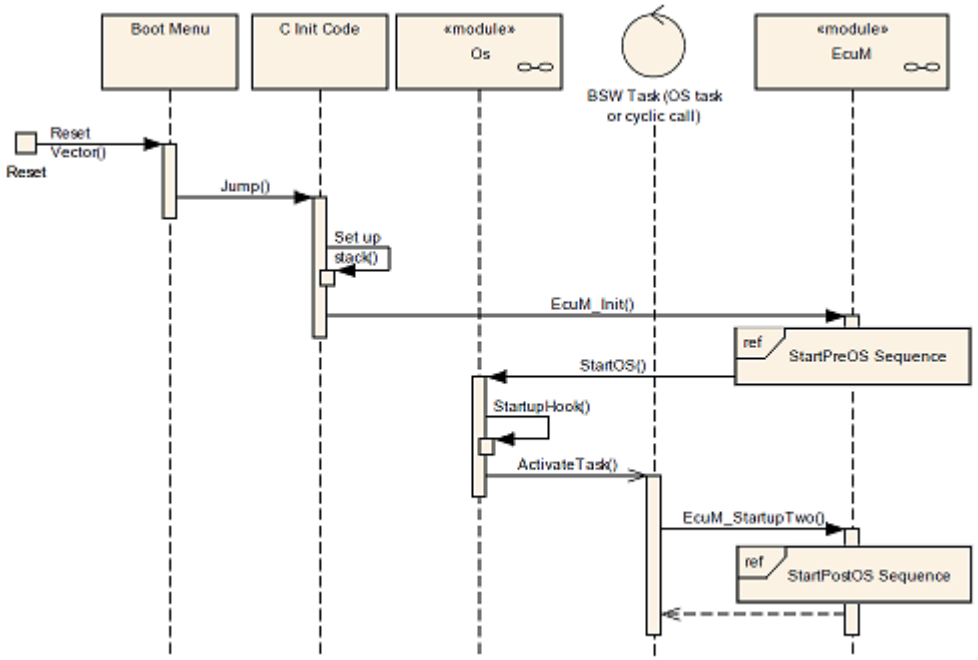

Figure 2.2: Example of a UML diagram [AUT11]

2.1.2 Repository systems

During software development many users work on the same project and on the same amount of files. Usually a good software design leads to modularity and clearly defined constraints, but unfortunately a final design mostly is found during the development process. Especially agile methods support fast changes in functionality, therefore also fast changes in the design must be supported in the development process.

Repository systems are intended to share files between different users and provide mechanisms to change those information. This is a very important feature, which can lead to much higher efficiency. It allows different developers to make changes, provide them to others and synchronize the whole project.

Most repository systems have a common server. At this central point the whole project including the files, the different versions and all meta information is located. To work on the project, a client software to connect to the server is necessary. This client software then is able to manage and synchronize all relevant information.

For the sake of completeness, a second strategy for repository systems has to be mentioned at this point. Differently to a central version storage system, most of the synchronization, history and other managing commands apply locally, therefore the systems are much faster and do not need a network. Only explicit synchronization with the server repository needs network communication [HP11].

Obviously, the realization of a repository system shows up problems, which base on concurrent changes and/or simultaneous changes. The following subchapters describe some mechanisms for avoiding or solving these problems. Usually these mechanisms or similar can be found in every software tool, which calls itself a repository system.

Locking

A very simple and kind of obvious solution for the problem of changes in the same file by two or more developers at the same time is known as *Locking*. It uses the concept of "first come, first serve". The first one, who needs the specific file to change, can lock the file for everybody else. Therefore, simultaneously taken changes cannot occur, since the file is not unlocked by the owning user.

Compare and merge

The idea behind the merging strategy is to have two different versions of the same file including changes of two developers and compare them textually. The comparison contains three different possible regions: equal text sections, differing text sections and sections, which are available in one version but are missing in the other one. The last two possible regions are called conflicts. The two involved developers have then to resolve this

conflicts, which means to take regions from one file and copy it to the other one. If all conflicts are solved, the two files are equal and one of them can be checked in as a new version.

As written in [BCE+06], "a number of factors make model merging complicated". If both files are seen as two quantities of information (e.g. Q_1 and Q_2), a mathematical notation can be used to describe the quantity of the resulting conflicts $Q_{Conflict}$ as in Equation 2.1.

$$Q_{Conflict}(Q_1, Q_2) = (Q_1 \cup Q_2) \setminus (Q_1 \cap Q_2) \tag{2.1}$$

3-way merge

A so called *3-way merge* is a more complex version of the normal comparing and merging of two files. The algorithm uses information for solving conflicts automatically. To achieve this goal it uses an additional file, the *common ancestor*. This file is the version, both other files are originated from. If the first file is called Q_1, the second file is Q_2 and the common ancestor file is named Q_0, then the following Equation 2.2 shows the quantity Q_{Fail}, where the automatic resolving algorithm fails.

$$Q_{Fail}(Q_0, Q_1, Q_2) = (Q_1 \cup Q_2) \setminus (Q_0 \cap Q_1) \setminus (Q_0 \cap Q_2) \cup (Q_0 \cap Q_1 \cap Q_2) \tag{2.2}$$

The advantage of this algorithm compared to the 2-way merge is the additional information we get in the case, a section of one new file that differs to the section in the second file, but equals to the section in the common ancestor file. In this case, the section in the differing file has to be a wanted change.

Equation 2.2 shows the single case, in which a section differs in all three files. Obviously, in this case both developers must have changed the same region; the algorithm is not able to decide, which change is the right one. This conflict must be resolved manually by the involved persons.

2.1.3 Model merging with XMI files

Chapter 2.1.2 highlighted, why merging of files is a very likely action in modern software development. This is not only related to source files, but also to all kind of files, which are used nowadays. As a rule of thumb, we can say, that every file, which will be changed during the development process, has eventually to be merged with a different version of its own.

Referencing to Chapter 2.1.1, modern software development deals with files representing models. The files containing these models can be of different kinds of formats. One very common format is *XML Metadata Interchange* (*XMI*), invented and defined by the OMG for interchange of objects on basis of meta-meta-models after the MOF. Additionally also other formats like *Domain Specific Language*s (*DSL*s) can be used for storing models. As the name implies, these languages have to be designed for the domain of the

specific problem and have therefore a very low reusability.

Unfortunately, some differencing problems respectively merge problems come up by using an XML format for saving models. First of all, XMI is not row oriented. In other words, the position of an opening tag after a closing tag is not necessarily defined. It can be either follow directly after the last character of the closing tag or start in the next line. Therefore both structures of the same information shown in Figure 2.3 are allowed.

```
<Test>
    <data1></data1>
    <data2></data2>
</Test>
```

```
<Test><data1></data1><data2></data2></Test>
```

Figure 2.3: The same information represented in four rows and one row

Also, the definition of XMI includes the permission of different orders of the same information on the same level inside a file. As a consequence, different ordered lines in files are able to represent the exactly same model. Figure 2.4 shows an example of this situation: Both sides represent the same information, but differ in their line-order. This example also covers the case of properties with a higher multiplicity than one. If the order of different values of elements with the same name within the same level is varying, a line based comparison tool will detect differences, even if there are none.

```
<Test>
    <data1></data1>
    <data2></data2>
</Test>
```

```
<Test>
    <data2></data2>
    <data1></data1>
</Test>
```

Figure 2.4: The same information represented in a different row order

2.2 Methods for defining requirements and evaluating results

During the development of a new system, defining requirements can be a hard and difficult process. Some points can be obvious, but mostly some points come up after the official requirement definition process. In this case, the project development is forced to step back or even to restart. Therefore the requirement finding process has a high significance in the whole project realization.

Another problem regularly rises after finding the requirements. Some points are necessary, where else other points only describe *nice-to-have*-features. If more than one possible

solution fits all *knock-out-criteria*, the decision has to be made by comparing the fittings of the features. In reality this situation is very often used to play political power games, which has nothing to do with finding the best solution to a problem.

The following subsections describe on the one hand some different methods for finding knock-out-criteria. On the other hand also methods for impartially comparing properties of different solutions are specified. Finally, the ideas of those methods are collected together and reused in a process, which is applied to the problem definition of this master's thesis.

2.2.1 4+1 view model

As mentioned before, the process of finding requirements is crucial for a fast and constructive course of a project. The goal of the process is to get a compilation of requirements.

For finding requirements, the target system first was defined with an architectural view model called "4+1 view model" after Kruchten [Kru95]. The idea behind is to define a system by four views, each owned by a different stakeholder. The architecture is represented with a single view for each stakeholder, so he can easily access the information he is interested in. A graphical representation of the view model is shown in Figure 2.5.

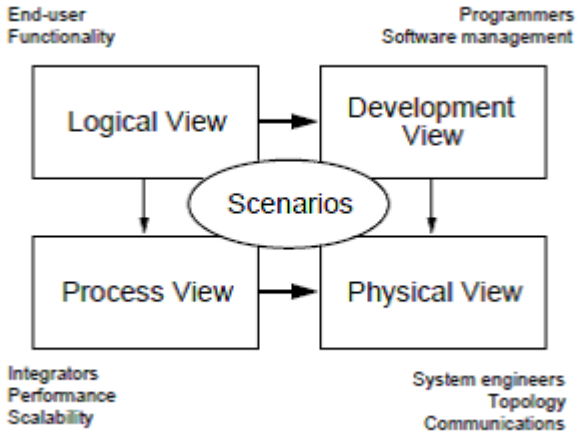

Figure 2.5: 4+1 view model [Kru95]

The four main views are:

- *The logical view* leads to requirements of the user's point of view. This includes the overall functionality of the system for the user.
- *The process view* addresses requirements that meet runtime issues, such as concurrency, performance and availability.
- *The development view* leads to requirements defined through modules and subsystems. It should represent the static architecture of the system.
- *The physical view* leads to distribution and hardware specific problems.

The sum of these views result in a fifth view, called *scenarios*. Those scenarios are nothing else than use cases of the whole system. They connect to every view, as every stakeholder has its own scenarios by using the system.

2.2.2 Volere Requirements Techniques

The *Volere Requirements Techniques* are originally developed by Suzanne and James Robertson. As it is written in [RR08], the idea was the development of a common language for all concerning stakeholders of a project. Those people can differ in education level, in profession, in age, etc. and therefore, their point of view can vary extremely.

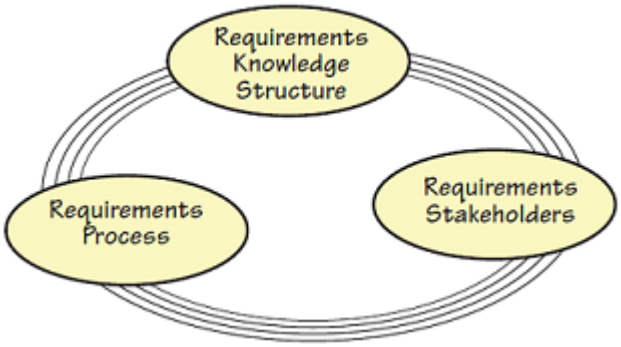

Figure 2.6: The basic idea behind the *Volere Requirements Specification* [RR08]

2.2.3 Multi attribute utility theory (MAUT)

The *multi attribute utility theory* (MAUT) is a method for impartially comparing different choices. It calculates an overall benefit of every alternative which can be compared. This subsection describes the procedure of calculating the best choice relating on document [vW86] and on document [Sch01].

To calculate the main utility value $v(x)$ of each alternative x, MAUT uses the simple idea of summing up weighted value dimensions. Therefor the overall value is defined by the *overall value function*

$$v(x) = \sum_{i=1}^{n} w_i * v_i(x), \tag{2.3}$$

where n is the number of value dimensions.

For every value dimension the evaluation value $v_i(x)$ is quite similar to be calculated. A list of relevant attributes A_i of every dimension d_i can be weighted by their *relative importances* $w_{a,i}$. Thus, for all $d_i(i = 1, ..., n)$ the evaluation value is calculated by the formula

$$v_i(x) = \sum_{a \in A_i} w_{a,i} * v_{a,i}(x), \tag{2.4}$$

where A_i is the list of attributes in the dimension d_i.

To get the evaluation values for the relevant attributes, the easiest way is setting up a scale e.g. from 0 to 10. An attribute, which does completely not fit, gets a zero and is therefore not considered in the formula, and an attribute, which does completely fit, gets the maximum value 10. During the creation of a scale there are two important rules that have to be considered:

- Start the scale at zero. If there is an alternative, which does completely not fit to a relevant attribute, a zero value will be ignored in the equations. Any other value would have an effect on the result, even though it is the smallest value that can be chosen.
- All dimensions must use the same scale. If e.g. one dimension uses scales from 0 to 10, and another uses 0 to 20, then a full fitting of a solution to an attribute in the second dimension would be double important as in the first dimension. Since the intention to declare the importances between different dimensions by extra weighting values, different scales are not allowed.

It is also important to keep in mind, that the resulting values $v_i(x)$ of every dimension will be compared among each other. After taking a deeper look at Equation 2.4, it becomes obvious, that dimensions with a higher number of attributes will always produce a higher value than dimensions with only a little number of relevant attributes. The consequence is a parasitic weighting between dimensions. To avoid this effect it is important to normalize the sum of all relative importances per dimension to one, which is shown in following Constraint 2.5

$$\forall i \in 1, ..., n : \sum_{a \in A_i} w_{a,i} = 1. \tag{2.5}$$

For the sake of completeness, it is also a good idea to normalize the weightings of the dimension values in Equation 2.3. However, as the resulting utility is not participating in a sum with comparable values, it is not necessary.

Chapter 3

Detailed description of the current system

3.1 WAMAS

One idea of the Salomon's software product line is the division of every final software for a customer in a common part, which is mostly constant and the base of every project, and a variable part, which must be adapted for every customer. The constant part is called WAMAS and has been developed as a logistic program that matches most of the usual requirements for a customer. Therefore it can also be used as a standalone software tool. It consists of Java 6 source code, UML 1.4 models which define some parts of WAMAS, and generators, which produce source code out of these models.

To satisfy the needs of a new customer, first of all a new project is started by copying the current release of the product WAMAS. Around this core system, customer depending code and models have to be developed and added to the project. In Figure 3.1, this

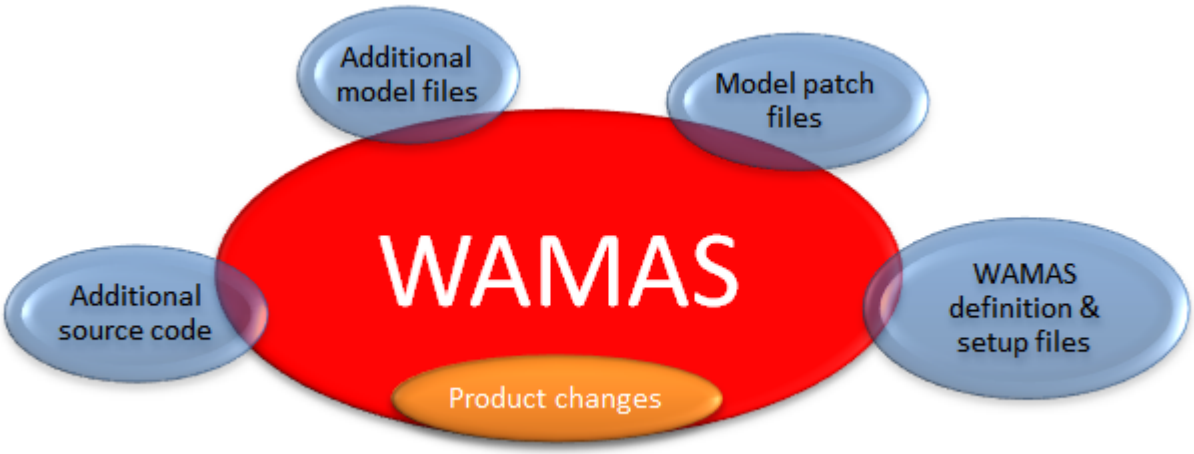

Figure 3.1: Schematic description of the modular architecture of a project

modular design should be indicated by the big red oval in the middle and the extensions around the product, pictured as the blue ovals. For the sake of completeness, an orange oval shows the part of changing the product source code. This is against the idea of the modular concept, but is not always avoidable. However, the intention is to reuse as much as possible of the original product code and models.

3.2 The current modeling, code generation and documentation tool chain

The software adaptions for the projects are not intended to be developed directly as source code. They furthermore will be carried out by additional model files and the usage of a tool chain, which generates definition files and source code. As a description language for those models, diagrams of the UML 1.4 specification are currently used.

In addition to executable code and definition files, also reports are generated from the models. These reports are usable as documentation. In case of Salomon, the report files are linked *Hypertext Markup Language*(*HTML*) files reachable through the intranet, called "Public WIKI", and represent a help and documentation service.

Salomon actually uses a lot of different types of generators e.g. Hibernate definitions, state machines for mobile devices and software modules for communication. Some of those outputs are complete software modules, like the database, others only consist of a framework, which has to be filled up with manually written code. The created state machines for the mobile devices would mark an example for this.

The next subchapters describe the currently used tool chain in a more specific and detailed way. Actually, there are three, more or less different use cases of the tool chain:

- Generation of a database system for the real worlds customer warehouse, called *Persistency Model use case*
- Generation of definitions for the communication between the mobile clients and the server, called *Mobile Client-Server Communication use case*
- Generation of application and data flow definitions for the mobile client software, called *Mobile Task Concept use case*

3.2.1 Persistency Model use case

The first use case of the code generation tool chain is the Persistency Model. It should represent the complexity of the real world's customer warehouse. As it can be imagined, a customer's warehouse consists of a huge number of different structures inside. Therefore logistics software also consists of a high complexity to aim a most common usage. However, to describe this Persistency Model only class diagrams are necessary. Elder projects

showed, that a model of a real world warehouse, containing its whole complexity, usually results in around 600 classes. This amount of objects in one diagram is impossible to handle; therefore, a kind of modularity had to be brought in. Designing smaller parts of the model in several files lead to a much higher lucidity. Thus, splitting the big model into several sub models and sub model files, makes designing a model with this size much easier. For an easier implementation of a model driven development, it is advantageous to start with only one file containing the whole information. Hence, for this use case the tool chain starts with a linker, which combines all files of the mentioned model parts to one big model file. This linker only works with input model definitions in the file format of XMI 1.0. The result of the linker is further modified by a model patcher (refer to Chapter 3.2.1).

The output file of the patcher is still in the file format of XMI 1.0, but the generation layer expects a file with an XMI definition version 1.1; hence a converter is required to translate the XMI version after execution of the linker and the patcher. The whole modeling phase of this use case is depicted in Figure 3.2.

The second step in this model driven development begins with a model tester, which checks all relationships, the uniqueness of class names and the correctness of the entire model. In case of an error, the tester is able to cancel the execution of the tool chain and reports all errors found. If no errors occur, the main intention of this layer starts and the generators create the files for the database system and the documentation, as shown in Figure 3.3.

The output of the generators, combined inside the light blue rectangle in the beneath area of Figure 3.3, contains different types of files. The database access layer consists of XML files for mapping the Hibernate layer and Java 6 code for the Meta model and type-safe entity container layer. Also *Structured Query Language Data Definition Language* (*SQL-DDL*) scripts are generated to setup the database.

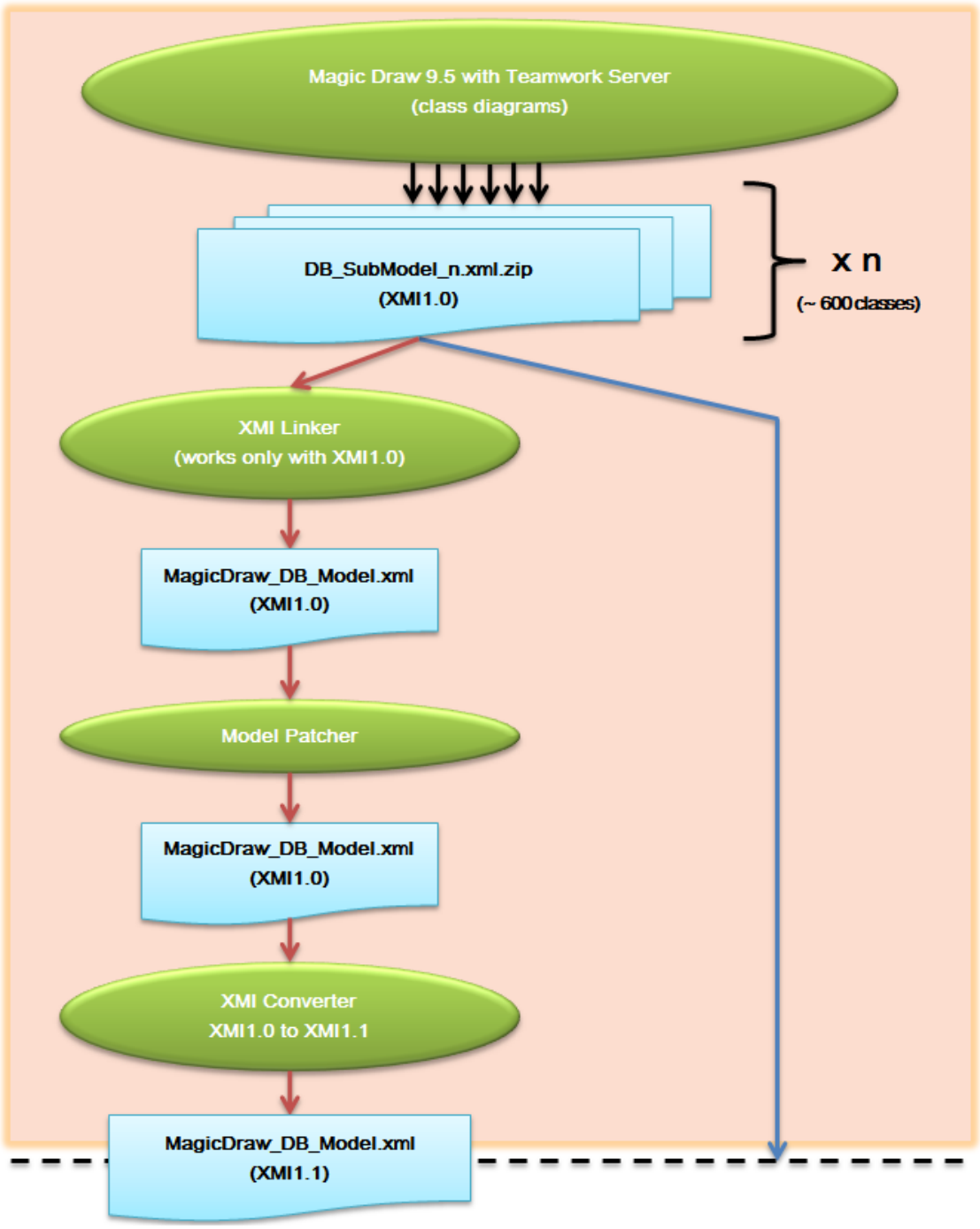

Figure 3.2: Modeling layer of the *Persistency Model use case*

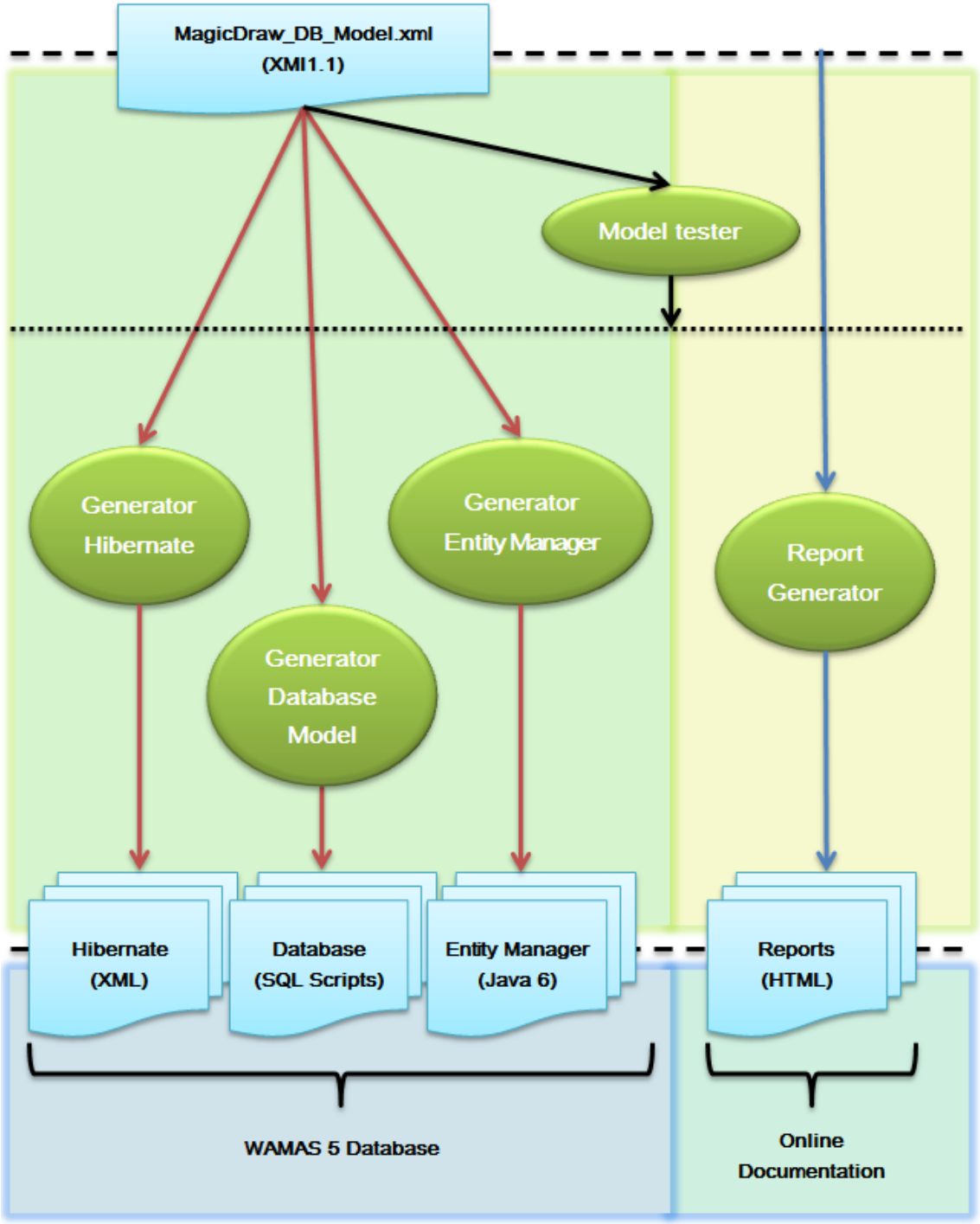

Figure 3.3: Generation and output layer of the *Persistency Model use case*

Model patcher

The modeling layer of the persistency use case actually offers an additional functionality. It is possible to modify the product model files exclusively by adding other model files. These other files are the project model files. The idea behind this patching system is the reusability of the product models, which should be the same in every project. The project models expand and change the product models by containing instructions and information for the model patch tool. The output of the tool is a new model file consisting of the original model with all applied changes. This file is only temporarily available as an input for the following generators.

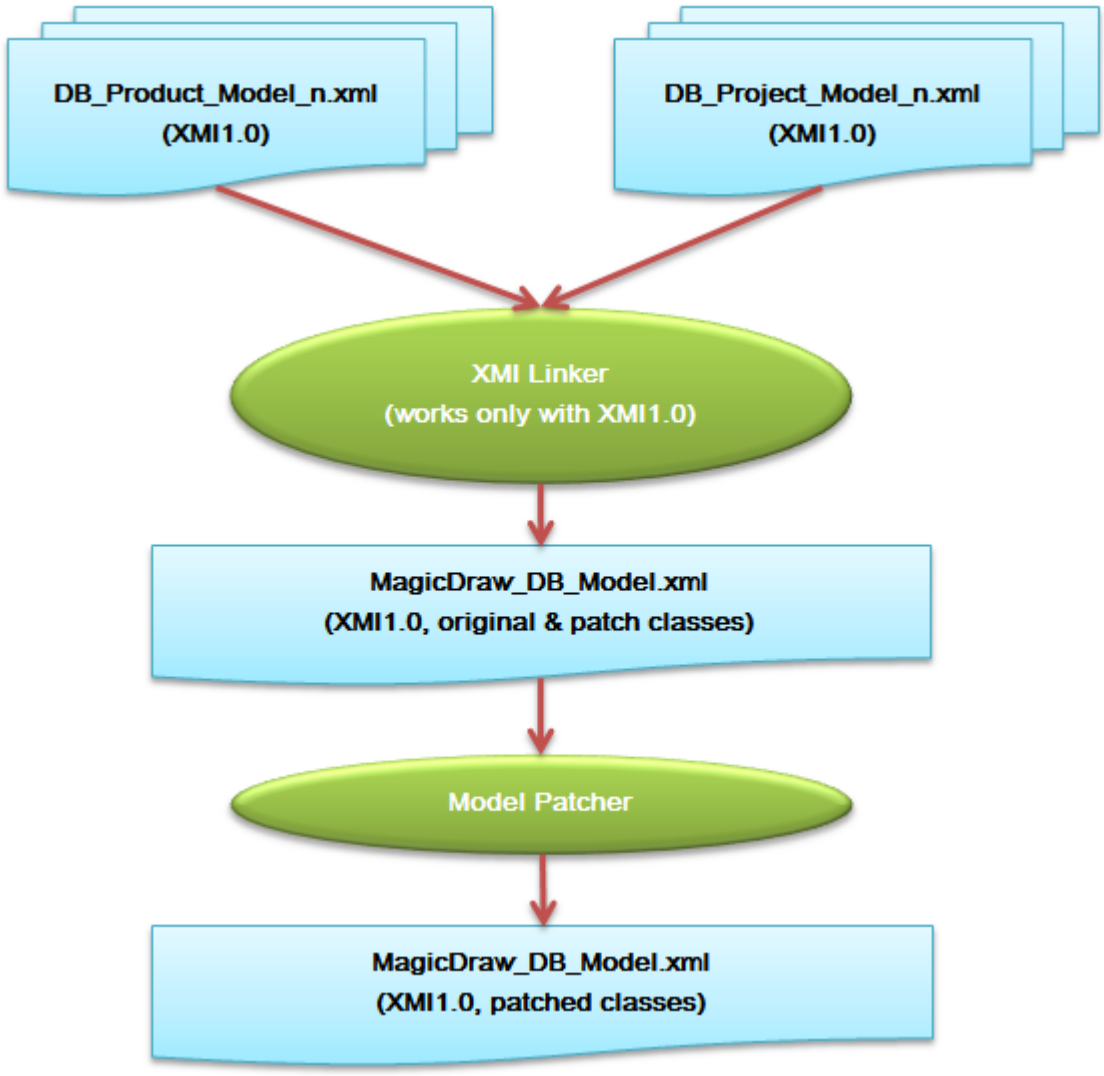

Figure 3.4: Description of the patching functionality

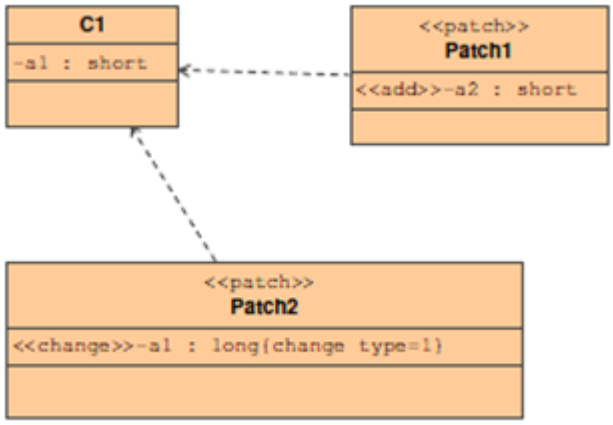

Figure 3.5: Example class with patch classes

Example: A product model contains a class called "C1". This class consists of a property "a1" of the type "short". The project model contains two other classes called "Patch1" and "Patch2", both are of the stereotype "<<patch>>". "Patch1" consists of a property "a2" of type "short", with the stereotype "<<add>>", "Patch2" consists of a property "a1" of type "long", with the stereotype "<<change>>". The patching classes are connected to the original class by dependencies. This example is shown in Figure 3.5. Running the patcher creates a new class based on the original one. It takes the information of the two patching classes and applies it to the new class. The result is a class called "C1" (the name has not been processed by the patcher) with the property "a1" of the new type "long" and the new property "a2" of type "short" (Figure 3.6).

The model patcher is exclusively used in the *Persistency Model use case*.

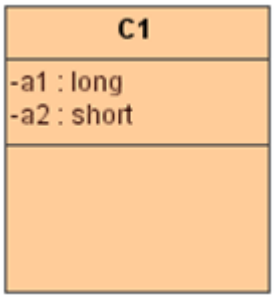

Figure 3.6: Resulting class after execution of the model patcher

3.2.2 Mobile Client-Server Communication use case

Another use case of the code generation tool chain is the definition of the communication between the mobile devices, called clients, and the server. Compared to the mentioned use case before, the modeling of the communication use case is much easier. Because of the less complexity, the model can be represented in a few class diagrams and stored in only one file. Therefore no linker or model patcher is necessary. Without a linker in the tool chain, there is also no need for a converter of the file format XMI 1.0 to XMI 1.1. The input file for the following generator can be saved directly in the right format from the used UML editor.

The generation layer again starts with one model file in the format XMI 1.1 and checks first, using a model tester, the correctness of the input. If a negative result is produced, the model tester is able to stop the execution of the tool chain and reports the errors found. In case of no error, the code generators create the program code files. The report generator creates the documentation files as described in Chapter 3.2.4.

The output of the source code generator consists of two parts. The first part is the definition of the communication module on the client side, which uses the technology of *Connected Device Configuration* (*CDC*) 1.0. The second part is the definition on the server side, again in Java 6.

A graphical description of this use case is pictured in Figure 3.7.

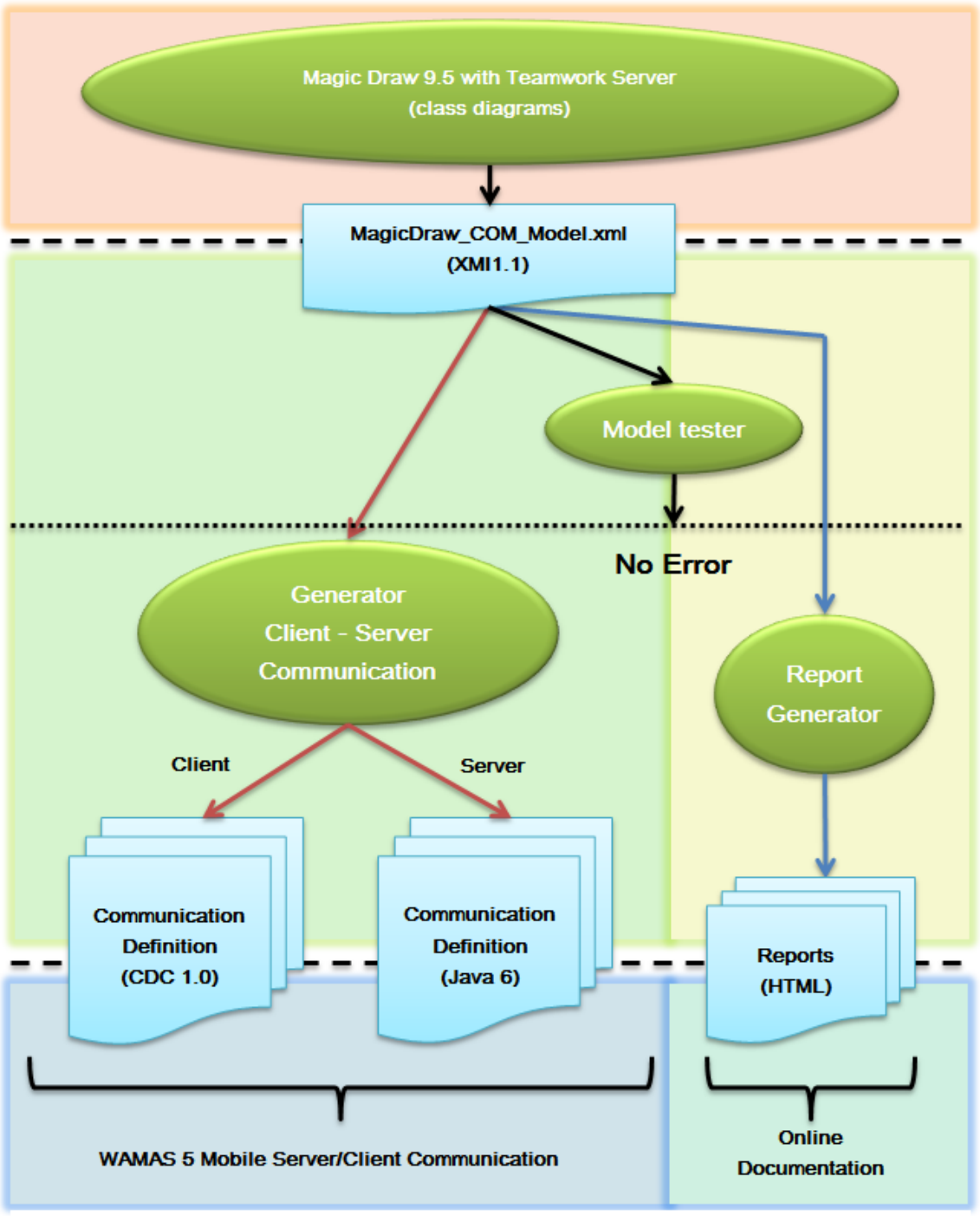

Figure 3.7: Layers of the *Mobile Client-Server Communication use case*

3.2.3 Mobile Task Concept use case

The model of the Mobile Task Concept use case has a big difference to the use cases described before. It not only consists of a structure diagram, like the class diagrams, it also describes behavior of a program flow using state machine diagrams.

Similar to the Mobile Client-Server Communication the whole tool chain starts with only one file containing all diagrams. So there is again no need for a linker or a converter. Also no model patcher is used. Also similar to the last use case is the usage of a model tester, which first checks the correctness of the model inside the file and gives its permission for a further processing or not.

The output of the source code generator consists of state machines, coded in Java 3, which are targeted to run on the mobile devices. The behavior of the single states cannot be modeled. Therefore some additional code has to be written manually and added to the generated state machine framework.

The report generator creates the online documentation, as it is described in Chapter 3.2.4.

An illustration of this use case is pictured in Figure 3.8.

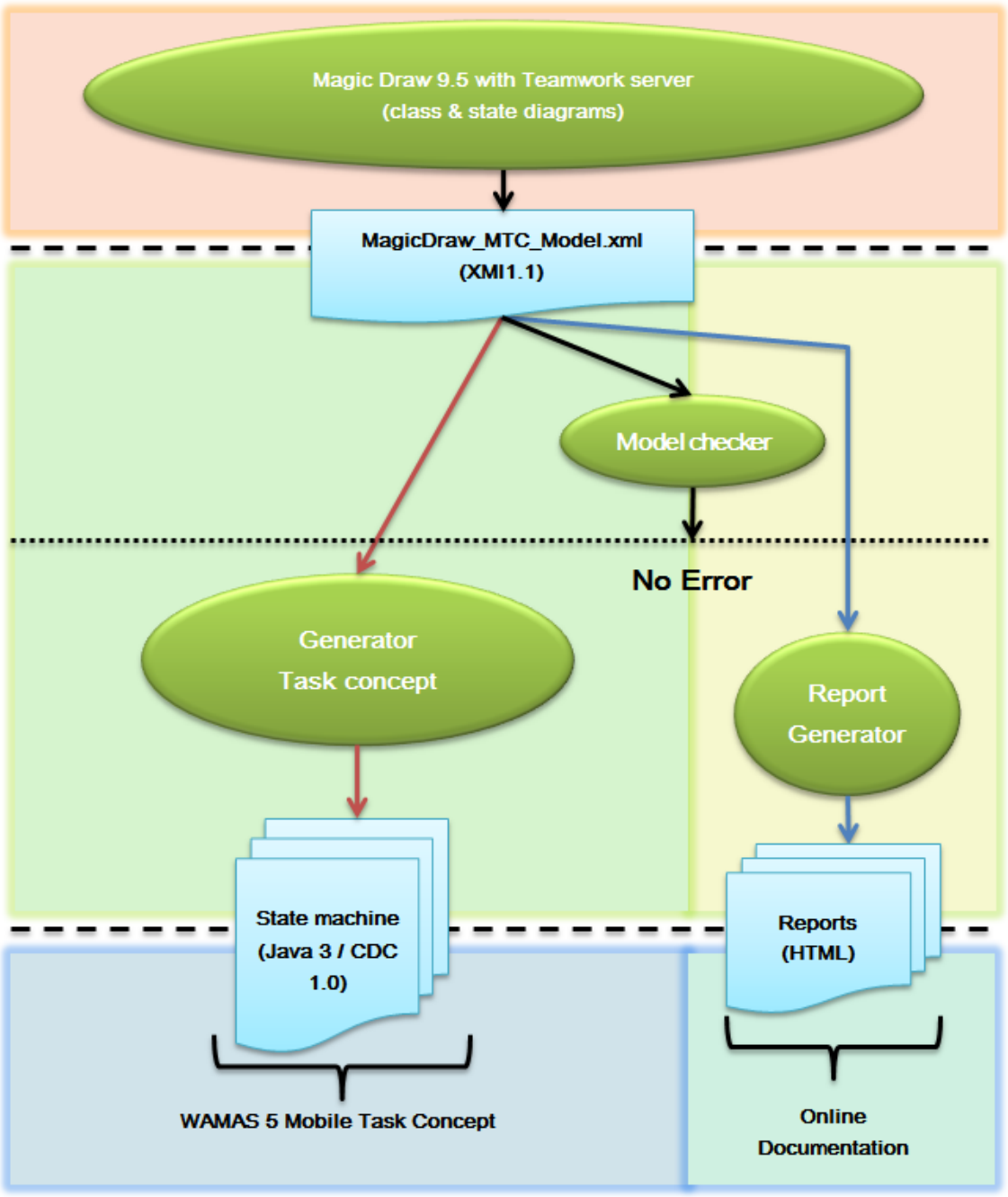

Figure 3.8: Layers of the *Mobile Task Concept use case*

3.2.4 Report generation

The generation of the reports is slightly more complex, than mentioned before. First of all, it has to be explained, that the main part of the process currently is made by the UML editor used in the tool chain (refer to subchapter 3.3.2). Actually, this is an advantage, as no compiler from the MagicDraw XMI (1.0 and 1.1) file format to HTML had to be developed.

Unfortunately the resulting report files from the UML editor are leaking of some overview functionality. This is not a huge problem, because HTML can easily be read and manipulated. A following self-developed report post-processor eliminates the missing functionality and adds index files to create an overview over all generated documentation.

The Persistency Model use case requires another additional functionality. In contrast to the other uses cases, which both turn over only one file to the report generator as an input, the Persistency Model use case has at the beginning a lot of files, each containing a part of the whole model. In addition to index files for the different diagrams, a linking process is executed to connect the produced reports of these several files.

In the end, the resulting documentation is available via web browser in the company's documentation intranet, called "Public Wiki". As a matter of fact, this is seen as a big support for the developers. They even prefer using the online documentation rather than looking up the models by using the graphical editor and the model files.

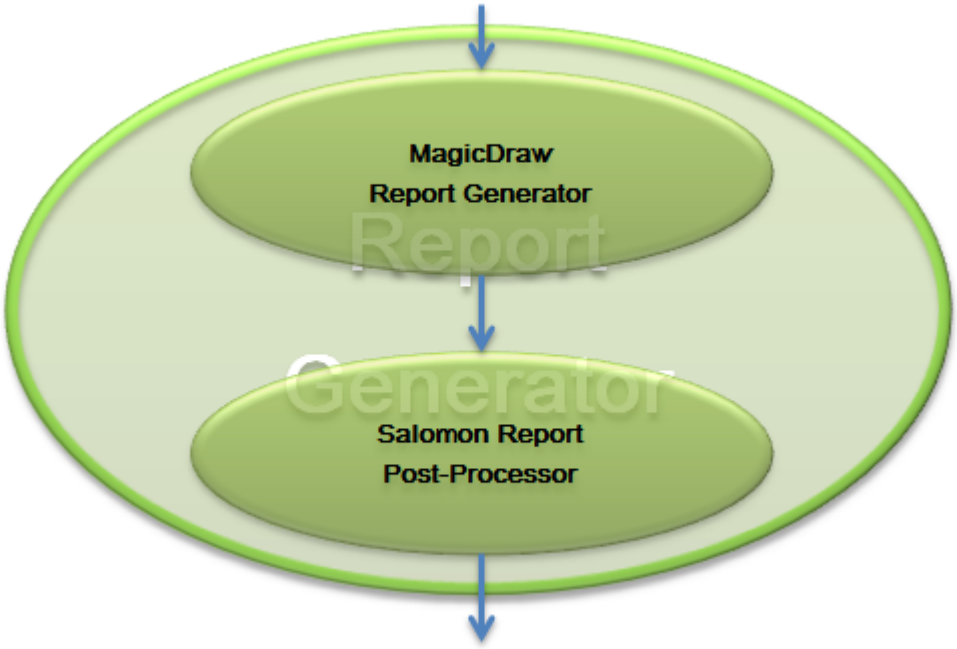

Figure 3.9: Real implementation of the report generation

3.3 Aspects of currently used software, file formats and tools

The subsequent chapters describe properties of software, file formats and tools used in the current version of the tool chain. The descriptions are focused on issues, which are important to understand the reasons of usage. They also point out disadvantages and workarounds, which have a high potential of improvement.

3.3.1 Operating systems

Salomon uses Microsoft Windows and Linux as operating systems. During creation of this document there is an upgrade taking place from the Windows version of XP to the, at this time, newest version of Windows called Windows 7. The used Linux distributions cannot be specified, as every developer is allowed to choose his favorite distribution.

As a consequence of the different operating systems, the used programs must be platform independent concerning distributions of Microsoft Windows (XP and 7) and Linux.

3.3.2 MagicDraw 9.5 with Teamwork Server 9.5

MagicDraw is a visual modeling tool for i.a. UML. It provides transformation of UML models to specific XML Schema and DB models, as well as any to any transformation. It consists of many converters with different specifications, which provides a big range of possibilities to import, export and create of files in different formats.

The MagicDraw version currently in use is 9.5. It works with UML 1.4 and is therefore not up to date. As a file format it uses XMI and supports the versions 1.0, 1.1 and 1.2. As mentioned before, the versions of XMI 1.0 and XMI 1.1 are currently in use.

The first reason for the usage of MagicDraw as the model designing tool is the simple handling of the designing *Graphical User Interface* (*GUI*). Also the platform independency is an important reason, because the company works with different operating systems (refer to Chapter 3.3.1). But the main reason for using MagicDraw is the potential of automatic processing of the model file format. XMI is a specified format and therefore, other tools can read the output files and use it as input files.

An additional advantage is the built-in HTML report generator. It solves the problem of developing an own generator, which reads model files and produces graphics and text in HTML files. The output is not quite perfect, but changing of or adding functionality to website files is much easier, than developing a MagicDraw model reader and converter. More information about this topic can be found under Chapter 3.2.4.

NoMagic, the producer of MagicDraw, adds also a team supporting tool for MagicDraw called Teamwork Server. It provides a version control system, which is reachable

over network by more than one instance of MagicDraw at the same time. This offers simultaneous access to the files managed by the Teamwork Server.

Furthermore, it allows an employee to lock specific parts, he currently works on. After locking, other instances do not have any possibilities to edit them. This avoids concurrent changes in the same part of a diagram.

Figure 3.10: Locking a class with all its relationships in MagicDraw 9.5

Unfortunately, the Teamwork Server is not able to make branches and merge them back later, which is a big disadvantage in a workflow containing team driven development. In contrast to the rest of the code of WAMAS and all its projects, the models are only available in one version (and its history of course) for all development teams. This creates overhead work, as usually many different branches of the non-model files exist. Currently all branches are sharing the same model files.

In this version of the NoMagic Teamwork Server it is not possible to integrate another version control system, like SVN. Therefore, the only handy system for managing all types of files, including the usage of its editors and avoiding overhead work for solving concurrent changes in same files, is driving a two-tier file repository system. In case of Salomon, on the one hand *Subversion* (*SVN*) is used for managing all files of projects (and the product), and on the other hand, the NoMagic Teamwork Server 9.5 is used for synchronized editing of the model files. A more detailed description of the current handling of the model files is available in Chapter 3.4.2.

3.4 Workflow of the current tool chain

After the description of the use cases and several used tools, file formats and more, now the reader of this document should be able to understand the following description of the workflow and the problems caused by some properties.

3.4.1 Managing WAMAS and customer projects

The basic idea behind is the splitting of the whole system in a constant part, called WAMAS, and project specific parts, which use WAMAS as a fundament. Modifications, depending on every customer, are made in the repositories of every project. Because of the modular system mentioned in the chapters before, the WAMAS part has not to be necessarily changed, if modifications for the projects are made.

Additionally, the used software repository system remembers the connection between the checked out product and the current repository, hence it is possible to merge files between them. This fact makes it very convenient to develop the WAMAS product parallel and independent to the projects. Any code changes can be overtaken to the projects by updating the code. Also the other direction might be thought of, as a developer can find a bug, which only occurs because of a specific use case in a customer project, and merge it back.

In Figure 3.11 a schematic description shows an example of this workflow. Every project (A, B, C and D) first starts as a so called "SVN Copy" of the WAMAS repository, represented by the red arrows starting at the red product rectangle. For example, after creating "Project A" some modifications have been made, so the version incremented. Also the product itself has been updated; therefore it is available in version 1.1. The definition of "Project D" starts after this WAMAS update and begins already with this newer version.

An interesting fact now is the ability to perform an update of the projects "A" and "C", which only consists of the so called "SVN update" of the WAMAS code. This is a merge process, which usually is not free of conflicts. Although it needs some integration time to solve the conflicts, it is a faster and handier way to update changes in the projects, as it would be by integrating the changes manually.

3.4.2 Managing model files

The chapter before should explain the handling of projects, or to say it in a more precise way, of its containing source code files. It has to be mentioned, that the WAMAS software system not only consists of textual source code. Some parts are generated by model files in the notation of MagicDraw XMI 1.0 or 1.1. These files are not able to be merged by SVN.

Explained in Chapter 3.3.2, the current system uses a central point of storage, where a repository of the model files resides. The question may be raised, why it is necessary to have the model files in the Eclipse workspace or rather in the SVN repository, but the answer is easy: the generator needs the files as inputs. It has no access to the NoMagic Teamwork Server and needs therefor a local copy of the file. Also, it is a good idea to have the model files together with the produced code files.

As a consequence of this dual system, the model files in the SVN project and product repositories have to be synchronized manually.

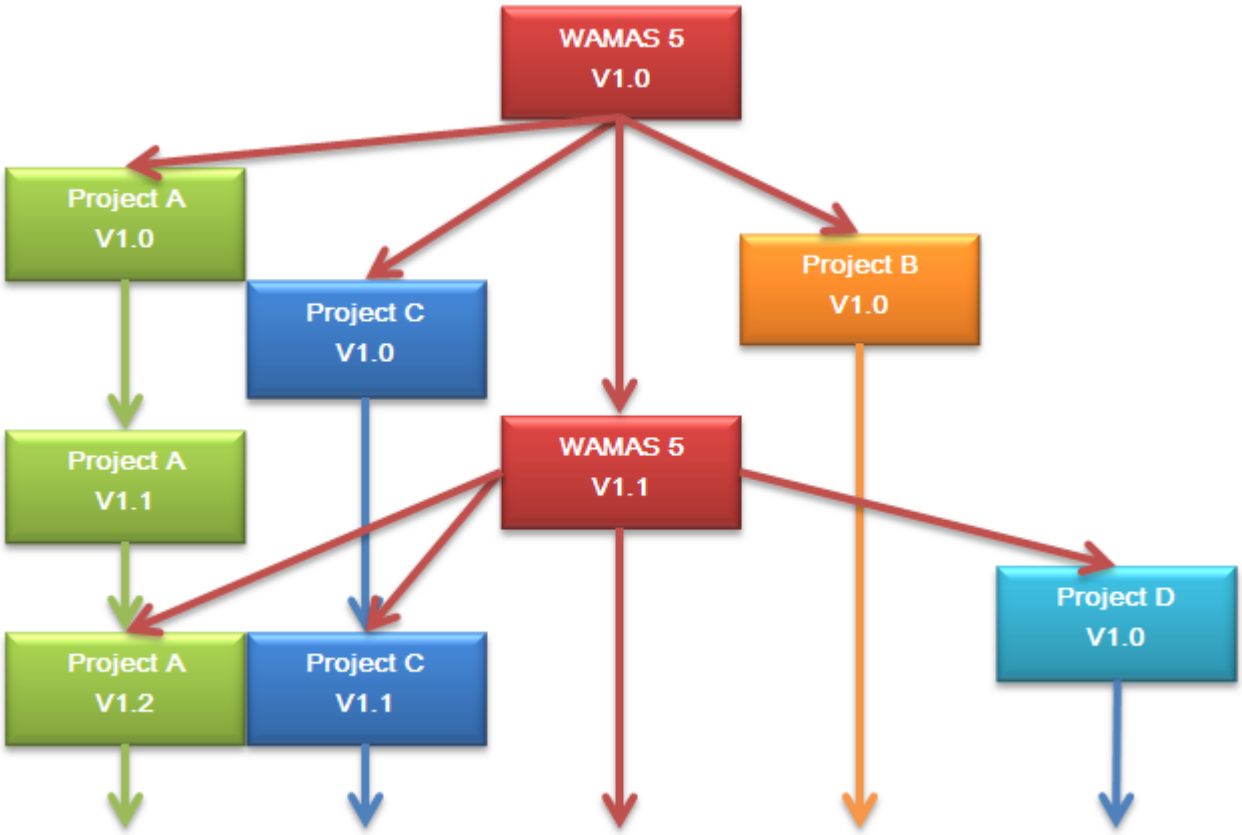

Figure 3.11: Schematic description of the connections between project and the product WAMAS

Figure 3.12 pictures an example of the file synchronization workflow. Each of the four colored bars represents a single program. The green bar is the repository server, where the whole WAMAS code is checked in. The orange bar stands for the instance of *Eclipse*, rather its workspace. The two bars on the right side represent the *NoMagic* system, consisting of *MagicDraw 9.5* (purple) and the *Teamwork Server 9.5* (blue).

First, it is assumed, that a model file has to be modified. It is currently available in the version 1.3. To start with the modification, the file has to be checked out from the repository into the workspace of a local instance of *Eclipse*. Next, we can start the graphical editor and open the specific file. However, the order of these two steps can also be switched, because *MagicDraw* does not load the file from the *Eclipse* workspace. The program loads it, as already mentioned, from a copy of the file on the *Teamwork Server*. Thus they are pictured parallel in Figure 3.12 to indicate their independence. There is also no connection between the *Eclipse* workspace bar and the *MagicDraw* bar.

After *MagicDraw* is opened and has loaded the model file, the next step is locking the parts of the diagram, which will be modified. This avoids a concurrent modification by

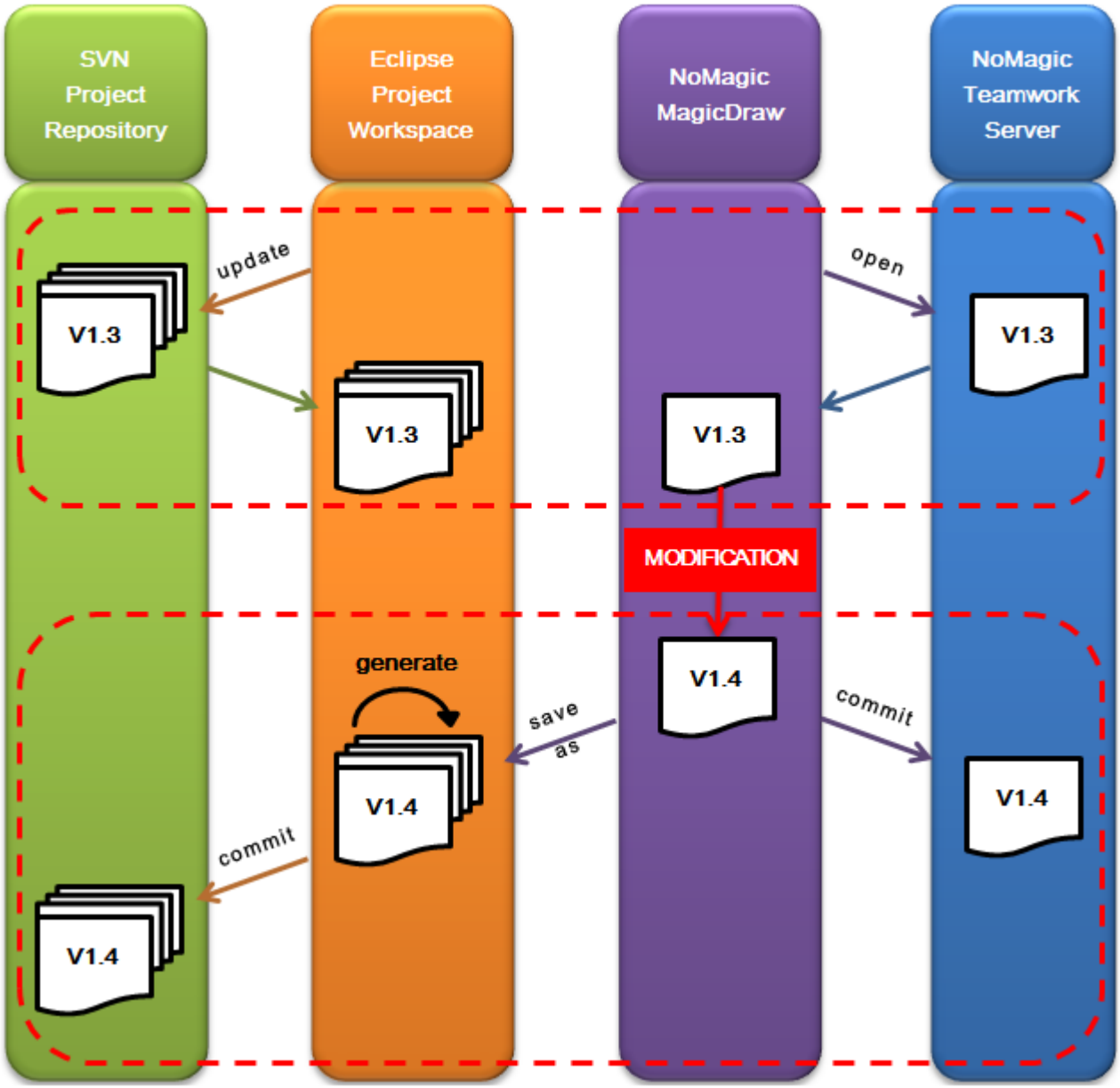

Figure 3.12: Workflow of keeping the two repositories synchronized after modifying a model file

another user. Once the classes in the diagram are locked, the user/developer is allowed to access, modify and change them. In this example, the new submodel owns the incremented version number 1.4 after the modification.

If the adaptions are finished, the user/developer wants to save their changes concentrate themselves to something else. Unfortunately, pressing a save-button is not enough. On the one hand, the *Teamwork Server* has to be informed to adopt the new project. This is made by explicitly committing the project or just unlocking the locked parts. On the

other hand, a copy of the model file has to be saved into the workspace of *Eclipse* to make it reachable for the software generators. This is achieved with "File" → "Save Project As". A dialog window pops up, where the name and the location can be chosen. The name must not be changed, as the existing file in the old version should be overwritten with the new one in the *Eclipse* workspace.

There are also some radio buttons on the dialog to select the file format, refer to Figure 17. As described in Chapter 3.13, it is important to manually choose the right version of XMI, depending on the use case. At this point it has to be mentioned, that the check box with the text "Rich XMI" below the radio buttons must be enabled. This causes the file writing algorithm to use the whole specification of XMI. If it remains unchecked, it would only save information, which is import for MagicDraw to read the file. Other tools, like the following tool chain, would not be able to reconstruct the models in the files because of missing information, i.e. default values.

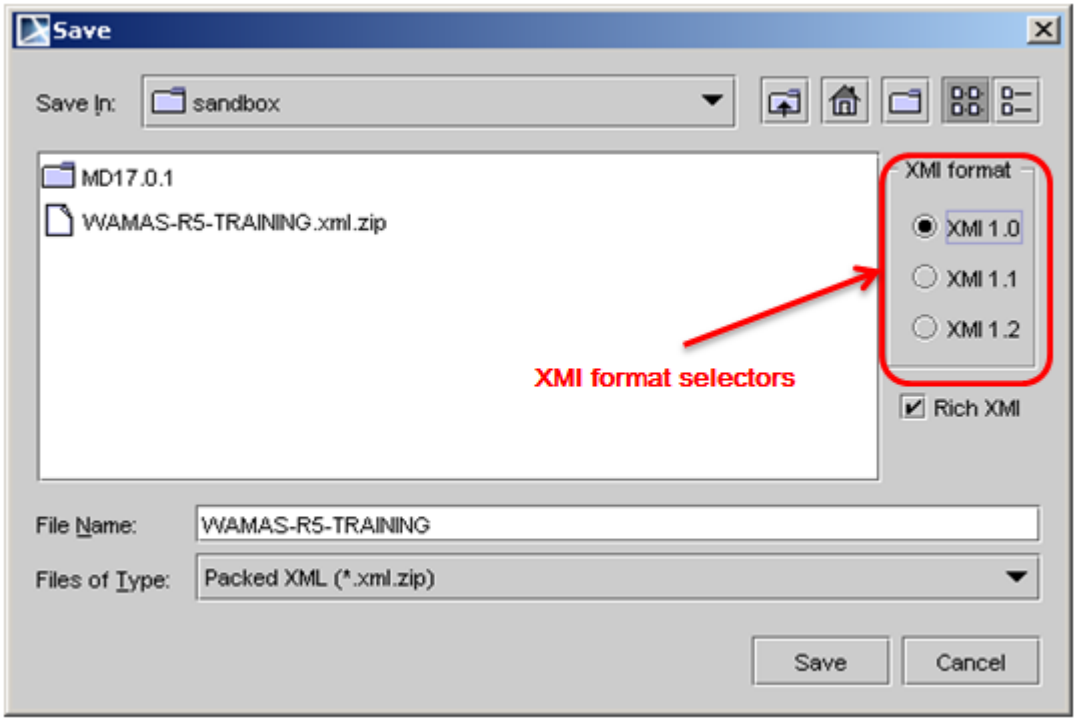

Figure 3.13: The "Save As..."-dialog

When the file is saved into the Eclipse workspace, the generators can be started. They produce the new source files and of course, the reports. These new files have automatically overwritten the old ones. Now the new system can be saved back into the repository system by committing all files.

Chapter 4

Requirement finding and evaluation methods applied

4.1 Description of the finding and evaluation methods

4.1.1 Applied method for finding requirements

The method for finding requirements leans on a mix of the 4+1 view model, described in Section 2.2.1, and the Volere Requirements Specification Template, described in Section 2.2.2. The idea behind this method is the observation of the desired system by different point of views and the definition of the stakeholders, having this point of views.

The duty of every stakeholder then is to write down attributes of the system, he desires or he thinks, that are important for the system. A very crucial point is, that he is only allowed to list attributes with regard to his role. For example, a user of a system must not list the price of a program, if there is another stakeholder, who sees the system of a financial point of view. Therefore, the participating people should first of all ask themselves, what their role is and what they are really interested in. This game is one of few in the world, in which selfishness and narrow-mindedness is not only allowed, but it is welcomed.

After clearing out the points, they are divided into two categories. The first category contains the necessary requirements. They are called *knock out criteria* (KOC) and represent properties, which are crucial for the whole system. If a solution candidate lacks of one of those points, it can be ruled out immediately. Therefore it is not necessary to divide the KOCs into different categories, they can be put together into one group. Evaluating alternatives by those criteria can be seen as a first filtering level.

The second group contains the rest of the mentioned points. These attributes of the system can be seen as *nice-to-have*-features. Possible candidates, which survived the first filtering by the KOCs, can be evaluated and compared by this points. For the further evaluation process, it is necessary to build groups, called views. Every view consists of a list of attributes (called *view lists* (VL)) and a corresponding list of their importances in the view, both defined by the stakeholder of the view (see Figure 4.1).

4.1.2 Applied method for evaluating possible solutions

To evaluate our candidates the MAUT (explained in Section 2.2.3) is used in a slightly different notation. First, different relevant attributes are declared as the non necessary requirements of the views. Therefore these views represent the dimensions. As a second, the lists of weightings are written as row vectors. For the next equations n is defined as the number of views, i as the index of a view and m_i as the number of relevant attributes inside a view i. Further, let the scalar variable v_{view_i} be the evaluation value of one dimension, the row vector $\vec{w}_{attributes,view_i} = [w_{1,i}, ..., w_{m_i,i}]$ be the list of weightings for the relevant attributes and the row vector $\vec{v}_{attributes,view_i}(x) = [v_{1,i}(x), ..., v_{m_i,i}(x)]$ be the list of the evaluation values for the relevant attributes of a product x.

Keeping in mind, that the upper limits of the scale inside a view should sum up to one, the defined row vector $\vec{w}_{attributes,view_i}$ has to be normalized to a row vector $\vec{w}_{attributes,view_i,norm}$. This can be achieved by using a sum row vector $\vec{1}_i$, which has the same length m_i as the weighting vector and consists only of ones.

$$\vec{w}_{attributes,view_i,norm} = \frac{1}{(\vec{1}_i * (\vec{w}_{attributes,view_i})^T)} * \vec{w}_{attributes,view_i} \tag{4.1}$$

Then the following Equation 4.2

$$v_{view_i}(x) = \vec{w}_{attributes,view_i,norm} * \vec{v}_{attributes,view_i}(x)^T \tag{4.2}$$

represents the same mathematical result as shown in Formula 2.4.

The same mathematical notation can be used for the overall value function. In Equation 4.3 the scalar value $v(x)$ is the utility value of a product x, the row vector

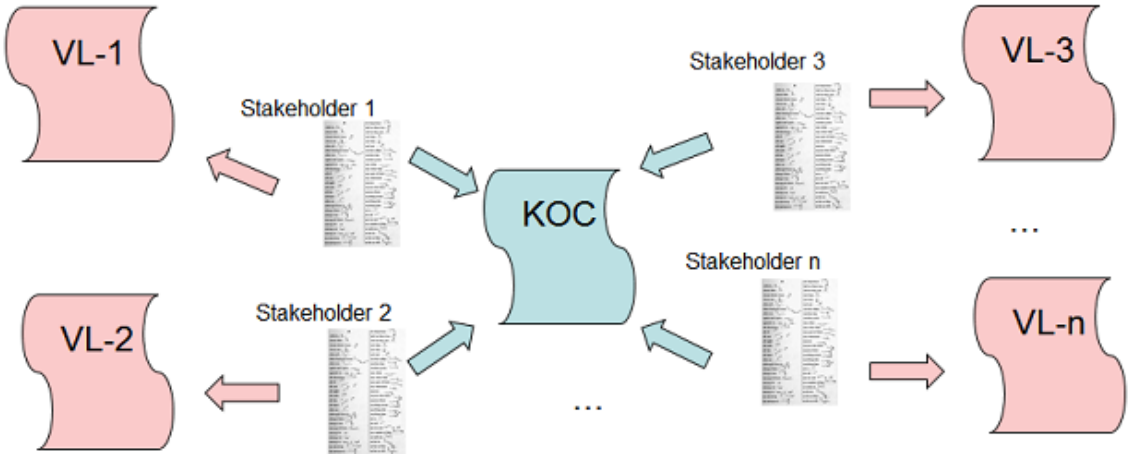

Figure 4.1: Schematic description of the n+1 lists by n stakeholders

$\vec{w}_{views} = [w_1, ..., w_n]$ represents the weightings of the different views and the row vector $\vec{v}_{views}(x) = [v_{view_1}, ..., v_{view_n}]$ contains the view values (dimension values) calculated before.

$$v(x) = \vec{w}_{views} * \vec{v}_{views}(x)^T \tag{4.3}$$

Therefore to evaluate the utility value a number of vectors is needed. First, there are vectors independent to the alternative x. These vectors are:

- The weighting vector of the views $\vec{w}_{views}$. As it is independent of the alternative x, it is only one vector with the size of n.
- The weighting vectors for the attributes of every view $\vec{w}_{attributes,view_i}$. These are about n vectors with a different number of elements.

For an impartial result, it is recommended to define these vectors before the definition of the evaluation vectors. Actually it can be done at the time of finishing the requirement finding process.

As a second, for every possible candidate x, one evaluation vector per the view is needed:

- The evaluation vectors for the attributes of every view $\vec{v}_{attributes,view_i}(x)$. These are about n vectors with different numbers of elements.

Consequences of this method

Like many other methods, also this one has benefits and disadvantages.

Benefits:

- A great benefit of this method is the detachment of the evaluation of the requirement fittings and the evaluation of the importances of a requirement. Since the weightings are equal for every solution, they can be contracted even before any possible solution is found. This matter of fact reduces prejudice a lot.
- Also splitting the whole system into different views is a big advantage. Usually a bigger system is owned by many stakeholders. A discussion between those stakeholders has a high potential to lead to disputations, caused by the different views the different stakeholders have. By letting every "expert" evaluating his own domain, this situation can be avoided. Also everyone has a feeling of participating in the result, because his evaluation is a crucial input for the following decision making process.
- The ability of implementing the equations to an automatic evaluation system (*Microsoft Excel*, *Matlab*, *MathCad*, etc.) allows different variants of evaluations. Several versions leading to the same decision can make a possible solution obvious without arguing on specific attributes' weightings or evaluation values.
- The data of the evaluation is also a documentation for non included persons.

Disadvantages:

- If the whole process of finding the decision will not be kept transparently after getting the evaluations and weightings, the stakeholders may feel overruled. This can lead to a refusal by a coworker the next time a decision has to be made.
- The utility value depends strongly on the weightings of the different views. Therefore finding proper values for this vector should be made by a person which has expertise in every different view.

4.2 Definition of stakeholders, knock out criteria and view lists

4.2.1 Stakeholders

The first thing to do is to define the stakeholders. The idea is to give everyone an own view, who is related to the resulting system. Concerning this, four different stakeholders emerge:

1. **Model developer**: The first one is obviously the user of the system. This role is covered by the model developer or project owners, which are using the model creating and modifying functions, and also the file distribution and managing parts of the system.
2. **Product developer**: Taking a closer look on the role of the user shows up a group of people, using the resulting tool chain: the product developer is responsible for the core product and also uses some functionality of the system. He is also assigned to connect the product with the output of the generators, therefore he creates the tool chain for a proper use. This means he is responsible for creating the generators and writing tools (i.e. converters) for connecting the single tool chain links.
3. **System administrator**: Another stakeholder can be found by looking onto the role of the developer: the system administrator. He is responsible for keeping the system at work and deals with the server and client programs.
4. **Finance manager**: As in every other system in the world one essential view is the finances. Not only acquisition costs are an important point, also maintenance costs and costs for training the users on the new system have to be considered.

After finding the stakeholders and accordingly the views, the persons have been asked for creating a list containing all important points of the system from their point of view. The following Tables 4.1, 4.2, 4.3 and 4.4 show the results of the opinion surveys.

Model developer (MD)	
Keyword	**Description**
suitable editor	A graphical or text editor to develop UML2 class and state machine diagrams OR equivalent types of diagrams (graphical editors are preferred)
repository system	Full revision control system support for model files, which includes the ability of file storage, revision control, branching and merging, concurrent modification control and accessibility over network (including the Internet)
report generation	Graphical report generation to HTML with external accessibility (API or command line executable)
stability	Ability to handle large models (600 classes), inadequate waiting times or regular software crashes are not acceptable
migration	Ability of migrating the existing models (importing, converting etc.)
loading time	Typical waiting times when starting program, loading files, connecting to server etc., the faster the better

Table 4.1: Requirements list of the model developer's view

Product developer (PD)	
Keyword	**Description**
API for file access	Accessibility of the model files by the programs of the tool chain (API for controlling the repository system)
suitable file format	Machine readability of the model files and their language for further processing

Table 4.2: Requirements list of the product developer's view

System administrator (SA)	
Keyword	**Description**
server: Linux	Any server program must support Linux
clients: Win XP, 7 and Linux	Any client programs must support Windows XP, Windows 7 and Linux
user management	If there is a user management system, it would be a feature to connect it to the company's existing system.

Table 4.3: Requirements list of the system administrator's view

Finance manager (FM)	
Keyword	**Description**
product price	The products price should be affordable including costs for maintenance and software updates, and also applies: the cheaper, the better
training costs	A new system may lead to necessary training for the model developers, which leads to additional costs
development costs	Adoptions of the old system or even a new development leads to costs

Table 4.4: Requirements list of the finance manager's view

4.2.2 Knock out criteria list

The first step after getting a response is to talk to every stakeholder for his opinion, which of the points of his list are knock out criteria. In this domain, knock out criteria should be really seen as points, which are able to rule out a possible solution, if even only one of them is not fulfilled.

The following Table 4.5 shows the resulting knock out criteria list. This list has assigned a number to every point, which will be used further on in this document.

KOC List		
#	**Keyword**	**brought in by**
KOC1	suitable editor	MD
KOC2	repository system	MD
KOC3	API for file access	PD
KOC4	suitable file format	PD
KOC5	report generation	MD
KOC6	stability	MD
KOC7	migration	MD
KOC8	server: Linux	SA
KOC9	clients: Win XP, 7 and Linux	SA
KOC10	product price	FM

Table 4.5: Knock out criteria list

4.2.3 View lists

The next step is to eliminate the found KOC's in the original lists, given by the stakeholders. At this point, a special situation can occur: even if a point of the list is used for the KOC list, it can also be used for further evaluation of a solution. On a general view, this can happen, if a property of a system must have a kind of minimum level to be a solution

candidate. But after this level has been reached, it still can be a factor for comparison. An example for this situation is here the suitable editor (KOC1). The new system must have an editor, which can be used to edit all important properties of the models. But even if this requirement is fulfilled, there is a wide range of more or less handy editors, in the shown scenario, a graphical editor for UML will be preferred against a text editor for UML.

After reviewing every point and ruling out the KOC points, which can not be used for comparison, the remaining points on every list can be seen as the criteria for a new system's rating. These lists are now called view lists and the stakeholders now have to weigh the remaining points on a scale.

Note: For this thesis a weighting scale from 1 to 10 has been chosen. This is not the scale for the evaluation, it is the scale for the weightings, which should not start with zero. A zero weighting would rule out an attribute in every alternative, which would make it irrelevant! The scale for the following evaluation of alternatives is chosen from 0 to 10.

The Tables 4.6, 4.7, 4.8 and 4.9 show the rating of the four stakeholders.

View List MD		
#	**Keyword**	**Weight**
VLMD1	suitable editor	8
VLMD2	report generation	3
VLMD3	loading time	5

Table 4.6: View list of the model developer

View List PD		
#	**Keyword**	**Weight**
VLPD1	suitable file format	8

Table 4.7: View list of the product developer

View List SA		
#	**Keyword**	**Weight**
VLSA1	user management	7

Table 4.8: View list of the system developer

View List FM		
#	**Keyword**	**Weight**
VLFM1	product price	7
VLFM2	training costs	8
VLFM3	development costs	5

Table 4.9: View list of the finance manager

After the view lists only the weightings between the views are missing. They have been chosen by the project owner and are shown in following Table 4.10:

View's Weighting List	
View	**Weight**
Model developer (PD)	10
Product developer (PD)	5
System administrator (SA)	2
Finance manager (FM)	8

Table 4.10: Weighting list of the different views

Chapter 5

Different approaches to a new tool chain

The following chapters describe different tools, which are candidates for a solution. The descriptions of the properties are focused on the requirements (KOC's) found in the chapters before.

5.1 MagicDraw

5.1.1 MagicDraw 17.0

The first idea that comes up is updating the currently used system for developing the models to the newest version. This would be *MagicDraw 17.0.1 beta 2*, but because of the beta stadium, the version 17.0 is in the focus of this estimation.

Different editions

The software is available in different editions (ordered from most expensive to cheapest, which is congruent to the order of most features to least features):

- Enterprise
- Architect
- Professional C++/C#/Java
- Standard
- Personal

To decide in favor of an edition, it is enough to consider only points related to the requirements, defined in Chapter 4.2. Related points in the feature list are:

- UML support

  - Support for UML 2 metamodel and notation
  - Import of UML 1.4 metamodel
  - Class diagram - includes Package and Objects diagrams
  - State Machine diagram

- UML extensions
  - Customizable stereotypes, constraints, tagged values
  - Ability to assign stereotypes from shortcut menu or type directly near the model element name

- Editor operations
  - Cut/copy/paste elements
  - Manipulations with entities: moving, resizing; copying
  - Automatic class, package, subsystem, message names, attribute, parameter types, and operation return type completion
  - Capability to draw generalization/realization in the opposite direction

- Save/load
  - Support for XMI 2.1. Native files are stored in XMI (XML metadata interchange) format
  - Import for XMI version 1.0, 1.1, and 1.2

- GUI
  - Floating diagram window
  - New Project window is the single place to start different types of projects: Blank Project, New Project from Existing Source Code, New Project from Template, Use Case Project
  - Themes for Swing GUI. Includes *MagicDraw* and *Big MagicDraw themes*

- Reports
  - The type of template files that the Report Wizard supports: normal text, RTF, Open XML (DOCX, XLSX, PPTX), HTML, Spreadsheet template (need to be saved as HTML format), and XML template (DocBook or FO) files
  - Diagram images embedded in reports: SVG, EMF, WMF, JPG and PNG formats
  - Generate reports from console without running *MagicDraw*

- IDE Integrations
  - Integration window allows integrating *MagicDraw* with multiple IDEs on the first startup.
  - Seamless integration with *Eclipse* 3.1 or later(JDT or Java IDE)

  - Integration with *Eclipse* Workbench

- Languages
  - GUI is available in these languages: English (US), German, Japanese, French, Russian and Thai

A full feature list of the different versions can be found under reference [Maga].

The cheapest edition of *MagicDraw 17.0*, which satisfies all these points, is the "Standard Edition".

Evaluation and tests

MagicDraw is able to integrate itself seamless into *Eclipse*. After integration it provides a graphical UML editor inside the *Eclipse* GUI Framework. If a file is opened through the file browser, for the first time after starting *Eclipse*, it loads and starts *MagicDraw* automatically in the background; hence *MagicDraw* can be used as a normal plugin. Also the connection to the *Teamwork Server* can be directly used inside *Eclipse*.

Unfortunately the newest version 17.0.1 beta2 supports only *Eclipse 3.7* or later, older versions are officially not supported. For older *Eclipse* versions, like *Helios*, the *NoMagic Support Center* recommends the *MagicDraw* Version of 17.0 for integration. Actually, the integration of 17.0.1 beta2 into *Eclipse 3.6* has been tested, and both the graphical editor and the *Teamwork Server* worked.

Although integration into the *Eclipse* GUI is possible, it has some disadvantages. Starting the *MagicDraw* plugin for the first time requires some time, around five minutes. The official statement in an Email (20.09.2011) of the *NoMagic* support to this issue is: "Due to performance during starting *MagicDraw* within *Eclipse*, this problem is already known for us. This issue has been added into our defect list and we will try to fix it in the future *MagicDraw* releases."

MagicDraw provides an API for external control. This feature allows other programs to use *MagicDraw* and to remote control its features.

As mentioned in Chapter 3.2.1 the database model is split into several files. These submodel files can differ in their size. Because of the fact, that the linked file consists of the sum of all submodel files, it is implied that no submodel file can be bigger than the linked file. Therefore, to test the usability of *MagicDraw 17.0* with realistic sizes of model files, the linked database file can be opened. As a result of this test, after importing the file, which took about a minute, there were no inappropriate delays or frozen GUI elements, *MagicDraw* seemed to work fine.

Pricing

Independent of the functionality of *MagicDraw*, it has to be considered, that it is not free software. The chosen edition, which fits all the requirements, also depends on the type of license and a so called "Software Assurance". This assurance is a one or multiple year agreement between *NoMagic* and the end user in which *NoMagic* provides the latest *MagicDraw* versions, bug fixes and support.

The different license types are:

- *Standalone license:* This license allows using *MagicDraw* on one machine where it is installed.
- *Mobile license:* This type of license allows the owning user to install it three times. Only one instance at the same time can use the license.
- *Floating license:* This license type allows the owner to install and use the same license on multiple machines independent of the user. Every license using computer must have a network connection to a license server. If it is used on a laptop, an internet connection to the server is mandatory. The license server is included with the first floating license and is free of charge.

The following Table 5.1 shows the listing of prices depending on the assurance duration and the type of license:

		Assurance duration		
Type of license	License	1 year	2 years	3 years
Standalone	€ 531,00	€ 106,00	€ 212,00	€ 318,00
Mobile	€ 631,00	€ 126,00	€ 252,00	€ 378,00
Floating	€ 849,00	€ 170,00	€ 340,00	€ 510,00

Table 5.1: Prices of *MagicDraw 17.0* depending on different license types and assurance durations

It is also possible to buy assurances separately. Assurances include version updating; therefore it is a cheaper way of getting the newest version of the program than buying a new license. Currently *Salomon Automation* owns nine licenses of type "Standalone" and two licenses of type "Floating" for *MagicDraw 9.5*, all for the Standard edition of the client.

The price list of the separately buyable software assurances is shown in Table 5.2.

5.1.2 Teamwork Server 17.0

Evaluation and tests

The Teamwork Server (*TWS*) still contains its features of team synchronization, concerning locking single parts in diagrams, to let different users work simultaneously in the same diagram. Additionally to the old functionality, this release also supports branching. It works similar to other repository systems like SVN, as it consists of trunks, branches and

	Assurance duration		
Type of license	1 year	2 years	3 years
Standalone	€ 170	€ 276	€ 382
Mobile	€ 202	€ 328	€ 454
Floating	€ 272	€ 442	€ 612

Table 5.2: Prices for separately bought software assurances concerning *MagicDraw*

tags.

There are three different storage systems available to choose:

- Native storage system
- SVN
- Clear Case

In this case, only the first two repository systems are of interest for the new tool chain. Git is not supported.

Figure 5.1 shows an extended project view with a trunk and a branch. This view is intended to manage the whole repository. It is possible to create new branches or open every version of every branch, including the trunk. Also every single version can be set as the newest one.

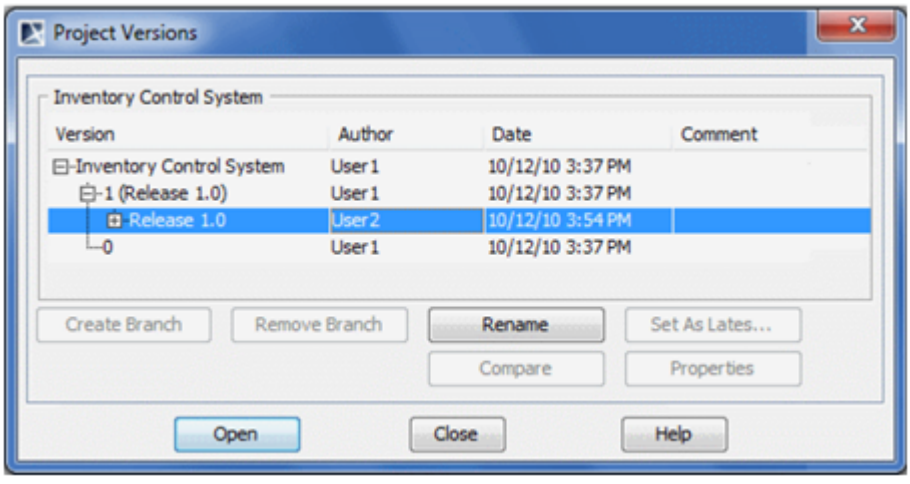

Figure 5.1: Tree view of an example repository including all versions of the trunk and a branch

Unfortunately it has to be pointed out that a branch in SVN is not the same as a branch made by the *Teamwork Server*. The *Teamwork Server* copies files to different folders in one SVN branch or version, therefore using branching with files in an SVN repository is

kind of having a two-level repository system. SVN manages the basic structure of file version control, but the files and folders inside the SVN repository itself represent a repository system, managed by the TWS. Therefore a common storage of the source code files and the model files is not possible.

Another benefit of the new version of the TWS is the support of an external user management system. By using *Lightweight Directory Access Protocol* (LDAP) it is possible to connect the TWS with the user data of the Windows and Linux Networks, therefore extra user names and passwords are avoidable. This feature saves managing time, as no extra user management system has to be supervised.

The *Teamwork Server* is available for Windows, Mac and Linux versions.

More detailed information to the *Teamwork Server* can be found under reference [Magb].

Pricing

Also the *Teamwork Server* is not free of charge. The license needed is depending on how many simultaneous connections of clients should be allowed and, of course, the duration of the so called "Software Assurance". The following Table 5.3 shows the listing of the prices.

		Assurance duration		
Quantity of simultaneous connections	License	1 year	2 years	3 years
up to 5	€ 1.586	€ 317	€ 634	€ 951
up to 10	€ 3.174	€ 635	€ 1.270	€ 1.905
more than 10	€ 6.349	€ 1.270	€ 2.540	€ 3.810

Table 5.3: Price list of the *Teamwork Server 17.0* depending on the quantity of simultaneous connections and assurance durations

Currently Salomon has a *Teamwork Server 9.5* license for more than ten connections and also here it is possible to buy assurance separately for updating to the newest version. The prices of separately assurances are shown in Table 5.4.

	Assurance duration		
Quantity of simultaneous connections	1 year	2 years	3 years
up to 5	€ 508	€ 825	€ 1.142
up to 10	€ 1.016	€ 1.651	€ 2.286
more than 10	€ 2.032	€ 3.302	€ 4.572

Table 5.4: Prices for separately bought software assurances concerning the *Teamwork Server*

5.1.3 Model Merge Plugin

Evaluation and tests

The *Teamwork Server*'s ability to create and manage branches would not be a real big advantage without the ability to merge different branches together by tool support. Fortunately, *NoMagic* has a solution for this problem.

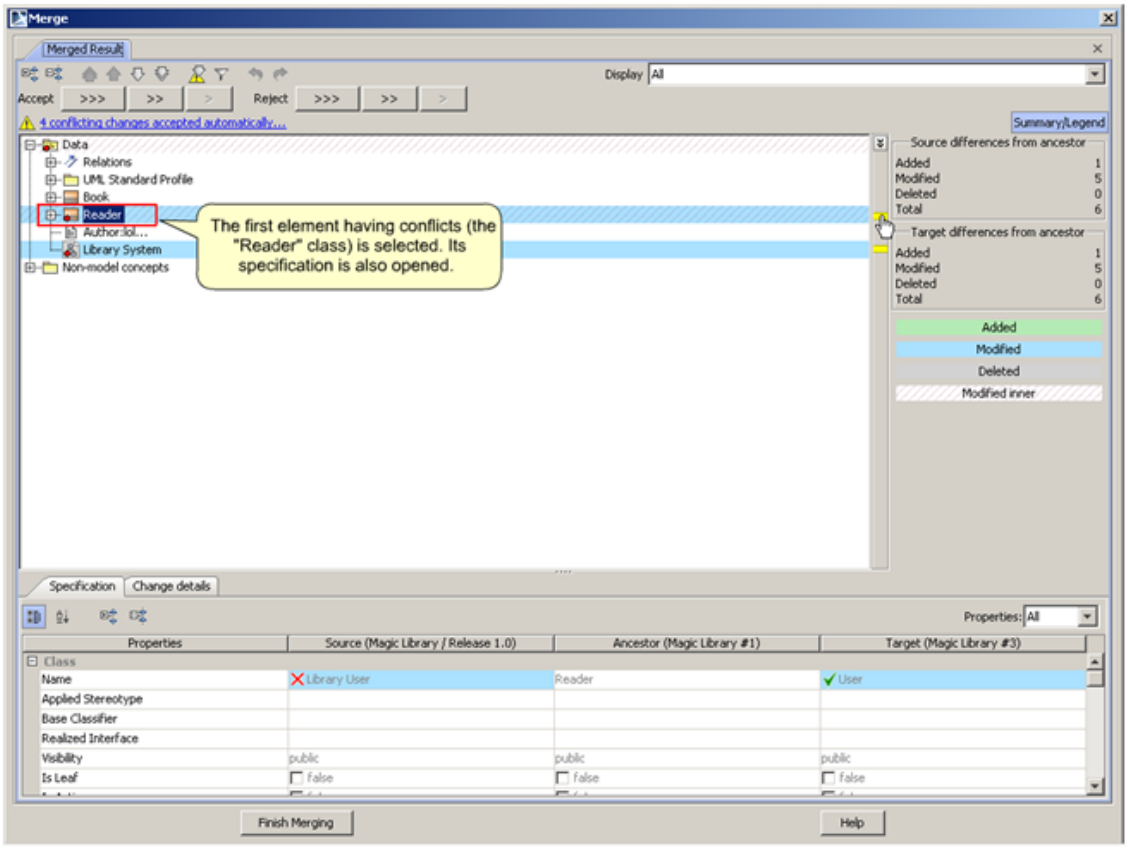

Figure 5.2: The "merge view" of two versions of the same model

The *Model Merge Plugin* enables both 2-way and 3-way merging of model files. Both local projects and remote projects inside a version control system can be processed. This plugin serves the ability to create differences between two different models, i.e. two branches, and can store it in a file.

As the name of the plugin implies, it is also possible to merge changes between different versions of models. Similar to text based merge windows, the "merge view" of the plugin graphically points out differences. These conflicts can be solved from a user by accepting suggestions of the merge algorithm or set merge actions manually.

Figure 5.2 shows an example of this "merge view". In this example a class, originally named "Reader", has been renamed in two branches of the same model (actually one of

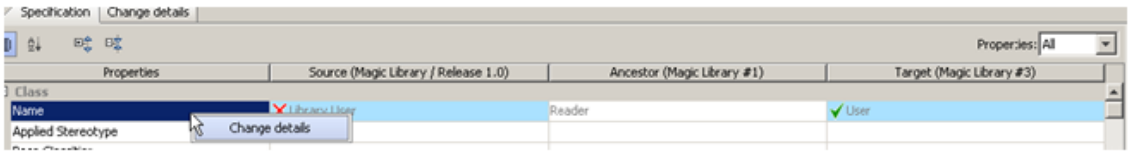

Figure 5.3: Example for changing an automatic merge suggestion

the two branches is the trunk). The *Merge Plugin* suggests a solution to this conflict by taking the change of the target, in this case "User", indicated by the green check mark in the specification tab at the bottom of the window.

The suggestions of the merge algorithm do not necessarily have to be right. By right-clicking on the conflict property the selection can be changed, which is shown in Figure 5.3.

For the sake of completeness, also an example with merging of state machine diagrams has been tested with the same positive results.

Pricing

The price of this plugin can be chosen similar to the *MagicDraw* license. The list of prices is shown in Table 5.5.

	Assurance duration		
Type of license	1 year	2 years	3 years
Standalone	€ 493,00	€ 575,00	€ 657,00
Mobile	€ 642,00	€ 749,00	€ 963,00
Floating	€ 790,00	€ 922,00	€ 1.054,00

Table 5.5: Prices of *Merge Plugin* with different license types and assurance durations

Although Salomon Automation does not own any *Merge Plugin* licenses, the prices for separately bought licenses may be interesting for prolongation or future update purposes. These prices are shown in Table 5.6.

	Assurance duration		
Type of license	1 year	2 years	3 years
Standalone	€ 132,00	€ 214,00	€ 296,00
Mobile	€ 171,00	€ 278,00	€ 385,00
Floating	€ 211,00	€ 343,00	€ 475,00

Table 5.6: Prices of separately bought assurances for the *Merge Plugin*

5.1.4 MagicDraw Project Converter

NoMagic provides a free converter, which is able to open *MagicDraw* projects in the version 9.x or earlier and save it to a *MagicDraw 17* project. This program is standalone and is therefore independent of the usage of *MagicDraw 17* as an editor.

Testing the *Project Converter* with the linked database model file might lead to a message that more memory for converting should be allowed. After adapting this setting, the generated output looks fine and also the duration of conversion is acceptable.

5.1.5 Summary

The *NoMagic* products do fit all requirements, as shown in Table 5.7. Therefore it provides a solution for a new tool chain. It will be considered as a candidate for a possible solution in the following detailed view list calculation. Therefore, the view list points have to be evaluated, see Table 5.8.

KOC1	✔
KOC2	✔
KOC3	✔
KOC4	✔
KOC5	✔
KOC6	✔
KOC7	✔
KOC8	✔
KOC9	✔
KOC10	✔
=	✔

Table 5.7: Fitting of KOC's using *MagicDraw* with *Teamwork Server* and *Merge Plugin*

VLMD1	9
VLMD2	10
VLMD3	3
VLPD1	8
VLSA1	9
VLFM1	4
VLFM2	5
VLFM3	4

Table 5.8: Fitting of VL-points using *MagicDraw* with *Teamwork Server* and *Merge Plugin*

5.2 Visual Paradigm

5.2.1 Visual Paradigm for UML 8.3

Different editions

Visual Paradigm (VP) is a graphical UML designing tool. Similar to *MagicDraw* it consists of a GUI for developing the UML diagrams. It is also able to define own stereotypes.

There are several different editions of Visual paradigms. The cheapest one, which fits all requirements, can be chosen. These requirements are:

- UML Modeling
  - Class diagram
  - State machine diagram
- Modeling Toolset
  - Visual Diff
  - UML Profile support
- Documentation Generation
  - Generate HTML document
- Team Collaboration
  - Subversion collaboration
- Interoperability & Integration
  - XMI import and export
  - UML2 model import and export
  - Command-line operations
- Supported Standards
  - UML
  - XML Metadata Interchange (XMI)

On the web page [Par] a list of all these points implemented that can be considered as an edition filter can be found. Comparing these points with the requirements lead to three different editions, the cheapest one of them is the "Standard Edition".

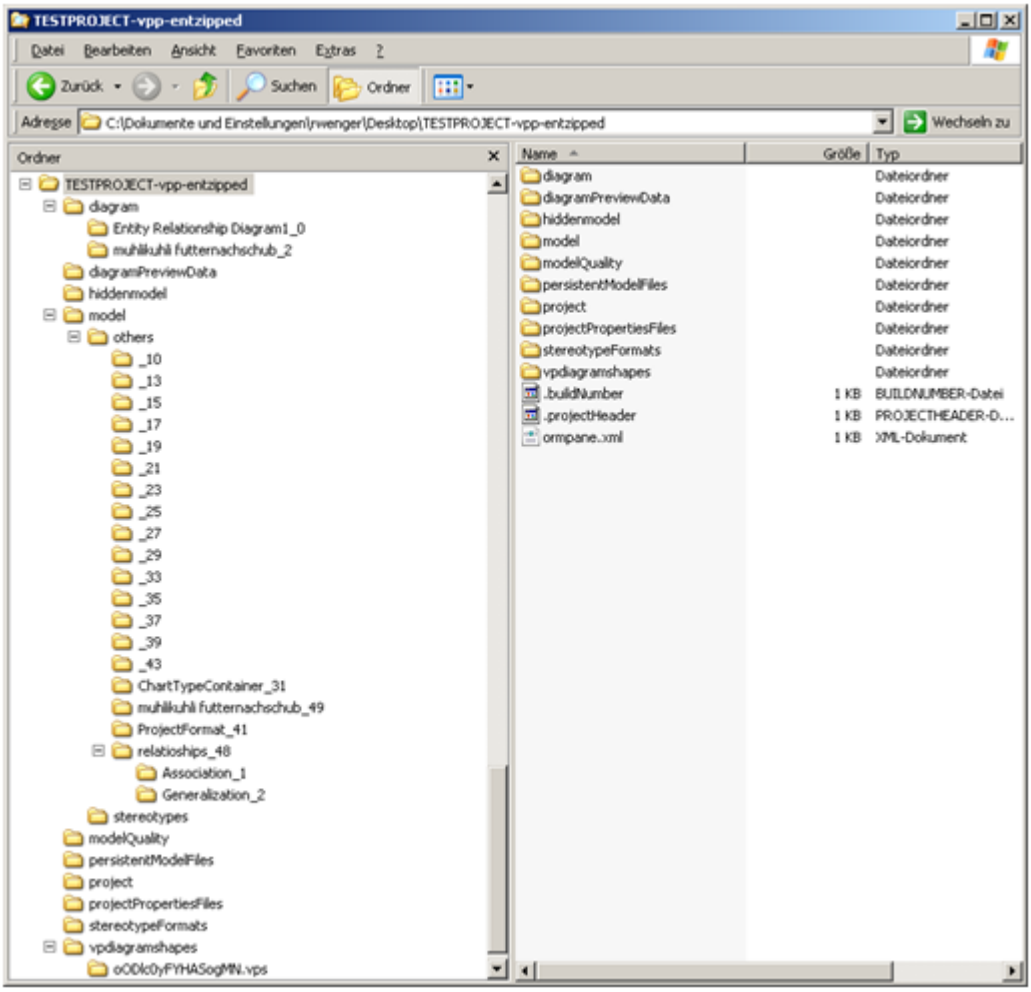

Figure 5.4: Extracted content of *Visual Paradigm*'s native file format using a ZIP-program

Evaluation and tests

As a small disadvantage *Visual Paradigm* by default saves its projects in an own file format, called "Visual Paradigm Project" (*.vpp). Opening the file with a text editor leads to an unreadable text, but opening the file using a ZIP-program shows that the file format consists of a packed folder containing many subfolders and files, see Figure 5.4.

Taking a deeper look on the name of these folders and files the interesting information can easily be discovered. For example all classes of this project can be found in a folder called "model", its subfolders are named like the diagram and contain the defined classes. Reading these files with a simple text editor shows an obvious coding of the information. They seem to be text files with a simple encoding, but for a safer statement, these files have to be analyzed in a more specific way.

To finally sum up this issue, there is a very good chance to read the native file for-

mat of *Visual Paradigm* without a conversion into an interchangeable model file format. However, the software is capable of exporting the projects to XMI 2.1. Figure 5.5 shows the dialog window with the necessary settings. This feature is not only usable through the GUI, but also can be called by command line. Therefore, as a plan B, it is possible to "read" the native file format of *Visual Paradigm* by automatically converting the native file to a temporary file, containing XMI 2.1 and parse this one.

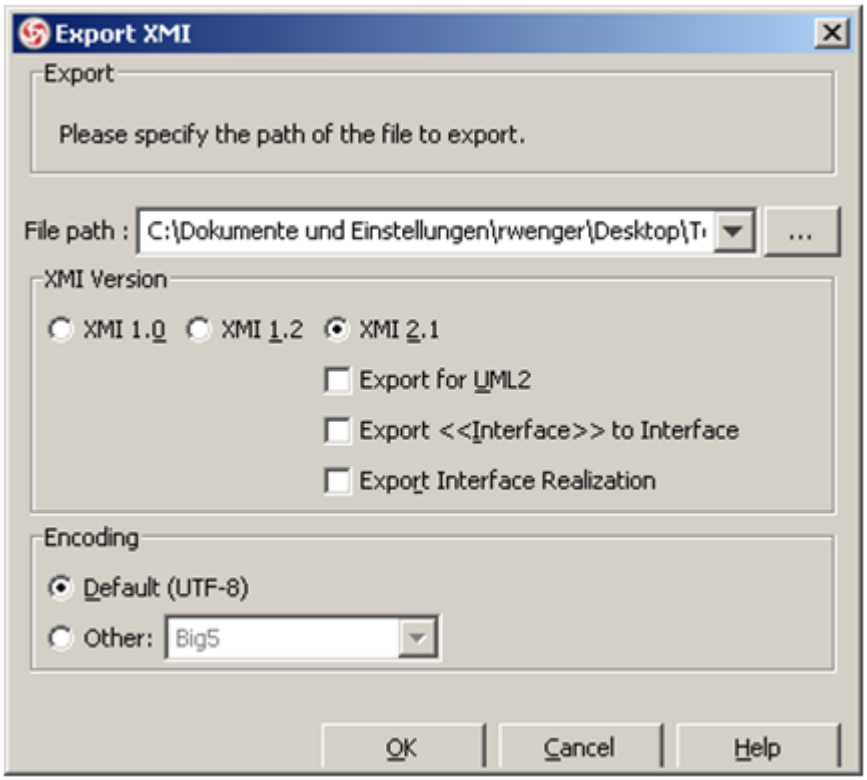

Figure 5.5: Dialog for exporting project into XMI2.1 file format

Pricing

The price of the "Standard Edition", which supports all necessary features, depends on the type of license. There are three different license types:

- *Single-Seat License:* This type of license allows one specific user to install VP on up to different three computers. Simultaneous usage or usage by a different user is not allowed.
- *Floating License:* This type of license is similar to the "Single-Seat" type, but allows anyone to use the license. A license server manages the number of licenses and assigns everyone, who wants to use the client license, until the number of licenses is reached.
- *Site License:* Under this type of license anyone within the organization can use the software, as the number of licenses is not fixed.

The price of one unit is also depending on the amount of the order. Within a quantity of five to nine licenses, the customer gets a discount of 5%, between ten and 49 licenses a discount of 10%. To get information about discounts when ordering an amount of more than 50 licenses, a request to the sales support is necessary.

Additionally it is possible to buy maintenance for one year. This means that receiving updates for the chosen product are for free within one year. The price is about 20% of the product.

Regarding to this information, the following table shows the prices of the different opportunities, which can be considered. All prices are in US Dollar and for one unit. Every year extending the maintenance costs about 20% of the product list price, buying maintenance separately costs about 30%. It is also possible to buy a site license, which includes one year maintenance and costs about $ 28.000. It would be usable by anyone in the company at the same time.

	Single-Seat License		Floating License	
Quantitiy	no maintenance	1 year maintenance	no maintenance	1 year maintenance
1-4	$ 299,00	$ 358,50	$ 388,50	$ 466,00
5-9	$ 284,05	$ 340,86	$ 369,27	$ 443,12
10+	$ 269,10	$ 322,92	$ 349,83	$ 419,80

Table 5.9: Prices of *Visual Paradigm* depending on the type of license, the quantity and the maintenance option

5.2.2 Teamwork Server 5.3

Evaluation and tests

Visual Paradigm integrates team support for model files by a *Teamwork Server*. It can use a native type of repository or can be configured with SVN, but Git is not supported. Differently to some other UML tools with team support, *Visual Paradigm* really uses the SVN API for branching. In other words, creating a new branch through *Visual Paradigm* results also in a new branch in e.g. Tortoise SVN and vice versa. All team features only work with the *Visual Paradigm* file format *.vpp and not with exported files like XMI.

Visual Paradigm's *Teamwork Server* supports the usage of LDAP and next to the Windows versions, there is also an explicit version for Linux available.

As a very useful tool, *Visual Paradigm* provides a feature called "Visual diff", which can be found under menu point "Tools". With this tool it is possible, to compare two diagrams in their graphical notation. The differences are listed and connected to the graphical representation in two frames, containing the diagrams to compare. The settings for comparison can be changed in the upper part of the window. Figure 5.6 shows an example of this supporting tool. This feature is only usable for locally saved projects.

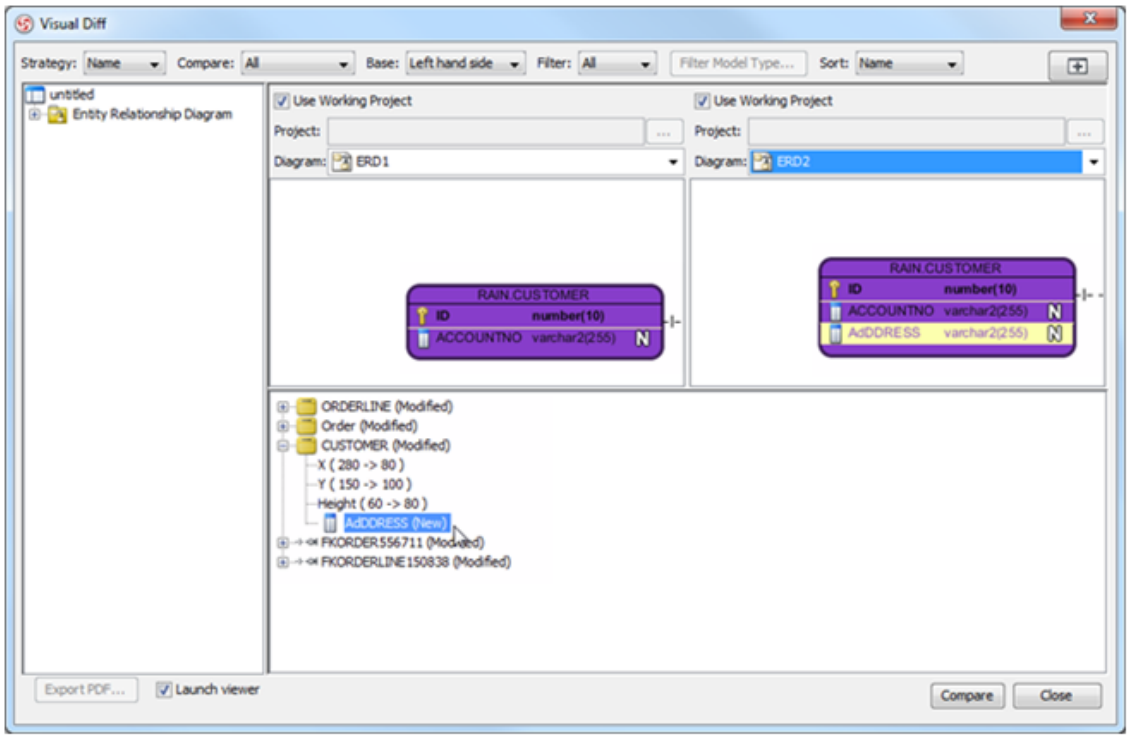

Figure 5.6: Example of two versions of a class diagram, compared with the "Visual diff" feature

In addition a merging feature is provided and can be found on the toolbar under "Team". The merge functionality lists all differences between two versions and a common ancestor, if available, and points them out in a dialog window. It supports the usual features like accepting suggestions of the merging algorithm, declining and reverting original values.

Figure 5.7 shows an example of a merge conflict of two branches. The original name was "Watch a stock", and then two branches were made. In one branch the name has been changed to "Keep track of a stock", in the other branch to "Monitor a stock". The first merge action, between the trunk and the second branch did not bring up any problems. To avoid misunderstanding, the user also has to accept the changings manually, but this is not a very interesting act, as every change simply can be accepted. The more interesting case is the next merge attempt of the first branch with the modified trunk. Of course, this causes a problem, as it cannot be decided, which change should be accepted and which should be declined. The merge window shows this conflict and provides the possibilities to keep the current value, change to the conflict value or even revert it to the value of the common ancestor.

For the sake of completeness, also an example with merging of state machine diagrams has been tested with the same positive results.

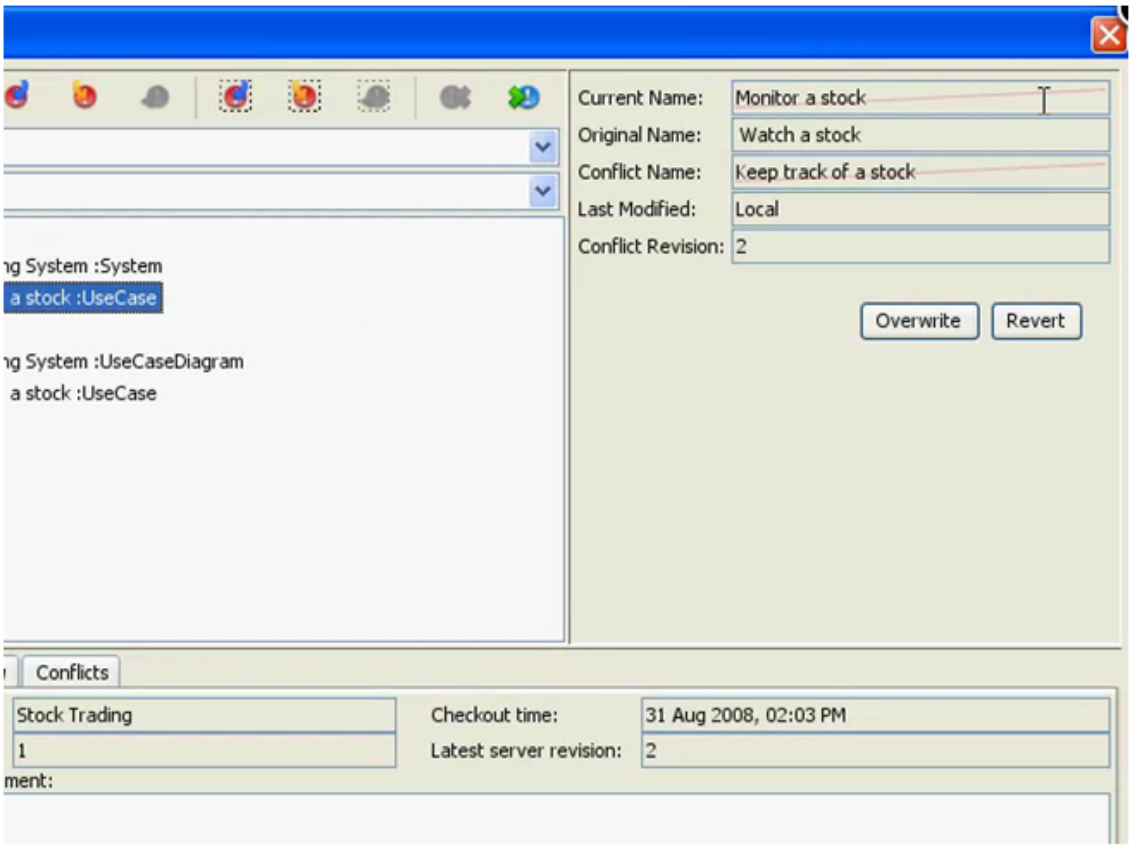

Figure 5.7: Example of a merge conflict in *Visual Paradigm*

Migration of model files of the current tool chain can be achieved in two slightly different ways. The first possibility is to unzip the files and import them directly into *Visual Paradigm* by "File → Import → XMI". For some reason this method causes the loss of the class diagram's names and all stereotype information. The second variant of migration uses the *NoMagic* Project Converter. It can take the old files in *.xml.zip or *.xml format. The output depends on the input format; therefore it must be unzipped before or after usage of the converter to get an XMI file. After converting, the model files can be imported by *Visual Paradigm* in the same way as mentioned before.

Visual Paradigm supports two different ways to get external access. First it supports command line actions, like importing and exporting projects to a specific file format or generating HTML reports. The second possibility to get remote control of *Visual Paradigm*'s features is a plugin support. VP allows external access to all commands, like in the menu bar available, by implementing a plugin written in Java and using a VP-API-Java library. Thus it is possible to handle the team support by a self-developed check-out and check-in tool and a plugin, which delegates the actions to *Visual Paradigm*'s built in features.

Similar to the tools before, this tool was also tested with the linked model file of the database use case. Because of the loss of information using the direct import function, the *MagicDraw Project Converter* was used first. Importing the converted file then allowed a normal usage, without any frozen GUI elements or inappropriate waiting durations.

Pricing

The version of the *Teamwork Server*, that fits all requirements, like a repository for files and concurrent modeling, is the "Corporate" edition. Its price depends on the amount of licenses and the duration of maintenance. The following Table 5.10 shows the list of prices. All values are in USD.

Quantitiy	no maintenance	1 year maintenance
1-4	$ 2999,00	$ 3598,50
5-9	$ 2849,05	$ 3418,58
10-49	$ 2699,10	$ 3238,65

Table 5.10: Prices of *Visual Paradigm* depending on the type of license, the quantity and the maintenance option

Every year extending maintenance costs 20%, a year maintenance separately bought costs 30%.

5.2.3 Summary

Visual Paradigm and the *Teamwork Server* do fit all requirements and provide a solution for a new tool chain. The fitting of requirements is shown in Table 5.11. Therefore it will be considered as a candidate for a possible solution in the following detailed calculation of evaluation points.

5.3 Microsoft Visio 2010

Evaluation and tests

Microsoft Visio 2010 is a commercial graphic editor for UML diagrams, business workflows and many more. It is capable of connecting to other *Microsoft Office* based files for interdisciplinary data sharing.

Visio is also able to create documentation in other Microsoft Office file formats, HTML and more. Code Generation is unfortunately not possible, as it is just a drawing tool. But it supports UML2 as well as exporting the models into XMI file format using an add-on called "UML Background Add-On".

KOC1	✔
KOC2	✔
KOC3	✔
KOC4	✔
KOC5	✔
KOC6	✔
KOC7	✔
KOC8	✔
KOC9	✔
KOC10	✔
=	✔

Table 5.11: Fitting of requirements using *Visual Paradigm* with *Teamwork server*

VLMD1	9
VLMD2	10
VLMD3	8
VLPD1	8
VLSA1	5
VLFM1	8
VLFM2	5
VLFM3	9

Table 5.12: Fitting of VL-points *Visual Paradigm* with *Teamwork Server*

The support of team development is limited to distributing models in only readable versions. They even can be opened without *Microsoft Visio*, but are not able to be modified. This feature uses the *Microsoft Sharepoint Server*.

Pricing

Microsoft Visio 2010 is not part of any *Microsoft Office 2010 Suite*, although it has the same GUI strategy and provides very high support for the other programs of the office suite. It is available as standalone program and the premium version costs about €1.299 per license.

5.3.1 Summary

The leak of the necessary team features clearly rules out *Microsoft Visio 2010* as a possible solution. Table 5.13 shows the fitting of requirements.

KOC1	✔
KOC2	✘
KOC3	✘
KOC4	✘
KOC5	✔
KOC6	✔
KOC7	✔
KOC8	✘
KOC9	✘
KOC10	✔
=	✘

Table 5.13: Fitting of requirements using *Microsoft Visio 2010*

5.4 IBM Rational Software

5.4.1 IBM Rational Software Architect

Evaluation and tests

The *IBM Rational Software Architect* (RSA) is a part of the *Rational Rose Suite* and is intended to support architects and designers to produce UML models. It is completely based on *Eclipse 3.6* (*Helios*).

The installation of the software is possible as standalone program or as a plugin for an already installed *Eclipse* version (*Helios* or later). Before any of the two versions can be installed, an "Installation Manager" is needed, which is designed to manage all installations of IBM products. To use RSA as a plugin, the fundamental *Eclipse* program has to be extended by some plugins and patches.

Among other types of description diagrams RSA supports UML2 diagrams, containing class and state machine diagrams. Examples for these two diagrams modeled with RSA are available in Figure 5.8 and 5.9.

The file format for its models of the *Rational Software Architect* is based on XML to satisfy UML2.0 requirements for its interconnections and semantics. It supports two types of models: single and logical. Single models reside in single files. The logical models are divided into so called fragments, that are logically related, even if they are physically separated. In *Rational Software Architect*, these subunits or fragments are stored in *.efx-files that are referenced by a main model file, saved with the file extension *.emx. However, there seems to be a good chance to directly read the native file format. As an alternative, the program supports exporting files into format XMI 2.2.

RSA also offers possibilities for importing files. Unfortunately it only supports the import of XMI 2.2. This fact would make another conversion tool necessary, to ensure a migration of the existing files into RSA.

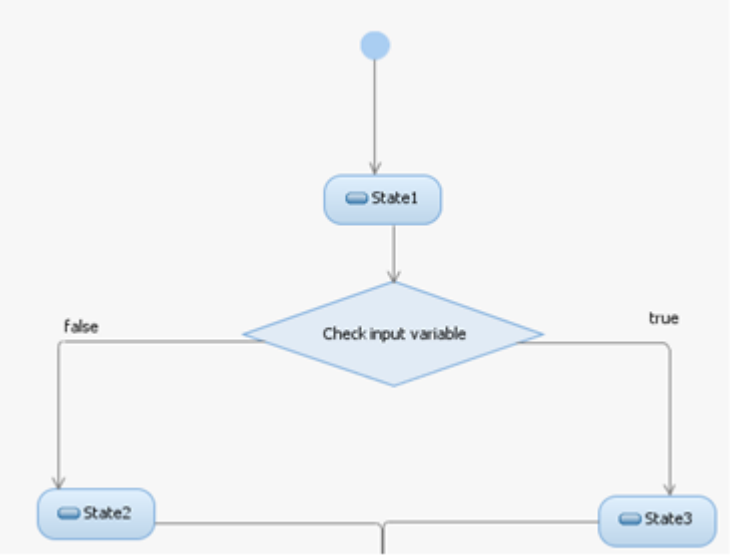

Figure 5.8: Example for a simple state machine diagram designed with RSA

The *Rational Software Architect* also provides a possibility for external control. It offers the concept of so called "pluglets", which allows external programs and tools to use RSA and its features.

Pricing

RSA offers two different license types, which are defined as follows:

- *Authorized user license:* This kind of license allows an installation on any number of computers and each *Authorized User* may have simultaneous access to any number of instances of the program at one time. An entitlement is unique to that person and may not be shared, nor may it be reassigned other than for the permanent transfer to another person.
- *Floating user license:* The program may be installed on any number of computers, but if the user simultaneously accesses more than one installation another license is required.

Independent to which kind of license, it is only for a usage of 12 months with included service. After end of the duration, another license must be bought. Therefore it can be seen as an annual rent of the software. Table 5.14 shows the prices for one year including support depending on the kind of license. All values are in EUR.

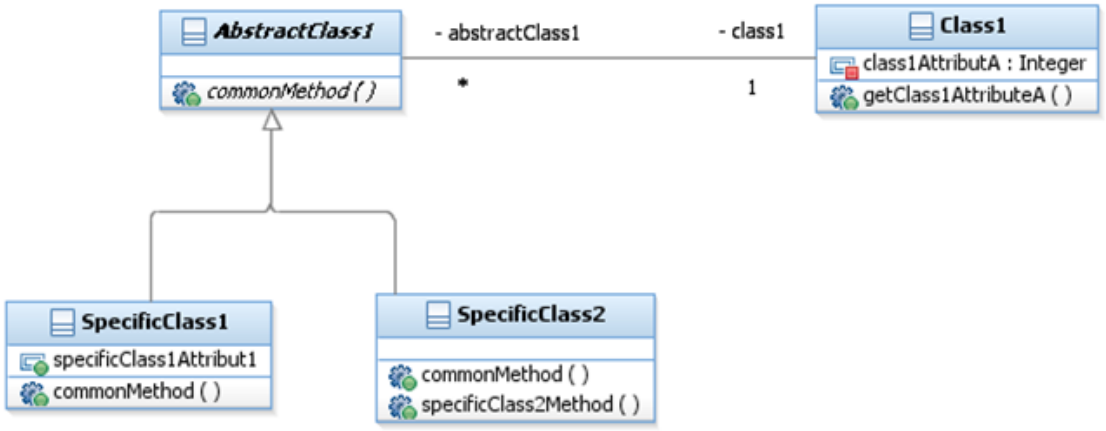

Figure 5.9: Example of a simple class diagram designed with RSA

Authorized user license	Floating user license
€ 529,20	€ 1.059,60

Table 5.14: Prices of different licenses of *Rational Software Architect*

5.4.2 Rational Team Concert

Evaluation and tests

Rational Team Concert (RTC) is built on client-server architecture and includes source control, reporting and support for build management. The product is optimized for small and medium sized teams and is capable of being integrated into a wide range of other products. It consists of a server program and clients, which can be included as features into other *IBM Rational* programs, like RSA.

With the *Rational Team Concert* software bundle it is possible to branch and merge models. It works with either a native repository system or a given one, like Subversion. For branching, the included methods of SVN can be used, and for merging, *Rational Team Concert* offers two possibilities. Depending on which file format in RSA has been chosen, either a logical compare or non-logical compare can be performed. An example of a visual logical comparison is shown in Figure 5.10.

Finally it has to be mentioned, that the amount of features aim a more complex purpose than this project requires. It targets a full support for Scrum, continuous building and integration and much more.

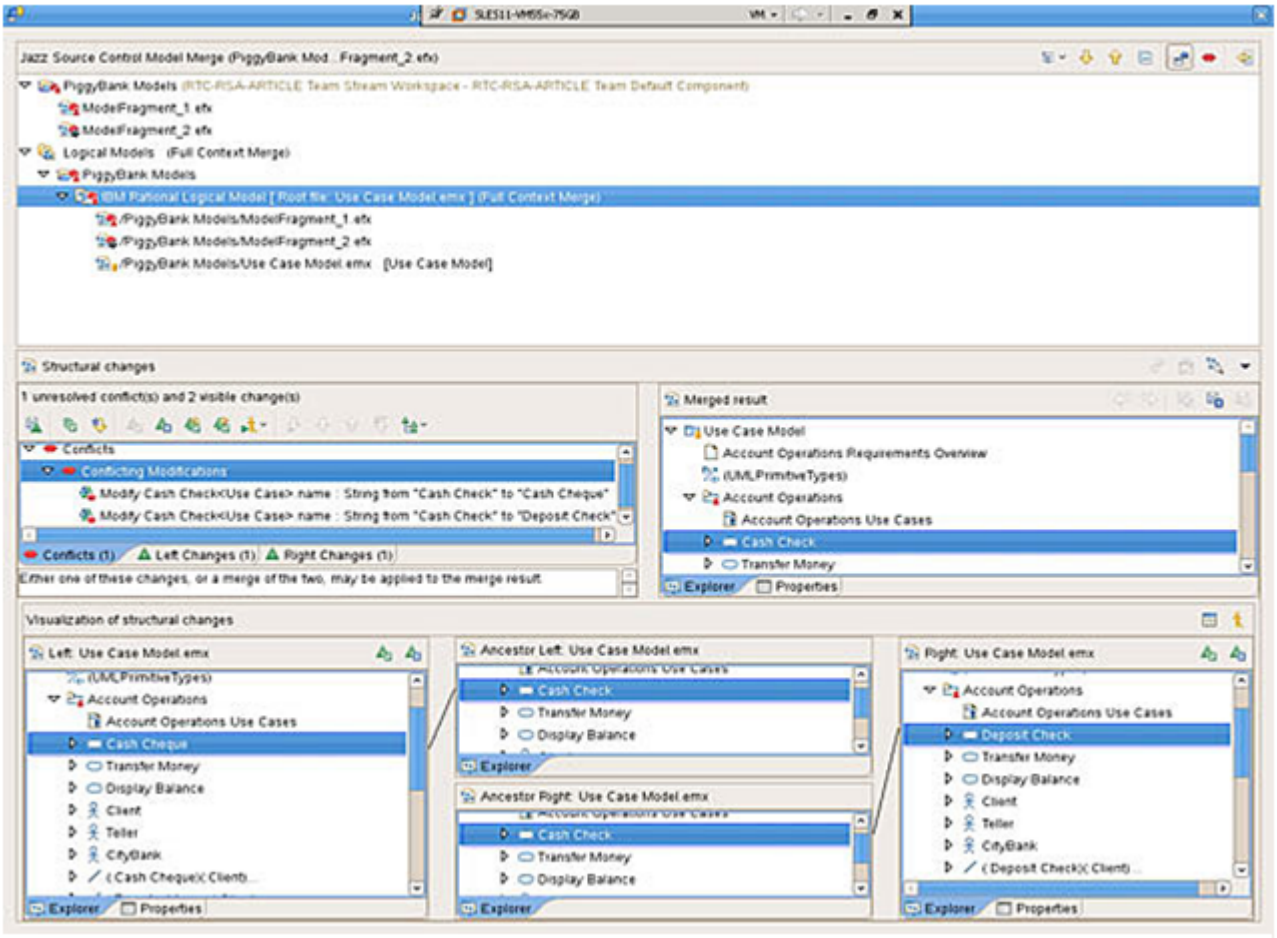

Figure 5.10: Example of a logical compare of a UML model

Pricing

The following table lists the prices of Rational Team Concert for the respective license type.

Authorized user license for Workgroups	Floating user license for Workgroups
€ 1.970,00	€ 6.180,00

Table 5.15: Prices of different licenses of *Rational Team Concert*

5.4.3 Summary

IBM's *Rational Software Architect* and *Team Concert* fit all requirements, as it is shown in Table 5.16. It allows users to use full team support for model files by using an own repository and provides also external access for the files. The only negative point that could be mentioned is the intentional purpose of the RTC. It has many more features than

necessary for Salomon's needs. Although it might be oversized, the software is evaluated for the further process (see Table 5.17).

KOC1	✔
KOC2	✔
KOC3	✔
KOC4	✔
KOC5	✔
KOC6	✔
KOC7	✔
KOC8	✔
KOC9	✔
KOC10	✔
=	✔

Table 5.16: Fitting of requirements using IBM's *Rational Software Architect* and *Team Concert*

VLMD1	9
VLMD2	8
VLMD3	6
VLPD1	6
VLSA1	5
VLFM1	1
VLFM2	5
VLFM3	9

Table 5.17: Fitting of VL-points using IBM's *Rational Software Architect* and *Team Concert*

5.5 Enterprise Architect

5.5.1 Enterprise Architect 9.1

Different editions

The *Enterprise Architect* (EA) is a model designing tool manufactured by *Sparx Systems*. The software is available in different editions:

- Desktop
- Professional
- Corporate
- Business and Software Engineering

- Systems Engineering
- Ultimate

These editions differ mainly in the supported features, and of course in the price. The interesting points of the feature compare list are:

- Advanced UML 2.3 Modeling
- Automation API
- Report Generation: HTML and Rich-Text
- Version Control Integration
- XMI Import and Export (2.1, 1.2, 1.1, 1.0)
- Shared Models
- Baseline Diff/Merge
- Lazy Load
- OMG XMI
- *Eclipse* Integration

The cheapest edition, which supports all these features, is the so called "Business & Software Engineering" edition.

Evaluation and tests

EA provides an intuitive GUI for designing UML2 diagrams, such as state machine and class diagrams. Figure 5.11 and Figure 5.12 show examples for diagrams build with this editor.

Enterprise Architect also provides powerful document generation and reporting tools with a full WYSIWYG ("What you see is what you get") template editor. Amongst others also HTML documentation of models is available.

Migration of the current model files is also possible. EA supports the XMI versions of 1.0, 1.1, 1.2 and 2.1. Unfortunately the trial version does not support import and exporting of models to XMI, therefore only the product's description site, available under reference [Sys], can be used as an information source. According to this description, it supports every important requirement:

- Support for XMI 1.0, 1.1, 1.2 and 2.1
- Export complete EA models to XMI
- Standard XML for use by 3rd party tools, such as MDA generators and report writers

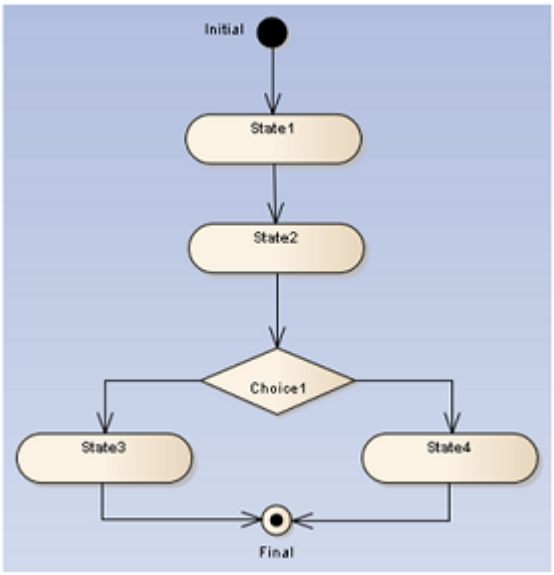

Figure 5.11: Example for a state machine diagram modeled with EA 9.1

- Import from other XMI compliant tools in UML 1.1, 1.3 or 2.x format

EA also supports a version control system and a diff/compare-feature. The version control system allows storing any compliant system of standard XMI text files. These may then be set under version control and distributed to developers, analysts, managers and team members in general, either for inclusion in privately assembled models ("private mode") or as part of the control and management of a shared DBMS ("shared mode"). As a native repository system CVS is used, but also Subversion is possible.

The diff/compare-feature unfortunately leaks of an important feature. As the name suggests it is possible to compare different models and create a difference view. With the aid of this tool it is possible to detect differences, but it is not possible to merge those models. Therefore any differences must be changed in one of the two involved models manually. According to the homepage, also a "Diagram compare is currently not supported", this means, that only elements and not their dependencies can be compared.

Pricing

The price of the *Enterprise Architect 9.1* in the "Business & Software Engineering" edition is depending on the amount of licenses. Additionally it depends on the kind of license ("Standard License" or "Floating License"). The following Table 5.18 shows the different prices. All prices are in US Dollar and for one unit.

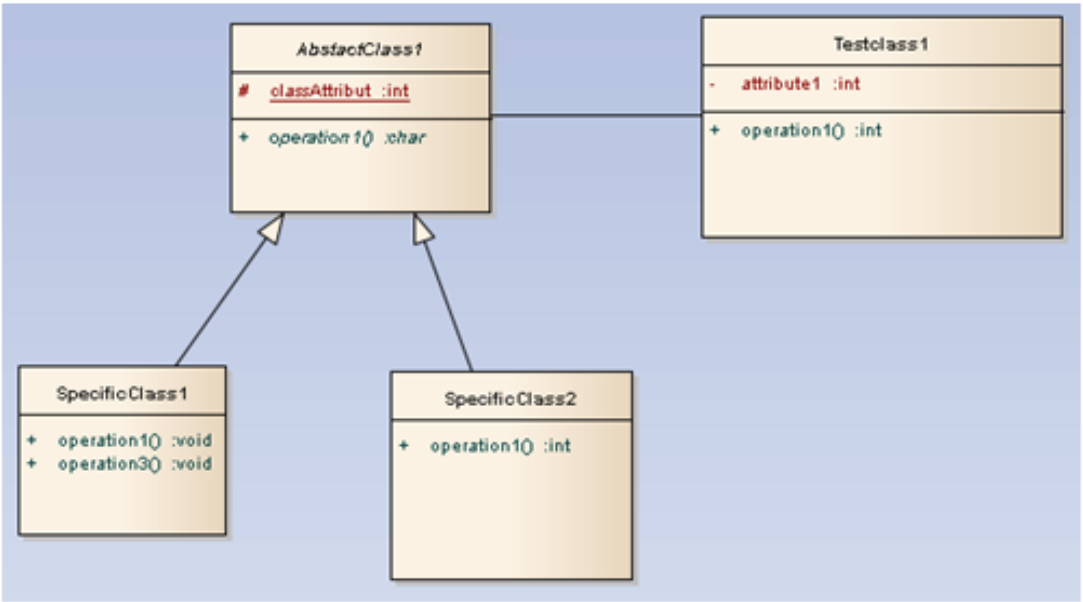

Figure 5.12: Example of a class diagram modeled with EA 9.1

Quantity	Standard License	Floating License
1-4	$ 599,00	$ 799,00
5-19	$ 539,00	$ 719,00
20-100	$ 479,00	$ 639,00
101+	$ 419,00	$ 559,00

Table 5.18: Prices of different *Enterprise Architect 9.1* licenses types and amounts

5.5.2 Integration for Eclipse

Evaluation and tests

Integration for Eclipse allows integration of the *Enterprise Architect* into *Eclipse*. In detail, this *Eclipse* feature does not integrate EA into the *Eclipse* GUI. More or less it only tells a running instance of EA, that it should synchronize a project folder with a given *Eclipse* workspace. In other words, changes that will be saved to a normal project folder would automatically be saved to a folder within the given *Eclipse* workspace too. Therefore, a manual "Save As"-command, as described in Section 3.4.2 and Figure 3.12, would not be necessary any longer.

Pricing

This feature is free of charge.

5.5.3 Summary

Unfortunately the *Enterprise Architect* lacks of some properties and features, which would have been necessary for the Knock-Out-Criteria List (see Table 5.19). Therefore no further evaluation has been executed.

KOC1	✔
KOC2	✔
KOC3	✘
KOC4	✔
KOC5	✔
KOC6	✔
KOC7	✔
KOC8	✘
KOC9	✘
KOC10	✔
=	✘

Table 5.19: Fitting of requirements using *Enterprise Architect*

5.6 Gentleware products

5.6.1 Poseidon for DSLs 2.0

Evaluation and tests

As the name already suggests this is not software especially for UML2. *Poseidon for DSLs 2.0* is an editor, which is able to create editors for own DSL definitions.

Defining a DSL causes some inconveniences. First of all, the DSL has to be developed. Logistics software engineering is a very complex industrial sector; a development of a DSL may lead to much effort.

A second disadvantage is the migration of the current models. If they should be brought to a self-developed language, also the converter has to be self-developed. This also may lead to a high amount of time and manpower to invest.

As a third point, the users of the language must be trained. In case of Salomon these users would be the *WAMAS* developers. Currently they are trained on UML1.

Despite these negative points, a DSL can improve development of future models. Once the language and its editor are finished and the developers are trained, usually the usage of the designing tool can be easier and more intuitive than a solution with a common description language. Also a file format can be chosen, which is capable of being merged by a self-developed merging tool. Those advantages surly depend on the quality of the

developed language, but a well-engineered DSL may turn Salomon to a pioneer on its industrial sector.

Pricing

The price of *Poseidon for DSLs 1.x* is about € 839, the price of the new release 2.0 is not yet available.

5.6.2 Poseidon for UML 8.0

Evaluation and tests

Poseidon for UML is a standalone editor and is based on an editor created by *Poseidon for DSLs*. It supports class diagrams and state machine diagrams as well as all other defined UML2 diagrams. Theoretically it can be self-developed by using *Poseidon for DSLs*, but this would imply to pay more money for the *Poseidon for DSLs* editor and in addition more development time is required to gain the same result.

The native file format has the extension *.uml2 and, referring to the header of the file viewed in a text editor, consists of XMI 2.0 for UML 2.1.

Unfortunately the trial version does not support teamwork. Contacting the support results in no answer, therefore it was not possible to test this feature.

Pricing

The price of Poseidon for UML is €48 per year.

5.6.3 Apollo for Eclipse

Evaluation and tests

Apollo for Eclipse is, similar to *Poseidon for UML*, an editor for UML. As the name already suggests, it is a plugin for *Eclipse*, not a standalone editor.

The minimum *Eclipse* features required for *Apollo* are as follows:

- eclipse-SDK 3.3
- EMF Service Data Objects (SDO) SDK 2.3
- OCL SDK 1.1.0
- EMF Model Query SDK 1.1.0
- EMF Model Transaction SDK 1.1.0

- EMF Validation Framework SDK 1.1.0
- Graphical Editing Framework SDK 3.3.0
- Graphical Modeling Framework SDK 2.0.0
- Apache Batik 1.6.0 (included in GMF-SDK)
- UML2 SDK 2.1.0

Although the version of *Eclipse* used for testing *Apollo* had all these features installed and the installation instructions had been followed, it was not able to integrate *Apollo* and test its features. As already mentioned the support of *Gentleware*, the manufacturer of *Apollo for Eclipse*, just did not answer to give any helping instructions.

A very disadvantageous attribute of *Apollo* is the support of only class diagrams. Also on this topic no information was available through the support.

Pricing

The price of *Apollo for Eclipse* is depending on the payment interval. Paying the rent monthly the costs are about USD 6, quarterly the price is about USD 16 and annually it is about USD 56 (€48).

5.6.4 Summary

As the description above may suggest, many of the necessary points for this project are not fulfilled (see Table 5.20). Therefore, none of *Gentleware*'s products will be considered as a solution candidate.

KOC1	✘
KOC2	✘
KOC3	✘
KOC4	✔
KOC5	✔
KOC6	✔
KOC7	✔
KOC8	✘
KOC9	✘
KOC10	✔
=	✘

Table 5.20: Fitting of requirements using *Gentleware* products

5.7 Papyrus

Evaluation and tests

Papyrus is an *Eclipse* based project for a graphic editor for UML2. The project is available as a plugin, but also as a standalone solution based on the *Eclipse* version 3.5 called *Galileo*.

The assumed newest version of *Papyrus* as a plugin is 1.12.3, which can be found in the *Eclipse Galileo* under "Help → Install new feature...". Using the appropriate update site the UML modeler can be installed. The plugin first needs an external plugin called ANTLR from the same site (Figure 5.13), as well as an *Eclipse* version, which includes the modeling bundle.

For the *Eclipse* version 3.5, called *Galileo*, this solution works fine, unfortunately not for the versions 3.6 (*Helios*) and 3.7 (*Indigo*).

As a following project a component called *MDT Papyrus* is under development. Figure 5.14 shows the appearance of the plugin for *Helios*, but is similar to *Indigo*. As illustrated, the plugin is currently available in the version 0.7.3. In this version some important GUI-functions, like adding a property to a class, are missing, hence a deployment of the *Papyrus* plugin with *Helios* and *Indigo* is not possible yet.

Although it is not (yet) possible to use *Papyrus* as a graphic editor inside *Eclipse Helios*, it can be considered as a standalone program. It saves the models in files with the extension *.uml. According to the headers inside the file, it is the format of XMI 2.1 using *Ecore* as a meta model definition for UML2.

Pricing

As an project running under the *Eclipse license*, it is free of charge.

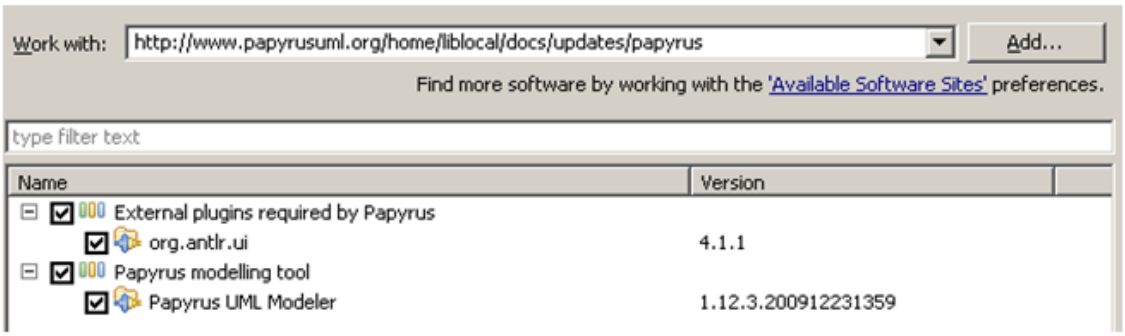

Figure 5.13: Update site URL for the assumed newest version of *Papyrus*

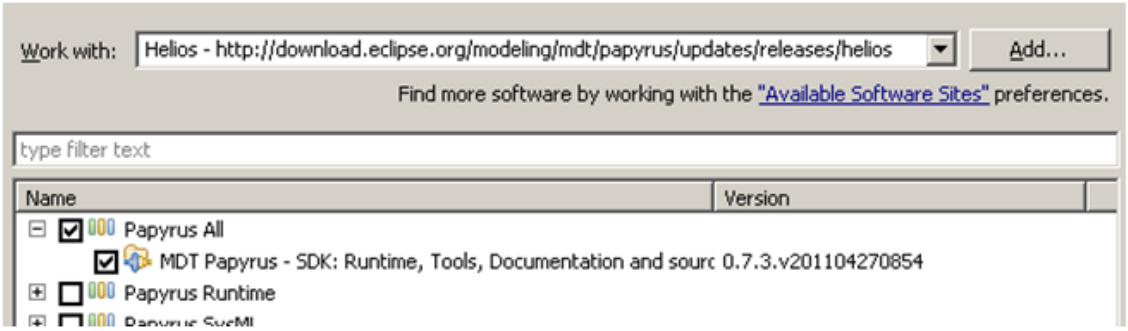

Figure 5.14: Update site URL for the assumed newest version of *Papyrus* for *Helios*

5.7.1 Summary

This editor is more or less mentioned as a design GUI. There are no team work features or file version control features at all. Therefore it is not considered as a candidate at all, as it is shown in Table 5.21.

KOC1	✔
KOC2	✘
KOC3	✘
KOC4	✔
KOC5	✘
KOC6	✔
KOC7	✘
KOC8	✘
KOC9	✔
KOC10	✔
=	✘

Table 5.21: Fitting of requirements using *Papyrus*

5.8 Eclipse Modeling Framework

The *Eclipse Modeling Framework* is an open source approach to address *Model Driven Software Development* in *Eclipse*. It consists of many single projects, which all should be connectable and each is intended to solve a specific subproblem in a final tool chain. It implements an EMOF definition for describing UML2, called *Ecore*. This file format is easily readable by other programs because of its public accessible definition and also public accessible code fragments available in Java, which are used for creating model data structures of *Ecore* files.

5.8.1 Graphical Editors

One editor for *Eclipse* has already been described: *Papyrus*. Like mentioned before, it was a standard until *Eclipse Galileo* version. Unfortunately since *Helios*, this editor is no longer supported. Also the development of a new main version of the *Eclipse SDK*, called *e4*, suggests a low chance of getting a new version of *Papyrus* for *Helios* in the next time.

Fortunately the *Helios* EMF version has a second plugin for graphical modeling of UML 2.x diagrams. It simply is called *UML2*. Figure 5.15 shows an example, designed with this editor. On the right side of the picture the created class is shown, in the middle is the corresponding model in tree view. As you can see, these two tabs also reference to two different files. The .ecore-file contains the model information. The .ecore_diagram-file gets the model information from the other file and adds graphical information to create the graphical representation. As a big disadvantage, the current version of this editor only supports class diagrams.

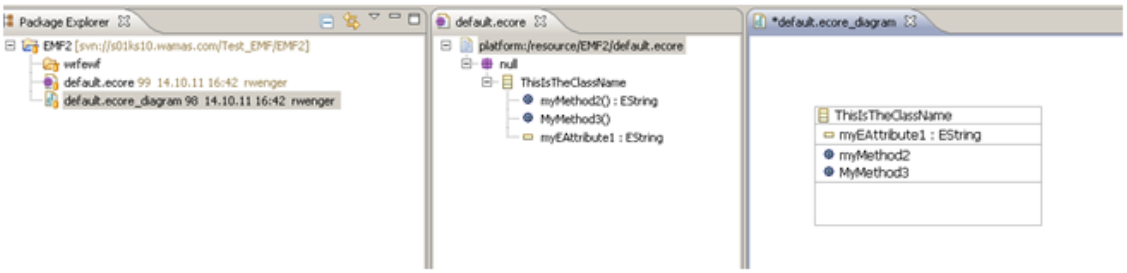

Figure 5.15: UML class designed by the editor *UML2*

Another promising *Helios* feature seems to be *Ecore tools*. Downloading this plugin leads to nearly the same editor, which only supports class diagrams. For *Helios* it is available in the incubation version 0.10.0. For *Indigo*, the following *Eclipse* version after *Helios*, it is available in version 1.0, but trying to install it in *Helios* leads to an error.

Another handy tool seems to be *Graphiti*. It is not a graphical editor for UML2; it is a tool to develop graphical editors. With the aid of *Graphiti* it would be possible to create a lightweight editor, which is completely based on *Eclipse* and exactly designed for fitting all requirements. Unfortunately for *Helios* it is only available in version 0.8.1 and is therefore, similar to many other *Eclipse* projects, still in development phase and not in an appropriate usage state. Other working graphic editors have not been found.

5.8.2 EMF Compare

EMF Compare is an *Eclipse* project that brings model comparison to EMF. It is able to compare two different model files and visualize the differences in a similar way the textual comparison feature does it.

In *Eclipse* it is possible to connect file extensions to different comparison and merging tools. Therefore it is not necessary to make a difference between model files and the source code files. The framework itself recognizes the type of file by its extension and calls the right merge plugin.

In the following example, a model file, created with the EMF standard editor, has been checked out with *Subclipse* directly into two *Eclipse* workspaces. After some modifications have been made to both diagrams, they have been synchronized with the repository. Clearly the second synchronization fails and shows up conflicts. Figure 5.16 shows this example.

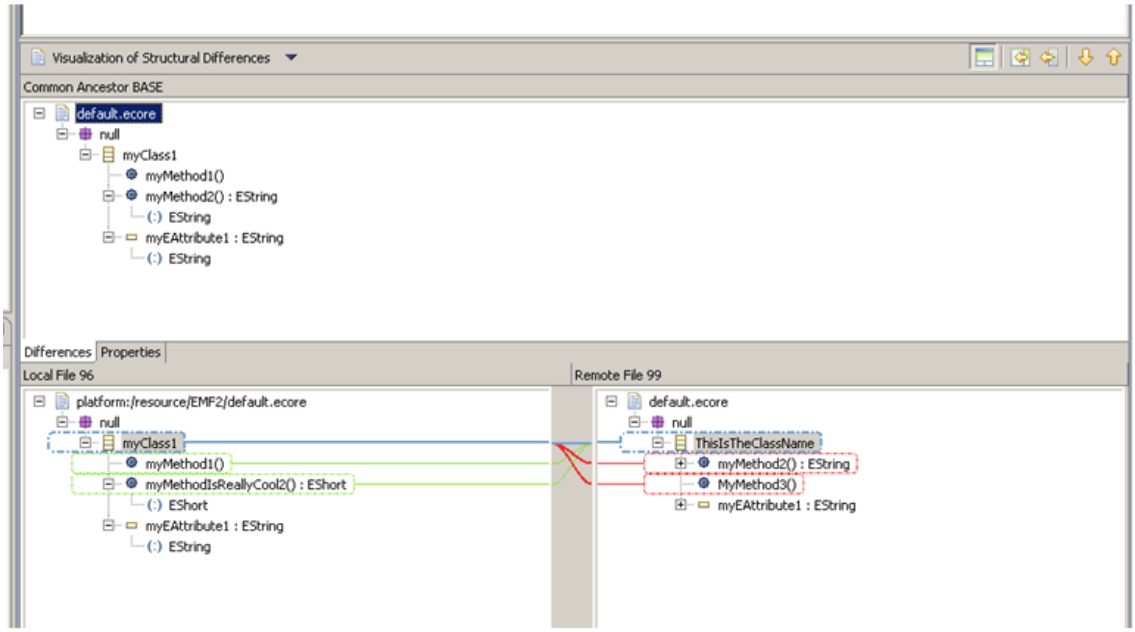

Figure 5.16: Graphical view of differences between two version of a model

With the buttons on the right upper side it is possible to copy changes from the repository to the local file and thus solve the conflicts. As a support it is possible to open a window part, which shows the content of the common ancestor.

In this example one detail has been concealed. The editor not only creates the file with the extension .ecore, which contains the model information and can be processed by EMF Compare. It also creates a file with the extension .ecore_diagram. When this file is opened with a text editor, the suggestion comes up, that this file contains graphic information for the diagram. Unfortunately this file is very close connected to the model files. Therefore it is a very big disadvantage, having this second file, because merged model files do not suit any longer to those diagram files. Figure 5.17 shows the comparison of the diagram files corresponding to the same model, which has been compared before.

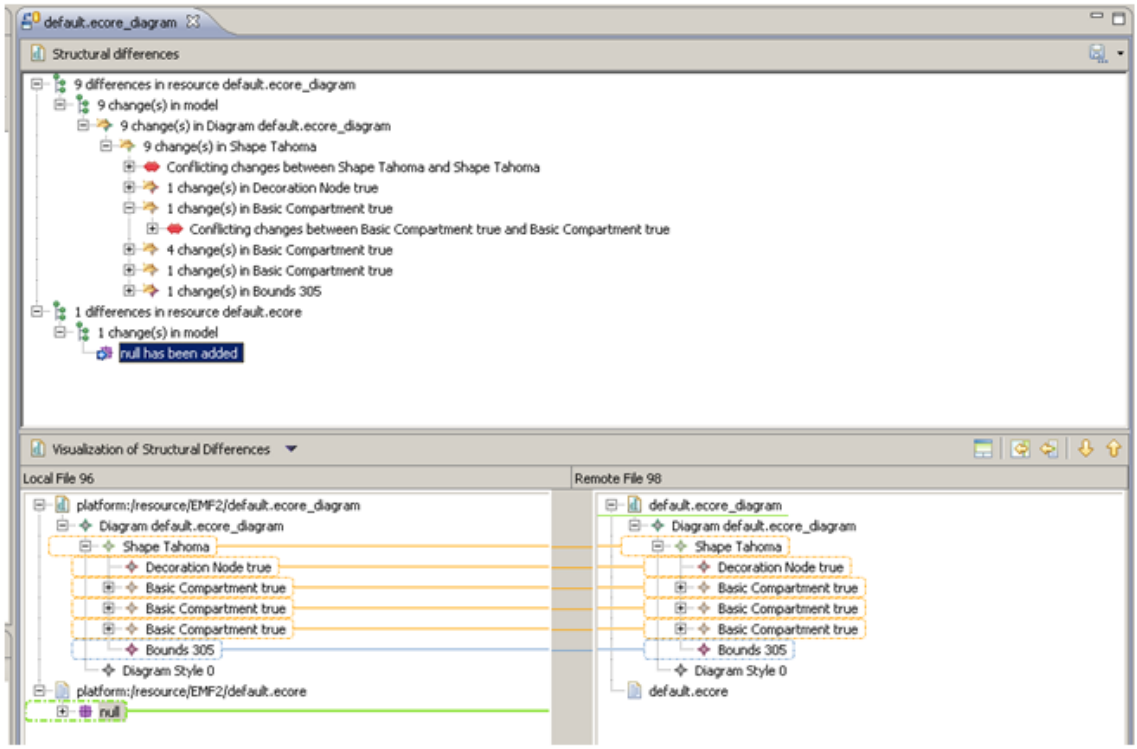

Figure 5.17: Comparison of the diagram file with EMF compare

5.8.3 CDO Model Repository

CDO (Connected Data Objects) is also a project in *Eclipse*. It has two purposes: First, it can be used as a runtime persistence framework or second, it can be used as a model repository system at development time.

CDO 4.0.0 is expected to be used with *Indigo*, but for *Helios* it is available in the version 3.0.0.

5.8.4 Dawn

Dawn is a sub-component of the *Connected Data Objects* (CDO) project and achieves to create collaborative network solutions for user interfaces based on CDO. For example, it provides collaborative access for GMF diagrams. Beside the real time shared editing Dawn provides conflict visualization and handling and other useful features for collaborative modeling. In addition to this, *Dawn* will also provide a Web-Viewer which allows viewing every diagram change online.

5.8.5 EMF Store

The *EMF Store* is a model repository that allows to store EMF model instances and keeps a version history of these instances. *EMF Store* follows the checkout/update/commit interaction paradigm known from SVN or CVS. The framework allows to checkout a copy of a model instance from the repository. Then it tracks changes on the model instances on the clients and provides an API to send the changes to the repository. Also the API allows updating the model instances according to changes of other clients via the repository.

5.8.6 Summary

To finally sum up the issue EMF, it has to be mentioned, that the *Eclipse* community really takes care of the problem of MDSD with teamwork support. Many different tools do really have the potential to be part of a solution to this problem, but two main drawbacks occur, considering a solution with *Eclipse*. First, many features are not yet in an appropriate state and second, some features only become available for newer versions of *Eclipse*, like *Indigo* (version 3.7) or *Juno* (version 4.2). Currently there is no tool chain realizable containing only *Eclipse* features for *Helios*, but it is very likely that in the next years, assuming *Salomon* has updated *Eclipse* to a higher version, a tool chain exclusively basing on this open source projects is a free and highly customizable solution for shared and distributed model driven software development.

Chapter 6

Evaluation of possible solutions

6.1 Calculations

According to Section 4.2.3, the weightings of the view lists and the attributes inside are as follows. First, the row vectors of the different view lists have to be defined, using their short names (see Equations 6.1, 6.2, 6.3 and 6.4):

$$\vec{w}_{attributes,view_{MD}} = (8, 3, 5) \tag{6.1}$$

$$\vec{w}_{attributes,view_{PD}} = (8) \tag{6.2}$$

$$\vec{w}_{attributes,view_{SA}} = (7) \tag{6.3}$$

$$\vec{w}_{attributes,view_{FM}} = (7, 8, 5) \tag{6.4}$$

After the definitions, the vectors have to be normalized to avoid parasitic weightings between the views (see Equations 6.5, 6.6, 6.7 and 6.8):

$$\vec{w}_{attributes,view_{MD},norm} = \frac{\vec{w}_{attributes,view_{MD}}}{(\vec{1}_{MD} * (\vec{w}_{attributes,view_{MD}})^T)} = (\frac{1}{2}, \frac{3}{16}, \frac{5}{16}) \tag{6.5}$$

$$\vec{w}_{attributes,view_{PD},norm} = \frac{\vec{w}_{attributes,view_{PD}}}{(\vec{1}_{PD} * (\vec{w}_{attributes,view_{PD}})^T)} = (1) \tag{6.6}$$

$$\vec{w}_{attributes,view_{SA},norm} = \frac{\vec{w}_{attributes,view_{SA}}}{(\vec{1}_{SA} * (\vec{w}_{attributes,view_{SA}})^T)} = (1) \tag{6.7}$$

$$\vec{w}_{attributes,view_{FM},norm} = \frac{\vec{w}_{attributes,view_{FM}}}{(\vec{1}_{FM} * (\vec{w}_{attributes,view_{FM}})^T)} = (\frac{7}{20}, \frac{8}{20}, \frac{1}{4}) \tag{6.8}$$

It can be seen, that the attribute weights for the PD and the SA are one. This is a direct consequence of their list lengths. The weightings are normalized to relative weightings inside a view. Therefore, if the list length is one, the scale is irrelevant, because the value always represents a 100% importance inside the view.

The next step is the definition of the weightings between the views, as in Definition 6.9.

$$\vec{w}_{views} = (10, 5, 2, 8) \tag{6.9}$$

6.1.1 MagicDraw

After evaluation of *MagicDraw* and its additional components in Section 5.1, the attribute's value vector for the different views are defined as follows (see Equations 6.10, 6.11, 6.12 and 6.13). To avoid confusion, for MagicDraw the shortname "MaDr" has been chosen as an abbreviation for the alternative x.

$$\vec{v}_{attributes,view_{MD}}(MaDr) = (9, 10, 3) \tag{6.10}$$

$$\vec{v}_{attributes,view_{PD}}(MaDr) = (8) \tag{6.11}$$

$$\vec{v}_{attributes,view_{SA}}(MaDr) = (9) \tag{6.12}$$

$$\vec{v}_{attributes,view_{FM}}(MaDr) = (4, 5, 4) \tag{6.13}$$

The further process requires the definition of the utility vector of the alternative (see Equation 6.18), which consists of the calculated utility values of every view (see Equations 6.14, 6.15, 6.16 and 6.17).

$$\begin{aligned} v_{MD}(MaDr) &= \vec{w}_{attributes,view_{MD},norm} * \vec{v}_{attributes,view_{MD}}(MaDr)^T \\ &= (\frac{1}{2}, \frac{3}{16}, \frac{5}{16}) * (9, 10, 3)^T = \frac{117}{16} \end{aligned} \tag{6.14}$$

$$\begin{aligned} v_{PD}(MaDr) &= \vec{w}_{attributes,view_{PD},norm} * \vec{v}_{attributes,view_{PD}}(MaDr)^T \\ &= (1) * (8)^T = 8 \end{aligned} \tag{6.15}$$

$$\begin{aligned} v_{SA}(MaDr) &= \vec{w}_{attributes,view_{SA},norm} * \vec{v}_{attributes,view_{SA}}(MaDr)^T \\ &= (1) * (9)^T = 9 \end{aligned} \tag{6.16}$$

$$\begin{aligned} v_{FM}(MaDr) &= \vec{w}_{attributes,view_{FM},norm} * \vec{v}_{attributes,view_{FM}}(MaDr)^T \\ &= (\frac{7}{20}, \frac{8}{20}, \frac{1}{4}) * (4, 5, 4)^T = \frac{22}{5} \end{aligned} \tag{6.17}$$

$$\begin{aligned} \vec{v}_{views}(MaDr) &= (v_{MD}(MaDr), v_{PD}(MaDr), v_{SA}(MaDr), v_{FM}(MaDr)) \\ &= (\frac{117}{16}, 8, 9, \frac{22}{5}) \end{aligned} \tag{6.18}$$

The resulting utility value of this alternative $x = "MaDr"$ is calculated as follows in Equation 6.19.

$$\begin{aligned} v(MaDr) &= \vec{w}_{views} * \vec{v}_{views}(MaDr)^T \\ &= (10, 5, 2, 8) * (\frac{117}{16}, 8, 9, \frac{22}{5})^T = \frac{6653}{40} \approx 166,3 \end{aligned} \tag{6.19}$$

6.1.2 Visual Paradigm

After evaluation of *Visual Paradigm* and its additional components in Section 5.2, the attribute's value vector for the different views are defined as follows (see Equations 6.20, 6.21, 6.22 and 6.23). To avoid confusion, for *Visual Paradigm* the shortname "ViPa" has been chosen as an abbreviation for the alternative x.

$$\vec{v}_{attributes,view_{MD}}(ViPa) = (9, 10, 8) \tag{6.20}$$

$$\vec{v}_{attributes,view_{PD}}(ViPa) = (8) \tag{6.21}$$

$$\vec{v}_{attributes,view_{SA}}(ViPa) = (9) \tag{6.22}$$

$$\vec{v}_{attributes,view_{FM}}(ViPa) = (8, 5, 9) \tag{6.23}$$

The further process requires the definition of the utility vector of the alternative (see Equation 6.28), which consists of the calculated utility values of every view (see Equations 6.24, 6.25, 6.26 and 6.27).

$$\begin{aligned} v_{MD}(ViPa) &= \vec{w}_{attributes,view_{MD},norm} * \vec{v}_{attributes,view_{MD}}(ViPa)^T \\ &= (\frac{1}{2}, \frac{3}{16}, \frac{5}{16}) * (9, 10, 8)^T = \frac{71}{8} \end{aligned} \tag{6.24}$$

$$\begin{aligned} v_{PD}(ViPa) &= \vec{w}_{attributes,view_{PD},norm} * \vec{v}_{attributes,view_{PD}}(ViPa)^T \\ &= (1) * (8)^T = 8 \end{aligned} \tag{6.25}$$

$$\begin{aligned} v_{SA}(ViPa) &= \vec{w}_{attributes,view_{SA},norm} * \vec{v}_{attributes,view_{SA}}(ViPa)^T \\ &= (1) * (9)^T = 9 \end{aligned} \tag{6.26}$$

$$\begin{aligned} v_{FM}(ViPa) &= \vec{w}_{attributes,view_{FM},norm} * \vec{v}_{attributes,view_{FM}}(ViPa)^T \\ &= (\frac{7}{20}, \frac{8}{20}, \frac{1}{4}) * (8, 5, 9)^T = \frac{141}{20} \end{aligned} \tag{6.27}$$

$$\begin{aligned} \vec{v}_{views}(ViPa) &= (v_{MD}(ViPa), v_{PD}(ViPa), v_{SA}(ViPa), v_{FM}(ViPa)) \\ &= (\frac{71}{8}, 8, 9, \frac{141}{20}) \end{aligned} \tag{6.28}$$

The resulting utility value of this alternative $x = "ViPa"$ is calculated as follows in Equation 6.29.

$$\begin{aligned} v(ViPa) &= \vec{w}_{views} * \vec{v}_{views}(ViPa)^T \\ &= (10, 5, 2, 8) * (\frac{71}{8}, 8, 9, \frac{141}{20})^T = \frac{4063}{20} \approx 203,2 \end{aligned} \tag{6.29}$$

6.1.3 IBM Rational Software

After evaluation of *IBM Rational Software* and its additional components in Section 5.4, the attribute's value vector for the different views are defined as follows (see Equations 6.30, 6.31, 6.32 and 6.33). To avoid confusion, for *IBM Rational Software* the shortname "RaSo" has been chosen as an abbreviation for the alternative x.

$$\vec{v}_{attributes,view_{MD}}(RaSo) = (9, 8, 6) \tag{6.30}$$

$$\vec{v}_{attributes,view_{PD}}(RaSo) = (6) \tag{6.31}$$

$$\vec{v}_{attributes,view_{SA}}(RaSo) = (5) \tag{6.32}$$

$$\vec{v}_{attributes,view_{FM}}(RaSo) = (1, 5, 9) \tag{6.33}$$

The further process requires the definition of the utility vector of the alternative (see Equation 6.38), which consists of the calculated utility values of every view (see Equations 6.34, 6.35, 6.36 and 6.37).

$$\begin{aligned} v_{MD}(RaSo) &= \vec{w}_{attributes,view_{MD},norm} * \vec{v}_{attributes,view_{MD}}(RaSo)^T \\ &= (\frac{1}{2}, \frac{3}{16}, \frac{5}{16}) * (9, 8, 6)^T = \frac{63}{8} \end{aligned} \tag{6.34}$$

$$\begin{aligned} v_{PD}(RaSo) &= \vec{w}_{attributes,view_{PD},norm} * \vec{v}_{attributes,view_{PD}}(RaSo)^T \\ &= (1) * (6)^T = 6 \end{aligned} \tag{6.35}$$

$$\begin{aligned} v_{SA}(RaSo) &= \vec{w}_{attributes,view_{SA},norm} * \vec{v}_{attributes,view_{SA}}(RaSo)^T \\ &= (1) * (5)^T = 5 \end{aligned} \tag{6.36}$$

$$\begin{aligned} v_{FM}(RaSo) &= \vec{w}_{attributes,view_{FM},norm} * \vec{v}_{attributes,view_{FM}}(RaSo)^T \\ &= (\frac{7}{20}, \frac{8}{20}, \frac{1}{4}) * (1, 5, 9)^T = \frac{23}{5} \end{aligned} \tag{6.37}$$

$$\begin{aligned} \vec{v}_{views}(RaSo) &= (v_{MD}(RaSo), v_{PD}(RaSo), v_{SA}(RaSo), v_{FM}(RaSo)) \\ &= (\frac{63}{8}, 6, 5, \frac{23}{5}) \end{aligned} \tag{6.38}$$

The resulting utility value of this alternative $x =$ "*RaSo*" is calculated as follows in Equation 6.39.

$$\begin{aligned} v(RaSo) &= \vec{w}_{views} * \vec{v}_{views}(RaSo)^T \\ &= (10, 5, 2, 8) * (\frac{63}{8}, 6, 5, \frac{23}{5})^T = \frac{3111}{20} \approx 155,6 \end{aligned} \tag{6.39}$$

6.2 Result

After calculation of all results, every solution candidate has its own utility value. A comparison of these values mentions the best solution for the problem, in our case *Visual Paradigm*. Figure 6.1 shows a graphical result of the calculated solution fittings for the defined requirements.

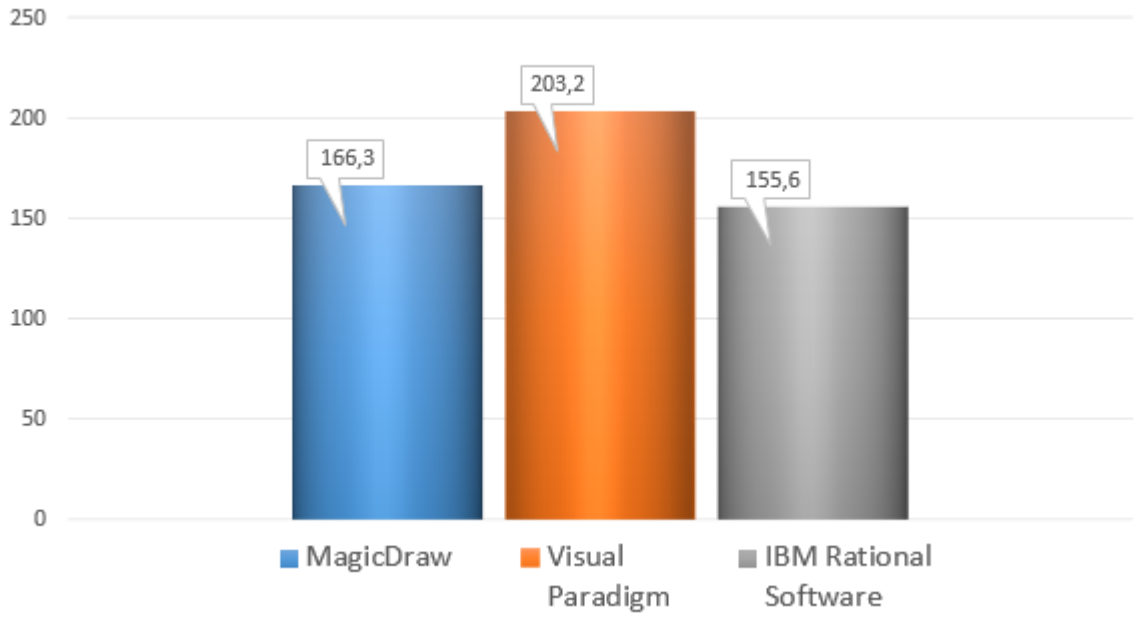

Figure 6.1: Result after comparison and calculation

At this point it has to be mentioned that the calculation is based on weightings and evaluation values, which are chosen from specific people, and maybe other persons would have chosen different values. But here the great benefit of this method becomes apparent: The whole process can easily be automated with simple tools like spreadsheets or calculation tools and therefore may be repeated at any time. Also the different views can be evaluated not only by one person but by groups, where the actual used values are calculated with statistical methods to rule out spikes.

Having a look at this thesis' result, two properties may be discussed: First, the number of properties have been on a lower side, therefore small changes at the weighting and utility vectors would result in greater change, than it would with a higher number of attributes. For this reason, systems should be analyzed for as many usefully properties as can be found.

Mentioning the second point, the result value of the "winner" has a significant difference to both of the other candidates. Expressing it in a mathematical way, the maximum utility value of a possible solution has to be calculated. This calculation uses attribute values, with the highest possible value 10. The actual weighting vectors still remain to their values, as they are independent to any alternative. Nevertheless, the sum of the weighting vector inside a view has been normalized to one, therefore any evaluation with equal values of every property, this value has to be the utility value. As a result the utility vector consists of the maximum scale values of every view evaluation. This is a reasonable result, as the highest rating must lead to the highest possible value, independent of the weighting inside. The calculation has been simplified to the following Equation 6.40:

$$v(MAX) = MaximumRatingValue * \sum \vec{w}_{views} = 10 * \sum (10, 5, 2, 8) = 10 * 25 = 250 \tag{6.40}$$

Compared to this maximum possible value, which is shown in Figure 6.2, the program with the best fitting is nearly 15% better than the second best solution. This might be an indicator, that the selection of the candidate still remains, even slightly different evaluations will be chosen.

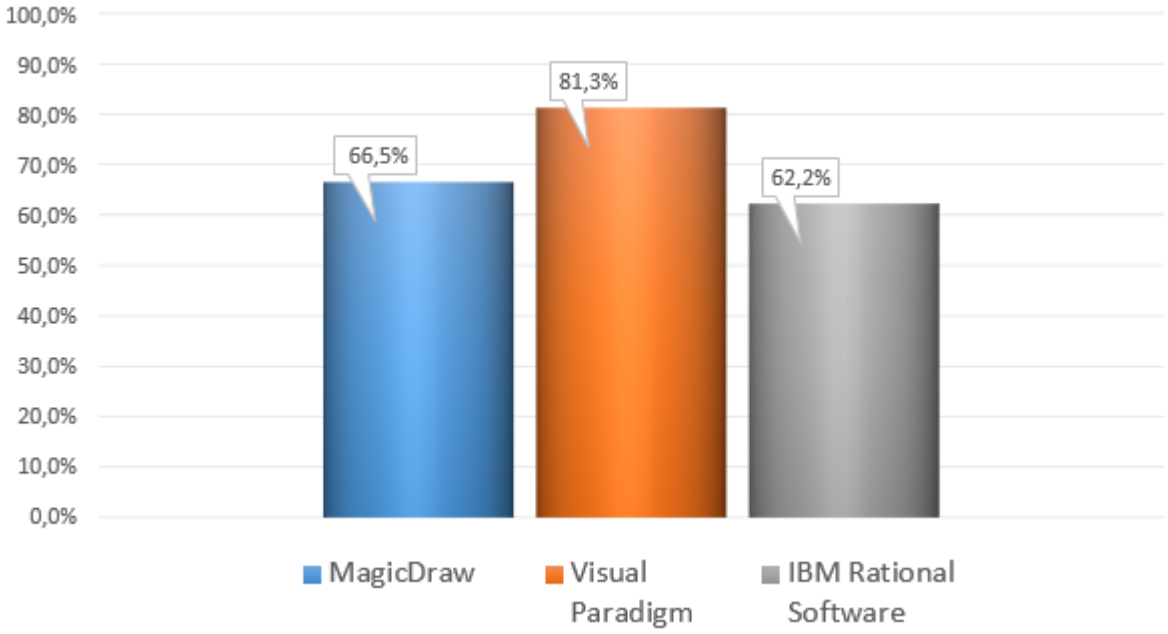

Figure 6.2: Result after comparison and calculation

Chapter 7

Conclusion and future work

This master thesis provides basic information for improving the actual situation in the company of *Salomon Automation*. It should be seen as a documentation of the current situation and also as a tool for a comparing and decision making process. It does not include the best or final solution, but it contains methods to find it. It also provides an example for using the mentioned methods by analyzing the current market of *MDSD*-tools and supporting technologies.

7.1 Future and open work

7.1.1 Implementation

This thesis was intended to be a theoretical work to build a basis for a practical implementation. Unfortunately, this practical work is therefor an open issue. This refers to the following points:

- **Updating information:** Since this master thesis has been developed over a longer period, some information might not be up to date. Especially version numbers, prizes and additional packages may have been updated.
- **Implementation of new tools:** After a decision for new tools, they have to be implemented by the companies IT department. Not only installing of the clients has to be done, also the servers has to be updated.
- **Updating models:** One of the major criteria was the ability to update the existing models to the new format. Therefor, this issue has to be handled either, independent from which solution has been chosen.
- **Adaption of existing generators:** The new model creating end editing tools will have a new file format and maybe also a new language. Therefor the input software modules of the generators have to be adapted.

7.1.2 Eclipse 4

The growing importance and practical use of model driven software development drives the Eclipse community more and more to build several solutions for parts of a tool chain. This fact, and also the fact, that Eclipse is free of charge and is allowed, to be reused in commercial Software (refer to the *Eclipse Public License* [Fou04]), makes it very interesting for a solution.

7.1.3 Domain Specific Language

Based on the idea of building an own system (maybe using Eclipse), it is also possible to develop a new language, which is more domain specific than common description languages like UML. On the one hand, it could be an easier and more efficient language, which can save time and money. A development of tools can use the found requirements in this thesis as an input for a very suitable tool chain. On the other hand, it will consume an initial effort for developing the language and creation of suitable tools. Therefor, I would recommend a new study, which can be also accomplished in the context of a bachelor or master thesis.

Appendix A

Abbreviations

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
CDC	Connected Device Configuration
CDO	Connected Data Objects
CVS	Concurrent Versions System
DB	DataBase
DBMS	DataBase Management System
DDL	Data Definition Language
DSL	Domain Specific Language
EA	Enterprise Architect
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
EUR	EURo
FAQ	Frequently Asked Questions
FM	Finance Manager
FO	Formatting Objects
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JDT	Java Development Tools
JPG	Joint Photographic experts Group
KOC	Knock-Out Criteria
LDAP	Lightweight Directory Access Protocol
MaDr	Magic Draw
MAUT	Multi Attribute Utility Theory
MD	Model Developer
MDA	Model Driven Architecture
MDD	Model Driven Development
MDG	Model Driven Generation
MDSD	Model Driven Software Development
MDT	Model Development Tools

MOF	MetaObject Facility
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
PD	Product Developer
PNG	Portable Network Graphics
RaSo	Rational Software
RSA	Rational Software Architect
RTC	Rational Team Concert
RTF	Rich Text Format
SA	System Administrator
SDK	Software Development Kit
SDO	Service Data Objects
SQL	Structured Query Language
SVG	Scalable Vector Graphics
SVN	Apache SubVersioN
TCP	Transmission Control Protocol
TWS	TeamWork Server
UML	Unified Modeling Language
US	United States
USD	United States Dollar
ViPa	Visual Paradigm
VL	View Lists
VP	Visual Paradigm
WAMAS	WArehouse MAnagement System
WMF	Windows MetaFile
WYSIWYG	What You See Is What You Get
XMI	XML Metadata Interchange
XML	Extensible Markup Language
ZIP	ZIPper (data format)

Bibliography

[AUT11] AUTOSAR. Specification of ecu state manager. Technical report, AUTOSAR, 2011. Webaddress: http://autosar.org/download/R4.0/AUTOSAR_SWS_ECUStateManager.pdf, visited on September 5th 2012. IV, 4

[BCE+06] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A manifesto for model merging. Technical report, Department of Computer Science, University of Toronto, 2006. Webaddress: http://www.cs.toronto.edu/~gbrunet/pubs/gamma06.pdf, visited on March 15th 2013. 6

[Fou04] Eclipse Foundation. Eclipse public license. Technical report, Open Source Initiative, February 2004. Webaddress: http://www.eclipse.org/legal/epl-v10.html, visited on March 16th 2013. 78

[HP11] Valentin Haenel and Julius Plenz. *Git - Verteilte Dokumentationsverwaltung für Code und Dokumente*. Open Source Press, München, 2011. 5

[Kru95] Philippe Kruchten. The "4+1" view model of software architecture. *IEEE Software*, 12:42–55, November 1995. IV, 8

[Lad03] Ramnivas Laddad. *Aspectj in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003. IV, 3, 4

[Maga] No Magic. Magicdraw feature list. Technical report. Webaddress: http://www.nomagic.com/files/MagicDraw_Features.pdf, visited on August 18th 2012. 39

[Magb] No Magic. Teamwork server feature list. Technical report. Webaddress: http://www.nomagic.com/files/brochures/a4/MagicDrawTeamworkServer.pdf, visited on August 18th 2012. 42

[Par] Visual Paradigm. Visual paradigm edition list. Technical report. Webaddress: http://http://www.visual-paradigm.com/aboutus/, visited on May 14th 2013. 46

[RR08] Suzanne Robertson and James Robertson. Volere requirements techniques: an overview. Technical report, The Atlantic Systems Guild, 2008. IV, 9

[Sch01] Ralph Schäfer. Rules for using multi-attribute utility theory for estimating a users interests. Technical report, DFKI GmbH, 2001. 9

[StOSSG00] Richard Soley and the OMG Staff Strategy Group. Model driven architecture. Technical report, Object Management Group, 2000. Webaddress: http://www.omg.org/cgi-bin/doc?omg/00-11-05.pdf, visited on September 5th 2012. 4

[SV06] Thomas Stahl and Markus Völter. *Model Driven Software Development - Technology, Engineering, Management.* dpunkt.verlag GmbH, Heidelberg, 2006. 3

[Sys] Sparx Systems. Enterprise architect feature list. Technical report. Webaddress: http://www.sparxsystems.com/products/ea/features.html, visited on January 25th 2013. 59

[vW86] Detlof von Winterfeld. *Decision Analysis and Behavioral Research.* Cambridge University Press, 1986. 9