UNIVERSITY OF OSLO
Department of informatics

Traceability in Model Driven Engineering

Master thesis
60 credits

Svein Johan Melby

01. November 2007

Acknowledgements

This thesis is submitted to the Department of Informatics at the University of Oslo as part of a Master degree. The work reported in this thesis has been carried out at SINTEF Information and Communication Technology, Department of Cooperative and Trusted Systems.

I would like to thank my supervisors Gøran K. Olsen, and Arne Jørgen Berre for guidance and support through the work with this master thesis. I would also like to thank:

- Thor Neple whose contribution through the work with this thesis has been much appreciated.
- My former supervisor Jan Øyvind Aagedal for his contributions at the earlier stages of the work with this thesis.
- Bjørn Nordmoen at Western Geco for the valuable experience gained during my summer job this summer.
- My friend Espen Hauge for taking time to read through my thesis at short notice, providing much needed feedback.

Oslo, November 2007

Svein Johan Melby

Table of Contents

Figures

Tables

1 Introduction

1.1 Motivation and Background

Due to advances in programming languages and technologies over the past decades developers are now capable of creating increasingly more advanced and complex computer systems. These systems often involve several different platforms and technologies, each with its own set of standards and domain concepts. As a result of this, developers often spend a considerable amount of time on details related to these different platforms and technologies [1]. This makes it harder to focus on the design intent and the business needs.

As a mean to help dealing with this complexity, a variety of different visual modelling languages has seen the light over the past couple of decades. These modelling languages are used as an abstraction mechanism, as they allow details to be hidden or shown in different ways according to what purpose a specific model or diagram is meant to serve. This allows focus to be kept on the right things at different stages of the development process. Not only does this help developers focus on the domain of the business, but also makes it easier to communicate with people without a technological background.

Modern computer systems do however often need rapid upgrading due to changes in the underlying technology or business needs. This means that a lot of effort needs to be made to keep the models documenting the system in synch with the implementation of the system. Moreover, documentation of the systems is often performed as a separate task, and is not linked to the artefacts they are supposed to document by any means. The result of this is often that the documentation eventually gets out of synch with the implementation, and hence does not document the system sufficiently.

The advent of Model Driven Engineering (MDE) promises a solution too many of these issues [1] by using models as the primary development artefact – instead of using models to document the implementation code, implementation code is generated from the models. This is achieved by using models based on metamodels that formally describes domain concepts and their relationships to each other, and then use transformations to transform these models to other models and/or implementation code. A result of this is that the development process may be automated and is formally (to some degree) described as a set of models implicitly related to each other through transformations. This allows developers to focus more on a particular domain rather than details related to a particular technology – the system can be described in a platform independent way and transformed into models with platform specific details. Moreover; due to the automatic nature of the process, MDE should make it easier to keep the different artefacts involved in a system development process in synch.

Although MDE promises to ease complexity at one level, it brings forth added complexity on another level; Due to the extensive use of intermediate models and automatic transformations it might become difficult to see how the different artefacts relates to each other [2]. I.e. the logic of a transformation may not always be quite obvious, and it may not always be straightforward to deduce which artefacts that were generated from a specific model and what transformation that was used to create it. Traceability therefore becomes a crucial part of any

MDE framework as a means to record the relationships between different artefacts involved in the development process [3].

1.2 Problem Identification

There are several problems associated with traceability in general and traceability in MDE in particular. E.g. how and where should the traceability information be stored? What information should be stored? How should the information be classified to support different kinds of use and analysis purposes? It is reasonable to assume that semantically rich traceability information could prove valuable for many purposes in an MDE process [3-10]. This does however come at a cost – semantically rich traceability information means that more information needs to be maintained and it might also require more effort to be maintained. Thus, richer semantics should only be added when it serves a purpose [11].

The aim of this thesis will be to address some of these issues by suggesting a classification scheme for traceability information in MDE which promote semantically rich traceability information with as little impact on the development process as possible. The high-level problem we aim to solve is hence:

What information should be maintained, and how should this information be classified, in order to support a rich set of analysis purposes with respect to traceability in MDE?

A critical success factor will be how transparently this information can be maintained.

Some issues related to this problem are:

- For what purposes may traceability information play a role in an MDE process?
- What kinds of analysis may be performed on the traceability information?
- What are the requirements for a classification scheme in MDE?
- How can the usefulness of a classification scheme be validated?
- For what purposes are the simple notion of "a relationship exists between artefact A and B" not sufficient?
- How can these insufficiencies be improved?
- How much of the traceability information can be generated automatically?
- How much of the traceability information must be tool specific?
- Are the improvements supported by current technologies?

A traceability classification scheme does however not have a value by it self; it requires tool support to come to real use. In fact, how such a classification scheme may be defined, and what information that may be stored is highly tool dependent. Thus, in order to define and utilised such a traceability classification scheme, tool support must be provided. This brings forth a few other issues:

- What are the requirements for a tool supporting traceability in MDE?
- How can a traceability tool be validated?

1.3 Hypotheses

The high-level hypotheses are that;

H1 Semantically rich traceability information will make it possible to conduct more precise analysis on the traceability information, and improve automation of the process.

H2 An EMF based tool with support for generic definition of traceability types and functionality for creating traceability information of the defined types will be suitable for defining and using semantically rich traceability classification schemes.

These hypotheses specify our assumptions regarding the solution to the problem, and will serve as guidance to validate the results. In Table 1 the hypotheses are divided into several sub-hypotheses, which will be used as a basis to validate the results of this thesis. The test-cases associated with the hypotheses are presented in chapter 6.

	Hypotheses
H1	Semantically rich traceability information will make it possible to conduct more precise analysis on the traceability information, and improve automation of the process.
H1.1	A suitable classification scheme will make it possible to conduct precise coverage analysis.
H1.1.1	A suitable classification scheme will make it possible to find all relevant parts of a model that is not utilised by a transformation.
H1.1.2	A suitable classification scheme will make it possible to validate to what degree an artefact is covered by other artefacts in the development process.
H1.2	A suitable classification scheme will make it possible to find all artefacts that may be impacted by a change, and ease the process of finding out how they are impacted.
H1.3	A suitable classification scheme will make it possible to conduct orphan analysis.
H1.3.1	A suitable classification scheme will make it possible to find artefacts that have been generated from elements that have been deleted.
H1.3.2	A suitable classification scheme will make it possible to find artefacts that are orphaned with respect to other artefacts at previous steps of the development process.
H1.4	A suitable classification scheme will make it possible to visualise traceability information in a meaningful way.
H1.5	A suitable classification scheme will maintain sufficient information to enable reverse engineering.
H2	An EMF [12] based tool with support for generic definition of traceability types and functionality for creating traceability information of the defined types will be suitable for defining and using semantically rich traceability classification schemes.
H2.1	A suitable EMF based traceability tool will be easy to integrate with external plug-ins.

Table 1: Hypotheses

1.4 Research Goals

Summarizing the previous sections we end up with two high-level research goals:

1. **Tool support** – provide a tool that is capable of defining and handling semantically rich traceability information in MDE.

2. **Classification scheme** – find a suitable classification scheme for traceability in MDE, capable of capturing semantically rich traceability information, and emphasising automation.

Reaching these goals may not solve all problems related to traceability in MDE, but should give us a starting point on which further research can be performed. Providing a working traceability tool will give us a means to gain further experience with traceability by conducting experiments in practice.

1.5 Scope

Traceability in MDE is a wide research field. Most branches of industry and organisations probably have their own specific traceability needs, in terms of what they want to trace and how [5, 9]. Covering all these needs simply is not possible within the work with this master thesis. In fact, one classification scheme covering every aspect may not even be desirable, as it probably will become to complex. The scope of this thesis is therefore to suggest a classification scheme that supports the most general cases of traceability in MDE, together with a working prototype of a traceability tool that supports traceability information to be captured using several different classification schemes. With such tool support, a classification scheme could be extended or combined with additional classification schemes supporting domain specific needs. This should provide a flexible way of dealing with traceability, and a good basis to give further experience in the area.

1.6 Organization of Master thesis

This master thesis is organised in the following chapters:

Chapter 1 gives a short introduction to the motivation and background of this master thesis, and describes the problem, hypotheses and research goals of this master thesis.

Chapter 2 presents the method that will be used in the later stages of the work with this master thesis. The evaluation criteria for the master thesis are also given here.

Chapter 3 gives an overview of the relevant theory of traceability and Model Driven Engineering (MDE), and discusses existing traceability solutions and tools.

Chapter 4 presents requirements that must be satisfied by the traceability tool, and presents the proposed traceability tool.

Chapter 5 presents the requirements for the classification scheme, and presents the proposed traceability tool.

Chapter 6 gives and introduction to the test-cases that will be used to validate the hypotheses presented in Table 1. For each hypothesis, a test case with an associated prediction regarding the result of the test is presented.

Chapter 7 discusses the results of the test-cases presented in chapter 7, and discusses whether the predictions were strengthened or falsified.

Chapter 8 discusses the fulfilment of the requirements are evaluated and the results of the work with this thesis are discussed.

Chapter 9 summarises the work with this thesis, and discusses the claimed contribution and weaknesses with the proposed solutions.

Chapter 10 gives an overview of related research.

Chapter 11 discusses future work.

2 Research Method

2.1 Introduction

The main goal of technology research is innovation – to create new artefacts, or to improve existing artefacts in order to support some identified needs [13]. In this section we will present the method that will be used to achieve this in the context of this thesis.

2.2 Method

According to Solheim and Stølen [13] technology research is an iterative process consisting of the following main steps:

- **Problem analysis** – find a problem to which a solution is needed.
- **Innovation** – construct an artefact, with the assumption that it solves the problem.
- **Validation of results** – validate that the artefact actually solves the problem. The validation process is based on *predictions* regarding the new artefact. If the predictions turn out to be correct, it can be argumented that the artefact solves the identified problem.

The results are validated by performing test-cases, which will either strengthen or weaken the hypotheses. This process may be repeated several times, depending on the result of the validation.

In addition the result must be validated by asking the following three questions, identifying whether it represents something of scientific value:

1. Does the new artefact represent new knowledge?
2. Is the new knowledge of interest to others?

Is the new knowledge and results documented in a way that enables validation by others?

In Figure 1 the research method is illustrated in the context of this thesis. The three steps are elaborated further in subsections 2.2.1, 2.2.2, and 2.2.3.

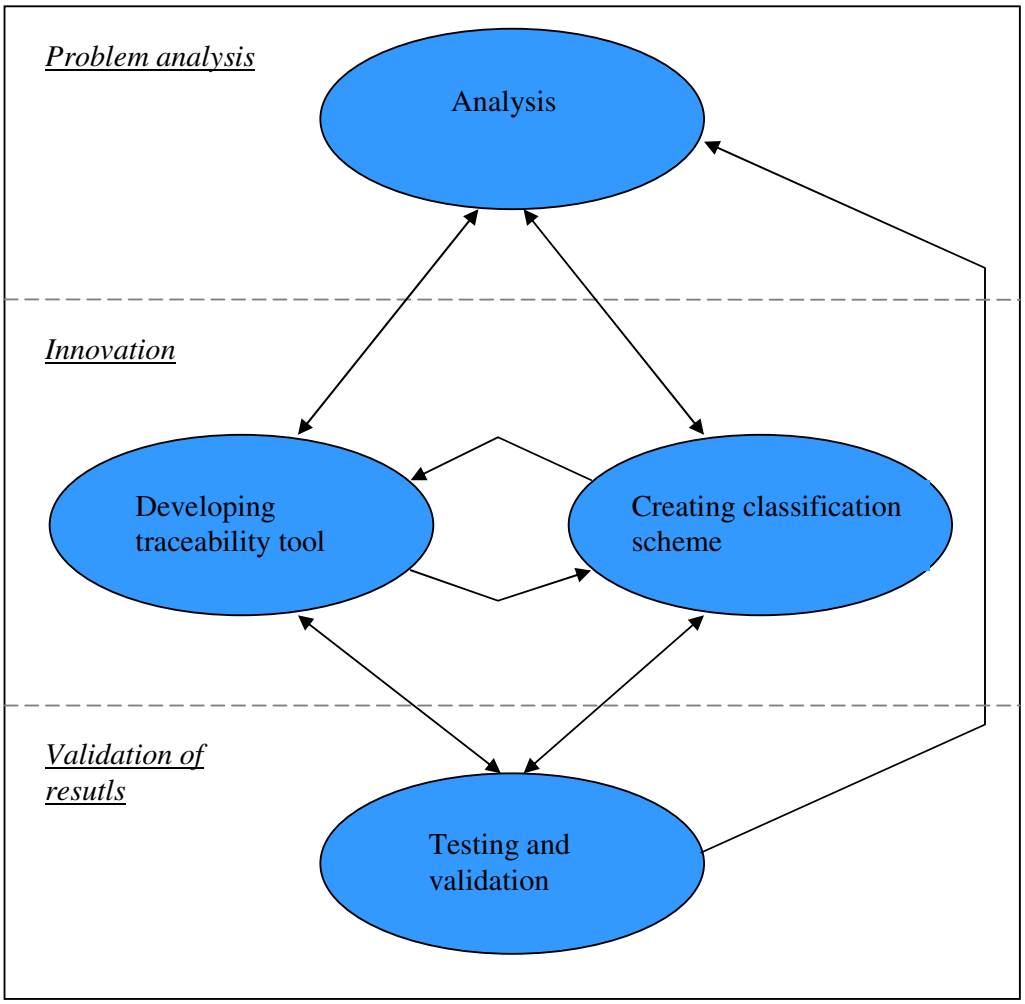


Figure 1: Illustration of the research method in context of this thesis

2.2.1 Problem Analysis

In this case, the identification of the problem was initiated by a proposal for a theme for a master thesis from SINTEF ICT. The problem analysis was continued by reading papers regarding traceability as a means to get an overview over the state of the art in the area. In chapter 3, the theoretical framework is discussed and we discuss some challenges regarding traceability in MDE.

2.2.2 Innovation

The problems that were identified in the problem analysis is analysed further in this step of the process, with the aim of identifying solutions to the identified problems. There are two artefacts that are produced through the process of this work; the traceability tool (chapter 4) and the classification scheme (chapter 0). Based on the identified problem, and the discussion in chapter 3, requirements are presented for each of them that must be satisfied by the resulting artefacts. There may however be situations during this process that requires that we take a step back, and analyse further as additional problems arise. Furthermore, the processes of developing the tool and the classification scheme are not independent of each other, as the

tool provides the language in which the classification scheme may be described. Creating the classification scheme will therefore serve as a first test of the tool. It will therefore be necessary to switch between the two processes when errors or weaknesses are found in the tool.

2.2.3 Validation of Results

Validating the results is very important in order to confirm that result actually solves the identified problem. Thus we must validate that the classification scheme actually improves traceability in MDE. The bases for the validation will be to create predictions regarding the hypotheses based on the classification scheme. Furthermore, the classification scheme will be tested on the simple library example. Thus; the prediction will try to predict how the new traceability classification scheme will improve traceability related to this example. It is just as important to try to falsify the predictions as it is to try to prove their correctness. The results will also be validated in according to the evaluation criteria in section 2.4. If one of the two artefacts proves to be insufficient, we need to take a step back to make improvements, or it might be necessary to go back to the problem analysis with the newly gained knowledge to analyse further.

2.3 Introduction to Experiment

As part of validating the traceability tool and classification scheme, we will perform a set of test cases with the aim of falsifying or strengthening our assumptions. These assumptions are expressed as a set of hypotheses with associated predictions. The hypotheses serve as the basis for the test cases described in chapter 6, and the predictions serves as a means to falsify or strengthen these assumptions.

The experiment is performed on simulated trace information for a simple MDE [1] example, starting at set of requirements for a simple library system. From these requirements a use-case model is created. The use-case model is transformed to a Platform Independent Model (PIM) [1, 14-16] which is transformed to a Platform Specific Model (PSM) [1, 14-16] of an EJB 3.0 [17] implementation. At the last stage of the example, this PSM is transformed to Java code. The traceability tool will be used to capture simulated traces throughout the whole process.

2.4 Evaluation Criteria

The results will be evaluated according to the following criteria:

- Does the classification scheme improve traceability in MDE?
- How much of the traceability information can be generated automatically?
- How much of the traceability information must be tool specific?
- Are the improvements supported by current technologies?
- Does the new artefact represent new knowledge?
- Is the new knowledge of interest to others?
- Is the new knowledge and results documented in a way that enables validation by others?

The tool will be evaluated according to the following criteria:

- Is it simple to use?
- Is it generic?

- Does the new artefact represent new knowledge?
- Is the new knowledge of interest to others?

3 Theoretical Framework

3.1 Introduction

In this chapter the theoretical framework for this thesis is discussed. We start with an introduction to Model Driven Engineering (MDE) in section 3.2 and proceed with an introduction to traceability in the context of MDE followed by a discussion on current tools and solution, and challenges with traceability in MDE in section 3.3. The chapter ends with a summary of the discussion.

3.2 Model Driven Engineering (MDE)

3.2.1 Background and Motivation

Over the past five decades, programming languages has evolved quite a bit from first and second generation languages in terms of raising the level of abstraction, allowing developers to focus on the design intent rather than the underlying computing environment. The more recent advent of more expressive object-oriented languages like C++, Java and C# has raised the level abstraction even further. Furthermore, the use of today's reusable class libraries and application framework platforms allows developers to reuse program code and domain specific middleware services. These advances helps developers create more advanced applications, as they can focus on the domain of the application, and do not have to reinvent the wheel each time.

A downside of the growing complexity of today's systems, however, is that it is hard for anyone to keep a full overview of a system as its complexity and the amount of implementation code grows. The use of complex middleware platforms, like J2EE, .NET and CORBA, containing thousands of classes and methods, in many of today's systems makes this even harder. Not only do these platforms have to be integrated and tuned with the domain application, but the complexity makes it hard for anyone to master them completely. Moreover, as these platforms, domain of the application, or the needs of a business often change rapidly, developers often spend considerable effort manually changing the application to reflect these new requirements or porting the code to different platforms. The effect of this growing complexity is that developers need to spend lots of effort on these implementation issues, rather than focusing on requirements and the domain of the application. The complexity of the systems also makes it difficult to know what parts of the system is affected by a change in the requirements or changes to the platform or language environment.

Another problem resulting from the growing complexity of computer systems and the rapid changes is that maintaining documentation requires a lot of effort, and is very time consuming. Also, since there is no direct linkage between the models and the implementations, there is a big chance that the documentation and the implementation will be out of sync – both due to changes during the initial development process and changes at later stages. This might also lead to that developers do not put in the effort needed to assure the accuracy of the documentation.

Model Driven Engineering (MDE) addresses these issues, and aims at offering means to handle the growing complexity of these systems, and allowing developers to focus on and express domain concepts. This is achieved by combining the concepts of *Domain-specific modelling languages* and *Transformations* [1].

3.2.2 Domain-Specific Modelling Languages and Metamodels

Domain-specific modelling languages use a type system that formalizes the application structure, behaviour and requirements within a particular domain [1]. These languages are described using metamodels, i.e. metamodels describe the abstract syntax of the domain-specific languages [18]. This is achieved by describing precise relationships between the concepts of a domain, thus formalizing a language which can be used to describe specific domain related concepts. The domain specific languages can then be used to describe applications using domain-specific concepts instead of concepts of a more general purpose modelling or programming language. This means that developers can focus on the domain which they are describing, rather than on a specific platform, thus raising the level of abstraction.

One example of a Domain-specific language is UML [19]. The UML metamodel describes a very general domain specific language for software development, and thus provides a common language that can be used by software developers to describe applications and business logic. A less general example could be a metamodel describing the relationships between a set of different concepts in the banking industry, providing a language to describe bank related concepts on a high level of abstraction.

Figure 2 shows a simplified metamodel for a class-model. This model describes the class-model domain, i.e. it describes the properties of the concepts comprising such a model, and the relationships between them. Thus it provides a formal language in which these concepts can be described. Additional examples on metamodels describing Domain-specific languages are shown in Figure 10 and Figure 12. These metamodels describes concepts used to describe traceability.

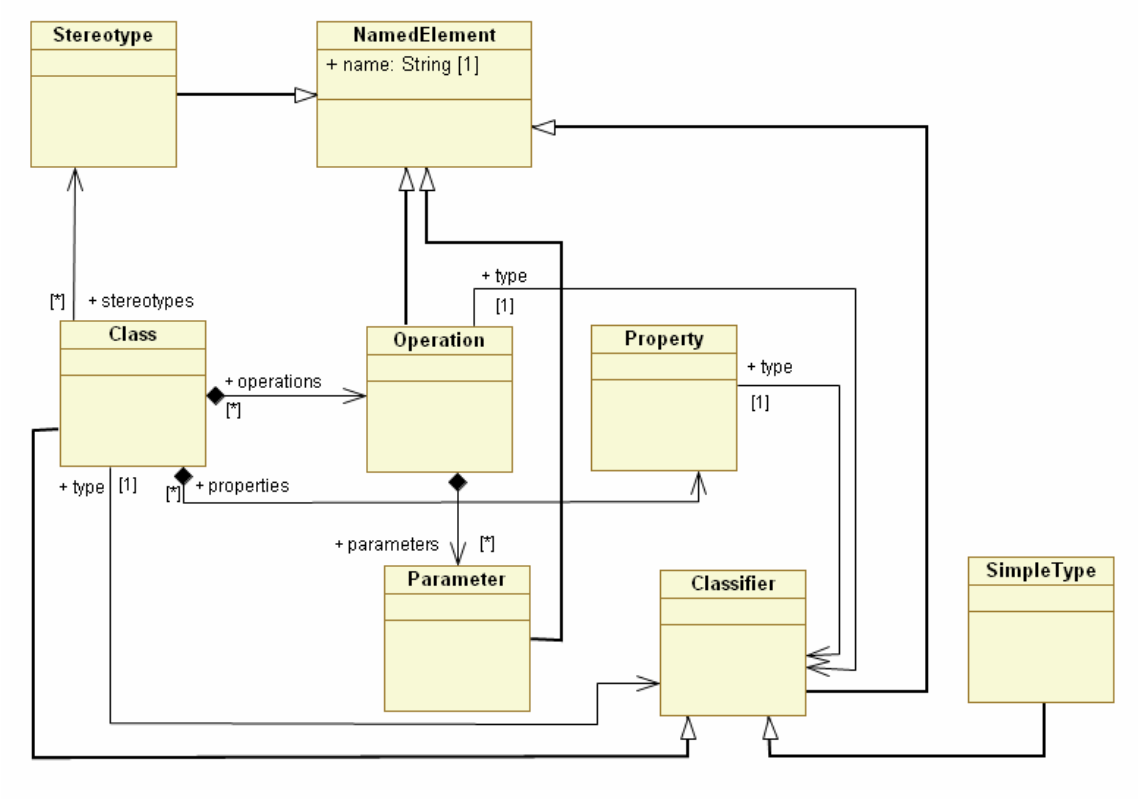


Figure 2: A simplified metamodel of a UML class-model

3.2.3 Metalevels

The Object Management Group (OMG) [20] defines four metalevels (Figure 3). These levels represent four levels in which a model may reside:

M0 - Contains the runtime instance of a model, e.g. the representation of a model element in a running application.

M1 - Contains the model, e.g. a UML model, which may be instantiated at the M0 level.

M2 - Contains the metamodel, e.g. the metamodel in Figure 2, describing the language used to define models at the M1 level.

M3 - Contains the meta-metamodel, i.e. the model describing the language used to describe a metamodel (i.e. a model residing at the M2 level). An example of such a model is the Meta Object Facility (MOF) [21], which is the language describing the UML metamodel. As models at this level are general enough to describe the concepts of languages used to describe other languages (i.e. metamodels), they will also be general enough to describe themselves. There is hence no need for an M4 level.

The four metalevels and their relationships to each other are illustrated in the figure below.

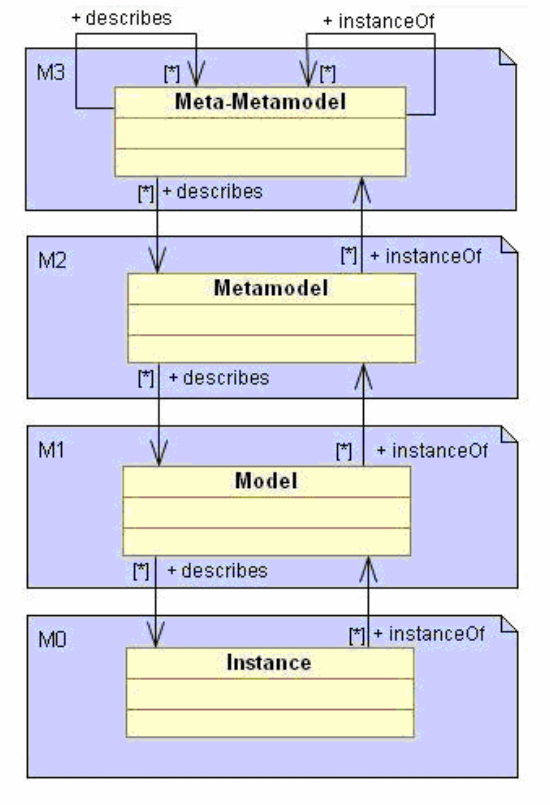


Figure 3: The four metalevels of OMG

3.2.4 PIM and PSM

Two terms that are often used in MDE are Platform Independent Models (PIMs) and Platform Specific Models (PSMs)[14-16, 22]. A PIM is a model that is independent of the implementation technology, and thus does not have any information regarding the technology used to implement the system – it describes the logic of the system. A PSM on the other hand is, as the name implies, a model describing an implementation of the system (or parts of a system) using a specific technology. A PSM is hence a refinement of the PIM.

The purpose of creating a PIM is to allow developers to capture the details of a system without having to dig into the details of a specific platform or technology. Thus it allows developers to focus on the business logic rather than how a system should be implemented and what technology to use – it is a means to raise the level of abstraction. Besides of making it easier to keep the focus on the right things during the development process PIMs also makes it easier to keep an overview and to get a better understanding of a system as all the details required to describe implementation issues are hidden away. Not only is this an advantage for developers during the development process but, it also make it easier for people without technological background or knowledge of a specific technology to understand the system.

Figure 4 shows a PIM of the library system described in the use-case models of Appendix A. In this model *Library* is shown as a UML-class containing references to the subsystems of the Library component and, the use-cases are shown as UML-operations. In addition the *Library* class contains the same operations as the operations contained in the classes it has references to. These operations are intended to be used as intermediate steps to access the operations in

CustomerSystem and *BookSystem*. This model is a little bit more detailed than the use-case model as it describes what parts the system will consist of, how they are related, and what operations they shall implement. In addition the classes of the PIM have been annotated with stereotypes, extending the UML2 metamodel with a set of types to give additional semantics to the model:

- ***Tool*** – means that the class is a component that may be used by a person to perform certain actions.
- ***Service*** – a class that provides a service or a set of related services, i.e. it provides a set of operations, which may be used by *Tool*s.
- **PersistentObject** – an object that contains some information that needs to be stored in a persistent form, e.g. a database.
- ***Id*** – an attribute that represents the identifier of a *PersistentObject*.

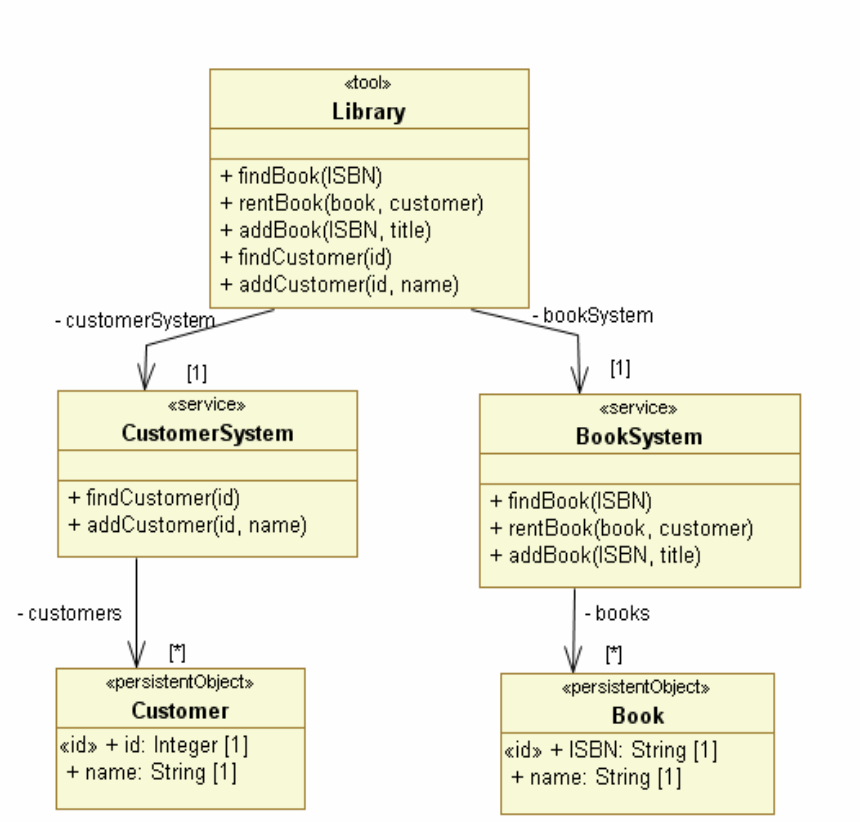


Figure 4: PIM for the library system

The PIM in Figure 4 provides all the information needed to get an understanding of the system, but it does not describe the actual implementation. The system could be implemented as a set of Java classes with a database containing books and customers, using java enterprise beans and a web interface, it could simply be implemented as 5 C++ classes storing books and customers in a file, or it could be implemented using several different technologies to meet different organizational needs. The logic of the system however would be the same.

To give more detailed information on the actual implementations of the system, developers could now proceed by creating a PSM describing the actual implementations of the system on

each of the desired target platforms. Figure 5 shows a possible PSM describing an EJB 3.0 [17] implementation of the library system.

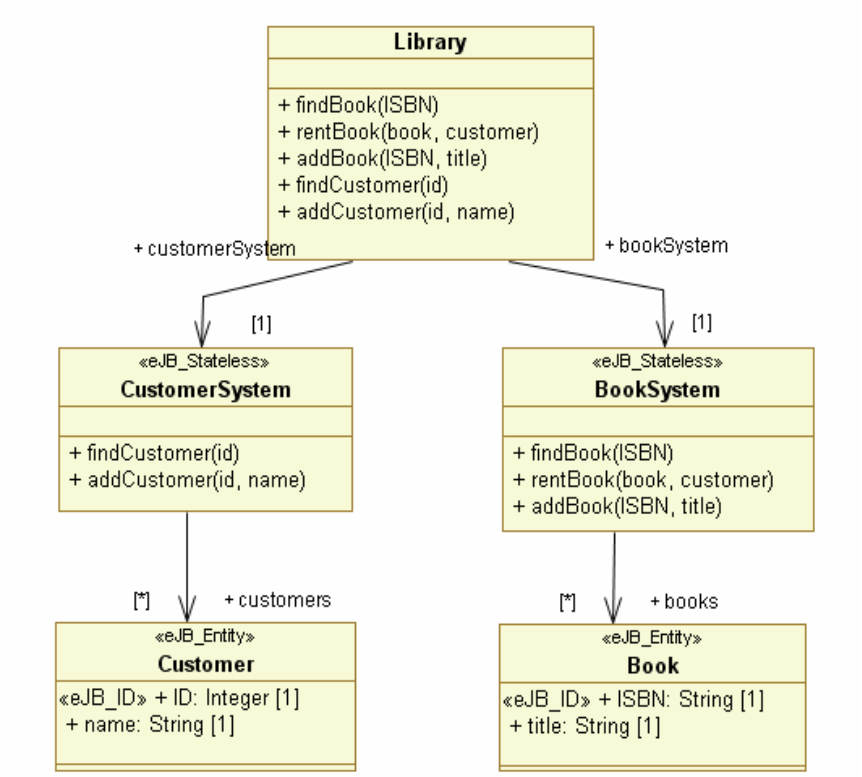


Figure 5: A PSM of the library system describing an EJB implementation of the system

This model differs from the PIM in that it is annotated with EJB specific UML-stereotypes;

- ***EJB_Stateless*** – represents a stateless session bean. This stereotype also contains an enumeration attribute ({local, remote, both}) specifying whether to use the local or remote interface, or if both may be used (in this example only the first is used).
- ***EJB_Entity*** – an entity bean.
- ***EJB_ID*** – the id-attribute of the entity bean.

By applying an UML-profile to the class-model the UML2 metamodel is extended with EJB specific types (in this case), hence the model is now platform specific to EJB.

The definitions of PIMs and PSMs are however somewhat vague. E.g. the PIM in Figure 4 could be said to be a PSM describing a java implementation, as the model is quite close to a simple java implementation. Indeed the PIM and the PSM used in this example does not even differ that much, however there is a difference in that the PIM may be used as the basis to create many different PSMs, while the PSM only describe the system with respect to a specific technology. In fact the java code itself could be said to be a PSM, as the implemented java code actually is a very detailed textual model of the PIM. Thus how to define a PIM and how to define a PSM must be decided based on what information is needed and desired to capture the information required to support the system development process. The main objective of PIMs and PSMs is to separate between *what* (PIM) needs to be implemented, and *how* it shall be implemented (PSM). The purpose of this distinction is to

make it easier to focus on the right things at the right stages of the system development process, to provide different views of the system to different people by providing information at different levels of abstraction, and to make it easier to adapt the system to new technologies, by separating business logic and technology.

Although both the PIM and the PSM in this example are UML models, it will often be the case that they are instances of different metamodels, e.g. the PSM in this example could have been described by a metamodel describing EJB concepts.

3.2.5 Transformations

Transformations has the potential to help ensure the consistency between different models, documentation, requirements and implementation code as the evolution of artefacts are formalized by rules, and can thus be automated [1]. For this to be possible the source artefacts of the transformation need to be precisely defined by e.g. a metamodel so that the artefacts might be processed by a computer. There are mainly two kinds of transformations used in MDE; ***model to model transformations*** and ***model to text transformations***.

Transformations are described by mapping rules. A mapping rule is a formal description of the relationship between the input and the output of the transformation, i.e. it describes how the output is created based on the input. The input and output artefacts might be a single artefact or it can be a collection of artefacts. Also the result of the transformation might be computed based on the properties of the artefacts or/and the relationships between the artefacts that comprise the input of the transformation. The output might be created based on a metamodel or just as artefacts without any particular syntax. The first approach is typically used to describe model to model transformations, while the latter is typically used to produce text.

Using transformations between all stages in a software engineering process therefore makes it possible to automatically generate all the artefacts needed, based on one or more source models. This does however require mapping rules for each of the transformation steps in the process and the existence of metamodels formally describing each of the models.

3.2.5.1 Model to Model Transformations

A model to model transformation is the process of creating a model based on another [23]. More precisely a model transformation creates instances, based on precise definition of the relationship between instances of one metamodel, based on instances of another. The set of rules that are used to describe the transformation is often called a mapping.

Using our running library example, a typical model to model transformation scenario would be to create the PSM based on the information in the PIM. This could be achieved by creating a mapping that takes a class model – where the classes are annotated with the stereotypes shown in Figure 4 – and generates the PSM based on the structure of the classes (attributes and operations) and the stereotypes they are annotated with:

- Classes annotated with stereotype ‘Service’ are mapped to classes with the same name, but with the stereotype ‘EJB_Service’.
- Classes annotated with stereotype ‘PersistentObject’ are mapped to classes with the same name, but with the stereotype ‘EJB_Entity’.

- All other classes are mapped to a new class with the same name, but without any stereotypes, regardless of any stereotypes contained by the input class.
- All attributes and operations contained by the input are mapped directly to attributes and operations with the same name, type, and signature as those contained by the input class. If an attribute in the input model is annotated with the stereotype 'Id', the corresponding attribute in the output model will be annotated with the stereotype 'EJB_ID'.

This transformation could be performed by applying the mapping illustrated in Figure 6 to the PIM. The illustrated mapping describes the rules for how the input model (PIM) will be turned into the output model (PSM). As both the PIM and the PSM are described by the same metamodel (UML) the mapping will create instances of the same type as the input instances, but with different stereotypes. The mapping rules in the illustration are declarative rules written in pseudo code. The rule mapStereoType() is not explained in details as it is a bit more complex than the others but the logic is explained above. In the illustration 'in' means the input artefact, an 'out' means the output artefact.

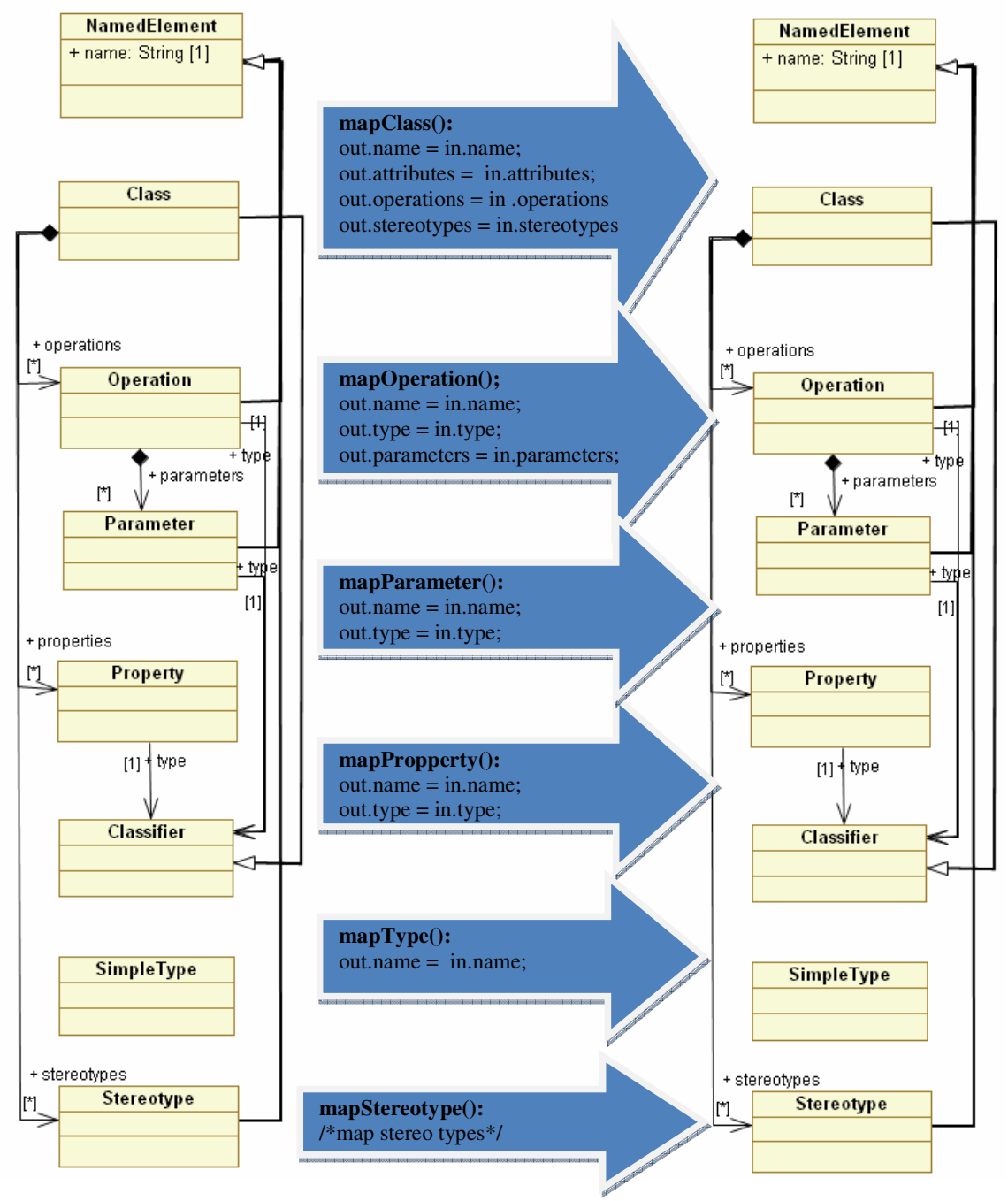


Figure 6: The PIM2EJB_PSM mapping illustrated on the Simplified UML metamodel.

Figure 7 illustrates the result of running the PIM2EJB_PSM transformation on class*CustomerSystem*.

Figure 7: Illustration of the model to model transformation of the class *Customer System*.

3.2.5.2 Model to Text Transformations

Model to text transformations refer to the act of generating text based on information from one or more models [24]. This is performed in much of the same way as model to model transformations, with the exception that only the input of the transformation (i.e. the source model) is defined by a metamodel - the output is just informal text. In MDE model to text transformations are mostly used to automatically generate implementation code from the models, but it might also be used to automatically generate documentation like java doc from the models. There are however many cases where not all the text can be generated automatically, but many parts of it, at least the skeleton of it can often be generated. The degree to which the process may be automated depends on the level of details of the model(s) used to generate the text, and how much logic one wants to encode in the transformation.

Generation of code is often performed based on the PSM, as the PSM describes the application in a platform specific way, e.g. an EJB implementation. In some cases however one can transform the code directly from the PIM, as it might be quite close to the actual implementation.

Using the running library example, one could use the PSM to generate java code annotated with EJB 3.0 annotations. The example is illustrated in Figure 8. The mapping responsible for the transformation is quite straight forward; for each class in the PSM, a java class with the same name is created, containing the same set of attributes and operations. Annotations are added according to the stereotypes in the PSM. Additionally, the attribute *em* of type *EntityManager* is added with the annotation *PersistenceContext* if the class uses a persistent object (<*Entity*> *Customer* in the case of *CustomerSystem*).

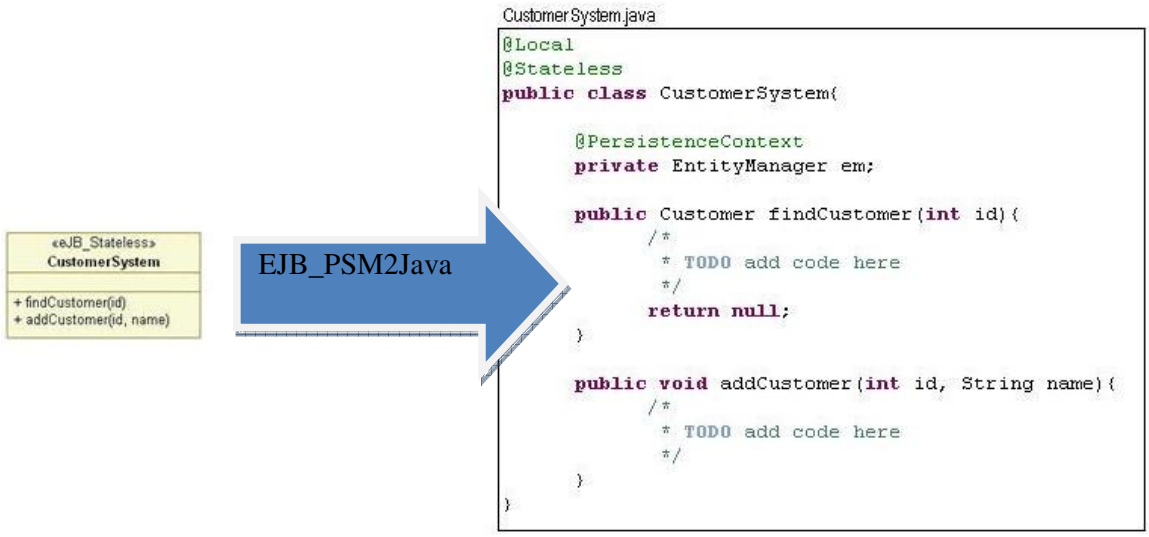


Figure 8: Illustration of the model to text transformation of the class *CustomerSystem*.

3.2.6 Model Driven Architecture (MDA)

The OMG is working on an MDE initiative called Model Driven Architecture (MDA) [14-16, 22]. MDA is based on a set of technologies that support existing and future OMG standards, including:

- **Meta Object Facility (MOF)** [21] – a standard for defining, manipulating and exchanging metamodels. The MOF resides at the M3 layer of the OMG's four meta-levels, and is hence a meta-metamodel – a modelling language used to define metamodels.
- **Unified Modelling Language (UML) [19]** – a general modelling language for describing software architecture and behaviour. The UML is described by the UML metamodel, which is described by the MOF.
- **XML Meta Interchange (XMI) [25]** – an OMG standard that maps the MOF to eXtensible Markup Language (XML). XMI allows MOF based models to be formally described using XML tags with meta-information, and allows them to be interchanged between different applications.
- **Queries/Views/Transformation (QVT) [26]** – an OMG standard for model to model transformations. QVT is a hybrid declarative/imperative language that conforms to the MOF.
- **MOF to Text Transformation language (MOF2Text) [27]** – an OMG standard for model to text transformations.

3.3 Traceability in MDE

3.3.1 Introduction

In general the word traceability is often used to reflect the degree to which all stages of process can be traced. Wiktionary.org [28] defines traceability as:

”***The ability to trace (identify and measure) all the stages that led to a particular point in a process that consists of a chain of interrelated events.*”**

In software development traceability often refers to the ability to trace the different stages in the software development process, i.e. trace the evolution of a system from start to finish. The IEEE standard glossary [29]defines traceability as follows:

“*The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.*”

The use of traceability in software development stem from the requirements community where the main goal of traceability is to provide a means to assure that a system satisfies the specified requirements.

In [30] Thomas Behrens defines two key goals of traceability in software development:

1. Ensure ***quality*** of the product – making sure that the product supports all the capabilities asked for by a stakeholder, and that the product does not have capabilities that were not asked for by any stakeholder (*Validation*). Furthermore traceability should be used to make sure that all the capabilities work properly, i.e. that they all have associated tests (*Verification*).
2. *Support* ***impact analysis*** – identifying artefacts that are affected by changes.

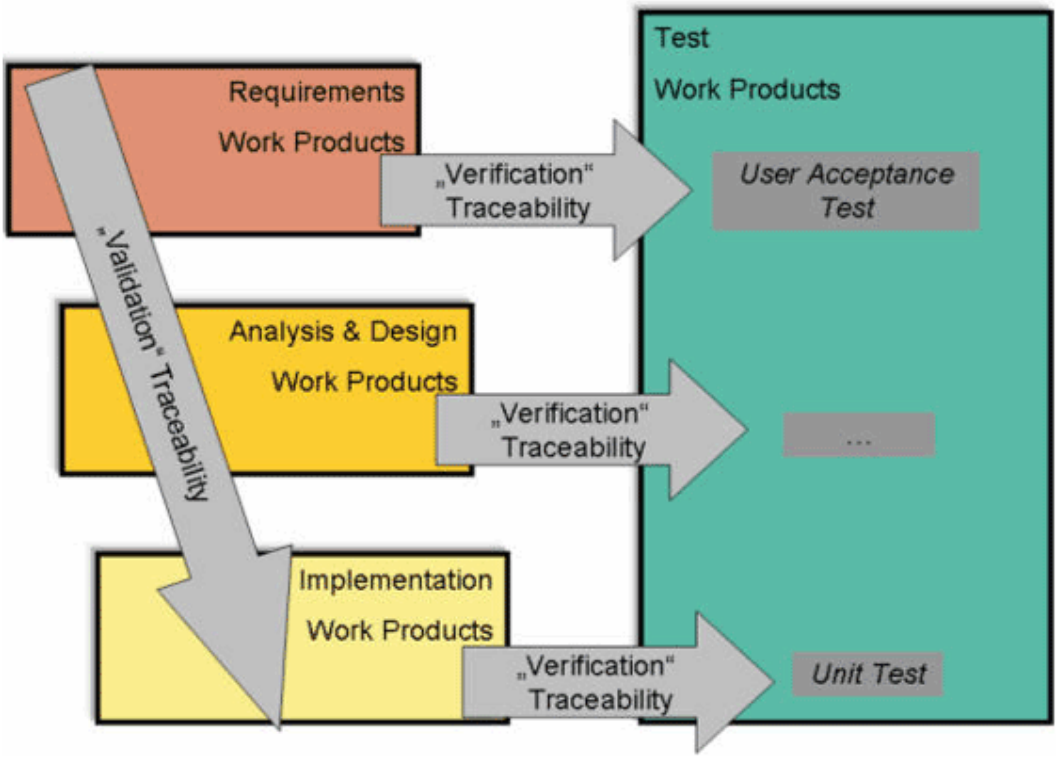


Figure 9: Simple traceability overview [30]

Figure 9 describes the most basic "dimensions" of traceability. In [9] Ramesh and Jarke describes a traceability metamodel supporting 6 dimensions of traceability:

1. *What* is represented? – A classification of the trace information that was captured. E.g. is it a *requirement*, *design* or *a rationale*.
2. *Who* are the stakeholders? – Stakeholders play different roles in the creation, maintenance and use of various trace information, and may view the information differently. Examples of stakeholders include project managers, system analysts, designers etc.
3. *Where* is the actual artefact that is being traced documented? – E.g. a meeting or design document.
4. *How* this information is represented – how the documentation documenting the actual artefact is represented.
5. *Why* was the trace information created? – The rationale behind the creation, modification, and evolution of the trace information.
6. *When* the information was captured.

The use of traceability varies with the development phase. Some typical use scenarios include [10]:

- **Planning** – it may be advantageous to link various decisions made during planning with the artefacts they have an impact on. This information may be used as a rationale at later stages.
- **Design** – requirements may be linked to the artefacts that are meant to satisfy them. Furthermore links may be made between artefacts representing a relationship between two software components.
- **Implementation** – links may be created manually or automatically between different models and between models and implementation code. This includes both manually created artefacts and artefacts that are the result of a transformation.
- **Testing/deployment** – Traceability information can be used to check that all requirements are satisfied by one or more artefacts in the system and that tests exists for each requirement. This analysis can also be performed during earlier stages to monitor the progress of the development process.
- **Maintenance** – traceability information may be used to identify bottlenecks.
- **Upgrade/change** – If changes are to be made to the system, traceability information may be used to conduct an impact analysis.

The level of details of the trace information that is captured varies a great deal between different projects and organizations, depending on time lines, organizational needs or the development strategy. Ramesh and Jarke separates traceability users into two main groups with respect to how they use and capture traceability information [9]**:**

1. ***Low-End*** use of traceability is typically used by organizations that use traceability to link various artefacts together without providing any semantics to the relationships between the artefacts that are being traced. Traceability in this context is typically used to link requirements to the actual system components that satisfy them. Low-end users typically lack in the capturing of rationale, making it difficult to find out how issues related to the requirements were resolved. This makes analysis of the trace information more difficult as one may not be able to determine how a system component actually satisfies a requirement. Furthermore, not knowing the rationale

behind a decision may make it difficult to accurately tell the impact of a given change.

2. ***High-end*** users of traceability use traceability in a much richer way. This is supported mainly by using semantically richer classification schemes, but often also by tracing a richer set of artefacts throughout the development process. This allows information regarding e.g. decisions, trade-offs, or the level of criticality of a requirement to be captured. This information may prove to be valuable later in the development process, and enables easier retrieval and more precise reasoning about traces.

What kind of analysis that may be performed on the trace information thus depends on whether low-end or high-end use of traceability is applied, i.e. how semantically rich the trace information is.

In an MDE process traceability is crucial. Due to the extensive use of transformations (i.e. automated creation of artefacts) used throughout an MDE process it becomes central to be able to understand how and why an artefact was created [8, 31]. Consider for instance the transformation of the PIM of the library system in Figure 4 to the PSM in Figure 5 described in subsection 3.2.5.1. The person responsible for running the transformation might not be the same person that created the mapping, and thus does not necessary know exactly how the transformation works and what will be the result. It could therefore be valuable and timesaving to be able to e.g. find the source of the class *<eJB_Stateless>LibrarySystem* (or the other way around) to get an understanding of why it was created. For more information about the logic of the transformation, the developer would perhaps want to check if there is a rationale explaining the mapping rule responsible for the transformation by performing a simple query on the trace information.

Traceability information should be created and maintained as transparently as possible [2, 24], meaning that developers should not have to maintain the traceability information themselves, at least as little as possible. It should rather be maintained automatically by the tools. Having the tools deal with traceability automatically means that traceability information will be maintained without making the developments process more complicated, while at the same time eliminating the risk that developers neglect it or makes errors doing it.

An interesting side effect of the automated or, in most cases, semi automated development process of MDE is that it actually provides an opportunity to automate the creation and discovery of trace relationships between artefacts, but also to maintain this information. Clearly, the transformation engine responsible for transforming the artefacts of one model to the artefacts of another, or to the text in a text-file is also aware that there is a relationship between the source and target of the transformation. In fact most transformation engines use an internal mechanism to keep track of these relationships.

3.3.2 Storing Traceability Information

In [31] Kolovos et al claims that there are two main approaches to deal with traceability in a model based environment. One is to keep the traceability information embedded in the model itself, as new model elements e.g. as stereotypes or attributes. The other is to keep the traceability information in an external model. Both these approaches have their pros and cons and they have both been implemented in various ways.

Kolovos et al discusses how the first approach *"is popular with modellers for its human-friendliness as it represents traceability links as visual model elements that people can easily inspect and navigate"* [31]. The reason for this is that keeping the traceability information in the model itself means that just looking at the model makes it possible to see what information is traced, and how. This approach also makes it easy to physically move the models that are being traced to different locations, as the trace information is contained in the same model. A downside of this approach is that the trace information pollutes the models, as the trace links/information becomes hard to distinguish from the rest of the elements in the model. This could make it harder for a tool to separate them from each other. Another downside of this approach is that it makes it difficult to create traceability links between artefacts contained in different models, as there is no obvious solution to where the trace information concerning the intermodel relationships should be stored.

The other approach discussed in [31], using external traceability links, e.g. using a separate model to store the traceability links, has the advantage of keeping the models clean (not polluting them with traceability information) by facilitating loose couplings between the models and the links. Thus, the models themselves do not have to know about the traceability links between them, as this information is kept in separate models. This makes sense, because the trace information is not really part of the source or target artefacts of a transformation, but rather a property of the transformation. The downside of this approach is that keeping the traceability information in a separate model could make it harder for human beings to understand it, because just looking at the traceability model does not necessarily make it clear what is being traced. Thus using this approach indicates that there might have to be a mechanism that makes the traceability information understandable for humans. Keeping the trace information in separate models also brings the challenge of keeping the traceability information in synch with the models that are being traced.

Despite the advantages and disadvantages of both approaches, Kolovos et al concludes that the external approach is to prefer, as it handles both internal and external traceability links, and is therefore more flexible. At the same time it prevents model pollution. They even discuss how it is possible to use the best from both of them in what they call *on-demand merging*. With this approach they keep the traceability information in an external model, thus avoiding model pollution, and then use regular transformations to merge the traceability models with the models they keep information about on demand. The new model produced by this transformation will then contain e.g. the classes from the models being traced, together with the information from the traceability model. This provides a model with all the information needed to describe how different models are linked together with traceability links, and what traceability information is being stored. At the same time the original models are kept intact and unpolluted. One could have several different merging strategies, or different "views", that could be applied to get different trace information from the models or show it in different ways.

Figure 10 describes a simple traceability metamodel. The metamodel describe a *TraceModel* containing *TraceLink*s. The *TraceLink* contains references to a source element and a target element, but the target element (which may be any kind of model element) is not contained in the *TraceModel* – it is part of an external model. In a real-life scenario this model would probably not provide the proper functionality expected by a traceability-metamodel, but it is sufficient to help illustrate the examples presented here.

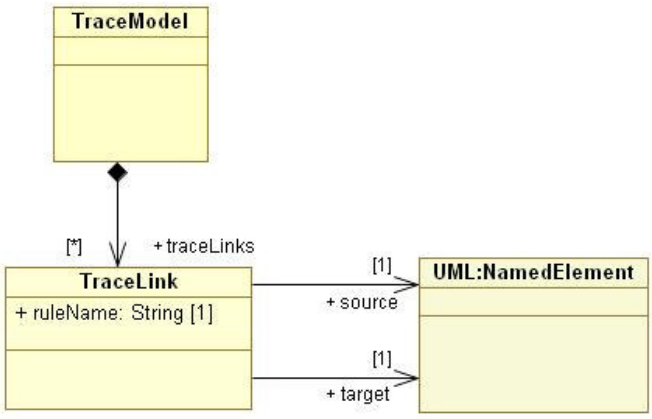


Figure 10: A simple traceability metamodel

Using the traceability metamodel in Figure 10, one would be able to create an external trace model containing the traceability information of the transformation from the PIM to the PSM in our running example. Figure 11 shows the result of using this traceability metamodel to generate trace links between the class 'CustomerSystem' in the PIM and PSM using implicit linkage between the input and output of the transformation. In the illustration, the different models are illustrated with packages. The traceability information is contained in a separate model called 'TraceabilityModel'.

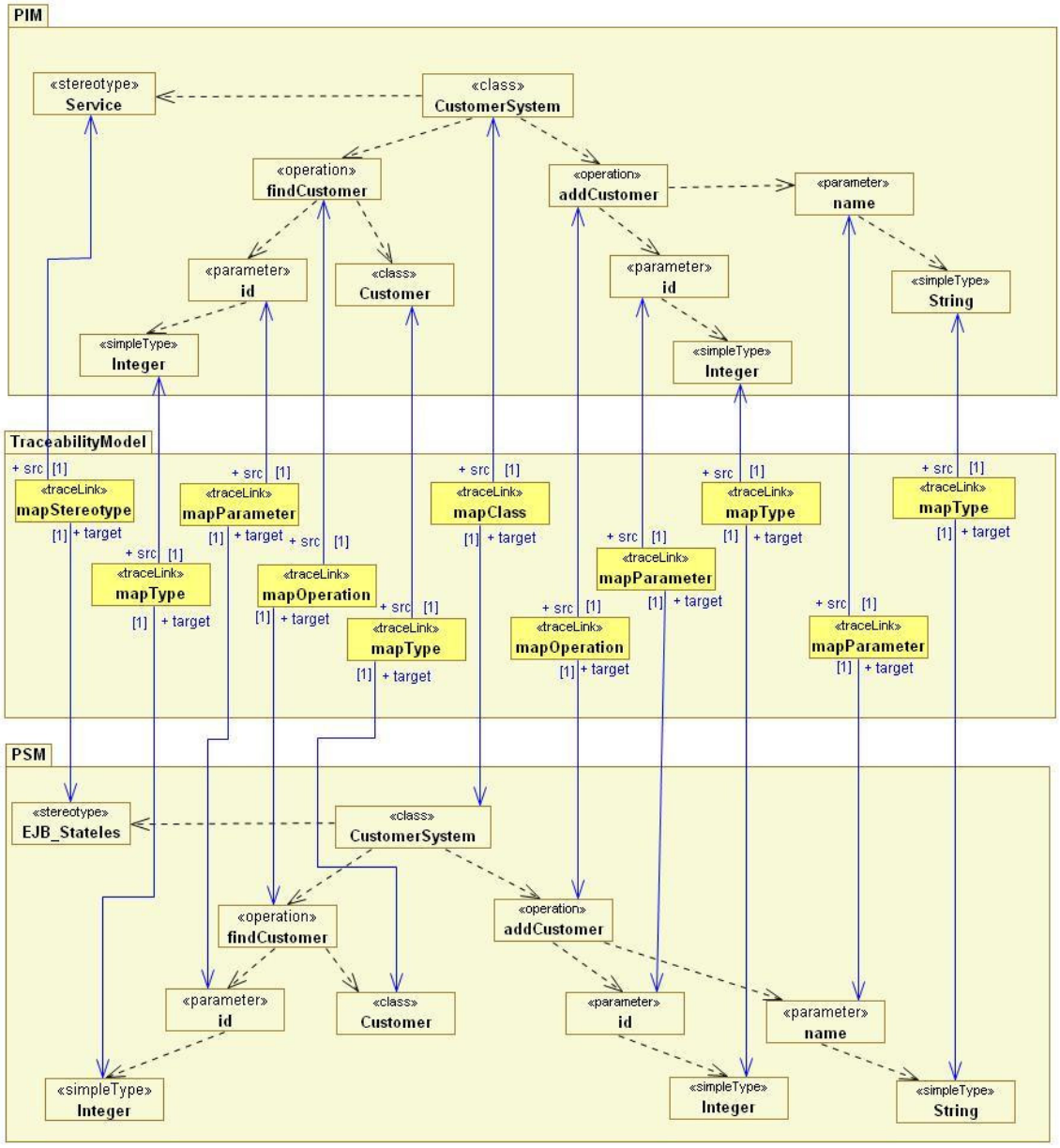


Figure 11: A simple traceability example

Although the trace links presented in this example is very simple, they illustrate how external trace links can be used to provide traceability between the source and target elements of a transformation. This example illustrates a model to model transformation, but the same principle can be used for model to text transformations as long as the traced artefacts in the text are uniquely identified. Provided a tool that allows developers to query and/or navigate these trace links, developers could now find all the artefacts that was generated by the transformation, and see what mapping rule that was used to generate it.

The example in Figure 11 also shows that the trace model quickly becomes quite large and hard to read. Even when it only contains the trace links from 1 class in the PIM it is starting to get hard to navigate through it visually. This problem could be solved to some degree by using a more suitable trace metamodel. One could for example have different types of trace

links, and each transformation rule could result in 1 trace link containing references to all source and target elements. Changes like these could help readability quite a bit, but with more complex source and target models, the trace information might still be difficult to read. As discussed above, this could be solved by applying on-demand merging, as described in [31].

3.3.3 Traceability in Model to Model Transformations

Trace links between source and target artefacts of a transformation may be created *implicit* (i.e. the transformation engine creates links between the source and the target automatically) or *explicit* (i.e. the user explicitly defines how to links the source and the target artefacts of a transformation) [8].

The fist approach is often used internally by transformation engines like Queries Views Transformation QVT [26] and Atlas Transformation Language (ATL) [32] to keep track of a transformation [11]. In QVT, the source and the target artefacts of a transformation are linked by *trace classes* containing properties that refer to objects and values in models that are related by a transformation [26]; each relation (defined by a declarative rule, maps a source artefact to a target artefact) is represented by trace class with properties referencing the features of the source artefact and the features that are generated in the target artefacts.

While making it possible to automatically create traceability links between the source and the target, implicit links means that developers has little control of how and when traceability information is created. This means that it is difficult to add additional semantics to the trace links, and means that the traceability information is limited to that defined by the tool.

In [11] Jouault suggests that with the help of any transformation language supporting two or more output models, like the ATL, one could define explicit links between the source and the target artefacts. What he suggests is that in addition to creating mappings from the target model to a source model, there should also be a mapping creating instances of the traceability metamodel, thus creating a traceability model containing information about the transformation. This does however require the existence of a metamodel that describes the external trace model to be created, in the same way as one need a metamodel describing the source and target model of any other transformation. This is a simple solution, and it does not require any additional functionality extending what is already part of the transformation language. One would simply have to create additional "output patterns" in the mapping rules, that describe the elements of the traceability model to be created. It would however require the developers to create the mapping to a trace model each time they created a mapping. This would make the development process more cumbersome, thus increasing the possibility that someone forgets to do it or does not care to do it, and even more likely; makes errors while doing it.

To avoid this Jouault describes how one can transform regular transformation code into transformation code with support for traceability by running the transformation code through a kind of pre-processor to perform a *higher order transformation* (i.e., transform the transformation code). This is possible because transformation rules are actually models themselves [11]. Given a library of traceability mappings, one can therefore transform the transformation code by adding code that create instances of a trace metamodel in addition to the regular transformation code, as explained above.

Another possibility is to provide special constructs in a transformation language that allows developers to explicitly define trace links between the input and the output. This approach is taken by the MOF to Text transformation language discussed in the next subsection.

Both the approaches presented in this subsection have their pros and cons, but an optimal solution should support a combination of both [8]. While the first approach provides a more accurate record of the transformation, the second may be used to link artefacts that are not possible to link automatically, and is more flexible with respect to how traceability links are created.

3.3.4 Traceability in Model to Text Transformations

In an MDE environment it is also desirable to use models to generate code. To maintain consistency between the models and the generated code, one also need to keep traces of which elements in the model that were used to generate specific parts of the code.

Because every artefact in an MDE environment is considered to be models, the code is also considered to be a model. To perform the code to text transformation one cannot use a model to model transformation language though, because they only create new model elements as the result of the transformations. A model to text transformation would of course need to create the textual code.

One such transformation language is the MOFScript. This language can be used to generate text from any MOF based model [21]. In [24] Oldevik and Neple discusses how traceability in model to text transformations could be achieved using this language. In the approach they describe, each text-file is represented by a model. In this model (described by the metamodel in Figure 12) the text is divided into protected blocks with a start offset and an end offset telling what line and position on the line the block starts and ends. The blocks are furthermore divided into traceable segments, that have their own start- and end offset, and describes where inside the block the traceable segment resides. A traceable segment could e.g. be the name of a class etc. These traceable segments are then linked to a trace element that tells what transformation rule that was used to generate it. The trace element is in turn linked to an element with a reference to a model element in the source model, i.e. the element used to create the code segment. To achieve this, the developer will not have to do anything because this is handled automatically by the tool. What the developers can do however is to use the keyword *unprotect*, which means that the following block is not protected, meaning that the code resided there can be changed, and there will not be kept traces of it. Such unprotected blocks would typically be used for areas where the developers are meant to enter their own code.

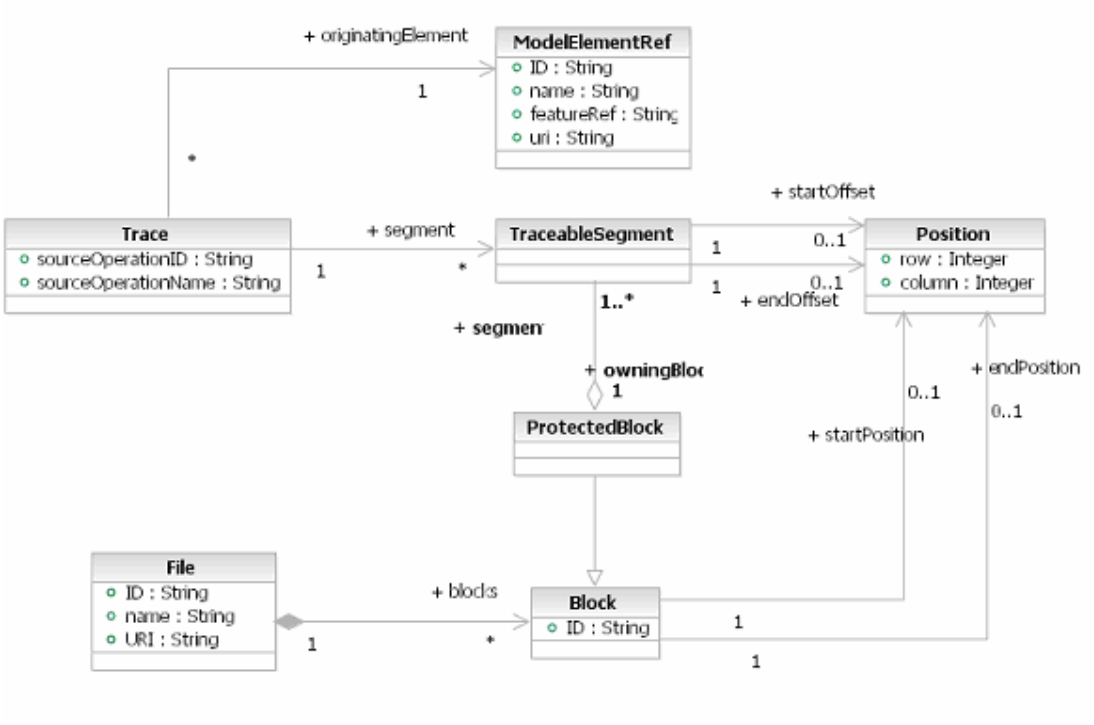


Figure 12: Trace Metamodel [24]

Figure 13 illustrates how the trace model with the traces from the code generation from class *CustomerSystem* in the PSM in the running example could look like. This transformation is illustrated in Figure 8. The model consists of a single *File* with a single *Block*. This *Block* contains several *TraceableSegment*s, which identifies an area within the *Block*. These *TraceableSegment*s are traced to a *ModelElementRef*, containing a reference to the model element from which the text represented by the *TraceableSegment* was generated. E.g. the *TraceableSegment* 'trSeg1' specifies the location of the text 'Local' (@Local) in the top of the java file. This text is located in the java file in line 3, starting at column 2 and ending in column 7 (the indexing of the file and the *Block* are the same as there are only 1 *Block* contained by this *File*). This can be traced ('trace1') back to the value of a *Property* contained by the *Stereotype* 'EJB_Stateless'.

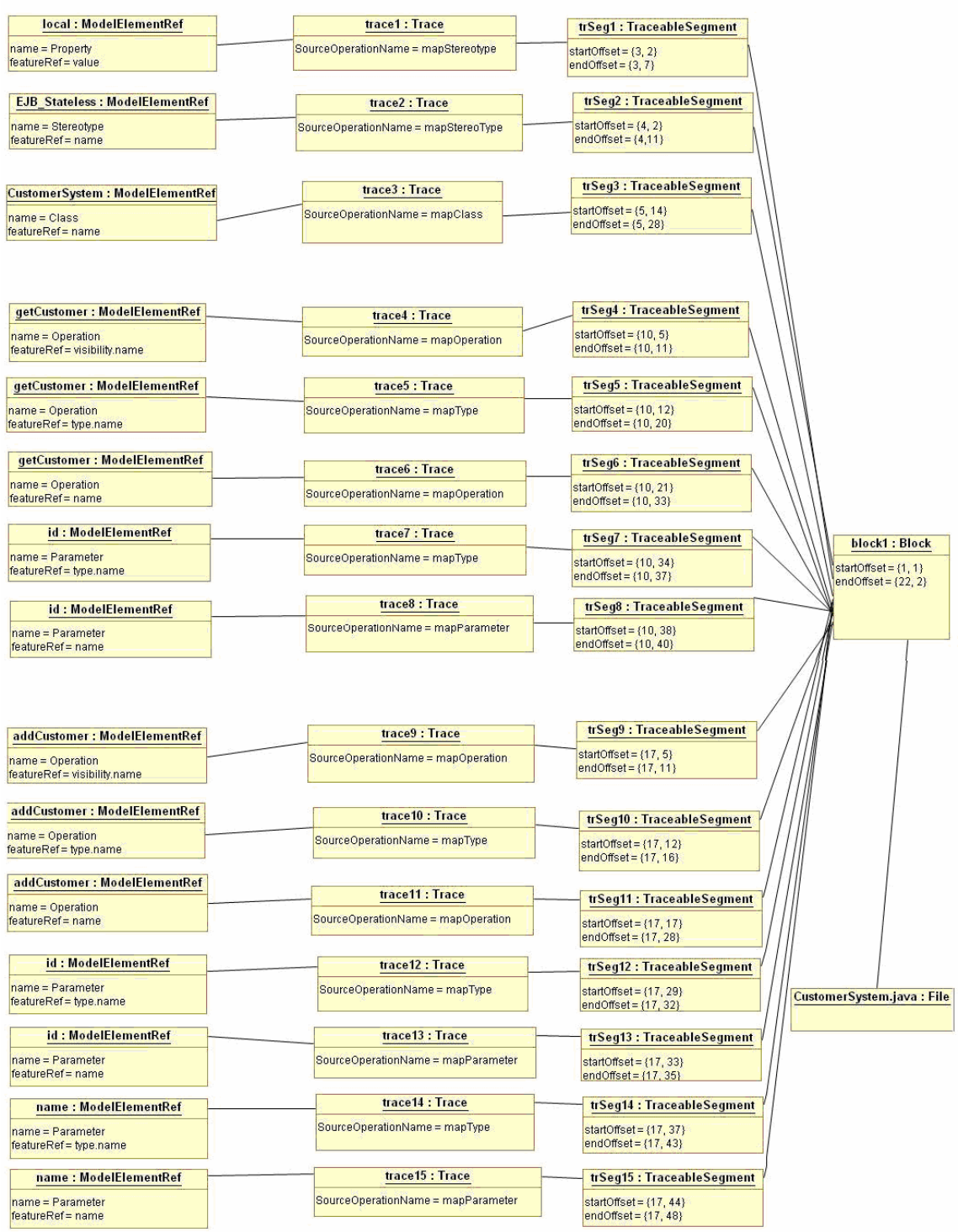


Figure 13: Traces from the transformation illustrated in Figure 8

A different approach is proposed in the MOF Model to Text Transformation language specification [27] from OMG. Instead of creating explicit links between the source and the target of a transformation, the language allows users to explicitly define trace blocks, and relating these trace blocks to model elements. This gives the user better control of the trace

generation, and is especially useful for adding traces to parts of the code that are not easily automated [8].

```
[template public classToJava(c : Class)]
[trace(c.id()+ '_definition') ]
class [c.name/]
{
   // Constructor
   [c.name/]()
   {
   [protected('user_code')]
   ; user code
   [/protected]

   }
}
[/trace]
[/template]
```

Figure 14: Definition of explicit trace block in MOF2Text [27]

Figure 14 shows the definition of a trace block containing a generated class. The block is explicitly linked to the class 'c'.

If both approaches were applied to the transformation engine responsible for running the mapping in Figure 14, the resulting explicit links would tell us how the names of features of Class 'c' where related to the text, while the explicit link would link the Class itself with the entire block of text comprising the Class in the Java file. This information could be utilised for different kinds of analysis.

3.3.5 Application of Traceability in MDE

As mentioned in the introduction to this section, the main application of traceability is to support analysis of the development process. Walderhaug et al [10] discuss several different traceability use cases; *Trace Inspection*, *Coverage Analysis*, *Orphan Analysis*, and *Reverse Engineering*. Vanhoof et al [33] discuss how traceability information can be used as input for model transformation s to improve the transformations. These scenarios will be discussed in the following subsections.

3.3.5.1 Trace Inspection

The purpose of trace inspection is to allow the traceability information to be visualised, navigated and queried, so that it may give the user better understanding of the system, both during development and maintenance [10]. Trace inspection could e.g. be used to retrieve all artefacts that were generated from a certain artefact, and find the transformations responsible for creating them.

3.3.5.2 Coverage Analysis

Through coverage analysis, the degree to which some artefacts of the system are followed up by other artefacts in the system can be determined [10]. This can be achieved by checking that trace relationships that should exist are present, e.g. that a requirement is covered by design

and implementation [10], or that all relevant parts of a model are utilised by a transformation [8].

3.3.5.3 Change Impact Analysis

Change impact analysis may be used to identify artefacts that may be affected by a change [10, 30], and to estimate the cost, resources and time required to perform the change [10].

3.3.5.4 Orphan Analysis

Orphan analysis may be used to find artefacts that are not the target of any trace link of a specific type [10]. Typical use of orphan analysis is to find elements that are not required by the system, e.g. a feature that was not described in the requirements. An other example could be the deletion of an artefact that was used to generate another through a transformation – the generated artefact would then become an orphan [8].

3.3.5.5 Reverse Engineering

Reverse engineering may be used to rebuild a source artefact from the target artefact[10]. Traceability information is necessary to be able to bring an artefact back to its original state[34]. E.g. the traceability information generated by MOFScript may be used to rebuild a class model based on the Java classes, provided that the required information is found. The degree of how accurately an artefact may be regenerated depends on how much of the information found in the original model that was used in the transformation.

3.3.5.6 Traceability as Input for Model Transformations

In cases where several transformations are used to generate several intermediate models through a chain of transformation, the traceability information may be used to retrieve information regarding the relationships between artefacts that cannot be found in the source model [33]. E.g. the same model may be used to generate a class model and a relational database model. If the two target models were to be used to generate java code, the traceability information could be used to retrieve the relationships between the classes and the database tables.

3.3.6 Current Traceability Tools and Solutions

Earlier in this chapter we have illustrated the use of two simple traceability metamodels (Figure 10 and Figure 12). Although the first one is very simple, they illustrate metamodels that would make it possible to maintain traces from model to model transformations and model to text transformations. This makes it possible to trace all the steps in an automated MDE process, thus we have (to some degree) achieved traceability in MDE. However, since the transformation engines in these examples uses different traceability metamodels, it will not be straight forward to analyse the trace information. E.g. traces related to the artefacts in the PSM were captured by two different trace models and this information was captured in two different formats with possibly different semantics. This means that one would need a mapping between the two metamodels to be able to follow the traces throughout the entire process, e.g. from PIM to PSM to code. Such mappings would have to be performed between trace models used by different transformation engines. Additionally, it might be desirable to be able to keep traces from requirements and/or rationales kept in a text document as well,

which would not be possible using the traceability metamodels in the examples. To be able store traces regarding the whole development process it would hence be desirable with a more general purpose traceability metamodel, covering all the different aspects associated with the software development process.

In [9] Ramesh and Jarke suggest a general purpose metamodel for requirements traceability which covers the most basic aspects of traceability (Figure 15). This metamodels allows traces between any kind of *Object*s to be captured, but also allows traces to the *Source* (documentation) and *Stakeholder* associated with the traced *Object* to be captured.

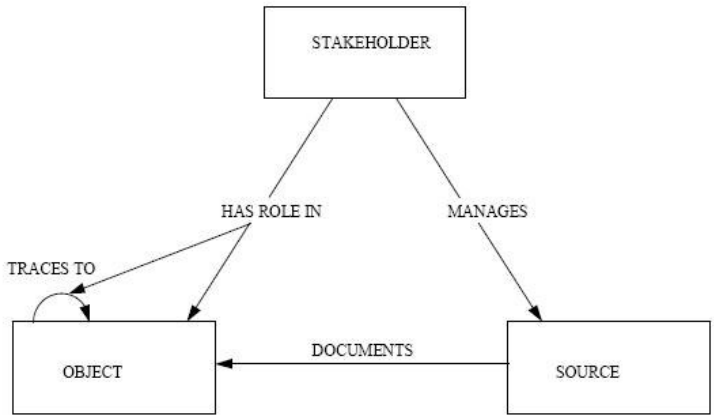


Figure 15: Traceability metamodel [9]

Traceability models based on this metamodels is assumed to be implemented in a trace repository. It is widely accepted that such a repository will comprise at least three layers [9]:

- The metamodel defining the language in which the traceability models can be defined (E.g. Figure 15).
- A set of customizable trace models defined in the language defined by the metamodel.
- A database storing the actual traces.

Walderhaug et al [10] suggests a more elaborated traceability metamodel for MDE aiming to provide flexible and customizable trace models, support throughout the lifetime of artefacts, and support traces between different tools. This trace model consists of 3 packages:

1. The **Traceability Metamodel** is where trace types are defined. In the trace model it is specified what kind of model artefacts that can be traced, what kind of traces that can be created, and what relationships they may have to each other.
2. The **Traceability System** is where models can be created, based on the types defined in the *Traceability Metamodel*. This is where the traceability models are stored.
3. The **Traceability Use** is the bases for the user interface, and is the interface to the repository stored in the Traceability System. This interface will be used by tools that want to access the repository.

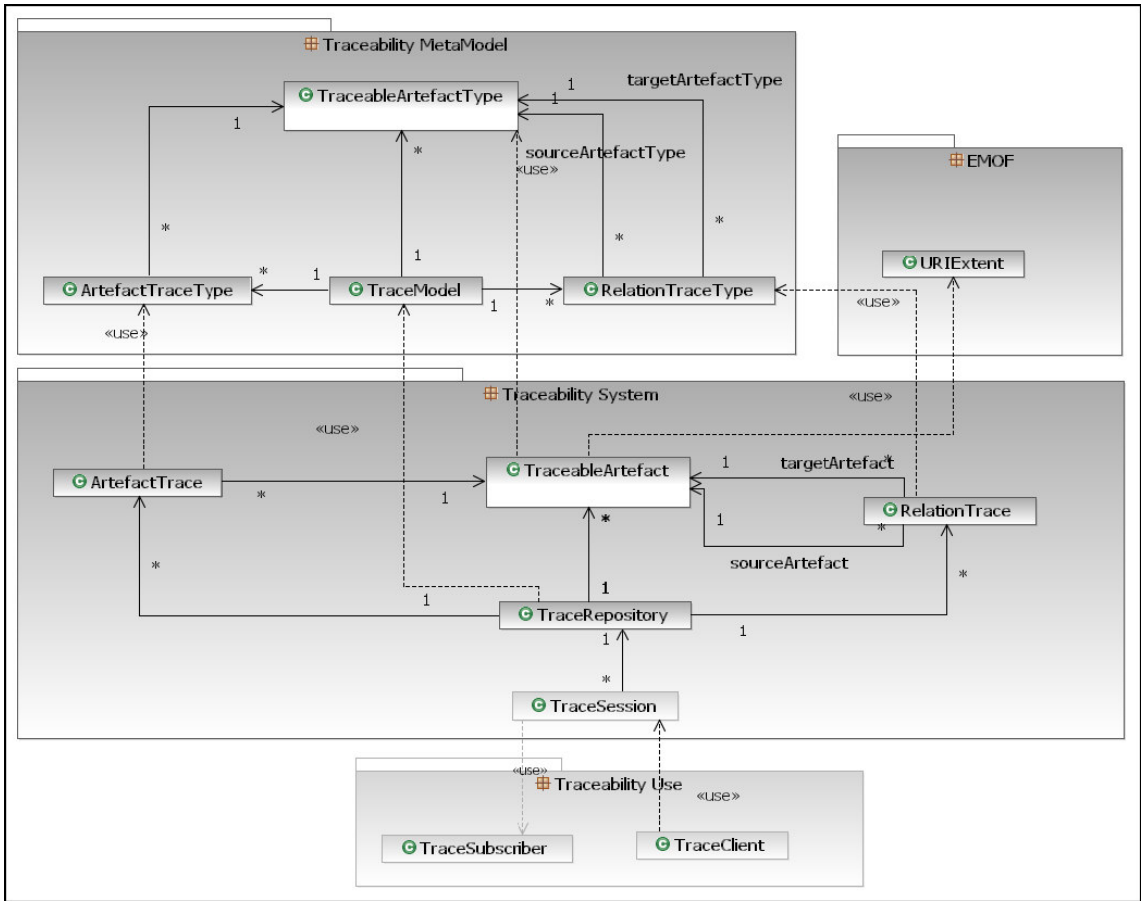


Figure 16: A generic solution for traceability [3]

This metamodel describes a language for describing both the trace types (*Traceability Metamodel*), allowing developers to create customizable traceability models, and the models that are used to capture the traces (Traceability System).

Developers would define what kind of model artefacts that may be traced by creating a *TraceableArtefactType* representing each model artefact type. The *TraceableArtefactType* may contain a mapping from the model artefact type to itself, i.e. how to transform the model artefact to a *TraceableArtefact* with the *TraceableArtefactType* as type. *ArtefactTraceTypes* may then be defined to represent types of traces which may be created for a *TraceableArtefactType*. Similarly, *RelationTraceTypes* may be created to represent types of traces that may be created for *TraceableArtefactType*s that represent relations between model artefacts. Developers may also specify sets of actions which may be executed when the different types are created (these are contained by the types them selves), they may also specify attributes to be created for the different types. When developers have defined the *TraceableArtefactTypes*, *ArtefactTraceType*s, and *RelationTraceType*s they want to use they can use the *TraceModel* containing these types as the types of the actual traces contained by a *TraceRepository*. The TraceModel hence define the types used in the *TraceRepository*, and may also define when these traces shall be created, how they shall be mapped, and what actions that may be performed when traces of a certain type is created.

There are however no references to an implementation of this solution, and the authors leave much of the work related to mapping of artefacts (specified by *TraceableArtefactType*), and specification of actions to future work.

3.3.7 Challenges of Traceability in MDE

3.3.7.1 Classification of Traceability Information

One of the major issues of traceability in MDE, and traceability in general, is the lack of standard classification schemes for trace information. A traceability solution like the one suggested by Walderhaug et al [3, 10], might provide the means to generically define customizable trace models with user defined types, and allow this information to be shared between different users and tools, but without standard classification schemes, the semantics of the trace information might be hard to determine. The semantics of the trace information is crucial for traceability management, both manually and automatic. Both tools and humans must for instance know when to create the right kind of trace information, and in order to analyse the trace information correctly it is important that the information is interpreted the way it was intended by the creator of the information. Semantically rich trace information is also a key factor in order to allow a rich set of analysis to be performed on the information.

A number of different trace link classifications have been presented in the literature, most regarding requirements traceability. Ramesh and Jarke define two base classes of traceability links[9];

1. ***Product-related*** – describes properties and relationships of design artefacts independent of how they were created. This class has two basic types of traceability links; *Satisfied* (e.g. to express that a requirement or standard is satisfied by an artefact on a lower level) and *Dependency* expressing a dependency between to artefacts.

2. ***Process-related*** – describes the history of actions taken in the process itself. The two base classes of product-related links are; *Evolves-To* expressing that the source has evolved to the target, and Rationale which is a link to the *Rationale* behind an action that led to the evolution.

Espinoza et al [6] performs an analysis on current requirements traceability approaches and identifies common features of existing traceability approaches. The traceability links are divided into six different categories, comprised of traceability links with similar semantics; *Satisfies*, *Dependency*, *Rationale*, *Validation*, *Verification*, *Evolution*. They conclude however, that there is little guidance on how the links shall be used, and for what artefact types the links should be applied to. Furthermore, the definitions found are often overlapping.

The literature discussed this far in this subsection primarily discusses classification of traceability information in the context of requirements traceability however. Walderhaug et al [10] suggests a set of extensions to the traceability system solution metamodel in Figure 16 that provides classification of traceability links for MDE. This includes the addition of an attribute of type *LinkType* on the *RelationTraceType* Class. *LinkType* is extended by the Classes *Automatic* and *Manual*, and hence reflect how a specific traceability link was created. *Automatic* links may be either *Generated*, *Derived*, or *Inferred*. The semantics of the links are described in Table 2.

Nr.	Classification	Sementics
LT_Ext 1	LinkType	The *LinkType* class describes how the *RelationTraceType* is created, either automatically or manually.
LT_Ext 1.1	Automatic	An automatic link may be created in one of three ways. This is specified by an enumeration attribute with the following values:
*	*Generated*	Creation of a *RelationTrace* initiated from a system event, condition or a user interaction. Typically occurs during transformations.
*	*Derived*	An explicit *RelationTrace* derived from other artefact and/or *RelationTraces*. Typically created as a response to a requirement to store specific types of relation between certain artefact types or relationships.
*	*Implied*	A *RelationTrace* that represents a one-way logical dependency between two artefacts based on one or more statements that are assumed to be true.
LT_Ext 1.2	Manual	Links created manually by a developer.

Table 2: LinkType extensions to RelationTraceType (described in [10])

The *LinkType* classification may be applied by any traceability link, as every link is either automatically or manually created. To add semantics to the traceability links, a *RelationTraceType* may be of one of the subclasses described in Table 3. This allows semantics regarding the creation of a link, and semantics regarding the meaning of a link to be captured.

Evolution				
Nr.	**Classification**	**Sementics**	**LinkType**	**Use Case**
RTT_Ext 1	Realization	The trace between two artefacts where the target implements the source.	Generated, manual	Impact, Coverage, Orphan
RTT_Ext 2	Transformation	The trace of a transformation from source to target artefact.	Generated	Impact, Coverage, Orphan
Satisfication				
Nr.	**Classification**	**Sementics**	**LinkType**	**Use Case**
RTT_Ext 3	Verification	A trace between two artefacts where the source is verification for the target. E.g. a JUnit that test the behaviour of a Java Class.	Generated, manual	Impact, Coverage, Orphan (for the verification itself)
RTT_Ext 4	Validate	A trace between two artefacts where the source validates the target.	Generated, manual	Impact, Coverage, Orphan
RTT_Ext 5	Acceptance	A trace between two artefacts where the source defines the acceptance for the target.	Generated, manual	Impact, Coverage, Orphan
RTT_Ext 6	Conflict	A trace that documents two conflicting artefacts	Generated, manual	Impact, Coverage
Dependency				
Nr.	**Classification**	**Sementics**	**LinkType**	**Use Case**
RTT_Ext 7	Dependency	A trace between a source that is dependent of the target.	Generated, manual	Impact, Orphan
Rationale				
Nr.	**Classification**	**Sementics**	**LinkType**	**Use Case**
RTT_Ext 8	Manages	Provides information about whom/what (source) manages who/what (target).Focused on the operational aspect of the artefacts.	Generated, manual	Impact
RTT_Ext 9	Rationale	A trace between two artefacts where the source is the rationale or justification for the target artefact.	Manual	Impact, Orphan
RTT_Ext 10	Reponsibility	Provides information about who/what (source) is responsible for who/what (target)	Manual	Impact
RTT_Ext 11	Ownership	Provides information about who/what (source) owns who/what (target)	Generated, manual	Impact

Table 3: RelationTraceType extensions (described in [10])

3.3.7.2 Maintaining the Correctness of Traceability Information

One of the most challenging aspects of traceability is how to maintain the correctness and relevance of relationships while the artefacts continue to change and evolve [35]. This is

especially the case when the trace information is maintained manually, as this requires a lot of effort. For traceability information to be useful, the information must reflect the current dependencies between artefacts [5]. Current requirements management tools, such as IBM RequisitePro [36] and Telelogic DOORS [37], contain features supporting the management of traceability information validity by monitoring changes of linked artefacts and indicating suspect links. However, the number of suspect links in most non-trivial projects quickly becomes excessive, drastically reducing the usefulness of this feature [5]. The degree of formality of the artefacts that are being stored also has an impact on how effective this feature can be utilised [35]. Furthermore, in subsection 3.3.2 we concluded that the trace information should be kept in a separate model. This might complicate things even more, as there might not be any direct linkage between the trace model and the artefact being traced.

3.4 Summary and Discussion

In the chapter we have discussed how MDE promises to ease complexity in the system development process by using transformations to automatically generate models, formalised by metamodels, at various levels of abstraction, and generating implementation code from the models. The extensive use of transformations in MDE does however bring added complexity in a new area, as it may not be obvious how models and code at various levels are related to each other.

Traceability therefore becomes a critical success factor in MDE, as it offers a way to keep track of the evolution of a system by recording the process that led to a particular point in a process. This information may be used for various analysis purposes throughout the lifetime of a system.

We have however seen that different transformation engines support traceability in different ways. Some generate *implicit* traceability information, while others leaves to the user to *explicitly* define this information themselves using special language constructs or by generating traceability models as additional target models. Moreover, even though most of the transformation engines use an internal trace model to record the relationships between source and target artefacts of a transformation, this information is created using different languages; specific to either *model to model* transformations or model to *model to text* transformations.

Even though tools like MOFScript offer some analysis functionality on the traceability information, the fact that each tool operates on different languages make it difficult to integrate the traceability information from different tools. This makes it difficult to get a holistic view of the system development process.

Adding the fact that different organisations use traceability information differently; some organisations use *high-end* traceability while others use *low-end* traceability, at various degrees makes the situation even more complicated, as the various levels of details might make it difficult to exchange the information between different organisations.

Indeed Walderhaug et al [10] propose a system solution for a more holistic traceability approach, and a classification scheme for traceability in MDE. There is however no references to an implementation of such a traceability tool and they do not provide information on how the classification scheme and tool could be utilised to support the different needs discussed above.

Summarising the discussion, we believe that there is a need for a traceability tool and a classification scheme supporting the different traceability strategies applied by different tools and organisations in a way that allows traceability information to be captured on different levels of details, but at the same time allow different tools and users to operate on the same traceability information.

4 The Traceability Tool

4.1 Introduction

In this chapter we present the design and functionality of the traceability tool. We start by listing a set of requirements which should be satisfied by the tool in section 4.2, before the design of the tool is presented in section 4.4. Through the rest of the chapter we discuss the features provided by the tool, before we end the chapter with a summary of the discussion in section 4.11.

4.2 Tool Requirements

In this section we introduce the requirements for the traceability tool. These are requirements that should be fulfilled for any generic traceability tool to be used in MDE. These requirements will hence also serve as success criteria for *H2* (hypothesis defined in section 1.3), and will be used to validate the fulfilment of this hypothesis.

Tool Requirement 1

The traceability tool shall use model-driven approaches.

As we operate in an MDE environment, it is logical to treat traceability as just another model. This is also in accordance with the discussion in section 3.3, and means that the traceability tool can be integrated more easily with other model-based tools and models in general.

Tool Requirement 2

The models used by the traceability tool shall conform to the 4 meta-layers of OMG.

In subsection 3.2.2 we saw that OMG defined 4 meta-layers to which a model could conform. The design of the tool should conform to these as this is a widely accepted standard, and hence makes the tool more easily integrated with other tools. This should however not be a problem, as most modelling tools today conforms to this standard.

Tool Requirement 3

The traceability tool shall support trace repositories.

The trace information must be stored in a repository for persistence. As discussed in subsection 3.3.1, such a repository should comprise at least three layers of details (metamodel, reference models, database of traces). This leads to the following requirements:

Tool Requirement 3.1

The traceability tool shall be based on a metamodel.

A metamodel formally defines the language on which the traceability tool will operate to support traceability.

Tool Requirement 3.2

The traceability tool shall support generic definition and customization of reference models.

In order to support various kinds of traceability information to be captured, the traceability tool shall allow users to define their own reference models. This allows the creation of project specific reference models, and customization of previously defined reference models.

Tool Requirement 3.3

The traceability tool shall support persistent storage of traceability information.

In order to support traceability in a sufficient way, persistence storage of traceability information is essential.

Tool Requirement 4

It shall be possible to identify where the traced artefact is located.

In order for the traceability information to be of any use, it must be possible to locate the artefacts that are being traced.

Tool Requirement 5

It shall be possible to identify when the traceability information was recorded.

When a piece of traceability information was recorded may be of interest for different analysis purposes.

Tool Requirement 6

The traceability tool shall be implemented as an Eclipse EMF plug-in.

Eclipse is today a commonly used and well tested standard for tool development, and should therefore serve as a good platform to build on [38]. Eclipse Modelling Framework (EMF) [12] has also become a widely appreciated open source platform within the Eclipse community that provide a variety of tools for creating, maintaining, and manipulating models. EMF supports the four meta-layers of OMG, and is used by a variety of modelling tools.

Tool Requirement 7

The traceability tool shall be easy to integrate with external plug-ins.

For the traceability tool to be of any real use, it is essential that it is easy to integrate with external tools. Due to the generality of such a traceability tool, it is essential that third party plug-ins is able to easily adapt the functionality of the traceability tool to use it to fit their own demands, and usage.

Tool Requirement 8

It shall be possible to create traceability information both automatically and manually.

Traceability information may be created in several different ways; the information may be recorded manually by a developer, or automatically by a modelling tool or through transformations (e.g. through the procedure presented in [11]). A traceability tool should therefore support all these use cases.

Tool Requirement 9

The traceability tool shall support both high-end and low-end use of traceability.

In subsection 3.3.1 we saw that Ramesh and Jarke [9] separates trace users into two groups; low-end and high-end. The authors furthermore define one reference model for each, with different level of details. We believe a traceability tool should be general enough to be used by both high-end and low-end users. This means that the traceability user may use a traceability approach suitable for his/her needs, but at the same time that traceability information may be interchanged between users regardless of the level of details used to capture traces. This also means that details may be added at later stages by different kind of analysis.

Tool Requirement 9.1

The traceability tool shall make it possible to capture traces at various levels of details, depending on what information is available at any given time.

It might not be possible or even desirable, to capture all the semantics that may be required at creation time. This depends on what information is available, and what effort is required to capture such information. This may especially be the case when trace information is generated automatically. It should therefore be possible to add semantics at later stages, either manually or through automatic analysis.

Tool Requirement 9.2

The traceability tool shall make it possible to trace an artefact through its whole lifetime.

In order to support high-end traceability, it must be possible to trace the whole lifetime of an artefact. This brings forth the following sub-requirements.

Tool Requirement 9.2.1

Each artefact must be uniquely identified.

In order to be able to trace the whole lifetime of an artefact it must be possible to uniquely identify an artefact across space and time.

Tool Requirement 9.2.2

The traceability tool must support different versions of an artefact.

In order to keep trace the evolution of an artefact, it must be possible to keep information regarding all versions of an artefact that has ever existed. E.g. if a model artefact is used as the source of a transformation and later changed, then it is the old version of the artefact that is related to the target of the transformation, not the one that has been changed. This information may have great impact on to what degree the development process may be analysed.

4.3 Introduction to Technology

This section briefly presents the technology foundation of the traceability tool. We do not discuss the technology in detail, but present the information that is of interest to understand the features of traceability tool.

4.3.1 Eclipse Modelling Framework (EMF)

One of the requirements for the traceability tool (Tool Requirement 6) was that it shall be implemented as an Eclipse Modelling Framework (EMF) [12] plug-in. This is an open source project, with growing popularity, which provides a modelling framework and code generation facility for building tools and other applications based on a structured data models described in XML Metadata Interchange (XMI) [39].

EMF consists of three fundamental pieces [12]:

- **EMF** – the core EMF framework includes the ECore meta-metamodel, which allows definition of metamodels. These metamodels can be used to generate Java Classes, including APIs for creating and retrieving model artefacts programmatically.

- **EMF.Edit** – the EMF.Edit framework allows generation of content and label providers and other convenience classes that allows EMF models to be displayed using JFace viewers and property sheets, in addition to command implementation classes for building editors that support fully automatic undo and redo.

- **EMF.Codegen** – the EMF.Codegen framework allows generation of Java classes that uses the Classes generated by the core EMF framework and the EMF.Edit framework to build a complete editor for an EMF model.

4.3.2 Graphical Editing Framework (GEF)

The Graphical Editing Framework (GEF) [40] provides functionality that makes it possible to create a rich graphical editor from existing application model.

4.3.3 Graphical Modelling Framework (GMF)

The Graphical Modelling Framework (GMF) [41] provide code generation functionalities that makes it possible to generate a complete graphical modelling editor for EMF models. This is achieved by combining the generated EMF classes with GEF features.

4.4 Design

The main goal of the traceability tool is to support generic definition of traceability types, and to allow external plug-ins to use these traceability types to populate the trace repository. This approach is indeed very similar to the traceability metamodel and system solution [3, 10] discussed in subsection 3.3.6. As there is no reference to an implementation of this solution, we will develop a traceability tool that use some of the ideas from this work, but we have made a few changes to the metamodel to allow more dynamic models of traceability types to be defined. These changes include:

- Allow multiple trace type models to be used on the same repository, hence allowing combinations of trace type models to be combined and used together on the same trace repository.
- We do not use the concept of *artefact trace* as we believe *traceable artefacts* and *trace links* will be sufficient in our context.
- Allowing definition of *extension types* which may be used to extend *traceable artefacts* and *trace links* dynamically. I.e. extensions are made on the model level (M1) rather than on the metamodel level (M2).
- Allowing compositions of trace links and traceable artefacts to provide a more expressive language.

An overview of the design of the traceability tool is shown in Figure 17. The tool is based on two metamodels described in the ECore meta-metamodel language;

1. *traceTypeLib.ecore* – the traceability type definition model.
2. *traceRepository.ecore* – the trace repository model

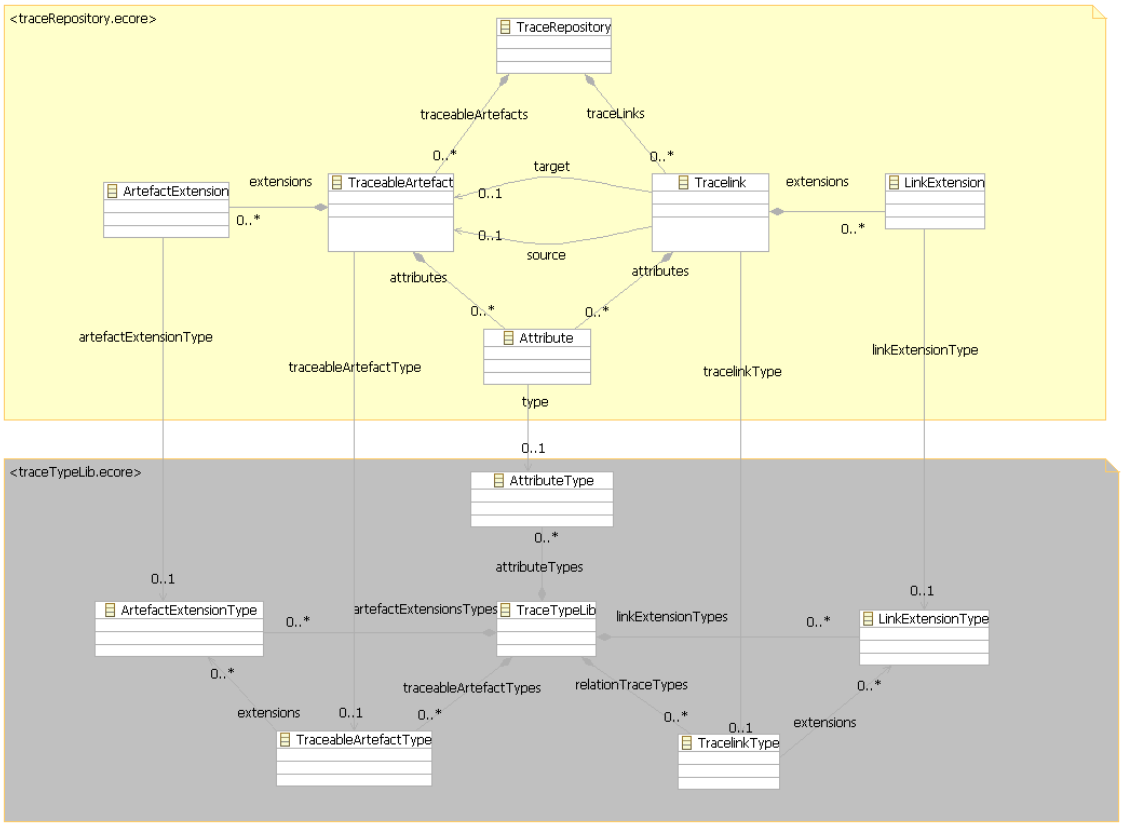


Figure 17: An overview of the traceability tool

The traceability tool is based on the idea that artefacts that shall be traced are represented as *TraceableArtefacts*. These *TraceableArtefacts* are abstractions of actual artefacts, and allow actual model artefacts (e.g. model elements or text-files) to be represented in a trace repository. *TraceLinks* can then be created between *TraceableArtefacts* in the trace repository without polluting the actual artefacts. This abstraction also allows *TraceLinks* to be created between artefacts it would have been difficult to link in another way (e.g. text-files). The *TraceableArtefacts* and *TraceLinks* may also be extended with *ArtefactExtensions* and *LinkExtensions* to provide additional semantics (a bit like UML stereotypes). This concept allows dynamic extension of *TraceableArtefacts* and *TraceLinks* without modification s to the metamodel.

Simple *TraceableArtefacts* and *TraceLinks* do however not offer much semantics on their own, except that there is a relationship between artefacts. The strength of the traceability tool is that all the components are typed with types that can be customized in another model. An illustration of this is shown in Figure 17. This allows different libraries of traceability types to be created, and used to populate trace repositories. These libraries may be used in combination and may be linked to each other.

This means that we are able to create fully generic models describing the trace types (i.e. the information to be traced) that comprises libraries of trace types, and then use these libraries to generated actual traces.

The purpose of the traceability tool is to allow developers to define their own traceability type libraries, and use these libraries to create traces programmatically through a java interface.

The traceability tool is not a complete traceability solution, but provides the functionality to define and create traces. The idea is that external tools and plug-ins may use the traceability tool to generate traces. How and when traceability information is generated is not governed by the tool.

4.4.1 The Metamodels

The traceability tool uses the EMF to define the metamodels, and to generate APIs and editors. The metamodels are described in the following subsections.

4.4.1.1 TraceTypeLib

The trace *typeTypeLib* metamodel defines the language used to describe a *traceTypeLib* model. This model is used to define libraries of traceability types. The complete model (except the linkage to the *traceRepository* metamodel) is shown in. It is explained in detail in the following subsections.

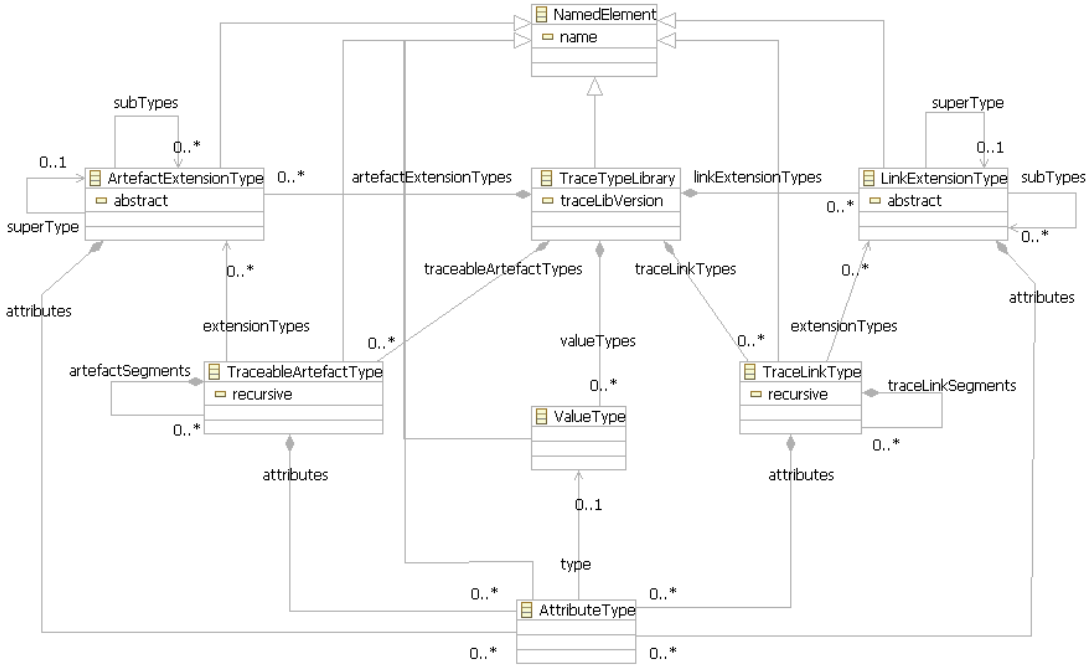


Figure 18: The TraceTypeLib metamodel

NamedElement

This class is a super class for all classes in the model.

Attributes:

- ***name : string*** – a name that works as an identifier for the type.

TraceTypeLibrary

This is the top-level element of the model and thus contains all the other elements directly, or through its contained elements.

Attributes:

- ***traceLibVersion* : string** – the version number of the library.

TraceableArtefactType

This element defines a type that may be applied to a *TraceableArtefact*.

Attributes:

- ***recursive : boolean*** – specifying whether a *TraceableArtefact* of this *TraceableArtefactType* can contain a *TraceableArtefact* of the same type. E.g. a model artefact that may contain other model artefacts. This allows the capturing of detailed information, without making the *TraceableArtefactType* too complex. The default value of this attribute is false.

Relationships:

- ***artefactSegments*** – contains all the *TraceableArtefactType*s that are contained by this *TraceableArtefact*. This makes it possible to create artefact compositions.
- ***attributes –*** contains all the *AttributeType*s of this *TraceableArtefactType*. Attributes of these types may be contained by a *TraceableArtefact* of this type.
- ***extensions*** – specifies which *ArtefactExtensionType*s that may be contained by a *TraceableArtefact* of this type.

TraceLinkType

A *TraceLinkType* defines a type that may be applied by a *TraceLink*.

Attributes:

- ***recursive : boolean*** – a value specifying whether a *TraceLink* of this type may contain a *TraceLink* of the same type or not. This allows the capturing of detailed information, without making the *TraceLinkType* too complex. The default value of this attribute is false.

Relationships:

- ***extensions*** – a set of references specifying which *LinkExtensionTypes* could be applied by this *TraceLinkType*.
- ***traceLinkSegments*** – contains all the *TraceLinkType*s that are contained by this *TraceLinkType*, e.g. a *TraceLinkType* 'Transformation' may contain a *TraceLinkType* 'MappingRule'. This makes it possible to create link compositions.
- ***attributes –*** contains all the *AttributeType*s of this *TraceLinkType*. *Attribute*s of these types may be contained by *TraceLink* of this *TraceLinkType*.

ArtefactExtensionType

This element defines extension types which may be applied by one or more *TraceableArtefactType*s. Artefact extensions are used to add additional information to the *TraceableArtefact*s in their basic form. The intended use is that a *TraceableArtefactType* may have several extensions that each has several sub extensions, thus working more like an enumeration.

Attributes:

- ***abstract : boolean*** – value that specifies whether the *ArtefactExtensionType* is abstract or not. An abstract extension-type cannot be assigned to a *TraceableArtefact*, but its subtype can.

Relationships:

- ***superType*** – a link to the supertype of this *ArtefactExtensionType*.
- ***Subtypes*** - links to the subtypes of this *ArtefactExtensionType*.

LinkExtensionType

This element defines extension types which may be applied by one or more *TraceLinkType*s. Link extensions are used to add additional semantics to the trace link in its basic form. The intended use is that a *TraceLinkType* may have several extensions that each has several sub extensions, thus working more like an enumeration. E.g. a *TraceLinkType* may have a link to the *LinkExtensionType* 'CreationType' (abstract) that is subtyped by the *LinkExtensionType*s 'Manual' and 'Automatic', and maybe another link to the *LinkExtensionType* 'State'(abstract) that is subtyped by *LinkExtensionTypes* 'Valid' and 'Violated'. Thus, making it possible to capture fine-grained information regarding a *TraceLink*.

Attributes:

- ***abstract : boolean*** – value that specifies whether the *LinkExtensionType* is abstract or not. An abstract extension-type cannot be assigned to a *TraceLink*, but its subtype can.

Relationships:

- ***superType*** – a link to the supertype of this *LinkExtensionType*.
- ***Subtypes*** - links to the subtypes of this *LinkExtensionType*.

ValueType

This simple element is used to define a specific simple type (e.g. String or Integer) that can be used by the attributes in this library.

AttributeType

This element defines the *AttributeType*s used by the other elements.

Relationships:

The *type* association specifies what type of value that may be contained by *Attribute*s of this *AttributeType*.

4.4.1.2 TraceRepository

The *TraceRepository* metamodel describes the actual traces that will be stored. The elements of this model all have a *type* reference to its corresponding element in the *traceTypeLib* metamodel.

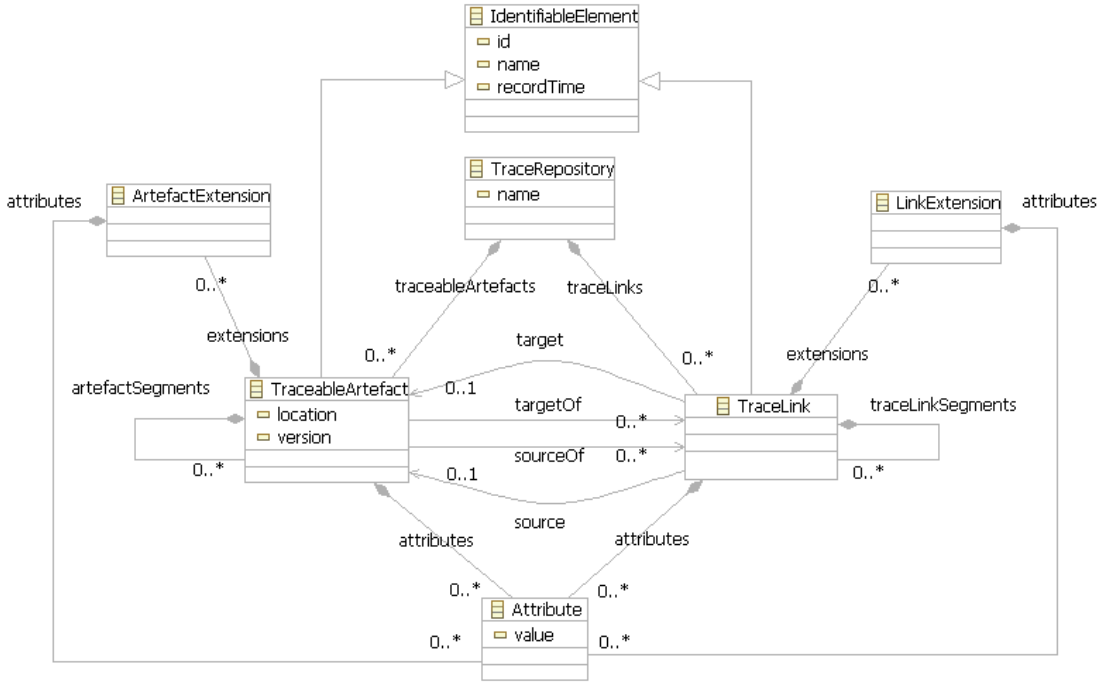


Figure 19: The TraceRepository metamodel

IdentifiableElement

This element is the supertype of *TraceableArtefact* and *TraceLink*, and contains properties that are common to both.

Attributes:

- ***id : string*** – a unique identifier.
- ***name : string*** – the name that will be shown in the traceRepository editor. Acts as a kind of visual identifier.
- ***recordTime : string*** – the time on which the element was recorded.

TraceRepository

The top-level element of the *traceRepository* model. This element contains all the other elements directly or indirectly.

Attributes:

- ***name : string*** – the name of the repository.

Relationships:

- ***traceableArtefacts*** – contains all the *TraceableArtefacts* of the repository model.
- ***traceLinks*** – contains all the *TraceLinks* of the repository.

TraceableArtefact

A *TraceableArtefact* is a representation of an actual artefact in the repository.

Attributes:

- ***location : string*** – the location of the actual artefact that is represented by this *TraceableArtefact*. Typically contains the URI (global or local) of the actual artefact.

- ***version : string*** – the version number of the artefact. Used to separate between different versions of the same artefact (e.g. when an artefact is changed a new version of the artefact may be created).

Relationships:

- ***traceableArtefactType*** – a reference to the *TraceableArtefactType* of the *TraceableArtefact*. *not shown in Figure 19.
- ***artefactSegments*** – contains all the *TraceableArtefacts* contained by this *TraceableArtefact*. May contain *TraceableArtefacts* of the types defined for its *TraceableArtefactType*.
- ***extension*** – contains all the *ArtefactExtensions* applied to this *TraceableArtefact*. May contain *ArtefactExtension*s of the types defined for its *TraceableArtefactType*.
- ***attributes*** – contains all the *Attribute*s of this *TraceableArtefact*. May contain *Attributes* of the types defined for its *TraceableArtefactType*.
- ***targetOf*** – a set of references to all the *TraceLink*s having this artefact as target. This association is an EOposite of *TraceLink.target*, meaning that the references are automatically maintained when the *TraceableArtefact* is made the target of a *TraceLink*. These references are used for navigation.
- ***sourceOf*** – a set of references to all the *TraceLink*s having this artefact as source. This association is an EOposite of *TraceLink.soure*, meaning that the references are automatically maintained when the *TraceableArtefact* is made the target of a *TraceLink*. These references are used for navigation.

TraceLink

A *TraceLink* is used to link TraceableArtefacts.

Relationships:

- ***traceLinktType*** – a reference to the *TraceLinkType* of the *TraceLink*. *not shown in Figure 19.
- ***linkSegments*** – contains all the *TraceLinks* contained by this *TraceLink*. May contain *TraceLinks* of the types defined for its *TraceLinkType*.
- ***extension*** – contains all the *LinkExtensions* applied to this *TraceLinkt*. May contain *LinkExtension*s of the types defined for its *TraceLinkType*.
- ***attributes*** – contains all the *Attribute*s of this *TraceLink*. May contain *Attributes* of the types defined for its *TraceLinkType*.
- ***source*** – a reference to the source of the link.
- ***target*** – a reference to the target of the link.

ArtefactExtension

An extension that is applied to a *TraceableArtefact*.

Relationships:

- ***artefactExtensionType*** – the type of the *ArtefactExtension*. *not shown in Figure 19.
- ***attributes*** – contains all the *Attribute*s of this *ArtefactExtension*. May contain *Attributes* of the types defined for its *ArtefactExtensionType*.

LinkExtension

An extension that is applied to a *TraceLink*.

Relationships:

- ***linkExtensionType*** – the type of the *LinkExtension*. *not shown in Figure 19.
- ***attributes*** – contains all the *Attributes* of this *LinkExtension*. May contain *Attributes* of the types defined for its *LinkExtensionType*.

Attribute

An Attribute may contain a *value* of the type defined for its *attributeType*.

4.5 The GMF Editor

To make it possible to both model and view trace type libraries in an easy manner, a GMF editor based on the tr*aceTypeLib.ecore* metamodel has been created. This editor is shown in Figure 20.

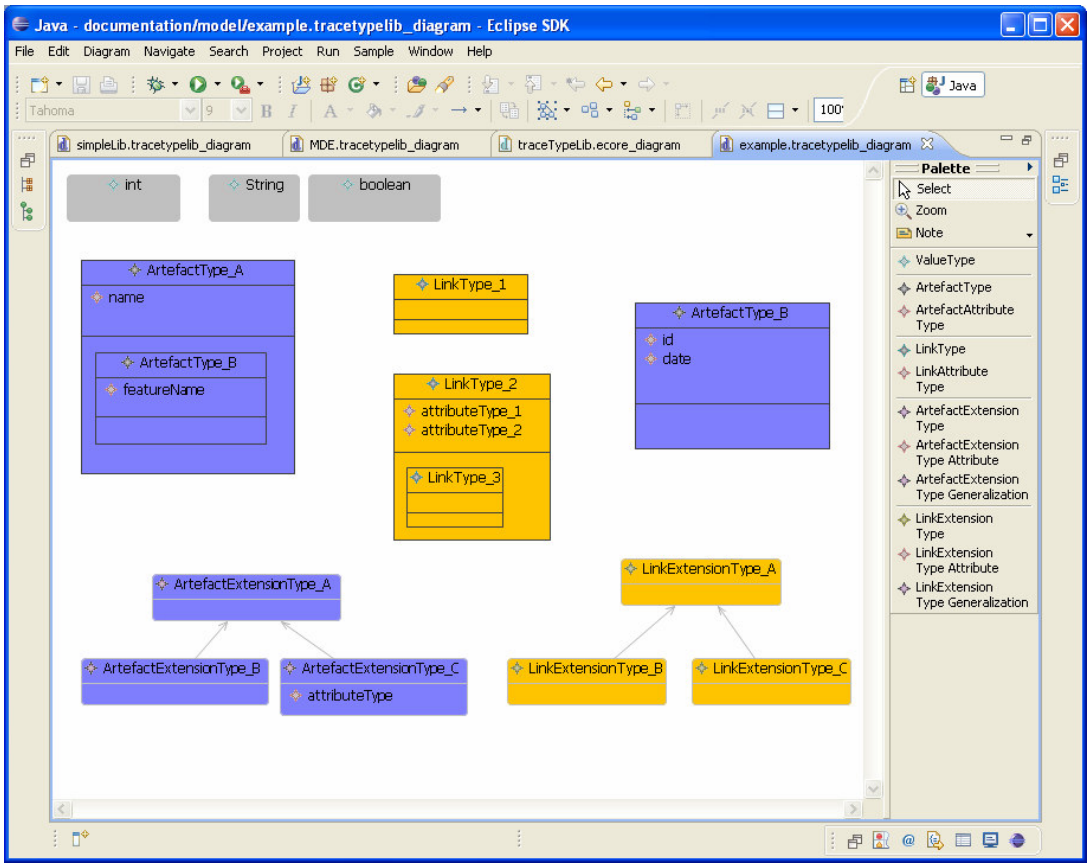


Figure 20: The GMF editor for the traceTypeLib metamodel

The GMF framework makes it possible to create a concrete syntax for EMF models. For the *traceTypeLibModel*, the syntax is as follows:

- ***ValueType*** is shown as a gray rounded rectangle.
- ***TraceableArtefactType*** is shown as a blue rectangle.
    - The upper compartment of the rectangle contains the *AttributeTypes* specified for the *TraceableArtefactType*.

- The lower compartment of the rectangle may contain *TraceableArtefactType*s specified for the *TraceableArtefactType*. I.e. to create compositions.

- ***TraceLinkType*** is shown as a orange rectangle
  - The upper compartment of the rectangle contains the *AttributeType*s specified for the *TraceLinktType*.
  - The lower compartment of the rectangle may contain *TraceLinkType*s specified for the *TraceLinkType*. I.e. to create compositions.
- ***ArtefactExtensionType*** is shown as a blue rounded rectangle.
  - The compartment of the rectangle contains the *AttributeType*s specified for the *ArtefactExtensionType*.
  - The ***supertype*** association is shown as an association pointing at the supertype.
- ***LinkExtensionType*** is shown as an orange rounded rectangle.
  - The compartment of the rectangle contains the *AttributeType*s specified for the *LinkExtensionType*.
  - The ***supertype*** association is shown as an association pointing at the supertype.

4.6 The TraceRepository Editor

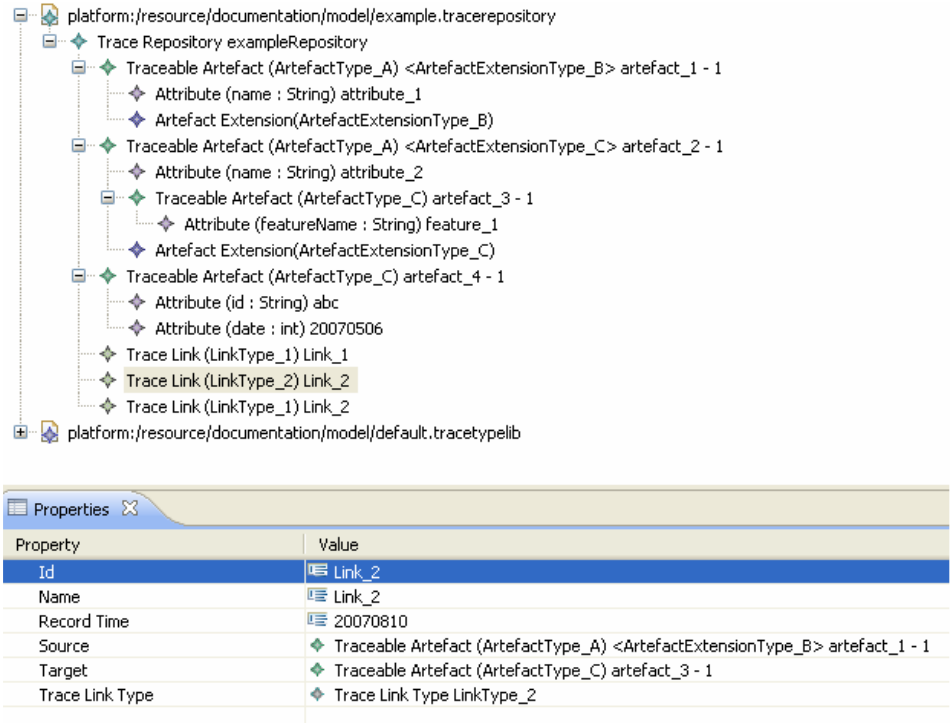


Figure 21: The TraceRepository editor

The *traceRepository* editor is tree based editor generated using the EMF facilities, supporting editing and browsing of *traceRepository* models. In Figure 21 the editor shows a *traceRepository* model using the types defined in Figure 20.

- The type of *TraceableArtefact*s and *TraceLink*s is shown as "(*type*)".
- The type of *Attribute*s is shown as "(*type* : *type.type*)".
- The extensions applied by *TraecableArtefacts* or *TraceLinks* are shown as "<*extension*>".

4.7 The TraceNavigator View

The *TraceNavigator* view is a tree viewer that allows trace information to be navigated and visualised. While the *TraceRepository* editor allows the traceability information is a view of the actual model, the *TraceNavigator* shows the traceability information in a way that is easier to view for a human being, by showing the trace links between traceable artefacts visually. The elements (*IdentifiableElement*) shown in the *TraceNavigator* may however not be unique, as the same traceable artefacts may be the source or target of multiple links, hence shown in multiple branches of the tree.

Furthermore, the *TraceNavigator* view has two different modes, both initiated by right-clicking on a *TraceableArtefact* in the *traceRepository* editor and selecting the menu choice with the respective name:

1. ***Show Descendants*** – allows the trace links that can be reached by following the links of the *sourceOf* set of the selected artefact to be navigated. This mode also shows aggregation, meaning that containment relationships between artefacts can be navigated. Traceable artefacts are shown as objects, while trace links are shown as arrows pointing to the right.

2. ***Show Predecessors*** – allows the trace links that can be reached by following the links of the targetOf set of the selected artefact to be navigated. This mode does not show aggregation relationships. Traceable artefacts are shown as objects, while trace links are shown as arrows pointing to the left.

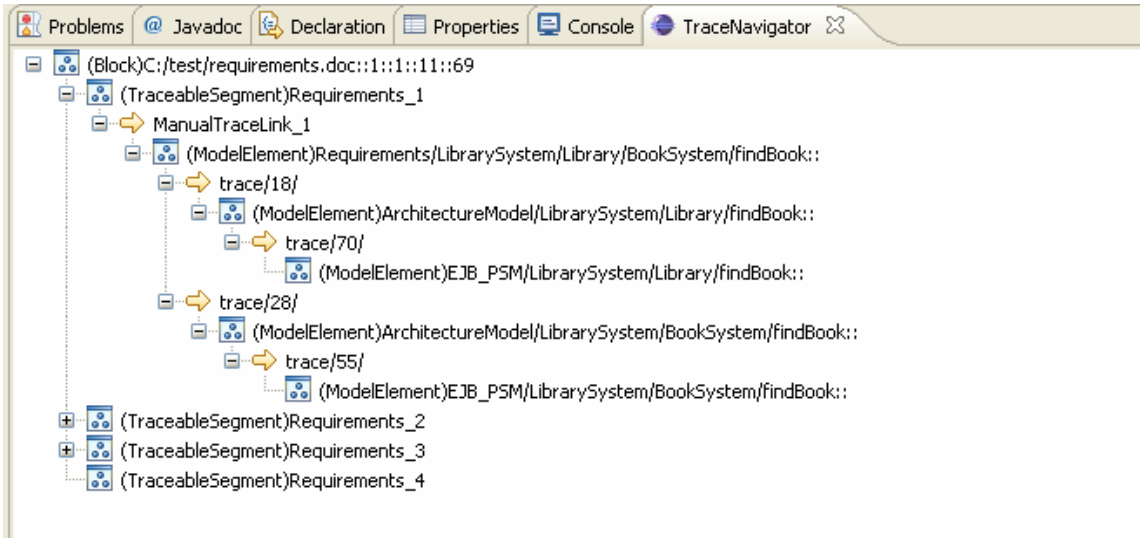


Figure 22: The TraceNavigator view showing descendants

In the figure above the *TraceNavigator* view shows the result of selecting *Show descendants* when right-clicking the traceable artefact representing the requirements document (using a *traceTypeLib* model describing the traceability metamodel of MOFScript) of the library example. We see that this traceable artefact contains a single traceable artefact of type *Block* which in turn contains four traceable artefacts of type *TraceableSegment*. The traceable artefact with name 'Requirements_1' can be traced down to two elements contained in the EJB-PSM (the operation findBook in classes Library and BookSystem).

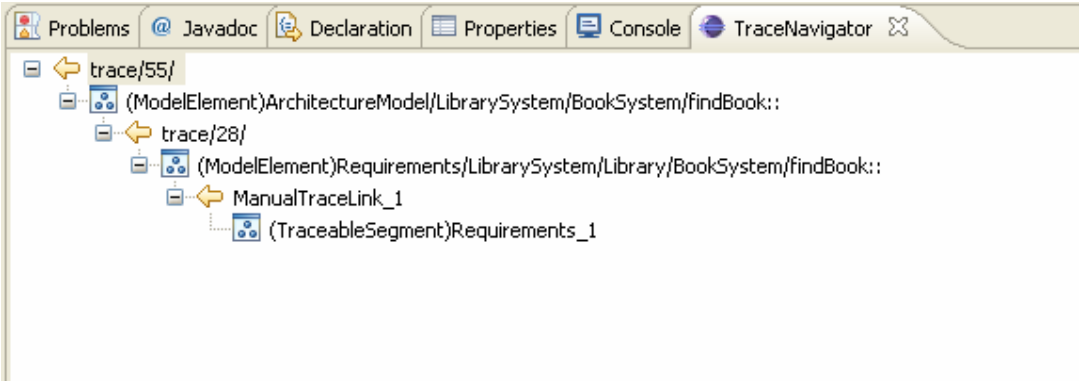


Figure 23: The TraceNavigator view showing predecessors

In the above Figure, 'Show predecessors' have been selected for the traceable artefact representing operation findBook in the class BookSystem of the EJB-PSM. This allows us to trace navigate the traces all the way back to a requirement in the requirements document. The same thing is actually shown in Figure 22 (the lower expanded branch), starting at the requirement. The different modes hence show the information in reverse direction of each other.

4.8 Java Interfaces

The java interface allows all model elements in the *traceTypeLib* model and the *traceRepository* to be programmatically created, through a factory class. This interface is however a bit cumbersome to use as it works on the meta-level. The names of classes and attributes that shall be made must therefore be past as strings.

4.9 The Code Generator

To ease development, the traceability allows code to be generated from the *traceTypeLib* models; for each *TraceLinkType* and *TraceableArtefactType*, a Java Class is generated. Each class contains hard coded method calls, using the names from the *traceTypeLib* model that are wrapped in methods. E.g. the constructor of the generated class for 'ArtefactType_A' in Figure 20 contains hard coded method calls that create an instance of a *TraceableArtefact* in the trace repository with the type 'ArtefactType_A'. A pointer to the created instance is stored in the Class. In this way it is possible to program with traceability types just like any Java class. The classes must however be created through a factory class to ensure that all classes uses the same repository and library.

At current stage, the code generation facilities are not supported by the traceability tool, and must therefore be run from the MOFScript source code, using the MOFScript facilities with a *traceTypeLib* model as input.

4.10 Validation

Our assumption regarding the traceability tool is that is will be suitable for defining and using semantically rich traceability classification schemes (H2). We can not prove that H2 is true,

but we may strengthen the assumption by putting the tool to test. This will be performed in to stages:

1) Using the traceability tool to define the traceability classification scheme. If the tool allows us to define a suitable classification scheme, our assumption is strengthened.
2) Integrating the tool with external plug-ins, i.e. using the traceability tool to generate traces for another eclipse plug-in.

If the traceability tool allows these two tasks to be performed, our assumptions are strengthened.

4.11 Summary

In this chapter, we have presented a traceability tool based on two metamodels:

1. The ***traceTypeLib*** metamodel defines a language used to define libraries comprised of traceability types with a set of attribute types for each type.
2. The ***traceRepository*** metamodel defines a language for describing traceability information of the types defined by a *traceTypeLibModel*.

Utilising the EMF facilities, these metamodels were used to generate classes and interfaces allowing instances of these models to be created programmatically. We also presented a GMF editor for the *traceTypeLib* metamodel and an EMF editor supporting browsing and editing of *traceRepository* models in addition to the *TraceNavigator* with support for navigating the trace links.

5 Proposal for Traceability Classification Scheme

5.1 Introduction

In this chapter we present a proposal for a classification scheme for traceability in MDE. The proposal will not be a complete classification scheme, but will present a scheme that covers the basic needs for traceability in MDE in the context of the library example (Appendix A).

We first present the requirements for such classification scheme in section 5.2. In section 5.3 we present some challenges associated with such a classification scheme, and continue by presenting the classification scheme is presented in detail in section 5.4. In this section we describe how the proposed classification scheme is meant to support the traceability-scenarios, and continue with a summary and discussion regarding the problems associated with capturing and maintaining the proposed information in section 5.7.

5.2 Trace Classification Scheme Requirements

CS Requirement 1

The classification scheme shall support traceability in all stages of the development process of the library system (Appendix C).

In Section 3.3 traceability in general was identified as "*The ability to trace (identify and measure) all the stages that led to a particular point in a process that consists of a chain of interrelated events*". This is the essence of traceability and also the most basic requirement for a traceability classification scheme.

CS Requirement 2

The traceability classification scheme shall allow semantically rich traceability information to be captured.

This is what we believe an is going to prove.

CS Requirement 3

The traceability classification scheme shall allow any artefact involved in the development process of the library system (Appendix C) to be traced.

As elements used in a software development process may be of any kind, not just model artefacts, it must be possible to maintain traceability between any artefact, including model artefacts, pieces of program code, text in a requirements document etc.

CS Requirement 4

It shall be possible to identify what information is represented.

The semantics of the artefacts that is represented in a trace model is not necessary easy to deduce. Is it for example a requirement, rationale, test class, or design class? The traceability information must contain sufficient semantics to separate such artefacts. This information may be important in order to support certain kinds of analysis.

CS Requirement 5

It shall be possible to identify how the traced artefacts of the library system are represented.

As the traceability model may contain traces of artefacts of many different types, e.g. model artefacts, java code, or requirements, it is necessary to be able to separate between these different types. This information may be necessary to locate the actual artefacts, to maintain the information, and to analyse the information.

CS Requirement 6

It shall be possible to identify how an artefact was created.

Artefacts may be created in many different ways, e.g. by a person using a tool, or by a transformation engine. This information is essential for the analysing the traceability information correctly.

5.3 Challenges

There are several challenges associated with creating a classification scheme for traceability in MDE. Some of these are discussed below.

5.3.1 Traceability Strategy

What traceability types to use in a specific project may depend to a large degree on the traceability strategy that is used e.g. is *low-end* or *high-end* traceability going to be used. The discussion in the following subsections may help the process of defining such a strategy.

5.3.2 Tracing Any Artefact Involved in the Development Process

A classification scheme supporting traceability in MDE must support traceability for a variety of different artefacts that may be involved in the development process. Such artefacts may include; requirements document, pieces of code, rationale documents and models. How these artefacts may be represented depends on the language in which the traceability types are described.

The language that is to be used to describe the traceability types in this thesis is described by the *tracTypeLib* metamodel presented in the previous chapter. This language uses the concept of *TraceableArtefactTyp*e to describe artefact types. Hence, the challenge is to provide a set of

TraceableArtefactTypes that is capable of describing the artefacts involved in the process at the right level of abstraction, and in a way that allows trace links to be defined between relevant artefacts in a meaningful way.

5.3.3 Uniquely Identifying Artefacts

Because the *TraceableArtefact*s captured in the *traceRepository* model do not have any direct linkage to the actual artefacts they are used to represent, it is crucial that the *TraceableArtefactType* describing their type contains attributes that allows the artefact to be uniquely identified (i.e. in terms of locating them) in a way that makes it possible to locate the actual artefact. The unique identifier does however not mean that the artefact must be universally uniquely identified; this depends on how the information is meant to be used (i.e. in some cases it may be sufficient that the information is uniquely identified within a project or workspace).

5.3.4 Classification of Traceability Information

As identified earlier there are many challenges associated with classifying traceability information. Semantically rich trace information is essential if the information is to be used for anything more than identifying that *a relationship exists between artefact A and B*. A problem with semantically rich traceability information however is that it requires more effort to capture and maintain, and may make it more difficult to automate the process. The challenge will therefore be to find a way to classify the traceability information in a way that supports typical use-scenarios of traceability in MDE, but at the same time does not make the process of maintaining this information to high.

5.4 Classification Scheme

Summarising the challenges discussed in section 5.3 we can identify some key aspects that should be considered when creating a traceability classification scheme, utilising the features of the traceability tool:

1. Decide what kinds of artefacts that should be traced, and find an appropriate abstraction level (i.e. create a set of *TraceableArtefactType*s) that allows the relevant parts of an artefact to be traced.
2. Decide how the different artefacts can be uniquely identified, in terms of uniquely locating them (i.e. add *AttributeType*s to store this information to the *TraceableArtefactType*s).
3. Find an appropriate set of fine-grained extensions that may be used to add semantics to the traced artefacts (i.e. *ArtefactExtensionType*s that may be assigned to a specific *TraceableArtefactType*).
4. Find an appropriate set of *base* trace link classes that may be used to describe the basic semantics of a relationship between two artefacts (i.e. add *TraceLinkType*s).
5. Specify a set of fine-grained extensions that may be assigned to certain trace link classes for added semantics (i.e. add *TraceLinkExtensionType*s).

Using this template should make the process of finding an appropriate classification scheme somewhat easier, as serves as a separation of concerns. We will therefore follow these five steps through this process.

5.4.1 Classification of Basic Traceable Artefact Types

Identifying which classes of artefacts that needs to be traced may, as previously mentioned, depend to a large degree on the domain and/or project. We will therefore focus on the most basic artefacts involved in an MDE development process, i.e. artefacts that may be expected to find in most projects. This will be done based on the running library example, but additional information will be added when found appropriate.

The library example (Appendix A) is composed of the following artefacts; a requirements document, a rationale document, a Use Case model, a PIM, a PSM, and the Java code. However; we also know that the PIM was generated from the Use Case model, the PSM from the PIM, and the java code from the PSM. Hence the transformation has obviously played a role in the process. The transformations (the execution) can however not be said to be an artefact – it is not a physical artefact – the mappings (the set of rules that describes a transformation) on the other hand must be said to be a physical artefact. The mappings must therefore be considered to be an artefact of interest to the MDE process – it is responsible for the output of the execution of a transformation, and hence explains the logic behind the transformations.

Summarizing the discussion in this subsection we end up with the following kinds of artefacts identified by a set of properties:

1. A requirements document.
    a. A text-file containing a set of informal words.
    b. Sets of words or sentences specify requirements that must be satisfied by the system.
2. A rationale document.
    a. A text-file containing a set of informal words.
    b. Sets of words or sentences describe the rationale behind the architecture model.
3. A Use Case model.
    a. A model containing a set of components with Use Cases having a relationship to an actor.
4. A PIM.
    a. A model containing a set of Packages, Classes, Operations and Associations.
5. A PSM.
    a. A model containing a set of Packages, Classes Operations and Associations.
6. The implementation code
    a. A set of text-files containing java code.
        i. Each text-file is comprised of formally described sentences that are comprised of keywords and names.
7. The UseCase2Architecture mapping
    a. A text-file (possibly set of text-files) containing transformation code.
        i. Each text-file is comprised of formally described sentences that are comprised of keywords and names.
8. The PIM2EJB-PSM mapping
    a. A text-file (possibly set of text-files) containing transformation code.
        i. Each text-file is comprised of formally described sentences that are comprised of keywords and names.
9. The EJB-PSM2Java mapping
    a. A text-file (possibly set of text-files) containing transformation code.

i. Each text-file is comprised of formally described sentences that are comprised of keywords and names.

This information could be classified in many different ways, however by analysing the properties of each kind of artefact used in the library example we end up with two basic types of artefacts that represent physical artefacts:

1. Text-files containing some kind of text, where pieces of the text represent information that has a value for the MDE process.
2. Formal models.

These two *base classes* is however stripped of all semantic sugar – the only semantics offered is regarding how an artefact is physically stored. Separating between the physical artefacts and the semantics of the artefacts (i.e. whether it is a requirement or implementation code) does however have some advantages. First of all it is worth to notice that in many cases a semantic property may not be specific for only one of the two base types, i.e. the requirements or the mappings may in theory just as well be described by a model. Furthermore the requirements or a mapping could in theory be generated from a model (e.g. a mapping could be generated using a higher-order transformation in ATL (as a model) or by generating textual code using MOFScript). If all the artefacts in theory may be the source or target of a transformation, it could simplify traceability in transformations a great deal to treat all model artefacts as the same basic thing, and all text-files as the same basic thing. This means that transformation engines do not have to deal with multiple different artefacts. At the same time specific semantic properties could be used both for a textual artefact or a model element.

Through the discussion in section 3.3 we saw that the metamodels of Figure 10 and Figure 12 could be used as a language to describe traceability information in transformations from model to model and from model to text. In fact, the features of the metamodel used to capture traces from model to model (Figure 10) could be applied to the metamodel used by MOFScript to capture model to text (Figure 12). The only required change would be to add functionality to create *Trace*s between two *ModelElementRef*s, e.g. by adding a new class 'TraceableArtefact' which would serve as a super type for both *TraceableSegment* and *ModelElementRef*. The source and target associations of a *Trace* could both be set to the new super type.

As the functionality provided by the metamodel in Figure 12 already has been shown to be sufficient to capture the required information regarding both model artefacts and textual artefacts in a way that allow traceability links to be created between them, we will use the same approach with some modifications, to describe the base classes of our classification scheme. Describing such classification types is quite easy using the GMF editor of the traceability tool.

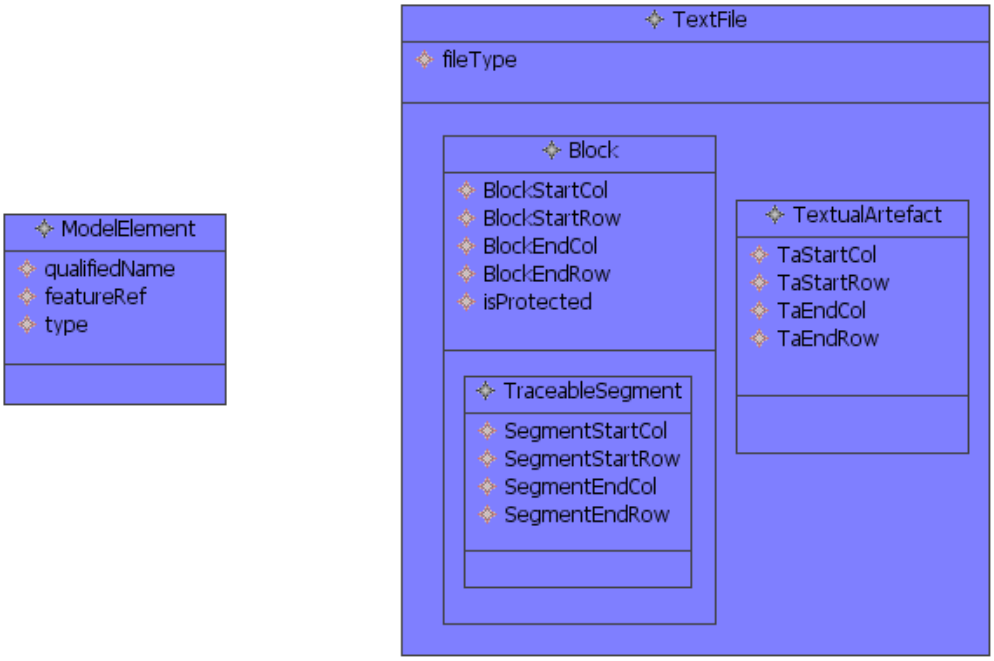


Figure 24: The basic traceable artefact types (inspired by [24])

Figure 24 shows the basic traceable artefact types of the classification scheme. It describes the type ***TextFile***, which may contain one or more *Blocks* and/or one or more ***TextualArtefacts***. The ***TextualArtefact*** is introduced to support explicit trace links, i.e. it is an explicitly defined block which may overlap with ***Block***s, and ***TraceableSegments***. Furthermore, the ***Block*** may contain one or more ***TraceableSegment***s. The semantics of the *TraceableArtefactType*s is explained in Table 4. None of the *TraceableArtefactTypes* are recursive (i.e. a *TraceableArtefact* of one of these types cannot contain another *TraceableArtefact* of the same type). The semantics of the *AttributeType*s is explained in Table 5 - Table 9.

Contained by	Classification	Semantics
TAT_1	***ModelElement* [24]**	Represents a model element
TAT_2	***TextFile* [24]**	Represents text-file
TAT_2.1	***Block* [24]**	Represents a logical block within a text-file. A Block may be protected or unprotected. An unprotected block represent an area within a file that may be changed by a user, typically used to define areas where developers are meant to enter code
TAT_2.2	***TraceableSegment* [24]**	Represents an area within a block that contains a value that is generated from a model, e.g. The name of Class or Operation.
TAT_2.3	***TextualArtefact***	Represents a logical area within a text-file, explicitly defined by a developer using a tool or a transformation language supporting explicit trace link creation. Typically used to define a block of text comprising a requirement, or a logical block of text representing an area within a file that will be the target of a trace link generated by a transformation.

Table 4: TraceableArtefactTypes

With these two basic traceable artefact types we should be able to create trace links between all textual artefacts and model artefacts. This should theoretically make it possible to automatically generate all the artefacts, use the artefacts to generate new artefacts, and at the same time keeping explicit and/or implicit traces between all the stages.

5.4.2 Uniquely Identifying the Traceable Artefacts

Traceability information is of little use if the traceable artefact types do not carry enough information to allow the actual artefact to be located. Due to the generality of the traceability tool, some of this information must be specific to the traceable artefact types. This information is captured by adding attributes to the traceable artefact types. All traceable artefacts do however contain the property *location*. The intended use of this property is to capture the unique location of an artefact, either with a local or a global URI, depending of how the information is meant to be used. Identification of elements further than the URI, e.g. a model artefact within a model or a feature within a model artefact, does however require additional attributes. The location and the identifying attributes provide a unique identification in terms of uniquely locating the actual artefact locally or globally. In Table 5 - Table 9 the identifying attributes of each *TraceableArtefactType* are marked with a '*'.

ModelElement		
AttributeType	**ValueType**	**Semantics**
*qualifiedName**	String	The location of the element within a model, (e.g. '*ModelName/PackageName/ClassName/AttributeName*'). This attribute is used to locate an element within the model it is contained by. (Requires that all traced model elements has a unique name)
*featureRef**	String	The name of a feature of a model artefact (e.g. *type* or *name* of a Class). If the value is an empty string, the *ModelFeature* represent the model artefact itself, else it is a feature within a model artefact.
type	String	The type of the model artefact that is being traced (e.g. '*UML.Class'* or '*UML.Operation*'). Captured for convenience, and makes it possible to reconstruct the artefact through reverse engineering.

Table 5: AttributeTypes of *ModelElement*

The nature of an artefact represented by a traceable artefact of type ***TextFile*** means that it does not require any further identification than that provided by the *location* property of *TraceableArtefact*.

TextFile		
AttributeType	**ValueType**	**Semantics**
fileType	Integer	The type of the file (e.g. Java, doc, or txt) used for convenience.

Table 6: AttributeTypes of *TextFile*

Block		
AttributeType	**ValueType**	**Semantics**
*blockStartCol**	Integer	the start column of the block within the *File*
*blockStartRow**	Integer	the start row of the block within the *File*
*blockEndCol**	Integer	the end column of the block within the *File*
*blok EndCol**	Integer	the end row of the block within the *File*
isProtected	boolean	specifies whether the block is protected or not

Table 7: AttributeTypes of *Block*

TraceableSegment		
AttributeType	**ValueType**	**Semantics**
*segmentStartCol**	Integer	the start column of the traceable segment within the *Block*
*segmentStartRow**	Integer	the start row of the traceable segment within the *Block*
*segmentEndCol**	Integer	the end column of the traceable segment within the *Block*
*segmentEndCol**	Integer	the end row of the traceable segment within the *Block*

Table 8: AttributeTypes of *TraceableSegment*

TextualArtefact		
AttributeType	**ValueType**	**Semantics**
*taStartCol**	Integer	the start column of the textual artefact within the *File*
*taStartRow**	Integer	the start row of the textual artefact within the *File*
*taEndCol**	Integer	the end column of the textual artefact within the *File*
*taEndCol**	Integer	the end row of the textual artefact within the *File*

Table 9: AttributeTypes of *TextualArtefact*

5.4.3 Extending the Basic Traceable Artefact Types

Although the basic traceable artefact types of Figure 24 allows textual artefact and model artefacts involved in the MDE development process to be traced, they do not give much clue regarding what kind of information the actual artefact represents. E.g. it will be difficult to get an understanding of the development process by knowing that there are links between a set of textual artefacts and a set of model artefacts. To allow richer analysis to be performed on the traceability information, one needs to apply richer semantics. There are mainly two ways of adding semantics to the traceable artefacts:

1. Adding semantics to trace links between traceable artefacts, providing semantics by expressing properties regarding the relationship (e.g. there is a Satisfy relationship from artefact A to artefact B, thus; artefact A is a requirement for artefact B).
2. Adding semantics to the traceable artefact it self, by expressing properties regarding the traceable artefact.

In many cases, both approaches may provide the same semantics and functionality. However, the first approach only provides additional semantics in cases were a trace link do exists, and may therefore not be sufficient for all situations. Thus; in order to provide semantics to artefacts regardless of whether or not it is the source or target of a trace link, it might be desirable to add some semantics on the artefact itself.

Adding semantic properties to traceable artefacts is possible using the *ArtefactExtensionType* of the *traceTypeLib* metamodel of the traceability tool. Such extension types specify possible extensions to traceable artefacts of a specific traceable artefact type. I.e. the *ArtefactExtensionTypes* referenced by the *extensionTypes* association of *TraceableArtefactType* specifies possible extensions that may be applied by a *TraceableArtefact* of that type. Using this feature it is hence possible to dynamically extend the artefacts with a specific artefact extension type.

There are many ways of classifying such extension types, and the importance and granularity may vary a great deal between different domains. We will therefore only provide a set of "typical" artefact extension types. In section 3.3 we gave a simple traceability overview, showing traceability across five types of artefacts; *Requirement*s, *Analysis*, *Design*, *Implementation* and *Test*. In addition we might add *Rationale*, *Mapping* and *MappingRule* to cover the different kinds of artefacts in the simple library example. These types should be sufficient for our basic trace type library.

Below is an overview of applying this classification to the artefacts found in the running library example:

- *Requirements* – The requirements document.
- *Analysis* – The use-case model.
- *Design* – The PIM and PSM.
- *Test* – None.
- *Implementation* - The java code.
- *Rationale* – The rationale document.
- *Mapping* – The mappings.
- *MappingRule* – The mapping rules.

These types serve as the artefact extension types in our proposed traceability classification scheme, super typed with the artefact extension type *ArtefactType* (abstract). These artefact extension types are shown in Figure 25. Applying the artefact extension type *ArtefactType* to the *extensionTypes* set of all the traceable artefacts in Figure 24 means that all the artefacts with these types may contain an artefact extension (only one) with one of it subtypes as type. We will however not add the extension types to ***Block***, as this artefact type does not require additional semantics.

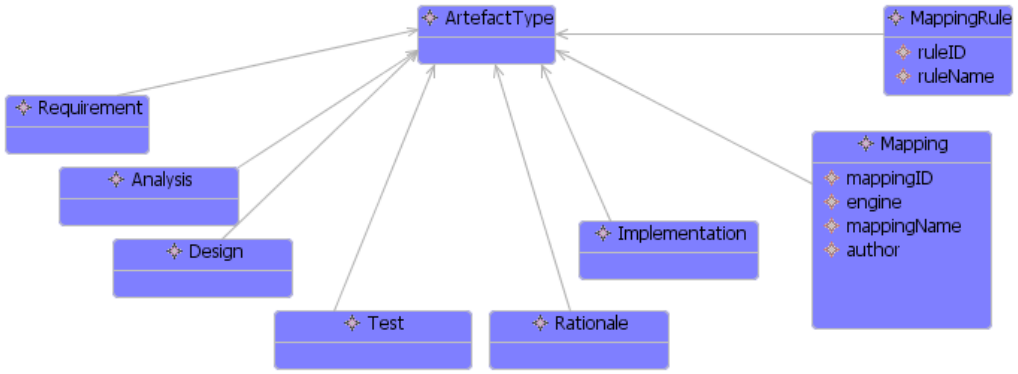


Figure 25: Traceable artefact extension types.

These artefact extension types may be extended to make more fine-grained types. One could e.g. extend *RequirementArtefact* with the subtypes *BusinessRequirement*, *FunctionalRequirement*, and *QualityRequirement*.

	Classification	Semantics	abstract
AET_1	ArtefactType	Semantics regarding to the type of a *TraceableArtefact*.	true
AET_1.1	Requirement	Specifies that the *TraceableArtefact* it is applied to is a *Requirement*.	false
AET_1.2	Analysis	Specifies that the *TraceableArtefact* it is applied to is an *Analysis* artefact.	false
AET_1.3	Design	Specifies that the *TraceableArtefact* it is applied to is a *Design* artefact.	false
AET_1.4	Test	Specifies that the *TraceableArtefact* it is applied to is *Test*.	false
AET_1.5	Rationale	Specifies that the *TraceableArtefact* it is applied to is a *Rationale*.	false
AET_1.6	Implementation	Specifies that the *TraceableArtefact* it is applied to is an *Implementation*.	
AET_1.7	Mapping	Specifies that the *TraceableArtefact* it is applied to is a *Mapping*.	false
AET_1.8	MappingRule	Specifies that the *TraceableArtefact* it is applied to is a *MappingRule*.	false

Table 10: ArtefactExtensionTypes

5.4.4 Classification of Basic Trace Links

In subsection 3.3.7.1 we discussed a number of different trace link classes, describing the relationship between artefacts with different semantics. We did note however that many of these classes overlapped in terms of semantics. Additionally the semantics of a link may be a combination of several classes, e.g. one link may be *manually-verification* while another may be *automatically-verification*. It could therefore be desirable to follow the same approach as was used to classify traceable artefact types when classifying trace links; identify a set of basic trace link types that could be dynamically extended using *LinkExtensionType*s.

In the context of MDE, information that will always be of great significance is how the link between two traceable artefacts was created. A link may exist between two different artefacts for several reasons, including:

- It may be created implicitly through a transformation, to link the input and the output of the transformation.
- It may be created explicitly through a transformation, to explicitly link artefacts from the input model to artefacts of the output model (e.g. the relationship between a UML Class and a Java Class).
- It may be created manually by a developer using a tool, to e.g. specify that there is a relationships between two artefacts, e.g. to specify that there is a link between a *ModelElement* and a *Rationale*.
- It may be created automatically by a tool to specify a relationship between two artefacts that was created automatically by a tool, i.e. without intervention by a person.

Walderhaug et al [10] suggests a classification scheme that separates between automatic and manual trace links, and makes it possible to extend automatic links with the type transformation. It could however be advantages to model links created by transformations in a different way than manual and automatic links due to the amount of links that may be generated during a transformation, and to ease analysis of trace links generated by a transformation.

For these reasons, we have defined transformations as a separate link type along side automatic and manual links. The different link types are discussed in the following subsections. Table 11 shows an overview of the different link types.

	Classification	Semantics	Constraint
TLT_1	***Manual* [10]**	A *TraceLink* created by a human being using a tool (e.g. to explicitly link a requirement to a Design artefact).	
TLT_2	***Automatic* [10]**	A *TraceLink* created automatically by a tool (i.e. not by a transformation engine or by a developer using a tool).	
TLT_3	***Transformation***	A record of a transformation, containing a set of ***MappingRule***s. This *TraceLinkType* shall not be a link between two artefacts, but serves as a container for the ***MappingRule***s that was executed by it.	No *source* and *target* artefacts
TLT_3.1	***MappingRule***	A record of the execution of a mapping rule containing a set of ***FeatureTransformation***s and/or ***ExplicitLink***s. This *TraceLinkType* shall not be a link between two artefacts, but serves as container for the links that are created by a specific rule.	No *source* and *target* artefacts
TLT_3.1	***FeatureTransformation***	A *TraceLink* implicitly created by a transformation to link the source and target artefacts of a transformation.	
TLT_3.1	***ExplicitLink***	A *TraceLink* created by a transformation when a TraceLink between two artefacts involved in a transformation is explicitly defined in the mapping (i.e. defined by the author of the mapping).	

Table 11: TraceLinkTypes

5.4.4.1 Transformations

One of the main issues regarding analysing relationships between artefacts of an MDE process is to identify links that was created as part of a transformation. If this information is not present, it becomes very difficult to analyse the information properly in terms of impact analysis, and general trace inspection. Considering that transformations are one of the main aspects of MDE, and that one of the major applications of traceability in MDE is to capture the relationships created by transformations, it becomes quite obvious that the trace type library must make it possible to capture this information.

How to capture traces of a transformation on the other hand is not that obvious. There are really two artefacts of interest concerning a transformation;

1. The mapping (transformation code or a mapping model), describing the logic of the transformation, which might be of interest as documentation or when new transformations need be performed.
2. The actual transformation, which is the process of creating target artefacts from the source artefacts based on the mapping rules. I.e. the mapping is the “static” rules that describe the actual transformations.

The mapping and mapping rule was already discussed in subsections 5.4.1 and 5.4.3 (a combination of *TraceableArtefact* and *TraceableArtefactExtensions*), but we also need to model the links created by the transformation. An important issue here is to be able find and locate mapping or mapping rules responsible for a transformation or part of a transformation. Another important issue is that a mapping may be responsible for the execution of several mapping rules that are more or less dependent on each other (i.e. the execution of one may include or trigger the execution of another). Similarly the execution of a mapping rule may be responsible for creating several links between traceable artefacts. This information may be of importance when analysing the development process. Hence, it could be reasonable to group trace links generated by the same transformation (a single execution of a mapping) together. This can be achieved by using composite links.

We have noted however that there are two different ways to generate traceability links from a transformation; the links may be *implicitly* or *explicitly* generated. It should be separated between these two kinds of trace link, as implicit links makes it possible to analyse the transformation, i.e. they link input and the output of the transformation. Explicit links however only exists because a developer wanted it to be created, and hence do not necessarily tell exactly what happened during the transformation. Furthermore it is not easy to add semantics to implicit links, as they in most cases are created without the knowledge or interference of the creator of a transformation. Explicit links however are created by a person, and can therefore more easily be extended with semantics. Transformation tools should support both in order to provide rich traceability information. We have therefore added support for both in our classification scheme; ***FeatureTransformation*** to capture implicit links, while ***ExplicitLink*** is used to capture explicit links.

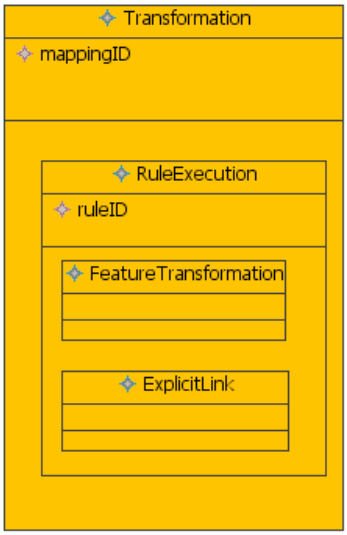


Figure 26: The *Transformation* link composition

As we can see, this link composition actually is a bit more than a link; it is more of a record of the transformation execution, containing links between the input and the output.

5.4.4.2 Manual and Automatic Trace Links

Knowing whether a trace link was manually or automatically created may be important for several reasons. First of all, the creation of a ***Manual*** trace link is not based on any formalised

rules, but exists by the volition of a developer. This means that it might need manual effort to maintain the linkage as the system evolves and the relationships between artefacts Change. Moreover, due to the fact that a ***Manual*** link is not created based on any formal rules, it might need to be treated differently when analysis is performed. Lastly, it might be valuable to store information about the creator of a ***Manual*** link on the link, so that it is possible to find the person responsible.

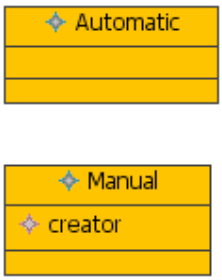


Figure 27: *Manual* and *Automatic* trace links

5.4.5 Extending the Basic TraceLinkTypes

In order to find the right links to use to conduct specific kinds of analysis, it should be possible to see whether a link is *product-related* (i.e. specifies a relationship regarding design objects) or *process-related* (i.e. specifies a relationship regarding the evolution of the system). E.g. when performing a coverage analysis regarding the coverage of a requirement, the implicit links created by a transformation may not be of interest. The explicit links used to specify relationships between a requirement, a design artefact, and an implementation artefact however might be what we are looking for.

5.4.5.1 Product-Related Extension Types

Product-related links are meant to be used to analyse relationships regarding design objects. As the semantics of the artefacts these links provide linkage between are kept on the artefacts themselves, the link does not have to carry semantics regarding whether the source is a requirement for the target etc. this information can be deduced by the semantics of the artefacts provided that the link between them is *product-related*. From the discussion on Figure 9 however, we know that such links may be used for *Validation* and *Verification*. Separating between these two kinds of links could make it easier to find the correct information for a specific analysis, and would probably make it easier to navigate the traceability information for a human. In the library example, we do however have an artefact whose relationship to other artefacts on the "*product-level"* does not fall under the *Validation* or *Verification* category. This artefact is the rationale document. This artefact does not validate or verify the product, but *explains* it (i.e. parts of it).

Knowing the difference between these kinds of relation ships should ease analysis of the traceability information, as it makes it possible to tell what links to follow and what links to not follow for a certain kind of analysis. An algorithm for conducting a coverage analysis does for instance not care about the ***Validation*** or ***Explanation*** dimension, while an algorithm for finding the rationale for a design artefact is only interested in the ***Explanation*** dimension.

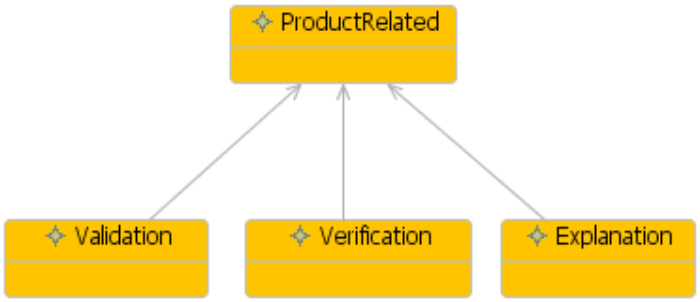


Figure 28: LinkExtensionTypess

Constraints

- Only one subtype may be applied by any link.
- May not be used on the same link as ***Realization*** extension.

	Classification	Semantics	abstract
LET_1	ProductRelated	The semantic relationship between the source and target. This extension type is abstract - only its subclasses may be applied.	true
LET_1.1	Validation	The source is a predecessor in the chain of artefacts comprising the validation dimension in traceability graph.	false
LET_1.2	Verification	The source is a predecessor in the chain of artefacts comprising the verification dimension in traceability graph.	false
LET_1.3	Explanation	The source is rationale for the target. I.e. a rationale regarding the product.	false
LET_2	Realization	The target is a realization of the source, but there has not been a transformation involved.	

Table 12: LinkExtensionTypes

5.4.5.2 Process-Related Extension Types

Process-related links are used to capture information regarding the evolution of a system. In MDE many of these links are automatically generated through transformations, and will therefore be captured using the ***FeatureTransformation*** link. It must however be possible to capture such links manually or automatically as well. We have therefore created the *LinkExtensionType* ***Realization*** to be able to achieve this. This is however the only process-related *LinkExtensionType*, so it has no *subtypes* or *supertype*.

Constraints

- May not be used on the same link as ***ProductRelated*** extensions.
- May only be used on ***Automatic*** or ***Manual*** links (not needed on ***FeatureTransformation*** link as this link type may only be used to record this kind of relationship).

5.5 Usage

A *traceTypeLib* model defines a set of types that may be used to populate a trace repository. The language used to describe these types does however not provide any means to define the logic behind when and how the different types shall be created. In this section we will present

a set of simple java methods that describe the logic behind the trace generation of this library. This will also serve to illustrate how the traceability tool could be used to create traces with this library. In these examples we use the *factory* generated from the traceTypeLib model.

5.5.1 Creating artefacts

The logic behind the creation and retrieving of traceable artefacts of the types in this library is really quite similar. We will start with the creation of a ***ModelElement***.

```
    public ModelElement createModelElement(String location,
            String qualifiedName,
            String featureRef, String type) throws Exception{

        String name = qualifiedName + "::" + featureRef;
'/      creates the name to be shown in viewers and editors

        ModelElement element = null;

'/      retrieves all versions of a ModelElement
        EList<TraceableArtefact> artefactVersions = findArtefacts(location, qualifiedName, featureRef);

        if(artefactVersions.isEmpty()){ //  if no earlier versions of this artefact exists
'/          create a new ModelElement
            element = factory.createModelElement(name, location, "1", qualifiedName, featureRef, type, "");
            element.setRecordTime(new Date().toString());
        }else{
'/          create a new version
            ModelElement tmp = factory.createModelElement(getCurentVersion(artefactVersions));
            String version = new Integer(Integer.parseInt(tmp.getVersion()) + 1).toString();
            element = factory.createModelElement(name, location, version,
                        qualifiedName, featureRef, type, "");
            element.setRecordTime(new Date().toString());
        }
        return element;
    }
```

Figure 29: Create ModelElement

When a tool wants to create a trace concerning the creation of a new ***ModelElement***, it needs to provide the values for the *location*, *qualifiedName*, *featureRef*, and *type* attributes. Some of these values are also used to construct a meaningful name that will be displayed in the trace repository. The *location*, *qualifiedName*, and *featureRef* are first used to check whether earlier versions of the artefact exist in the repository. If no such artefacts exist a new ***ModelElement*** will be created as *version* '1'. If earlier versions exist, a new ***ModelElement*** is created with a new *version* number.

Creation of ***TextFile***, ***Block***, ***TraceableSegment*** and ***TextualArtefact*** is performed in the same way, except that different parameters are used. *TraceableArtefacts* of type ***Block***, ***TraceableSegment*** and ***TextualArtefact*** will however also need to be added to their respective parent. Figure 30 shows how ***TextualArtefact***s are created.

```
public TextualArtefact createTextualArtefact(TextFile parent, int startCol,
        int startRow, int endCol, int endRow){
    String name = parent.getName() + "::" + startCol + "::" + startRow +
        "::" + endCol + "::" + endRow;
    TextualArtefact ta = null;
    try{
       ta = parent.addTextualArtefact(name, parent.getLocation(), "1",
               startCol, startRow, endCol, endRow);
       ta.setRecordTime(new Date().toString());
    }catch(Exception e){
        System.err.println("ERROR");
    }
    return ta;
}
```

Figure 30: Create TextualArtefact

5.5.2 Creating links

Creating ***Automatic*** and ***Manual*** trace links are performed much in the same way as the traceable artefacts; the attributes, source and target are sent as parameter, and used to create a link. The trace links should however use a global counter as a means to provide a number that can be added to the *name* of the trace link for easier visual identification.

```
private static int linkNr;

public Manual createManual(String creator,
        MDEArtefact source,
        MDEArtefact target) throws Exception{
    Manual link = null;
    link = factory.createManual("Link_" + linkNr++, creator, source, target);
    link.setRecordTime(new Date().toString());
    return link;
}
```

Figure 31: Creation of Manual trace links

Creation of ***Transformation*** trace link compositions is created a bit different that ***Manual*** and ***Automatic*** trace links in order to deal with the more complex structure of a composition. Due to the amount of links that may be created during a transformation, it is important that the composition can be efficiently handled.

```
    public Transformation transformation;
    public HashMap<String, RuleExecution> mappingRules = new HashMap<String, RuleExecution>();
    public FeatureTransformation createFeatureTransformation(String ruleID,
                                            String ruleName,
                                            MDEArtefact source,
                                            MDEArtefact target ) throws Exception{
        FeatureTransformation link = null;
        RuleExecution rule = mappingRules.get(ruleName);
        if(rule == null){
            rule = transformation.addRuleExecution(ruleName, ruleID, null, null);
            rule.setRecordTime(new Date().toString());
//          Adds the name of the rule to the (informal) name of the link to
//          ease readability and manual trace inspection
            mappingRules.put(ruleName, rule);
        }

        link = rule.addFeatureTransformation(transformation.getName() +
                " - Link_" + linkNr++, source, target);
        link.setRecordTime(new Date().toString());
        return link;
    }
```

Figure 32: Creation of Transformation link compositions

Assuming that a transformation engine knows the name of the mapping rule that is running at any given time, the composition can be created using a HashMap and an attribute containing the ***Transformation*** composition. The transformation engine initiates the *transformation* attribute, and can then use the method in the same way as the methods for creating artefacts with the addition of a parameter for the name of the rule (the type of the source and target parameters is the super type for all *TraceableArtefacts* generated from this library (MDE). The Hash Map is used to put the ***FeatureTransformation*** links under the correct ***MappingRule***. A new ***MappingRule*** is added the first time the name of a rule appears. With the addition of a similar method for creating ***ExplicitLink***s, using the same Hash Map, the effort of creating the composition is minimal.

5.5.3 Retrieving artefacts from the repository

Artefacts may also be added to the repository when a tool tries to retrieve an artefact that does not yet exist, e.g. when a transformation engine creates a link from a source that is not yet added to the repository. A new ***ModelElement*** should therefore be added if it cannot be found in the repository when it is to be used as the source of a trace link.

```
    public ModelElement findModelElement(String location,
            String qualifiedName, String featureRef, String type) throws Exception{
//      find the current version of an artefact
        EList<TraceableArtefact> artefactVersions = findArtefacts(location, qualifiedName, featureRef);
        ModelElement element = factory.createModelElement(getCurentVersion(artefactVersions));

        if(element.getTraceableArtefact() == null){
//          creates a new ModelElement if no artefacts are found
            element = factory.createModelElement(qualifiedName + "::" + featureRef,
                    location, "1", qualifiedName, featureRef, type, "creator");
            element.setRecordTime(new Date().toString());
        }
        return element;
    }
```

Figure 33: retrieving a *ModelElement*

5.5.4 Example of use

Figure 34 illustrates how a transformation engine could create an *ExplicitLink* between a ***ModelElement*** and a ***TextualArtefact***, using the suggested methods. The methods responsible for the creation are kept in a Class referenced as *manager*. This assumes that the semantics needed to add semantics to the artefact and link are possible to define by the transformation language.

```
e = manager.findModelElement(
        "/C:/test/EJB_PSM.uml/", "EJB_PSM/LibrarySystem/Book", "", "Class");
a = manager.createTextualArtefact(t5, 1, 1, 8, 2);
a.setAsImplementationArtefact();
manager.createExplicitLink("id", "mapClass", e, a).setAsImplementation();
```

Figure 34: Creation of a trace link

5.6 Validation

With respect to the high-level hypothesis, our assumption is that the classification scheme will make it possible to conduct more precise analysis of the traceability information and improve automation of the process. Validation of the classification scheme will be performed through a set of test-cases presented in subsection 6.2.1. For each test-case, a prediction related to hypothesis and a set of success criteria will be presented. The result of the tests will be discussed in 7.2.1. If the prediction turns out to be correct, we will strengthen our assumption regarding the classification scheme. If not it will be falsified. The requirements will be validated in chapter 8.2.2.

5.7 Summary and Discussion

The basic traceability classification scheme discussed in this chapter should be sufficient to capture traceability links, explicit and implicit, between artefacts represented as models or text-files. Provided functionality by tools and transformation engines, it should also be possible to add semantics to the traceability information in a way that allows analysis regarding the development process to be performed. The composition links captured by transformation should also provide the classification needed to analyse the implicit and explicit links created by a transformation engine.

In section 5.5 we also discussed how the classification scheme can be used to capture traces by creating simple Java methods that uses the interface of the traceability tool through the generated classes from this *traceTypeLib* model. These examples shows that the classification scheme is possible to use, and that it only requires that the tools and transformation engines extract information from the actual artefacts and make methods calls.

A thing to notice regarding this classification scheme is that it problems may occur if a model to text transformation language is used to generated models, in the form of XML tags. If this is the case, a model may be represented both as a ***TextFile*** and a set of ***ModelElement***s if it is later used as source for other trace links. This classification scheme hence assumes that model to text transformation languages are not used to generate models.

6 Design of Experiment

6.1 Introduction

This chapter introduces the experiment that will be performed to validate whether the assumption presented in the hypotheses (Table 1) can be strengthened or falsified. The experiment is divided into several test-cases for the classification scheme, and one test-case for the traceability tool.

The test-cases are all presented using the same form:

Description
Short description of the test-case.

Purpose
Short description of the purpose of the test-case.

Prediction
Prediction regarding how the classification scheme/traceability tool will perform in the test. If the hypothesis related to the test-case is true, then the prediction will also be true. The result of the experiments can therefore be used to strengthen or falsify the hypotheses.

6.2 Test cases

6.2.1 Test-Cases Related to the Traceability Classification Scheme

The test cases for the classification scheme are related to the use scenarios discussed in subsection 3.3.5. The number of the hypothesis related to each test is provided in parentheses in the heading of each test-case, e.g. '(H1.1)'.

6.2.1.1 Coverage analysis (H1.1)

According to the discussion in subsection 3.3.5, coverage analysis may be used to find out whether trace relationships that should exist between artefacts are present. This may be performed on *product- related* links (e.g. are all requirements covered?) and *process-related* links (e.g. are all relevant parts of a model used in a transformation?).

Ensuring that all relevant parts of the model is actually utilised by a transformation (H1.1.1)

Description
Coverage analysis may be achieved by checking that all relevant parts of a model (e.g. a UML model) are represented in the trace repository (as ***ModelElement***), and that they are linked to the output of the transformation. This does however require that it is possible to identify the

links expressing this kind of relationship. In this test-case, we will therefore try to identify and discuss how they can be analysed.

Purpose
The purpose of this test case is to validate that the classification scheme makes it possible to find the *implicit* links that is of interest when conducting a coverage analysis with respect to transformations, and to see how well this information supports analysis.

Prediction

P1. Utilising the proposed traceTypeLib model as a classification scheme during development of the LibrarySystem (Appendix A) will enable the user to find all relevant parts of the model that are not covered by a transformation.

Ensuring that all requirements are covered (H1.1.2)

Description
Checking that all requirements in the system are covered by other artefacts in the system may be achieved by locating all requirements of interest in the trace repository, and ensuring that they are covered by the artefacts they are expected to be covered by, e.g. that it is possible to follow trace links from a requirement and analyse whether all expected artefacts are found along the path.

Purpose
The purpose of this test case is to validate that the classification scheme makes it possible to analyse to what degree the requirements are covered by other artefacts of the system.

Prediction

P2. When using the proposed traceTypeLib model to generate traces from the library system, it will be possible to analyse to what degree a requirement is covered.

Change impact analysis (H1.2)

Description
In subsection 3.3.1 traceability change impact analysis was identified as one of the key goals of traceability. This ability is essential as it makes it possible to analyse to what degree a change may impact a system. In this test case analyse the impact of a change on both predecessors and descendants of an artefact. This requires the use of all kinds of trace links, to both descendants and predecessors of an artefact. Due to the complicated nature of this kind of analysis, we cannot expect the process to be fully automated.

Purpose
The purpose of this test case is to validate that the classification scheme makes it possible to conduct proper impact analysis.

Prediction

P3. Using the proposed traceTypeLib model to generate traces from the library system will make it possible to find all artefacts that may be impacted by a change, and find out how they are impacted.

6.2.1.2 Orphan analysis (H1.3)

Finding artefacts that were generated from an artefact that has been deleted (H1.3.1)

Description
Orphans may occur among generated artefacts – model or text – when the artefacts they were generated from are deleted [8]. Such traces are no longer valid. In order to be able to conduct such analysis, it must be possible to separate the traces that record the *implicit* links created by a transformation. We will hence use the implicit links and discuss

Purpose
The purpose of this test case is to validate that the classification scheme makes it possible to identify orphans with respect to transformations.

Prediction

P4. Utilising the classification scheme on the library example will enable the user to find all artefacts that have been generated from elements that have been deleted.

Finding artefacts that are orphaned with respect to artefacts on a previous stage in the development process (H1.3.2)

Description
With respect to requirements traceability, Orphans may occur if an artefact at one stage of the development process has no trace relationship to an artefact at a previous stage. E.g. a java method may be an orphan with respect to requirements if it cannot be traced back to a requirement.

Purpose
The purpose of this test case is to validate that the classification scheme makes it possible to identify orphans with respect to artefacts on a previous stage.

Prediction

P5. When using the proposed classification scheme on the library example, it will be possible to find all relevant artefacts that are orphaned with respect to the requirements.

6.2.1.3 Manual trace inspection (H1.4)

Description

Manual trace inspection is one of the features that should be supported by a traceability solution. It does however put some demands on the traceability classification scheme, as visualisation requires rich semantics in order to be easy and efficient to use.

Purpose
The purpose of this test case is to validate that the classification scheme enables manual trace inspection in the sense that it makes it possible to show the traceability information in a way that is easy to understand for a human.

Prediction

P6. Using the proposed traceTypeLib model to generate traces from the library system provides the information that is necessary in order to visualise the traceability information in a meaningful way.

6.2.1.4 Reverse engineering (H1.5)

Description
Reverse engineering was discussed in subsection 3.3.5.5 as the process of reconstructing a source artefact of a transformation from the target artefact (i.e. the input from the output). This requires that enough information can be found in the trace repository, and that it is possible to identify the information of interest for the reconstruction. We will therefore discuss how the traceability classification scheme supports reverse engineering.

Purpose
The purpose of this test case is to validate that enough information may be found in the trace repository to reconstruct target source artefacts, and that it is possible to find the information of interest.

Prediction

P7. Utilising the proposed traceTypeLib model on the library system will make it possible to reconstruct an artefact from the traceability information.

6.2.2 Test cases related to the traceability tool

The test case for the traceability tool is aimed at validating hypothesis H3.

6.2.2.1 Integrating the traceability tool

Description
According to *Tool Requirement 7*, the traceability tool shall be easy to integrate with external plug-ins. We will therefore implement an eclipse plug-in which uses the traceability tool to generate traces from UML2 [42] models.

Purpose
The purpose of this test is to validate how well the traceability tool may be integrated with other plug-ins.

Prediction

P8. The traceability tool can be easily integrated with UML2 [42] based editors on the Eclipse platform [38].

6.3 Summary

In this section we have presented test-cases regarding the classification scheme and the traceability tool. Each test-case is based on a hypothesis and has an associated prediction. The prediction will be used to falsify or strengthen the hypothesis.

7 Testing and results

7.1 Introduction

In this chapter we will discuss the results of the experiments described in chapter 6. The discussion will be illustrated using the *TraceNavigator* view described in section 4.7.

7.2 Analysis of Results

The purpose of the experiment is to validate our solutions in terms of whether our assumptions regarding the classification scheme and the traceability tool can be strengthened.

Because the tool is generic in nature, it does not support all kinds of analysis fully. It does however support browsing of the information contained in the repository, and should therefore be sufficient to illustrate the discussion.

7.2.1 Test cases Related to the Classification Scheme

7.2.1.1 Coverage analysis

Ensuring that all relevant parts of the model is actually utilised by a transformation

When conducting a coverage analysis with respect to a transformation, it is not surprisingly the trace links created by a transformation that are of interest. These links tell us what was generated during the transformation, and what it was generated from. It is however reasonable to only use trace links that was *implicitly* generated (***FeatureTransformation***) by a transformation engine to conduct the coverage analysis – these links tell the story of what happened during a transformation. The *explicit* links (***ExplicitLink***) on the other hand are created by the volition of a developer, and hence does not really provide a solid foundation to conduct a coverage analysis.

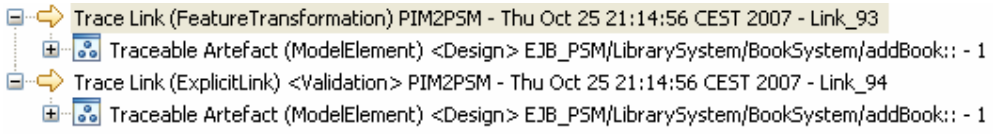


Figure 35: Descendants of BookSystem.addBook() in the PSM

Figure 35 shows two trace links that were generated from the PIM2PSM transformation. By using the classification of the links we can see that only *Link_93* is a ***FeatureTransformation***, and is hence the only link of interest. Even though they both point to the same artefact in this case, *Link_94* cannot be used to conduct the coverage analysis. If *Link_93* had not been present, the 'addBook' Operation in the PSM had not been generated from the PIM, and the

Operation in the PIM would hence not be covered by the transformation. Explicit links could still be created by a transformation from an input artefact to an artefact in the output model (e.g. from one artefact in the input model to an artefact in the output model that is generated from another artefact in the source model).

It may however often be the case that one element is used as input of several transformations. It is therefore also necessary to be able to find a ***FeatureTransformation*** link that was part of the transformation we want to find traces from. This is done using the *mappingID* Attribute of the ***Transformation*** that contains a ***RuleExecution*** containing the ***FeatureTransformation*** link. In the *TraceNavigator*, the transformation is identified visually by the *name* and *recordTime* of the mapping.

The fact that all trace links created by a transformation are stored in a composition makes it easy to find all trace links of interest when analysing one specific transformation. Figure 36 shows *Link_93* as it is stored in the *trace repository*.

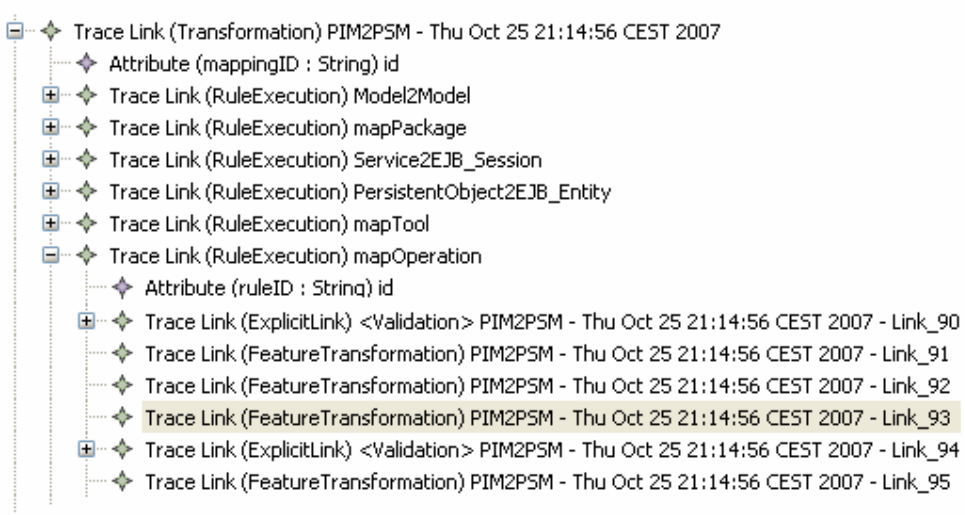


Figure 36: The feature transformation shown as it is contained in the repository

Each ***Transformation*** contains a complete record of all rules that was executed, and each ***RuleExecution*** contains a complete set of traces that was created by it. Conducting a coverage analysis for the whole transformation or for a single mapping rule could therefore be conducted by inspecting all the links of interest, or by checking that all artefacts of interest in the source model has ***FeatureTransformation*** link contained by a specific ***Transformation*** or ***RuleExecution*** to an artefact in the target model.

Evaluation

Finding the correct link to use for the coverage would have been difficult without classification on the links. If there had been ***Automatic*** and ***Manual*** links from the artefacts used to conduct the coverage analysis as well, the task would have been even more difficult. However, using the classification we were able to find the correct links to base the coverage analysis on.

Ensuring that all requirements are covered

In order to ensure that the requirements are covered, we can follow the links descending from the requirements in the requirements document. This allows us to see which artefacts have a relationship to the requirements. There are however not all links that are of interest for this specific analysis. What we are interested in are the links comprising the ***Validation*** dimension of the traceability graph. Following these links make it possible to find all artefacts that are supposed to support satisfy the requirements at various stages of the development process. In order to analyse to what degree the requirements are covered we do however need to know how the artefacts found along this dimension satisfy a requirement. These can be achieved using the *Extensions* (i.e. *Extensions* typed with subtypes of ***ArtefactType***) applied to the *TraceableArtefacts*.

Figure 37 shows the descendants of the first requirement in the requirements document. Following the ***Manual*** and ***ExplicitLink*** links with *Extension* ***Validation*** and using the *Extension*s applied to the traceable artefacts give enough information to see that this requirement is satisfied by two separate ***Implementation***s. We can also see that the development process is satisfying in the sense that there are an ***Analysis*** artefact (the Use Case) and two ***Design*** artefacts (from the PIM and the EJB-PSM) in each branch leading to a java implementation. The ***TextualArtefact*** representing a java method is an explicitly defined block used to identify the block of text comprising the implementation of the method.

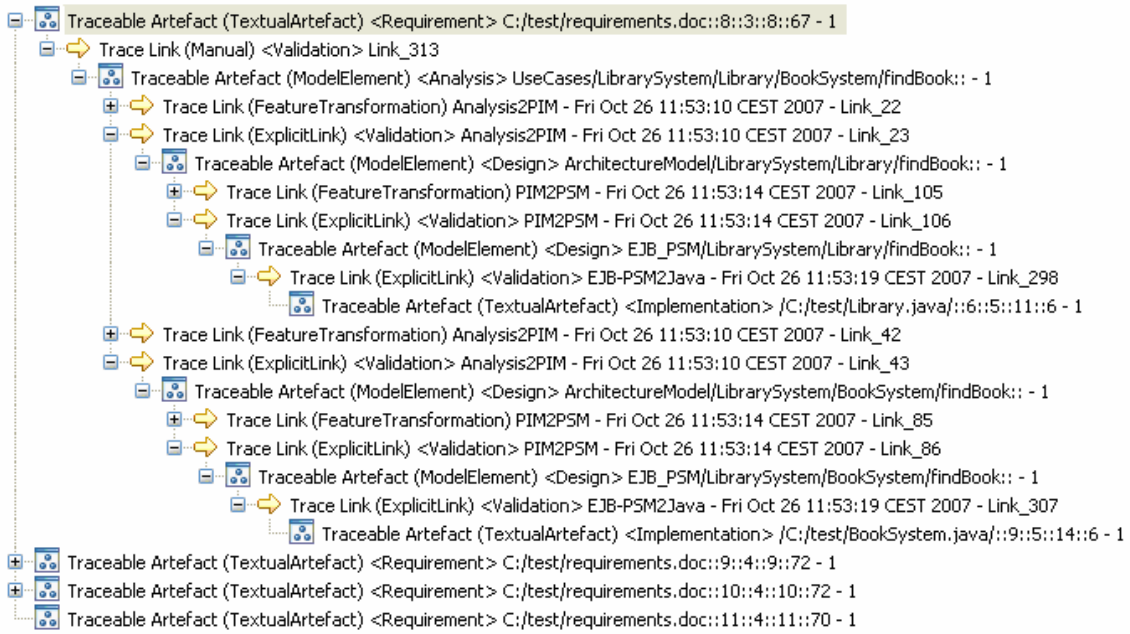


Figure 37: Show Descendants with requirements.doc as input

Evaluation

Conducting the coverage analysis as accurate as above would have been very difficult without the classification. We would have been able to find all the artefacts of interest, but we would also have found artefacts and links without interest for the analysis. Furthermore, without classification on the traceable artefacts, it would have been very difficult to deduce the degree of coverage.

With this simple experiment, we have show that it is possible to tell to what degree an artefact is covered. Based on the ***Requirement*** Extension on the traceable artefacts representing the requirements in the requirement document it is also possible to automatically find all requirements in the repository and find those that are not covered by a certain kind of artefact (using the ***Extensoin***), or "compute" whether all artefacts are completely covered (e.g. based on parameters used as input to a query regarding the ***Extensions*** that are expected to be found along the ***Validation*** path).

7.2.1.2 Impact Analysis

Change impact analysis may involve several steps, manually or automatic, which needs to be conducted in order to estimate the full impact of a change. The "cost" of a change may be seen as the sum of the dependencies that exists between the artefact in question and the artefact it is linked to directly or indirectly. Furthermore a change impact analysis on an artefact also involves conducting change impact analysis for all its sub components for a complete cost evaluation.

In Figure 38 we can see the descendants of the Class 'BookSystem' in the PIM. We can see that this Class has two links (a ***FeatureTransformation*** (*Link_65*) and an ***ExplicitLink*** (*Link_67*) with the extension ***Validation*** – both from the same transformation) with the same target Class in the EJB-PSM. The Class in the EJB-PSM is linked through an ***ExplicitLink*** (*Link_306*) extended with ***Validation***. From this information we can deduce that the Class 'CustomerSystem' in the PIM has been used to generate a new model artefact with the same name, and that the new artefact has been used to generate an ***Implementation***. This means that there are two artefacts that may be affected by a change to the Class in the PIM. The fact that these artefacts are generated however means that the impacted artefacts may be regenerated automatically. The fact that the ***ExplicitLink***s also are created automatically, suggests that the cost of changing the artefact may not be very high.

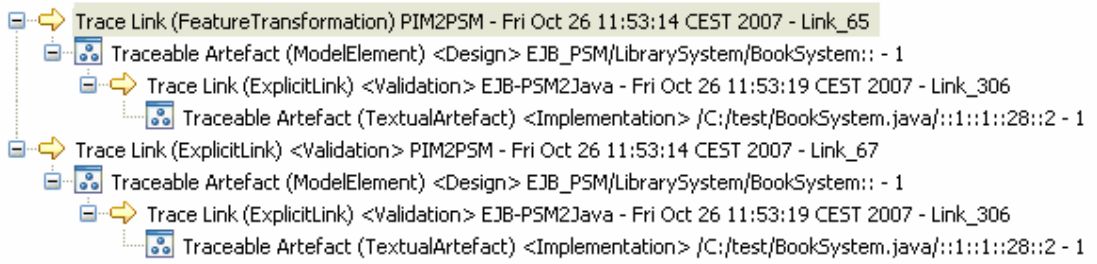


Figure 38: Descendants of BookSystem in the PIM

Changing an element that is part of a chain of artefacts used in transformation may however, also affect the predecessors, as e.g. deleting the Class 'BookSystem' in the PIM will lead to a break in the chain. In this case, its descendants would become orphans, while the requirement would no longer be satisfied.

Figure 39 shows predecessors of the Class 'BookSystem'.

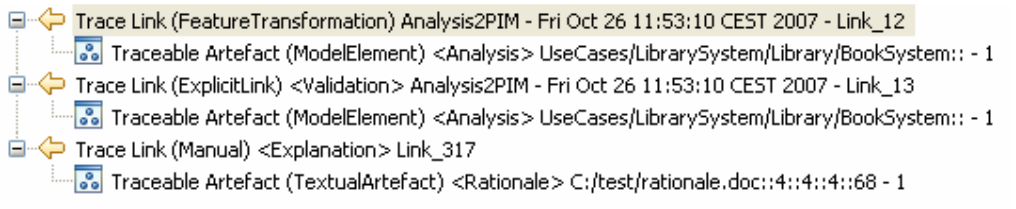


Figure 39: Predecessors of BookSystem in the PIM

As we can see, there are three trace links pointing at the Class 'BookSystem'; one ***FeatureTransformation***, expressing the fact that the Class is generated from a Component in the *useCases* model, an ***ExplitLink*** expressing the fact that the Class was generated as part of the satisfaction of the requirement (the ***Validation*** dimension), and a ***Manual*** extended with ***Explanation*** link to a ***Rationale*** explaining the rationale behind the Class. This information tells us that the Class is generated from another artefact, suggesting that changes to the artefact could be performed at the previous stage, or by changing the mapping responsible for its creation. The fact that there is a ***Manual*** link to the rationale suggests that this relationship must be manually maintained if the artefact is changed, and that the rationale may need to be changed manually.

For a more fine-grained impact analysis, the artefacts that are contained by the artefact that are the basis for the impact analysis should also be taken into consideration. This will give a more accurate cost analysis. The analysis can be conducted in the same way for each contained artefact of interest.

Evaluation
Conducting an impact analysis without classification would have been possible, but would have required considerably more effort in order to identify what kind of artefacts that are represented in the trace repository, and what kind of relationship they have to each other. With the use of the semantics on the trace links and traceable artefacts we were able to find all affected artefacts, and to estimate the cost of an impact in terms of which of the traceable artefacts and trace links that may be automatically generated, and identifying the transformations that are responsible for the creation of artefacts. Using the ***Transformation*** composition it is possible to identify what mapping rules that are responsible of the creation of certain artefacts. This information may be used to inspect how a given change will be reflected on artefacts generated by a transformation.

7.2.1.3 Orphan analysis

Finding artefacts that were generated from a deleted artefact

Finding artefacts that are generated from an artefact that have been deleted requires that we are capable of identifying the links to use for this purpose. As we have discussed in previous subsections, such links are classified as ***FeatureTransformation***. Following these links from target to source makes it possible to identify artefacts that were generated from deleted artefacts. If the actual source artefact (i.e. the artefact that is represented by the *TraceableArtefact* that is the source of the link) cannot be found, the target is an orphan. The actual artefact may be found by using the identifying attributes of an artefact. E.g. the actual artefact represented by a ***ModelElement*** may be found using the *location* property and the

qualifiedName and *featureRef Attribute*s of the ***ModelElement*** in question. If orphans are found, the transformation should be rerun.

The same approach may be used to find orphans with respect to ***Manual*** or ***Automatic*** links with the ***Realization*** *Extension*. In such cases, manual effort would have to be made to solve the problem. If the source of links extended with a ***ProcessRelated*** *Extension* is found to be deleted, we cannot necessarily conclude that the target is an orphan however, as links with these *Extension*s do not describe *process-related*, but *product-related* links, we could rather conclude that the link is no longer valid.

Evaluation
Identifying the trace links to use for the orphan analysis would have been more difficult without classification, as we could not identify links that were created by a transformation. Through the discussion above we have described how it is possible to use the traceability classification to find the right links to use, and to retrieve the information required from the *TraceableArtefact*s.

Finding artefacts that are not described in the requirements

In Figure 40 we can see the predecessors of the Operation rentBook in the Class BookSystem in the PIM. By following the ***ExplicitLink*** links (also ***Manual*** or ***Automatic*** if existing) with *Extension* ***Validation*** we are able to find all product-related relationships expressing something that should satisfy a requirement. The fact that we cannot find a ***Validation*** link to an artefact with a ***Requirement*** extension, tells us that the 'rentBook' Operation is in fact is an orphan.

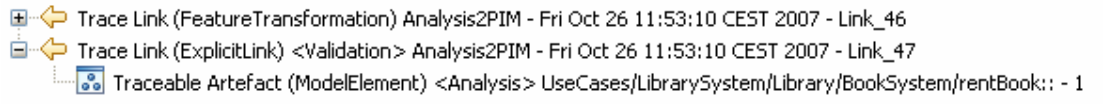


Figure 40: Show Predecessors with BookSystem.rentBook() in the PIM as input.

This process could be automated by checking whether *TraceableArtefact*s extended with ***Analysis***, ***Design*** or ***Implementation*** exists in the repository or in a model or file that does not have a ***Validation*** link to a *TraceableArtefact* extended with ***Requirement.***

Evaluation
The discussion above has shown that it is possible to automatically find all artefacts of certain types that are orphaned with respect to requirements. The analysis could have been conducted without classification, but it would require much more effort to identify the artefacts of interest.

7.2.1.4 Manual Trace Inspection

Manual trace inspection functionality depends on the implementation of such functionality, but using the classification suggested above supports the cases discussed in this thesis. The *TraceNavigator* supports most cases, but it is not capable of retrieving information from the actual artefacts. This is because it only works on the trace repository, but it would be sufficient with minor improvements. This does however require that it knows what kind of

artefacts to work on, i.e. it needs to know how to find contained elements on e.g. a UML model. A tool that integrates the traceability tool is expected to implement such functionality.

Evaluation
Through the figures used in the previous subsections, we have shown that the *TraceNavigator* makes it possible to visualise the traceability information with the classification scheme in a way that is relatively easy to understand. Without some kind of classification, i.e. without information regarding different link types, and artefact types, this is very difficult to achieve.

7.2.1.5 Reverse Engineering

Reverse engineering can be performed using traceability information by following trace links backwards, and using the information found in a *TraceableArtefact* among the predecessors of an artefact. In order to perform reverse engineering, a tool must be able to find the links that links the input and the output of a transformation. This can be achieved using ***FeatureTransformation*** links as these are the only links that tells us precisely how a source artefact is related to the target. Reconstruction of a model artefact may be performed by using the information found on the *TraceableArtefact* used to represent it.

In Figure 38 we can see that there is a ***FeatureTransformation*** link from the Class 'BookSystem' in the PIM to a Class with the same name in the PSM. Reconstruction of the Class in the PIM from the Class in the PSM could be performed following this link to the source. The source Class can then be reconstructed by creating an artefact of the type described by the *type* attribute of the ***ModelElement***, and the name can be extracted by using the last fragment of the *qualifiedName* attribute. The rest of the artefacts comprising the PIM could be reconstructed by finding all *TraceableArtefact*s having the same value in the *location* property, and reconstructing each artefact in the same way. The new artefacts can be used to create compositions by comparing the values found in the *qualifiedName* attribute (i.e. the location of a model artefact within a model). *TraceableArtfact*s with values in the *featureRef* attribute can be used to set properties of the reconstructed model artefacts. How accurately an artefact may be reconstructed depends on how much traceability information that exists. This procedure also requires that the tool responsible for the reconstruction can recognise the type described in the *type* and *featureRef* attributes of a ***ModelElement***.

Evaluation
In the discussion above we have described how a source artefact may be reconstructed from by following links backwards from the target artefact of a transformation. This would have been more difficult without classification of the trace links, as it would be more difficult to find the right artefacts to use. However, we saw that the *qualifiedName* and the *featureRef* attributes of ***ModelElement*** – if we had not known that it was a model element, it would be difficult to know how to reconstruct it. We can therefore conclude that the reconstruction could not have been achieved in the same way without classification of the information.

7.2.2 Test-Cases Related to the Traceability Tool

7.2.2.1 Integrating the traceability tool with an external tool

To validate that the traceability tool can be easily integrated with external plug-ins, the traceability tool has been used as the basis for an Eclipse plug-in called *ManaulTracer* that

allows trace links to be manually created between UML2 [42] model elements. This was done the summer of 2007 as part of the EU project "ModelPlex" [43], for Bjørn Nordmoen at Western Geco.

The classification scheme

The *ManualTracer* uses a very simple classification scheme consisting of a *ModelElement* which may be extended with one of the sub classes of *ArtefactType*. This is a very simple classification scheme, but it is sufficient to classify model elements at different stages of a development process. As the model shows, the only available *TraceLinkType* is *ManualTraceLink*. In this simple plug-in this was found to be sufficient as the type of a trace link could be deduced by its source and target, e.g. a *Requirement* to *UseCase* link means that the UseCase satisfy the requirement etc.

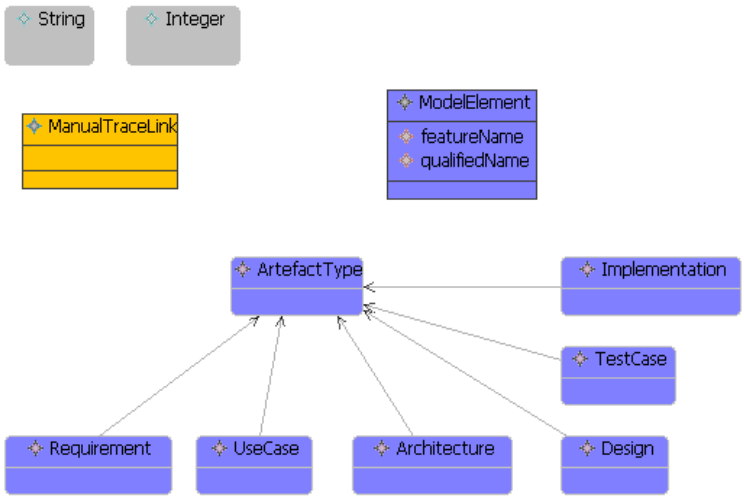


Figure 41: The classification scheme used by the ManualTracer plug-in

Features

The *ManualTracer* allows users to create traces directly from any UML2 models shown as graphical diagrams or from any viewer (e.g. tree-editor or table) that displays UML2 elements. The trace information is added to trace repository model, and can be inspected or queried.

All the actions provided by the *ManualTracer* are available from the popup menu (the menu that appears when an item is right-clicked) for any EObject (EMF) or EditPart (GEF), but the actions are only performed when an UML2 element is represented. In Figure 42 these actions are shown when a Use Case in a Papyrus [44] diagram is right-clicked. All the actions are found under 'Trace Management'. The actions are explained in the following subsections.

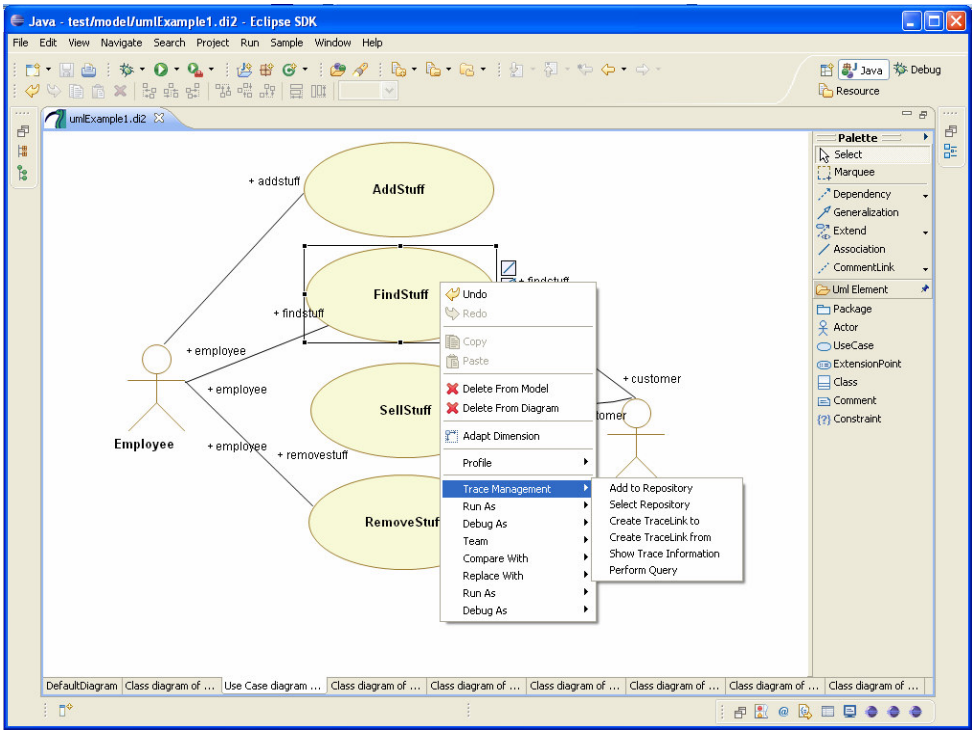


Figure 42: TraceManagement menu in the graphical editor of Papyrus

Add to Repository

The Add to Repository action allows users to add a model element to the repository, with an *ArtefactExtension*. It is not required to add element to the repository in this manner, but it makes it possible to performed more advanced queries on the traceability information. E.g. adding all design classes in a package to the repository with the *ExtensionType* 'Design' makes it possible to find all design classes in the package without a link to an 'Implementation' class. The information will not be added to the repository if it is already present.

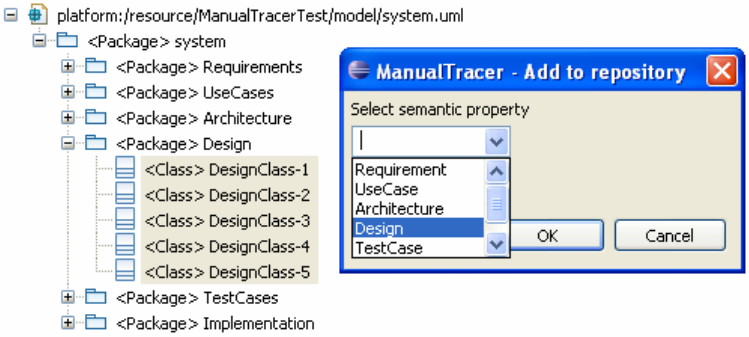


Figure 43: The add to repository dialog activated from the UML2 tree-editor.

In Figure 43 all the design classes are selected from the UML2 tree-editor, and added to the repository as *TraceableArtefacts* with an Extension of type *Design*.

Select Repository

The *ManualTracer* plug-in does not work on a default repository, it is therefore necessary to select a trace repository to use before any other actions can be performed.

Create TraceLink to/from

When a user selects one of these actions, a dialog appears that allows the user to navigate the work space to find the appropriate model file, and navigate the model found inside this file. The selected element will become source or target of the link, depending on whether 'Create Link from' or 'Create Link to' was selected. Before the trace link can be created, the user must also select (in the 'Set Semantic Properties' area) what artefact extensions shall be added to the source and target artefacts in the repository. In Figure 44 a link is created from a 'Requirement' to a 'UseCase'.

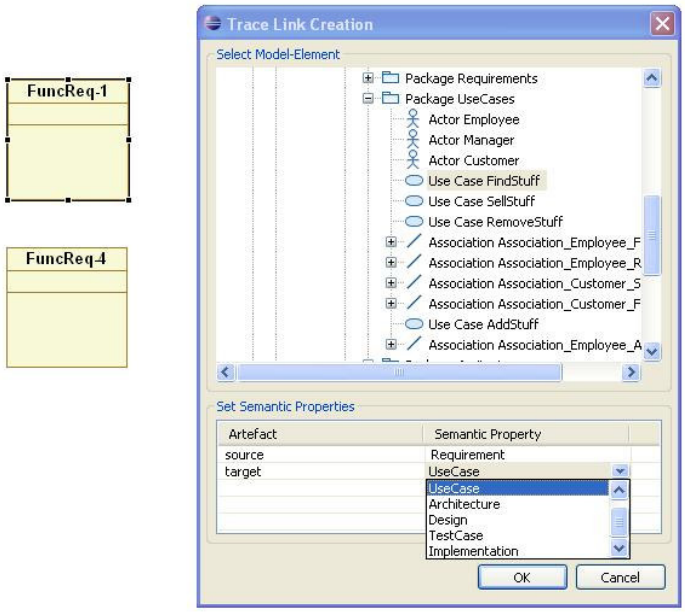


Figure 44: Trace link creation with ManualTracer.

Show Trace Information

This feature makes it possible to inspect all the traces to and from any UML2 model element. The predecessors are displayed in the left viewer, while the descendants are displayed in the right viewer.

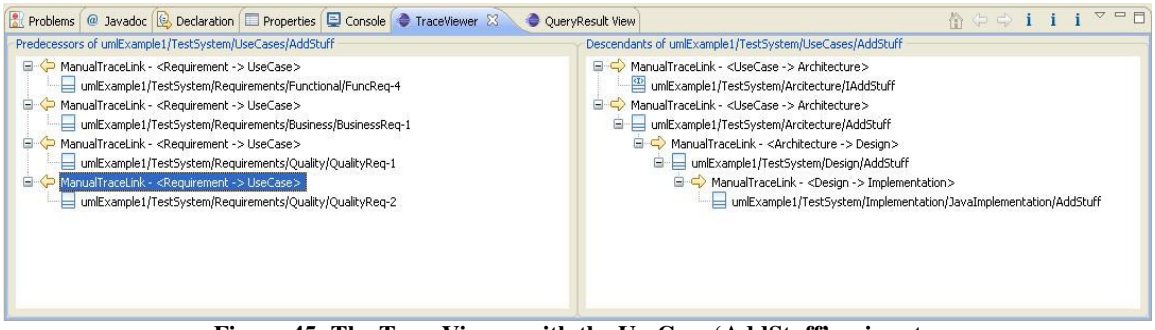


Figure 45: The TraceViewer with the UseCase 'AddStuff' as input.

Perform Query

The 'Perform Query' action allows a user to perform queries based on the information in the trace repository.

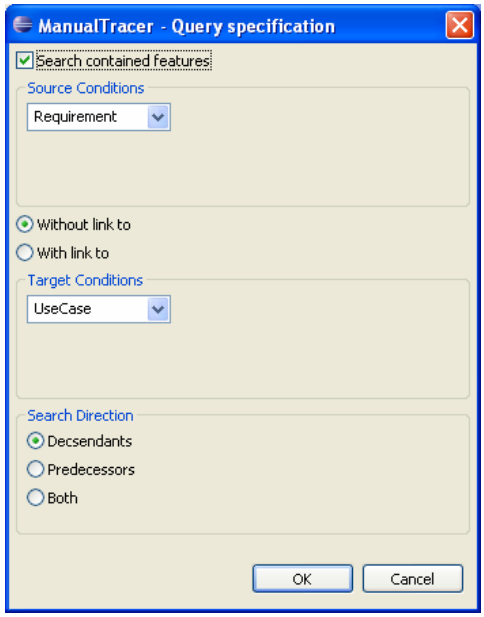


Figure 46: The Query specification dialog

The query uses the following parameters:

- ***Search contained features*** – specifies whether the selected element it self or the elements contained by it shall be input for the query.
- ***Source Conditions*** – filters out elements that are represented in the trace repository with a certain *Extension* which will be the input for the next step of the search ('Any' may be selected if no filter is desired). Only enabled if 'Search contained features' is checked.
- ***With/Without link to*** – specifies whether to find elements satisfying the source condition with or without trace links to elements satisfying the target condition. Only enabled if 'Search contained features' is checked.
- ***Target Conditions*** – condition regarding the *Extension* of the target *TraceableArtefacts*.
- ***Search Direction*** – specifies whether to follow the trace links found in the *sourceOf* or *targetOf* set of the input artefacts.

In Figure 45 the input to the query is a Package that contains the requirements (Classes). The conditions specifies that we want to find all elements in the UML2 model that is represented in the trace repository as a *TraceableArtefact* containing an *Extension* of type 'Requirement' that does not have a link to a *TraceableArtefact* with an *Extension* with type 'UseCase' (either directly or indirectly). The query will be performed by recursively following the *TraceLink*s found in the *sourceOf* set of the *TraceableArtefact*s. The result of this query will hence be all requirements that are not covered by a Use Case.

The result of the query is shown in the *QueryResult View*. In Figure 47 this view shows the result of the above query. In this case the result contained two 'Requirement' Classes.

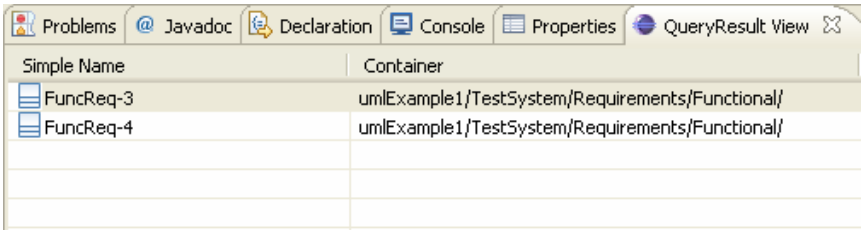


Figure 47: The QueryResult View

Implementation

The *ManualTracer* plug-in works as a layer between the traceability tool and UML2 models, providing functionality that allows users to manually create, inspect and query trace links between UML2 elements. This is achieved by providing a set of dialogs and views that allows users to fetch information from UML2 models and use it to store traces in a trace repository. The information in the repository can then be viewed and queried.

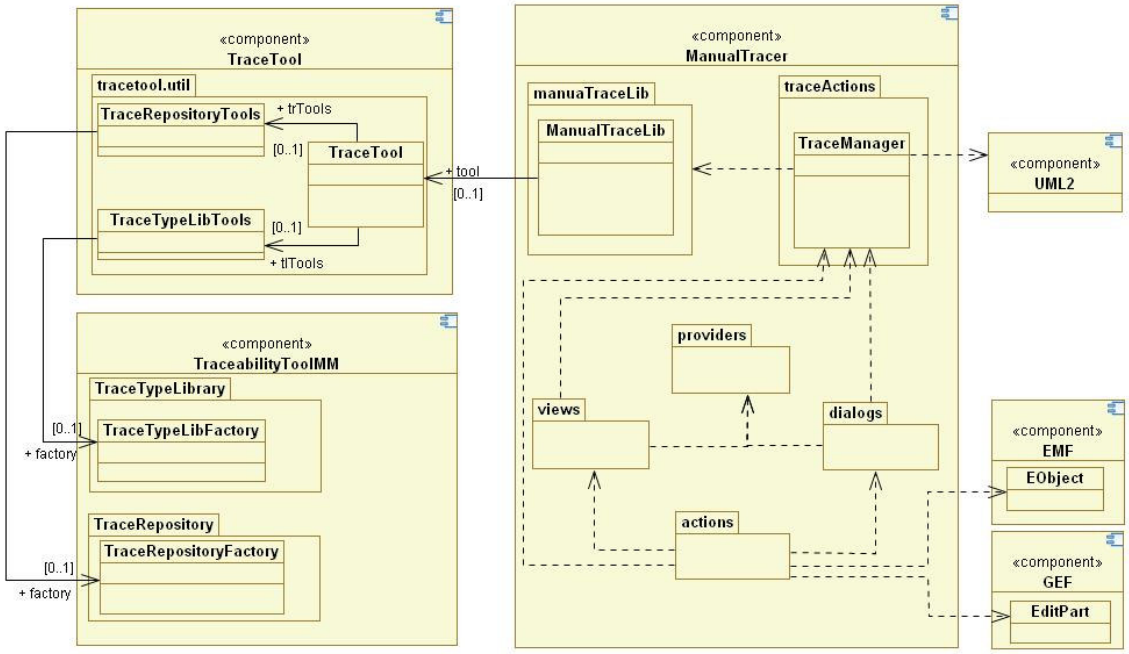


Figure 48: A simple overview of the ManualTracer implementation

Figure 48 shows a simple logical overview of the *ManualTracer* and its interaction with other plug-ins. The packages of *ManualTracer* are summarised below:

- **manualtracer.dialogs** – contains the dialogs used to interact with the user.
- **manualtracer.manualTraceLib** – contains the generated classes from the *traceTypeLib* model in Figure 41. These classes provide the linkage to the underlying *traceTypeLibrary* and *TraceRepository*.
- **manualtracer.popup.actions** – contains the action delegates that are triggered whenever an action is selected from the popup menu.
- **manualtracer.providers** – contains the providers that are used to provide the views with items, labels (text and image).
- **manualtracer.traceActions** – contains the *TraceManager*, which is the class responsible for the communication between the *ManualTracer*, the *TraceabilityTool*,

and UML2. This static class (to assure that only one repository is used) offers an interface that makes it possible to create and retrieve traceability information. It creates traceability information by using the *ManualTraceLibFactory* of the *manualTraceLib* Package.

- **manualtracer.viewes** – contains the views that are used to visualise information (TraceViewer and Query Result View).

The *ManulTracer* is hooked up with the Eclipse framework by declaring extensions to its User Interface; to its popup menus (*org.eclipse.ui.popupMenues*), and to its views (*org.eclipse.ui.views*). These extensions define;

1. The addition to the popup menu and for what objects the menu shall be added. The ManulTracer adds the popup menus to *org.eclipse.emf.ecore.EObject* (any object in the EMF framework is an extension of this – including UML2 objects) and *org.eclipse.gef.EditPart* (the controller of any graphical element in a GEF diagram – including GMF and Papyrus).
2. The views that are added.

Whenever an action from any popup menu is selected, an associated action in the *manultracer.actions* Package is triggered. These actions are updated on which elements that are selected (the elements from which the action was triggered), and are responsible for delegating the actions to other classes. This includes initialisation of views and dialogs, and setting their input. Whenever a user finishes or cancels a dialog, the action classes retrieves the parameters set by the user, and executes the appropriate actions through the *TraceManager*. The views and dialogs use the *TraceManager* to refresh information and some other convenience methods.

The basic structure of the *ManualTracer* can hence be said to be typical for any eclipse plug-in – it uses different UI extensions to provide functionality to the users. What makes it a traceability tool is what information it maintains. This is handled through the TraceManager class and the *manualTraceLib* Package. The latter is generated from the *traceTypeLib*, and hence makes it easier to handle traceability information specific to this library, while the first provides an interface for capturing traces. It does this by using the *manualTraceLib* Package (also uses the interface of the traceability tool when the generated code is not sufficient).

With this implementation we have shown that the Traceability tool can easily be integrated with other tools – it is simply a matter of using the interfaces provided by the Traceability tool, just like any other interface.

7.3 Summary

In this chapter we have evaluated the hypotheses presented in chapter 1. This was performed by performing a test-case for each of the hypotheses. The tests-cases were described in the previous chapter with a prediction regarding the result of the tests utilising the classification scheme and the traceability tool. Based on the discussion in this chapter, we can conclude that the assumptions presented by the hypotheses are strengthened, as none of the predictions were falsified.

8 Discussion and evaluation

8.1 Introduction

In this chapter we tart with a discussion on the fulfilment of the requirements for the traceability tool and the classification scheme presented in chapter 4 and chapter 5. We then continue with at more general discussion and criticism of the work with the thesis.

8.2 Evaluation

8.2.1 Fulfilment of Tool Requirements

Tool Requirement 1

The traceability tool shall use model-driven approaches.

Fulfilled! The tool is based on two metamodels, which are created with the use of EMF. This Framework allows creation of metamodels, and generation of code from the metamodels, that allows instantiation of the models to be made programmatically. The models are also used to generate tree editors for editing and browsing trace repositories and trace type libraries, which uses the generated code to create and modify models. In addition we have created a GMF editor for the trace type libraries, for graphical creation and modification of trace type libraries. This makes the definition process more user-friendly. Furthermore, the *traceTypeLib* models can be used to generate Class libraries in Java, making the process of populating a trace repository much easier.

In fact, there is nothing about the tool that is not based on models. All the core functionality is generated from models, and all data is stored as models.

Tool Requirement 2

The models used by the traceability tool shall conform to the 4 meta-layers of OMG.

Fulfilled! The tool is based on EMF, which uses ECore as its meta-metalanguage (meta-metamodel). This language is a simplified version of the MOF and is actually the basis for Essential MOF (EMOF) which is OMG's simplified version of its own MOF. As ECore conforms to the four metalayers of OMG (i.e. it resides at the M3 level), the models used by the traceability tool will also have to conform to the four metalayers.

Tool Requirement 3

The traceability tool shall support trace repositories

Fulfilled! Indeed one of the major components of the traceability tool is the *traceRepository* metamodel, and the functionality to support creation and modification of repositories.

Tool Requirement 3.1

The traceability tool shall be based on a metamodel

Fulfilled! The traceability tool is based around two metamodels (*traceTypeLib*, and *traceRepository*) which defines the language for definition of libraries of trace types, and the language used to capture traces based on the types defined in by a library. This was also discussed regarding the fulfilment of **Requirement 1**.

Tool Requirement 3.2

The traceability tool shall support generic definition and customisation of reference models

Fulfilled! The language for generic definition and customisation of reference models is described by the *traceTypeLib* metamodel. To ease the use of this language, a GMF editor was created to define the concreted syntax for the language at a higher level of abstraction. This graphical editor makes it relatively easy to define the types and composition of types comprising a reference model (i.e. a library of trace type definitions). Several libraries may be used in combination, or combined (use references between each other) to populate a trace repository with traces of a specific type. This functionality is supported by the EMF framework.

In addition to the ability to define traceability types in a model, the traceability tool also provides functionality to generate a Java library of the defined types. The classes comprising the library can be constructed through the use of a factory class, which assures that all the classes uses the same underlying *traceTypeLib* model and the create traces to the same repository. Creating a class of a specific type adds a trace of the specified type to the repository. This functionality eases creation of the types defined in a *traceTypeLib* model quite a bit, as a developer implementing a tool with support for traceability can program with traceability types in the same way as with any other Java library. This adds a new dimension to the generic part of the tool.

Tool Requirement 3.3

The traceability tool shall support persistent storage of traceability information

Fulfilled! The traceability tool supports persistence storage of traceability information through the use of *traceRepository* models. Models may not scale as well as traditional relational databases, but they are in turn more flexible as they are easier to integrate with other models, and modelling tools. E.g. using a model makes it possible to keep references between the repository and the traceTypeLibrary model, but also allows external tools like a transformation engine to populate trace repositories as output from a transformation. Functionality for performing queries against a repository can be provided with the use of e.g. EMFQuery.

Tool Requirement 4

It shall be possible to identify where the traced artefact is located

Fulfilled! The addition of the *location* property to *TraceableArtefact* in the *traceRepository* metamodel allows the location of an artefact that is being traced to be captured. This property may be used to store the location of an artefact both locally (e.g. *platform/resource/project/file*) or globally (e.g. *www.name.domain/location/file*) depending on whether the repository is used to stored traces in a specific workspace or traces from multiple users simultaneously on a server.

Due to the generic nature of the traceability tool, the *TraceableArtefact* does not have properties for identifying artefacts within a file (unless the location property is used for this purpose). Information regarding the identification of artefacts within a file could be kept in specific *Attributes* defined for a specific *TraceableArtefactType*.

Tool Requirement 5

It shall be possible to identify when the traceability information was recorded

Fulfilled! Identifying when the traceability information was recorded is supported by the *recordTime* property of an *IdentifiableElement* (the super type of *TraceableArtefact*s and *TraceLink*s). This property contains a string value representing the date and time of the capturing. The string may be on any form.

Tool Requirement 6

The traceability tool shall be implemented as an Eclipse EMF plug-in

Fulfilled! The traceability tool is indeed implemented as an Eclipse plug-in, using the EMF framework.

Tool Requirement 7

The traceability tool shall be easy to integrate with external plug-ins

Fulfilled! In 7.2.2.1 we saw that the traceability tool has been successfully used to provide functionality to manually create traceability links on UML2 models, by defining extensions to EditPart (GEF) and EObject (EMF). This implementation showed that it was easy to integrate the plug-in with other plug-ins, in the sense that the traceability tool can be utilised by other plug-ins by simply making method calls to the interface of the traceability tool or by using the generated Java library whenever information needs to be added or retrieved from the trace repository. The traceability tool can hence be used like a database storing traces.

Tool Requirement 8

It shall be possible to create traceability information both automatically and manually

Fulfilled! The traceability tool does offer functionality to create traceability information both automatically and manually.

Traceability information may be created automatically by a tool or through transformations with traceability generation code encoded within the transformation. The first may be performed by using the Java interface of the traceability tool. The latter has not been tested, but the fact that both the trace repository and the trace type library are defined by models means that any model to model transformation language with support for querying (i.e. to retrieve types from a *traceTypeLib* model to use in the creation of traces) should be capable of creating instances of the *traceRepository* model.

Support for manual creation of traceability information is supported by the traceability tool through the generated EMF editor. This editor allows *traceTypeLib* models to be used to create traceability information of a specific type. The editor is however, not very simple to use, as the repository has to be edited directly, and does not perform any constraint checking, hence relying on a developer to perform the task correct. The fact that the traceability tool is a generic tool however makes it difficult to provide a simpler means to manually generate traceability information. Functionality, for easing the task of manual trace creation, like the functionality provided by the *ManualTracer*, should therefore be provided by tools that uses the traceability tool.

Tool Requirement 9

The traceability tool shall support both high-end and low-end use of traceability

Fulfilled! Even though the traceability tool has not been tested in a real-life development process, the features fulfilled by the following sub requirement should make it possible to create traceability information at varying level of details and make it possible to exchange the information between high-end and low-end users, as long as they use the same *traceTypeLibrary* model.

Tool Requirement 9.1

The traceability tool shall make it possible to capture traces at various levels of details, depending on what information is available at any given time

Fulfilled! Through the concept of extension types (i.e. *ArtefactExtension* and *LinkExtension*) it is possible to dynamically extend *TraceableArtefact*s or *TraceLink*s at any given time during their lifetime. This means that e.g. a *TraceableAttefact* may be created as a basic *TraceableArtefactType*, and that semantics may be added at any given time by adding an *Extension* of a specific *ArtefactExtensionType* to the *TraceableArtefact* as long as the *ArtefactExtensionType* or its *superType* is found in the *extensions* set of the *TraceableARtefactType*. Moreover, this concept meant that traceability users may create traceability information at varying level of detail, and still be able to exchange the information with other traceability users using a different level of detail.

Tool Requirement 9.2

The traceability tool shall make it possible to trace an artefact through its whole lifetime

Fulfilled! How artefacts are traced through their lifetime depends to some degree on the trace type library that is used to capture traces. The fulfilment of the following sub-requirements does however provide some support on the area.

Tool Requirement 9.2.1

Each Artefact must be uniquely identified

Fulfilled! The ability to uniquely identify *TraceableArtefact*s and *TraceLink*s is supported by the *id* property of *IdentifiableElement*. This property contains a value that is composed of the MAC-address of the computer creating it, creation time in milliseconds, and a random number. This identification scheme should assure that no trace repository contains *IdentifiableElement*s with the same id.

Tool Requirement 9.2.2

Must support different versions of an artefact

Fulfilled! Versioning of artefacts is supported with the version property of *TraceableArtefacts*. We assume that linkage between the old and the new version is performed with a *TraceLink* of a designated type or with a specific *LinkExtension*.

8.2.2 Fulfilment of the classification scheme requirements

CS Requirement 1

The classification scheme shall support traceability in all stages of the development process of the library system (Appendix A).

Fulfilled! Through the experiments conducted in subsection 7.2.1 we can conclude that the classification scheme supports traceability in all stages of this simple example.

CS Requirement 2

The traceability classification scheme shall allow semantically rich traceability information to be captured.

Fulfilled! Through the experiments conducted in subsection 7.2.1 we can conclude that the classification scheme allow semantically rich traceability information to be captured, in the sense that it provided sufficient semantics to perform the test-cases in a satisfying way.

CS Requirement 3

The traceability classification scheme shall allow any artefact involved in the development process of the library system to be traced.

Fulfilled! Through the experiments conducted in subsection 7.2.1 we can conclude that the classification scheme allow any artefact involved in the development process to be traced – at least the one required to conduct the analyses in the test-cases-

CS Requirement 4

It shall be possible to identify what information is represented.

Fulfilled! Through the experiments conducted in subsection 7.2.1 we can conclude that the classification scheme makes it possible to identify what information is represented. This is achieved by using the ***ArtefactType*** *ExtensionTypes* for *TraceableArtefacts*. These extensions may not be sufficient for all projects, but allows some basic semantics to be added.

CS Requirement 5

It shall be possible to identify how the traced artefacts of the library system are represented.

Fulfilled! Through the discussion in subsection 5.4.1, we argued that the all the artefacts involved in the library example were either text-files or model elements. These are supported by the *TraceableArtefacts* ***ModelElement*** and ***TextFile*** in the classification scheme.

CS Requirement 6

It shall be possible to identify how an artefact was created*.*

Partly fulfilled! Through the discussion in subsection 7.2.1 we saw that the classification scheme made it possible to identify elements that were created by transformations. This is however not the only way an artefact may be created. We also know that some of the artefacts of the library example were created by developers using tool, e.g. the use-case model and the requirements document. Information regarding the creation of these cannot be captured using the classification scheme.

8.3 Discussion

In this section, we will continue the discussion on the traceability tool and the classification scheme conducted in the previous sections of his chapter and in chapter 7, by summarising the discussion and discussing the work with this thesis as a whole.

In chapter 3 we saw that different tools and users utilised traceability in different ways by using different languages to define traces, and utilising the traceability information for different kinds of analysis (i.e. *high-end* and *low-end* users).

The traceability tool and the classification scheme aims at supporting the needs of these different users in a way that allows them to operate on the same repository. In a perfect world, one might say that all users of traceability should use traceability in the same way. However as this is not the case it could make things easier if they could at least operate on the same repositories or exchange repositories.

By utilising the concepts presented with traceability tool and the classification scheme, different kinds of tools could use the interfaces provided by the tool to populate the same repository. E.g. a tool supporting manual trace creation could provide the functionality to define traces between text-files and models (any combination of the two) and use the traceability tool to store these traces in a repository as ***TextFile***s containing ***TextualArtefact***s or as ***ModelElement***s with ***Manual*** links between them. These artefacts might have been created by different transformation engines using ***FeatureTransformation*** links and/or ***ExplicitLink***s (as part of a ***Transformation*** composition) to record the relationships between them. The different tools operating on a trace repository would however not know about the different use of the traceability information; as far as they concern they just perform calls to an API that creates the traceability information for them.

Even though the different tools and users might not capture traceability information at the same level of details, they could all use the same trace repository to capture traces, or exchange traceability information as long as they use the proposed *traceTypeLib* model – the basic structure supporting linkage between different artefacts would be the same.

How this information may be utilised for analysis purposes however depends on how much semantics that is added through extensions. *High-end* users may however still use *low-end* traceability information by adding the required extensions to the information (by manual or automatic analysis) or just use the basic information as it is to get some understanding of the process. Utilising the traceability tool and the classification scheme to generate semantically rich traceability information at all stages of the development process makes it s possible to conduct many kinds of analyses on the development process as a whole.

8.3.1 Compliance With Existing Technology

The fact that the traceability tool is implemented as an EMF plug-in should make it possible to integrate the traceability tool with most EMF based tools, including modelling tools and transformation engines. In fact the generality of the traceability tool should make it possible to integrate it with most Eclipse plug-ins provided that EMF is installed.

Through the implantation of the *ManualTracer* (subsection 7.2.2) we have indeed shown that the traceability can be integrated with UML2 based modelling tools in Eclipse. This was done using a simpler version of the classification scheme presented in this thesis and should work just as well with the proposed classification scheme. We have however not tested the tool or classification scheme on a transformation engine. We have however discussed that most transformation engines use an internal traceability model, and some of them supports extended traceability usage. The classification scheme is also an extended version of the metamodel used by MOFScript. This should imply that it is possible to integrate the traceability tool and classification scheme also with transformation engines.

8.3.2 Criticism

The tested conducted to validate the traceability tool and classification scheme was performed on the relatively simple library example (Appendix A) and we it is therefore difficult to tell how the traceability tool may perform when a considerably larger amount of traceability information is generated, or if multiple users operate on the traceability tool simultaneously. The fact that no support for transactions is provided suggests that there might be trouble when multiple users use the same information.
Another problem that will become present on a real life project is that artefacts represented in the repository change over time. In such cases it would be desirable to create new versions of the artefacts in the repository in order to trace the evolution of artefacts. This might include linking the new version with the old version to specify the relation ship between them; no such semantics is however supported by the classification scheme. Analysing the traceability information when multiple versions exist might also be a challenge, and we have not tested how this could be performed.

The work with this thesis has furthermore been conducted without any contact with users of traceability, and we do therefore not know how the traceability tool or classification scheme suites their needs.

8.4 Summary

In this chapter we have discussed the traceability tool and classification scheme. In the beginning of the chapter we validated and discussed the fulfilment of the requirements for the traceability tool and the classification scheme. Except CS Requirement 6, all the requirements were fulfilled.

9 Conclusion

9.1 Summary

The motivation of this master thesis was the lack of a classification scheme for traceability in MDE. Although there are several different tools supporting traceability in various ways, there is a lack of a traceability classification scheme supporting the various needs of tools and users through the development process as a whole.

There are however no current implementation of a tool with support for definition and usage of such a classification scheme. The main objectives of this thesis were therefore to;

1. *Provide a tool that is capable of defining and handling semantically rich traceability information in MDE.*
2. *Find a suitable classification scheme for traceability in MDE, capable of capturing semantically rich traceability information, and emphasising automation.*

The traceability tool was developed as an EMF plug-in, based on two metamodels:

1. ***traceTypeLib*** – specifies a language in which traceability types can be defined.
2. ***traceRepository*** – specifies a language in which traceability information can be recorded using the types defined in the *traceTypeLib* model.

Based on the *traceTypeLib* metamodel we have created a GMF editor as a mean to define a concrete syntax for the language defined by the *traceTypeLib* model. This provides a user-friendly tool to define the traceability types. In order to allow external tools to use the traceability types defined in a *traceTypeLib* model to populate a *traceRepository* model, we have provided a Java interface which allows traceability information to be created programmatically. We have also provided support for generation of Java libraries from the *traceTypeLib* model comprised of classes representing the traceability types defined in the model. This allows developers to program with traceability types as regular Java classes created through a factory class.

In order to allow developers to brows the traceability information in a *traceRepository* model, we have utilised the EMF facilities to customise a generated EMF model editor, and created a view with support for basic navigation of trace links.

Utilising the GMF editor a basic classification scheme supporting traceability in MDE was created. It defines two basic traceable artefact types used to represent text-files and model elements in the *traceRepository* model in a way that allows trace links to be defined between them. Furthermore a set of basic trace link types was defined to allow trace links created manually, automatically or by transformations to be defined between the traceable artefacts. By using the concept of extension types, a set of types which may be used to extend the basic structure of traceable artefacts and trace links was defined. This makes it possible to dynamically extend the traceability information in the trace repository for added semantics.

Both the traceability tool and the classification scheme was validated through a set of test-cases related to the hypotheses defined in chapter 1, and bay validating the requirements specified in chapter 4 and 5. In addition the definition of the classification scheme serves as validation of the traceability tool, with respect to its ability to define semantically rich traceability classification schemes.

9.2 Claimed Contribution

9.2.1 The Traceability Tool

Through the work with this thesis we have presented a traceability tool built around two metamodels, where one defines the types used by the other, allowing traceability classification schemes to be generically defined, and used to populate trace repositories with traceability information. Similar approaches are suggested in [3, 10] and [9], but there are no references to an implementation of such approaches in a model based MOF compliant tool.

Our solution allows multiple libraries to of classification schemes to be used to populate repositories, thus allowing domain-specific or project-specific classification schemes to be used instead of or in addition to more general classification schemes.

By supporting composite structures of traceable artefact types and link types we are capable of defining a richer set of traceability types.

Additionally we have introduced the concept of extension types, which may be used as a mean to define extensions to the classes of trace link types or traceable artefact types. This allows different classes of fine grained semantics to be added and combined depending on the traceability strategy, and allows semantics be changed or added over time based on events or by performing analysis of the traceability information.

9.2.2 The Classification Scheme

The classification scheme support different kinds of usage by defining a basic structure of traceability types which may be dynamically extended by using extension types.

By treating all artefacts as ***ModelElement***s or ***TextFile***, we have provided a structure that allows all parts of artefacts of these types to be traced. This means that all artefacts may be the source and/or target of a transformation, and support both explicit and implicit linkage between them to be defined. This strategy hence supports tools with different traceability approaches.

The linkage between these artefacts may be of one of three basic link types; ***Transformation***, ***Manual***, or ***Automatic***, specifying how the link was created. The composition of the ***Transformation*** link type makes it easier to analyse traces that was created implicit or explicit by a transformation, as it groups the information together in a way that gives a good picture of the transformation.

By using information that is always present at creation time as basis for the link classification in combination with the basic traceable artefact types, we provide a basic structure that makes it possible to link the different artefacts in the development process together without

enforcing the addition of semantics that might add complexity to the process of creating traceability information.

To allow additional semantics to be added to the *TraceableArtefact*s we have defined a group of *ExtensionType*s with the supertype ***ArtefactType*** which may be added to the *TraceableArtefact*s. These types specify the semantics of an artefact at design level, e.g. ***Requirement*** or ***Design***, which may be used to perform a richer set of analyses.

The three basic link classes may also be extended with more fine grained information to ease analysis of the traceability information. There are two main classes of link extensions; *product-related* (***ProductRelated*** and its subtypes) and *process-related* (***Realizaiton***) which makes it easier to identify what links to follow when conducting certain kinds of analysis. The *product-related* link types are separated into three different subclasses (***Validation***, ***Verification***, and ***Explanation***) making it easier to navigate the traceability graph depending on what kind of analysis that is performed.

By separating between the basic traceability structure and the semantic sugar we have hence created a classification scheme which may be used differently by different users, depending on the traceability strategy. The same classification scheme may also be utilised differently by tools to capture traceability information in different ways. As long as the different users use the same *traceTypeLib* model they can still operate on the same trace repositories or exchange information.

9.3 Weaknesses

9.3.1 Maintaining Correctness of Traceability Information

A major weakness with the traceability tool is that it does not provide any functionality for keeping the traceability information up to date with the actual artefacts that are represented in the repository (e.g. if the name of a Class changes after it has been added to the trace repository, the *TraceableArtefact* representing this class is no longer valid). This is in fact a weakness with the concept of *TraceableArtefacts*, as they are not directly connected to the artefacts they are used to represent. This means that the responsibility of keeping the information current is left to the users of the traceability tool.

9.3.2 Extending Trace Type Libraries

One of the strengths with the traceability tool is that it allows several *traceTypeLib* models to be combined an extended (i.e. it is possible to extend extension types) by other *traceTypeLib* models, hence allowing customisation of libraries. Adding an *ArtefactExtensionType* in library B to a *TraceableArtefactType* in library A will however pollute library A with a reference to the *ArtefactExtenionType* in library B. It would have been better if this information was kept in library B. In some situations, it could also be desirable to be able to extend *TraceableArtefact*s and *TraceLink*s (i.e. generalisation) in other libraries.

9.3.3 Pollution of Mapping Code

The usage of ***ExplicitLink***s and ***TextualArtefact***s defined in the classification scheme assumes that the languages used to define mappings allow explicit links to be created, and that explicit blocks within text-files can be created (in model to text transformation languages). This means that the mapping code is polluted with traceability code, and decrease the reusability of the mappings [8].

9.3.4 Supporting Traceability Information Regarding Evolution

The traceability classification scheme does not allow much information about the evolution of artefacts represented in the trace repository to be traced, except that when transformations are involved. It would also be desirable to be able to capture creation of artefacts performed by e.g. modelling tools.

10 Related Research

Much of the related research is discussed in section 3.3. An overview of the related research is presented below.

- Ramesh and Jarke [9] discuss a generic metamodel and classification of traceability information for requirements traceability.

- Walderhaug et al [3, 10] discuss a traceability metamodel and system solution, supporting trace repositories where the traceability information is classified with generically defined traceability types. In [10] they also suggests a classification scheme for traceability in MDE by extending the metamodel. There is however no reference to an implementation of the solution or usage of classification scheme.

- Jouault [11] discusses how ATL can be used to generate traceability models with explicit links based on any traceability metamodels.

- Oldevik and Neple [24] discuss how traceability information can be automatically generated in model to text transformation using implicit links. The traceability information is described by a traceability metamodel that makes it possible to represent trace links between text-files model artefacts in external models.

- Olsen and Oldevik [8] discusses a how traceability information from model to text transformations generated by MOFScript can be used to perform various analysis on the traceability information. This is supported by a set of tools for MOFScript traceability models.

11 Future Work

In this chapter we discuss features that could be the basis for future work with the traceability tool and classification scheme.

11.1 Extending the Metamodels

In many cases, it would probably be advantageous with a more expressive language in which traceability information could be defined. Features like associations and generalization could e.g. be advantageous in many situations. One such situation is obvious when looking at the ***TextFile*** composition in the classification scheme. There are three *TraceableArtefactType*s recording positions using a set of *Attributes*. This could have been simplified a bit by using generalization or allow composite types.

11.2 Providing Better Support for Trace Type Libraries

In subsection 9.3.2 we discussed that capability for extending *traceTypeLib* models does not work optimally, as the model that is extended has to contain information about this. The fact that generalization of *TraceableArtefacts* and *TraceLinkTypes* is not supported also limits how libraries may be extended. More work should therefore be put into improving functionality for extending elements in a *traceTypeLib* model. Such functionality could allow developers to create project specific extensions to a standard library without having to add the information to the standard library itself. This would improve flexibility and reusability.

11.3 Extending the Classification Scheme

In subsection 9.3.4 we discussed that the classification scheme does not support much semantics regarding the evolution of artefacts except those created by transformations. More *LinkExtensionTypes* should therefore be added to support analysis with respect to evolution.

Further extensions or changes may also be found to be needed once the classification scheme is put into more use.

11.4 Complete Toolkit

One of the purposes of the traceability tool is to make it possible for different tools to use the same *traceTypeLib* models and the same *traceRepositories* to store traces. And hence get a more holistic traceability solution. A project for the future could therefore be to build a complete toolkit with support for definition of requirements (as models or text), manual trace creation, traceability in model to model and model to text transformation, and integration with a modelling tool to support automatic updating of traces to reflect changes and updates made in models. Using the traceability tool and the *MDE.traceTyopeLib* classification scheme to create traces, this toolkit could support many different kinds of traceability analysis, and provide functionality to provide consistency of the information over time.

11.5 Implement Support for Remote Repositories

Storing traceability information locally might not be desirable in large system development projects, as the traceability information will not provide the whole picture. It should therefore

be possible to keep trace repositories in external repositories on a server. Such use requires that a tool handles multiple users at the same time, and at the same time keeps the traceability in formation consistent. Such usage of trace repositories is discussed in [3, 10].

11.6 Maintaining Traceability Information Automatically

In subsection 9.3.1 we pointed out that a major weakness with the traceability tool is that it is not capable of maintaining the correctness of the traceability information, hence leaving this to the user of the tool (e.g. a developer using the traceability tool to achieve traceability support for a modelling tool). A subject for future work could therefore be to provide functionality to maintain such information automatically. E.g. [4].

12 References

1. Schmidt, D.C., *Model-Driven Engineering*. in Computer. February 2006: IEEE computer society: http://www.computer.org.

2. Falleri, J.-R., M. Huchard, and C. Nebut, *Towards a traceability framework for model transformations in Kermeta*, in *Second European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'06)*. 2006: Bilbao - Spain.

3. Walderhaug, S., et al., *Towards a Generic Solution for Traceability in MDD*, in *Second European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'06)*. 2006: Bilbao - Spain.

4. Aizenbud-Reshef, N., et al., *Operational Semantics for Traceability*, in *First European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'05)*. 2005: Nuremberg - Germany.

5. Antoniol, G., et al., *Problem Statements and Grand Challenges in Traceability: http://www.traceabilitycenter.org/files/COET-GCT-06-01-0.9.pdf. Center of Exelence for Traceability.* 2006.

6. Espinoza, A., P.P. Alarcón, and J. Garbajosa, *Analyzing and Systematizing Current Traceability Schemas*, in *NASA Software Engineering Workshop SEW-30*. 2006: Anaheim.

7. Limón, A.E. and J. Garbajosa, *The Need for a Unifying Traceability Scheme*, in *First European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'05)*. 2005: Nuremberg - Germany.

8. Olsen, G.K. and J. Oldevik, *Scenarios of Traceability in Model to Text Transformations*, in *Third European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'07)*. 2007: Haifa - Israel.

9. Ramesh, B. and M. Jarke, *Toward Reference Models for Requirements Traceability.* IEEE Tansactions on software engineering, 2001. **27**(1): p. 58 - 93.

10. Walderhaug, S., et al., *Traceability Metamodel and System Solution: http://www.modelware-ist.org/index.php?option=com_remository&Itemid=74&func=fileinfo&id=134*. 2006: Modelware.

11. Jouault, F., *Loosley Coupled Traceability for ATL*, in *First European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'05)*. 2005: Nuremberg - Germany.

12. *The Eclipse Modeling Framework (EMF)*. [cited 2007 10.24]; Web-Page]. Available from: http://www.eclipse.org/modeling/emf/.

13. Solheim, I. and K. Stølen, *Teknologiforskning - hva er det?* STF90 A06035. 2006: SINTEF Report.

14. Brown, A. *An introduction to Model Driven Architecture*. 2004 [cited 2007 October 10.]; Available from: http://www.ibm.com/developerworks/rational/library/3100.html.

15. Kleppe, A., J. Warmer, and W. Bast, *MDA explained The Model Driven Architecture: Practice And Promise*. Object Technology, ed. Booch, Jacobson, and Rumbaugh. 2003: Addison-Wesly.

16. Stephen J. Mellor, et al., *MDA Destilled Principles of Modell-Driven Architecture*. 1 st ed. Object Technology series, ed. Boch, Jacobson, and Rumbaugh. 2004: Addison-Wesley.

17. *Enterprise JavaBeans Technology*. [web page] [cited 2007 10.29]; Available from: http://java.sun.com/products/ejb/.

18. Tony Clark, et al., *Applied Metamodelling A Foundation for Language Driven Development*. 0.1 ed. 2004: Xactium.

19. OMG. *The Unified Modeling Language (UML)*. [Web-Page] [cited 2006 10.27]; Available from: http://www.uml.org/.

20. OMG. *The Object Management Group (OMG)*. [Web-page] [cited 2006 10.27]; Available from: http://omg.org/.

21. OMG. *the Meta Object Facility (MOF)*. [Web-Page] [cited 2006 10.27]; Available from: http://www.omg.org/mof/.

22. *Model Driven Architecture*. [Web-Page] [cited 2006 10.27]; Available from: http://www.omg.org/mda/.

23. Czarnecki, K. and S. Helsen, *Classification of Model Transformation Approaches.* OOPSLA 03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

24. Oldevik, J. and T. Neple, *Traceability in Model to Text Transformations*, in *Second European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'06)*. 2006: Bilbao - Spain.

25. OMG, *MOF 2.0/XMI Mapping Specification, v2.1: http://www.omg.org/docs/formal/05-09-01.pdf*. 2005.

26. OMG, *MOF QVT Final Adopted Specification: http://www.omg.org/docs/ptc/05-11-01.pdf*. 2005.

27. OMG, *MOF Models to Text Transformation Language Final Adopted Specification: http://www.omg.org/docs/ptc/06-11-01.pdf*. 2006.

28. *Wiktionary.org*. [Web-Page] [cited 2007 08.20]; Available from: http://www.Wiktionary.org.

29. IEEE, *IEEE Standard Glossary of Software Engineering Terminolgy*, in *IEEE Std 610.12-1990*.

30. Behrens, T., *Never 'Without a trace': Practical advice on implementing traceability:* *http://www.ibm.com/developerworks/rational/library/feb07/behrens/*. 2007.

31. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. polack, *On-Demand Merging of Traceability Links with Models*, in *Second European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'06)*. 2006: Bilbao - Spain.

32. *The ATL home page*. [Web-Page] [cited 2006 10.27]; Available from: http://www.sciences.univ-nantes.fr/lina/atl/.

33. Vanhoof, B., et al., *Traceability as Input for Model Transformation*, in *Third European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'07)*. 2007: Haifa - Israel.

34. Angyal, L., L. Lengyel, and H. Charaf, *An Overview of the State-of-The-Art Reverese Engineering Techniques.* 7th International Symposium of Hungarian Researchers on Computational Inteligence: p. 507-516.

35. Aizenbud-Reshef, N., et al., *Model Traceability.* IBM Systems Journal, 2006. **45**(3): p. 515-525.

36. Telelogic, *Doors: http://www.telelogic.com/corp/products/doors/index.cfm.*

37. IBM, *RequisitePro: http://www-306.ibm.com/software/awdtools/reqpro/.*

38. *Eclipse home page*. Web-Page [cited 2007 10.30]; Available from: http://www.eclipse.org/.

39. *XML Metadata Interchange (XMI)*. [Web-Page] [cited 2007 10.27]; Available from: http://www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI.

40. *Graphical Editing Framework GEF*. [Web-Page] [cited 2007 10.27]; Available from: http://www.eclipse.org/gef/.

41. *Graphical Modelling Framework (GMF)*. [cited 2007 10.25]; Web-Page]. Available from: http://www.eclipse.org/gmf/.

42. *UML2 Project*. [Web-Page] [cited 2007 10.25]; Available from: http://www.eclipse.org/modeling/mdt/?project=uml2.

43. ModelPlex. *ModelPlex*. [Web-Page] [cited 2007 10.30]; Available from: http://www.modelplex-ist.org.

44. Papyrus. *Papyrus UML*. [Web-Page] [cited 2007 10.30]; Available from: http://www.papyrusuml.org.

Appendix A – The Library Example

Introduction to example

The example starts with a simple requirements document, which is the basis for a use-case model. This model is transformed to a class model (Figure 4), which is refined by developers. The PIM is then transformed to an EJB-PSM (Figure 5) which finally is transformed to Java code. These are described in more detail in chapter 3.

Throughout this thesis we will use a running example of a simple library system as a means to illustrate our discussion and to provide test data for our experiments. The example will eventually provide test data from artefacts throughout a software development process – from requirements to implementation.

The aim of the library system is to support the librarian in his/her daily tasks. The system is divided into two subsystems *BookSystem* and *CustomerSystem*.

Requirements document

To find out what they need to create, developers sit down with librarians and discuss what they want from the system. These specifications are captured by the Requirements document:

”

Requirements for SuperLib2000

These are the requirements for the new Library system – SuperLib2000.

Non-functional requirements:

The system should mostly be up and running and it should at all times be an excellent library system.

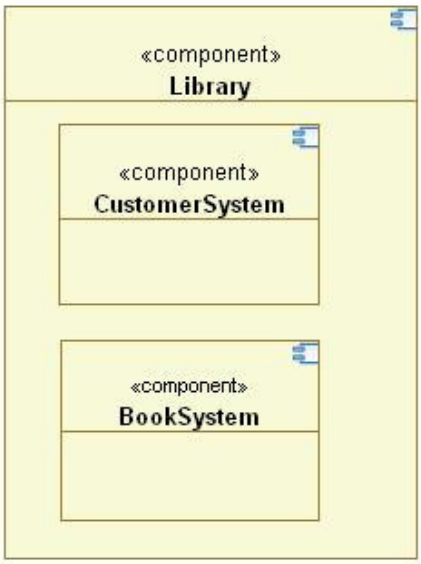
Functional requirements:

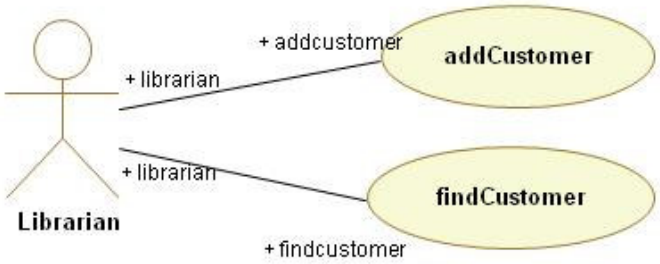
- *The system shall make it easier for the librarian to find books.*
- *The system shall make it easier for the librarian to find customers.*
- *The system shall have functionality to add new books, and customers.*
- *The system shall have functionality to remove books and customers.*

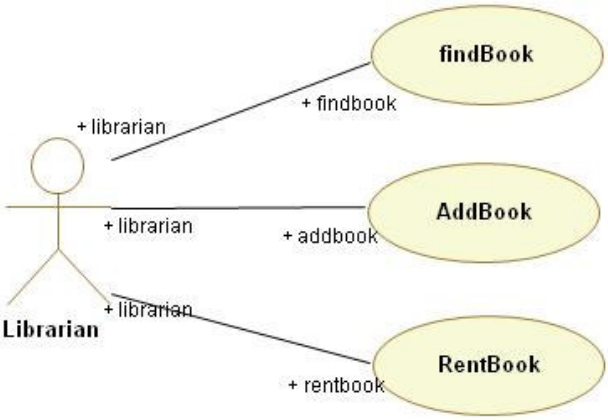
“

Use-case Model

To capture their requirements formally developers creates a use-case diagram of the system, with the use-cases described in the functional requirements. In the use-case model the system is divided into two subsystems – BookSystem to handle books, and CustomerSystem to handle customer related information.


«component»
Library
«component»
CustomerSystem
«component»
BookSystem


+ addcustomer
addCustomer
+ librarian
+ librarian
findCustomer
Librarian
+ findcustomer


findBook
+ findbook
+ librarian
+ librarian
AddBook
+ addbook
+ librarian
Librarian
RentBook
+ rentbook

The PIM

When developers are finished with the use-case model, they transform it into an architecture model describing a more precise architecture of the system. This model consists classes for each of the subsystems in the use-case model.

Transformation

The logic behind the transformation is quite simple, and works as follows:

- For each of the *Component*s in the use-case mode a *Class* with the same name is created in the architecture model.
- If a *Component* is contained in another *Component* a *Property* is created in the *Class* that is transformed fro the parent *Component*. This *Property* has the same name as the child *Component*, and is typed with the *Class* that is transformed from the child *Component*.
- For each of the *UseCases* in the use-case model an *Operation* with the same name is created in the architecture model, contained in the *Class* that is transformed from the *Component* that contained the *UseCase*.
- If the *Component* containing a *UseCase* in the use-case model is contained in another *Component*, each of the *UseCase*s in the child *Component* is transformed to *Operation*s in the *Class* that is transformed from the parent *Component*.

Refinement

After transforming the Use Case model to the PIM, the developers have to manually add the features that cannot be created by the transformation. The result of the refinement is shown in Figure 4.

Rationale document

After completing the architecture model, a rationale document is created to explain the architecture model:

”

Rationale

The library system is a system aimed at making the job of a librarian easier. It concists of two subsystems; 1) BookSystem, with support for finding, adding, and renting books, 2) and CustomerSystem with support for adding and finding customers.

The main artefacts of the system are books, and customers, both identified by a name. In addition information about the age of the customer is needed, as some books have an age restriction.

“

The EJB-PSM

The EJB-PSM is transformed from the PIM using an ATL transformation. The basics of the transformation are as follows.

- Classes annotated with stereotype ‘Service’ are mapped to classes with the same name, but with the stereotype ‘EJB_Service’.
- Classes annotated with stereotype ‘PersistentObject’ are mapped to classes with the same name, but with the stereotype ‘EJB_Entity’.
- All other classes are mapped to a new class with the same name, but without any stereotypes, regardless of any stereotypes contained by the input class.
- All attributes and operations contained by the input are mapped directly to attributes and operations with the same name, type, and signature as those contained by the input class. If an attribute in the input model is annotated with the stereotype ‘Id’, the corresponding attribute in the output model will be annotated with the stereotype ‘EJB_ID’.

The result of the mapping is shown in Figure 5.

Implementation

After completing the refinement of the PSM the developers use a MOFScript transformation to generate Java code for the PSM. The mapping is straight forward, and creates a Java class for each Class in the model.

Appendix B – Source Code

The source code for the traceability tool, the test cases or the mappings used for the library example will be made available upon direct request to the author or supervisor via e-mail. Please contact the following:

- Svein Johan Melby (author), smelby@gmail.com
- Gøran K. Olsen (supervisor), goran.k.olsen@sintef.no