View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NORA - Norwegian Open Research Archives

UNIVERSITY OF OSLO
Department of Informatics

Construction of Information Repositories for Managing Standards Compliance Evidence

Master thesis

Torbjørn Skyberg Knutsen

April 29, 2011

Acknowledgements

Thanks to the University of Oslo and Simula Research Laboratory. A special thanks to Rajwinder Kaur Panesar-Walawege, Shaukat Ali and Mehrdad Sabetzadeh.

Abstract

Safety-critical systems are often subject to certification in order to assure the public that they will not cause harm to either people or the environment during their use. Such certification is usually based on some industry specific standards; in the embedded systems domain, the most common standard for functional safety in the IEC61508 standard. Evidence from the attempted use of the standard has shown that using IEC 61508 has been met by difficulties in understanding the standards' scope, purpose and content, as well as and the need for an infrastructure for using the standard.

Panesar-Walawege et al. [1] provide in the form of a conceptual model a detailed description of the information that needs to be preserved during the development of safety-related software, based on information found in the IEC 61508 standard. This work tackles some of the issues concerning the understanding of the scope and content of the standard, and understanding how to use it. This thesis describes a concretization of this conceptual model, in the form of an information repository built on a relational database. The information repository is generated from the conceptual model, through the use of model driven technologies and model transformations. The work described in this thesis provides developers of safety-related software the possibility of storing the evidence information required for compliance with the IEC 61508 standard, in order to aid the certification of their software systems. It is the beginnings of the infrastructure required to use the IEC61508 standard effectively.

Table of Contents

1. Introduction 7
2. Background 9
3. Program description 11
   3.1 Information Repository 13
      3.1.1 Persistence Layer 15
      3.1.2 Logic Layer 17
      3.1.3 View Layer 19
   3.2 Model Transformations 23
      3.2.1 RDBMS Meta-Model 23
      3.2.2 Model-To-Model Transformation 29
      3.2.3 Model-To-Text Transformations 35
   3.3 User Roles 39
4. Discussion 41
   4.1 Design Choices 41
      4.1.1 Creation of a Repository from Scratch 41
      4.1.2 Use of Model-Driven Technologies 42
      4.1.3 Use of an Intermediate Model 43
      4.1.4 UML to Relational Database Transformation Rules 44
      4.1.5 Tool Selection 46
      4.1.6 Constraints on the Database 47
      4.1.7 Using a Web-based User Interface 47
   4.2 Lessons Learned 49
      4.2.1 Software Development - More Than Programming 49
      4.2.2 Learning about the Tools 49
      4.2.3 Tool Problems 50
      4.2.4 Loading Input UML Models with Kermeta 50
      4.2.5 Setting up the Required Libraries 51

5. Implementation.. 53
5.1 UML.. 55
5.2 Eclipse.. 55
5.3 Kermeta... 56
5.4 MOFScript.. 57
5.5 Java... 57
5.6 Apache Maven.. 58
5.7 Apache Derby... 58
5.8 Hibernate... 59
5.9 Spring.. 60
5.10 Apache Struts... 60
5.11 Apache Tomcat... 61
5.12 JavaScript... 62
5.13 SiteMesh... 62
5.14 JavaServer Pages (JSP).. 63
6. Conclusion.. 65
Appendix A... 67
A.1 Learning Materials... 67
A.1.1 Kermeta.. 67
A.1.2 Hibernate.. 68
A.1.3 Struts.. 68
A.2 Example of use.. 69
A.2.1 Model Transformations... 69
A.2.2 Information Repository.. 87
List of Figures... 99
References... 101

1. Introduction

The motivation for the work described in this thesis is to enable the development of information repositories for managing safety evidence for safety critical systems. The thesis draws on earlier work by Panesar-Walawege et al [1], where a model-based solution is provided for characterizing the safety evidence information required by IEC 61508 [2] - a widely used standard for functional safety of electrical/electronic/programmable electronic systems. Specifically, we develop a tool infrastructure to transform an evidence information model into a database schema, a user interface to populate the schema, and an automated check of the consistency between the evidence information model and the user-input data of the repository. The tool utilizes various model-driven technologies as we are going to explain throughout the thesis.

The creation of a repository for storing information that demonstrates compliance with the IEC 61508 standard reflects a direct industrial need, as the largely textual form of the standard makes it hard for developers of safety critical software to know exactly what information to record.

The repository is created in a generic manner such that if the underlying model were to be modified, then the repository can be automatically regenerated. Hence the method we provide can be used not only in the specific case that we discuss in this thesis but also as a general-purpose tool for automating the creation of repositories based on UML [3] class diagram descriptions of data models.

The remainder of this thesis is structured as follows: We describe some of the background on IEC61508 and the use of relational databases in information repositories in Chapter 2. Chapter 3 contains a description of program code resulting from the work described in this thesis, both the model

transformations and the information repository itself. In Chapter 4, design choices and lessons learned during the development are discussed, and Chapter 5 contains brief descriptions of the tools and technologies used during the development. We conclude the thesis in Chapter 6. Guidelines on how to use the model transformations and the information repository, as well as an overview of the written materials used during the course of this project, are provided in the appendix.

2. Background

IEC 61508 is an international standard published by the International Electrotechnical Commission (IEC) [4], titled "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems". The standard is concerned with improving the development of safety-related systems, whose failure might lead to harm to people, equipment and/or the environment. The standard is generic, and can either be used directly, or as a basis for the creation of domain-specific standards in industries that require a level of safety equivalent to that described in the standard. IEC 61508 is concerned with functional safety; its goal is to ensure that safety-related systems operate correctly in response to their inputs.

The standard adopts an overall safety lifecycle, in order to systematically deal with the necessary activities for achieving the required level of safety. During this lifecycle, the standard imposes a number of verification, management and assessment activities. The software for a system must be implemented in a way that fulfills the safety requirements allocated to it. In order to be able to show that this has been done, it is important to maintain traceability between the safety requirements, the decisions taken during the design of the software, and the actual implementation of the code. This complex task needs to be performed as the system is being developed, rather than once the development is finished. [1]

In his article “Installing IEC 61508 and Supporting Its Users - Nine Necessities” [5], Felix Redmill proposes nine necessities for the successful use of the IEC 61508 standard. Evidence from the attempted use of the standard has shown that using IEC 61508 has been met by difficulties, and even “attempts to read and understand the standard seem to have had a low success rate.” Among the necessities noted by Redmill are understanding the

standards scope, purpose and content, knowing how to apply the standard, as well as providing an infrastructure for using the standard.

Panesar-Walawege et al. [1] provide in the form of a conceptual model a detailed description of the information that needs to be preserved during the development of safety-related software, based on information found in the IEC 61508 standard. This work tackles some of the issues raised by Redmill, both when it comes to understanding the scope and content of the standard, and understanding how to use it. The work described in this thesis provides a concrete way of gathering the information described in the conceptual model, and can be thought of as a step towards the infrastructure requirement proposed by Redmill.

A common way of organizing and storing data is through the use of a database. A database consists of a collection of data, and is intended to organize, store, and retrieve large amounts of data in a simple manner. There are a few types of databases, of which the relational databases are the most widely used. In a relational database, data is stored in relations, which can be viewed as tables, where each row corresponds to an entity, and each column to an attribute of the entity. For an information repository, using a relational database is advantageous because it is a mature and widely used technology, and has the ability to handle large amounts of data.

3. Program description

The work done in this thesis project can be divided into two main parts; (1) the creation of a set of model transformations that generate artifacts of an information repository, based on a UML class diagram, and (2) the creation of the information repository itself, consisting of a relational database and a web-based user interface that allows the user to manipulate the data in the relational database. Some of the artifacts of the information repository are generic (usable for any input class diagram), while others are the end products of the model transformation. The word “repository” is from hereon used to denote the software system consisting of the relational database and user interface used to manipulate the database.

The model transformations consist of a model-to-model transformation, (written in Kermeta [6]) that takes as input a UML class diagram, and produces as output an intermediate model corresponding to a meta-model that describes the structure of a database schema, as well as several model-to-text transformations (implemented in MOFScript [7]), that generates a number of text files containing code to be used as part of the repository, based on the intermediate model.

The repository is based on a relational database, and also includes a web-based user interface, allowing the user to perform basic create, read, update, and delete (CRUD) operations on the database. The repository is implemented in Java [8], and utilizes a number of technologies, most importantly Hibernate [9] and Struts [10]. Hibernate is an Object-Relational Mapping (ORM), which is used to map relational database tables with classes, in order to mimic an object-oriented database system. Having an object-relational layer in the form of an ORM is necessary because we use an object-oriented approach, both for the specification of data content (through

UML class diagrams), and for the implementation of the information repository.

An ORM allows a direct integration of a relational database in an object-oriented system, by allowing the entries in the database to be treated as objects. The mismatch of how data is represented in an object-oriented system versus a relational database normally leads to a significant increase in development cost, as well as repetitive and tedious coding work in order to integrate the two types of systems. Using an ORM reduces the significance of this mismatch, by allowing the integration of an object-oriented system and a relational database to be achieved with significantly reduced development time, and fewer lines of code. Hibernate provides a buffer between the two ways of representing data, and allows the more elegant use of object-orientation while at the same time keeping the relational database schema normalized and guaranteeing data integrity [11].

Struts is used as a mapping between objects and the user interface, as it allows objects and their attributes to be viewed and manipulated through a web-based user interface in a simple manner.

A description of the technologies used can be found in Chapter 5.

3.1 Information Repository

The information repository consists of artifacts divided into two main categories; (1) *generic* artifacts, which are independent of the structure of the database schema, and (2) *generated* artifacts, which are based on an input UML class diagram, containing code for initiating and manipulating the concrete database tables defined through the model transformations. The information repository is implemented using a three-tiered architecture. Figure 1 shows the architecture of the repository, as well as the main artifacts, separated into generic and generated artifacts.

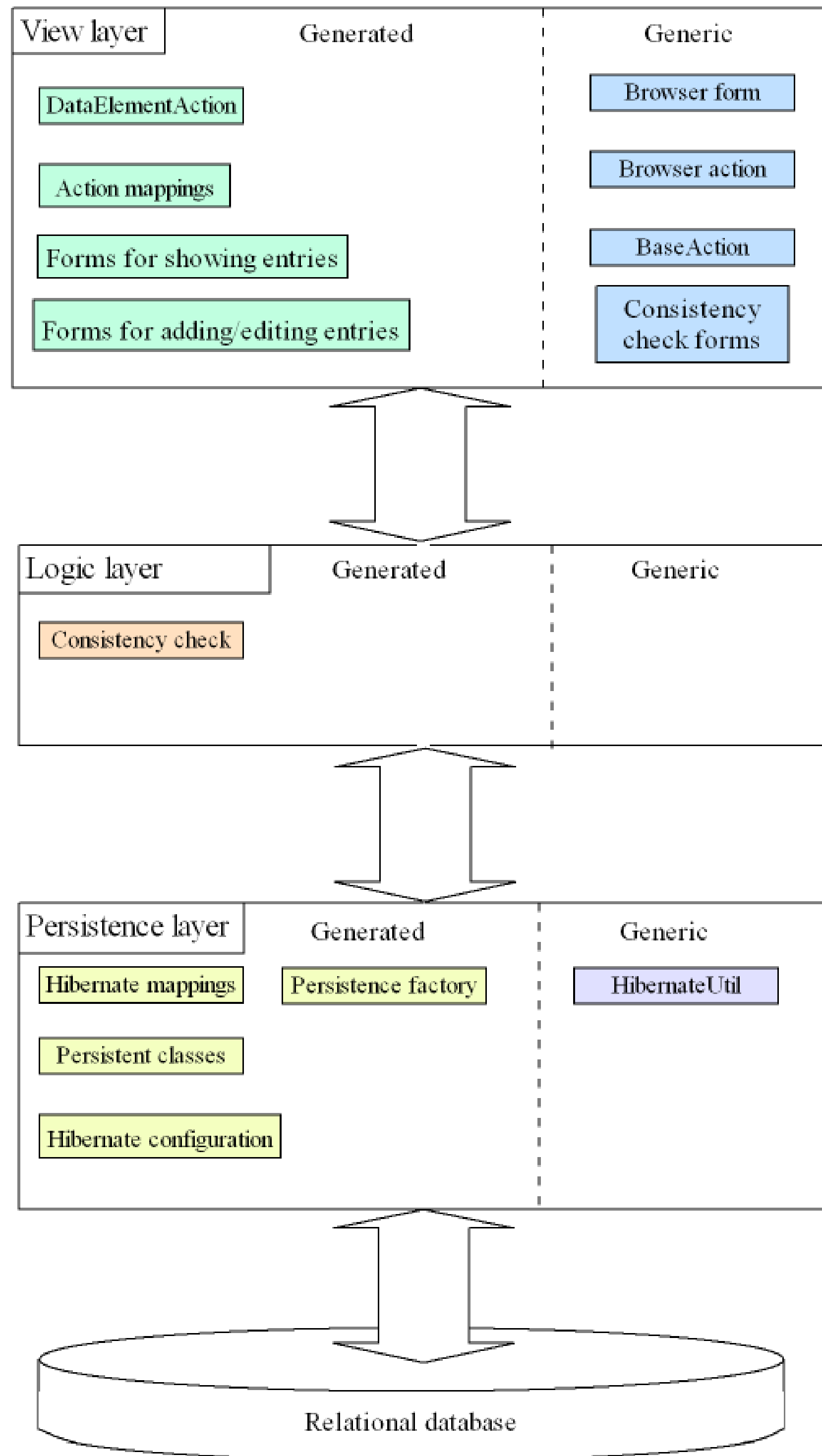


Figure 1: The architecture of the information repository.

At the bottom is an Apache Derby relational database, which is used to store the data. Apache Derby is an open-source relational database, implemented entirely in Java. It has a small footprint, only 2.6 MB, and supports embedded mode, allowing it to be embedded in any Java-based solution. [12] On top of this, we find the Persistence or Data layer, which contains artifacts associated with Hibernate, used for the Object-Relational Mapping (ORM), as well as some helper functions for accessing and manipulating entries in the database. On top of the persistence layer, is the Business or Logic layer, which in this case is fairly small, containing only the consistency check. The top layer is the View or Presentation layer, which contains artifacts that facilitate the interaction with the user. This layer contains artifacts associated with Struts and JSP [13].

3.1.1 Persistence Layer

The persistence layer provides functionality for the communication with the relational database.

3.1.1.1 Persistent Classes

The persistent classes are Java classes that are generated from the input UML class diagram. Each persistent class corresponds to a table in the database, and each attribute of the persistent class correspond to a column. The persistent classes also hold set- and get-methods for each of the attributes, as well as some helper methods used by the presentation layer.

3.1.1.2 Hibernate Mappings

Hibernate mappings are XML files used to map each of the persistent classes to a table in the database. The mapping files also map each attribute of the persistent class to a column in the table, and define associations between

tables. The Hibernate mappings describe the structure of the relational database schema, and can be used by Hibernate to generate the SQL-statements necessary for initializing the database.

3.1.1.3 Hibernate Configuration

The Hibernate configuration file (hibernate.cfg.xml) is an XML file that defines the properties of the database connection, e.g. which database type to connect to, what SQL dialect to use, the name of the database to connect to, whether to connect to an already existing database or create one at program start-up. The Hibernate configuration file also contains file paths to the Hibernate mapping files, allowing Hibernate to read these on program start-up (and if specified, generate database tables based on them).

3.1.1.4 HibernateUtil

HibernateUtil.java is a generic helper-class, used to abstract away parts of the communication with the database. HibernateUtil holds a Hibernate SessionFactory, which is used to request new sessions from Hibernate, and provides methods that handles the sessions and transactions required by Hibernate for database manipulation. Calls to methods in the HibernateUtil can be made in order to achieve less complex method calls to Hibernate functionality for the rest of the system.

3.1.1.5 Persistence Factory

The persistence factory (PersistenceFactory.java) provides methods for retrieving one or all elements in a table, as well as for deleting an entry in a table. The persistence factory abstracts away any Hibernate Query Language (HQL) [14] queries from the rest of the system, allowing it to view the database as a collection of Java objects, retrieved by simple method calls. The persistence factory relies heavily on HibernateUtil, as all requests to the

database are passed through methods of HibernateUtil, to avoid the repetitive writing of method calls needed to communicate with Hibernate.

3.1.2 Logic Layer

The Logic layer of the system is fairly small, containing only the Consistency check.

3.1.2.1 Consistency Check

The consistency check (ConsistencyCheck.java) contains methods for verifying that the state of the database is in accordance with the multiplicity constraints defined in the UML class diagram that formed the basis for the database. The consistency check is derived from the multiplicities of UML associations. These multiplicities are not being preserved in the database schema itself, where all associations are represented as many-to-many, which is the least stringent constraint. This allows more freedom for the user when adding entries in the database, but also calls for the implementation of a consistency check, to verify that the data in the database is in accordance with the constraints defined in the UML class diagram. The consistency check is composed by a set of tests on the tables that are derived from UML associations, to check that the multiplicities of the associations are upheld. For instance, a table corresponding to an association with an upper bound of 1 on one of its connector ends should contain at most 1 reference to each instance of the referenced persistent class. If a persistent class is referenced more than once in a column with a maximum multiplicity of 1, it is inconsistent with the specification in the UML class diagram. If the minimum multiplicity is 1 on a connector end, each instance of the referenced class should appear as part of an entry in the table corresponding to the association, and an instance that it not referenced causes an inconsistency.

The consistency check also checks for “dangling” entries in the tables corresponding to UML associations, to make sure that both sides of the association are defined. The detected inconsistencies in the database are collected in a report, which is presented to the user once the consistency check is finished. The consistency check is written in Java, and utilizes the ability of the persistence factory to return all the entries of a table as a collection of Java objects.

3.1.3 View Layer

The view layer contains functionality for user interaction, allowing the user to perform CRUD (Create, Read, Update, Delete) operations on the database, as well as running the consistency check, through a web-based user interface.

3.1.3.1 Action Mappings

The action mappings are artifacts of Struts. The mappings are found in the XML-file struts.xml. An action is a link between HTML and Java code, which allows Java code to be executed by calls made from a HTML page. An action mapping defines the name of an action, maps this to a method within a Java action class, and optionally defines what to present upon the completion of the execution of the method in the action class, depending on the return value of the called Java method.

3.1.1.2 Browser Action and Form

The browser action class (Browser.java) is a simple action that retrieves a list of strings containing the names of all the tables in the database. This list is used in a JSP-form called Browser.jsp, which presents the table names in a select box. When a table name is selected, a HTTP GET-request is carried out through a JavaScript function, and the entries of the selected database table are injected into the browser page. Similarly, if the user wants to create a new entry in the selected database table, a click on the “New”-button of the browser page causes a form for adding an entry in the database table to be injected into Browser.jsp.

3.1.1.3 Forms for Showing, Adding and Editing Database Entries

There are two main types of web-forms generated from the input UML class diagram; (1) forms for showing the entries of a table, and (2) forms for adding

or editing an entry in a table. Since tables differ with regards to their attributes, different forms are needed for the different tables. The forms for showing database entries creates a HTML table based on the entries residing in the database tables, which are obtained through a call to a Struts action. The collection of database entries is iterated, and presented in the form of a HTML table, containing information from the columns of the database table. Forms for adding/editing also differ from table to table, and provide a way for the user to input the attribute values of the entry to be added. Upon editing an existing entry, the entry is fetched and its attribute values are displayed in the same form that is used for adding new entries. These forms are implemented using JSP.

3.1.1.4 DataElementAction

DataElementAction.java is an action class used by most of the Struts actions. It contains methods used by the user interface, both for adding entries in the database, as well as for editing and deleting already existing elements. This is achieved by issuing method calls to the persistence factory, storing the return values in variables that are made available to the JSP-forms through Struts.

3.1.1.5 BaseAction

The BaseAction (BaseAction.java) is a simple action class that acts as a parent for the other action classes. This class holds a reference to the persistence factory, which can be used by the action classes extending BaseAction. BaseAction also contains some standard methods for Struts actions. The class extends ActionSupport, allowing it, and all its subclasses, to be treated as action classes.

3.1.1.6 Consistency Check Action and Forms

The consistency check action class contains a method that make a call to the consistency check to start checking. The result of the consistency check is stored in a collection of strings, which is made available to a form that presents the results of the check to the user. The user interface of the consistency check consists of two JSP-forms, one containing the means for initiating the check, and one for displaying the results of the consistency check.

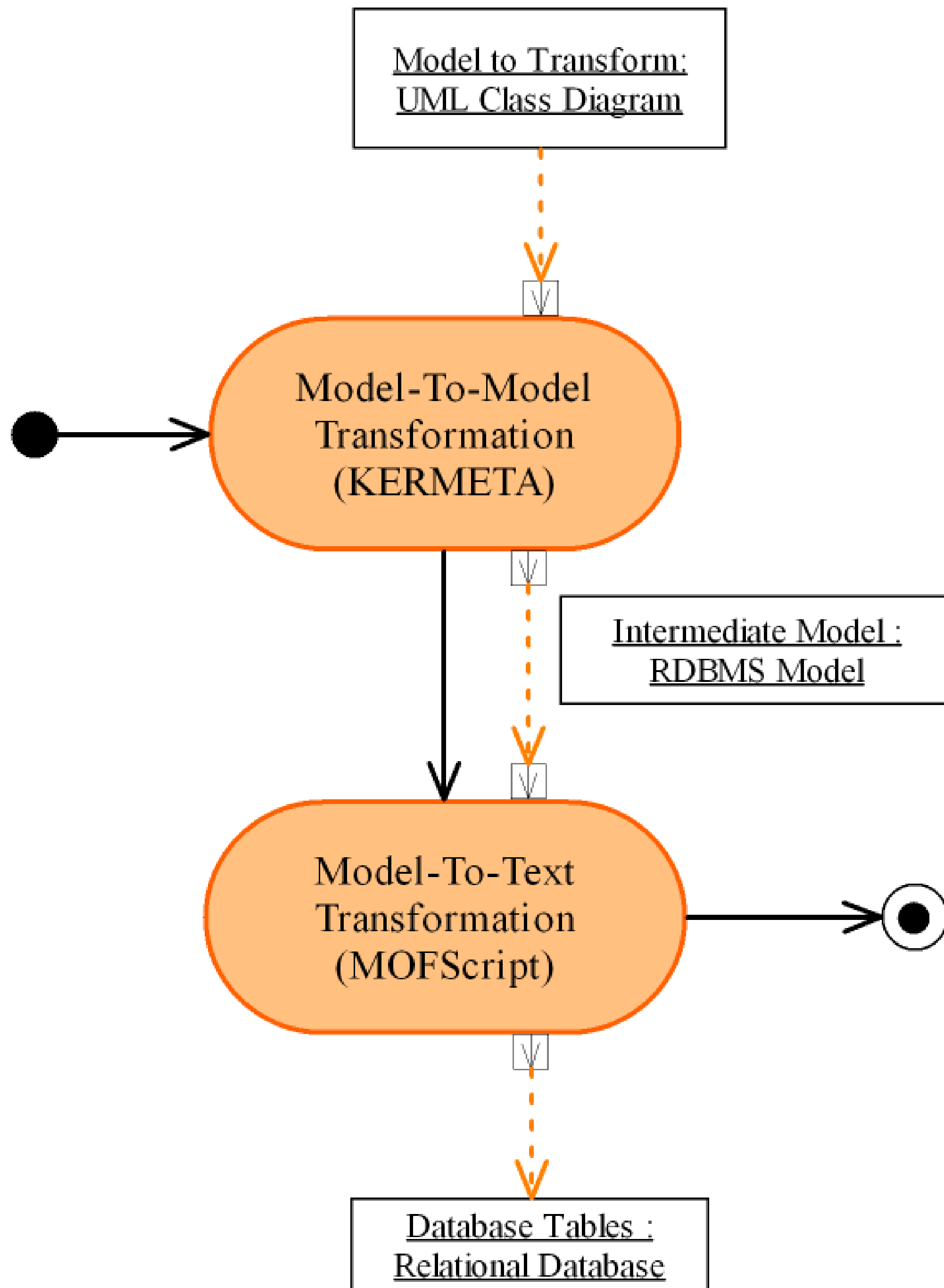


Figure 2: The main program flow of the model transformations.

3.2 Model Transformations

This section contains a description of the model transformations used to generate artifacts of the information repository, from a UML class diagram. Figure 2 describes the main program flow of these model transformations.

The input to the model transformations is a UML class diagram (see Chapter 3.2.2.1 for a full description of how the input model should be formed.) This diagram is transformed by a model-to-model transformation into a model corresponding to the structure of a relational database. This model is in turn used as input for several model-to-text transformations, whose output text files compose the generated parts of the repository. We employ Kermeta for the model-to-model transformation and MOFScript for the model-to-text transformations.

3.2.1 RDBMS Meta-Model

The RDBMS meta-model (RDBMSMM.ecore) is the meta-model of the intermediate model, which is generated by the model-to-model transformation, and used as input by the model-to-text transformations. The meta-model describes the structure of a relational database, with table and column-elements, as well as elements representing other constructs of a relational database, like primary keys and foreign keys. The use of the intermediate model allows for the most complex logic of the model transformations, the implementation of the transformation from class diagram to the structure of a relational database, to be abstracted away from the production of text files. The intermediate model can be used as a basis for several model-to-text transformations, leading to the generation of more than just the description of the database structure.

The RDBMS meta-model is based on a meta-model used in a Kermeta example [15], which transformed a simple class diagram into a basic relational database structure. This meta-model was augmented with new elements as they were needed, when creating the model-to-model transformation. The original meta-model from the Kermeta example contained the basic elements of the RDBMS meta-model; RDBMSModel, Table, Column and Fkey, as well as the association defining the primary key of a table. We have extended this model to fit the requirements of our model transformation, and introduced all the attributes shown in Figure 3 (except for "name" in Table and Column, and "type" in Column), as well as the association for defining subclasses. The structure of the meta-model can be seen in Figure 3.

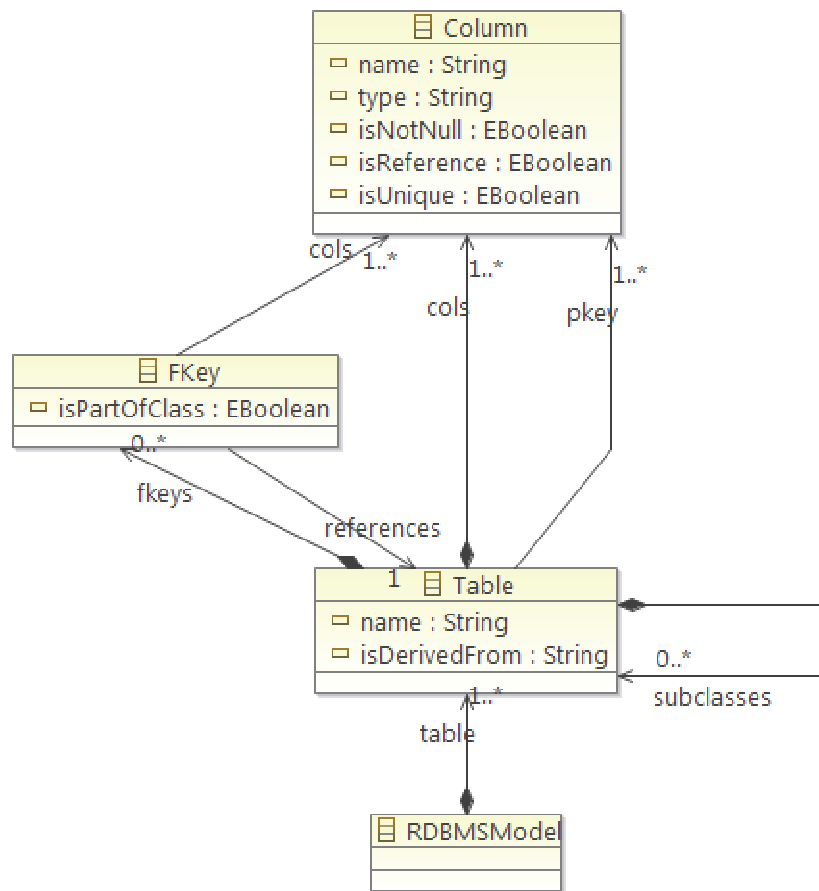


Figure 3: RDBMSMM.ecore.

The top-level container element of the meta-model is RDBMSModel, and this contains the table elements that compose a database schema. The “Table”-elements have a name, as well as a property “isDerivedFrom”, which indicates the type of the UML element that served as the basis for table. A table may contain a number of columns, as well as a reference to one or more columns as primary key. A table can also contain foreign keys, which consists of a reference to one or more columns, and a reference to a table. The “isPartOfClass” attribute of the Fkey element is used to separate the two sides of an association between two tables, since they will be implemented differently. A table can contain a number of other tables, which is the way that class inheritance is represented, by a table corresponding to a subclass being contained within its superclass. Column elements have a name and a type, as well as some Boolean flags which are used to indicate whether the column contains a reference to another table and to represent multiplicities when the columns represent connector ends in UML Associations.

An example of the relation between a UML class diagram and a RDBMS model follows.

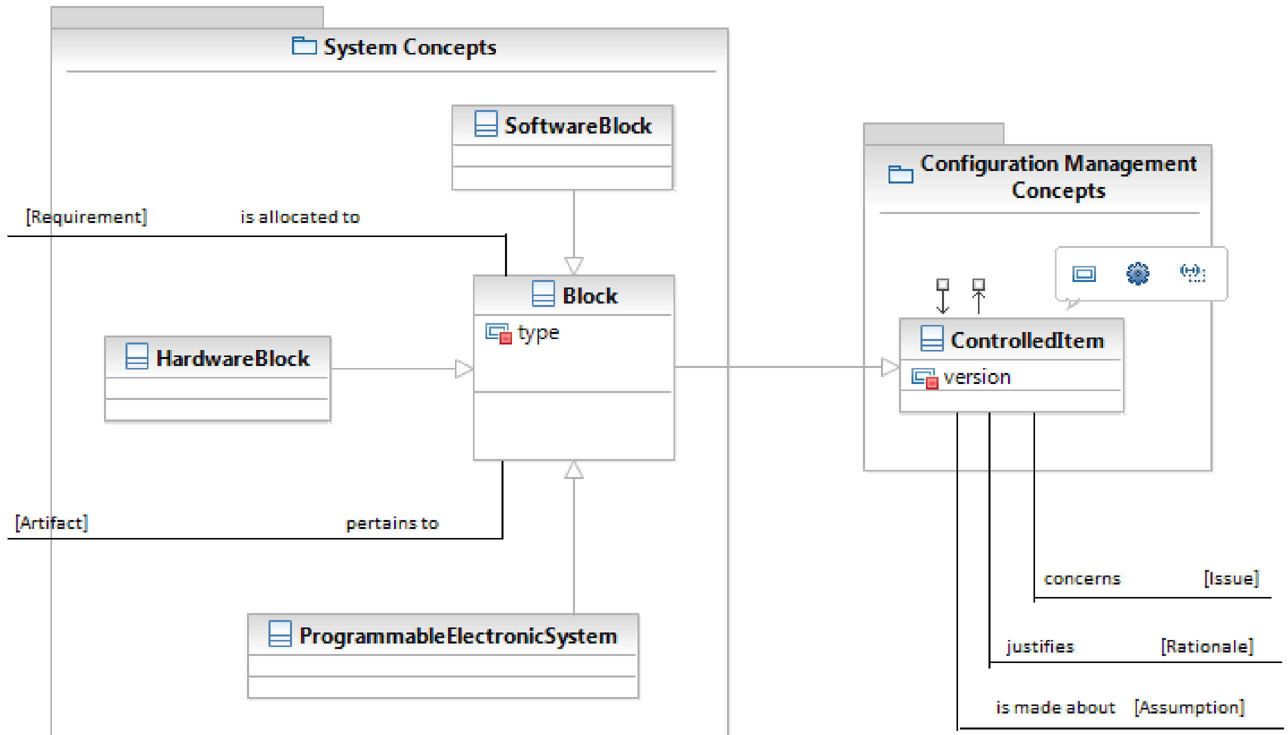


Figure 4: Parts of a UML class diagram.

This UML class diagram in Figure 4 contains four classes in a class hierarchy. The “ControlledItem” class has an attribute “version”, and is part of three associations; “justifies”, “concerns” and “is made about” (the classes of the opposite ends are omitted, their names appear in brackets). “Block” is a subclass of “ControlledItem”, and contains an attribute “type”. In addition, “Block” is part of the associations “pertains to”, and “is allocated to”. “HardwareBlock”, “SoftwareBlock” and “ProgrammableElectronicSystem” are all subclasses of “Block”. Figure 5 shows the instance of the RDBMS meta-model that corresponds to the example UML class diagram.

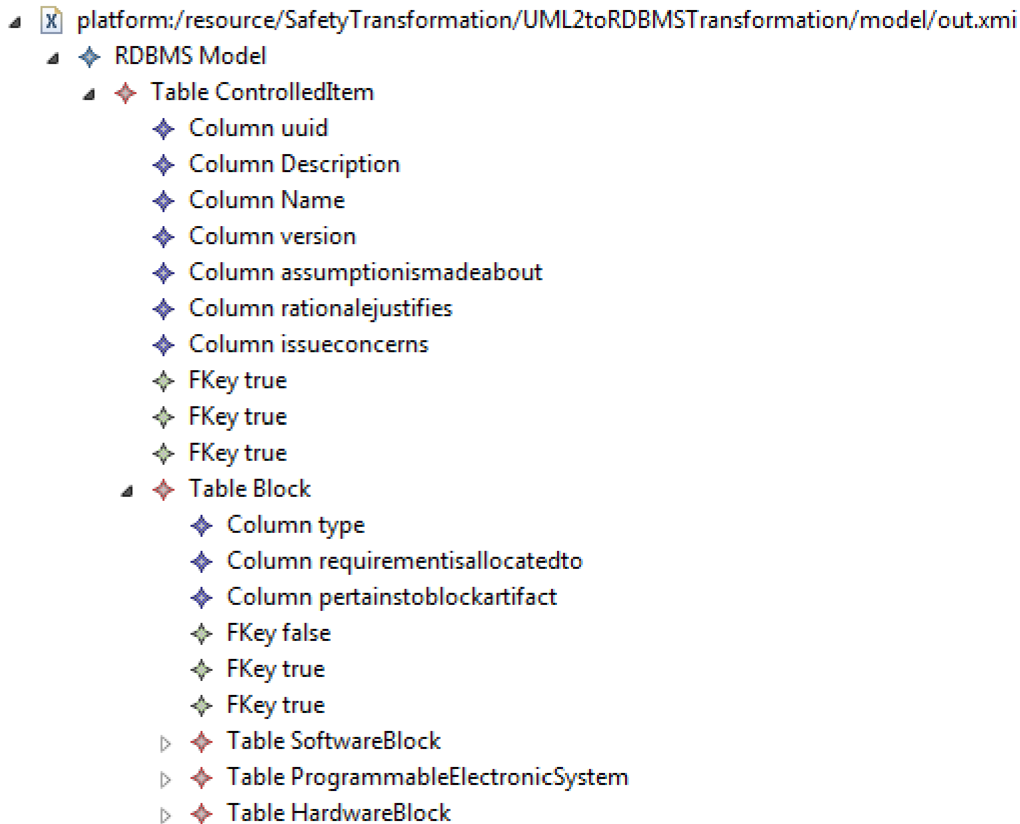


Figure 5: Parts of an instance of RDBMSMM.ecore.

In accordance with the description of the meta-model above, the corresponding example instance of the RDBMS meta-model, shown in Figure 5, has an “RDBMS Model”-element as its top-level container. Within this element are “Table” elements, which in turn contain elements representing columns and foreign keys. The nesting of tables in order to represent inheritance is also shown, by “Table” elements being contained inside other “Table” elements. The properties of the various elements are not shown.

The table “ControlledItem” contains a column for its attribute “version”, as well as the mandatory columns “uuid”, “Name” and “Description”, which are included in all tables. It also contains one column for each of the associations it is part of, each with a corresponding foreign key. (Each UML association is transformed into a table; the foreign keys of “ControlledItem” thus reference

this table, not the table corresponding to the other end of the association.) Since “Block” is a subclass of “ControlledItem”, the “Block” table is contained within the “ControlledItem” table. Block contains a column for its attribute “type”, as well as two columns for the two associations that “Block” is a part of. The “Block” table contains three foreign keys, one for each of the columns derived from an association, and one for the attribute “type”, since this is typed as an enumeration in the UML class diagram, and will thus reference an entry in the table corresponding to the enumeration. The tables corresponding to the “Block” class’ three subclasses are contained within the “Block” table. The “Block” table does not contain the mandatory columns, since these are inherited from the “ControlledItem” table. See Chapter 4.1.4 for a detailed description of the transformation rules.

3.2.2 Model-To-Model Transformation

The model-to-model transformation takes as input a UML2 class diagram, and outputs a model corresponding to the RDBMS meta-model, described in Chapter 3.2.1, which represents the structure of a relational database schema, with tables, columns, foreign keys, etc. This transformation contains logic for transforming UML class, association, associationclass and enumeration elements into relational database tables, and is the most complex part of the model transformations. We use Kermeta for this transformation, as it provides a mechanism for adding methods and variables to the elements of a meta-model, through the use of aspect weaving.

3.2.2.1 Constraints on Input Model

The meta-model for UML class diagrams contains a large amount of elements and constructs, and including all of these in the transformation would be way beyond the scope of this project. Also, the names of the database tables and columns are based on names given in the input model. Therefore, there are some constraints on the input, both when it comes to the types of elements accepted, and the naming of these.

The UML elements accepted by the transformation are class, associationclass, association and enumeration. Package is also accepted, even though it is not transformed. Including other elements in the input model might lead to this information being lost (since there are no transformation rules for any other type), or the transformation crashing. The class elements may contain attributes that are either of a primitive data type (string, integer, etc.), or typed as an enumeration defined in the class diagram. Attributes typed as another class or an associationclass are not allowed. The names of the classes must be unique, and the name of the attributes must be unique within a subclass tree, to avoid more than one column with the same name in a table corresponding to a subclass. Naming elements with words that are

reserved in Java (e.g. List, String) is not recommended, since it requires an explicit import each time it is used in the generated system (which is not handled by the transformation, and thus has to be done manually). Multiple inheritance is not allowed.

Enumerations must have a unique name. Associations must have both role names filled, as well as a name for the association itself. This is due to the fact that the names of the table and columns corresponding to the association are inferred from the association and role names. The combination of role names and association name must be unique. If there is need for more than one association with the same name, a single-digit integer may be added in order to achieve a successful validation of the model, this digit will be dropped in the transformation. Containment associations are allowed, but they will not be treated any differently than regular associations. For associationclasses, both the constraints on classes and associations apply.

3.2.2.2 Program Flow of the Model-to-model Transformation

We explain the details of the model-to-model transformation next; the steps involved are shown in Figure 6.

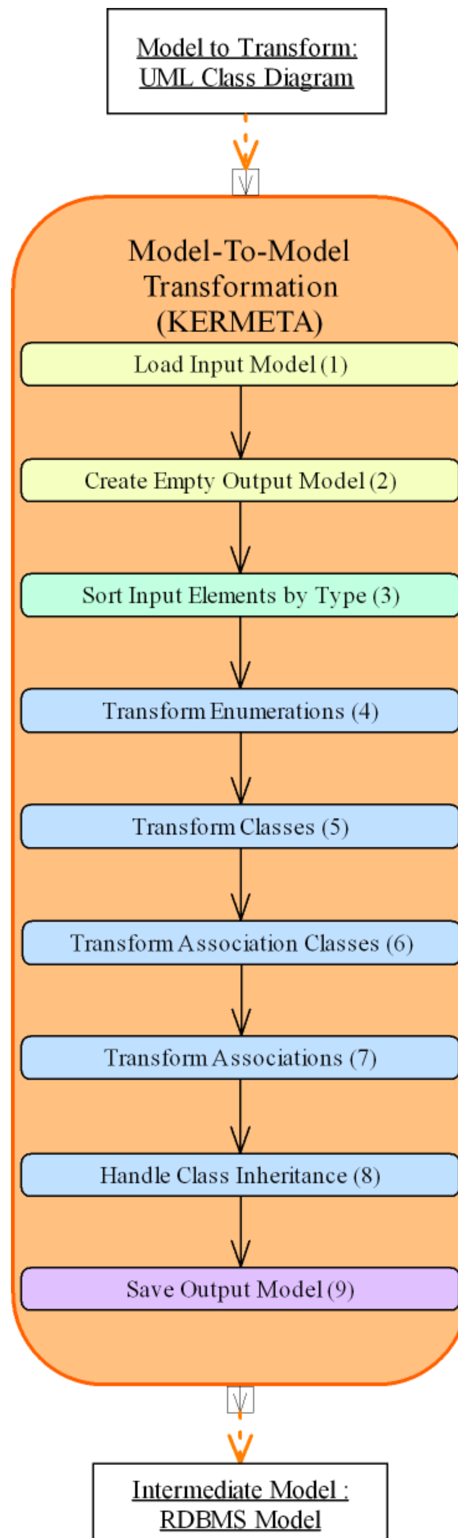


Figure 6: Program flow of the model-to-model transformation.

The transformation starts with a setup phase, which is initiated by loading the input model into memory (1). The input model is associated with its meta-model, which in this case is the UML2 meta-model. An instance of the RDBMS output model is created and kept in memory until the end of the transformation (2). Then, the elements of the input model are iterated, and each element is placed into one of four collections depending on its type (3). This results in four collections, containing UML elements of type association, associationclass, class and enumeration, respectively. This is achieved through aspect weaving, where a method "getElementsByType(allClasses, allAssociations, allAssociationClasses, allEnumerations)" is added to each of the four UML element types, where the parameters are the collections where the elements are stored. When called on an element, this method will add the element itself to the proper collection and then return.

When the elements have been categorized, each of the collections is iterated, and its elements transformed (4-7). This is also done through aspect weaving, by calling the added "toRDBMSModel(...)" method on each of the elements. This method extracts the necessary information from the element in question, and calls methods that create a table in the output model.

The order of which the types of elements are transformed is of great importance with regards to defining associations between the tables. For instance, a table derived from a class might have an attribute typed as an enumeration, which requires a foreign key to the table corresponding to the enumeration in question. If this enumeration has not been transformed into a table prior to the transformation of the class, the foreign key in the table derived from the class, indicating a relationship with an enumeration, will not be able to find the table it is supposed to reference, and thus become an incomplete foreign key. Similarly, a table corresponding to an association need both of its connector ends transformed into tables before it can reference them, to ensure no incomplete foreign keys.

To deal with this, the order of which the types of elements are transformed is the following:

1. Enumeration (not dependent on anything)
2. Class (might be dependent on an enumeration)
3. AssociationClass (dependent on class)
4. Association (dependent on class or associationclass)

This ordering ensures that all generated foreign keys reference an existing table, as long as all relationships between classes are expressed through associations (not as attributes inside classes).

After all of the elements of the input model have been transformed, the class hierarchy of the input model must be retained in the output model (8). There are several ways of handling inheritance in a relational database, but since the end product of the model transformations will use Hibernate mappings to generate the underlying database schema, the transformation will handle inheritance in a way analogous to how class hierarchies are expressed in Hibernate mappings. In the Hibernate mapping files, class hierarchies are defined through nested mappings, regardless of how the inheritance is to be handled by the underlying database. Thus, in the model-to-model transformation, class hierarchies are handled by placing the table corresponding to a subclass within the table corresponding to its superclass.

After the class inheritance is handled, the resulting output model is written to file (9), and the transformation is finished.

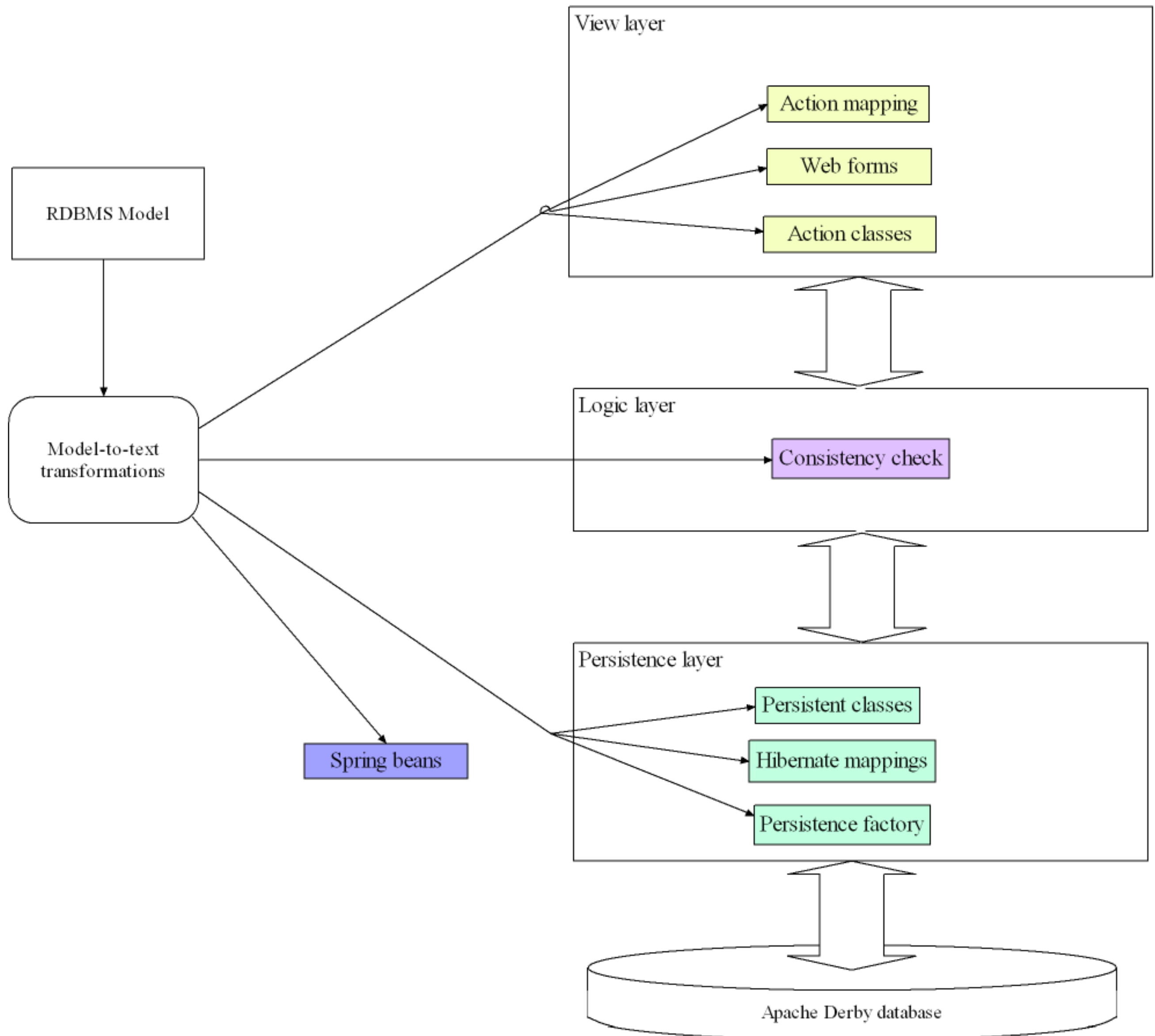


Figure 7: Artifacts generated by MOFScript transformations.

3.2.3 Model-To-Text Transformations

Once the model-to-model transformation has generated an instance of the RDBMS meta-model based on the input UML class diagram, this generated intermediate model can be used as a basis for several model-to-text transformations, implemented in MOFScript. The output of these model-to-text transformations are textual code, primarily written in Java, XML and HTML, which form parts of the information repository that we create. Figure 7 contains an overview of the artifacts generated by the model-to-text transformations.

For the persistence layer, Hibernate mappings and persistent classes are generated. These files compose an ORM (Object-Relational Mapping), which allows relational database entries to be treated as Java objects. The set of Hibernate mappings provide a description the underlying database structure, which can be used by Hibernate to generate the SQL-statements necessary to initiate the underlying relational database. The mappings also link each database table with a persistent class. The persistent classes are Java classes that, when linked to a database through Hibernate mappings, can be used to access entries in a database as Java objects. For the persistence layer, a persistence factory is also generated, which provides methods for easy access of the entries in the database, as well as methods for deleting entries. In addition, file paths to the various mapping files are added to a Hibernate configuration file, which is read at program start-up, and used to set up a connection to the underlying relational database. Adding the paths to the mapping files causes them to be read on program start-up as well, and if specified, used to initiate the database schema.

For the view layer, Struts action mappings, action classes, and JSP files are generated. These artifacts facilitate the viewing and manipulation of the persistent objects through a web-based Graphical User Interface (GUI). As an

analogy to the ORM concept of the persistence layer, these artifacts might be thought of as an Object-Interface Mapping (OIM), which maps each object to a set of GUI elements.

In addition to the artifacts of the persistence and view layer, some other files are generated. A consistency check is generated, based on the multiplicities of the associations found in the input UML class diagram, and is used for determining whether a database instance is consistent with the multiplicity constraints defined in the input. Also, Spring [16] beans are generated, and these serve two major purposes; (1) to link the view and persistence layer together, allowing the database connection of the persistence layer to be set up by running the web application defined by the view layer, and (2) for dependency injection. Dependency injection is a way of handling dependencies between objects, without having to initiate the dependencies through creating new object instances in the Java code. Instead, object instances, as well as dependencies between them, are defined in an XML configuration file. At program start-up, the instances defined in the configuration file are created, and the defined dependencies are achieved by *injection,* setting the values of variables referring to other objects through set-methods. As an example, consider a class A, that utilizes methods defined in an interface B. Without dependency injection, the implementation of interface B to use (call it class C) would have to be specified in the code of class A, by creating a new instance of the class C. If the system changes, and class C is to be replaced by another implementation of interface B, class D, then the calls to new within class A must be modified. If the system is large, and interface B used by numerous classes, the calls to new must be changed in every class. With dependency injection, on the other hand, no calls to new are needed in order to define dependencies between classes. Class A will contain a variable holding an object typed interface B, as well as a method for setting this variable. In the configuration file, an instance of class C is defined, and the fact that this should be injected into any instance of class A is also

defined. Then, substituting class C with class D would only require changes in one place, the configuration file, regardless of the number of classes that have a dependency to interface B. In addition, only one instance of the class implementing interface B is needed, the same instance can be injected into any number of dependent classes. As an example of dependency injection in this project, consider the persistence factory, which is used by a number of classes. With dependency injection, the implementation of the persistence factory can be substituted by changing one entry in the configuration file, without having to change the implementation of these classes (as long as the new implementation provides the same methods). Also, only a single instance of the persistence factory is needed, the same instance might be injected into multiple classes.

The program flow of the MOFScript transformations are fairly trivial, they all involve iterating the elements of the intermediate model, creating output files, and writing text to the output files based on the input elements. There are two main variants, depending on how the output files are created. Some of the model-to-text transformations outputs one file for each table (e.g. the persistent classes), while others produce only one file, with code based on information in all table elements (e.g. the persistence factory). The model-to-text transformation outputting one file per table element starts by iterating the input model, and then creates a file for each table element. The other variant starts by creating the file, and then iterates the input model, adding text derived from all table elements to the same file. The program flow of the two variants is shown in Figure 8.

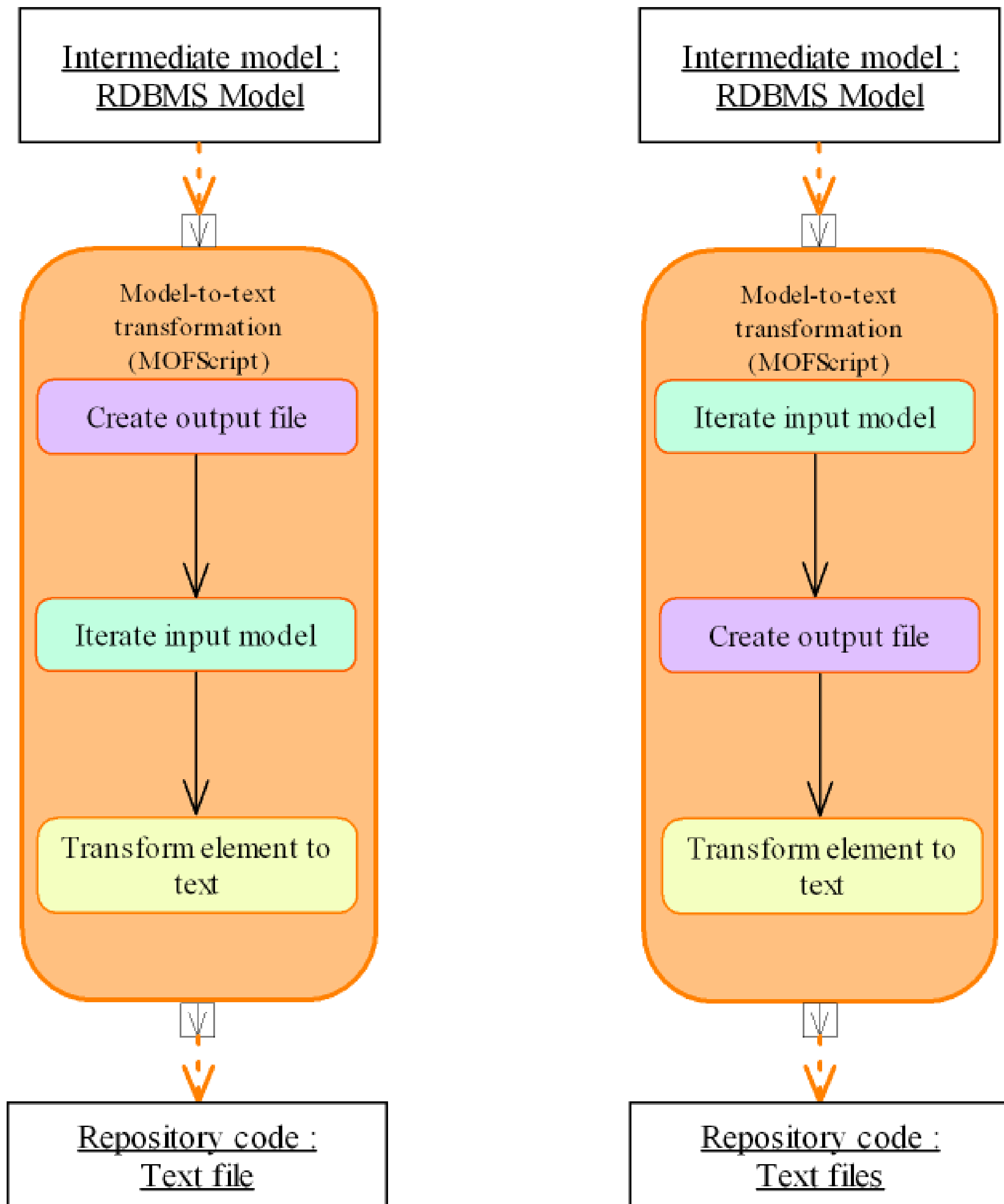


Figure 8: Program flow of the MOFScript transformations (2 main variants).

3.3 User Roles

The typical use of the system is initiated by an administrator or administrative body, which runs the transformations, and sets up the database and the webserver. Once the repository is set up, the users can view and manipulate the data, as well as ensure that the data is consistent with the constraints defined in the UML class diagram, through the consistency check. The users could be developers, who registers data during the development of the safety-related system, or certifiers, who upon the completion of the safety-related system are allowed access to the data in the database directly, or to generated compiled reports based on the data in the database (yet to be implemented), in order to aid the certification of the safety-related system.

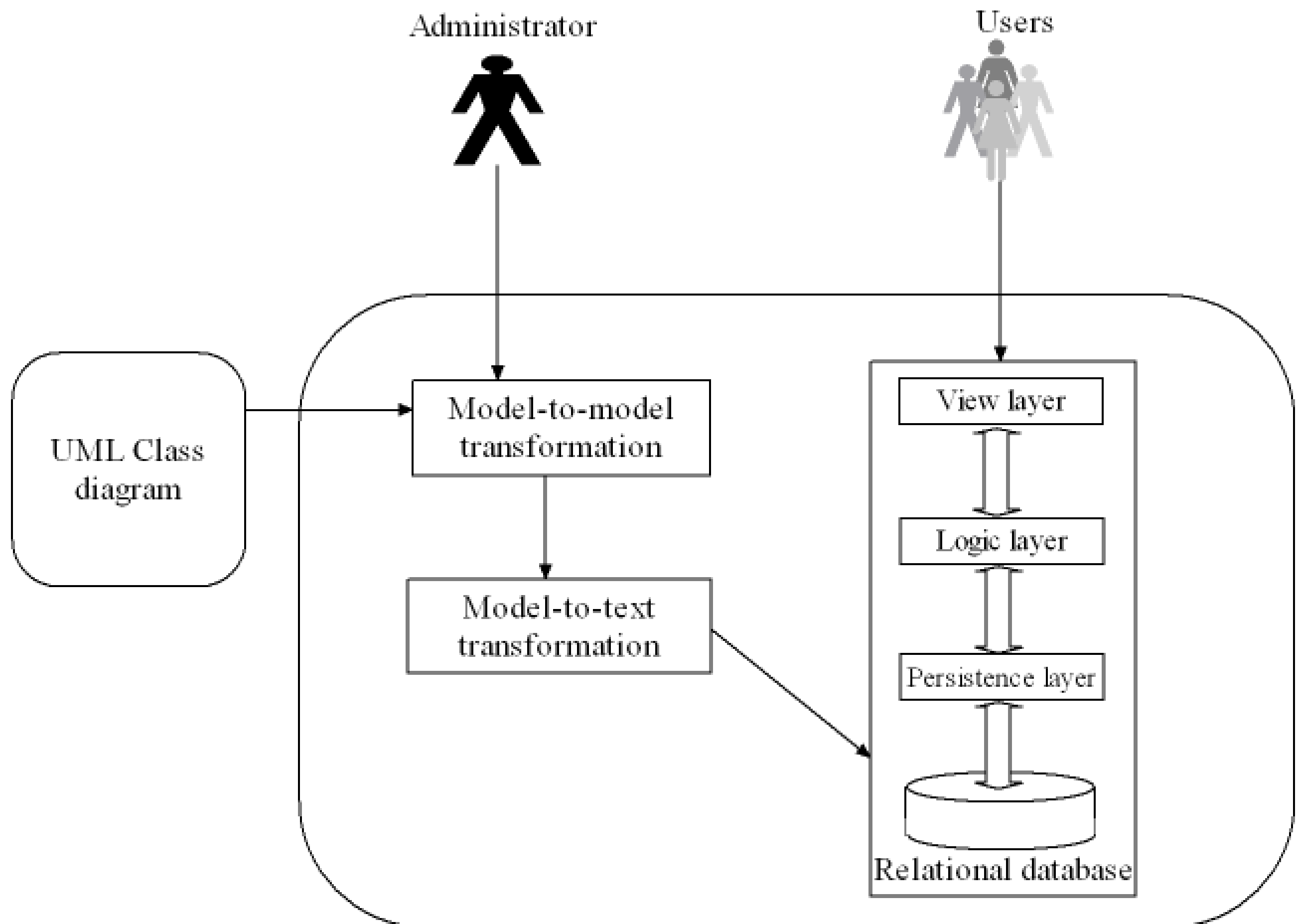


Figure 9: User roles in the transformation and the information repository.

4. Discussion

4.1 Design Choices

4.1.1 Creation of a Repository from Scratch

When starting this project, there were alternatives to creating the repository from scratch. There exists tools that transform a UML class diagram into a database schema in the form of SQL-statements [17], which in turn can be used as input to a database system (e.g. Microsoft Access). The problem with this approach is that it only allows the generation of the SQL-statements that can be used to create a relational database. Hence the SQL statements would have to be applied manually to create the database. The user interface for adding and manipulating the data in the database would also need to be created. There also exist tools that allow the generation of other artifacts besides SQL-statements, like UMT-QVL [18], which in addition to SQL-statements can generate a number of other artifacts, for instance Java interfaces. The problem with using this tool is that it does not provide all of the required transformation. The tool is a general-purpose transformation framework, within which the actual transformation code required to transform the UML model would still need to be written. Extending this transformation tool would require knowledge of how the existing code of the tool works, which introduces different challenges when creating the model transformations. There also exists a tool for generating Hibernate mappings based on UML class diagrams [19]. This tool was not used, for the same reason as with UMT-QVL, as it does not provide all of the needed transformations.

The creation of a model transformation tool allows an automated creation of both the database and the user interface to manipulate the data in the

database, as well as any other artifact that depends on the structure of the database schema. Using already existing technologies would allow less flexibility compared to creating a repository system from scratch, which would allow us to tailor and extend the system to our needs. In addition, adding new functionality is easier with full control over the source code. Although existing transformation tools could be extended to achieve the generation of the required artifacts, the challenges associated with this were not judged to be of such a magnitude smaller than the challenges associated with creating the model transformations from scratch. All-in-all, the ability to tailor the system for our specific needs made it more reasonable to create something new, than using already existing technologies.

4.1.2 Use of Model-Driven Technologies

Through the use of model-driven technologies, we could automate the transformation of a UML class diagram to a relational database through a series of automated transformations, rather than creating a database manually. The use of model-driven technologies had a number of positive effects. First, and foremost, it led to the creation of a tool that is not limited to just one input model, but can be used on any input model that adheres to the constraints defined in Chapter 3.2.2.1. Secondly, it is much easier to ensure the correctness of rules for transforming the elements of an input model, than it is to ensure the correctness of a manual transformation of such a model, especially if the input model is large. Also, even though quite some time has been spent creating the transformation tool, a lot of time will be saved if there are changes to the input model, or if a repository is to be created based on a different input model.

4.1.3 Use of an Intermediate Model

We chose to use an intermediate model, even though a direct model-to-text transformation with the UML class diagram as input would be possible. The use of an intermediate model had a number of advantages. It allowed the most complex logic of the transformation tool to be kept in one place, the model-to-model transformation. This led to the model-to-text transformations being fairly trivial, since the data structure was already defined, and all that was necessary was to iterate over the intermediate model, and output text for each of its elements. As the work on the repository progressed, it turned out the intermediate model could be used for much more than just generating the artifacts concerning the establishment of the database itself, in addition artifacts for the user interface that allows manipulation of the database, as well as the persistence factory and the consistency check could be generated with basis in the intermediate model.

If no intermediate model were to be used, the logic for transforming a UML class diagram into the structure of a relational database, and the logic for creating and writing to a large number of text files would have to be interwoven. This would lead to a complex model transformation, that would be difficult to implement and extend, compared to when an intermediate model is used.

Through the use of an intermediate model, creating new applications of the information in the UML class diagram is simple, since all it takes is the creation of another model-to-text transformation with the intermediate model as input, without having to make changes to how the UML class diagram is transformed into the intermediate model.

The use of an intermediate model made Kermeta the natural choice of language when implementing the model-to-model transformation, since Kermeta is a powerful tool for creating model-to-model transformations.

4.1.4 UML to Relational Database Transformation Rules

The rules for transforming the elements of a UML class diagram into a relational database was extracted from the book “Database Systems: The Complete Book”, by Garcia-Molina et.al. [20]. The book gives directions on how to transform Entity/Relationship (E/R)-diagrams [21] into a relational database tables. These directions were adapted to fit a UML-to-relational database conversion. The allowed elements in the input UML class diagram are class, associationclass, enumeration and association (as well as package, which is not transformed in any way). All tables will have a mandatory column UUID (Universally Unique Identifier), which is used as a primary key, and a way to distinguish the entries in the database. In addition, all tables have the mandatory columns “Name” and “Description”, due to the fact that many of the elements of the input UML class diagram did not have any attributes, and would without these mandatory columns only consist of a UUID, which from a user perspective would not be enough. The “Name”-field is used to distinguish entries, and further information about the entry can be stored in the “Description” column.

The rules for transforming the different element types are as given in the following sections.

4.1.4.1 Enumeration

Each enumeration is transformed into a table. Since enumerations don’t have attributes, and is not part of associations, the tables corresponding to enumerations will contain only the three mandatory columns: uuid, name and

description. The value of the enumeration entry can be stored in the name-attribute.

4.1.4.2 Class

Each class is transformed into a table. Each attribute of the class is transformed into a column in the table. The attributes can have one of two types; either (1) that of an enumeration or (2) a primitive data type, like string or integer. When the type of an attribute is an enumeration, a foreign key linking the column to the table corresponding to the enumeration is added, making the value of the column a reference to the table that corresponds to the enumeration.

4.1.4.3 Association

Each association is transformed into a table. The connector ends of the association are added as columns, and foreign keys tying together these columns and the tables corresponding to the classes of the connector ends are added. The multiplicities of the association are persisted in the output model through the use of Boolean flags contained within the column elements.

4.1.4.4 AssociationClass

Each associationclass is transformed into a table. Attributes of the associationclass are added as columns, and the connector ends are transformed into columns with foreign keys, analogous to the way the connector ends of the association are transformed.

4.1.4.5 Inheritance

In “Database Systems: The Complete Book”, the authors describe three ways of handling class inheritance when converting a class diagram to a relational database schema. The possibilities are (1) one table per top level superclass,

with the use of null values, (2) one table per class containing all of the attributes of its superclass in addition to the attributes of the class itself, and (3) one table per sub-tree of the class hierarchy. But, since class inheritance is handled by Hibernate with nested class-table mappings, inheritance is handled the same way in the model-to-model transformation, regardless of the way it will be handled in the finished system. The class inheritance is handled by specifying that a table corresponding to a subclass is to be contained within the table of its superclass (multiple inheritance is not allowed). This may result in a tree of tables in the output model (see Figure 5), and simplifies greatly the complexity of handling the class inheritance in the model-to-model transformation.

4.1.5 Tool Selection

The choice of using Kermeta for the model-to-model transformation was based on the fact that it is widely used for model-based development in scientific research, and that it has been used for this purpose by scientists at Simula Research Laboratory. Kermeta provides a good infrastructure for model-to-model transformations. MOFScript became the chosen technology for model-to-text transformations, because it is a well-made and widely used tool, and because I had some prior knowledge of the tool after using it in a course taught at the University of Oslo.

When it became clear that the repository was to be implemented in Java, using tools like Hibernate, Struts, Spring (which is a platform for building and running enterprise Java applications) and Maven (which is a build manager for Java projects) became obvious choices, since these tools are widely used in the industry, and there exists large amounts of documentation on each of these tools. Also, I had some experience with the tools from a course taken at the University of Oslo.

The decision to use of Eclipse as a development platform throughout the project was also quite easy to make, since it is a host to both Kermeta and MOFScript, and also a powerful tool for developing Java projects. Apache Derby was chosen as the host of the underlying relational database, since it allows an embedded mode in addition to the conventional client/server mode, allowing it to run as part of a software system. Also, Apache Derby is a well-established database management system.

4.1.6 Constraints on the Database

We chose to keep the database as free from constraints as possible, in order to allow the user a maximum amount of freedom when manipulating the data in the database. Null values are accepted in every column (except for the uuid-column, which is automatically populated by Hibernate). Conformance with the multiplicity constraints on associations in the input UML class diagram is not handled by constraints on the database schema, but rather ensured through the consistency check, which in addition to checking that the multiplicity constraints are held, checks for null values where they are not allowed (e.g. in columns referencing an entry in another table as part of an association).

4.1.7 Using a Web-based User Interface

The use of a web-based solution for the user interface had a number of advantages. First, a web-based user interface requires no set-up on the user side in order to access the repository, all that is needed is a network connection to the webserver that hosts the repository, an internet browser, as well as a permission to access the repository. This means that the repository can be updated from practically any computer with an internet connection. Also, a web-based solution makes it easy to allow multiple concurrent users of the repository. Finally, the exchange of data is greatly simplified; all that is

needed is to grant the person/organization in need of the data access to the repository.

4.2 Lessons Learned

4.2.1 Software Development - More Than Programming

During the course of this project, I have learned that software development involves a lot more than just programming. A significant amount of time has been spent setting up the various tools used, reading up on them, understanding how they are used, and understanding their error messages. I also learned about the importance of having a good design, and a clear structure of the artifacts in a software system, in order to ensure flexibility and maintainability of the system.

4.2.2 Learning about the Tools

In this project, the amount of experience I had with the different tools varied, some I had some experience with through courses taught at the University of Oslo, while others, most notably Kermeta, I had no experience with. This led to a period of learning about the tools before effectively using them, both when it came to the tools that I had experience with, and those that I did not. For most of the tools, there were a lot of online documentation and tutorials, minimizing the time spent on learning to set up and use the tool. Others, on the contrary, had very little available documentation. Kermeta, used in the early stages of the project, did not have much available documentation, only a reference guide and a couple of examples. This led to a quite time-consuming process of trial and error, in order to learn how to use Kermeta effectively. (See appendix A.1 for an overview of the teaching materials used during the course of this project.)

4.2.3 Tool Problems

In addition to the time spent learning how to use the tools, there were some problems with the tools themselves.

4.2.3.1 Kermeta

At the time of writing the Kermeta transformation, not all of the functionality for elements in the UML specification was implemented. For instance, getting the other end of an association when inspecting one end was supposed to be easy, but required a workaround since the method for doing this was not yet implemented in Kermeta.

4.2.3.2 Hibernate

There were some problems with the functionality of Hibernate, with regards to cascade options when deleting. When mapping the relationships between tables as a unidirectional association, it was only possible to cascade in this same direction, which was the opposite of what was wanted behavior. This led to a quite large work-around, resulting in all the associations being bi-directional in order to achieve a cascading delete.

4.2.4 Loading Input UML Models with Kermeta

Getting Kermeta to load a UML model created on another platform (in this case Rational Software Architect) turned out to be quite tricky. Upon defining which input model and meta-model to use, the transformation just crashed, giving an error message without much detail as to what was the problem. After a lot of trial and error, it turned out that the UML profiles of the software that created the class diagram had to be included, in the same folder as the class diagram itself. This information was not part of the Kermeta documentation, which led to quite some time spent figuring it out.

4.2.5 Setting up the Required Libraries

When starting the project, I did not have a full understanding of how to include third party libraries in the project classpath, making them available to the compiled system. After some trial and error I managed to include the third party libraries, by adding them to the classpath of the Eclipse project, as well as adding their jar-files in a the proper folder, so that the compiled program could access them.

5. Implementation

The implementation of the work described in this thesis can be divided into two parts; (1) a transformation from a UML class diagram to a database management system, and (2) the database management system itself. The total number of lines of code for (1) is approximately 2000 (800 for Kermeta transformation, 1200 for MOFScript transformations). The total number of lines of code for (2) depends on the size of the input model, with the conceptual model based on IEC 61508, which was used as a basis when developing, the total number of lines of code exceeded 10,000, most of it generated.

Our implementation builds on a number of existing technologies. An overview of how these technologies are used in our work is provided in Figure 10.

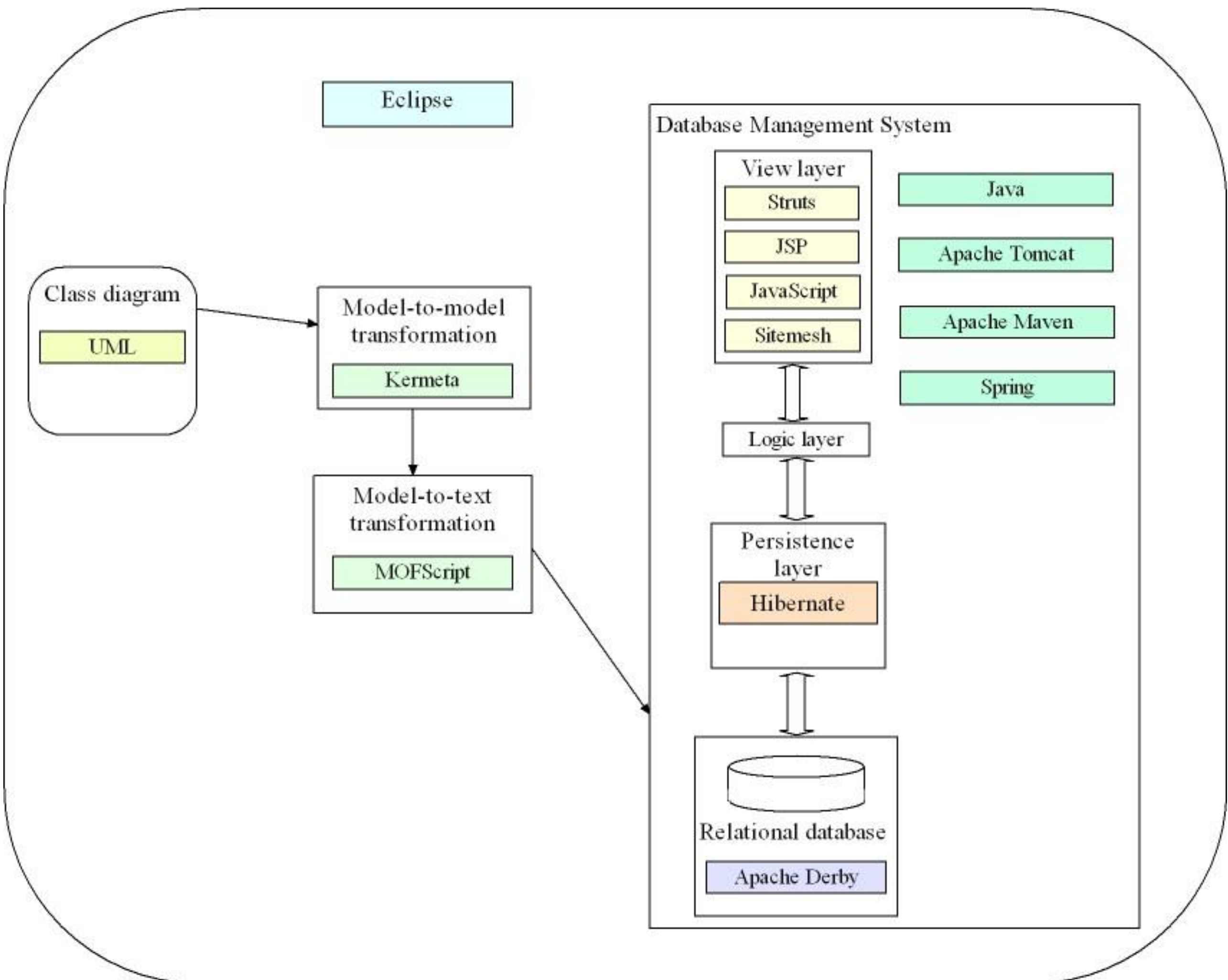


Figure 10: Overview of the technologies used and the context of their use.

In Figure 10, the colored squares represent the technologies used in the project. The context of the technologies used is represented through where the technology is contained in Figure 10. The outer rounded square represents the project as a whole, and contains Eclipse, which was used throughout the project. Kermeta and MOFScript were used for the model-to-model and model-to-text transformation, respectively. Within the information repository, some technologies fall under a specific layer, and others are used across the system as whole.

For the thesis to be self-contained we provide a short description of each of the technologies used below.

5.1 UML

The Unified Modeling Language (UML) [3] is a specification published by OMG (Object Management Group). It provides a standardized way of creating a variety of models, ranging from structure, behavior and architecture models, to business process and data structure models. UML utilizes graphical notation techniques, in order to facilitate the creation of visual models of object-oriented software systems. The UML standard has matured considerably since first released, and UML 2.0 was released in 2005. The latest revision of UML was version 2.3, released in May 2010. The UML standard (version 2.2) defines 14 types of diagrams, divided into two main categories; behavioral and structural diagrams.

In this project, UML Class diagrams, which are one of the structural diagram types, form the basis of the input to the model transformations resulting in a database management system.

5.2 Eclipse

Eclipse [22] is “an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle.” [23] Eclipse is hosted by the Eclipse Foundation, which is a non-profit member supported cooperation. The Eclipse project offers a variety of products free of charge, which are variations of the Eclipse integrated development platform (IDE), including Eclipse IDE for Java and C/C++ developers, as well as Eclipse Modeling Tools [24]. The Eclipse IDE also

offers a high level of user modification through plugins, and is used as a basis for other software development platforms, like IBMs Rational tools, as well as a platform for development in programming languages like Kermeta and MOFScript.

The Eclipse IDE was the main development platform for the work described in this thesis. In addition to the Eclipse's built-in functionality for Java programming, plugins for Kermeta, MOFScript and Maven were used.

5.3 Kermeta

Kermeta [6] is an object-oriented DSL (Domain Specific Language) optimized for meta-model engineering. The Kermeta workbench runs on top of Eclipse, and is available both as a Kermeta+Eclipse bundle, and a plugin to Eclipse [25]. Kermeta can be used for specifying abstract syntax, static and dynamic semantics with connection to the concrete syntax, as well as model transformation, aspect weaving, and model and meta-model prototyping and simulation. Kermeta is an open source project, initiated within the Triskell [26] team of IRISA [27], and the programming language borrows concepts from languages such as OCL, MOF and QVT, as well as BasicMTL, a model transformation language created by the Triskell team. [28]

In my project Kermeta's model-to-model transformation features was used to transform an input UML class diagram to a model corresponding to the structure of a relational database. In particular, Kermeta's aspect weaving functionality was used in order to achieve this.

5.4 MOFScript

MOFScript [7] is a subproject of Eclipse, and a programming language for model-to-text transformations. MOFScript is distributed as an Eclipse plugin [29]. MOFScript supports model-to-text transformations from MOF-based models, including UML or any kind of domain model, as well as control mechanisms like loops and conditional statements, string manipulation and production of output files [30].

In this project, MOFScript was used to generate code-containing text files, based on the intermediate model output from the model-to-model Kermeta transformation, through several model-to-text transformations.

5.5 Java

Java [31] is an object-oriented programming language, originally created by Sun Microsystems. Java derives most of its syntax from C and C++, and is one of the most popular programming languages today, used by over 6.5 million developers, in every major industry segment [32]. Among the primary goals in the creation of Java we find that the language should be “simple, object-oriented and familiar”, “robust and secure”, as well as having “"an architecture-neutral and portable environment" [33].

Java was used as the programming language for creating the information repository, along with Java-specific tools.

5.6 Apache Maven

Apache Maven [34] is an open-source build manager, used for managing software projects, as well as build automation. Maven can be used in the whole lifecycle of software development, and is useful for compiling, packaging and installing, as well as running tests on, a large number of code files, through simple commands. Maven also provides a convenient way of handling dependencies to Java libraries and plug-ins, through the use of repositories. There are several public Maven repositories, and the user can make requests for libraries through adding them as dependencies in their Maven configuration file (called pom.xml). If the dependency cannot be resolved in the user's local repository, it is downloaded from one of the public Maven repositories upon use. This simplifies the handling of dependencies to third party libraries significantly.

Maven was used for building the system when implementing the back-end of the system. It was also used to handle the dependencies to third party libraries and plug-ins. When starting the work on the front-end, the system was built by starting the web server through Eclipse, though this could have been done from through Maven as well.

5.7 Apache Derby

Apache Derby "is an open source relational database implemented entirely in Java and available under the Apache License, Version 2.0" [12]. Derby provides both the familiar client-server mode, and an embedded mode, which allows a Derby database to be embedded into a Java program, starting the database once the Java program is started.

Apache Derby was used as the host of the underlying relational database of the information repository.

5.8 Hibernate

Hibernate is a “collection of related projects enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object/Relational Mapping.” [9] The primary feature of Hibernate is as an Object-Relational Mapping (ORM), which can map Java objects with database tables. This allows entries in a relational database to be treated as Java objects, creating a layer of abstraction on top of the relational database. Hibernate also provides data query and retrieval functionality, through the Hibernate Query Language (HQL), which is similar to SQL. Queries are executed through calls to Java methods of the Hibernate library, and Hibernate then generates the SQL-queries to the database, and converts the results to a set of objects.

The main artifacts of Hibernate are a configuration file, which defines the settings to use when connecting to a relational database (e.g. database type/brand, SQL dialect, username and password), a set of persistent classes, which are plain Java classes with set- and get-methods for each attribute, and a set of mappings, which maps each of the persistent Java classes to a relational database table. The mappings can be represented through Hibernate annotations on the persistent classes themselves, or as separate mapping files, in which paths to the mapping files are added to the Hibernate configuration file, in order to be read at program start-up.

In this project, Hibernates ORM functionality was used to access, create, edit and delete elements of the relational database through the use of Java methods and objects. Hibernate was also used to initialize the database itself, through the definition of the Hibernate mapping files.

5.9 Spring

The Spring [16] framework is an open source development framework for Java projects. Spring consists of a package of core functionality, as well as a number of extensions, amongst others for building web applications. Spring provides an "Inversion of control"-container, which is used for managing Java objects, through the use of Java beans. This feature also provides dependency injection capabilities, which can be used for instantiating an object, and injecting this instance into other objects through the use of set- and get-methods for the attributes in a class. This eliminates the need for calls to "new" in the Java code, and allows changing parts of the system, without having to change the Java code of other parts. Spring also provides a number of other capabilities, like aspect orientation and transaction management.

Spring was necessary in this project, in order to connect the persistence layer (Hibernate) with the view layer (Struts), to initiate the persistence layer upon the running the web application on a web server. Dependency injection was also used to inject (singleton) instances of PersistenceFactory.java and HibernateUtil.java into other objects.

5.10 Apache Struts

Apache Struts [35] is an open source framework for creating Java web applications. Struts is designed to facilitate the use of the Model-View-Controller (MVC) architecture. Struts provide a simple way of integrating web-based user interfaces with Java applications, both by facilitating the invocation of Java methods through a web-based user interface, and by providing mechanisms for presenting and manipulating Java objects through a web-based user interface. In order to achieve this integrations, Struts utilizes the concept of actions, which consists of three parts; (1) the actions

class, which is a Java class that implements the ActionSupport interface (which defines the class as an action class), (2) JavaServer Pages (JSP) files, which are files that can contain both Java and HTML code, and (3) action mappings, that defines actions, which are used to call Java methods from a web-based user interface. The action mappings define a name for the action, as well as an action class and a method to be called when the action is called. The action mapping can also define a number of JSP-pages to redirect to once the method has been run, where which page to redirect to depends on the return value of the Java method called. Once the user initiates the calling of an action defined in the action mapping, the method of the action class defined in the mapping is called, and upon its return the user is redirect to the appropriate web-page. Struts also provides a tag-library, which allows the viewing and manipulation of Java objects in a simple manner, in order to facilitate the creation of interactive form-based applications with server pages.

Struts was used to provide user interfaces access to persistent objects and their attributes, in order to present and manipulate the objects residing in the database. Struts actions were also used to provide parts of the navigation on the user interface.

5.11 Apache Tomcat

Apache Tomcat is an open source servlet container, which implements the specifications for Java Servlet and JavaServer Pages. Tomcat provides a HTTP web server environment for running Java code, and is used to power “numerous large-scale, mission-critical web applications across a diverse range of industries and organizations.” [36]

In this project, Apache Tomcat was used to publish the information repository on a (local) web server, which could be accessed from a web browser.

5.12 JavaScript

JavaScript [37] is an object-oriented scripting language, used primarily in web pages, in order to provide enhanced user interfaces and dynamic websites. JavaScript is dynamic, weakly typed, and is considered a functional programming language. JavaScript is a client-side language, meaning that the code is executed in the user's web-browser, as opposed to on a server.

In this project JavaScript was used in parts of the view layer, mainly for posting a HTTP request and including the result in an existing web page.

5.13 SiteMesh

SiteMesh is "a web-page layout and decoration framework and web-application integration framework to aid in creating large sites consisting of many pages for which a consistent look/feel, navigation and layout scheme is required." [38] SiteMesh allows the definition of decorators, which can be used to add elements to web-pages when the requests are filtered through SiteMesh. SiteMesh can also be used to include an entire HTML-document as a panel within another web-page. SiteMesh is based on the "Decorator" design pattern [39].

The main artifacts of SiteMesh are a decorator, which is a JSP-file that contains the definition of the graphical elements that are to be present in every page using the decorator. This decorator is applied through a filter in the web application, sending all requests through SiteMesh. Which decorator to use for a particular web-page, if any, is defined through a SiteMesh configuration file.

SiteMesh was used to include the menu in the various web-pages of the information repository.

5.14 JavaServer Pages (JSP)

JavaServer Pages (JSP) [40] is a Java technology that helps Java developers create dynamic web-pages. JSP is the Java-based counterpart of ASP and PHP. JSP allows a mixture of Java and HTML and the JSP-file is compiled by a JSP-compiler into a Java servlet, where any Java code can be executed, and any HTML code is printed to a file read by the web browser. JSP also allow extensions of HTML through the use of tag libraries.

In this project, JSP was the format of the web-forms composing the user interface. The possibility of integrating Java code with HTML code was not used, but the ability of using extension through tag libraries was used in order to utilize features of Struts, namely the Struts tags for presenting objects and their attributes.

6. Conclusion

The work described in this thesis enables the creation of an information repository for storing safety-related information, where the structure of the information to be stored is described as a conceptual model and represented as a UML class diagram. The work meets a direct industrial need, as it provides a concrete solution for the storage and manipulation of data regarding the lifecycle of the creation of safety-related software systems, in accordance with the IEC 61508 standard. While our work has been primarily driven by IEC 61508, the tool infrastructure developed is generic and can be applied to any standard whose evidence information requirements have been captured using a UML class diagram.

The work done in this project provides the basic functionality of an information repository, which can be extended in future work. Possible extension of the information repository might include adding logic for checking that the data in the repository is consistent with constraints other than the ones defined through the multiplicities of the associations in the input UML class diagram (which are already ensure through the consistency check), as well as functionality for generating reports based on the data in the repository. Improvements of the design and usability of the user interface might also be a topic of future work. Adding new functionality to the repository is made easy by the fact that the information repository is implemented from scratch as a Java project, using essential technologies that simplifies the development of the repository and its functionality. In addition, new functionality can be generated through model-to-text transformation, which greatly reduces the time spent creating functionality dependent of the data structure defined in the UML class diagram.

Appendix A

A.1 Learning Materials

During the course of this project there was a returning need to learn about the various technologies used. The amount of written material varied from technology to technology, some had numerous tutorials available online, while others had just a reference manual.

A.1.1 Kermeta

The written material regarding Kermeta consisted for the most part of the Kermeta reference manual [41]. I also used the Kermeta online forums [42] to ask for help regarding issues that were not described in the reference manual. Since Kermeta is mostly used for scientific work, there was not much written material available.

A.1.1.1 Using Kermeta's Aspect-oriented Features

When first starting the project I had no experience with Kermeta, and I struggled a bit to get started. The Kermeta documentation was also limited, and there were no good tutorials for programming in Kermeta. I visited the online forums of Kermeta, asking for pointers on where to start. There, I got a reply from Didier Votisjek, one of the creators of Kermeta, leading me on to Kermeta's aspect-weaving functionality, which allows the creation of new operations within the elements of the input model.

A.1.2 Hibernate

In order to learn how to set up Hibernate I mostly used the Hibernate reference documentation [43]. I also did some web searches in order to work out minor problems, and when this was not sufficient I asked for help at the Hibernate community forum [44].

A.1.3 Struts

When learning about Struts I used two online tutorials, mainly [45], but also [46] as a reference. I also used parts of the official Struts tutorials [47].

A.2 Example of use

This section contains an example of the use of the tools described in this thesis, both the model transformations, and the information repository. This example is slightly simplified, and is intended to give the reader a concrete understanding of how the tools work, through showing the most important facets of the tools.

The transformation tool and the information repository are distributed as an archive file, containing two other archives, “UML2RDB-Transformation.zip”, and “InformationRepository.zip”. The first of these contains the model transformations, and the other contains a folder skeleton of the information repository, containing the libraries required by the repository, as well as an Eclipse project configuration file.

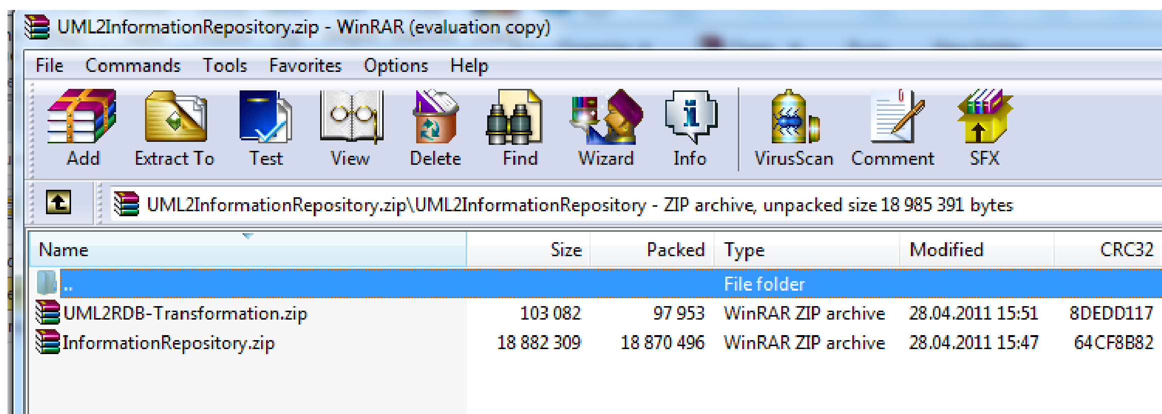


Figure A.1: The archive file containing the source code.

A.2.1 Model Transformations

To run the model transformations, one needs Eclipse installed, with plugins for Kermeta and MOFScript. The plugins can be installed through Eclipse’s download manager.

Start by importing the “UML2RDB-Transformation.zip”-file into the Eclipse workspace, by selecting “File”, then “Import”, and then “Import existing projects from Workspace”.

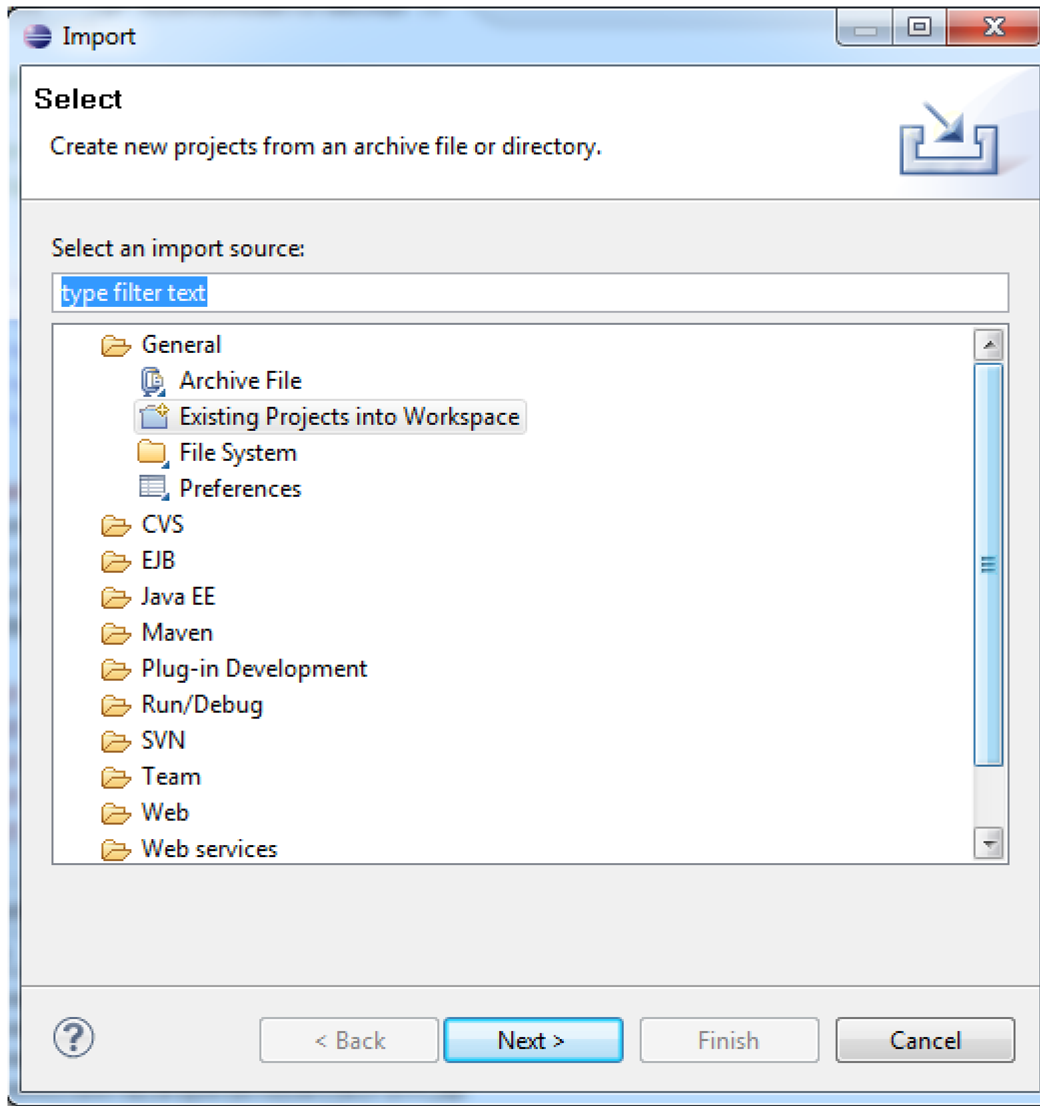


Figure A.2: Import existing project into workspace.

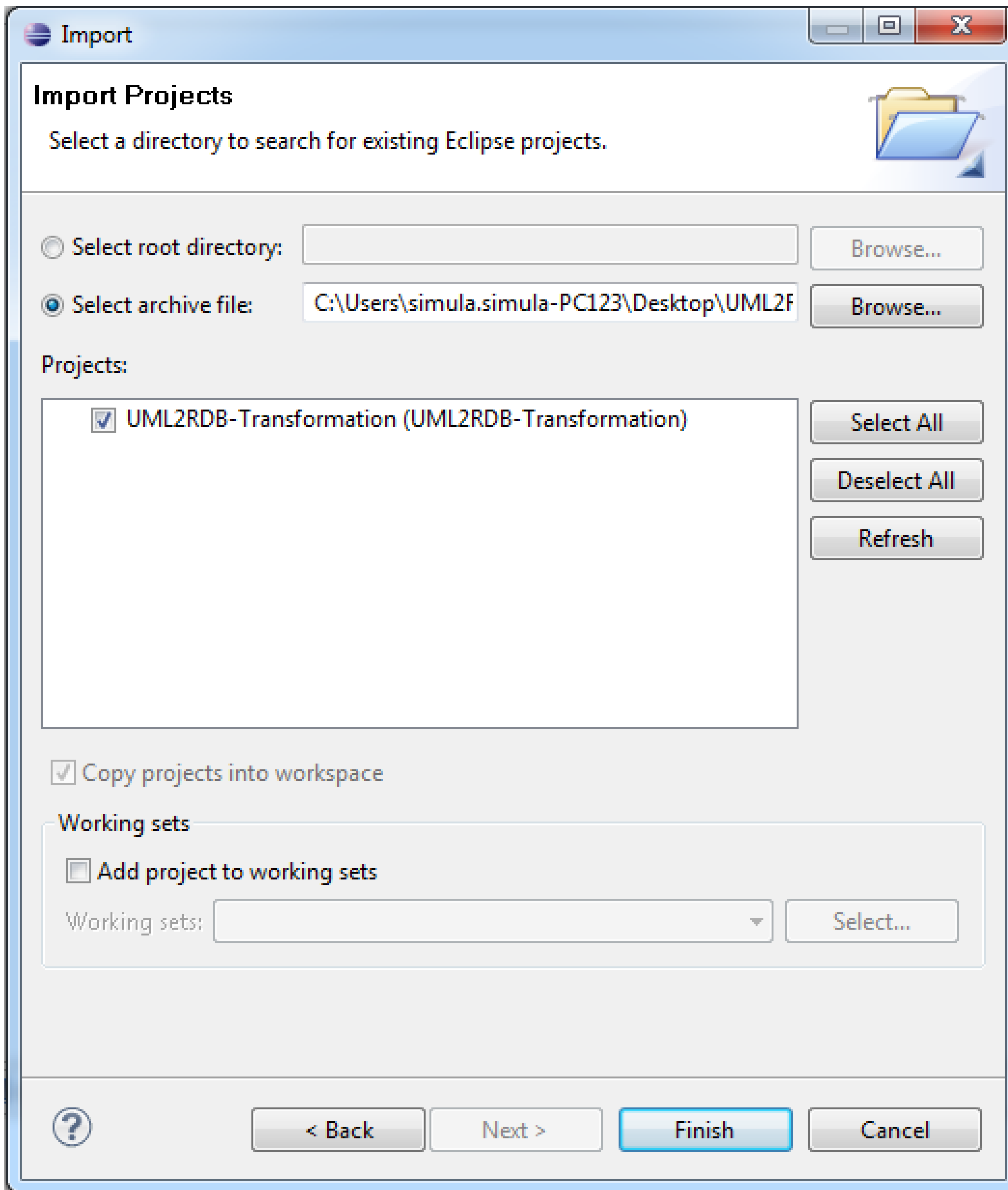


Figure A.3: Import UML2RDB-Transformation.zip into the Eclipse workspace.

This will import the project used for the model transformations. The outline of this project is shown in Figure A.4.

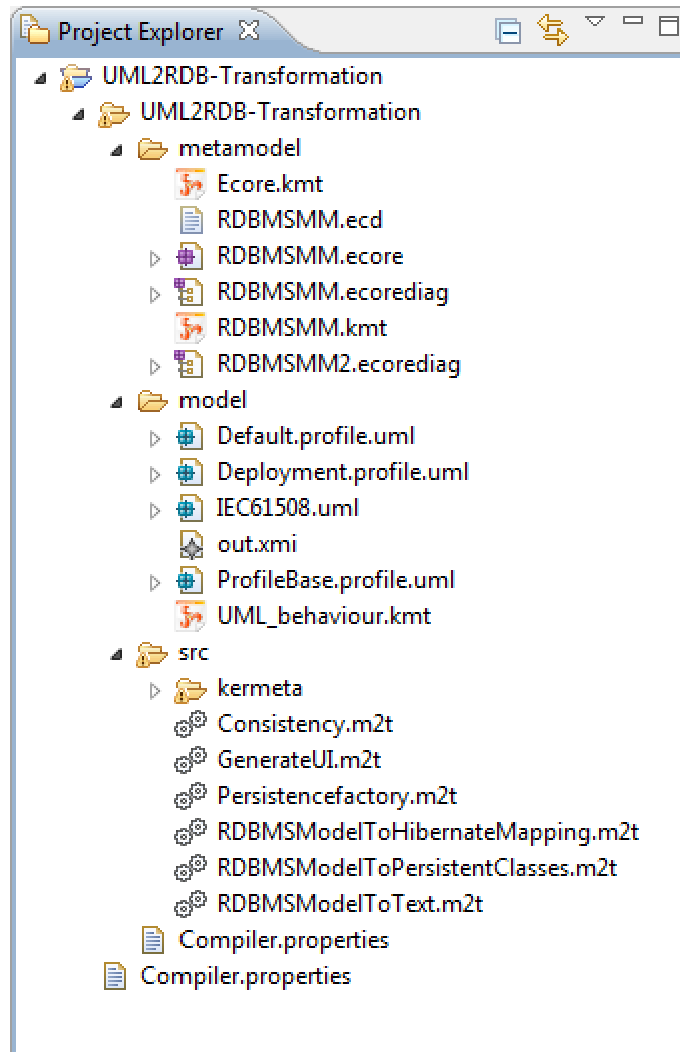


Figure A.4: Outline of the Eclipse project containing model transformations.

The project has three main folders, (1) “metamodel”, containing the meta-model of the intermediate model, (2) “model”, which is where the input and output models are stored, and (3) “src”, which contains the model transformations, implemented in Kermeta and MOFScript.

The imported project does not contain any input model; the next step will be importing this into the workspace. This is done by right-clicking the “model” folder, and selecting “Import”, then “File system” or “Archive file”, depending on how the model is stored.

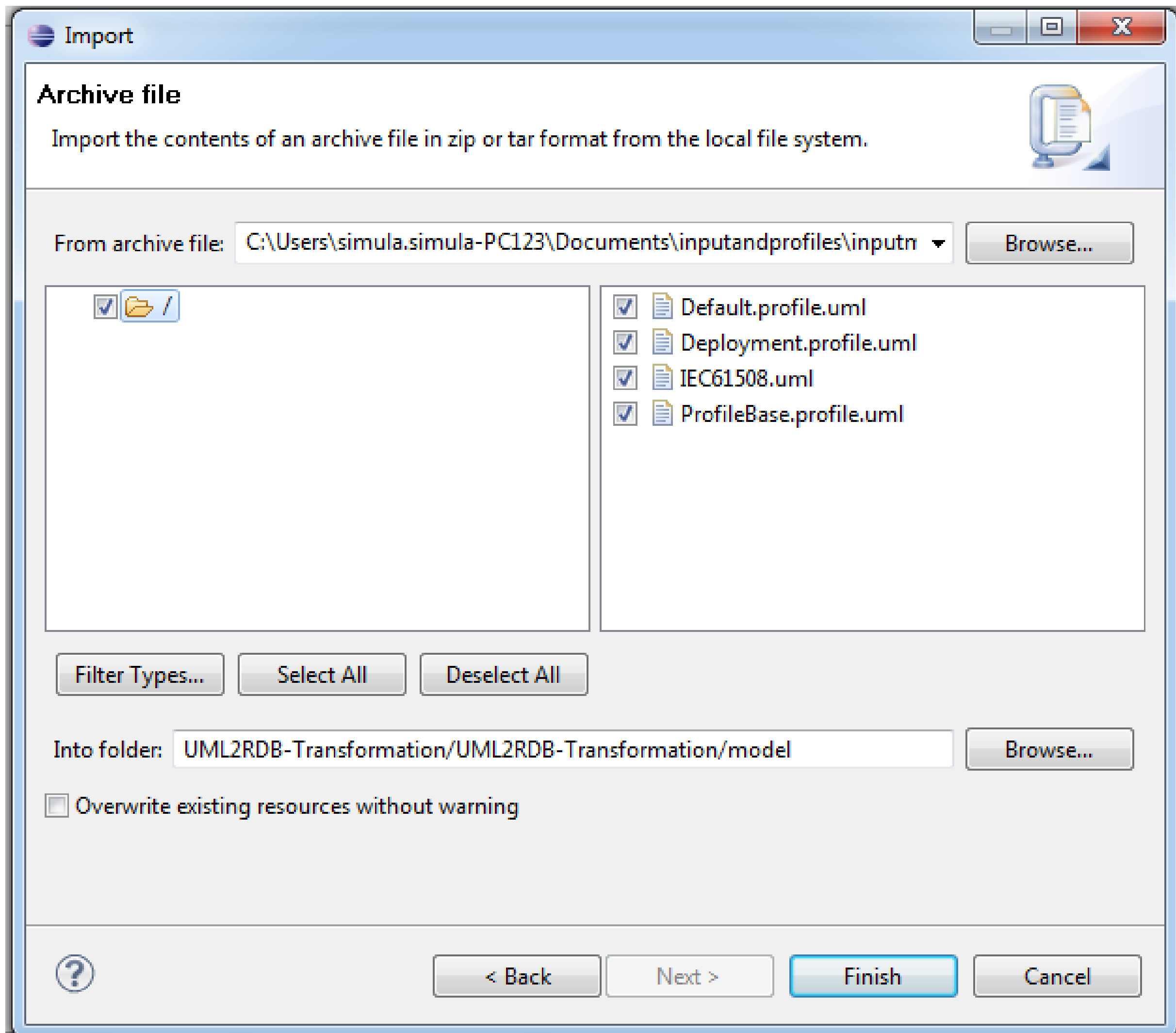


Figure A.5: Import input models into workspace.

In this example, the conceptual model based on the IEC 61508 standard will be used as input. Any UML class diagram conforming to the constraints defined in Chapter 3.2.2.1 can be used. It is important to import the applied profiles used when creating the input model, along with the model itself.

Next, the name of the input model must be specified in the Kermeta transformation. This is done by adding the name of the input model in the loadInputModel() operation of transformation.kmt.

```
operation loadInputModel() : Resource is do
    var repository : EMFRepository init EMFRepository.new
    repository.ignoreAllLoadErrors := true
    var resource : Resource init repository.createResource("platform:/resource/UML2RDB-Transformation/UML2RDB-Transformation/model/IEC61508.uml",
                "platform:/plugin/org.eclipse.uml2.uml/model/UML.ecore")
    resource.load
    stdio.writeln("Input model loaded")

    result := resource
end
```

Figure A.6: Specify the input model to load in transformation.kmt.

When the input model has been imported into the workspace, and its name specified in the model transformation, the transformation can be run. This is done by right-clicking transformation.kmt, and selecting “Run As”, then “Run as Kermeta Application”.

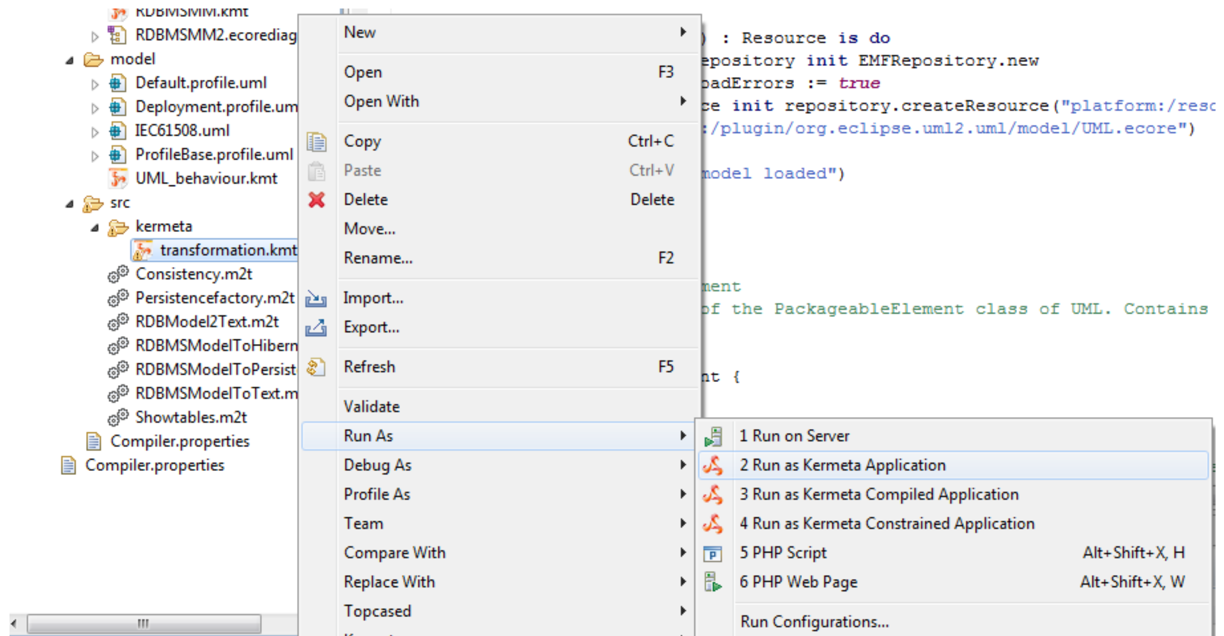


Figure A.7: Running the Kermeta transformation.

The transformation will then start transforming the elements of the input model, outputting text to the console as it runs.

```
Tasks | Properties | Console | Progress | EMF registered packages
transformation.kmt_uml___Main_main [Kermeta Application] platform:/resource/UML2RDB-Transformation/UML2RDB-Transformation/src/kermeta/transformation.kmt run
Transforming input Association: part of.
Transforming input Association: succeeds.
Transforming input Association: based on.
Transforming input Association: requires.
Transforming input Association: pertains to2.
Transforming input Association: produces.
Transforming input Association: pertains to1.
--------------------------------------
Transformed a total of 51 classes.
Transformed a total of 46 associations.
Transformed a total of 6 associationclasses.
Transformed a total of 8 enumerations.
--------------------------------------
```

Figure A.8: Example of text output from the Kermeta transformation.

Once the transformation is completed, the output is stored in the file “out.xmi”, in the “model” folder of the project. An example of the contents of “out.xmi” is shown in Figure A.9.

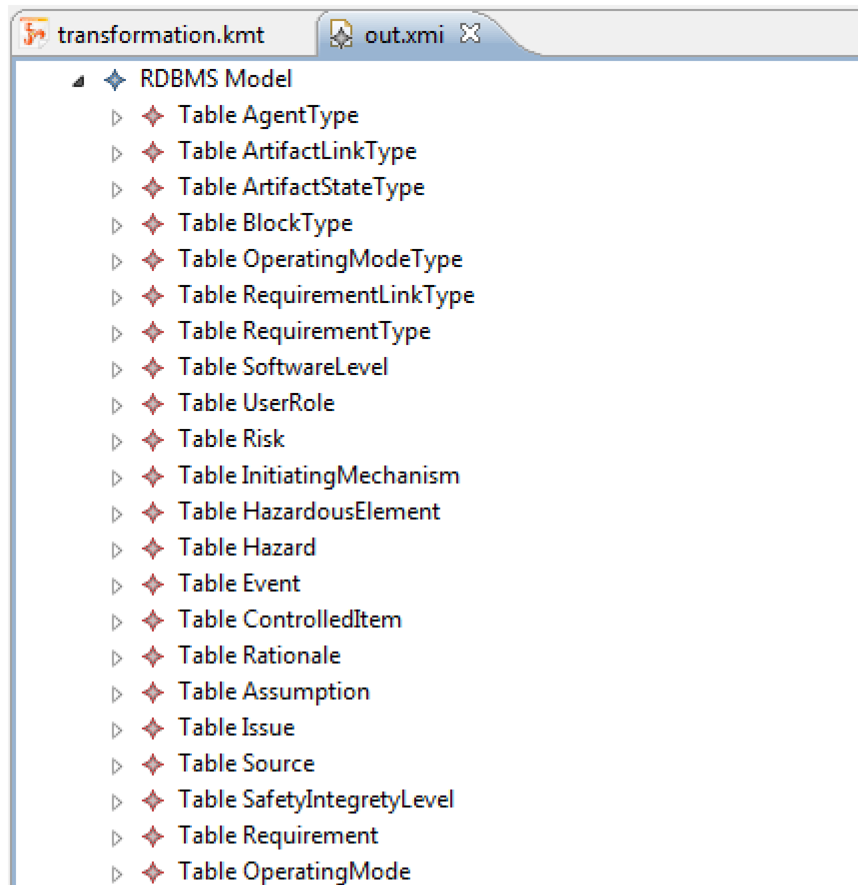


Figure A.9: Example of the contents of “out.xmi”.

This model is in turn used as the input for the model-to-text MOFScript transformations.

In the “src”-folder of the project, there are several MOFScript model-to-text transformations, each producing different artifacts used in the information repository. The model-to-text transformations can all be executed by running one transformation, “RDBMSModelToText.m2t”, which initiates the execution of the other model-to-text transformations by method calls. To execute this transformation, open it in the Eclipse editor. Upon opening a MOFScript file, a toolbar for compiling and running MOFScript transformations appear above the editor. The toolbar is shown in Figure A.10.

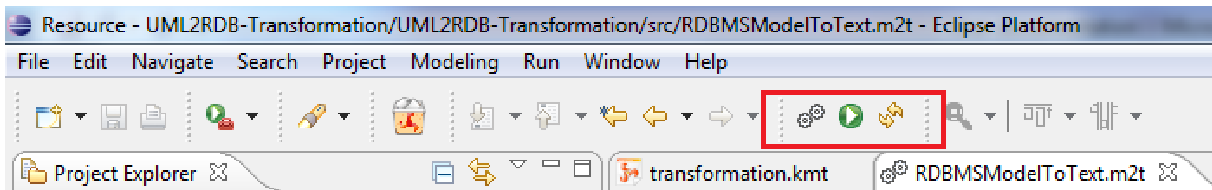


Figure A.10: The MOFScript toolbar.

Before running the model-to-text transformations, some setup work is required:

1. Unzip the “InformationRepository.zip” archive file to a chosen folder.
2. Open “RDBMSModelToText.m2t” in the Eclipse editor.
3. Change the value of the variable “baseFolder” to the folder chosen in step 1.
4. Set the package name of the information repository in the variable “package”.
5. Set the variable “packageFolder” to be the same as the package name, only with “/” as separators, instead of “.”.

An example of steps 2-5 is shown in Figure A.11.

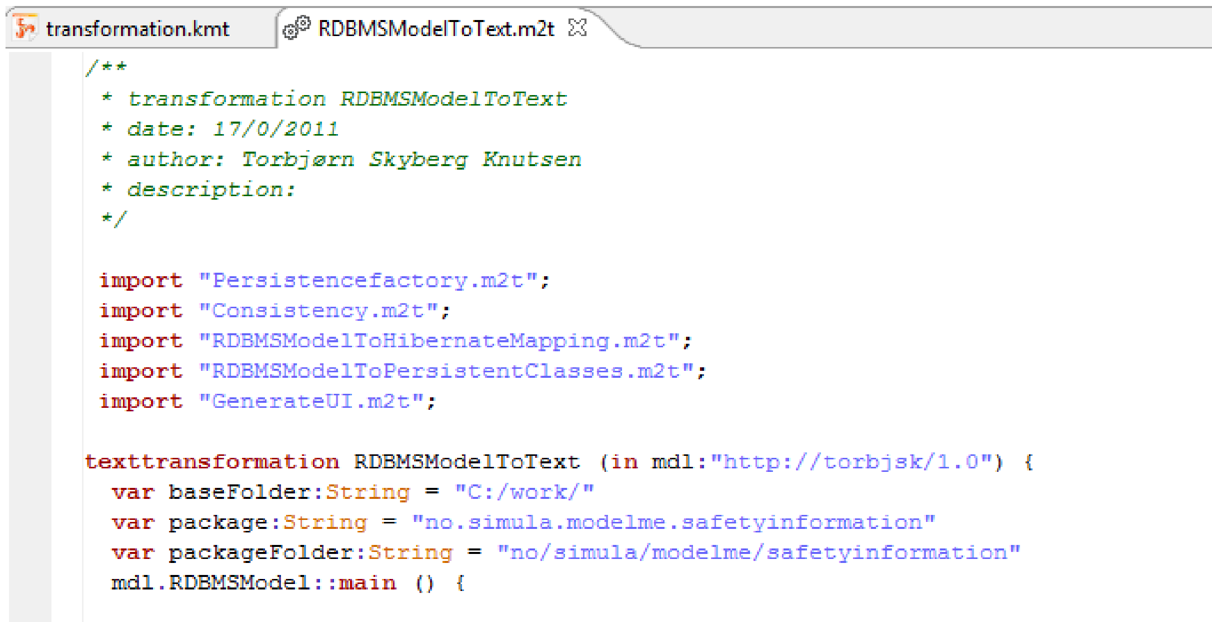


Figure A.11: Setting up the model-to-text transformations.

These variables are used to define the package of the output Java-files, as well as the file paths for the output files. In this case, the base folder where the code for the repository will be stored is defined as “C:/work/”. The base package name is “no.simula.modelme.safetyinformation”, with a corresponding folder structure defined in the “packageFolder” variable.

Once this is done, the model-to-text transformations can be run. This is done by clicking the middle (play) button of the MOFScript toolbar. Then choose the input file to be used in the transformation, in this case “out.xmi”, which is the output of the model-to-model transformation.

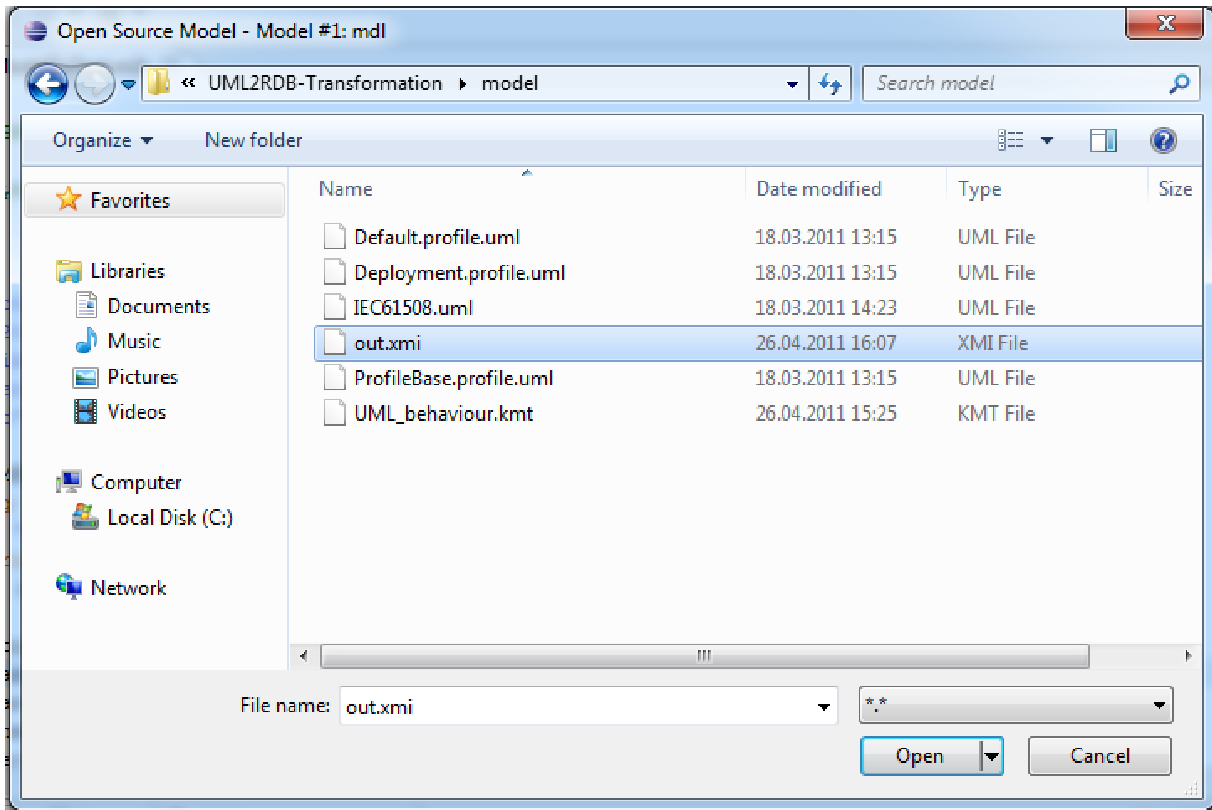


Figure A.12: Select input file for model-to-text transformations.

The model-to-text transformation produces a number of text files, into subfolders of the folder defined as the base folder. This folder will contain the Eclipse project with the code of the information repository. In order to import this project into Eclipse, we must first specify the name of the project. This is done by editing the ".project"-file found in the root of the base folder (from "InformationRepository.zip"). We change the value of the name-tag to specify the name of the project. (The name that is given will form part of the URL when publishing the repository on a web-server.)

```
<projectDescription>
  <name>safetyinformation</name>
  <projects/>
  <buildSpec>
    <buildCommand>
        <name>org.eclipse.wst.jsdt.core.javascriptValidator</name>
        <arguments>
        </arguments>
    </buildCommand>
    <buildCommand>
```

Figure A.13: Give the Eclipse project a name.

Now the project is ready to be imported into Eclipse. This is done by selecting “File”, then “Import”, then “Import existing projects into workspace” (see Figure A.2).

Then, specify the root directory of the project, and import it into the workspace.

.

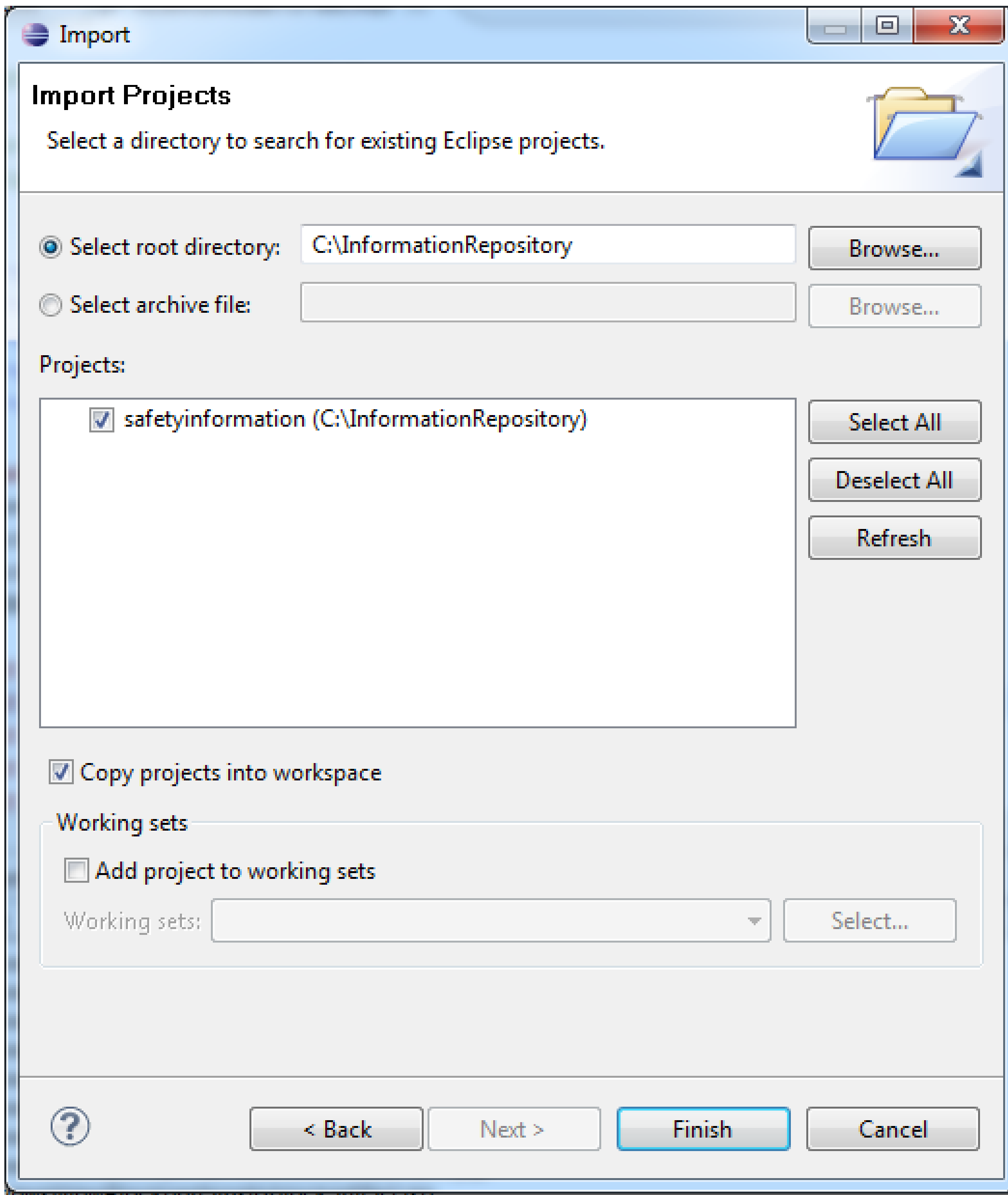


Figure A.14: Specify the root directory of the project.

Now, the project is imported into Eclipse, ready to be published on a web-server.

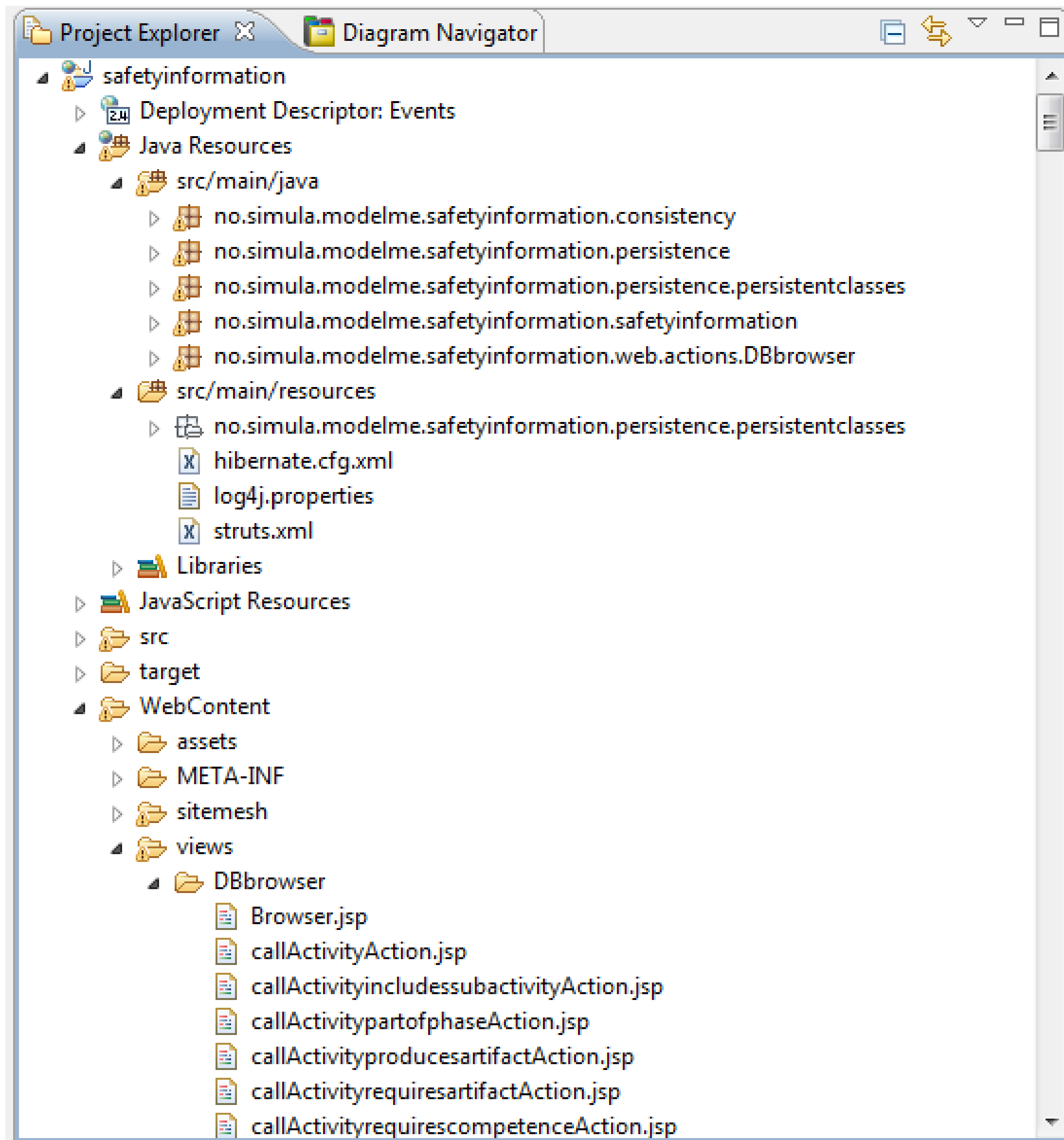


Figure A.15: Outline of the Eclipse project containing the repository code.

In order to publish the repository as a website, a server is needed. In this project, Apache Tomcat was used. To create a new server, open the "Servers" view of Eclipse, shown in Figure A.16.

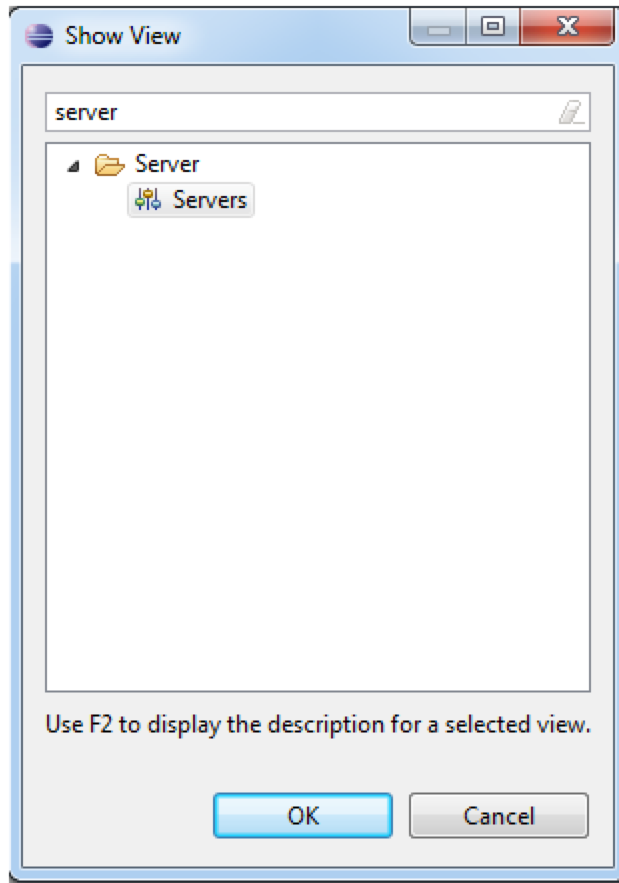
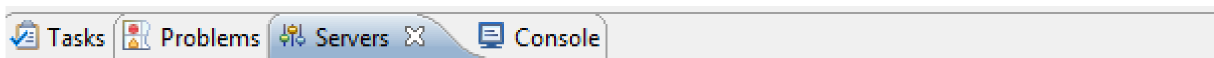


Figure A.16: Open the servers view.

In the Servers view, right-click and select “New”, then “Server”.



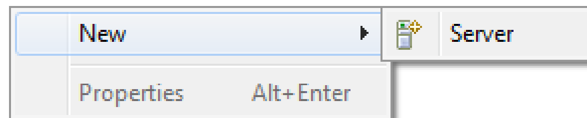


Figure A.17: Create a new server.

Select “Tomcat v6.0 Server”, and then “Next”.

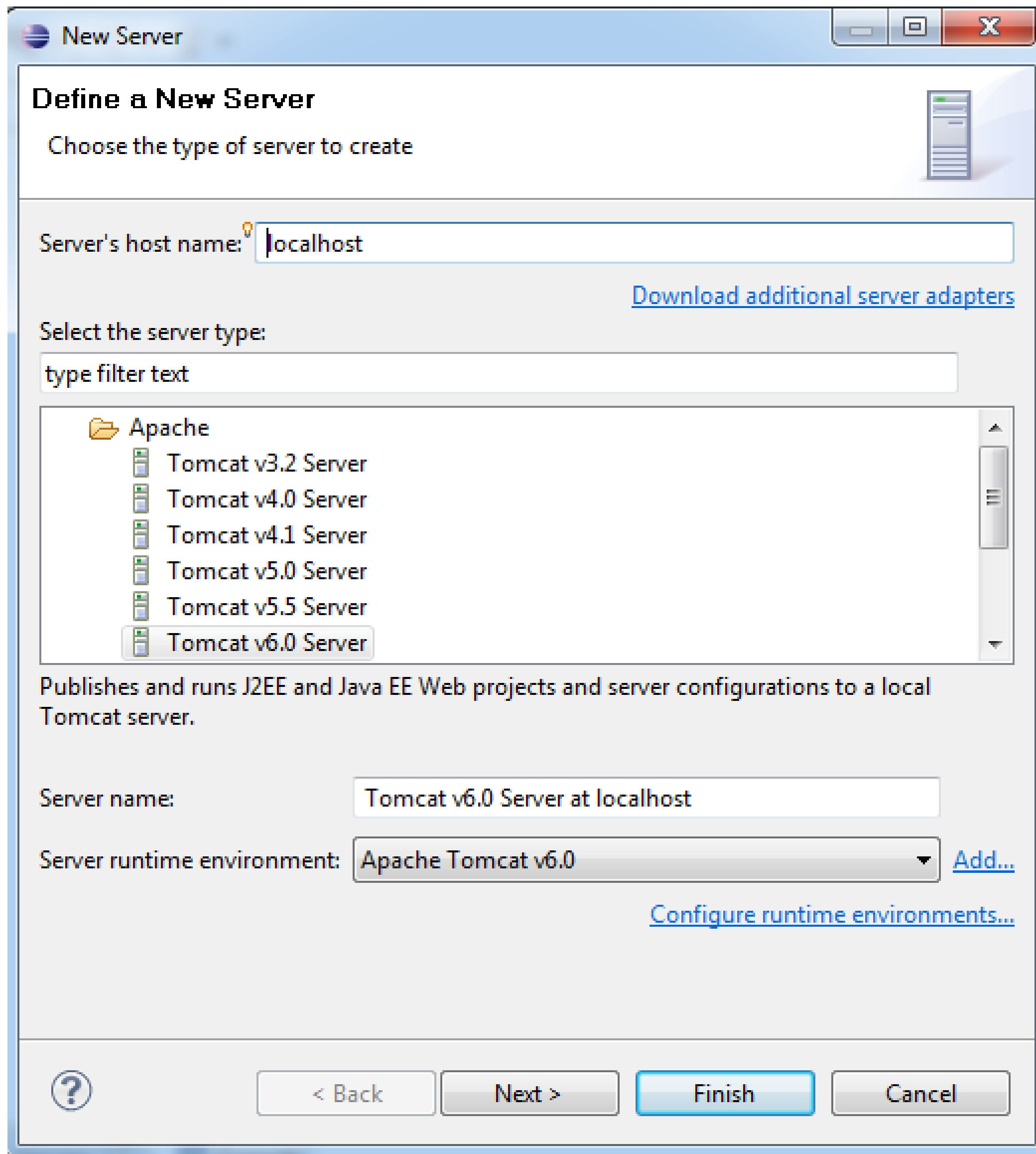


Figure A.18: Select a server.

Then, add the project to the list of configured resources, as shown in Figure A.19, and click Finish.

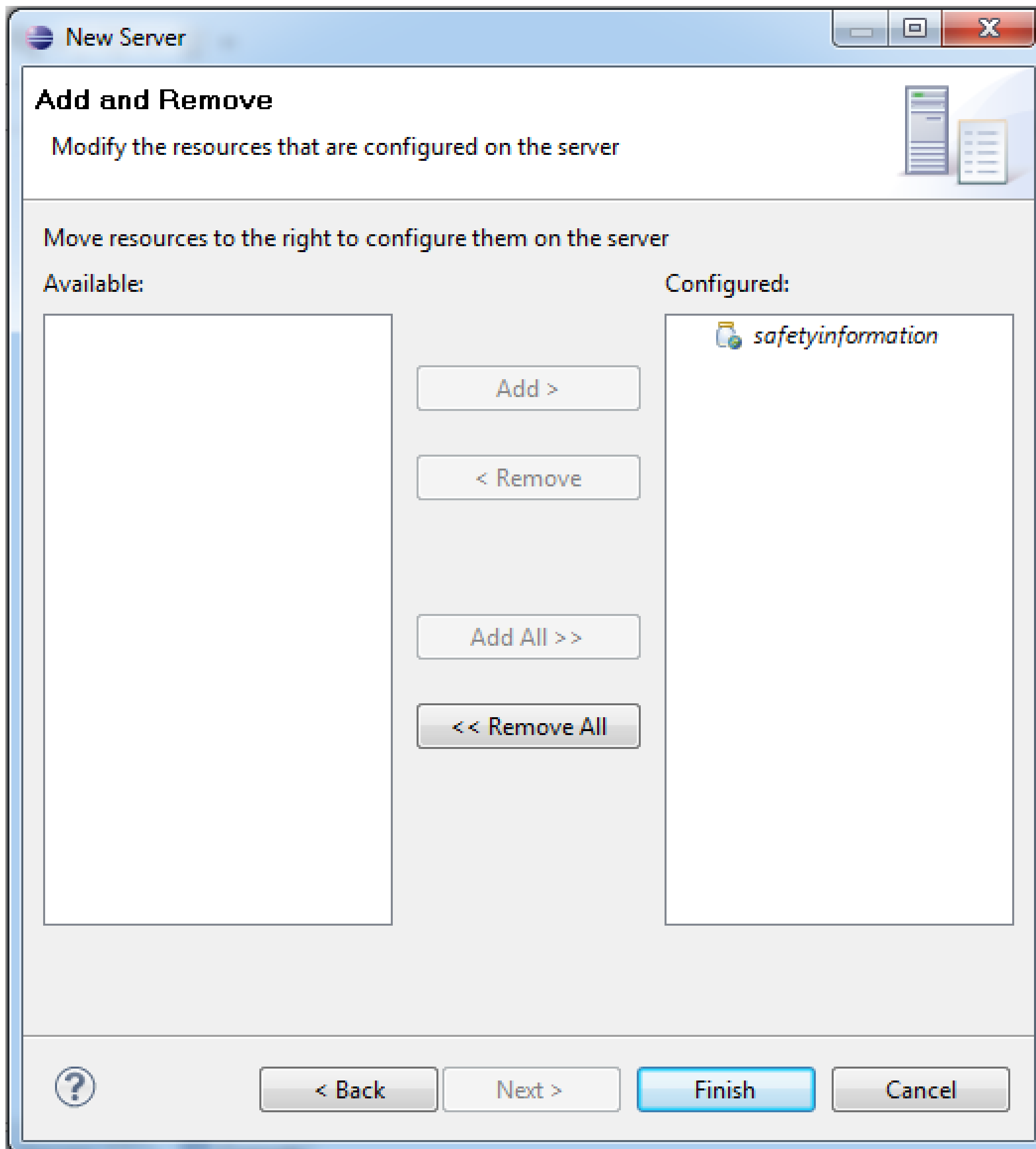


Figure A.19: Add the project to the list of configured resources.

Now, the server is shown in the “Servers” view. Right click the server and select “Start”.

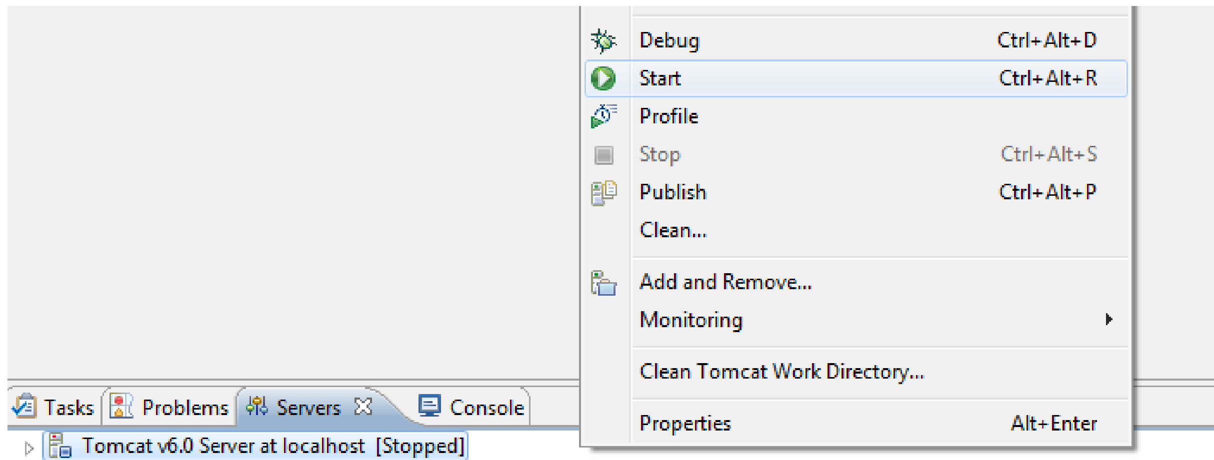


Figure A.20: Start the server.

The server will now publish the application, making it locally available through a web-browser. If all goes well, the status of the server will change from "Stopped", to "Started, Synchronized", as shown in Figure A.21.

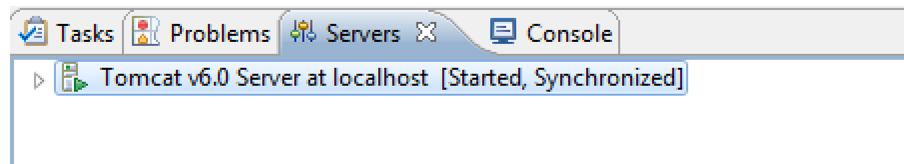


Figure A.21: The server has started, and published the application.

A.2.2 Information Repository

Once the model transformations have been executed, and the information repository system assembled and published on a web-server, the user can start adding and manipulating data in the repository, through the use of a web-based user interface. In order to access this user interface, the user must open a web browser, and input an URL in the address bar. The URL is of the form “http://{server address}:{port}/{name of project}”, where server address is the IP-address or URL to the server, port is the port defined in the Tomcat servlet container, and name of project is the name of the Eclipse project holding the repository.

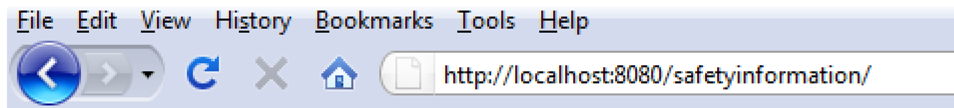


Figure A.22: Example of URL to access the repository

Upon loading the URL, the web-page shown in Figure A.23 is shown.

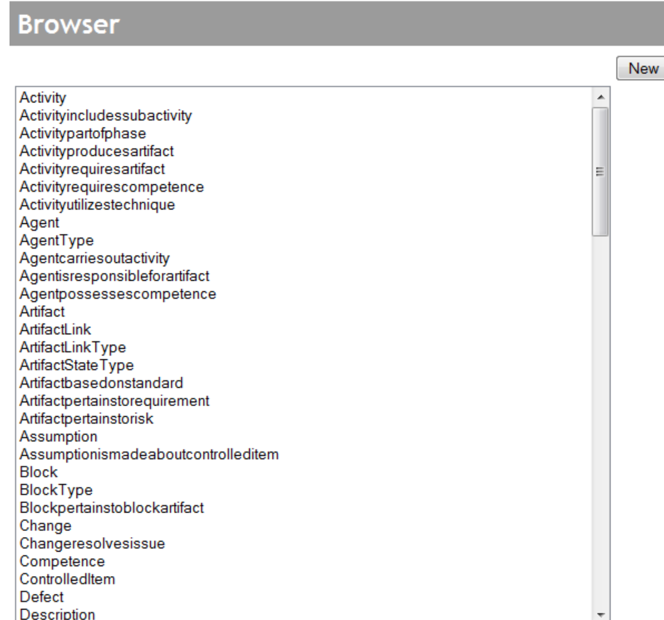


Figure A.23: The user interface of the database browser.

At the top of the page, there are two links, one for browsing the database (which is what is already showing), and one for running the consistency check. Below these links is a headline, telling the user what page is showing. Below this, the user interface is divided vertically, where the left side contains a list of the tables in the database, and the right side is where the entries in the selected table will be shown.

The rest of this section contains an example of use of the information repository.

If the user wants to browse the entries of a table, this can be done by clicking the table name in the list on the left.

Figure A.24: Entries of the “AgentType” table.

Here, the table “AgentType” is selected. When this is done, the entries of the table will be shown on the right. Since there are no entries in the table yet, only the names of the columns in the table are shown.

To add an element to the selected table, the user can press the “New”-button. Upon doing this, the entries of the table are replaced with a form for adding a new entry, shown in Figure A.25.

Browse database Consistency check

Browser

New

Activity
Activityincludessubactivity
Activitypartofphase
Activityproducesartifact
Activityrequiresartifact
Activityrequirescompetence
Activityutilizestechnique
Agent
AgentType
Agentcarriesoutactivity
Agentisresponsibleforartifact
Agentpossessescompetence
Artifact
ArtifactLink
ArtifactLinkType
ArtifactStateType
Artifactbasedonstandard
Artifactpertainstorequirement
Artifactpertainstorisk
Assumption
Assumptionismadeaboutcontrolleditem
Block
BlockType
Blockpertainstoblockartifact
Change
Changeresolvesissue
Competence
ControlledItem
Defect
Description

Description: This agent type denotes a software developer

Name: Software developer

Cancel Submit

Figure A.25: Form for adding an entry in the “AgentType” table.

Here, the user may input values for the columns in the new entry, and either submit to add a new entry, or cancel to go back to browsing. If the entry is added to the table, it is shown when selecting the table in the list, as shown in Figure A.26.

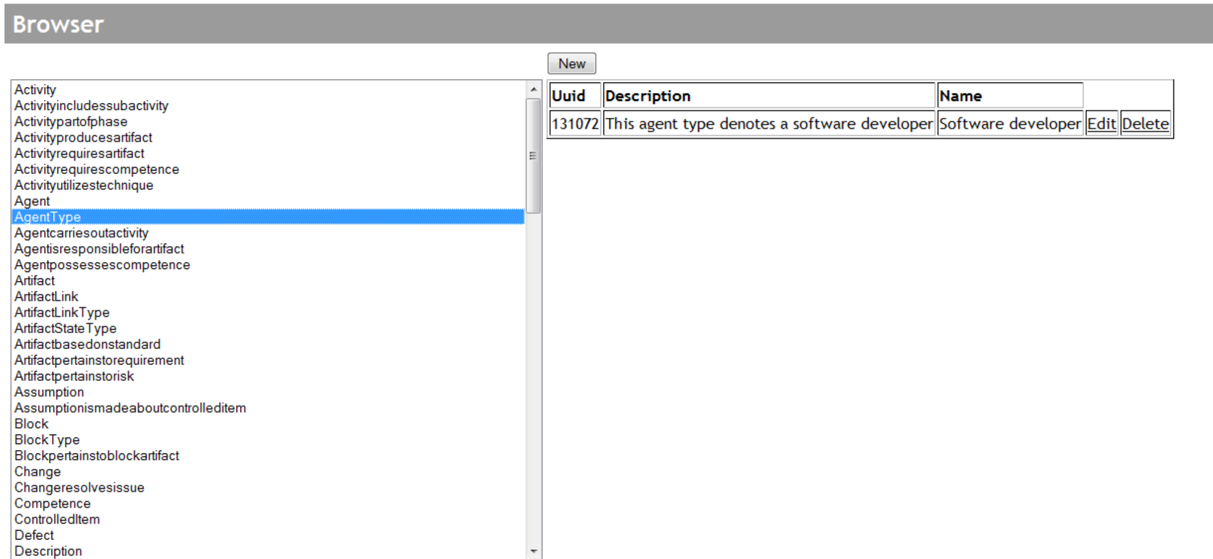


Figure A.26: The added entry in the “AgentType” table.

Since “AgentType” corresponds to an enumeration, entries in this table can be used as values for columns of other tables, in this case the “Agent” table.

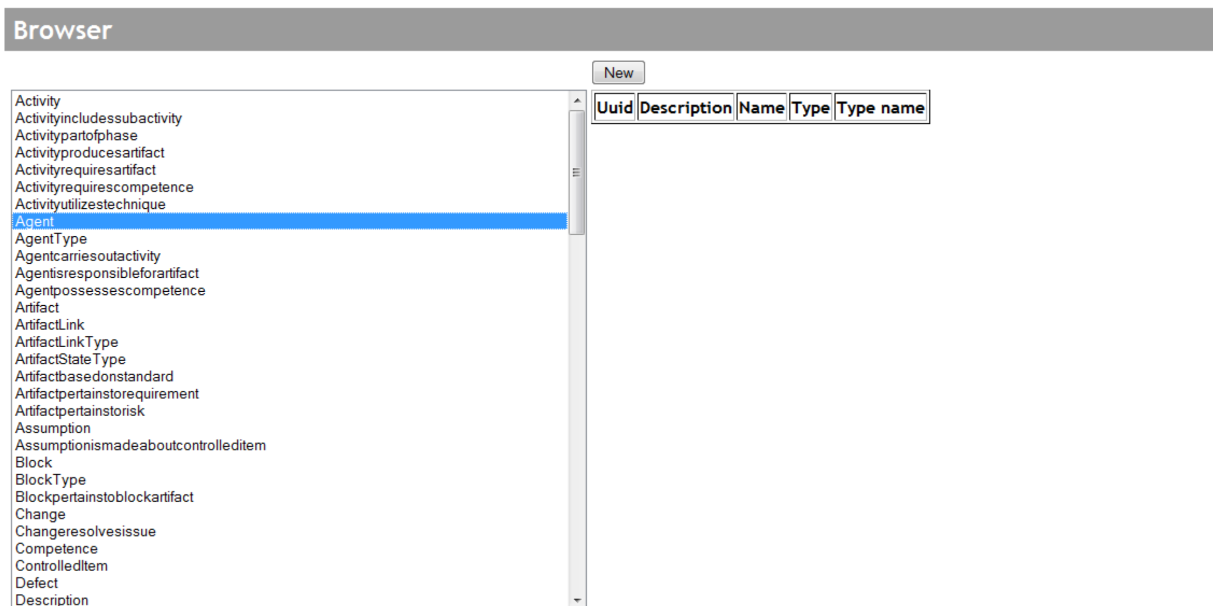


Figure A.27: View of the (empty) “Agent” table.

The “Agent” table does not contain any elements yet, so only its attributes is showing. In addition to the mandatory attributes, two other attributes are shown; Type and Type name. These both represent the same column, which contains a reference to an entry in the “AgentType” table. The “Type” field will contain the uuid of the “AgentType” entry referenced in the “Agent” entry, and the “Type name” field will contain the name of this “AgentType” entry. When adding a new entry to the “Agent” table, a form similar to that of “AgentType” is shown, with the addition of a drop-down select box for selecting an entry in the “AgentType” table. This is shown in Figure A.28.

Browse database Consistency check

Browser

New

Activity
Activityincludessubactivity
Activitypartofphase
Activityproducesartifact
Activityrequiresartifact
Activityrequirescompetence
Activityutilizestechnique
Agent
AgentType
Agentcarriesoutactivity
Agentisresponsibleforartifact
Agentpossessescompetence
Artifact
ArtifactLink
ArtifactLinkType
ArtifactStateType
Artifactbasedonstandard
Artifactpertainstorequirement
Artifactpertainstorisk
Assumption
Assumptionismadeaboutcontrolleditem
Block
BlockType
Blockpertainstoblockartifact
Change
Changeresolvesissue
Competence
ControlledItem
Defect
Description

Description: John is a software developer.
Name: John
Type: 65536 | Software developer
-Select-
65536 | Software developer
Cancel Submit

Figure A.28: Adding of a new entry in the “Agent” table.

In Figure A.28, the name and description of a new entry in the Agent table are input from the user. Additionally, an “AgentType” is selected in the “Type” field of the “Agent”. In this case, the type of the agent is “Software developer”, as added earlier. Once the entry is submitted, it can be seen when browsing the “Agent” table. This is shown in Figure A.29.

Browse database Consistency check

Browser

New

Activity
Activityincludessubactivity
Activitypartofphase
Activityproducesartifact
Activityrequiresartifact
Activityrequirescompetence
Activityutilizestechnique
Agent
AgentType
Agentcarriesoutactivity
Agentisresponsibleforartifact
Agentpossessescompetence
Artifact
ArtifactLink
ArtifactLinkType
ArtifactStateType
Artifactbasedonstandard
Artifactpertainstorequirement
Artifactpertainstorisk
Assumption
Assumptionismadeaboutcontrolleditem
Block
BlockType
Blockpertainstoblockartifact
Change
Changeresolvesissue
Competence
ControlledItem
Defect
Description

Uuid	Description	Name	Type	Type name		
2	John is a software developer.	John	65536	Software developer	Edit	Delete

Figure A.29: “Agent” table with one entry.

The added entry contains the name and description specified when adding, and also both the uuid and name of referenced “AgentType” entry is shown.

Next, we might want to associate our new agent with an entry in another table. This is done by adding an entry in one of the tables corresponding to an association. “Agent” is part of multiple associations, for instance “Agentpossessescompetence”. In order to add an entry to this table, it is first selected in the list of tables. Then, the creation of a new entry is initiated by a click on the “New”-button, and the form for adding an entry is shown.

Browse database Consistency check

Browser

New

Activity
Activityincludessubactivity
Activitypartofphase
Activityproducesartifact
Activityrequiresartifact
Activityrequirescompetence
Activityutilizestechnique
Agent
AgentType
Agentcarriesoutactivity
Agentisresponsibleforartifact
Agentpossessescompetence
Artifact
ArtifactLink
ArtifactLinkType
ArtifactStateType
Artifactbasedonstandard
Artifactpertainstorequirement
Artifactpertainstorisk
Assumption
Assumptionismadeaboutcontrolleditem
Block
BlockType
Blockpertainstoblockartifact
Change
Changeresolvesissue
Competence
ControlledItem
Defect
Description

Description: John has some competence

Name: John's competence

Agent: 2 | John

Competence: -Select-

-Select-

Cancel Submit

Figure A.30: Adding an entry in a table corresponding to an association.

We can then give the association a name, and select our agent “John” as one side of the association. When trying to select an entry for the other side of the association, which is referencing an entry in the “Competence” table, a problem occurs: There are no entries in the “Competence” table. Still, it is possible to save the entry, without a value in the column referencing an entry in the “Competence” table. The added entry is shown in Figure A.31.

Browse database Consistency check

Browser

New

Activity
Activityincludessubactivity
Activitypartofphase
Activityproducesartifact
Activityrequiresartifact
Activityrequirescompetence
Activityutilizestechnique
Agent
AgentType
Agentcarriesoutactivity
Agentisresponsibleforartifact
Agentpossessescompetence
Artifact
ArtifactLink
ArtifactLinkType
ArtifactStateType
Artifactbasedonstandard
Artifactpertainstorequirement
Artifactpertainstorisk
Assumption
Assumptionismadeaboutcontrolleditem
Block
BlockType
Blockpertainstoblockartifact
Change
Changeresolvesissue
Competence
ControlledItem
Defect
Description

Uuid	Description	Name	Agent	Agent name	Competence	Competence name		
98304	John has some competence	John's competence	2	John			Edit	Delete

Figure A.31: An (incomplete) entry in a table corresponding to an association.

If we now choose to run a consistency check over the data in the repository, by clicking the “Consistency check”-link, the page shown in Figure A.32 is displayed.

Browse database Consistency check

Consistency check

This is the consistency check, used to ensure that the data in the database is consistent with the multiplicity-constraints of the database definition.
Press the 'Start' button to start the consistency check. Start

Figure A.32: Page for initiating the consistency check.

Here, we can initiate the consistency check by clicking the “Start”-button. When we click the “Start”-button, a confirmation dialogue pops up, shown in Figure A.33.

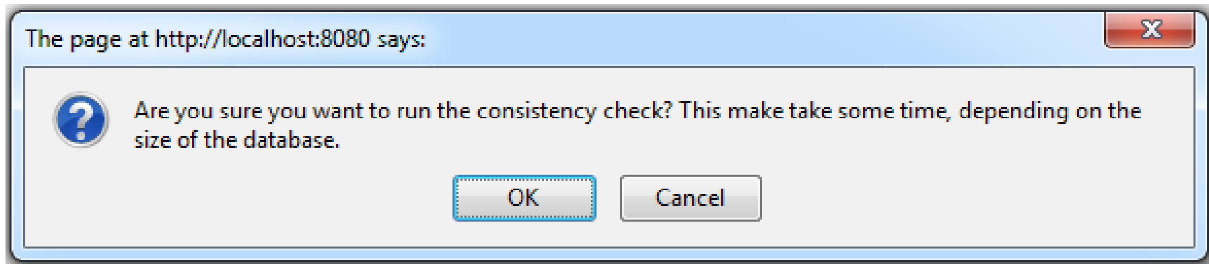


Figure A.33: Confirmation dialogue for running the consistency check.

Depending on the size of the database, the consistency check might take several minutes. In this case, with only two entries in the database, it is executed fast, and a report is presented. The resulting report is shown in Figure A.34.

Browse database **Consistency check**

Insert title here

The consistency check has been run.

Results:

Failed consistency check in table Agentpossessescompetence, UUID = 98304. NULL value in connector end competence.

Figure A.34: Consistency check report.

The report from the consistency check tell us that there was an inconsistency in the database, more specifically a null-value in the connector end "competence", in an entry of the table "Agentpossessescompetence", with uuid 98304. This is considered an inconsistency, since it is meaningless to have an association with only one of its connector ends defined. The consistency check also checks that the multiplicity constraints of the UML class diagram used as the basis for the data structure of the repository are upheld. This means that if a connector end has an upper bound of 1, then each element of the connector end should be referenced in at most one entry in the table representing the association. If the lower bound is 1, then each entry of the referenced table must be referenced in at least one entry in the table representing the association.

To fix the inconsistency uncovered by the consistency check, we have two options: (1) delete the entry causing the inconsistency, and (2) make the entry consistent by adding a reference to an entry in the "Competence" table. An entry can be deleted by clicking the "Delete"-link to the right of the entry when browsing the table (see Figure A.31).

In this example we chose the second approach, and in order to make the data consistent, we need to create an entry in the "Competence" table. The adding of this entry is shown in Figure A.35.

Browse database Consistency check

Browser

New

Activity
Activityincludessubactivity
Activitypartofphase
Activityproducesartifact
Activityrequiresartifact
Activityrequirescompetence
Activityutilizestechnique
Agent
AgentType
Agentcarriesoutactivity
Agentisresponsibleforartifact
Agentpossessescompetence
Artifact
ArtifactLink
ArtifactLinkType
ArtifactStateType
Artifactbasedonstandard
Artifactpertainstorequirement
Artifactpertainstorisk
Assumption
Assumptionismadeaboutcontrolleditem
Block
BlockType
Blockpertainstoblockartifact
Change
Changeresolvesissue
Competence
ControlledItem
Defect
Description

Description: Entry for competence in Java programming

Name: Java programming

Cancel Submit

Figure A.35: The adding of an entry in the "Competence" table.

Once we have a "Competence" entry to reference, we can go back to the "Agentpossessescompetence" table and edit our entry to make it reference the new entry in the "Competence" table. This is done by clicking the table name to show the entries, and then selecting the "Edit" link to the right of the entry in question. Once this is done, the form used for adding new entries is shown, populated with the information stored in the entry. This is shown in Figure A.36.

Figure A.36: Editing an entry in the “Agentpossessescompetence” table.

Here we can select our newly added entry in the “Competence” table, and then save the entry by clicking submit. Once this is done, the changes are reflected when browsing the table.

Figure A.37: The updated entry in the “Agentpossessescompetence” table.

If we now re-run the consistency check, we get the report shown in Figure A.38, saying that no inconsistencies were found in the database.

Browse database Consistency check

Insert title here

The consistency check has been run.

Results:

No inconsistencies found.

Figure A.38: Report after the re-run of the consistency check.

This concludes the example of use of the tools developed during the work described in this thesis.

List of Figures

Figure 1: The architecture of the information repository.
Figure 2: The main program flow of the model transformations.
Figure 3: RDBMSMM.ecore.
Figure 4: Parts of a UML class diagram.
Figure 5: Parts of an instance of RDBMSMM.ecore.
Figure 6: Program flow of the model-to-model transformation.
Figure 7: Artifacts generated by MOFScript transformations.
Figure 8: Program flow of the MOFScript transformations (2 main variants).
Figure 9: User roles in the transformation and the information repository.
Figure 10: Overview of the technologies used and the context of their use.
Figure A.1: The archive file containing the source code.
Figure A.2: Import existing project into workspace.
Figure A.3: Import UML2RDB-Transformation.zip into the Eclipse workspace.
Figure A.4: Outline of the Eclipse project containing model transformations.
Figure A.5: Import input models into workspace.
Figure A.6: Specify the input model to load in transformation.kmt.
Figure A.7: Running the Kermeta transformation.
Figure A.8: Example of text output from the Kermeta transformation.
Figure A.9: Example of the contents of “out.xmi”.
Figure A.10: The MOFScript toolbar.
Figure A.11: Setting up the model-to-text transformations.
Figure A.12: Select input file for model-to-text transformations.
Figure A.13: Give the Eclipse project a name.
Figure A.14: Specify the root directory of the project.
Figure A.15: Outline of the Eclipse project containing the repository code.
Figure A.16: Open the servers view.
Figure A.17: Create a new server.

Figure A.18: Select a server.
Figure A.19: Add the project to the list of configured resources.
Figure A.20: Start the server.
Figure A.21: The server has started, and published the application.
Figure A.22: Example of URL to access the repository
Figure A.23: The user interface of the database browser.
Figure A.24: Entries of the “AgentType” table.
Figure A.25: Form for adding an entry in the “AgentType” table.
Figure A.26: The added entry in the “AgentType” table.
Figure A.27: View of the (empty) “Agent” table.
Figure A.28: Adding of a new entry in the “Agent” table.
Figure A.29: “Agent” table with one entry.
Figure A.30: Adding an entry in a table corresponding to an association.
Figure A.31: An (incomplete) entry in a table corresponding to an association.
Figure A.32: Page for initiating the consistency check.
Figure A.33: Confirmation dialogue for running the consistency check.
Figure A.34: Consistency check report.
Figure A.35: The adding of an entry in the “Competence” table.
Figure A.36: Editing an entry in the “Agentpossessescompetence” table.
Figure A.37: The updated entry in the “Agentpossessescompetence” table.
Figure A.38: Report after the re-run of the consistency check.

References

(All referenced websites were accessed 29/4/2011.)

1: Panesar-Walawege, R.K., Sabetzadeh, M., Briand, L., and Coq, T. (2010) *Characterizing the Chain of Evidence for Software Safety Cases: A Conceptual Model Based on the IEC 61508 Standard*, in 3rd IEEE International Conference on Software Testing, Verification, and Validation (ICST'10), Paris, France, April 2010.
http://modelme.simula.no/assets/ICST10.pdf

2: http://www.iec.ch/functionalsafety/

3: http://www.uml.org/

4: http://www.iec.ch/index.htm

5: Redmill, F. (2000) *Installing IEC 61508 and Supporting Its Users - Nine Necessities*, in the Fifth Australian Workshop on Safety Critical Systems and Software, Melbourne, Australia, 24 November 2000

http://www.csr.ncl.ac.uk/FELIX_Web/4B.IEC 61508 Nine Necessities.pdf

6: http://kermeta.org/

7: http://www.eclipse.org/gmt/mofscript/

8: http://www.java.com/

9: http://www.hibernate.org/

10: http://struts.apache.org/

11: http://www.hibernate.org/about

12: http://db.apache.org/derby/

13: http://www.oracle.com/technetwork/java/javaee/jsp/index.html

14: http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html

15: http://www.kermeta.org/examples/ (class2RDBMS example)

16: http://www.springsource.org/

17: http://modeling-languages.com/content/uml2db-full-code-generation-sql-scripts-databases

18: http://umt-qvt.sourceforge.net/

19: http://www.visual-paradigm.com/product/dbva/

20: Garcia-Molina, H., Ullman, J.D. and Widom, J. (2002) *Database Systems: The Complete Book.* New Jersey: Prentice Hall (pp. 65-80)

21: Garcia-Molina, H., Ullman, J.D. and Widom, J. (2002) *Database Systems: The Complete Book.* New Jersey: Prentice Hall (pp. 23-59)

22: http://www.eclipse.org/

23: http://www.eclipse.org/org/

24: http://www.eclipse.org/downloads/

25: http://kermeta.org/download/

26: http://www.irisa.fr/triskell

27: http://en.wikipedia.org/wiki/IRISA

28: http://en.wikipedia.org/wiki/Kermeta

29: http://www.eclipse.org/gmt/mofscript/download/

30: http://www.eclipse.org/gmt/mofscript/about.php

31: http://java.com/

32: http://java.com/en/about/

33: http://en.wikipedia.org/wiki/Java_(programming_language)#Principles

34: http://maven.apache.org/

35: http://struts.apache.org/

36: http://tomcat.apache.org/

37: http://en.wikipedia.org/wiki/JavaScript

38: http://www.opensymphony.com/sitemesh/

39: E. Gamma, R. Helm, R. Johnson, J. M. Vlissides: “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995 (Chapter 4)

40: http://www.oracle.com/technetwork/java/javaee/jsp/index.html

41: http://www.kermeta.org/documents/user_doc/manual/

42: https://gforge.inria.fr/forum/?group_id=32

43: http://docs.jboss.org/hibernate/core/3.5/reference/en-US/pdf/hibernate_reference.pdf

44: https://forum.hibernate.org/

45: http://www.scribd.com/doc/25244173/Java-Struts-Spring-Hibernate-Tutorial-github-com-chrishulbert-JavaTutorial

46: http://www.vaannila.com/struts-2/struts-2-example/struts-2-crud-example-1.html

47: http://struts.apache.org/2.2.1/docs/tutorials.html