HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Developing a graphical Editor for Entitiy-Relationship Models using DSLs

Stephan Hagios

Konstanz, 31.08.2011

BACHELORARBEIT

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der

Hochschule Konstanz

Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang Software-Engineering

Thema: **Developing a graphical Editor for Entitiy-Relationship Models using DSLs**

Bachelorkandidat: Stephan Hagios, Mosbruggerstraße 9, 78462 Konstanz

1. Prüfer: Prof. Dr. Marko Boger
2. Prüfer: Prof. Dr. Oliver Eck

Ausgabedatum: 31.05.2011
Abgabedatum: 31.08.2011

Zusammenfassung (Abstract)

Thema:	Developing a graphical Editor for Entitiy-Relationship Models using DSLs
Bachelorkandidat:	Stephan Hagios
Firma:	HTWG
Betreuer:	Prof. Dr. Marko Boger Prof. Dr. Oliver Eck
Abgabedatum:	31.08.2011
Schlagworte:	Entity Relationship Editor, Domain Specific Language, DSL, Model Driven Software Development, MDSD, Model Driven Development, MDD, Eclipse Modeling Framework, EMF

The present thesis outlines the development of an Entity Relationship Editor (ER-Editor) using Poseidon for DSL, for both implementation and the design. Poseidon for DSL is a professional tool for developing graphical Domain Specific Languages (DSLs). It provides an easy way to generate code for a graphical editor. The development of the ER-Editor will be conducted utilising the concept of Model Driven Software Development (MDSD). The created ER-Editor supports standard functionalities of entity relationship modelling, like creating entities, attributes and relationships. Furthermore, it is also possible to choose the cardinality of a relationship between two entities and the data type of an attribute.

The second focus of the thesis lies on the creation of a code generator with which it is possible to create SQL code out of an ER diagram. The editor is able to generate the whole SQL code which is necessary for creating tables within a relational database automatically.

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Stephan Hagios, geboren am 03.07.1985 in Freiburg, im Breisgau*, dass ich

(1) meine Bachelorarbeit mit dem Titel

Developing a graphical Editor for Entitiy-Relationship Models using DSLs

bei der HTWG unter Anleitung von Prof. Dr. Marko Boger selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 31.08.2011

(Unterschrift)

Contents

List of Figures **5**

Listings **6**

List of Abbreviations **7**

1 Introduction **8**

2 Technical concepts **11**
- 2.1 The Entity Relationship Model . . . 11
- 2.2 Model Driven Software Development . . . 14
- 2.3 Domain Specific Languages . . . 15
- 2.4 The Eclipse Modeling Framework . . . 16

3 Generating an ER-Editor using DSLs **19**
- 3.1 Setting up the environment . . . 19
- 3.2 Creating the meta model for the editor . . . 20
- 3.3 Creation of all needed Elements in the Model . . . 24
  - 3.3.1 Creation of the needed Nodes and Edges . . . 25
  - 3.3.2 Adding the Elements as Tools . . . 27
  - 3.3.3 Adding Rapid Buttons . . . 28
- 3.4 Customising the Look of all Elements . . . 30
- 3.5 Modifying the ER-Editor . . . 35
- 3.6 Limitations of the ER-Editor . . . 38

4 Generating SQL Statements from an ER-Model **40**
- 4.1 Adjust the Ecore meta model . . . 40
- 4.2 Preparing the Editor . . . 42
- 4.3 Writing the Generator . . . 46
- 4.4 Limitations of the code generation . . . 50

5 Conclusion **51**

Bibliography **54**

List of Figures

2.1 Sample of an Entity Relationship Diagram 11
2.2 The different attribute types 12
2.3 A sample 1:1 cardinality demonstration 13
2.4 A sample 1:n cardinality demonstration 14
2.5 EMF unifies XML, Java and UML; Source: [Steinberg, 2009] . . . 16
2.6 Hierarchy of the Ecore Components;
Source: [IBM Corporation and others, 2011] 17
2.7 The EMF editor for Ecore files 18

3.1 The two EEnums of the meta model 21
3.2 Meta model of the Entity Relationship Editor 23
3.3 Empty Poseidon for DSL 24
3.4 The icons for multi valued attributes, key attributes and relationships 26
3.5 The ER-Editor after defining all models 28
3.6 An entity with its Rapid Buttons 30
3.7 The ER-Editor when it is finished. 38

4.1 The prepared entity-relationship-editor.ecore file 41

Listings

2.1 A sample CREATE TABLE command for an Oracle database . . 13
3.1 The defined nodes in the model 25
3.2 The defined edges in the model. 26
3.3 The defined tools in the model. 27
3.4 Edge rules in the file EdgeRules.erule 28
3.5 Definition of the Rapid Buttons 29
3.6 The code for the adjustment of the attribute. 31
3.7 The code for the underlined text inside the key attribute. Source: [Künneth, 2010] 32
3.8 The code for the double ellipse of the multi valued attributes. . . 32
3.9 The code for painting the rhombus of a relationship. 33
3.10 The code for displaying the cardinality. 34
3.11 The help method getLabel. 35
3.12 The method rejectElementFromTree in CustomTreeMediator.java 35
3.13 The overwritten methods checkStart and checkEnd 36
3.14 Avoiding the Rapid Buttons 37
4.1 The ResourceIds for the new menu and menu item 42
4.2 The setupMenu method in the file CustomMainMenu.java 42
4.3 The Pop-Up in GenerateOutputFromModelActionListener.java . 43
4.4 The containers for the attributes. 44
4.5 The container for the key attributes. 45
4.6 The container for the multi valued attributes 45
4.7 The code for storing the two connected entities in the relationship, in the file LineFromEntityToRelationshipArcGem.java. . . . 46
4.8 The code for creating an empty SQL file with the generator. . . . 47
4.9 The entity define block in the generator. 48
4.10 The define block for the relationships "1:1" and "n:m". 49

List of Abbreviations

DDL .... Data Definition Language
DML .... Data Manipulation Language
DSL .... Domain Specific Language
EMF .... Eclipse Modeling Framework
ER .... Entity Relationship
ER-Editor .... Entity Relationship Editor
GPL .... General Purpose Language
IDE .... Integrated Development Environment
IT .... Information Technology
JDK .... Java Development Kit
MDA .... Model Driven Architecture
MDD .... Model Driven Development
MDSD .... Model Driven Software Development
OMG .... Object Management Group
OSGi .... Open Services Gateway initiative framework
SDK .... Software Development Kit
SQL .... Structured Query Language
UML .... Unified Modeling Language
XMI .... XML Metadata Interchange
XML .... Extensible Markup Language

Chapter 1

Introduction

In most software development processes a model is used during the part of planning and implementation. It is utilised to visualise the architecture or the functions of a software. The developers write their code with the help of this model, using it as reference or later for the documentation.
The model specifies what a software should do, but not how it does it. This means that the developer needs to decide how the code is written, because there is no connection between the model and the software which is described by it. For example, if you want to develop an editor for entity relationship diagrams you have to define a model for the software. After this it is also necessary to write the code for the editor to get a runnable piece of software.
This could raise the question: If the software is already designed, so why does it still need to be implemented? The answer of this question is obvious, because it is not possible to execute a model: it is just a definition shown as a diagram or a text file.
A non Information Technology (IT) example for this could be a coffee machine. At this coffee machine the user has to define which type of coffee, how much sugar, how much milk and so on he wants. The plan for the coffee is now stored in the machine, but the machine is not able to prepare the coffee. The result of this is, that the user has to cook the coffee by himself and can use the coffee recipe if he does not remember how much sugar it was. Although this is really inefficient, it is the usual way in which a model gets used in a software development cycle. Considering this, Model Driven Software Development (MDSD) offers the possibility to improve the process of development, by deleting this inefficiency.
In MDSD the model is more than just a reference, it can be seen as a part of the implementation [Stahl, 2007]. The reason for this is that a model which was created with the MDSD concept can be used to create runnable code out of it. The part of the code creation also should be automatic, what means that the developer just has to define the model. In the coffee machine example this would enable the user to only push the generate button with the effect of receiving the perfect coffee, prepared exactly along the previously defined plan that is stored in the machine. The same will happen in this thesis: a model will be defined and runnable code will be generated. The aim of the present

thesis is to explain the necessary steps for the development of a graphical Entity Relationship Editor (ER-Editor) using the idea of MDSD.
At the end of the thesis a complete ER-Editor, with all necessary features, will be created. All steps are described in detail, so the reader will be able to recreate the whole editor by himself. For developing the ER-Editor the tool 'Poseidon for DSL' is used, which is a professional tool for developing Domain Specific Languages (DSL). A DSL is a formal language that is especially designed for one domain. The domain of this thesis is the entity relationship diagram. This diagram is used to represent a set of data, which is a part of the real world. The entity relationship diagram is also a model that is used to create a blueprint of a database. With the help of this model the entries of a database can be created. This is again similar to the coffee machine example, because of the fact that the user has to create the code by himself. Therefore the ER-Editor will be receiving a generator that is able to create runnable code out of the entity relationship model. This code is of the type of SQL statements, which is the language that is used to execute database queries and commands.

In the first chapter of the present thesis an overview of the concepts that will be used to develop the ER-Editor is given. First the entity relationship model will be described and it will be explained how it is possible to create SQL code out of an entity relationship model. After this the MDSD approach and its main ideas will be illustrated. The next overview is about DSLs, what they are and what different kinds of DSL exist. At the end of the first chapter the Eclipse Modeling Framework (EMF) is described, which is used for the work with the model of the ER-Editor. The used meta modelling language 'Ecore', which is a dialect of the well known Unified Modeling Language (UML), will be described in that context.
In Chapter 3 the ER-Editor will be designed and implemented. This starts with the presentation of the therefore needed environment and tools, that are necessary to create the editor. Additionally it is described how to set them up. After this step the meta model for the editor gets created, by using the above mentioned Ecore language. This works with the graphical Ecore editor 'Poseidon for ecore', which is one of the used tools in the present thesis. The next step is to define all elements of the model that are necessary for the ER-Editor. Thereafter the editor is being built for the first time with the effect of a runnable entity relationship editor. This editor is being created with the above mentioned MDSD approach: defining a model and generating runnable software. After this the ER-Editor and its elements are getting adjusted. For these steps a basic knowledge of the programming language Java is important, because the generated and written code is completely written in Java. The last section in this chapter will focus on the limitations of the ER-Editor, with what is possible and what not. Moreover, the constraints of the used tool Poseidon for DSL are getting outlined.
The next chapter illustrates the creation of a generator, which is necessary to generate SQL code from the diagram. Therefore the Ecore meta model gets adjusted, which is the before created Ecore model, that can also be used for this purpose. Before writing the generator the ER-Editor has to be customised,

which is shown in Section 4.2. After this the generator can be created. For this purpose the template language 'Xpand' is used. At the end of this chapter an overview of the constraints of the code generation can be found. With a focus on the limitations and the possibilities of the generator.
At the end a brief summary of the whole process of creating the ER-Editor will be presented and the method of Model Driven Software Development will also be reflected.

Chapter 2

Technical concepts

In this section the technical information can be found, which is necessary to follow the presented development of the editor. First the Entity Relationship Model will be explained and it will be shown how an Entity Relationship (ER) Model gets mapped to Structured Query Language (SQL) code. Also the basics of Model Driven Software Development (MDSD), Domain Specific Languages (DSL) and the Eclipse Modeling Framework (EMF) will be defined. This chapter is intended to lay the groundwork for the following chapters, in detail that will describe the development of the Entity Relationship Editor (ER-Editor).

2.1 The Entity Relationship Model

The Entity Relationship Model is used in software engineering to represent a set of data. This type of model is often used for designing a schema of a databases. It was first mentioned in a paper of Peter Chen [Chen, 1976].

There are four elements an entity relationship model consist of. First there is the entity which is pictured as a box with its given name inside. An entity often represents a real or physical object, for example a house or an employee. The following element is a relationship. A relationship is mapped as a rhombus with its given name shown inside of it. A relationship is always between two entities and describes their relation to each other. E. g. there are two entities, one is called 'Employee' and the other is called 'Project'. A possible relationship between the two could be called 'works-on', having the meaning 'an employee works on a project', as shown in Figure 2.1. The 'n' and '1' underneath the

Figure 2.1: Sample of an Entity Relationship Diagram

lines on the picture represents the cardinality. This cardinality pictures the relation between two entities. There are different types of cardinalities: the

'1:1', '1:n', 'n:1' and 'n:m'[1] cardinality. The '1' signifies that this entity just appears one time at the opposite entity while 'n' or 'm' means that it appears one or more times. In the sample of Figure 2.1, at the employee entity a 'n' is pictured, which means that one or more employees are working on a project. On the opposite of it is a '1', which shows that one employee works on only one project.
Other important elements of an entity relationship model are the attributes. An attribute can either belong to an entity or to a relationship. It is pictured as an ellipse with its given name on the inside and represents a feature of an entity or of a relationship. For example the entity employee could have an attribute called birthday, what would also imply that every employee only has one birth date. There are various types of attributes, differing in characteristics and design. One of them is the key attribute which is also pictured as an ellipse, but with its name underlined. The content of the key attribute has to be unique because it is often used to find and identify a specific entry. Referring to the employee example of Figure 2.1 this could be the personnel number, meaning that every employee comes with its own unique personnel number for identification. Another type of attributes are the so-called multi valued attributes. They are also pictured as an ellipse but with a smaller ellipse inside. If an attribute is multivalued, the attribute can appear more than one times. This could be the case when an employee has more than one telephone number. The different types of attributes are presented in Figure 2.2.

Figure 2.2: The different attribute types

The Entity Relationship Model belongs to the conceptual layer in the design of a database [Kemper and Eickler, 2009]. It does not depend on the used database, which is why the ER model can get created and used for every database. In Chapter 3 it is shown how to create a generator which generates Structured Query Language (SQL) code. This SQL code is covered for an Oracle Database[2], which has the effect that the created code just works with Oracle databases. SQL is a standardised database language that is used to create, manipulate and query data in a database. It can be divided into two languages, the Data Definition Language (DDL) and the Data Manipulation Language (DML) [Brücher et al., 2011]. For the present thesis the DML is not relevant, because it is used for manipulating, inserting and querying data of a given database. The DDL is in contrast important, because it is used for creating tables in databases, a feature that will be needed in Chapter 4. In order to create a table within a database you can use the command *CREATE TABLE* and simultaneously add the name of the entry and the attributes as it can be

[1] The colon is pronounced as 'to', e.g. 1:1 is pronounced as 1 to 1.
[2] An object-relational database management system (ORDBMS) from Oracle

seen in Listing 2.1. In general one can say, that every entity of an ER model is a table and it has to be created with the above mentioned command. Furthermore all attributes of an entity also need to be created using this command. Listing 2.1 also shows the creation of a constraint that tells the database which is the key attribute.

```
CREATE TABLE anEntity
(
  id        int   NOT NULL,
  created   date,
  CONSTRAINT pk_anEntity PRIMARY KEY(id)
);
```

Listing 2.1: A sample CREATE TABLE command for an Oracle database

There are two different types of cardinality that need to be mapped to SQL. In doing so the cardinality '1:1' and 'n:m' are treated the same way. To create tables with these cardinalities a new table needs to be created which represents the relationship. This table holds two foreign keys, each referring to the entity to which it is connected (see Figure 2.3). The sample shows a '1:1' cardinality, as it was mentioned before, the 'n:m' cardinality is handled the same way. For

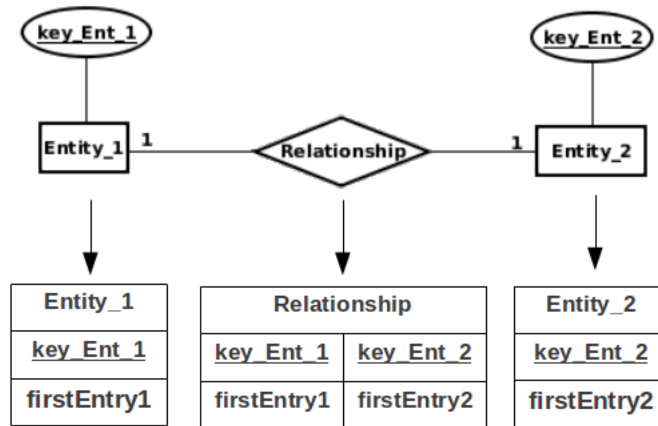

Figure 2.3: A sample 1:1 cardinality demonstration

the next type of cardinality, '1:n' or 'n:1' cardinality, an extra table is not necessary, because the entity, that displays the cardinality 'n', gets a foreign key to the opposite entity. The other entity does not know anything about this relation, like it is shown in Figure 2.4.

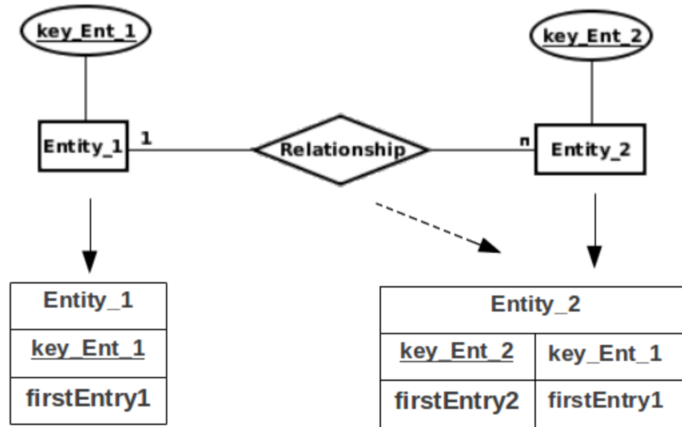

Figure 2.4: A sample 1:n cardinality demonstration

2.2 Model Driven Software Development

In a normal development cycle a model, for example a Unified Modeling Language (UML) model, is just used for the documentation or as reference for the programmer. There is no direct connection between a model and the written code. One aim of Model Driven Software Development (MDSD), also known as Model Driven Development (MDD), is to fill this gap. The idea of MDSD is to generate runnable software automatically from formal models [Stahl, 2007]. Formal models can be defined as models which should completely describe a facet of a software. In most cases this is impossible to do, which is why clear rules need to be defined that are declaring what the model describes and what not. These rules are defined in a so-called meta model, that is used to define meanings and structures of a model [Jouault and Bézivin, 2006]. The creation of a meta model is also the first step, when using the MDSD approach. The used technique to describe the model or meta model is thereby not important, because MDSD does not specify this.
The next step in the concept of MDSD is generating runnable software. MDSD offers two possibilities for doing this: either by using an interpreter or a generator. The difference between them is that the generator creates code from the model and the interpreter parses the model during the runtime. Depending on the content an action will be executed. In Section 4.3 it is shown how such a generator is created. The part where code is generated out of a model has to work automatically, what means that a model is not just for the specification - there has to be an automatic build process. As the model does not describe the whole system, there will always be code in the generated software that has to be written by hand [Stahl, 2007]. This, for example, can be seen in Section 3.4.

This sounds similar to the approach of Model Driven Architecture (MDA), which is a standard of the Object Management Group (OMG).
But MDA is a special interpretation of MDSD, where closed guidelines have been made by the OMG. For example MDA specifies to use UML in order to define a model and the Meta Object Facility (MOF) for the meta model [Trompeter and Beltran, 2007].
The most important reasons for using MDSD are that the quality of the software is improved, the reusability of the software becomes better and the development efficiency increases, whereas it has to be said that for a better reusablity the design often needs to be adjusted. Another reason for using MDSD, which could also be considered as being an advantage of it, is that with MDSD there is a higher level of abstraction. For that reason it is also read and understandable for persons who not familiar with developing software, e.g. customers or experts of a given domain. Before employing MDSD there are some prerequisites that have to be met. These prerequisites are the support of the management, the organisational general conditions, existing procedure models, required tools and know-how of the employees [Trompeter and Beltran, 2007].

2.3 Domain Specific Languages

"*A domain-specific language (DSL) is a small, usually declarative, language that offers expressive power focused on a particular problem domain*" [Van Deursen et al., 2000]. The domain of the present thesis is the entity relationship diagram that was explained in Section 2.1. This means that a DSL gets developed to describe this domain. The difference between a DSL and a general purpose language (GPL), e.g. C++ or Java, is that a DSL is developed just for a specific domain or problem. Which is an advantage of DSLs compared to GPLs. Another advantage of DSLs is that the syntax is more readable and easier to understand than the syntax of GPLs. Additionally DSLs are more time-saving when it comes to programming [Raja and Lakshmanan, 2010]. But there are also some disadvantages of DSLs, Van Deursen [Van Deursen et al., 2002] observes the following ones: First of all, the use of DSL is high priced, considering the necessary user education and the process of implementation as well as the design. In addition DSLs are not always available and finding a proper area of application can also lead to difficulties.
Before developing a DSL, one must know that there are two different types of it. Internal and external DSL [Raja and Lakshmanan, 2010]. Internal DSLs are inside a programming language, e.g. Ruby or Scala. These programming languages are often called host languages and have the benefit, that the developer does not need to be familiar with the grammar or the language parsing [Fowler, 2010]. Though, the biggest plus of internal DSLs is, that they can be developed inside a project that is implemented with the host language. Additionally the compiler of the host language can also be used for the DSL. External DSLs are different from this, because they do not have a host language. Everything has to be developed from scratch: they need parsing techniques, compilation, lexical analysis, grammars and code generation [Ghosh, 2011]. This also impli-

cates that there is a higher flexibility when designing and developing a DSL. A framework that provides everything, that is needed for developing an external DSL, is Xtext[3]. A well-known example for an external DSL is SQL, which is also being used in Section 4 for generating SQL code from the created diagram in the editor. But there are also two different types of external DSLs, the textual and the graphical DSLs. Textual DSLs are described in a text file while graphical DSLs get modelled in an Editor. Poseidon for DSL, being an editor for this purpose, also uses Xtext. For this reason Xtext is also needed in the present thesis, but only as a baseline as the DSL gets developed with Poseidon.

2.4 The Eclipse Modeling Framework

Eclipse is an open source Integrated Development Environment (IDE) which was primarily created for developing Java projects. Today it is possible to work with many other programming languages in Eclipse as well by using Plug-ins. Eclipse is based on the OSGi[4] framework 'Equinox' which was specifically developed for it. The Eclipse Modeling Framework (EMF) is the core of the Eclipse Modeling Project, which is one of the four main subprojects of Eclipse. The others are the above mentioned 'Equinox', the Plug-in Development Environment, which provides tools for eclipse plug-in development, and the Java Development Tools that provides all needed tools for Java development [Steinberg, 2009]. The EMF is a framework which provide everything what is necessary to model and generate code of Eclipse applications. These Eclipse applications have to be based on structured data models [Biermann et al., 2006]. A data model is a representation of a data, for example a bachelor thesis. This thesis comes with certain attributes, like the name of the author and the title. EMF provides the function for defining a data model either in Java, Extensible Markup Language (XML) or Unified Modeling Language (UML). Consequently no matter in which language the model is defined it always turns out to be the same, as EMF unifies it, like it is shown in figure 2.5. It is possible to let EMF create an XML scheme from UML or Java and vice versa. Additionally EMF is able to create the implementation code out of the EMF model. The meta model

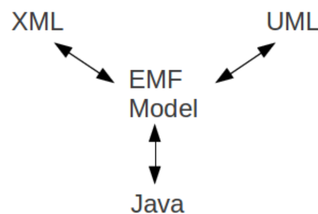

Figure 2.5: EMF unifies XML, Java and UML; Source: [Steinberg, 2009]

which describes an EMF model is called 'Ecore model'. Ecore provides differ-

[3] http://www.eclipse.org/Xtext/
[4] Short for Open Services Gateway initiative framework.

ent types of elements to create this model. The elements which are the main aspect of Ecore and that are also being used in this thesis are the 'EClass', the 'EDatatype', the 'EEnum', the 'EAttribute' and the 'EReference'. An EClass represents a class that can have attributes, which are represented with the EAttributes. These EAttributes can have a name and an EDatatype, which can be an integer or a string for example. An EReference is an association between two EClasses and consequently is attached to one of these. The EEnum is an enumeration type which is used to store the different opportunities of the cardinality in the entity relationship editor. Another element, that is getting used during the development of the editor, is the EEnumLiteral which is needed to store a value in the EEnum.
The hierarchy of these Ecore elements is shown in Figure 2.6, which also visualises that on top of the hierarchy there is the EObject. This EObject is the so called root element of all Ecore elements, which means that every generated interface extends this element. An advantage of this procedure is that EObject also extends the Notifier interface, making every object to a notifier as well [Steinberg, 2009]. Therefore the Observer design pattern[5] is used here, because it notifies objects when depending objects were changed. EMF stores

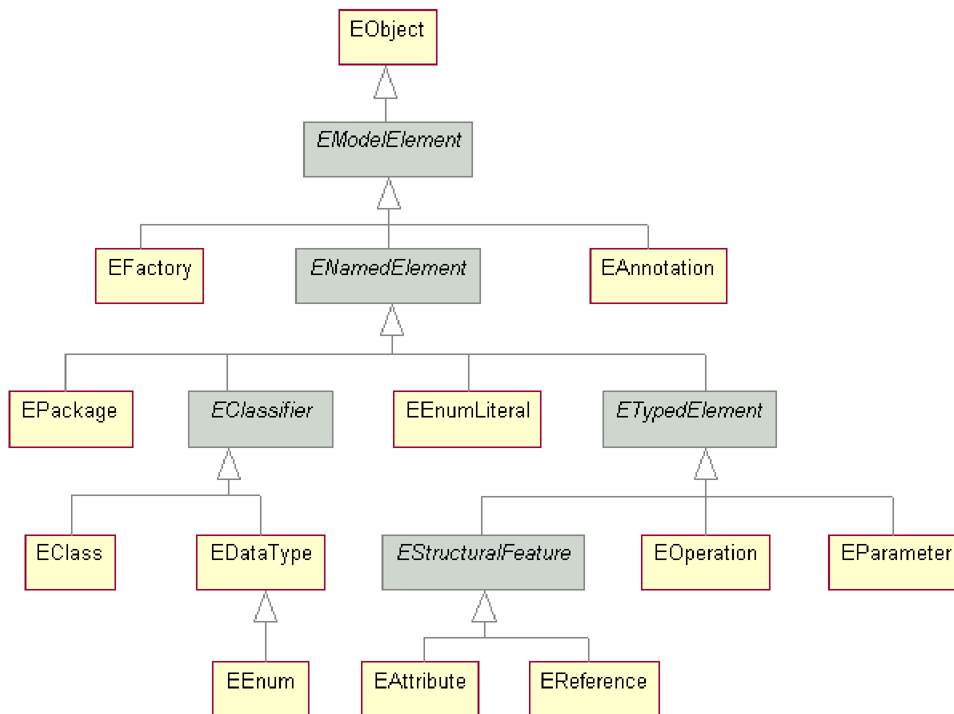

Figure 2.6: Hierarchy of the Ecore Components;
Source: [IBM Corporation and others, 2011]

an Ecore model in an XML Metadata Interchange (XMI) file, which is defined as a standard of the Object Management Group (OMG) for meta data. XMI is

[5]The Observer design pattern is one of the Behavioural patterns [Gamma et al., 2010]

based on XML and is often used to exchange or transform UML models [Kovse and Härder, 2002]. The EMF also provides an editor which allows to view and edit Ecore files, like it is shown in figure 2.7.

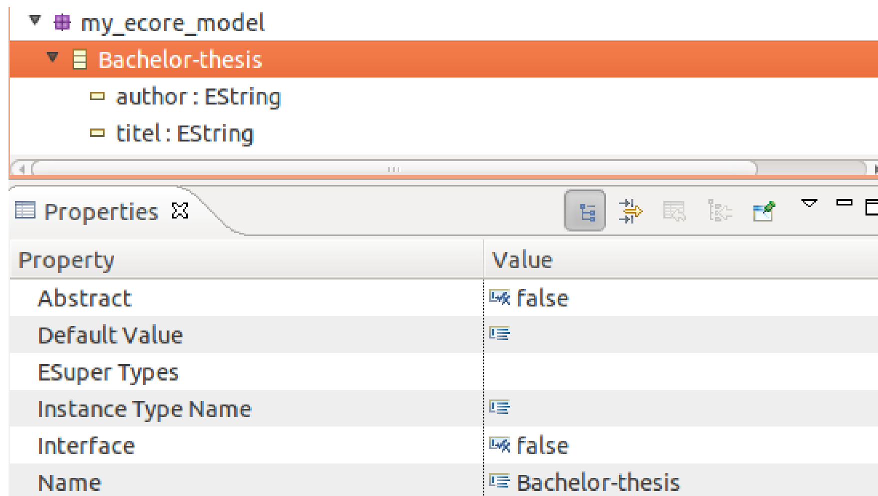

Figure 2.7: The EMF editor for Ecore files

Chapter 3

Generating an ER-Editor using DSLs

In the following paragraphs the creation of an Entity Relationship Editor will be shown. For creating and adjusting this editor, the tool Poseidon for DSL is required, as well as the IDE Eclipse[1] with the Eclipse Modelling Framework (EMF). All steps which are necessary to build this editor will be illustrated in detail. First the environment will be setted up, then the meta model will be defined and after this the model will be specified by using DSLs. According to this, the code gets generated and adjusted so that it fits the needs of an entity relationship diagram.

3.1 Setting up the environment

For setting up the environment a prepared workbench of Eclipse is necessary, which can be downloaded on the Gentleware homepage[2]. Furthermore also a template workspace is required, which is also available at the homepage of Gentleware.
The first step, for setting up the environment, is to import the prepared template workspace into the Eclipse IDE. In order to do this Eclipse needs to be started and the Menu item *'File'* - *'Import..'*[3] needs to be chosen. A new window opens where under *'General'* the entry *'Existing Project into Workspace'* has to be marked and the button *'Next'* needs to be pushed. After doing this, it is necessary to choose the workspace archive file and press the finish button. Now the whole workspace, which is required for the creation of an Entity Relationship Editor, is imported. There are a few errors which appear as red crosses, but they vanish later after building the editor for the first time.
The second step is to import the run-configurations. For doing this, again the menu item *'File'* - *'Import..'* needs to be chosen. Now marking the entry

[1] http://www.eclipse.org
[2] http://www.gentleware.com/downloadcenter.html
[3] Menus, menu items and buttons are shown in quotes and in italic.

'Launch Configurations' before pressing the 'Next' button. After this it is necessary to choose the Poseidon workspace, for example */home/user/workspace/-Poseidon*[4]. Additionally the Poseidon entry needs to be marked and the dialogue needs to be confirmed by pressing *'Finish'*.
The next step is to include the Ant[5] scripts which are required for building the workspace. For doing this you have to push ,at the Ant view in Eclipse, the button *'Add Buildfiles'*. The two files *'generate poseidon code from models.xml'* and *'make poseidon distribution.xml'* need to be selected and confirmed by pressing the *'Ok'* button.
Finally you need to proof that the correct Java SDK[6] was chosen. If this is not the case you need to set up the Java Development Kit (JDK). For Poseidon a JDK of 1.5 or higher is required. On Linux based systems the JDK can be found at *'/usr/lib/jvm/java-version[7]'*. To check out if the right JDK is activated the menu at *'Window - Preferences'* should be opened. Then the submenu *'Installed JREs'* under *'Java'* needs to be marked. If the wrong JDK version is selected the right version should be added by pressing the *'Add..'* button.[8] A new window opens where you choose *'Standard JVM'* and browse the path where the JDK is located. After that you can confirm by pressing the *'Finish'* button [Gentleware AG, 2011].
If all these steps are executed correctly the editor can be created following the steps which are shown in the next sections.

3.2 Creating the meta model for the editor

For the Entity Relationship Editor (ER-Editor) first a meta model needs to be created. This is also the first step of applying the idea of MDSD, as it was mentioned in Section 2.2. The meta model is created in the Ecore meta modelling language, which is included in the Eclipse Modelling Framework (EMF). Both of them have already been described in Section 2.4. To generate the model the editor 'Poseidon for ecore' is used, which is a graphical editor for Ecore meta models. The editor is also available in the template workspace of Poseidon. It can be started by pressing the 'run' button in Eclipse and choosing 'Poseidon for ecore'.

First you need to create two EEnums, which are called `CardinalityEnum`[9] and `DatatypeEnum`. Like the names of the two enums[10] announce they are being

[4] Packages, paths, folders and files are shown in italic.
[5] A Java library and command-line tool for building Java applications
[6] Short for Software Development Kit.
[7] java-version stands for the right Java version name, e.g. java-6-sun-1.6.0.24
[8] On the most Linux distributions the JDK is in the same path like the JRE and usually nothing needs to be changed. Unlike this on Windows systems the JDK is in an extra folder, for example in *C:\Program Files\Java\jdk*.
[9] Variable names, class names and function calls are shown in typewriter font.
[10] Short for enumeration.

used for the cardinality and the data types. The CardinalityEnum is filled with four EEnumLiterals, of which each represents a cardinality state. The possible states have been described in Section 2.1. As illustrated in Figure 3.1 every literal receives a different value. This is necessary in order to be able to identify a state, when the cardinality is implemented. In this case the field literal just serves the purpose of a better overview and will not be used in the model.

The other EEnum is also receiving a few EEnumLiterals, which are used for the data types of the attribute, so that the user can choose between these types for the created attribute in the diagram. They are restricted to 'char', 'varchar', 'float', 'date' and 'integer'. For an entity relationship diagram these types are unimportant though. They will be used later in Section 4 for the code generation.

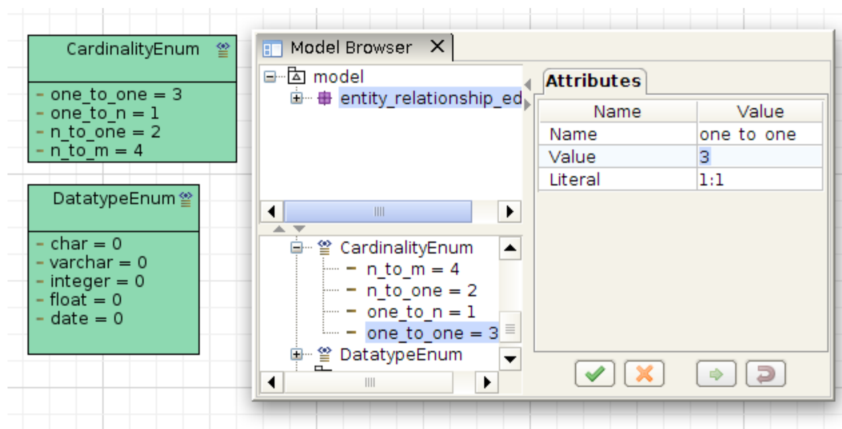

Figure 3.1: The two EEnums of the meta model

The next step of the creation of the meta model for the editor is to add an EClass which is called `ERNode`. This ERNode represents an abstract interface and all other EClasses inherit from it, except the EClass Line. To make the ERNode an abstract interface it is necessary to set the values of '*Abstract*' and '*Interface*' in the Model Browser to true. The Model Browser can be found in '*Poseidon for ecore*' beside the Palette context. After this two more EClasses need to be created. They are called `Entity` and `Relationship` and, like it was mentioned before, they have to inherit from the ERNode hereby. Entity represents the entities and Relationship the relationships in the ER-Editor. The EClass Relationship needs two EAttributes, which can be created by right clicking on the Relationship and choosing *'add EAttribute'*. The first EAttribute is called `connections`, which is an integer that stores the amount of connections that are belonging to the Relationship. This is necessary for the implementation of the cardinality and will be explained later. The next EAttribute is also required for the cardinality and is therefore called `cardinality`. It is from the type CardinalityEnum which was created earlier.

The next EClass that needs to be created is called `Line`. It is the only EClass that does not inherit from the ERNode. The Line represents an edge in the meta model, which pictures, as indicated by its name, the line between all nodes on the diagram. It has an EAttribute of the type EBoolean that is called `isFirst`. This EAttribute is used to specify if this is the first edge on a relationship or not. The second EAttribute is of the type EInt and also stores the cardinality. Line has two EReferences on the ERNode, that receive the names `start` and `end`. Later they will be used as a storage for the references of the nodes. The next node, which is necessary to make a valid entity relationship model, is called `Attribute`. Attribute is also represented as an EClass and inherits from ERNode too. In this class the above mentioned DatatypeEnum has to be added as an EAttribute, like it was done in the relationship EClass. The EAttribute receives the name `type`. The EClass Attribute has three incoming EReferences, all named `containsAttribute`. First the node Entity has a EReference on Attribute, because an entity can contain an attribute. The same applies to the node Relationship. The other EReference, which is as well a outgoing reference, points from Attribute to itself and is called `containsAttribute` too. The reason for this is, that an attribute can contain an attribute as well. It is important to set the upper bound of all containsAttribute EReference to '-1'. This means that they all work as a container that can hold an infinite amount of attributes. These EReference will later be used for the code generation in Section 4.2.
For an entity relationship model which is used for designing a database two more nodes are necessary. As described in Section 3.2 there are two more kinds of attributes: The key attributes and the multi valued attributes. These elements are necessary because later the editor will create SQL syntax, that can be used to create all needed tables in a database. Therefore every table needs a key attribute to identify its entries. For this reason two more EClasses have to be created in the meta model, which are called `KeyAttribute` and `MultiValuedAttribute`. Both also inherit from the ERNode and have an incoming EReference from the Entity node. The EReference to the key attribute is called `containsKey` and the other one is called `containsMVA`. As well as the node Attribute has a EReference on itself, it has an incoming EReference from the MultiValuedAttribute EClass, because a multi valued attribute can contain an attribute. The EReference is also called `containsAttribute`. Again the upper bound from both have to be set to a value of '-1'.
If everything was created in the right way the model should look like as it is shown in Figure 3.2. Is this the case the Poseidon Ecore meta model can be created by selecting *'Generate Poseidon Ecore Metamodel'* in the *'Poseidon Metamodel'* menu entry. After creating it, the generated ecore file can be found in the project *generators* in the folder *models*. In the same project there is a file which is called *user_emf.properties*. Inside this file the *metamodel_emf_package_class* needs to be set to the value
`com.gentleware.poseidon.dsl.entity_relationship_editor.USERPackage`.
At this point everything is prepared for generating the Poseidon model code. To do this the file *User-Model.genmodel* has to be opened and you need to right click on *'User-Model'*. Now the entry *'Generate Model Code'* needs to be cho-

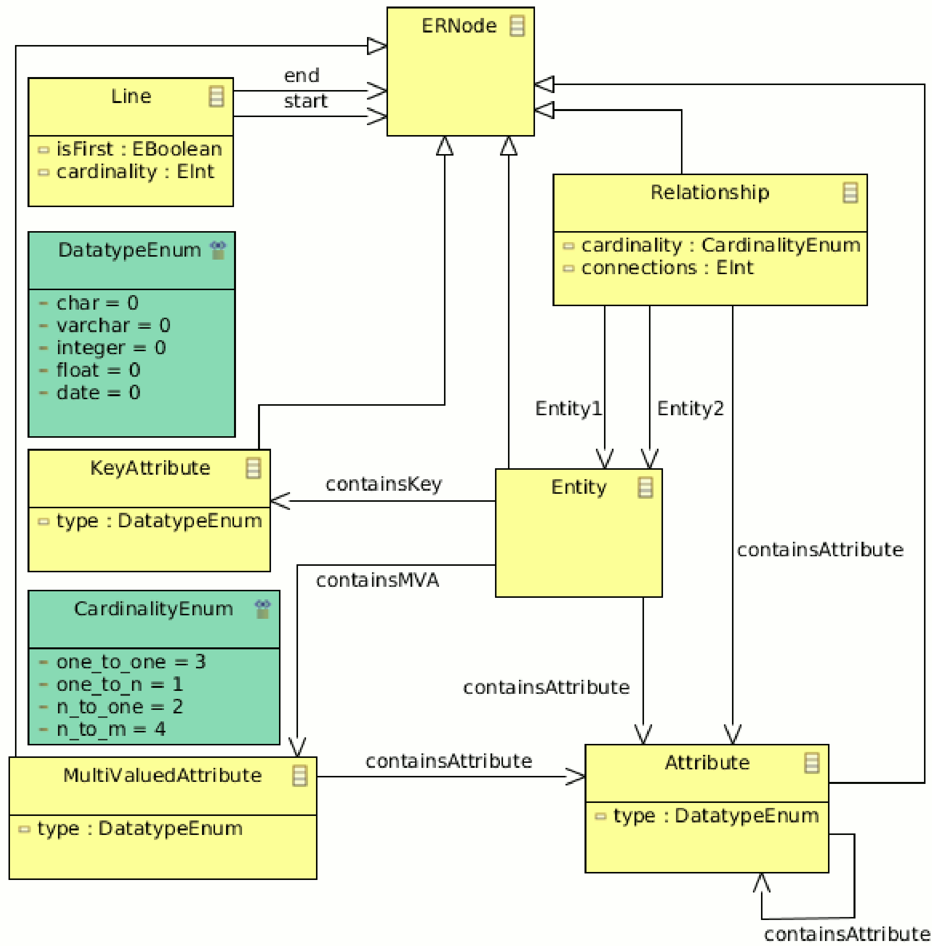

Figure 3.2: Meta model of the Entity Relationship Editor

sen. When there are no errors occurred it is now time to generate the Poseidon code from the model. This works by clicking on the Ant file *generate poseidon code from models*. If everything worked fine the new editor can be started by pressing the run button and choosing Poseidon. At the moment there are not much possibilities in the editor, because the model is still empty. This will be defined in the following sections. The editor should now look like it can be seen in Figure 3.3.

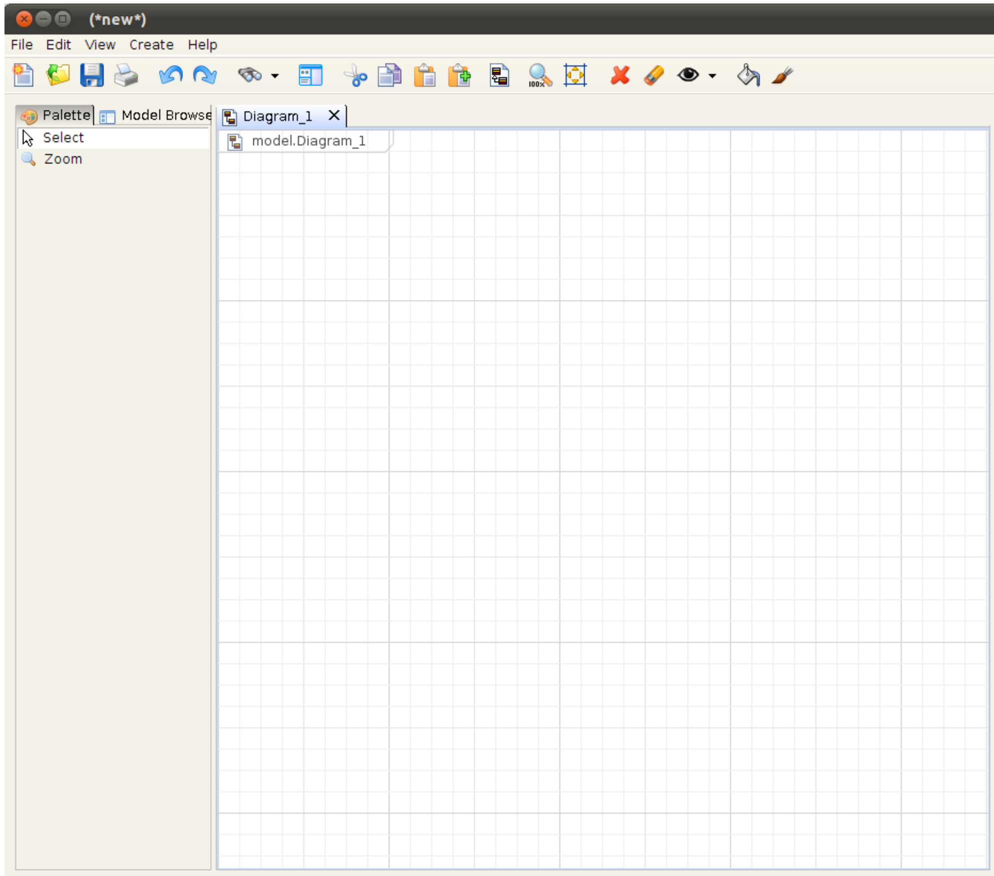

Figure 3.3: Empty Poseidon for DSL

3.3 Creation of all needed Elements in the Model

When the model code was generated and the editor was built only the standard editor is shown without having elements to choose, like it can be seen in Figure 3.3. The reason for this is, that the model code which was generated is just the code for an empty Poseidon for DSL editor. The important parts of the model for the ER-Editor are still empty. To get access to the elements that are used for an entity relationship model, they have to be created and specified in various files. All files which are important for the creation can be found in the workspace in the folders *Poseidon - Models*. There are the files *EcoreMetamodelMapping.map, EdgeRules.erule, PoseidonDiagramModel.dgm, PoseidonTools.tools, Properties.prt and RButtons.rbtn*. Here also the Domain Specific Languages (DSL) come in, because all of these files are including a DSL. All of them are needed to be filled with a textual DSL that describes a part of the model. The syntax and functions of these DSLs are defined by Poseidon. The noun DSL is used in the majority, because they are all equal in some aspects but not in total, thus it can be said that these are different DSLs. This also can be seen in the file ending of the different files.

3.3.1 Creation of the needed Nodes and Edges

All elements that are necessary for the editor have to be defined in the file *PoseidonDiagramModel.dgm*. First all nodes need to be added. They are representing the EClasses Entity, Relationship, Attribute, KeyAttribute and MultivaluedAttribute. In every node the attributes for the nodes have to be defined. The first one is `metamodel_element` which specifies the semantic element for the individual node. This should be the created EClasses, that are defined in Section 3.2. It is important to define the `metamodel_element`, because without this attribute the code is not being created. The other attributes are just for the appearance: They define the shapes, sizes, backgroundcolor and the position of the names of the nodes. The icon attribute is the picture that is shown in the Palette context of the editor. There are much more attributes to define, but for the Entity Relationship Editor the above mentioned are enough. The code for the nodes should look like it is shown in Listing 3.1.

```
1  node Entity {
2    metamodel_element: Entity
3    shape: RECTANGLE
4    icon: "object-node"
5    name_position: INSIDE
6    property FillColor : COLOR_RGB(210, 255 , 180)
7  }
8  node Relationship {
9    metamodel_element: Relationship
10   shape: RHOMBUS
11   default_size: 5*1
12   name_position: INSIDE
13 }
14 node Attribute {
15   metamodel_element: Attribute
16   shape: ELLIPSE
17   name_position: INSIDE
18   property FillColor : COLOR_RGB(240, 255 , 150)
19 }
20 node KeyAttribute {
21   metamodel_element: KeyAttribute
22   shape: ELLIPSE
23   name_position: INSIDE
24   property FillColor : COLOR_RGB(240, 255 , 150)
25 }
26 node MultiValuedAttribute {
27   metamodel_element: MultiValuedAttribute
28   shape: ELLIPSE
29   name_position: INSIDE
30   default_name: "MVAttribute"
31   property FillColor : COLOR_RGB(240, 255 , 150)
32 }
```

Listing 3.1: The defined nodes in the model

The next elements which have to be defined in this file are the edges. The edges are specifying how to connect two nodes with each other. In this case every edge represents a line from one node to an other. There are five possible lines in the Entity Relationship Editor: first there is the line from an entity

to a relationship. The cardinality which is pictured at the side of the entity will be ignored at this point, but it will be implemented later in Section 3.4. In every edge one or more sources, indicating where the line starts, and one or more targets, indicating where the line ends, need to be defined. Now the in Section 3.2 created EReferences `end` and `start` are being used. The edges also need the attribute `metamodel_element` which is, in this case, the EClass Line from the meta model. Unlike the nodes the `metamodel_element` is the same for every edge. The key word icon is used to define an icon for the element. In Listing 3.2 there are three icons used which are not in the standard icon set that Poseidon provides: these icons are keyAttr, multivaluedAttr and relationship. The problem here is, that Poseidon does not provide any icons that look like a key attribute, multi valued attribute or a relationship. Hence, they have to be created and stored in the project *resources* in the folder *icons*, they should look like it is shown in Figure 3.4 and can be created with every graphic manipulation software, for example Microsoft Paint.

Figure 3.4: The icons for multi valued attributes, key attributes and relationships

After storing them into the folder it is possible to use them in Poseidon. The whole code for the edges can be seen in Listing 3.2.

```
1  edge LineFromEntityToRelationship {
2    sources: Entity Relationship
3    targets: Relationship Entity
4    metamodel_element: Line
5    store_source_in Line__start
6    store_target_in Line__end
7    icon: "relationship"
8    default_name: "n"
9  }
10 edge LineFromERAToAttribute {
11   sources: Entity Relationship Attribute MultiValuedAttribute
12   targets: Attribute
13   metamodel_element: Line
14   store_source_in Line__start
15   store_target_in Line__end
16   icon: "usecase"
17 }
18 edge LineFromEntityToKeyAttribute {
19   sources: Entity
20   targets: KeyAttribute
21   metamodel_element: Line
22   store_source_in Line__start
23   store_target_in Line__end
24   icon: "keyAttr"
25 }
26 edge LineFromEntityToMultiValuedAttribute {
```

```
27    sources: Entity
28    targets: MultiValuedAttribute
29    metamodel_element: Line
30    store_source_in Line__start
31    store_target_in Line__end
32    icon: "multivaluedAttr"
33  }
```

Listing 3.2: The defined edges in the model.

3.3.2 Adding the Elements as Tools

After defining all nodes and edges all necessary elements for the Entity Relationship Editor are available. In order to get access to them in the editor, the tools, which are shown in the Palette context, have to be defined too. All tools that should appear in this context need to be defined in the file *PoseidonTools.tools* Therefore, the first thing to do is to specify a category. This category represents the sub menu in the Palette context. Inside this category there are the tools which can create the elements. The first entry in the *PoseidonTools.tools* file is a `node_tool` which defines a tool on the category. This is also the only `node_tool` entry because all other entries are from the type `edge_tool`. The reason for this is, that the attributes and relationships are always connected to an entity. Therefore no attribute or relationship can stand alone in the diagram - it is always connected to another node. Inside the `node_tool` there has to be specified which node should be created within the diagram: in this case it is the Entity. To get access to the edges an `edge_tool` has to be added for every edge. They are the same that were defined above in the *PoseidonDiagramModel.dgm* in Section 3.3.2. Moreover, they have quite the same syntax as the `node_tool`, just the `diagram_node` is a `diagram_edge` now. The implementation of it can be seen in Listing 3.3.

```
1   category EntityRelationshipDiagram {
2     node_tool Entity {
3       diagram_node: Entity
4     }
5     edge_tool Relationship {
6       diagram_edge: LineFromEntityToRelationship
7     }
8     edge_tool Attribute {
9       diagram_edge: LineFromERAToAttribute
10    }
11    edge_tool KeyAttribute {
12      diagram_edge: LineFromEntityToKeyAttribute
13    }
14    edge_tool MultiValuedAttribute {
15      diagram_edge: LineFromEntityToMultiValuedAttribute
16    }
17  }
```

Listing 3.3: The defined tools in the model.

The last step is to define one rule in the file *EdgeRules.erule*. This rule tells the editor that an entity or a relationship is created alternately. The code for this file can be seen in Listing 3.4

```
1 rules_for LineFromEntityToRelationship {
2   from Entity create Relationship
3   from Relationship create Entity
4 }
```

Listing 3.4: Edge rules in the file EdgeRules.erule

If all implementations were made correctly the model for the ER-Editor is finished. For remembrance: After having created the model, using the approach of MDSD, runnable code will be generated out of the model. This works by pressing the Ant file *generate poseidon code from models* again. Now all elements can be created by either using the Palette context. After finishing the before mentioned steps the editor should look like it is shown in Figure 3.5.

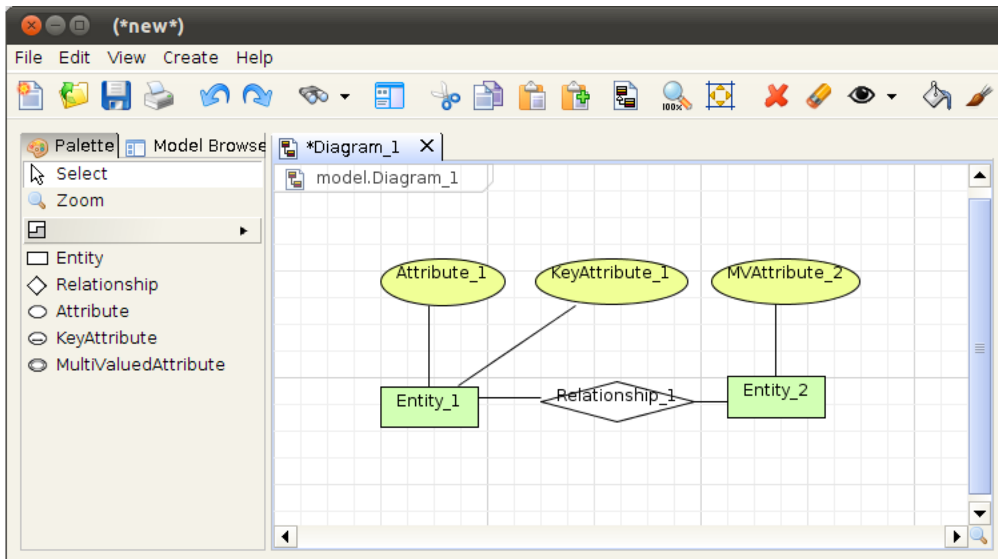

Figure 3.5: The ER-Editor after defining all models

3.3.3 Adding Rapid Buttons

The last file that needs to be edited is *RButtons.rbtn*. Poseidon provides a nice feature which improves the flow of creating a diagram. It is called 'Rapid Button'. Rapid Buttons appear on an element when the mouse focus lies on it. The Rapid Buttons recommend all elements, that can be added to the focused element. With one click on one Rapid Button the newly chosen element is created immediately. This is an awesome feature which is not that important for a model that only supports five different elements, but the implementation is facilitated by using it. For this purpose some additional definitions in the model are necessary: the Rapid Buttons are defined with the keyword button.

Furthermore, the attribute anchor specifies the elements that contain the rapid button. The attribute edge defines the target edge that is created after pressing the rapid button. With the value position the position of the rapid button can be specified. The icon keyword is used to define an icon for the button which is displayed. For the ER-Editor the icons are the same as the ones that are pictured in the Palette context. The last attribute which has to be defined is the `direction` attribute. It specifies if the edge is incoming or outgoing. This is just important regarding the cardinality, because it has to be specified if the text is shown at the end or at the beginning of the line. Consequently only the Rapid button for entity, which contains the relationship, is an incoming edge, all others have to be outgoing. The whole code for the Rapid Button implementation can be found in Listing 3.5 and an entity with its Rapid Buttons in Figure 3.6. If the changes are made correctly, the code needs to be generated again. This works like it was done before by pressing the *generate poseidon code from models* Ant file in Eclipse.

```
1  button Relationship {
2    anchors: Entity
3    edge: LineFromEntityToRelationship
4    position: LEFT RIGHT
5    icon: "relationship"
6    direction: OUTGOING
7  }
8  button AttributeForEntity {
9    anchors: Entity
10   edge: LineFromERAToAttribute
11   position: TOP_LEFT BOTTOM_RIGHT
12   icon: "usecase"
13   direction: OUTGOING
14 }
15 button KeyAttributeForEntity {
16   anchors: Entity
17   edge: LineFromEntityToKeyAttribute
18   position: TOP BOTTOM
19   icon: "keyAttr"
20   direction: OUTGOING
21 }
22 button MultiValuedAttributeForEntity {
23   anchors: Entity
24   edge: LineFromEntityToMultiValuedAttribute
25   position: TOP_RIGHT BOTTOM_LEFT
26   icon: "multivaluedAttr"
27   direction: OUTGOING
28 }
29 button Entity {
30   anchors: Relationship
31   edge: LineFromEntityToRelationship
32   position: RIGHT LEFT
33   icon: "object-node"
34   direction: INCOMING
35 }
36 button AttributeForRelationship {
37   anchors: Relationship
```

```
38     edge: LineFromERAToAttribute
39     position: TOP BOTTOM
40     icon: "usecase"
41     direction: OUTGOING
42 }
43 button AttributeForAttribute {
44     anchors: Attribute
45     edge: LineFromERAToAttribute
46     position: TOP_LEFT TOP_RIGHT BOTTOM_LEFT BOTTOM_RIGHT
47     icon: "usecase"
48     direction: OUTGOING
49 }
50 button AttributeForMultiValuedAttribute {
51     anchors: MultiValuedAttribute
52     edge: LineFromERAToAttribute
53     position: TOP_LEFT TOP_RIGHT BOTTOM_LEFT BOTTOM_RIGHT
54     icon: "usecase"
55     direction: OUTGOING
56 }
```

Listing 3.5: Definition of the Rapid Buttons

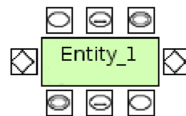

Figure 3.6: An entity with its Rapid Buttons

3.4 Customising the Look of all Elements

For now the skeleton of the Entity Relationship Editor is finished, but there are some elements that are not looking like they have to yet. The first two elements that have to be changed are the key attributes and multi valued attributes. At the moment they are both looking like normal attributes, but as they are special attributes they have to be pictured differently. For keeping in mind the key attribute has its name underlined and the multi valued attribute is pictured as a double ellipse. At this point the adjustment of the generated code, that was mentioned in Section 2.2, comes in. To implement this some Java code needs to be written. Poseidon provides the ability to customise every element via overwriting the corresponding method. For the ER-Editor this has to be done for all nodes except the one for the entity. The entity node is pictured as a rectangle with its name on the inside. All attributes which were defined in the *PoseidonDiagramModel.dgm* file are enough for this case and a adjustment of the code is not necessary. In this section the often mentioned cardinality is also implemented, which can be realised by overwriting a method for the corresponding edge. Everything that is changed in this section, could not have been defined in the previously defined model of the ER-Editor, be-

cause the possibilities therefore are lacking. This means that the created and defined DSLs are not offering definitions to fit the wanted needs.
The first node which is customised now is the attribute node. The appearance of the attribute is almost fine, the only thing that does not fit is the position of the name. Poseidon just provides the following cases for the position of the node's name: either the name is hidden, outside or inside of a node. But it is not possible to define the accurate position of the name. In the model it was defined that the name is pictured inside, though actually it is pictured on the top end inside the element and not in the middle of it, like it usually had to be in an entity relationship model. For this reason the file *AttributeNodeGem.java* has to be edited. This file can be found in the package *com.gentleware.poseidon.custom.diagrams.node.impl.* To change the appearance of the attribute the method `createBasicNodeAppearanceFacet` needs to be overwritten in the class `AttributeNodeGem`. This method normally returns an `EllipseShapeAppearanceFacet` which is the default Poseidon implementation for an ellipse. To overwrite this a new `BasicSeparateNameNodeAppearanceFacet` has to be returned, in which the overwriting of the method `formView` is necessary. This method is responsible for the appearance of the element. For default the `BasicSeparateNameNodeAppearanceFacet` class creates a rectangle with its name beneath it. Therefore these two default elements have to be removed from the container which is created with the constructor of the class. After this the new ellipse has to be added. To get the right position for the name the middle of the bounds needs to be calculated. For the text a ZLabel is used, which is also shown in Listing 3.6. ZLabel is an element of Jazz[11], which is used by Poseidon to render the objects.

```
1  @Override
2  public NodeAppearanceFacet createBasicNodeAppearanceFacet() {
3    return new BasicSeparateNameNodeAppearanceFacet(initialFillColor,
4    initialFillColor, figureFacet, contents, figureName,
         textableFacet,
5    subject, false) {
6      @Override
7      public ZNode formView() {
8        ZGroup zGroup = (ZGroup) super.formView();
9        ClassifierSizes sizes = (ClassifierSizes) makeCurrentInfo().
             makeSizes();
10       Rectangle bounds = sizes.getOuter().getBounds();
11       double height = bounds.height;
12       double width = bounds.width;
13       double x = bounds.x;
14       double y = bounds.y;
15       ZShape ellipse = new ZEllipse(x, y, width, height);
16       ellipse.setFillColor(new Color(240, 255 , 150));
17       zGroup.removeChild(1);
18       zGroup.removeChild(0);
19       ZLabel zLabel = new ZLabel(this.getSubjectName());
20       zLabel.setTranslation(x + (width/2) - zLabel.getText().length
             ()*3, y + (height/2) - 7);
```

[11] A Toolkit for Object-Oriented 2D Graphics in Java

```
21          zGroup.addChild(new ZVisualLeaf(ellipse));
22          zGroup.addChild(new ZVisualLeaf(zLabel));
23
24          return zGroup;
25      }
26    };
27  }
```

Listing 3.6: The code for the adjustment of the attribute.

The same works for the KeyAttribute, which is still pictured as a normal attribute, but its name has to be underlined in order to distinguish it from other attributes (See Section 2.1). In the code everything is similar to the implementation of the attribute, except the font of the label. The file *KeyAttributeNodeGem.java* has to be customised like the *AttributeNodeGem.java* file. The ZLabel uses the Java AWT[12] font class to define the font. For this reason a new font needs to be created, because the font class does not support underlined text. After creating the font, the label has to be set to this font, like it is shown in Listing 3.7. This has to be done before adding the label to the container of the zGroup.

```
1  Map<TextAttribute, Integer> fontAttributes = new HashMap<
       TextAttribute, Integer>();
2  fontAttributes.put(TextAttribute.UNDERLINE, TextAttribute.
       UNDERLINE_ON);
3  Font underline = new Font(this.getFont().getAttributes()).
       deriveFont(fontAttributes);
4  zLabel.setFont(underline);
```

Listing 3.7: The code for the underlined text inside the key attribute.
Source: [Künneth, 2010]

Like the KeyAttribute the MultiValuedAttribute differs only in one aspect from the standard attribute. To do this the file *MultiValuedAttributeNodeGem.java* has to be also extended like the *AttributeNodeGem.java* file. The difference lies in the double ellipse, so nearly the whole code looks like the one of the Attribute. Hence, the only thing that needs to be added is a second ellipse. This ellipse is smaller and the coordinates, where it is painted, are different from the other ellipse. For a little optical difference between the two ellipses, the bigger one gets a darker yellow as its background. Now the new ellipse has to be created and the background color gets changed. Of course, the second ellipse also needs to be added to the zGroup, which is returned by this method. This process can be seen in Listing 3.8.

```
1  ZShape ellipse2 = new ZEllipse(x+5, y+5, width-10, height-10);
2  ellipse2.setFillColor(new Color(240, 255 , 150));
3
4  zGroup.addChild(new ZVisualLeaf(ellipse2));
```

Listing 3.8: The code for the double ellipse of the multi valued attributes.

[12]Short for Abstract Window Toolkit.

After adding these lines of code, all kinds of attributes look like they should. Now the next element that has to be adjusted is the relationship. In this case there is the same problem like it was with the attribute: the name is not pictured directly in the middle of the element. In order to center the name in the middle of the relationship the file *RelationshipNodeGem.java* needs to be edited. This means that the same method, like it was done for the attributes before, needs to be overwritten. Regarding to the relationship this turns out to be more complex, because Jazz does not provide a rhombus by itself. To paint a rhombus it is necessary to picture four single lines which together make up a rhombus. For doing this the points where the lines start and end have to be calculated. Except of these four lines the code is the same like it is for the attribute and can be seen in Listing 3.9.

```
1  UPoint middleTop = new UPoint(x + width/2, y);
2  UPoint middleRight = new UPoint(x + width, y + height/2 );
3  ZLine zLineTopRight = new ZLine(middleTop, middleRight);
4
5  UPoint middleLeft = new UPoint(x, y + height/2);
6  UPoint middleBotton = new UPoint(x + width/2, y + height);
7  ZLine zLineButtonLeft = new ZLine(middleLeft, middleBotton);
8
9  ZLine zLineTopLeft = new ZLine(middleTop, middleLeft);
10 ZLine zLineButtonRight = new ZLine(middleBotton, middleRight);
```

Listing 3.9: The code for painting the rhombus of a relationship.

Now all nodes are looking like they should. The next step is to adjust the line from an entity to a relationship. In this step the cardinality is being implemented. This is also quite similar to the adjustment of the nodes: The method `createBasicEdgeAppearanceFacet` has to be overwritten, but the file which includes this method is not in the same package as the files for the nodes. It is called *LineFromEntityToRelationshipArcGem.java* and can be found in the *com.gentleware.poseidon.custom.diagrams.edge.impl* package. The lines from an entity to a relationship and vice versa use the same type of edge, so it is only necessary to change one single line. Like it was defined in Section 3.2 the cardinality is stored as an attribute of the relationship. This attribute is represented with an enum with the class relationship providing a function to get the corresponding object of this enum. In order to get access to the enum the relationship object, which holds the enum has to be fetched. To do this it first has to be checked if the relationship is at the start or the ending of the line. The start and the ending were also be defined in Section 3.2 and they represent the EReferences start and end from the EClass Line. With the methods `basicGetStart` or `basicGetEnd` the relationship is presented and with the above mentioned function the enum is accessible. After receiving the enum it is important to check if the line has to display the first or the last value of the cardinality. In the relationship it is stored how many incoming lines it has and in the line is stored if it concerns the first or the second line. These values can also be used for the purpose of checking where the cardinality has to be displayed. It is stipulated, that if a line is the first one on a relationship it displays the first value and if it is the second one it displays the second value. For remembrance:

in Section 3.2 the attribute isFirst of the EClass line was created. This is the variable that stores the value if a line is the first one that is connected to a relationship or not. The whole code is shown in Listing 3.10. In this listing there is a function call, which starts the function `getLabel`. This function is a help method, which returns the right value that is pictured beside the line. The method belongs also to the `LineFromEntityToRelationshipArcGem` class and can be seen in Listing 3.11.

```java
@Override
protected BasicArcAppearanceGem createBasicEdgeAppearanceFacet() {
  return new CustomBasicArcAppearanceFacet(fillColor, lineColor,
    figureFacet, figureName, subject) {
    @Override
    public ZNode formAppearance(ZShape mainArc, UPoint start,
      UPoint second, UPoint secondLast, UPoint last,
      CalculatedArcPoints calculated, boolean curved) {
      ZNode formAppearance = super.formAppearance(mainArc, start,
          second, secondLast, last, calculated, curved);
      EObject eObject = super.subject;
      WrappedLineImpl wraLiIm = (WrappedLineImpl) eObject;
      Relationship rel = null;
      if(wraLiIm.basicGetEnd() instanceof Relationship ){
        rel = (Relationship) wraLiIm.basicGetEnd();
      } else {
        rel = (Relationship) wraLiIm.basicGetStart();
      }
      CardinalityEnum carEnum = rel.getCardinality();
      if(wraLiIm.isIsFirst() == false && rel.getConnections() == 0)
          {
        wraLiIm.setIsFirst(true);
        rel.setConnections(rel.getConnections() + 1);
      } else if(wraLiIm.isIsFirst() == false && rel.getConnections
          () == 1) {
        rel.setConnections(rel.getConnections() + 1);
      }
      ZLabel text = new ZLabel(getLabel(carEnum.getValue(), wraLiIm
          .isIsFirst()));
      text.setTranslation(last.getX() > start.getX() ? start.getX()
          + 5 : start.getX() - 10,
        last.getY() > start.getY() ? start.getY() + 5 : start.getY
          () - 15);
      ((ZGroup) formAppearance).addChild(new ZVisualLeaf(text));

      return formAppearance;
    }
  };
}
```

Listing 3.10: The code for displaying the cardinality.

```java
private String getLabel(int val, Boolean isFirst){
  switch (val) {
  case 1:
    return isFirst == true ? "1" : "n";
  case 2:
    return isFirst == true ? "n" : "1";
  case 3:
    return isFirst == true ? "1" : "1";
  case 4:
    return isFirst == true ? "n" : "m";
  }
  return "1";
}
```

Listing 3.11: The help method getLabel.

3.5 Modifying the ER-Editor

The Entity Relationship Editor is finished now. All elements that are necessary can be created, but there are some defective appearances in the editor. To correct them some of the methods need to be overwritten. The first suboptimal feature that needs to be changed is the fact that the edges are shown in the Model Browser. This is not acceptable, because only entities, relationships and the different attributes are a part of an entity relationship model and consequently only they should be accessible in the Model Browser. The easiest way to implement this constraint is to hide all edges in the Model Browser. In order to do this the method `rejectElementFromTree` needs to be overwritten in the class `CustomTreeMediator`. This class can be found in the *com.gentleware.poseidon.custom.repobrowser* package in the model_browser project inside the workspace. The method `rejectElementFromTree` returns true if the EObject has to be hidden in the Model Browser. Therefore it only needs to be checked if the EObject is an instance of Line. If this is the case true can be returned and false otherwise. The code for this can be found in Listing 3.12.

```java
@Override
protected boolean rejectElementFromTree(EObject parent, EObject
    child) {
  boolean rejected = super.rejectElementFromTree(parent, child);
  if (!rejected && child instanceof Line) {
      rejected = true;
  }
  return rejected;
}
```

Listing 3.12: The method rejectElementFromTree in CustomTreeMediator.java

The next step to improve the editor is to avoid more than two entities on a relationship, because if more than two entities are allowed the cardinalities cannot be displayed. This would also lead to the occurrence of difficulties in generating the code out of the model. This constraint can be implemented in the

class `LineFromEntityToRelationshipCreateFacetImpl` which is stored in the same package like the class for the cardinality. In this class two methods need to be overwritten: the `checkStart` and `checkEnd` method. As the names of these two methods suggest they check the start or the end of an edge. They both return a boolean value which tells if the edge can be assembled or not. For the needed case it is only necessary to return false if there are more than two connections at the relationship. If the code was implemented like it is shown in Listing 3.13, it is now impossible to connect more than two entities at one relationship.

```
1  private boolean startIsEntity = false;
2  @Override
3  protected boolean checkStart(AnchorFacet start) {
4    boolean checkStart = super.checkStart(start);
5    if(start.getFigureFacet().getRoles().contains(DslGenElementRoles.
       RELATIONSHIP)){
6      Object subject = start.getFigureFacet().getSubject();
7      Relationship rel = (Relationship) subject;
8      if(rel.getConnections() >= 2){
9        return false;
10     }
11   }
12   if(start.getFigureFacet().getRoles().contains(DslGenElementRoles.
       ENTITY)){
13     startIsEntity = true  ;
14   }
15   return checkStart;
16 }
17 @Override
18 protected boolean checkEnd(AnchorFacet end) {
19   boolean checkEnd = super.checkEnd(end);
20   if(end.getFigureFacet().getRoles().contains(DslGenElementRoles.
       RELATIONSHIP)){
21     Object subject = end.getFigureFacet().getSubject();
22     Relationship rel = (Relationship) subject;
23     if(rel.getConnections() >= 2){
24       return false;
25     }
26   }
27     if(end.getFigureFacet().getRoles().contains(DslGenElementRoles.
         ENTITY) && startIsEntity == true){
28     startIsEntity = false;
29     return false;
30   }
31   return checkEnd;
32 }
```

Listing 3.13: The overwritten methods checkStart and checkEnd

But it is still possible to create more entities, connected to the relationship, by pressing the Rapid Buttons. To avoid this behaviour, the Rapid Buttons of the relationship need to be changed. This can be implemented in the same file like it was done for the appearance of the relationships, the file *RelationshipNodeGem.java*. The easiest way is to remove all Rapid Buttons of the

relationships if two entities are connected to it. To realise this idea a new class is created in this file which checks how many entities are connected to a relationship. If there are two Entities connected to a relationship the class just prevents the painting of the Rapid Buttons by overwriting the existing mouse listeners. A few more changes are necessary in order to get this functionality working, as it can be seen in Listing 3.14.

```
1  List<RapidButtonManipulator> rbm;
2  Object subject2;
3  @Override
4  public NodeAppearanceFacet createBasicNodeAppearanceFacet() {
5    final BasicSeparateNameNodeAppearanceFacet bsnnaf = new
         BasicSeparateNameNodeAppearanceFacet(initialFillColor,
6    initialFillColor, figureFacet, contents, figureName,
         textableFacet,
7    subject, false) {
8    @Override
9    protected void createRButtonManipulators(UPoint location,
10     UBounds bounds,
11     ToolCoordinatorFacet coordinatorFacet) {
12     super.createRButtonManipulators(location, bounds,
           coordinatorFacet);
13     subject2 = getSubject();
14     rbm = rButtonManipulators;
15   }
16   @Override
17   public ZNode formView() {
18     // The code from Listing 3.9
19   };
20   RemoveRB bb = new RemoveRB(fillColor, initialFillColor,
         figureFacet, contents, figureName, textableFacet, subject,
         false);
21   bsnnaf.setRapidButtonMouseListener(bb.rapidButtonMouseAdapter2);
22   return bsnnaf;
23 }
24 protected class RemoveRB extends
       SeparateNameUnderNodeAppearanceFacet {
25   public RemoveRB(Color fillColor, Color initialFillColor,
         BasicNodeFigureFacet figureFacet, SimpleContainerFacet
         contents, String figureName, TextableFacet textableFacet,
         EObject subject, boolean proportionalResizing) {
26     super(fillColor, initialFillColor, figureFacet, contents,
           figureName, textableFacet, subject, proportionalResizing);
27   }
28   public ZMouseAdapter rapidButtonMouseAdapter2 = new ZMouseAdapter
         () {
29     public void mouseEntered(ZMouseEvent e) {
30       Relationship r = (Relationship) subject2;
31       for (RapidButtonManipulator manipulator : rbm) {
32         if(r.getConnections() != 2){
33           manipulator.mouseEntered();
34         }
35       }
36     }
37     public void mouseExited(ZMouseEvent e) {
38       Relationship r = (Relationship) subject2;
39       for (RapidButtonManipulator manipulator : rbm) {
```

```
40          if(r.getConnections() != 2){
41            manipulator.mouseExited();
42          }
43        }
44      }
45    };
46  };
```

Listing 3.14: Avoiding the Rapid Buttons

Now the editor is finished and everything works how it has to. The final editor and all of its elements should look like it is shown in Figure 3.7.

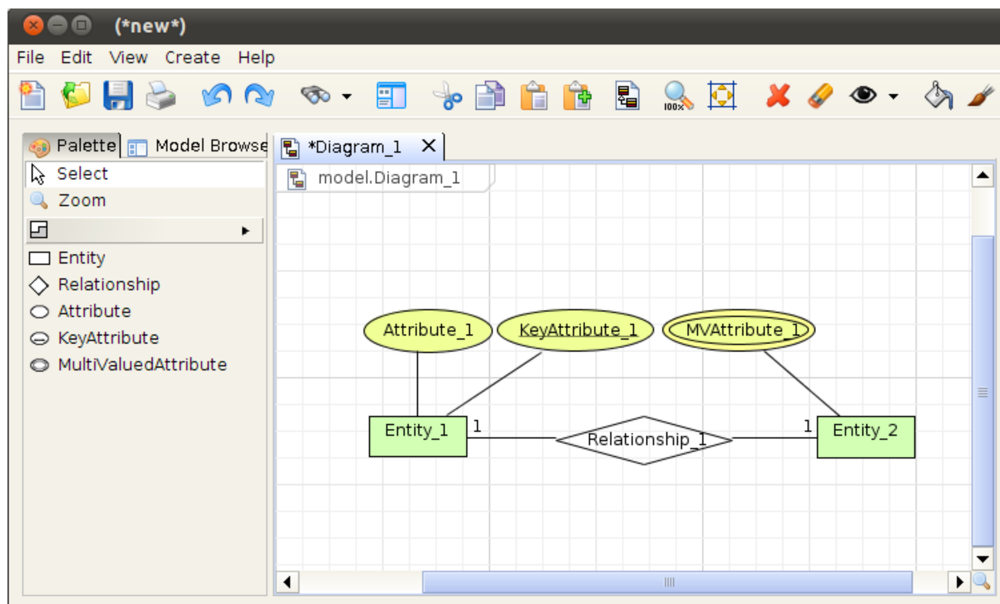

Figure 3.7: The ER-Editor when it is finished.

3.6 Limitations of the ER-Editor

There are a few limitations of the ER-Editor that need further discussion. This concerns some features that were planned in advance but turned out not to be implementable and some of them are constraints chosen by the author. First, an entity relationship diagram is not valid if a relationship stands alone or is just connected to one entity. In the Editor it is though possible to create this scenario. However, the relationship will be pictured red which should remind the user that the created diagram is not a valid one. This could have been avoided, if a relationship was to be painted between just two entities. This circumstances was not implemented because two steps had been combined into one: the step to create a relationship and the step to create an entity. Both steps include the creating of an edge and a node, which means that four objects are created and they all should be connected to each other. Although this is

theoretically possible by invoking a new generation process for an entity when a relationship is created, it was not implemented. The reason why it was not implemented is that the generation process for elements in Poseidon always uses the same generation routine. Consequently if an entity, a relationship or an attribute is generated, this routine is always executed. In order to avoid this procedure the routine should be overwritten or extended by checking if the elements that need to be created are from the type relationship. And it also has to be checked if an entity already exists at the end of it or if a new one should be created. An experimental implementation of these was aborted after the detection of the above mentioned constrained checking, due to the fact that there are two methods for checking if the start or the ending of an edge are correct (See Section 3.5). But the routine for creating an element neither checks the end nor the start of an edge: They are checked prior to the starting of the routine, because the routine is only executed if no violated constraints were detected. As the second edge is just created as a result of the first one, this could not be checked.

The next limitation is the behaviour of the Rapid Buttons. In the case of two entities being connected to a relationship the Rapid Buttons disappear, in order to avoid the possibility to connect more than two entities to the relationship. The problem here is that all Rapid Buttons are removed and not only the ones that are used for creating an entity: The Rapid Buttons for the creation of an attribute are also being removed. The reason for this is that it is not possible to only remove certain buttons, you have to remove all buttons at once.

In other entity relationship editors, for example JERM[13] only an attribute can be connected to an entity and only later, as a value of the attribute, it can be chosen if the attribute is a key or a multi valued attribute. The reason why in the present ER-Editor all types of attributes are already available on the Palette context is that the created edges are differing from attribute to attribute (See Section 2.1). For example as a starting point an attribute can have a usual attribute or a multi valued attribute, but not a key attribute. This would have had to be allowed if the solution, using only one attribute, had been chosen. Other consequences would have been that the constraint, which specifies that a key attribute is not allowed to contain an attribute, would have been violated.

Another limitation of the ER-Editor is that the relationship cannot be displayed with a background color. The reason for this is that the relationship is drawn with four separate lines and there is no area which can be filled with a colour.

[13] Just an Entity Relationship Modeler, available in version 2.5.1

Chapter 4

Generating SQL Statements from an ER-Model

In this chapter the MDSD approach is being used again. First the meta model that describes the behaviour of the model will be adjusted. After this a generator will be written which is able to create runnable code from a model. But this time the model has to be created by the user, because the model for this purpose is the entity relationship model. This is in the same time a graphical DSL, that is used to create the SQL statements. For generating code from the Entity Relationship Editor the generator framework Xpand[1] is used. The template of Poseidon provides a project, which is prepared for writing a generator. This project can be found in the folder *poseidon_generator* in the workspace. It includes a sample template that generates a file which shows all diagrams that are used in the created editor. It is stored in the package *templates* and is called *Main.xpt*, .xpt indicates a file used for Xpand templates. In the project there are also the Ecore files that are used in the Xpand template. The last important file that is stored in the project is named GenerateFromPoseidon.mwe. This file defines the workflow of the generating process and is also the entry point from the editor. This means that this file is activated by the ER-Editor for starting the SQL code generation.

4.1 Adjust the Ecore meta model

The sample template Main.xpt uses two meta models: The Ecore files *poseidon-core.ecore* and *user.ecore*. The Ecore file *poseidon-core.ecore* provides all elements that are used in the Poseidon EMF object. This object is also used in the internal process by saving a diagram. For example it provides the name, role and attributes of an object in the editor. After importing the file into the Xpand template it is possible to access all the entries in the template. For the generator additionally the Ecore file that represents the elements of the ER-Editor is needed. To get access to this file inside the Xpand file a few adjustments in the Ecore file are necessary. It is also the same file which was generated in

[1]The Xpand documentation can be found at:
http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xpand.doc/help/ch01s06.html

Section 3.2, by using Poseidon for Ecore. It can be found in the project *generators*. First the file has to be copied into the poseidon_generator project in the package *models*. For a better clarity it is recommended to rename the file into *entity-relationship-editor.ecore*. After the successful copying and renaming the file can be opened and the *poseidon-core.ecore* needs to be added as a resource. For doing this a right click on the opened *entity-relationship-editor.ecore* is necessary and the entry *'Load Resource..'* has to be chosen. Now by browsing the workspace the above mentioned *poseidon-core.ecore* needs to be picked. After doing this the resource will appear in the Ecore file and it is possible to access the entries of the *poseidon-core.ecore*. The next step is to add a reference in the *entity-relationship-editor.ecore* file. To execute this step a new child needs to be added by right-clicking on the entity-relationship-editor entry in the file. As the new child an EAnnotation needs to be chosen. In the newly created EAnnotation the reference has to be the PoseidonCoreRootElement, which works by clicking into the empty field of the reference and by opening the Reference Browser. In the Reference Browser the PoseidonCoreRootElement needs to be added. After confirming with the OK button the Browser closes and the reference is made. The EAnnotation can be named in the Source entry as 'reference_to_Poseidon_DSL'.
The last step to prepare the meta model is to make the ERNode inherit from the PoseidonCoreElement. This is important, because in the generator template it is necessary to access some attributes of it, e.g. the name of the element. In order to make this happen the ERNode in the *entity-relationship-editor.ecore* has to be marked and in the field ESuperTypes it is necessary to add the PoseidonCoreElement. This works, like in the previously described step, by clicking into the empty field and adding the PoseidonCoreElement in the Browser window. If all steps were executed correctly the *entity-relationship-editor.ecore* file should look like the one shown in Figure 4.1.

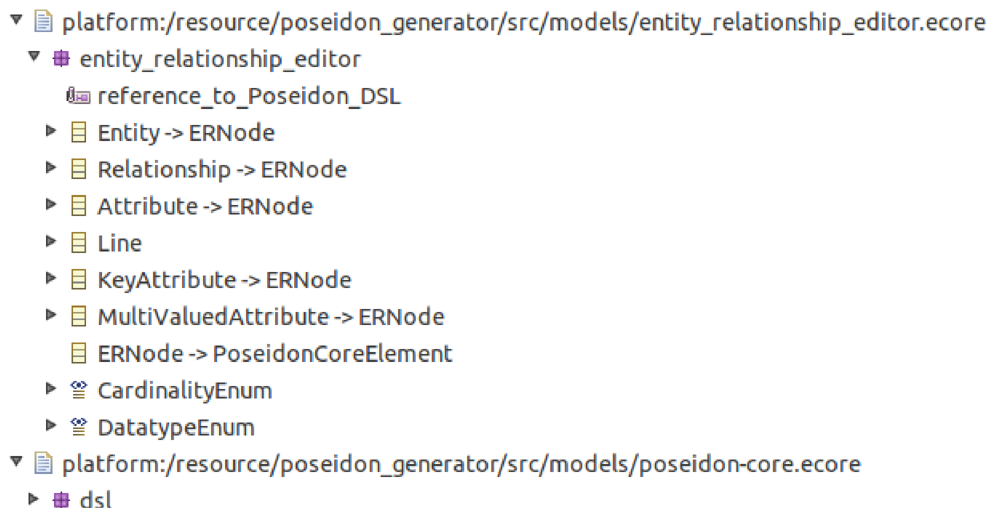

Figure 4.1: The prepared entity-relationship-editor.ecore file

4.2 Preparing the Editor

Now that the Ecore meta model has sucessfully been adjusted, the editor needs some adjustments as well before the generation of code will be possible. The first step is to implement a new menu that includes a menu item which is able to start the generation process. For doing so firstly the two ResourceIds for the menu as well as the menu item have to be added. This can be done in the class `CustomMainMenuResourceBundle` which can be found in the *com.gentleware.poseidon.custom.gui.menu* package. The ResourceIDs that need to be added can be seen in Listing 4.1

```
1 public static ResourceId CREATESQL = new ResourceId(
      MAIN_MENU_BUNDLE, "Create SQL Statements");
2 public static ResourceId CreateSQLStatement = new ResourceId(
      MAIN_MENU_BUNDLE, "Generate");
```

Listing 4.1: The ResourceIds for the new menu and menu item

Secondly the menu and the menu item can be implemented. Therefore the method `setupMenus` has to be overwritten. This can be done in the class `CustomMainMenu`, which is in the same package as the `CustomMainMenuResourceBundle` class. In order to overwrite `setupMenus` the menu must be added to the ER-Editor by passing over the ResoureceId to the responsible factory. After this a new `JMenuItem` needs to be created, which can be done with the method `addMenuItem`. The icon that is pictured in the menu item also has to be set up. This works by using the `setIcon` method, where an icon can be chosen, for example the icon 'check-one' which pictures a green arrow. The menu item also needs an action listener, which should execute the code when the item is clicked. The whole implementation can be seen in Listing 4.2.

```
1  @Override
2  protected void setupMenus() {
3    super.setupMenus();
4    ResourceId myMenuItemResourceId = CustomMainMenuResourceBundle.
       CREATESQL;
5      factory.addMenu(myMenuItemResourceId, "e");
6
7      JMenuItem createCreateItem = factory.addMenuItem(
         CustomMainMenuResourceBundle.CREATESQL,
8      MainMenuResourceBundle.Create, CustomMainMenuResourceBundle.
         CreateSQLStatement, "b");
9    createCreateItem.setIcon(com.gentleware.poseidon.util.IconLoader.
       getIconLoader().get("check-one"));
10     createCreateItem.addActionListener(new ActionListener() {
11     public void actionPerformed(ActionEvent e) {
12       GenerateOutputFromModelActionListener g = new
           GenerateOutputFromModelActionListener(applicationWindow);
13       g.actionPerformed(e);
14     }
15   });
16 }
```

Listing 4.2: The setupMenu method in the file CustomMainMenu.java

Inside the action listener a new `GenerateOutputFromModelActionListener` should be created. This is the class that actually enables the code generation. The method `actionPerformed` of this class initialises all necessary parameters and objects that are needed in the *GenerateFromPoseidon.mwe* file. Furthermore it also starts the mwe workflow. The method that runs the workflow returns a boolean value which is true when everything works fine and no errors occur during the code generation. Hence, this boolean could be used for a pop-up window that appears when something goes wrong during the generation process. This can be implemented easily in the `actionPerformed` function in the class `GenerateOutputFromModelActionListener` which is located in the package *com.gentleware.poseidon.custom.listeners*, as it is shown in Listing 4.3.

```
1 boolean worked = new WorkflowRunner().run(WORKFLOW_FILE_NAME, null,
      properties, slotContents);
2 if (!worked) {
3   this.applicationWindow.getCoordinator().invokeErrorDialog("
       FAILURE", "Failures occurred during the generation process");
4 }
```

Listing 4.3: The Pop-Up in GenerateOutputFromModelActionListener.java

If all steps were executed correctly it is possible to start the mwe workflow. At the moment there is just a sample output file which lists the names of all diagrams of the Entity Relationship Editor. This file can be found in the Poseidon project in the folder *generated-from-poseidon-project*.
The next change which has to be made in the Java code is the assembling of the connections between the nodes. This has to be done, because for now the only object that possesses information about connections between two nodes is the class Line. In order to be able to determine if an attribute belongs to an entity all lines have to be checked, whether they have references to the entity and the attribute. But in the Ecore model there are EReferences from both the entity and the relationship to the attribute. They were made for rendering possible that all attributes of an entity can be fetched by just having an entity object. In Poseidon an EReference is represented as a container which includes all objects that become referenced. The reason that they are realised as containers can be traced back to the value 'Upper Bound' which was set to '-1', as described in Section 3.2. These containers are still empty, even if an edge is connected to the node possessed by the container. This is caused by the fact that references are only stored in the lines and not in the intended container. In order to fill the container correctly some parts of the code have to be changed: the first modification considers the EReferences containsAttribute. Several of them exist, for example from node to attribute or from relationship to attribute, but only one method needs to be changed, because they all have the same edge which represents the line between them. The edge is LineFromERAToAttribute, which was created in Listing 3.2. Therefore it is necessary to change one Java class. It is called LineFromERAToAttributeArcGem and can be found in the package *com.gentleware.poseidon.custom.diagrams.edge.impl.* As it was done for picturing the cardinality the method `createBasicEdgeAppearanceFacet` needs to be overwritten. The end of the edge is always an attribute, so just

the start of the edge has to be distinguished. The start can either be an entity, a relationship, an attribute or a multi valued attribute. If the needed one was found the attribute can be stored in the container of the start node, like it is shown in Listing 4.4

```
@Override
protected BasicArcAppearanceGem createBasicEdgeAppearanceFacet() {
  EObject eObject = super.subject;
  WrappedLineImpl wraLiIm = (WrappedLineImpl) eObject;
  if(wraLiIm == null){
    return super.createBasicEdgeAppearanceFacet();
  }
  if(wraLiIm.basicGetStart() instanceof Entity ){
    Entity ent = (Entity) wraLiIm.basicGetStart();
    ent.getContainsAttribute().add((Attribute) wraLiIm.basicGetEnd
        ());
  }
  if(wraLiIm.basicGetStart() instanceof Relationship ){
    Relationship rel = (Relationship) wraLiIm.basicGetStart();
    rel.getContainsAttribute().add((Attribute) wraLiIm.basicGetEnd
        ());
  }
  if(wraLiIm.basicGetStart() instanceof Attribute ){
    Attribute att = (Attribute) wraLiIm.basicGetStart();
    att.getContainsAttribute().add((Attribute) wraLiIm.basicGetEnd
        ());
  }
  if(wraLiIm.basicGetStart() instanceof MultiValuedAttribute ){
    MultiValuedAttribute mvatt = (MultiValuedAttribute) wraLiIm.
        basicGetStart();
    mvatt.getContainsAttribute().add((Attribute) wraLiIm.
        basicGetEnd());
  }
  return super.createBasicEdgeAppearanceFacet();
}
```

Listing 4.4: The containers for the attributes.

The same has to be done for the key attributes and multi valued attributes, that both can be connected to an entity. For every type of these attributes there are edges, which are responsible for them. The same changes have to be made in both of the classes `LineFromEntityToKeyAttributeArcGem` and `LineFromEntityToMultiValuedAttributeArcGem`, which are in the same package like the `LineFromERAToAttributeArcGem` class. In both classes the method `createBasicEdgeAppearanceFacet` needs to be overwritten, as well. This works like it is shown in Listing 4.5 and 4.6.

```java
@Override
protected BasicArcAppearanceGem createBasicEdgeAppearanceFacet() {
  EObject eObject = super.subject;
  WrappedLineImpl wraLiIm = (WrappedLineImpl) eObject;
  Entity ent = (Entity) wraLiIm.basicGetStart();
  ent.getContainsKey().add((KeyAttribute) wraLiIm.basicGetEnd());
  return super.createBasicEdgeAppearanceFacet();
}
```

Listing 4.5: The container for the key attributes.

```java
@Override
protected BasicArcAppearanceGem createBasicEdgeAppearanceFacet() {
  EObject eObject = super.subject;
  WrappedLineImpl wraLiIm = (WrappedLineImpl) eObject;
  Entity ent = (Entity) wraLiIm.basicGetStart();
  ent.getContainsMVA().add((MultiValuedAttribute) wraLiIm.
      basicGetEnd());
  return super.createBasicEdgeAppearanceFacet();
}
```

Listing 4.6: The container for the multi valued attributes

The next and the last iteration, which affects the editor, is to store the connected entities in the relationship. This works relatively similar to the changes that have been made at the edges of the various attributes. A difference regarding this case is that a relationship has exactly two entities that are connected to it. These EReferences, that are called Entity1 and Entity2, are not exactly containers, because they can only store one entity. They are no containers, because in Section 3.2 the 'Upper Bound' of these two EReferences were left on their default value, which is '1'. According to this a differentiation between the two connected entities has to be done before storing them. The same had also been done in the Section 3.4 in Listing 3.10. The differentiation there was made for the cardinality, but it differs between two connected entities which is exactly what is needed here. For remembrance: The idea was to store a value which shows if an entity is the first or the second one that is connected to a relationship. Therefore a boolean variable was implemented, which was called `isFirst`. This variable will now be used for the storing of the entities too. Still another differentiation has also to be done, it is necessary to check if the start or the end is of the type entity or relationship. In order to do this a new method is implemented, which is called `storeEntity`. This method has to be in the class `LineFromEntityToRelationshipArcGem`. The next change contains starting this method at the end of the overwritten `createBasicEdgeAppearanceFacet` function before returning the `formAppearance`. This works by calling the storeEntity with `storeEntity(wraLiIm.isIsFirst())`. The exact implementation can be seen in listing 4.7.

```java
private void storeEntity(boolean isFirst){
  EObject eObject = super.subject;
  WrappedLineImpl wraLiIm = (WrappedLineImpl) eObject;
  Relationship rel = null;
  if(wraLiIm.basicGetEnd() instanceof Relationship ){
    rel = (Relationship) wraLiIm.basicGetEnd();
    if (isFirst) {
      rel.setEntity1((Entity) wraLiIm.basicGetStart());
    } else {
      rel.setEntity2((Entity) wraLiIm.basicGetStart());
    }
  } else {
    rel = (Relationship) wraLiIm.basicGetStart();
    if (isFirst) {
      rel.setEntity1((Entity) wraLiIm.basicGetEnd());
    } else {
      rel.setEntity2((Entity) wraLiIm.basicGetEnd());
    }
  }
}
```

Listing 4.7: The code for storing the two connected entities in the relationship, in the file LineFromEntityToRelationshipArcGem.java.

If all steps were executed as described it is now possible to access all referenced nodes from the nodes that are connected to them. As we will see this is very helpful for implementing the generator.

4.3 Writing the Generator

As mentioned in Section 4.2, the generator is implemented in the file Main.xpt in the poseidon_generator project. The default implementation can be deleted and the first thing is to import all needed ecore models. After doing this a define block for the PoseidonCoreRootElement can be made in which the generator iterates over every diagram that is opened in the ER-Editor. The generator creates an SQL file with the name of the diagram and starts the next define block with the keyword EXPAND. In this define block the generator iterates over every entity, which is stored in the container `__ownedMember` and executes the defined entity template. A criterion for iterating an entity is that it still needs to be in the diagram and that it was not deleted by the user. The same happens with all relationships, but this will be explained later. Until now the generator looks like it is shown in Listing 4.8.

```
1  «IMPORT dsl»
2  «IMPORT entity_relationship_editor»
3
4  «DEFINE main FOR dsl::PoseidonCoreRootElement»
5  «FOREACH j_diagramHolder.diagram AS e-»
6
7  «FILE e.__poseidonName + '.sql'-»
8  «EXPAND diagramNames FOR this»
9  «ENDFILE»
10 «ENDFOREACH»
11 «ENDDEFINE»
12
13 «DEFINE diagramNames FOR dsl::PoseidonCoreRootElement-»
14 «EXPAND Entity FOREACH __ownedMember.typeSelect(
       entity_relationship_editor::Entity).select(e|e.j_deleted == 0)-
       »
15 «EXPAND Relationship FOREACH __ownedMember.typeSelect(
       entity_relationship_editor::Relationship).select(e|e.j_deleted
       == 0)-»
16 «ENDDEFINE»
17
18 «DEFINE Entity FOR entity_relationship_editor::Entity»
19 «ENDDEFINE»
20
21 «DEFINE Relationship FOR entity_relationship_editor::Relationship»
22 «ENDDEFINE»
```

Listing 4.8: The code for creating an empty SQL file with the generator.

At this point the generator just creates new files and nothing else happens. The reason for this is that the define blocks for the entities and relationships are still empty. Hence, they need to be implemented. The first block that should be realised is the one for the entities. Therefore the define block of the entities has to be implemented. In this block also some relationships and lines are needed, because they are necessary for the '1:n' cardinalities. For this type of cardinality a foreign reference has to be created in the table, like it was shown in Section 2.1. For this reason the lines and the relationships are needed. But first the table will be created and all attributes, multi valued attributes and key attributes are added. Now the containers, that were implemented in Section 4.2, are being used. Therefore it is necessary to iterate over the container and get the names and types of the attributes, that are stored inside of it. For the key attributes the keyword CONSTRAINT needs to be added. After all attributes were created the cardinality can be implemented. This is only necessary if the entity is on the side of a relationship where the 'n' is displayed, like it was mentioned in Section 2.1. Hence, this has to be checked and if it is the case the reference can be inserted. This needs to be done for every 'n' cardinality which is connected to an entity. The constraint 'ON DELETE SET NULL' is chosen as default behaviour that takes place if the foreign entry is deleted. This can be seen in Listing 4.9.

```
«DEFINE Entity FOR entity_relationship_editor::Entity-»
«LET this.eRootContainer.eContents.typeSelect(entity_relationship_editor::Relationship) AS relantionships-»
«LET this.eRootContainer.eContents.typeSelect(entity_relationship_editor::Line) AS lines-»
CREATE TABLE «__poseidonName»
(
«FOREACH containsAttribute.select(e|e.j_deleted == 0) AS attribute-»
  «attribute.__poseidonName» «attribute.type»,
«ENDFOREACH-»
«FOREACH containsMVA.select(e|e.j_deleted == 0) AS mvattribute-»
  «mvattribute.__poseidonName «mvattribute.type»,
«ENDFOREACH-»
«FOREACH containsKey.select(e|e.j_deleted == 0) AS key-»
  «key.__poseidonName» «key.type» NOT NULL,
  CONSTRAINT pk_«this.__poseidonName» PRIMARY KEY(«key.__poseidonName»),
«ENDFOREACH-»
«FOREACH relantionships.select(e|e.j_deleted == 0) AS relationship-»
«IF relationship.cardinality.toString() != "1:1" && relationship.cardinality.toString() != "n:m"-»

«FOREACH lines.select(e|e.cardinality == 2) AS line-»
«IF (line.end == this && line.start == relationship) || (line.start == this && line.end == relationship)-»
«IF relationship.Entity1 == this-»
  «relationship.__poseidonName» «relationship.Entity2.containsKey.get(0).type»,
  CONSTRAINT fk_«__poseidonName»_«relationship.__poseidonName» FOREIGN KEY(«relationship.__poseidonName»)
            REFERENCES «relationship.Entity2.__poseidonName»(«relationship.Entity2.containsKey.get(0).__poseidonName»)
            ON DELETE SET NULL
«ELSEIF relationship.Entity2 == this-»
  «relationship.__poseidonName» «relationship.Entity1.containsKey.get(0).type»,
  CONSTRAINT fk_«__poseidonName»_«relationship.__poseidonName» FOREIGN KEY(«relationship.__poseidonName»)
            REFERENCES «relationship.Entity1.__poseidonName»(«relationship.Entity1.containsKey.get(0).__poseidonName»)
            ON DELETE SET NULL
«ENDIF-»
«ENDIF-»
«ENDFOREACH-»
«ENDIF-»
«ENDFOREACH-»
«ENDLET-»
«ENDLET-»
);
«ENDDEFINE»
```

Listing 4.9: The entity define block in the generator.

The next step is to implement the last define block which is the one for the relationships. It covers the relations with the cardinalities '1:1' and 'n:m'. They are both treated the same way, like it is explained in Section 2.1. This means that it only needs to be checked if the relationship is one of these types of cardinality. Is this the case the table will be created with all its attributes, primary and foreign keys. At this step the references Entity1 and Entity2, that were created in Section 3.2 and implemented in Section 4.2, are being used. Although they store the connected entities, it is nevertheless possible to get the primary keys of these entities. They are again accessible via the EReference containsKey of the entity. Thus, it is not necessary to initialise other elements for the define block of the relationships. The whole define implementation is shown in Listing 4.10. If the whole generator template was implemented correctly, it is now possible to generate an SQL file from the created diagram. This file can be found in the above mentioned folder *generated-from-poseidon-project* in the Poseidon workspace.

```
1  «DEFINE Relationship FOR entity_relationship_editor::Relationship»
2  «IF this.cardinality.toString() == "1:1" || this.cardinality.toString() == "n:m"-»
3  «IF Entity1.containsKey.size > 0 && Entity2.containsKey.size > 0-»
4  
5  CREATE TABLE «__poseidonName»
6  (
7  «FOREACH containsAttribute.select(e|e.j_deleted == 0) AS attribute-»
8    «attribute.__poseidonName» «attribute.type»,
9  «ENDFOREACH-»
10   «Entity1.containsKey.get(0).__poseidonName» «Entity1.containsKey.get(0).type» NOT NULL,
11   «Entity2.containsKey.get(0).__poseidonName» «Entity2.containsKey.get(0).type» NOT NULL,
12 
13   CONSTRAINT pk_«__poseidonName» PRIMARY KEY («Entity1.containsKey.get(0).__poseidonName», «Entity2.containsKey.get(0).__poseidonName»),
14   CONSTRAINT fk_«__poseidonName»_1 FOREIGN KEY(«Entity1.containsKey.get(0).__poseidonName»)
15                      REFERENCES «Entity1.__poseidonName»(«Entity1.containsKey.get(0).__poseidonName»)
16                      ON DELETE SET NULL,
17   CONSTRAINT fk_«__poseidonName»_2 FOREIGN KEY(«Entity2.containsKey.get(0).__poseidonName»)
18                      REFERENCES «Entity2.__poseidonName»(«Entity2.containsKey.get(0).__poseidonName»)
19                      ON DELETE SET NULL,
20 
21 );
22 «ENDIF-»
23 «ENDIF-»
24 «ENDDEFINE»
```

Listing 4.10: The define block for the relationships "1:1änd "n:m".

4.4 Limitations of the code generation

Like it is shown in Section 3.6, there are some limitations of the code generation too. The first one is that all attributes which are belonging to a relationship that has the cardinality '1:n' or 'n:1' are being ignored. This is caused, because if a relationship has one or more attributes a separate table for the relationship has to be created. But for this type of cardinality this is not the case.
The next limitation or constraint is that a multi valued attribute is mapped to the SQL code just like a normal attribute. The reason for this is that there is no way to differ the various multi valued attributes in the generated code, and so they would all have the same name. As well there is no information about how often the multi valued attribute should be created.
The last limitation of the code generation is that all attributes that are connected to other attributes or multi valued attributes are also being ignored. This is a constraint chosen by the author for simplify matters, because if the attributes were considered they were created like normal attributes. This would have resulted in the attributes not appearing in the code which would lead to ignore these attributes. Another constraint of the code generation is that the generated SQL code just works with Oracle databases. The reason for this is outlined in Section 2.1.

Chapter 5

Conclusion

It is very impressive how long it takes to build an editor with Poseidon for DSL. It took only a few weeks, after researching and solving a few tutorials, to build an Entity Relationship Editor with Poseidon for DSL. For this reason it can be said that it is faster to use Poseidon and the MDSD approach to build this editor instead of programming it from scratch. The good thing is that Poseidon provides an empty editor, like it is shown in Figure 3.3. In this picture you can see some basic functions, like undo or saving, that are already provided by Poseidon and do not have to be implemented, which saves a lot of time. But there is also some work to do, that needs a bit more effort. First of all it is important to familiarise yourself with MDSD, Poseidon for DSL and the principles of a DSL and the EMF. This can take a lot of time if you are absolutely new to these fields. This was also mentioned in Section 2.2, where the know-how of the employees was quoted as a prerequisite for MDSD, which can be confirmed after having created the ER-Editor. In this section it was also said that after having the generated code there will still be code which has to be written by hand. This is a declaration that can be affirmed as well. This can be considered as a small disadvantage of MDSD or Poseidon, because for the creation of the ER-Editor it was necessary to write a lot of code, so that the editor fits all needs. This step is very hard to avoid, because there are unpredictable needs for every domain. It is presumable that the developers of Poseidon never thought about the creation of a double ellipse or an ellipse with its name being underlined. Creating those took a lot of time, because there were no references or documentations about how certain issues could be solved. Consequently finding solutions for occurring problems turned out to be difficult and would have been impossible to solve without the support of a Poseidon developer. One explanation for this is the fact that Poseidon is closed source and sometimes it is very hard to understand what the purpose of a certain method or a class is. My experience is that the purpose of an unknown class often can be discovered by reviewing the source code of it which is of course restricted to available code.

What really impressed me was that in almost every step of the creation of the ER-Editor a piece of MDSD can be found. This starts with defining the meta model, creating a model, generating runnable code and adjusting it. This

shows that MDSD is a concept that truly works and can be used to develop runnable software.
I also liked the way how to create the code generator, because everything was given by Poseidon for DSL. The only thing to do was to write a template in Xpand and the code could be generated. During the work of this thesis the new Xtext framework, Xtext2, was released. It would have been nice if the ER-Editor used the new version of Xtext. But there have been a lot of changes and new features have been installed compared to the former Xtext version, like some fundamental principles and interfaces. Hence, in order to be able to use the new Xtext2 with Poseidon a lot of adjustments would have been necessary and the migration process would have taken a lot of time. Additionally the most of the changes could only have been done by Poseidon developers, because the changes effect the code generation and the concepts of the Poseidon Xtext models. Though I am sure it is just a question of time until Poseidon will be available for Xtext2.

I also liked the syntax of the DSLs that is specified by Poseidon. In my opinion it is very clear what the purpose of a defined variable is, because the names of them are chosen very well and self explaining. I also enjoyed the fact that Poseidon uses Xtext, which provides the feature of the code completion in a DSL file like it is known from Eclipse for Java. This feature made the development a bit more effortless, because it proposes possible options to made during the process of defining the model. The principle of DSLs is in my opinion very competent. The idea that the developer can define a language by his own is really helpful, because all needs can be fitted and there are no unnecessary features that nobody would use anyway. I think the advantage that a DSL is easy to read, especially for users that are not familiar with programming, is very helpful too. This results in the possiblity that your DSL can be shown to other people in order to get a feedback about the logic of it and if all aspects were defined correctly or if something is missing. I think this works especially well if the DSL is a graphical one, like the one that can be created with the ER-Editor. The fact that with the ER-Editor an entity relationship model can be created which is a graphical DSL at the same time could be a little bit confusing. But the user does not have to think about it, he just makes the entity relationship diagram and creates a graphical DSL without even knowing it. I think the ER-Editor is a very good example that for almost every type of diagram a graphical DSL can be developed by using Poseidon for DSL.

In my opinion the approach of MDSD is very helpful, but I do not know if it is used in companies often, because of the lack of personal experiences. I can imagine that some project leaders are afraid of switching the way how they usually develop software to developing it whit the MDSD approach. I am not sure if I would recommend the MDSD approach for any project, because I think it just works if every developer is familiar with the concept of it. It also depends on the software which will be developed and on the tools that are available for it. In my eyes MDSD makes only sense if the developed software is object oriented and is not hardware oriented, for example like a driver for a printer.

The effort that is necessary in order to become familiar with the used tools and MDSD should be determined in advance and it should be compared to the effort of a conventional developing way. If the effort of using MDSD is worth it I think I would recommend to use it in projects.

Bibliography

[Biermann et al., 2006] Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., and Weiss, E. (2006). Graphical definition of in-place transformations in the eclipse modeling framework. *Model Driven Engineering Languages and Systems*, pages 425–439.

[Brücher et al., 2011] Brücher, C., Jüdes, F., and Kollmann, W. (2011). *SQL thinking: vom Problem zum SQL-Statement.* mitp, 1. aufl. edition.

[Chen, 1976] Chen, P. P.-S. (1976). The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36.

[Fowler, 2010] Fowler, M. (2011 [erschienen 2010]). *Domain-specific languages.* Addison-Wesley, 1. print. edition.

[Gamma et al., 2010] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2010). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Pearson Education.

[Gentleware AG, 2011] Gentleware AG (2011). Developing a dsl with poseidon for dsls 2.0 - petri nets. Website. Available online at http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon-for-dsls/tutorial/2.0.0/Developing_a_DSL_with_Poseidon_for_DSLs_2.0_Petr.html; visited on June 10th 2011.

[Ghosh, 2011] Ghosh, D. (2011). *DSLs in action.* Manning, Greenwich, Conn. [u.a.].

[IBM Corporation and others, 2011] IBM Corporation and others (2011). org.eclipse.emf.ecore (emf javadoc). Website. Available online at http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html#details; visited on August 10th 2011.

[Jouault and Bézivin, 2006] Jouault, F. and Bézivin, J. (2006). Km3: a dsl for metamodel specification. *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185.

[Kemper and Eickler, 2009] Kemper, A. and Eickler, A. (2009). *Datenbanksysteme.* Oldenbourg Wissensch.Vlg.

[Kovse and Härder, 2002] Kovse, J. and Härder, T. (2002). Generic xmi-based uml model transformations. *Object-Oriented Information Systems*, pages 183–190.

[Künneth, 2010] Künneth, T. (2010). Texte in swing unterstreichen. Website. Available online at http://kuennetht.blogspot.com/2010/01/texte-in-swing-unterstreichen.html; visited on July 17th 2011.

[Raja and Lakshmanan, 2010] Raja, A. and Lakshmanan, D. (2010). Domain specific languages. *International Journal of Computer Applications*, 1(21):99–105.

[Stahl, 2007] Stahl, T. (2007). *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt-Verl., 2., aktualisierte und erw. aufl. edition.

[Steinberg, 2009] Steinberg, D., editor (2009). *EMF - Eclipse modeling framework*. The eclipse series. Addison-Wesley, Boston, Mass., 2. ed., revised and updated edition.

[Trompeter and Beltran, 2007] Trompeter, J. and Beltran, J. C. F., editors (2007). *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. entwickler.press.

[Van Deursen et al., 2000] Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36.

[Van Deursen et al., 2002] Van Deursen, A., Klint, P., and Visser, J. (2002). Domain-specific languages. *The Encyclopedia of Library and Information Science*, pages 113–127.