A tool for generating relational database schema from UML class diagram
I agree with the publication of this diploma thesis according to the request no. 26, para. 9 in *Educational and examinational rules of Master study programme at VŠB-TU Ostrava*.

Ostrava 7. May 2010

I hereby declare that I have elaborated this diploma thesis independently and enumerated all the literature sources and publications that I have used.

Ostrava 7. May 2010
I would like to thank everyone who supported me during this work, without them it can never be done.
Abstract

The aim of this thesis is to develop an application with graphical user interface for drawing class diagram and ability to generate an SQL script from the created diagram. The development process will examine the possibility of mapping UML class diagram to relational data model. The state of the class diagram can be saved to an XML file for later use.

Keywords: UML, class diagram, relational database, XML

Abstrakt

Cílem této práce je vyvinout aplikaci s grafickým rozhraním pro vytváření třídního diagramu a schopností generování SQL skriptu z tehož diagramu. Proces vyvoje bude ověřovat možnost mapování diagramu tříd na relační model. Stav třídního diagramu může být uložen do XML souboru pro pozdější použití.

Klíčová slova: UML, třídní diagram, relační databáze, XML
## List of abbreviations and symbols

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>ER</td>
<td>Entity Relationship</td>
</tr>
<tr>
<td>EER</td>
<td>Enhanced Entity Relationship</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
</tbody>
</table>
# Contents

1 Introduction 6

2 Relevant technologies 8
   2.1 Conceptual data modeling ........................................... 8
   2.2 UML ................................................................. 9
   2.3 Relational database management system ......................... 11
   2.4 UML modeling tools ............................................... 12
   2.5 ER modeling tools ............................................... 14

3 Requirements 16
   3.1 Requirements specification ....................................... 16
   3.2 Use cases ......................................................... 16
   3.3 Implementation environment ...................................... 21
   3.4 Analysis model .................................................. 21
   3.5 States of diagram ............................................... 22

4 Design 24
   4.1 Concepts .......................................................... 24
   4.2 Design classes .................................................... 24
   4.3 Visualization ...................................................... 30
   4.4 Working with SQL ............................................... 35
   4.5 Working with XML ............................................... 36

5 Mapping class diagram to relational data model 40
   5.1 Basic mapping ...................................................... 40
   5.2 Data type mapping ............................................... 40
   5.3 Connection mapping ............................................... 40
   5.4 Inheritance mapping .............................................. 43
   5.5 Method mapping .................................................. 44

6 Implementation 45
   6.1 Enumerations and constants ...................................... 45
   6.2 DiagramState structure .......................................... 46
   6.3 Class representation .............................................. 47
   6.4 Connection representation ....................................... 49
   6.5 SQL export implementation ..................................... 51
   6.6 Handling the XML tasks ......................................... 52
   6.7 Editing attributes and methods .................................. 55
   6.8 Other problems ................................................... 57
7 Results
7.1 Class with attributes .............................................. 59
7.2 Mapping association to new table ..................................... 59
7.3 Mapping association to table constraint .............................. 60
7.4 Mapping recursive association ...................................... 61
7.5 Inheritance .......................................................... 62
7.6 Mapping method to different SQL dialects .......................... 63
7.7 Mapping a complete diagram ....................................... 64
7.8 In comparison with other tools ..................................... 64

8 Conclusions .......................................................... 66

9 References .......................................................... 67

Appendix ............................................................. 68

A Contents of CD ......................................................... 69

B SQL code of a complete database schema ............................ 70

C XML schema of saved file ........................................... 73

D Quick start guide ......................................................... 78
D.1 Installation .......................................................... 78
D.2 Main interface ......................................................... 78
D.3 Basic functions ....................................................... 78
List of Tables

1.Multiplicity values ................................................. 12
2. Relative positions and orthogonal forms ............................... 34
3. SQL-92 data types .................................................. 41
4. Mapping programming language types .................................. 41
5. Primary key decision of bidirectional association mapping .......... 42
6. Mapping of aggregation, composition and unidirectional association ... 43
7. Source class position ............................................... 50
8. Attributes for customizing property ................................. 56
# List of Figures

1. Example of an ER diagram ........................................... 9
2. Example of a UML class diagram ................................. 10
3. Defining components ................................................. 17
4. Manipulating components ........................................... 18
5. XML tasks .......................................................... 19
6. SQL tasks .......................................................... 20
7. Other tasks ......................................................... 20
8. Analysis classes ....................................................... 22
9. States of diagram .................................................... 23
10. Classifier and connection concepts .............................. 25
11. Logical and visual parts of classifier ............................. 26
12. Logical and visual parts of connection .......................... 27
13. Design of attribute and method .................................. 29
14. Diagram and DiagramState ....................................... 29
15. General view of design classes ................................... 30
16. Relative position of source class ................................. 31
17. The alpha angle .................................................... 32
18. Orthogonal forms .................................................... 32
19. Class sections ....................................................... 33
20. Connection examples ............................................... 35
21. Recursive association ............................................... 35
22. The process of exporting to SQL ................................. 36
23. The process of exporting to XML ................................. 37
24. The process of loading from XML ............................... 39
25. Additional parts of connections .................................. 51
26. Class with attributes ............................................... 60
27. Mapping association to junction table .......................... 60
28. Mapping association to table constraint ........................ 61
29. Recursive association ............................................... 61
30. Inheritance ........................................................ 61
31. Method .............................................................. 63
32. Example of a complete diagram .................................. 64
33. Installation complete ............................................... 79
34. Main interface ....................................................... 79
List of Listings

1. Undo ................................................................. 46
2. Redo ................................................................. 46
3. Mark diagram as changed ................................. 47
4. MouseDown event handler ................................. 48
5. MouseMove event handler ................................. 48
6. MouseUp event handler ..................................... 48
7. Calculate polyline points ................................. 49
8. SQL export at diagram level ............................ 51
9. Get primary keys ............................................... 52
10. XML export at diagram level ............................ 53
11. XML export at connection level ....................... 53
12. Load from XML .................................................. 54
13. Load classes ..................................................... 54
14. AttributeLabel MouseClick event handler ......... 55
15. AttributeTypeConverter class ......................... 55
16. AttributeType property ..................................... 56
17. Image export ..................................................... 58
18. Result table ....................................................... 59
19. Association mapped to junction table .............. 59
20. Association mapped to table constraint ............. 60
21. Mapping recursive association ......................... 61
22. Mapping inheritance to database view .............. 62
23. Mapping method according to MSSQL ............... 63
24. Mapping method according to Oracle ............... 63
25. Mapping method according to MySQL ............... 64
26. Mapping a complete diagram ......................... 70
27. XML schema ....................................................... 73
1 Introduction

In the era of the information explosion, software applications with the need of persisting data into a database are more and more popular. As a result, modeling and creating database schema should be more convenient with acceptable time spending and effort to reduce the overall cost of a software project. The traditional method is writing scripts and then executing them in a database management system, but it is an exhausting and error-prone work, especially in large systems. In order to facilitate the work of creating database scripts, graphical representation of database schema can be used to present the database structure visually. Various techniques have been developed to support modeling and design. The most advantageous way to capture the database schema quickly and conveniently is to express it in a diagram. The concrete types of diagrams will be discussed in the next sections. Based on the diagram, developer can write the database scripts in shorter time. But with the demand on the rate of progress of projects, a faster way to produce database scripts came up. The database script can be created directly from the diagram of database schema by an automatic process mapping the diagram objects to the database objects. This process brings both advantages and disadvantages. The greatest advantage is that time spending on writing database scripts is no longer necessary, thus hastening the progress of project. On the other hand, automatically created scripts cannot be as accurate as manually created ones. The mapping process is determined by predefined algorithms, hence it is not flexible. To improve the result of mapping process, many solutions have been proposed and this problem has become a separate branch of interest in computer science field. Until today, there has not been any perfect solution that can make the automatic mapping process completely replace the manual method yet.

This thesis concentrates on the problem of mapping UML class diagram to relational database schema. The differences between the source and the destination of mapping will be discussed. The reason why an ultimate mapping solution is hard to find will also be analyzed. Currently, there are several tools for creating database schema automatically from diagram, but they still do not satisfy the demand of the users. The aim of this work is to design and implement an application that is capable of visually creating conceptual data modeling of persistent data and exporting it to the schema of the relational database. The application will have the main functionality creating database script from the diagram of database schema. In this work, beside the design and implementation of the application, a great amount of time is dedicated to studying the method of automatically mapping the diagram objects to the database objects. Although not proposing a new approach to mapping problem, the thesis will try to employ and combine existing solutions to produce a result as good as possible.

The work is organized as follows: Section 2 will introduce the relevant technologies needed to implement the application and the brief overviews of existing conceptual data modeling tools, their advantages as well as disadvantages. All the requirements and
expected functionality as well as the analysis will be discussed in Section 3. In Section 4 the process of design will be described. Entire Section 5 is dedicated to explaining the mapping between data models. Section 6 goes into the implementation based on the design from previous chapters. Section 7 shows the result of the implementation, mainly the mapping aspect, with several practical examples. Conclusions and evaluations are presented in Section 8.
2 Relevant technologies

2.1 Conceptual data modeling

Conceptual data modeling is the process of developing a semantic description of a database application. Its goal is to capture real-world requirements in a simple and meaningful way that is understandable by both the database designer and the end user. The process of conceptual data modeling is database independent, involving the analysis of requirements and the development of a high-level semantic design of database contents and application constraints. In order to perform that process, various solutions have been proposed. The Entity Relationship (ER) model is one of the earliest and most well-known techniques associated with conceptual data modeling. It was introduced by Peter Chen in 1976 [1] and has become the milestone in the area of data modeling and database design. The ER model provides a database independent approach to describe the entities involved in a database application, together with the relationships and constraints that exist between such entities. Since the introduction of the ER model, many models were defined as enhancements to the ER modeling approach. The most significant one is the Enhanced Entity Relationship (EER) model which extends the original ER model with more advanced features such as generalization and specialization.

2.1.1 Entity relationship diagram

The basic ER model consists of three types of components: entity, relationship and attribute. An ER diagram can be used to express an ER model. Figure 1 shows an example of ER diagram. More details of ER diagram can be found in [3, 7]. An ER diagram includes the following components:

**Entity** An entity is represented by a rectangle with the name of the entity inside the rectangle. It usually denotes a person, place, thing or event of informational interest. Generally, entities can be seen as abstractions of the objects in the real world. A particular occurrence of an entity is called an entity instance.

**Relationship** A relationship is denoted as a diamond with lines connecting the diamond to the entities involved in the relationship. The name of the relationship is placed inside the diamond. Relationship represents real-world association between entities.

**Attribute** Both entity and relationship can have attributes. An attribute represents value that characterizes its owner. In the ER diagram, an attribute is denoted as an ellipse with the attribute name inside and attached by a line to the entity or relationship it describes. There are two kinds of attributes: identifiers and descriptors. An identifier is used to uniquely determine an instance of an entity. Identifier is referred to as key attribute of entity and its name is underlined. A descriptor is used to specify a common characteristic of an entity instance. Descriptor is referred to as non-key attribute.
Cardinality represents the number of entity instances participating in a relationship. In ER diagram, cardinality can be one-to-one (1:1), one-to-many (1:N), many-to-one (N:1) or many-to-many (M:N). The cardinality number is placed next to the related entity on the line representing the relationship.

Participation constraints can either be partial or total. Partial participation indicates that the participation of the entity in a relationship is optional. It is denoted by a single line between a rectangle and a diamond. On the other side, total participation indicates required participation of the entity in a relationship. It is denoted by a double line between a rectangle and a diamond.

2.2 UML

In the 1990s, with the rapid evolution of the object-oriented programming, the object-oriented modeling became more and more important. Database design was also influenced by this trend. Instead of entities, a database application will be described by objects. As a result, many different modeling languages existed at the same time. But one of them gained population and became the standard in the object-oriented modeling: the UML. The Unified Modeling Language\(^1\) (UML) is a visual, object-oriented language that captures both the static and dynamic aspects of a software system. UML is the result of the combination of various modeling techniques in separate fields such as data modeling, business modeling and component modeling. The first version of UML appeared in

\(^1\)Further information at http://www.omg.org/technology/documents/formal.uml.htm
1997 and was adopted by Object Management Group\(^2\) (OMG) in the same year. The language has gradually developed from UML 1.x to UML 2 as more types of diagram have been added. It has become the de-facto standard in software engineering and been supported by the top companies in the software industry. With UML, developers have a tool for visualizing system’s blueprint, recording the artifacts and exchanging ideas among the development teams. UML 2 offers 14 types of diagram divided into two categories: structure diagrams and behavior diagrams. This thesis concentrates on the modeling the database schema, so the static view, concretely class diagram, will be examined in detail. Specification of other types of diagram can be found in [4].

### 2.2.1 Class diagram

Perhaps class diagram is the most well-known diagram type in UML. It is a useful tool for capturing the structure of an object oriented system. The main purpose of class diagram is to describe the static aspect of software system, including business assets and resources or system architecture. Figure 2 shows an example of the class diagram. More about the class diagram can be found in [5].

![Figure 2: Example of a UML class diagram](http://www.omg.org)

\(^2\)Further information at http://www.omg.org
Classifier  The purpose of the class diagram is to show the types being modeled within the system. A type could be a class, an interface, a data type or a component. These types are called by a common name: classifier. Every classifier is represented by a rectangle containing three compartments stacked vertically. The top compartment shows the name of the classifier, the middle compartment lists its attributes and the bottom compartment lists its methods representing its operations. The top compartment is mandatory but the others are optional.

The middle compartment shows each attribute of the classifier on a separate line. The line has the following format: 
\[
\text{\langle stereotype \rangle} | \text{visibility} | \text{attributeName}: \text{type} = \text{defaultValue}
\]

Only the attribute name is mandatory, all the components in the square brackets are optional.

Like the attributes, each method is placed on its own line in the bottom part. Each line has the following format:
\[
\text{\langle stereotype \rangle} | \text{visibility} | \text{methodName}(\text{parameterList}): \text{returnType}
\]

Again only method name is mandatory, other components are optional.

Inheritance  Inheritance is an important concept in object-oriented design. It refers to the ability of one class (child class) to inherit the functionality of another class (super class), and then add its own functionality. To model inheritance on the class diagram, a solid line is drawn from the child class with a closed and unfilled arrowhead pointing to the super class. In the example in Figure 2, Admin and Member are the child classes of the super class User.

Association  When the classifiers are related to each other, an association is used to represent the relationship between them. There are several types of association such as bidirectional and unidirectional association, aggregation and composition. Depending on the type, an association will have different visual representation. For example, a bidirectional association is indicated by a solid line drawn between two classifiers. At either end of the line, a role name and a multiplicity value are placed. The name of the association is placed in middle of the line. A unidirectional association in addition has an arrowhead at the second end of the line, but does not have the role name at the first end. A complete specification of each association type can be found in [5]. Table 1 shows the common multiplicity values and their meanings.

With a lot of advanced features, class diagram can be used to form the database schema where classifiers represent the tables and associations represent the relationships between them.

2.3 Relational database management system

Relational database management system (RDBMS) is a database management system based on relational data model. The theory of relational data model was firstly intro-
<table>
<thead>
<tr>
<th>Multiplicity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>Zero or one</td>
</tr>
<tr>
<td>1</td>
<td>Only one</td>
</tr>
<tr>
<td>*</td>
<td>Zero or more</td>
</tr>
<tr>
<td>1..*</td>
<td>One or more</td>
</tr>
<tr>
<td>a..b</td>
<td>minimally a and maximally b</td>
</tr>
</tbody>
</table>

Table 1: Multiplicity values

duced by E.F. Codd in 1970 [6]. Such data model persists data into relations. A relation, which is defined as a set of tuples that have the same attributes, is usually described as a table, which is organized into rows and columns. Each tuple of a relation must be uniquely identified by one or more attributes. That attribute (or group of attributes) is referred to as the primary key. A reference from a relation to another one results in a foreign key, meaning that the referencing tuple has an attribute with the value of a key in the referenced tuple.

**SQL** The Structured Query Language [9] is a standard language for defining and manipulating data in RDBMS. SQL consists of Data Definition Language (DDL) and Data Manipulation Language (DML). DDL is used to create, drop or alter the structure of database, while the purpose of DML is to add, update and delete data. The SQL standard has been developed through several revisions with the first is SQL-86 and the latest is SQL2008. The standard used in this thesis is SQL-92, since object-oriented features are not used in the result SQL scripts.

### 2.4 UML modeling tools

This section presents the overviews of some typical UML modeling tools widely used on the world, their pros and cons, and the reasons why a new tool should be developed to offer an alternative way in data modeling.

**Microsoft Office Visio**³ Visio is a commercial diagramming software product of Microsoft for visualizing various types of diagram: business workflow, engineering, software and database, network etc. For the purpose of conceptual data modeling in UML, Visio offers UML diagram with ability to specify characteristics of several Microsoft-related programming languages (Visual Basic, C#, Visual C++, IDL). Visio is purely used for drawing diagram, it does not support neither generating source code for specific language nor creating database schema from UML. Diagrams created in Visio are saved into its own file formats or XML format. Microsoft also released XMI Export component for exporting UML diagram to standard XMI file, but only for Visio version 2003. Microsoft branded Visio as a Microsoft Office application, but customers have to buy it separately because it has never been included in any basic Office suite.

ArgoUML\textsuperscript{4} ArgoUML is the open source UML modeling tool based on Java. It supports OCL, standard UML 1.4, code generation for Java, C++, C#, PHP and also has the ability to reverse engineer compiling Java code and generate UML diagrams for it. The greatest advantage of ArgoUML is that users do not have to pay to use the tool since it has been an open source. On the other side, ArgoUML lacks of some features including database schema generation, and nothing ensures that they will be added soon. The development of ArgoUML totally depends on the voluntary developers, thus it may not have a stable deveploment plan.

IBM Rational Software Architect\textsuperscript{5} This is the tool that combines modeling and development in one common approach. The core of Rational Software Architect is based on Eclipse platform, but since IBM develops it as a commercial product, it has more powerful features as model-to-code and code-to-model transformations, predefined templates and design patterns, code review and architectural discovery. Every feature is added to the development environment as a plugin. In the latest version, Rational Software Architect supports transforming UML to not only programming languages (Java, C++, C#) but also other standards (WSDL, XSL, CORBA). Diagrams can be imported from and exported to XMI file. This is the ideal complex tool for modeling and development, but developers must pay to use it.

Visual Paradigm for UML\textsuperscript{6} This tool covers almost every need of software lifecycle. It supports UML 2 and Business Process Modeling Notation (BPMN). Aside from creating UML diagrams, it also has database modeling (ERD diagrams), object-relational mapping (ORM diagrams), code engineering and other useful utilities. It supports importing/exporting XMI and handling even files created in other tools like Rational Rose. Regard to database modeling aspect, this tool supports designing database in more levels (conceptual, logical and physical), generating and executing database schema, generating triggers and stored procedures. Visual Paradigm for UML is released in both commercial and free edition (community edition), but the free edition only supports creating diagrams without generations of code and database tables.

Enterprise Architect\textsuperscript{7} Enterprise Architect is an ultimate tool for full life cycle modeling, visualization and design based on UML 2 standard. Beside the basic functionality of modeling in UML, it supports requirements management, automated document generation, generation and reverse engineering of source code for more than ten programming languages, project management etc. Not being just a pure modeling tool, it can also compile, debug, test and execute deployment scripts. One of the Enterprise Architect features that is interesting in the context of this thesis is the capability of database modeling. It supports mapping from the database concepts of tables and relationships onto the UML

\textsuperscript{4}Further information at http://argouml.tigris.org/
\textsuperscript{5}Further information at http://www-01.ibm.com/software/awdtools/swarchitect/standard/
\textsuperscript{6}Further information at http://www.visual-paradigm.com/product/vpuml/
\textsuperscript{7}Further information at http://www.sparxsystems.com/
concepts of classes and associations as well as generating SQL scripts to eleven SQL dialects. Just like IBM Rational Software Architect, Enterprise Architect is a commercial product.

As we have seen, the number of UML modeling tools is plenty in many forms. They can be commercial or free, independent application or plugin to existing program. However, there are certain insufficiencies in these products. The commercial product requires users to buy, whereas free edition does not provide advanced functionalities. Plugins satisfy the requirement of flexibility, but users must have knowledge about environment, to which plugins are added.

2.5 ER modeling tools

Although the aim of this thesis is to create a UML modeling tool, in this section several ER modeling tools will also be reviewed for the purpose of comprehending more about mapping diagram to database schema.

**ER Modeller**

This tool has been gradually developed at Czech Technical University in Prague through several bachelor and diploma theses. ER Modeller can create diagram in form of Chen notation, Crow’s Foot notation or UML notation. Diagram can be saved to or loaded from XML file. ER Modeller allows user to define custom data types or edit existing types. Program contains basic functions for creating ER diagram including entity, relationship and attribute as well as manipulating the diagram. If user wants to export the current diagram to SQL script, program will check the validity of the diagram first. If everything is valid, the diagram will be exported with the option of sending the result script to database through a predefined database connection. ER Modeller is still under development, thus some necessary features such as undo/redo are not supported at current time.

**ERTOS**

ERTOS is the result of a bachelor thesis at Charles University in Prague. Beside common tasks like creating and editing ER diagram components, ERTOS also supports advanced features such as self-reference and ISA inheritance. Self-reference is the capability of defining a relationship between two entities of a same type. ISA inheritance demonstrates the relationship in which an entity type is the subtype of another entity type. Program supports saving diagram in XML format and validating a diagram before exporting it to SQL script.

**ERGedit**

Program ERGedit is another bachelor thesis at Charles University in Prague. It supports two kinds of diagram, being capable of working with multiple diagrams simultaneously. One of the remarkable features of ERGedit is that it can draw relationship

---


in both normal form and orthogonal form. Its SQL export function also checks the validity of exported diagram and supports exporting to six SQL dialects and standards. Created diagrams could be saved to file in XML format but with custom file extension. Program can cut/copy/paste the components of a diagram but the function of undo/redo is not supported.
3 Requirements

3.1 Requirements specification

An application cannot be complete and meet user’s needs without accurate requirements specification. This phase of development makes clear of what application should be capable of or what functions it should possess. In the case of the program for generating database schema from class diagram, it will have the following requirements:

- Graphical user interface with necessary menus and surface for creating UML class diagram components (classes, connections, attribute and method definitions etc.)
- Support function drag-and-drop
- Add and remove components
- Alter component position on the design surface
- Change the text representation of components
- Save and load diagram to/from XML file
- Export diagram to database script in standard SQL-92
- Export current diagram to image file
- Function Help is available from main menu

3.2 Use cases

For the purpose of better comprehension, requirements specification will be divided into several parts, each part represents a group of functions that application should have. Use case diagrams will be used to visualize the specifications.

3.2.1 Define components

This section deals with adding and removing components on the design surface. Each type of component will be treated differently. The use case diagram is shown in Figure 3.

- Add new class - User can create new class component by either way: drag-and-drop component from menu or select component in menu and then click on the design surface. In both ways a new class with default name will appear on the design surface.
- Remove class - User selects the class to be removed, then selects the deletion option either from particular menu or by pressing specific key.
Figure 3: Defining components

- Add new connection - User selects type of connection from menu, selects the first class and then the second class of the relationship. Based on concrete type, connection will appear with or without connection name, roles and multiplicities.

- Remove connection - User selects connection to be removed, then selects the deletion option either from particular menu or by pressing specific key.

- Add new attribute - User selects the class to which attribute will be added, then selects the option for adding new attribute, fills in attribute properties and finishes the action.

- Remove attribute - User selects attribute to be removed, then selects the deletion option from particular menu.

- Add new method - User selects the class to which method will be added, then selects the option for adding new method, fills in method properties and finishes the action.

- Remove method - User selects method to be removed, then selects the deletion option from particular menu.
3.2.2 Manipulate components

This section concerns with altering components on the design surface, more concretely changing component position and representation. The use case diagram is shown in Figure 4.

- Move component - Users selects the component to be moved and drags it to new position.
- Move class - Specialized case of “Move component”, applied for class component.
- Move connection name label - Specialized case of “Move component”, user can move the name label of the connection.
- Move role name label - Specialized case of “Move component”, user can move the class role label of the connection.
- Move multiplicity label - Specialized case of “Move component”, user can move the class multiplicity label of the connection.
- Change component properties - User selects the component to be changed, then alters component properties in particular menu or dialog and finishes the action.
• Change connection properties - Specialized case of “Change component properties”, user selects the connection and then changes its name, roles or multiplicities.

• Change attribute properties - Specialized case of “Change component properties”, supporting changing attribute properties.

• Change method properties - Specialized case of “Change component properties”, supporting changing method properties.

• Change class name - User can change the class name on the design surface.

3.2.3 XML tasks

This section shows the functions for saving and loading diagram. The use case diagram is in Figure 5.

![Figure 5: XML tasks](image)

• Save to XML - User selects the option to save the current diagram on the design surface. Program exports diagram structure to XML file.

• Load from XML - User selects the option to load diagram to the design surface from XML file.

3.2.4 SQL tasks

The tasks concerning with SQL export are describe in this section. The use case diagram is shown in Figure 6.

• Export to SQL - User selects option to export current diagram on the design surface to SQL script. User fills in necessary information and program carries out the exporting.

• Set database name - User sets up a valid database name.
Figure 6: SQL tasks

- Export methods to procedures - User can select whether to export even class methods.
- Create new database - User decides whether to create new database or use existing database.

3.2.5 Other tasks

All the tasks that do not belong to above sections will be listed here. The use case diagram is shown in Figure 7.

Figure 7: Other tasks

- Export to image - User selects the option to export current diagram to supported image formats.
• Get help document - User can access help document in main menu in case of encountering problems.

3.3 Implementation environment

Application will be coded in programming language C# with the support of Visual Studio 2008. C# is one of the supported languages of Microsoft .NET Framework with the similarities to C++ and Java, but it has some advanced features. Visual Studio 2008 is an IDE for .NET application containing all the essential functionality to create either personal project or large enterprise project. More about C# and Visual Studio 2008 can be found in [8].

3.4 Analysis model

The purpose of analysis is to transform requirements into a high-level architecture of the system that defines what components will be included. It means that analysis just points out abstract models and concepts and ignores the details of the system as well as how to implement these details. Analysis concentrates on the functional requirements, ensuring that they are handled properly. Nonfunctional requirements and implementation environment constrains are left for the later phases. In other words, analysis expresses a nearly ideal picture of the system [11]. Object-oriented model treats all the entities in a specific system as objects that have properties and behaviors. The objects with common characteristics can be described by a class which is usually found out by investigating the requirements specification. From the use cases in the previous chapter there are several candidates that could be classes in the system. The overall situation is expressed in Figure 8.

• The first purpose of the application is to depict the components of a database schema in the form of diagram, so there should be a class Diagram.

• Components could be connections, classifiers, attributes, methods and labels, thus their respective classes are Connection, Classifier, Attribute, Method and Label. A diagram consists of classifiers and connections. A classifier in turn can have its attributes and methods.

• A connection contains its own name, the role names and multiplicities of the classifiers participating in the relationship which that connection represents. The information of connections, classifiers as well as attributes and methods should be shown on the diagram, and it can be done by labels. Label displays the name of the classifier, the detail characteristics of an attribute or a method, or maybe the role names and multiplicities.

• Classifier and Connection should have their specific types. Specialized type of Classifier can be AbstractClass or NormalClass. Connection is the generalized type of BidirectionalAssociation, UnidirectionalAssociation, Aggregation, Composition or Generalization.
In an application that has graphical interface, there will be components interacting with user and components processing logical part behind the scene. This fact recommends that each component of a diagram should have a visual part controlling the tasks with GUI (defining and manipulating component graphically) and a logical part concerning the operations behind (managing component’s structure, working with XML, SQL). The concrete design will be explained in the next chapter.

3.5 States of diagram

A diagram will have some simple states: newly created, changed, saved and loaded from file. The transitions between them are a little more complicated, for example the new diagram might be loaded from file either when the old diagram was saved or not, the application can close when the diagram has arbitrary state. Therefore, the process of handling those tasks should be elaborated clearly. The states of diagram is shown in Figure 9.

Saving and loading diagram belongs to logical part of the application. The XML format is chosen because of its popularity and power of simplicity. The task of exporting diagram to XML can be done by components themselves when each component exports its own part one after another. Importing diagram from XML will require a separate object
to manage and control the process of file parsing. Thus there should be an XmlLoader class performing the mentioned task.
4 Design

Design is the refinement of analysis and concentrates on how the components in the system will be implemented. It adapts the result of analysis to the constraints imposed by nonfunctional requirements (implementation environment etc.). In some cases, the design may be elaborated to detail that the system can be implemented directly through a systematic transformation of the design into code. But in general the design is elaborated only far enough to ensure that the implementation phase can produce a set of components that satisfy the requirements [11].

4.1 Concepts

For the convenience of working with graphical interface, some concepts (shown in Figure 10) will be introduced and explained as follows:

- A connection is created by connecting two classifiers. If there is a connection between two classifiers, the classifier which user selects first to initiate the connection will be called the "source class", and the other classifier which is the destination of the connection will be called the "destination class". A classifier can be "source class" for one connection, but "destination class" for another.

- If a classifier is a "source class" for a connection, it means that connection comes out of the classifier and will be called the "out connection". In the other hand, if a classifier is a "destination class" for a connection, it means that connection comes toward the classifier and will be called the "in connection".

The reason why those concepts are necessary is that the connection needs to distinguish its two involved classifiers for the purpose of correctly applying the role and multiplicity for each, and the classifier needs to separate its connections into two categories to avoid the duplication when exporting to XML or SQL. The connection in Figure 10 is unidirectional, but in general the connection could be any kind (bidirectional association, unidirectional association, aggregation, composition, generalization).

4.2 Design classes

As discussed in previous chapter, the diagram consists of classifiers and connections that should be separated into visual part and logical part. The logical part controls and processes the operations in logical layer, while the visual part takes care of the works in the graphical interface. The following sections will describes concretely the structure of those parts.

4.2.1 Classifier

- The Classifier will be replaced by two components: BaseClassType and ClassPanel. The design of these classes is shown in Figure 11.
Figure 10: Classifier and connection concepts

- BaseClassType represents the logical part of a classifier. It will contain a collection of attributes, a collection of methods, a collection of “out connections”, a collection of “in connections”, class visibility, class stereotype, an indication of whether the class is abstract and the reference to the visual part. BaseClassType will have the methods for working with those collections (adding and removing), methods for exporting to XML and SQL. BaseClassType acts as a general class, concrete types of classifier will inherit from this class.

- ClassPanel represents the visual part of a classifier. Its task is to show the information of class on the design surface, and therefore it will consist of a class name label, a collection of attribute label, a collection of method label and the reference to the logical part.

- The concrete types of the classifier are NormalClass or AbstractClass. These classes are distinguished by the way they are displayed on the design surface and the way they are exported to file.

4.2.2 Connection

- The Connection is separated into BaseConnection and VisualConnection. The design is shown in Figure 12.

- BaseConnection represents the logical part containing the references to the source class and the destination class, roles and multiplicities of those classes, an indication of whether it is recursive connection, connection name and the reference to the visual part. Logical part takes care of exporting component to XML and SQL, so BaseConnection will have methods ExportToXML and ExportToSQL. BaseConnection is the general class for subclass to inherit.
BidirectionalAssociation, UnidirectionalAssociation, Aggregation, Composition and Generalization are the subclasses of BaseConnection. Each of them has a different way to export its own data to file.

VisualConnection represents the visual part and contains the labels of connection name, role names and multiplicities. It also has the reference to the logical part. Visual part is responsible for displaying connection and its information on the design surface, hence it will have methods for drawing connection and displaying labels.

The special case of connection is recursive connection, in which the source class and destination class are identical. A class VisualRecursiveConnection is created for this case. It inherits from VisualConnection.

4.2.3 Label

Label is the medium to display component’s information on the design surface. Since C# has been chosen as the programming language for implementation, the need of creating a new class Label is no longer necessary. C# has its own built-in components for working with graphical objects including label. The existing class Label will reduce a large amount of work because a new class Label does not have to be implemented from scratch, instead subclasses that inherit from existing class Label will be specified to be suitable for concrete goals (displaying information of attributes and methods, connection name etc.).
Figure 12: Logical and visual parts of connection
4.2.4 Attribute and method

- Just like other components, an Attribute will have logical and visual parts. The logical part is represented by a new class named AttributeElement, while the visual part is represented by class AttributeLabel derived from class Label of C# class library.

- AttributeElement holds the attribute name, type, visibility, default value, stereotype and the reference to the visual part. Moreover, because the aim of the application is to create database schema, AttributeElement will have some more specific properties: an indication of whether it is primary key, an indication of whether it can be null, data length and decimal length. The logical part is responsible for working with XML, SQL and updating its visual part, so it will have corresponding methods.

- AttributeLabel has the characteristics of existing class Label, thus it will have just one more thing: the reference to the logical part.

- Similarly to Attribute, a Method is separated into MethodElement and MethodLabel.

- MethodElement holds the method name, stereotype, visibility, return type, collection of parameters and the reference to MethodLabel. It also has the methods for working with XML, SQL and updating its label.

- MethodLabel is derived from Label and has the reference to MethodElement.

- The design of those class is shown in Figure 13.

4.2.5 Diagram

The class Diagram belongs to both logical layer and visual layer of the application. It manages and controls all the classifiers and connections, taking care of overall exporting and loading data while still concerning with displaying components. Therefore, a Diagram object will contain a collection of BaseClassType objects, a collection of BaseConnection objects, an indication whether it was changed and the path in the file system of the XML file to which it will be saved. It will have ability to add and remove items from its collections, save and load data from XML file and export data to SQL script.

With all the complex operations that a Diagram object must manage, it is necessary to have a static class DiagramState to hold the state of the diagram at a specific moment. The data that DiagramState holds will be the reference to the current diagram, the selected ClassPanel or VisualConnection, what type of classifier or connection is selected, the number of classifiers on the design surface and some other data. DiagramState will be the one that mark current diagram as changed if any change is made. Figure 14 shows the more specific design.
Figure 13: Design of attribute and method

Figure 14: Diagram and DiagramState
The result of class design is shown in Figure 15, which depicts the overall structure of the application.

![Diagram](image)

**Figure 15: General view of design classes**

### 4.3 Visualization

#### 4.3.1 Initialize and manipulate class

According to requirements, a new class can be created by either drag-and-drop method or clicking on the design surface when class creating option is selected. Conveniently, C# has built-in functions to support these kinds of actions, so the detail implementation will be explained in the next chapters.

Moving class around on the design surface requires the application to track the position of the class. The function of tracking should be accessed globally because the solution
that every class has its own tracking is not optimal. For that reason a static method GetMouseRelativeLocation will be added to DiagramState. It will be used to track not only the position of class but also the position of component in general.

4.3.2 Initialize and manipulate connection

Displaying the connections on the design surface is an important part of graphical aspect. The visual representation of a connection must express clearly the relationship between two classes and also satisfy the aesthetic criteria. In the case of class diagram, the most appropriate way to display a connection is to draw it in the orthogonal form. The following section will explain the method and algorithm for that kind of drawing.

The most necessary information is the relative positions of two involved classes of the connection. Figure 16 presents all the positions that the source class can have.

Figure 16: Relative position of source class

- If the destination class locates on the section 1, the source class will have the relative position "LowRight" (in regard to the destination class).
- If the destination class locates on the section 2, the source class will have the relative position "LowLeft".
- If the destination class locates on the section 3, the source class will have the relative position "HighRight".
- If the destination class locates on the section 4, the source class will have the relative position "HighLeft".

The more concrete position is calculated by examining the angle created by the line connecting two centers of the classes and the horizontal axis. Figure 17 explains the idea. The alpha angle could have value from 0 to 90 degrees with the critical point at 45
degrees. The form of connection when alpha is less than 45 degrees will be different from the form when alpha is greater than 45 degrees. The difference will be explained shortly next. In order to find alpha, the tangent of alpha will be calculated firstly based on two side edges (with the lengths $a$ and $b$ in Figure 17) of the right triangle created by alpha. A class SimpleLine representing those edges will be created to support the calculating.

![Figure 17: The alpha angle](image)

There is no formula or mathematical definition for a drawing of a diagram. It can be defined informally that a drawing is more aesthetic than another one if it is “nicer” or “more readable” [12]. Some considered criteria are orthogonality and the bends of a connection. It means that the connections should have the orthogonal form and its number of bends should be minimal. In the case of this application, a connection will have one of two orthogonal forms and maximally two bends. Two orthogonal forms are horizontal form (a) and vertical form (b) as seen in Figure 18.

![Figure 18: Orthogonal forms](image)
The next problem is to find the point of the class which the connection should attach to. This is where the relative position of the classes and the alpha angle come in handy. Four edges of a class will be divided into several parts and the design surface will also be divided into several sections. With every combination of the relative position and the alpha angle the connection will have a particular form. Figure 19 shows the sections after the division.

![Figure 19: Class sections](image)

Every class is marked with eight points from A to F. A, B, C and D represent the vertices while E, F, G and H represent the central points of the edges. Eight points divide fours edges into eight parts. A connection will start from a point in one part of the source class and end at a point in another part of the destination class. These two points are called intersection-with-source point and intersection-with-destination point. The bends of a connection, if there are, will occur between those two points. Table 2 shows the concrete situations that can happen.

The drawing of a connection will be represented by a class Polyline which contains a point array of the connection. To guarantee the orthogonal form of the connection, the points in the array will satisfy one condition: two adjoining points will have the same either horizontal or vertical coordinate. The algorithm for drawing connection can be presented as follows:

1. Determine the relative position between two classes.
2. Calculate the alpha angle.
Table 2: Relative positions and orthogonal forms

<table>
<thead>
<tr>
<th>Section</th>
<th>Source class’ position</th>
<th>Source class’ part</th>
<th>Destination class’ part</th>
<th>Orthogonal form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LowRight</td>
<td>$0^\circ - 45^\circ$</td>
<td>AH AE</td>
<td>Horizontal</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$45^\circ - 90^\circ$</td>
<td>CF CG</td>
<td>Vertical</td>
</tr>
<tr>
<td>3</td>
<td>LowLeft</td>
<td>$45^\circ - 90^\circ$</td>
<td>BE BF</td>
<td>Vertical</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>$0^\circ - 45^\circ$</td>
<td>DG DH</td>
<td>Horizontal</td>
</tr>
<tr>
<td>5</td>
<td>HighLeft</td>
<td>$0^\circ - 45^\circ$</td>
<td>CF CG</td>
<td>Vertical</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>$45^\circ - 90^\circ$</td>
<td>AH AE</td>
<td>Horizontal</td>
</tr>
<tr>
<td>7</td>
<td>HighRight</td>
<td>$45^\circ - 90^\circ$</td>
<td>DG DH</td>
<td>Vertical</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>$0^\circ - 45^\circ$</td>
<td>BE BF</td>
<td>Horizontal</td>
</tr>
</tbody>
</table>

3. Determine the orthogonal form according to Table 2.
5. Calculate the coordinates of the points in the Polyline object’s point array.
6. Draw the connection.

In order to exactly determine the intersection-with-source and intersection-with-destination points, a formula for calculating the coordinates of these points will be presented. The formula has two forms corresponding to two forms of connection. The idea is to calculate the distance of the intersection point from the center of the connected edge of the class.

- If the connection is horizontal, the distance is calculated by formula $d = \frac{\alpha}{45} \cdot l \cdot \Theta$
- If the connection is vertical, the distance is calculated by formula $d = \frac{90 - \alpha}{45} \cdot l \cdot \Theta$

In above formulas, $d$ is the distance, $\alpha$ is the mentioned angle, $l$ is the half of the length of the connected edge and $\Theta$ is coefficient for adjusting the distance ($0 \leq \Theta \leq 1$). If $\Theta$ equals 0, all the connections connecting to the same edge will have the same intersection point at the center of that edge. The greater $\Theta$ is, the further the distance is. When $\Theta$ equals 1, the intersection point will be the vertex of the rectangle representing the class. Figure 20 shows two particular cases of two connection forms: horizontal form (a) and vertical form (b).

**Recursive association** The above method and algorithm are applied for the case in which source class and destination class are separate. When the source class is also the destination class, the connection becomes recursive association\(^{11}\). The task of drawing recursive association is more trivial, because the application does not have to do the complex calculating as presented. The only necessary information is position of the class.

---

\(^{11}\)recursive generalization cannot exist, hence when a connection is recursive, it is an association.
Based on the position, the coordinates of the points in the point array of Polyline object will be calculated. The result is a connection in the form of a rectangle as seen in Figure 21.

![Figure 21: Recursive association](image)

### 4.4 Working with SQL

The SQL export task can be separated into several steps based on the structure of the classes. At the beginning, the application instructs the current diagram to export to SQL. The Diagram object, which contains the BaseClassType collection and the BaseConnection collection, will create a writer for writing to file and invoke the SQL export functions of the elements in its collections. Firstly, the export will be invoked in the BaseClassType collection. Each BaseClassType object has an AttributeElement collection and a MethodElement collection. For the reason that methods should be written at the end of the SQL file, at this moment the class just exports its AttributeElement collection. The BaseClassType object takes the writer passed from the Diagram object and passes it again to its AttributeElement collection. AttributeElement object then exports its content to file. It is the lowest level in the class structure, so the passing of the writer ends here. The steps
repeat with the BaseConnection collection and finally the MethodElement collection of each BaseClassType object. The overall process is shown in Figure 22.

![Diagram](image.png)

Figure 22: The process of exporting to SQL

4.5 Working with XML

4.5.1 Saving diagram to XML

A diagram can be saved to XML similarly to the way it is exported to SQL. Every component in the architecture has its own method to export. The diagram will manage and control the overall process which invokes the exporting firstly in the BaseClassType collection and then the BaseConnection collection. The exporting of AttributeElement and MethodElement collections are invoked within the exporting of their container, a BaseClassType object. The process is shown in Figure 23. The XML schema of saved file can be found in Appendix C.
Figure 23: The process of exporting to XML
4.5.2 Loading from XML

The loading is realized by the class DiagramXmlLoader which has the task of taking the information from XML file and recreating the diagram structure. At the beginning, the program will create a new Diagram object and request it to load data from specific file. Using the methods of DiagramXmlLoader, the diagram loads its class collection and connection collection. The process of loading is carried out iteratively, at first creating new ClassPanel object and initializing the AttributeElement and MethodElement collections. The reason that ClassPanel object is created instead of BaseClassType object is that the XML file contains the data of size and coordinates of the representation of class on the design surface, while BaseClassType (as logical part) knows nothing about those data. The reason for creating VisualConnection object instead of BaseConnection object is similar. The common steps for loading any component are presented as follows:

- Load data from XML file
- Analyzing data
- Create corresponding object based on analyzed data
- Add object to diagram structure

The process of loading from XML file is shown in Figure 24.
Figure 24: The process of loading from XML
5 Mapping class diagram to relational data model

The need of mapping rises from the fact that object-oriented paradigm and relational data model are incompatible and have various differences. The main aim of this thesis is to study the generating of the relational database schema, which applies relational data model, from UML class diagram, which represents object-oriented paradigm. Thus studying the mapping has become an essential part of this work. The following sections present the approach to solve the mapping problems. Some important ideas are taken from [3, 7].

5.1 Basic mapping

- Class is mapped to table, as class name will be table name.
- Class attribute becomes database attribute, as attribute name will be column name. Some attributes of the class could be marked (by stereotype) as “primary key”. They will become the primary keys of the database table when the class is mapped. From now on such attributes will be referred to as “the primary keys of the class”.
- Connection is mapped to constraint, junction table or database view depending on its type and multiplicities.

5.2 Data type mapping

The SQL-92 standard supports the following data types: char, varchar, bit, bit varying, smallint, int, numeric, decimal, real, float, double, date, time, timestamp, interval [2, 10]. Some types have customizable data length, the others have fixed length. Some numeric types can also define the decimal length. Table 3 lists all the supported data types and their possibility of customizing data length.

When a new AttributeElement object is created in the class diagram, user can define its type as one of those types mentioned above. Besides, user also has the option to define the type of AttributeElement as programming language type. Such types will be mapped to database types at the SQL export phase with the default data length. The supported types are string, int, float, double, boolean. Table 4 shows the mapping programming language types to database types.

5.3 Connection mapping

When mapped to the relational database, connections can be divided into three groups. The first group contains bidirectional association which will be mapped to junction table. The second group consists of unidirectional association, aggregation and composition, which will become junction table or constraint depending on the multiplicities. The last group contains generalization which will be mapped to database view. In a relationship,
<table>
<thead>
<tr>
<th>Data type</th>
<th>Customizable data length</th>
<th>Customizable decimal length</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>varchar</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>bit</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>bit varying</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>smallint</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>int</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>numeric</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>decimal</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>real</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>float</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>double</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>date</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>time</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>timestamp</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>interval</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3: SQL-92 data types

<table>
<thead>
<tr>
<th>Programming language type</th>
<th>Database type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>varchar</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>boolean</td>
<td>bit</td>
</tr>
</tbody>
</table>

Table 4: Mapping programming language types
Relationship | Primary key decision for new table
---|---
M:N | Each foreign key is also a primary key
1:1 | Each foreign key is also a primary key
0..1:0..1 | Each foreign key is also a primary key
1:N | Primary key is the foreign key from the destination class table
0..1:N | Primary key is the foreign key from the destination class table
N:1 | Primary key is the foreign key from the source class table
N:0..1 | Primary key is the foreign key from the source class table
0..1:1 | Primary key is the foreign key from the source class table, the foreign key from the destination class table is unique
1:0..1 | Primary key is the foreign key from the destination class table, the foreign key from the source class table is unique

Table 5: Primary key decision of bidirectional association mapping

the multiplicity of the source class will come first, followed by the destination class’ multiplicity. For example, if the relationship is M:N, M is the multiplicity of the source class and N is the destination class’ one.

- A bidirectional association is mapped to a junction table that can be used to traverse the association in both directions. The table will take the primary keys from both involved tables as its foreign keys. The primary key of the table is decided depending on the multiplicities of the connection. Based on the type of the relationship, all the foreign keys or only some of them will become the primary keys. Table 5 presents all the cases of primary key decision of bidirectional association.

- The second group has more complex mapping. The relationship decides not only the primary keys but also the strategy of mapping. If the multiplicity of the destination class is maximally one, then it is not necessary to create junction table, instead a foreign key will be added to the source class table. Otherwise the table will be created with foreign keys from both classes. The relational database knows nothing about the concept of oriented relationship, so it will treat cardinalities 1:N and N:1 the same and always add the foreign key to the table at N side. Because UML has strict rules about the orientation of an association, cardinalities 1:N and N:1 need to be clearly distinguished, thus the mapping of an oriented association can result in two different cases. Table 6 presents the full mapping for the second group.

- Recursive association could be any kind of connection except generalization and mapped to database in the same manner as any other association. The main problem with recursive association is that the source class refers to itself, hence it is also the destination class. In the case of creating junction table, the foreign keys will be taken from a class twice. In the case of creating constraint, the primary key of the class will be added back to it as the foreign key. In both cases the duplication will happen. The solution for this problem is to rename the foreign key when adding it the second time.
<table>
<thead>
<tr>
<th>Relationship</th>
<th>Unidirectional association</th>
<th>Aggregation</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>M:N</td>
<td>Not exist</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:N</td>
<td>Create new table, foreign keys are the primary keys from both class. Each foreign key from the destination class table is the primary key of the new table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..1:N</td>
<td>Not exist</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:1</td>
<td>Each primary key from the destination class table is a foreign key in the source class table. Such foreign key is not null and unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:0..1</td>
<td>Each primary key from the destination class table is a foreign key in the source class table. Such foreign key is unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N:1</td>
<td>Each primary key from the destination class table is a foreign key in the source class table. Such foreign key is not null and unique</td>
<td></td>
<td>Not exist</td>
</tr>
<tr>
<td>0..1:1</td>
<td>Not exist</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N:0..1</td>
<td>Each primary key from the destination class table is a foreign key in the source class table. Such foreign key is unique</td>
<td></td>
<td>Not exist</td>
</tr>
<tr>
<td>0..1:0..1</td>
<td>Not exist</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Mapping of aggregation, composition and unidirectional association

- Generalization is a special connection because it does not contain any multiplicity. A generalization expresses the inheritance relationship in which the subclass possesses the properties of the super class. As a result, a generalization will not be mapped to a table or a constraint but a database view. This view is created by joining two tables mapped from corresponding classes, thus it will have all the attributes from two classes. Details of inheritance mapping are explained below.

### 5.4 Inheritance mapping

Inheritance is one of the typical features of the object-oriented data model but it does not exist at all in the relational data model. This fact causes a lot of difficulties when mapping object-oriented data model to relational data model. Because studying all the cases of the inheritance mapping problem will need an entire separate work, this thesis will only concern with mapping single inheritance and one-level hierarchy defined as follows:

- In single inheritance a class can only have one super class. That means it cannot inherit from more than one class.

- One-level hierarchy is the concept that if a class is the subclass, it cannot have the subclass itself, and if a class is the super class, it cannot inherit from another class. In other words, the hierarchy of the inheritance cannot have more than one level.

In order to adapt to the situation, a new attribute is added to the class BaseClassType. This attribute indicates whether the object has a super class object or not. When a gen-
eralization is established between two BaseClassType objects, the source class becomes the subclass and has a reference to the destination class as the super class. If the generalization is removed, the reference will be removed as well. The strategy for mapping inheritance in this thesis is to create separate tables for the subclass and the super class. The primary key of the super class will become the primary key of the subclass. If the subclass has an existing primary key, it will become normal attribute. When a generalization is created, some conditions will be checked to ensure that single inheritance and one-level hierarchy are guaranteed. The generalization will not be created in the following situations:

- There is a generalization in the outConnection collection of the source class, so it has had a super class already (violating the single inheritance condition).

- There is a generalization in the inConnection collection of the source class, so it has had a subclass already (violating the one-level hierarchy condition).

- There is a generalization in the outConnection collection of the destination class, so it has had a super class already (violating the one-level hierarchy condition).

When mapping to the database, the super class is the only class containing primary key. The generalization will be mapped to a view named after the subclass name plus some special characters for the purpose of distinguishing. If the subclass has any association, the primary key taken in the process of mapping the association will be the primary key of the super class.

5.5 Method mapping

Because standard SQL-92 does not contain a definition of stored procedure or function, method will be exported based on one of three dialects: Microsoft SQL Server, MySQL and Oracle. If a method returns a value, it will be mapped to stored function, otherwise stored procedure will be applied. Mapping only creates the declaration of procedure or function, the body will be left empty for user to fill in. The parameter type and return type are mapped based on the description in section 5.2. Although the compared results of mapping to each dialect are not so much different, each dialect has its own characteristics. For example, Microsoft SQL Server adds character “@” before parameter name or Oracle uses the command CREATE OR REPLACE PROCEDURE instead of CREATE PROCEDURE like the other two dialects. The details of syntaxes for creating procedure and function can be found in [15, 16] (for Microsoft SQL Server), [17, 18] (for Oracle) and [19] (for MySQL).
6 Implementation

This section presents the phase when all the designs above are implemented in particular programming language and development environment selected in the requirements. Some changes can be made to adapt the design to the environment.

6.1 Enumerations and constants

In order to keep the control and convenience of coding, it is necessary to have constants and enumerations. Constant helps developer saving the effort when changing the same value in different places by redefining the constant which those places refer to instead of manually changing value at each place. Enumeration gives a list of properties a particular meaning, so developer does not have to use the obscure integers to define the property, such as the state or the type of an object. The following sections present the enumerations and sets of constants that are used in the application.

6.1.1 Enumerations

All the enumerations are placed in GlobalEnums.cs file.

- ClassType: indicates the type of a class. Its values can be Base (not initiated class), Normal or Abstract.
- ConnectionType: type of a connection. It can be BidirectionalAssoc, UnidirectionalAssoc, Aggregation, Composition or Generalization.
- Visibility: used by BaseClassType, AttributeElement and MethodElement to specify their visibility. Its values: None, Public, Protected and Private.
- ClassPosition: important factor in the process of drawing connection. Its values: HighLeft, HighRight, LowLeft and LowRight.
- AttributeDataType: used exclusively by AttributeElement when defining or changing an attribute. It has three values: NullType (not initiated attribute), GeneralType and DatabaseType.
- SQLdialect: specifies the type of SQL dialect used when exporting methods to SQL. It can be None (methods are not exported), MSSQL (Microsoft SQL Server), Oracle and MySQL.

6.1.2 Constants

The constants are put in class DiagramConstants in the file with same name. There are two groups of constants, one defines the properties of class and connection, another defines data types. Because the number of constants needed in the application is big, all the constants will not be listed here.
• Constants for class and connection: used mainly when working with XML. These constants will define the tags of the XML file.

• Constant for data type: used in both working with XML and exporting to SQL. These constants define the standard type name and the length of some data types.

6.2 DiagramState structure

Class DiagramState is marked as static, so a DiagramState object will never be created. Instead, the static members of the class will keep the description of the state of a diagram at a specific moment. Besides, the class also provides general methods for working with diagram such as MarkDiagramAsChanged or GetMouseRelativeLocation. Several typical properties are:

• SelectedClassPanel: refers to the visual part of the current selected class.

• SelectedConnection: refers to the visual part of the current selected connection.

• SelectedLabel: the current selected label of an attribute or a method.

• TypeOfClass: type of selected class.

• TypeOfConn: type of selected connection.

• CurrentDiagram: refers to the current Diagram object.

Restoring diagram state  Undo/redo are the common tasks that an application should have. The mechanism of undoing or redoing an action of the application in this work is based on saving the state of the diagram to a temporary file. A state is actually an XML file containing the complete structure of the diagram at a moment. This method improves the simplicity of the undo/redo tasks because application does not have to analyse detailly which component has been changed and record that specific change. The list of states is kept in two variables undoList and redoList in class DiagramState. Each state has a unique name that is also the name of the temporary file. Listings 1 and 2 show the way a state could be passed between undoList and redoList.

```java
String lastAction = DiagramState.UndoList.Last();
DiagramState.UndoList.Remove(lastAction);
DiagramState.RedoList.Add(lastAction);
String previousAction = DiagramState.UndoList.Last();
...
diagram.LoadFromXML(DiagramState.DiagramTempFolderPath + "," + previousAction);

Listing 1: Undo
```

```java
String nextAction = DiagramState.RedoList.Last();
DiagramState.RedoList.Remove(nextAction);
DiagramState.UndoList.Add(nextAction);
...
```

Listing 2: Redo
State change If there is some kind of action occurring on the design surface (adding attribute, changing connection, moving class etc.), the diagram will be marked as changed, so the application will know whether it needs to be saved later. When a change occurs, the current state of the diagram will be saved to undoList and then redoList will be cleared. Listing 3 presents the method MarkDiagramAsChanged.

```csharp
if (!CurrentDiagram.IsChanged)
{
    CurrentDiagram.IsChanged = true;
    mainForm.Text += "∗";
}
AddToUndoList();
ClearRedoList();
```

Listing 3: Mark diagram as changed

### 6.3 Class representation

If the logical part of a class is a BaseClassType object, the visual part is represented by a ClassPanel object. Class ClassPanel is actually a control inheriting from TableLayoutPanel. This approach brings some advantages, for example the layout of the panel will be automatically divided into rows when components are added. In the case of ClassPanel, its layout contains three separate parts for class name, attribute list and method list.

In order to conveniently work with attributes and methods on the design surface, specialized types of label should be used. The built-in class Label of C# has almost everything that a label needs, so the derived types of Label do not have to specialize a lot. An AttributeLabel object will be used to display attribute in the layout of ClassPanel. Adhering to the idea that every component should have logical and visual part separately, AttributeLabel object itself represents the visual part of a label and has a reference to an AttributeElement object as the logical part. AttributeElement object also contains an inverse reference. The similar relation applies for MethodLabel and MethodElement, the components that represent method.

Section 4.3.2 discusses about eight points constituted of four vertices and four centers of the edges. They divide four edges of a class representation into eight parts that are used to determine the form of the connections connecting to the class. In class ClassPanel, they are represented by eight objects of type Point: upLeft, upMid, upRight, leftMid, rightMid, downLeft, downMid, downRight. A method recalculateBorderPoints will recalculate the coordinates of those points every time the class changes its location.
The most important events of a class representation are MouseDown, MouseMove and MouseUp. They detect the movement of the class and deal with creating connection. When a mouse action occurs on the ClassPanel, there are two situations that can happen. The first one is that user simply clicks on a class and moves it to a new location. Application will not move the class right away, but display a shadow of the class moving with the mouse. The second one is when user decides to create a new connection. Application has to create a new VisualConnection object and draws its representation as user drags mouse from source class to destination class. Listings 4, 5 and 6 show the core of each mouse event handler.

```csharp
if (DiagramState.IsConnLabelSelected)
{
    VisualConnection conn = new VisualConnection(DiagramState.TypeOfConn);
    conn.VisualLine.StartPoint = this.GetCentralPoint();
    conn.LogicComponent.SourceClass = this.LogicComponent;
    DiagramState.SelectedConnection = conn;
} else
{
    DiagramState.ShadowOfMovingControl = new Rectangle(this.Location, this.Size);
}
```

**Listing 4: MouseDown event handler**

```csharp
if (DiagramState.IsConnLabelSelected)
{
    DiagramState.SelectedConnection.VisualLine.EndPoint = DiagramState.GetMouseRelativeLocation();
} else
{
    DiagramState.ShadowOfMovingControl = new Rectangle(DiagramState.GetControlRelativeLocation(), this.Size);
}
```

**Listing 5: MouseMove event handler**

```csharp
if (DiagramState.IsConnLabelSelected)
{
    DiagramState.SelectedConnection.VisualLine.EndPoint = DiagramState.GetMouseRelativeLocation();
    ClassPanel targetClassPanel = DiagramState.CanvasControl.GetChildAtPoint(DiagramState.GetMouseRelativeLocation()) as ClassPanel;
    if (targetClassPanel != null)
    {
        if (!targetClassPanel.Equals(this)) // normal connection
        {
            ...
            createConnection(targetClassPanel);
        }
        else // recursive connection
        {
            ...
            createConnection(targetClassPanel);
        }
    }
}
```

**Listing 6: MouseUp event handler**
\begin{verbatim}
if (DiagramState.TypeOfConn != ConnectionType.Generalization)
{
    createRecursiveConnection(targetClassPanel);
}
else // class moved
{
    DiagramState.SelectedClassPanel.Location = DiagramState.GetControlRelativeLocation();
    recalculateBorderPoints();
}
\end{verbatim}

Listing 6: MouseUp event handler

6.4 Connection representation

If class BaseConnection is depicted concretely in design section, this section will concentrate on the visual part of a connection - class VisualConnection, because it concerns with more practical problems. As mentioned in 4.3.2, the form of a connection is determined partially by the angle created by the line connecting two centers of the involved classes and horizontal axis. To find out that angle, its tangent is firstly calculated. Hence three lines, as seen in Figure 17, are needed for the calculation. They are represented by three objects visualLine, projectionLine and perpendicularLine of type SimpleLine. Also the value 0.8 is set to the parameter $\Theta$. The other part for determining the form of the connection is the position of the source class. For this purpose, four properties of type Point are included in class VisualConnection: highPoint, lowPoint, leftPoint and rightPoint. The central points of two involved classes will be compared to find out those four points. This comparison is realized every time a class changes its location or its size. One point is considered “low point” if it has greater horizontal coordinate, otherwise it is “high point”. Similarly, one point is “right point” if it has greater vertical coordinate. Table 7 shows all the positions that the source class can have based on four points above. The concrete implementation is in method getSourceClassRelativePosition. Listing 7 shows the contents of method calculatePolylinePoints that implements the design in Table 2.

```java
getSourceClassRelativePosition();
switch (sourceClassPosition)
{
    case ClassPosition.HighLeft:
        if (alphaAngle < 45)
        {
            // initialize horizontal form CF - AH
        }
        else
        {
            // initialize vertical form CG - AE
        }
        break;
```
Just like AttributeLabel and MethodLabel at class representation, a VisualConnection object uses DiagramLabel to display its name, roles and multiplicities. DiagramLabel also inherits from built-in class Label of C#. In class VisualConnection the following properties are of type DiagramLabel: connLabel (name of connection), sourceLabel (source class role), destLabel (destination class role), srcMultipLabel (multiplicity of source class) and destMultipLabel (multiplicity of destination class).
In the cases of aggregation, composition, generalization and unidirectional association, application has to draw an additional visual part at the end of the connection as seen in Figure 25. Four methods drawAggregationDiamond, drawCompositionDiamond, drawGeneralizationArrow, drawUnidirectionalArrow can be called in method DrawConnection to deal with these additional parts.

![Figure 25: Additional parts of connections]

### 6.5 SQL export implementation

The process of exporting diagram to SQL runs step by step, beginning with instruction of creating new database if requested. The lists classCollection and connCollection are examined to invoke the SQL export function of every element in each list. If user decides to export the methods (user selects some concrete SQL dialect), all the MethodElement objects of the diagram are exported as well. Listing 8 shows the detail of method ExportToSQL of class Diagram.

```csharp
writer = new StreamWriter(directory + "/" + filename);
if (createNewDb)
{
    writer.WriteLine("CREATE DATABASE " + dbname + ";");
    writer.WriteLine();
}
writer.WriteLine("USE " + dbname + ";");
foreach (BaseClassType clazz in classCollection)
{
    writer.WriteLine();
    clazz.ExportToSQL(writer);
}
foreach (BaseConnection conn in connCollection)
{
    writer.WriteLine();
    conn.ExportToSQL(writer);
}
if (dialect != SQLdialect.None)
{
    writer.WriteLine();
    foreach (BaseClassType clazz in classCollection)
    {
        foreach (MethodElement method in clazz.Methods)
Each BaseClassType or BaseConnection object has its own way of exporting based on its type. BaseClassType and BaseConnection have methods with the same name ExportToSQL, of course with different implementations that take care of exporting. Another significant method in class BaseClassType is GetPrimaryKey. This method returns the list of marked-as-primary-key attributes of the object. If the object represents a subclass, that list will be taken from its super class. The variable setAsChild indicates whether the list of primary key is used by a recursive association, therefore appending some kind of discrimination to the end of the name of attribute if needed to avoid the duplication when adding foreign key to table. Listing 9 shows the concrete implementation of the method.

Listing 8: SQL export at diagram level

Listing 9: Get primary keys

6.6 Handling the XML tasks

There are many different ways to work with XML in C#. In this thesis, built-in class XmlDocument of C# is chosen to perform the tasks of saving and loading XML. Some other useful classes that mainly come from namespace System.Xml are also employed to assist the work. An XmlDocument object loads an XML file and offers various tasks that can be done with it. The most frequently used method of XmlDocument is CreateElement which creates new XML tag. A complete tutorial of how to use XmlDocument can be found in [13].
6.6.1 Saving XML file

Listing 10 shows the body of method ExportToXML of class Diagram. The method Load and Save of XmlDocument object opens respectively closes the XML file. When the file has been initiated, it creates the root tag of XML document (the “classdiagram” tag). Then XmlDocument object and the root tag are passed to lower levels of diagram structure to gradually append new tags to document. Listing 11 shows the example code of method ExportToXML of class BidirectionalAssociation. XmlDocument object creates new XElement objects representing the XML tags, then those newly created objects set XML attributes for the tags and finally append one after another to the “parent” (higher level tag).

```csharp
XmlDocument document = new XmlDocument();
using (XmlTextWriter writer = new XmlTextWriter(filename, Encoding.UTF8))
{
    writer.Formatting = Formatting.Indented;
    writer.WriteProcessingInstruction("xml", "version='1.0'", "encoding='utf-8'");
    writer.WriteStartElement(DiagramConstant.CLASSDIAGRAM);
}
document.Load(filename);
foreach (BaseClassType cl in classCollection)
{
    cl.ExportToXML(document, document[DiagramConstant.CLASSDIAGRAM]);
}
foreach (BaseConnection conn in connCollection)
{
    conn.ExportToXML(document, document[DiagramConstant.CLASSDIAGRAM]);
}
document.Save(filename);
```

Listing 10: XML export at diagram level

```csharp
XmlElement conn = doc.CreateElement(DiagramConstant.CONNECTION);
conn.SetAttribute(DiagramConstant.CONNLABEL, connName);
conn.SetAttribute(DiagramConstant.CONNTYPE, DiagramConstant.BIDIRECTION);
conn.SetAttribute(DiagramConstant.RECURSIVE, isRecursive.ToString().ToLower());
XmlElement source = doc.CreateElement(DiagramConstant.SOURCE);
source.SetAttribute(DiagramConstant.ROLE, sourceClassRole);
source.SetAttribute(DiagramConstant.MULTIPLICITY, sourceClassMultiplicity);
...
conn.AppendChild(source);
conn.AppendChild(dest);
parent.AppendChild(conn);
```

Listing 11: XML export at connection level

6.6.2 Loading XML file

The task of loading is realized by several components such as XmlDocument, XmlNodeList and XmlAttribute. Firstly, method LoadFromXML of class Diagram loads XML
file by XmlDocument object, then two XmlNodeList objects are used to retrieve the lists of “class” tags and “connection” tags. The coding is shown in Listing 12.

```csharp
XmlDocument document = new XmlDocument();
using (XmlTextReader reader = new XmlTextReader(filename))
{
    document.Load(reader);
}
// class nodes
XmlNodeList classNodes = document.GetElementsByTagName(DiagramConstant.CLASS);
DiagramXmlLoader.LoadClasses(classNodes);
// connection nodes
XmlNodeList connNodes = document.GetElementsByTagName(DiagramConstant.CONNECTION);
DiagramXmlLoader.LoadConnections(connNodes);
```

Listing 12: Load from XML

In the second phase, static methods of class DiagramXmlLoader will do the loading, creating new objects and building the diagram structure. Listing 13 shows the example code of method LoadClasses. It takes the XmlNodeList object that contains the list of “class” tags as parameter, and then it creates another XmlNodeList object to get the lower level tag list (“attribute” and “method” tags). After getting all the XML attributes of the tag, it passes the loading to the lists of attributes and methods. Finally, the class representation (ClassPanel object) is added to design surface. The loading of other components is similar.

```csharp
foreach (XmlNode classNode in classNodes)
{
    XmlNodeList children = classNode.ChildNodes;
    XmlAttributeCollection classXmlAttrs = classNode.Attributes;
    ClassPanel newClass = new ClassPanel();
    ...
    foreach (XmlAttribute xmlattr in classXmlAttrs)
    {
        if (xmlattr.Name == DiagramConstant.CLASSNAME)
        {
            className = xmlattr.Value;
        }
        ...
    }
    newClass.Initialize (className, type, location, width, height);
    foreach (XmlNode child in children)
    {
        if (child.Name == DiagramConstant.ATTRIBUTE)
        {
            loadAttribute (child, newClass.LogicComponent);
        }
        if (child.Name == DiagramConstant.METHOD)
        {
            loadMethod(child, newClass.LogicComponent);
        }
    }
    ...
```
6.7 Editing attributes and methods

PropertyGrid is a useful control of C# that displays all the public properties of the object binding to it. User can then view or edit those properties directly depending on the access setting of each property. With such advantage, PropertyGrid is suitable for editing AttributeElement and MethodElement objects on the fly. The only thing needs to be done to bind an object to PropertyGrid control is to set that object to the property SelectedObject of PropertyGrid. In the case of this work, AttributeElement or MethodElement object will be set to SelectedObject. But the component that is displayed on the design surface is an AttributeLabel or MethodLabel, then how to set AttributeElement object to SelectedObject when user clicks on AttributeLabel? According to design, AttributeElement and AttributeLabel objects have a bidirectional relationship, so when AttributeLabel is clicked, it will set its reference to AttributeElement object to PropertyGrid control which has globally access through class DiagramState. Listing 14 shows the detail of MouseClick event handler of an AttributeLabel. The same rule is applied for MethodLabel and MethodElement.

As mentioned above, property grid will display a property as read-only or editable depending on its access setting. It means that if property defines only ”get” method, it will be displayed as read-only. But if it defines both ”get” and ”set”, it will be editable. The way a property is displayed in the grid can be affected by another factor: C# Attributes. Table 8 shows some typical attributes.

These attributes are very useful, especially when some properties need to be customized. For example, property grid will display the original name of a property which is sometimes too long or too obscure for user. In that case, DisplayNameAttribute will change the name of property to whatever more appropriate. It also occurs sometimes that the type of property is not one of the primitive types or the value of property should be in a set of predefined values. In such case, an attribute called TypeConverterAttribute can be used to handle the display of complex property. AttributeElement has a property named AttributeType which should have value in a predetermined set (int, varchar, double etc.). To force the property grid to display that property as a drop-down list where user can select value from it, a new class AttributeTypeConverter is created to adapt the property to the grid. Listing 15 shows the essential part of that class.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisplayNameAttribute</td>
<td>Displays the desired name of property on the grid</td>
</tr>
<tr>
<td>DescriptionAttribute</td>
<td>Sets the text for the property that is displayed in the de-</td>
</tr>
<tr>
<td></td>
<td>scription help pane below the properties</td>
</tr>
<tr>
<td>CategoryAttribute</td>
<td>Sets the category that the property is under in the grid</td>
</tr>
<tr>
<td>BrowsableAttribute</td>
<td>Indicates whether the property is shown in the grid</td>
</tr>
<tr>
<td>ReadOnlyAttribute</td>
<td>Indicates whether the property is read-only</td>
</tr>
</tbody>
</table>

Table 8: Attributes for customizing property

```csharp
return true;
}
public override StandardValuesCollection GetStandardValues(ITypeDescriptorContext context)
{
    AttributeElement attrObj = (context.Instance as AttributeElement);
    if (attrObj != null)
    {
        if (attrObj.DataType == AttributeDataType.DatabaseType)
        {
            return new StandardValuesCollection(new string[] {
                DiagramConstant.DB_INT,  
                DiagramConstant.DB_CHAR,
                ...
            });
        }
        else if (attrObj.DataType == AttributeDataType.GeneralType)
        {
            return new StandardValuesCollection(new string[] {
                DiagramConstant.G_STRING,
                DiagramConstant.G_INT,
                ...
            });
        }
    }
    return new StandardValuesCollection(new string[] {});
}
public override bool GetStandardValuesExclusive(ITypeDescriptorContext context)
{
    return true;
}
```

Listing 15: AttributeTypeConverter class

The class must inherit from class TypeConverter or one of its subclass (StringConverter in this case). It also has to override three methods. GetStandardValuesSupported returns whether this object supports a standard set of values that can be picked from a list. GetStandardValuesExclusive returns whether the collection of standard values returned from GetStandardValues is an exclusive list (it means user cannot add new value to the list). Finally, GetStandardValues returns a collection of standard values from the default context for the data type this type converter is designed for. Once AttributeTypeConverter has been created, TypeConverterAttribute will do the rest of the work as in Listing 16.

```csharp
[DisplayName("Type")]
Some other complex properties are adapted the same way. The techniques and knowledges about PropertyGrid in this section are referenced in [14].

6.8 Other problems

6.8.1 Unique class name

Every class of the diagram must have a unique name to avoid the misunderstanding and duplication. For this purpose, a static method SetUniqueClassName is added to class DiagramConstant. This method will control the uniqueness of the tested name when a new class is created. If the name exists, the method will append character “2” to the name to make it unique. In case user renames an existing class, method IsClassNameUnique, also in class DiagramConstant, will check the new name. If the new name has already possessed by another class, it will return “false”, otherwise “true”.

6.8.2 Primary-key attribute of a class

Because the aim of the application is to generate database schema from class diagram, each class displayed on the design surface must have at least one attribute marked as primary key, so it will become primary key of the later generated table. Method validatePrimaryKeys of the class SQLExportForm checks all the BaseClassType objects in class collection of exported Diagram object. If some class lacks marked-as-primary-key attribute, the exporting will be cancelled and an error will be displayed. In case a class takes marked-as-primary-key attribute from its super class (see 5.4), the method will check the super class instead of the subclass.

6.8.3 Image export

Exporting the surface of a control is simple and convenient in C#. There are four steps to export the control’s content to image. Firstly, two Bitmap objects are created to determine the exported area. The first object specifies right edge and bottom edge of drawing area and the second object determines the actual area containing shapes. Secondly, the control calls its method DrawToBitmap to support rendering its content to first Bitmap object. Thirdly, four edges of drawing area will be then fitted into the second object. Finally, the second Bitmap object call its method Save to finish the work. The problem is that Bitmap object only exports the visible part of the control. It means that if the size of the control is bigger than the size of its container, its content will not be fully exported. The solution here is to maximize the container temporarily and resize it to previous state when the work is done. The example code is shown in Listing 17.
... bool maximized = false;
if (DiagramState.MainFormWindowState != FormWindowState.Maximized)
    DiagramState.MainFormWindowState = FormWindowState.Maximized;
else
    maximized = true;
...
Bitmap bitm = new Bitmap(rightEdge, bottomEdge);
Bitmap buff = new Bitmap(rightEdge − leftEdge, bottomEdge − topEdge);
canvas.DrawToBitmap(bitm, new Rectangle(0, 0, rightEdge, bottomEdge));
Graphics g = Graphics.FromImage(buff);
g.DrawImage(bitm, new Rectangle(0, 0, buff.Width, buff.Height),
            new Rectangle(leftEdge, topEdge, buff.Width, buff.Height),
            GraphicsUnit.Pixel);
...
buff.Save(filepath, imgFormat);
if (!maximized)
    DiagramState.MainFormWindowState = FormWindowState.Normal;

Listing 17: Image export
7 Results

In this thesis the problem of mapping from object-oriented data model to relational data model has been examined and discussed. The result of the study is a complete application that satisfies the requirements specification. The graphical aspect of the application is also designed and adjusted so that it will fit into the idea of visualizing database schema. The following examples will demonstrate the capability of the application through practical cases. All the figures and SQL code are created completely by the application.

7.1 Class with attributes

Figure 26 shows the visual representation of a class created on the design surface. The class contains several attributes, one of them is marked as primary key. Mapping of the class to database schema results in the SQL code shown in Listing 18.

```
CREATE TABLE Class1
(
    firstAttribute int NOT NULL,
    secondAttribute varchar(255),
    anotherAttribute date,
    PRIMARY KEY (firstAttribute)
) ;
```

Listing 18: Result table

7.2 Mapping association to new table

When an association is mapped to a new table (see 5.3), the primary keys of the involved tables will become the foreign keys in the new table. Depending on concrete scenario, such foreign keys can act as the primary keys of the new table. Figure 27 shows the case in which two classes are connected by a bidirectional association. The new table created from mapping takes primary keys from both involved tables and makes them its own primary keys. The detail is presented in Listing 19.

```
CREATE TABLE Order
(
    orderId int NOT NULL,
    total double,
    orderDate date,
    deliverDate date,
    PRIMARY KEY (orderId)
) ;
```

```
CREATE TABLE Album
(
    albumId int NOT NULL,
    title varchar(255),
```
price double,
year int,
PRIMARY KEY (albumId)
);

CREATE TABLE consistsOf
(
orderId int NOT NULL,
albumId int NOT NULL,
CONSTRAINT Order_359325e154c2 FOREIGN KEY (orderId) REFERENCES Order(orderId),
CONSTRAINT Album_00040e12df7f FOREIGN KEY (albumId) REFERENCES Album(albumId),
PRIMARY KEY (orderId, albumId)
);

Listing 19: Association mapped to junction table

7.3 Mapping association to table constraint

In contrast with creating new table, in some case an association is mapped to table constraint. This scenario is usually applied for unidirectional association. Figure 28 shows a many-to-one relationship which is mapped to the SQL code in Listing 20.

CREATE TABLE User
(
login varchar(255) NOT NULL,
password varchar(255),
PRIMARY KEY (login)
);

/*****************************/
7.4 Mapping recursive association

When a recursive association is created, there is only one class involving in the relationship. Instead of drawing an association connecting two classes, two ends of a recursive association connect to the same class as seen in Figure 29. Mapping a recursive association requires one more step: rename key to avoid duplication. Because the primary key is taken twice from the same table, the second turn will rename the key by appending some discriminator to its name. The code in Listing 21 generated from Figure 29 above appends the suffix “Child” to the name “empId” of the key.

CREATE TABLE Employee

<table>
<thead>
<tr>
<th>CREATE TABLE Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
</tr>
<tr>
<td>orderld int NOT NULL,</td>
</tr>
<tr>
<td>total double,</td>
</tr>
<tr>
<td>orderTime time,</td>
</tr>
<tr>
<td>orderDate date,</td>
</tr>
<tr>
<td>deliverDate date,</td>
</tr>
<tr>
<td>login string,</td>
</tr>
<tr>
<td>PRIMARY KEY (orderld)</td>
</tr>
<tr>
<td>) ;</td>
</tr>
</tbody>
</table>

/*****************************/

ALTER TABLE Order ADD CONSTRAINT User_ea1e6837d6bd FOREIGN KEY (login) REFERENCES User(login);
empId int NOT NULL,
name varchar (255),
PRIMARY KEY (empId)
);

/***********************

CREATE TABLE isSupervisedBy
(
empId int NOT NULL,
empIdChild int NOT NULL,
CONSTRAINT Employee_b4d81acc8098 FOREIGN KEY (empId) REFERENCES Employee(empId)
CONSTRAINT Employee_8332f3a0de61 FOREIGN KEY (empIdChild) REFERENCES Employee(empId),
PRIMARY KEY (empIdChild)
);

Listing 21: Mapping recursive association

7.5 Inheritance

As explained in 5.4, table mapped from subclass will have the attributes of super class and each class is mapped to a separate table. A database view will be created to express the inheritance between super class and subclass. The concrete example is shown in Figure 30 and then in Listing 22.

![Figure 30: Inheritance](image)

CREATE TABLE Member
(
login varchar (255) NOT NULL,
password varchar (255),
name varchar (255),
address varchar (255),
phone int,
email varchar (255),
PRIMARY KEY (login)
);

/***********************

CREATE TABLE User

(  
  login  varchar (255) NOT NULL,  
  password  varchar (255),  
  PRIMARY KEY (login)  
);  
  
/*****  
CREATE VIEW MemberView  
AS SELECT D.login, D.password, S.name, S.address, S.phone, S.email  
FROM User D, Member S  
WHERE D.login = S.login;  

Listing 22: Mapping inheritance to database view  

7.6  Mapping method to different SQL dialects  

Because SQL-92 does not define stored procedure or function, the method of the class can  
be mapped to three different SQL dialects. If the method does not return a value, it will  
be mapped to stored procedure. Otherwise, it will be mapped to stored function. The  
example in Figure 31 shows the method Discount that returns a value. The listings show  
three different ways the method is mapped. Three SQL dialects are Microsoft SQL Server  
(Listing 23), Oracle (Listing 24) and MySQL (Listing 25).  

Figure 31: Method  

CREATE FUNCTION Discount  
(@percent double, @sale bit)  
RETURNS double  
BEGIN  
END  

Listing 23: Mapping method according to MSSQL  

CREATE OR REPLACE FUNCTION Discount  
(percent IN double, sale IN bit)  
RETURN double  
BEGIN  
END;  

Listing 24: Mapping method according to Oracle
7.7 Mapping a complete diagram

In the previous sections, every aspect of a diagram is presented by both visualization and code. This section combines all those aspects and shows what a complete diagram could look like. The SQL code generated from the diagram in Figure 32 can be found in Appendix B.

![Diagram](image)

Figure 32: Example of a complete diagram

7.8 In comparison with other tools

The original purpose of this application is to supply the features that other conceptual data modeling tools have not completely satisfied their user. Compared with other tools, this application is quite simple and has fewer functions, but it can be useful in some cases:
• It can generate SQL script from visual representation of components of class dia-
gram, the thing that Microsoft Visio and Rational Software Architect do not sup-
port.

• The diagram is saved to XML file with a clear and readable structure for human.
  Thus user can understand the diagram just by reading the XML file without open-
ing it in the application.

• It supports undo/redo function which is not implemented in ArgoUML.

• Its simplicity and intuitive interface are also the advantages, because user will eas-
ily find out how the application works after a short time and not get lost in tons of
menus and buttons.
8 Conclusions

Throughout the elaboration of this thesis, the problem of implementing a tool for generating database schema from UML class diagram has been studied and solved. The result application named DiPdiagram is a demonstration of the analysis, the design and the implementation of mentioned problem. The application supports creating UML class diagram with all the necessary functionality and convenience. User can add, remove or edit every component on the design surface. The capability of exporting diagram to SQL script, which is the main part of the application, is implemented successfully. From a class diagram user can create the SQL script in standard SQL-92, with the exception of exporting methods that can specify the concrete SQL dialect. Some other functions such as exporting to image, undo/redo and help document are also implemented to improve the comfort of working with application. Finally, to fulfill the requirements, XML format is selected to keep the structure of the diagram. User can save current diagram to XML file, and then load it back to design surface later.

In the current state the application contains only the basic functionality for working with graphical components and SQL tasks. In the future version of the application, a lot of things can be added or improved. For example the application will export diagram not only to SQL script but also to code of specific programming language, the aesthetic aspect can be meliorated with better drawing algorithm and user can have more options to customize the application.
9 References


   http://www.andrew.cmu.edu/user/shadow/sql/sql1992.txt


   http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/


   http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_6009.htm

   http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_5009.htm

A  Contents of CD

- **Installer**: folder containing install files of the application.
- **Samples**: folder containing example of XML saved file and SQL script generated from it.
- **Source code**: folder containing source code of the application.
- **Text**: folder containing diploma thesis text.
- **XML schema**: folder containing XML schema of saved file.
- **README.txt**: text file containing information about CD contents.
B SQL code of a complete database schema

This appendix shows the SQL code generated from the diagram in Figure 32.

```sql
CREATE DATABASE cdshop;

USE cdshop;

CREATE TABLE Member
(
  login  varchar (255) NOT NULL,
  password varchar (255),
  name varchar(100),
  address varchar (255),
  phone int,
  email varchar (255),
  PRIMARY KEY (login)
);

CREATE TABLE ShopUser
(
  login  varchar (255) NOT NULL,
  password varchar (255),
  PRIMARY KEY (login)
);

CREATE TABLE ShopOrder
(
  orderId  int NOT NULL,
  total double,
  orderTime time,
  orderDate date,
  deliverDate date,
  login  varchar (255) NOT NULL,
  PRIMARY KEY (orderId)
);

CREATE TABLE Album
(
  albumId int NOT NULL,
  title  varchar (255),
  price double,
  releaseYear int,
  amountInStore int,
  sold int DEFAULT 0,
  rate decimal(3,2),
  PRIMARY KEY (albumId)
);

CREATE TABLE Song
(
  songId int NOT NULL,
  title  varchar (255),
  track int,
```
genre varchar (255),
PRIMARY KEY (songId)
);

/*------------------------------*/
CREATE TABLE Artist
(
artistId int NOT NULL,
name varchar (255),
nationality varchar (255),
PRIMARY KEY (artistId)
);

/*------------------------------*/
CREATE TABLE Admin
(
login varchar (255) NOT NULL,
password varchar (255),
PRIMARY KEY (login)
);

/*------------------------------*/
CREATE VIEW MemberView
AS SELECT D.login, D.password, S.name, S.address, S.phone, S.email
FROM ShopUser D, Member S
WHERE D.login = S.login;

/*------------------------------*/
ALTER TABLE ShopOrder ADD CONSTRAINT ShopUser_4d33bd90f65a FOREIGN KEY (login)
REFERENCES ShopUser(login);

/*------------------------------*/
CREATE TABLE consistsOf
(
orderId int NOT NULL,
albumId int NOT NULL,
CONSTRAINT ShopOrder_f6d8b85f4f37 FOREIGN KEY (orderId) REFERENCES ShopOrder(orderId),
CONSTRAINT Album_c860440652b4 FOREIGN KEY (albumId) REFERENCES Album(albumId),
PRIMARY KEY (orderId, albumId)
);

/*------------------------------*/
CREATE TABLE contains
(
albumId int NOT NULL,
songId int NOT NULL,
CONSTRAINT Album_1e36cb3bb9e4 FOREIGN KEY (albumId) REFERENCES Album(albumId),
CONSTRAINT Song_4acb78f826ee FOREIGN KEY (songId) REFERENCES Song(songId),
PRIMARY KEY (songId)
);

/*------------------------------*/
CREATE TABLE performs
(
songId int NOT NULL,
artistId int NOT NULL,
CONSTRAINT Song_dba4eddb7e0f FOREIGN KEY (songId) REFERENCES Song(songId),
CONSTRAINT Artist_55af83c34224 FOREIGN KEY (artistId) REFERENCES Artist(artistId),
PRIMARY KEY (songId, artistId)
);
CREATE VIEW AdminView AS SELECT D.login, D.password FROM ShopUser D, Admin S WHERE D.login = S.login;

Uncomment and insert the body of stored procedure or function

CREATE FUNCTION GetDays
(
)
RETURNS interval
BEGIN interval
END

CREATE FUNCTION Discount
(  
@percent double
, @sale bit
)
RETURNS double
BEGIN double
END

CREATE PROCEDURE ChangeTitle
@newTitle varchar
AS
BEGIN
END

Listing 26: Mapping a complete diagram
C XML schema of saved file

```xml
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="classdiagram">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="class" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="connection" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="class">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="attribute" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="method" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="abstract" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="true" />
            <xs:enumeration value="false" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="visibility">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="public" />
            <xs:enumeration value="protected" />
            <xs:enumeration value="private" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="stereotype" type="xs:string" />
      <xs:attribute name="locX" type="xs:unsignedShort" use="required" />
      <xs:attribute name="locY" type="xs:unsignedShort" use="required" />
      <xs:attribute name="width" type="xs:unsignedShort" use="required" />
      <xs:attribute name="height" type="xs:unsignedShort" use="required" />
    </xs:complexType>
  </xs:element>
  <xs:element name="connection">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="sourceclass" />
        <xs:element ref="destinationclass" />
      </xs:sequence>
      <xs:attribute name="label" type="xs:string" />
      <xs:attribute name="type" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
          </xs:simpleType>
        </xs:attribute>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:enumeration value="bidirection" />
<xs:enumeration value="unidirection" />
<xs:enumeration value="aggregation" />
<xs:enumeration value="composition" />
<xs:enumeration value="generalization" />
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="recursive">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="true" />
<xs:enumeration value="false" />
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:element name="attribute">
<xs:complexType mixed="true">
<xs:simpleContent>
<xs:extension base="xs:string">
<xs:attribute name="visibility" type="xs:string" />
<xs:attribute name="datatype" use="required">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="generaltype" />
<xs:enumeration value="databasetype" />
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="type" use="required">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="string" />
<xs:enumeration value="int" />
<xs:enumeration value="float" />
<xs:enumeration value="double" />
<xs:enumeration value="boolean" />
<xs:enumeration value="varchar" />
<xs:enumeration value="char" />
<xs:enumeration value="bit" />
<xs:enumeration value="bit varying" />
<xs:enumeration value="smallint" />
<xs:enumeration value="numeric" />
<xs:enumeration value="decimal" />
<xs:enumeration value="real" />
<xs:enumeration value="date" />
<xs:enumeration value="time" />
<xs:enumeration value="timestamp" />
<xs:enumeration value="interval" />
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
<xs:attribute name="datalength">
  <xs:simpleType>
    <xs:restriction base="xs:unsignedByte">
      <xs:minInclusive value="0" />
      <xs:maxInclusive value="255" />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:attribute name="decimallength">
  <xs:simpleType>
    <xs:restriction base="xs:unsignedByte">
      <xs:minInclusive value="0" />
      <xs:maxInclusive value="3" />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:attribute name="defaultvalue" type="xs:string" />
<xs:attribute name="stereotype" type="xs:string" />
<xs:attribute name="notnull" use="required">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="true" />
      <xs:enumeration value="false" />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:attribute name="primarykey" use="required">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="true" />
      <xs:enumeration value="false" />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:attribute name="static" use="required">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="true" />
      <xs:enumeration value="false" />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>

<xs:element name="method">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="parameter" />
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required" />
    <xs:attribute name="stereotype" type="xs:string" />
    <xs:attribute name="visibility" type="xs:string" />
  </xs:complexType>
</xs:element>
<xs:attribute name="returntype">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="int"/>
      <xs:enumeration value="float"/>
      <xs:enumeration value="double"/>
      <xs:enumeration value="varchar"/>
      <xs:enumeration value="char"/>
      <xs:enumeration value="bit"/>
      <xs:enumeration value="bit_varying"/>
      <xs:enumeration value="smallint"/>
      <xs:enumeration value="numeric"/>
      <xs:enumeration value="decimal"/>
      <xs:enumeration value="real"/>
      <xs:enumeration value="date"/>
      <xs:enumeration value="time"/>
      <xs:enumeration value="timestamp"/>
      <xs:enumeration value="interval"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>

<xs:element name="sourceclass">
  <xs:complexType mixed="true">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="role" type="xs:string"/>
        <xs:attribute name="multiplicity" type="xs:string"/>
        <xs:attribute name="locX" type="xs:unsignedShort" use="required"/>
        <xs:attribute name="locY" type="xs:unsignedShort" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name="destinationclass">
  <xs:complexType mixed="true">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="role" type="xs:string"/>
        <xs:attribute name="multiplicity" type="xs:string"/>
        <xs:attribute name="locX" type="xs:unsignedShort" use="required"/>
        <xs:attribute name="locY" type="xs:unsignedShort" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name="parameter">
  <xs:complexType mixed="true">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="type" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
Listing 27: XML schema

```xml
<xs:enumeration value="int" />
<xs:enumeration value="float" />
<xs:enumeration value="double" />
<xs:enumeration value="varchar" />
<xs:enumeration value="char" />
<xs:enumeration value="bit" />
<xs:enumeration value="bit varying" />
<xs:enumeration value="smallint" />
<xs:enumeration value="numeric" />
<xs:enumeration value="decimal" />
<xs:enumeration value="real" />
<xs:enumeration value="date" />
<xs:enumeration value="time" />
<xs:enumeration value="timestamp" />
<xs:enumeration value="interval" />
</xs:restriction>
</xs:attribute>
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
</xs:schema>
```
D Quick start guide

This section aims to provide an overview of the application DPdiagram. The guide will present the installation and then the basic functions of DPdiagram. A complete guide is included in the application in the form of the help document.

D.1 Installation

• Run setup.exe in folder Installer on the CD. The setup wizard will guide you through the steps of installation.

• Customize the configurations if you want to, then click Next to go to next step.

• If the installation process is successful, the result will be shown as in Figure 33.

• To remove DPdiagram from your computer, uninstall it from program manager in Control Panel.

D.2 Main interface

Main interface of DPdiagram is shown in Figure 34.

1. **Menu**: contains commands such as Open, Save, Export...

2. **Toolbar**: contains shortcuts of the commands on menu

3. **Title**: displays the path to saved file in file system. If a diagram is newly created and has not been saved, the title will display “New diagram *”.

4. **Class section**: buttons for creating new class

5. **Connection section**: buttons for creating new connection

6. **Property grid**: displays component’s properties

7. **Design surface**: displays all the components of current diagram

D.3 Basic functions

• To create a class, select a button in Class section (area 4) and drag it to the design surface (area 7).

• To add a new attribute, right-click a selected class and choose Add attribute from context menu.

• To add a new method, right-click a selected class and choose Add method from context menu.
Figure 33: Installation complete

Figure 34: Main interface
• To edit the properties of a component (class, attribute or method), select that component and edit its properties in the property grid (area 6).

• To create a connection, first select a button in Connection section (area 5), then click on the first class with the mouse still pressed, drag the pointer to the second class and release the mouse when the pointer lies within the second class.

• To edit properties of a connection, double-click that connection and edit information in the pop-up dialog.

• To delete a component (class, attribute, method or connection), select that component and press Delete key on the keyboard.

• To export current diagram to SQL script, choose Export to SQL on File menu and fill in necessary information in the pop-up dialog.

• To export current diagram to image, choose Export to image on File menu and fill in necessary information in the pop-up dialog.