Intelligent Data Layer:
An approach to generating data layer from normalized database model.
Abstract

Model View Controller (MVC) software architecture is widely spread and commonly used in application’s development. Therefore generation of data layer for the database model is able to reduce cost and time. After research on current Object Relational Mapping (ORM) tools, it was discovered that there are generating tools like Data Access Object (DAO) and Hibernate, however their usage causes problems like inefficiency and slow performance due to many connections with database and set up time. Most of these tools are trying to solve specific problems rather than generating a data layer which is an important component and the bottom layer of database centred applications.

The proposed solution to the problem is an engineering approach where we have designed a tool named Generated Intelligent Data Layer (GIDL). GIDL tool generates small models which create the main data layer of the system according to the Database Model. The goal of this tool is to enable and allow software developers to work only with object without deep knowledge in SQL.

The problem of transaction and commit is solved by the tool. Also filter objects are constructed for filtering the database. GIDL tool reduced the number of connections and also have a cache where to store object lists and modify them. The tool is compared under the same environment with Hibernate and showed a better performance in terms of time evaluations for the same functions. GIDL tool is beneficial for software developers, because it generates the entire data layer.

Keyword: Object Relational Mapping (ORM), Generated Intelligent Data Layer (GIDL), Relational Database, Microsoft SQL Server, Object Oriented Design Pattern, Model, Model View Controller, High Query Language, Structured Query Language (SQL) and Data Layer.
Acknowledgements

Great thanks go to the supervisor Ms. Danylenko who guided me through my thesis work. Also I would like to thank my father for guiding me to the right way in my life. Thank you both for being kind to me.
Table of Contents

CHAPTER 1: INTRODUCTION .................................................................................................................. 1
  1.1 PROBLEM DESCRIPTION ............................................................................................................. 1
  1.2 MOTIVATION ............................................................................................................................. 1
  1.3 AIMS AND OBJECTIVES ........................................................................................................... 2
  1.4 STRUCTURE OF THE REPORT ................................................................................................. 2

CHAPTER 2: LITERATURE REVIEW ................................................................................................. 3
  2.1 RELEVANCE OF LITERATURE REVIEW ................................................................................... 3
  2.2 TERMINOLOGY ......................................................................................................................... 3
  2.3 OBJECT RELATIONAL MAPPING (ORM) ................................................................................... 3
  2.4 HIBERNATE ............................................................................................................................. 4

CHAPTER 3: ANALYSIS AND DESIGN ............................................................................................ 6
  3.1 SHORT DESCRIPTION OF THE PROPOSED SOLUTION .......................................................... 6
  3.2 SOFTWARE REQUIREMENT ...................................................................................................... 6
  3.3 ANALYSIS OF GIDL TOOL ...................................................................................................... 7
  3.4 TECHNOLOGY AND METHODOLOGY ..................................................................................... 8
  3.5 DESIGN OF DATA LAYER MODEL .......................................................................................... 8

CHAPTER 4: IMPLEMENTATION ..................................................................................................... 13
  4.1 DEVELOPMENT TOOLS AND ENVIRONMENT ...................................................................... 13
  4.2 GIDL ECLIPSE PLUGIN ........................................................................................................... 13
    4.2.1 Structure Description ......................................................................................................... 13
    4.2.2 Interface Description ....................................................................................................... 15
  4.3 DATA LAYER JAVA PROJECT ................................................................................................ 15
  4.4 TESTING .................................................................................................................................. 20
  4.5 EXPERIMENT RESULTS: COMPARISON WITH HIBERNATE ................................................ 22

CHAPTER 5: CONCLUSION ................................................................................................................. 24
  5.1 SUMMARY ............................................................................................................................... 24
  5.2 FUTURE WORK ......................................................................................................................... 25
  5.3 EVALUATION ............................................................................................................................ 25

REFERENCES ..................................................................................................................................... 26

APPENDIXES ..................................................................................................................................... A1
  APPENDIX A: CLASS AND SYSTEM SEQUENCE DIAGRAM ......................................................... A1
  APPENDIX B: SOURCE CODE ......................................................................................................... B1
  APPENDIX C: SUPERVISOR-SUPERVISEE MEETING LOGS ......................................................... C1
# Table of Figures

**Figure 3.1** Overview of Data Layer ................................................................................................................................. 6  
**Figure 3.2** Data Layer Package Diagram .......................................................................................................................... 9  
**Figure 3.3** “com.server” Package ................................................................................................................................. 10  
**Figure 3.4** Package “db” Class Diagram ....................................................................................................................... 11  
**Figure 3.5** Transaction Problem ........................................................................................................................................ 12  
**Figure 4.1** GIDL Plug-in .................................................................................................................................................... 14  
**Figure 4.2** GIDL User Interface ....................................................................................................................................... 15  
**Figure 4.3** Data Layer Java Project .................................................................................................................................. 16  
**Figure 4.4** Student Table .................................................................................................................................................. 20  
**Figure 4.5** CRM Database Model ..................................................................................................................................... 21
Table of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>ORM Tools</td>
<td>4</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Use Case Generate Layer</td>
<td>8</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Package “GIDL-Action”</td>
<td>15</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Controller Class</td>
<td>17</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Transaction Manager Class</td>
<td>18</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Data Layer Packages and Classes</td>
<td>20</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Comparison Result</td>
<td>22</td>
</tr>
</tbody>
</table>
Abbreviations

UML – Unified Modeling Language
ORM – Object Relational Mapping
GIDL – Generated Intelligent Data Layer
RDBMS – Relational Database Management System
OOA – Object Oriented Architecture
MVC – Model View Controller
SQL – Structured Query Language
HQL – High Query Language
MSSQL – Microsoft Structured Query Language
XML – Extensible Markup Language
CRUD – Create Read Update Delete
SDLC – Software Development Life Cycle
Chapter 1: Introduction
In first section of this chapter, the problem is described in details. Then motivation is shown for giving a solution to the main problem. The following section summarizes the main aim and a set of objectives. Finally in the structure of the report is summarized.

1.1 Problem Description
There is a variety of ORM frameworks or tools in the software market. Most of them are open source and few commercial tools. It is important to mention that many programmers develop and maintain their own ORM tools according to their need. ORM tools help software developers to maintain connection between database and the system. During the development phase in software life cycle, it is noticed that design of the data layer of a system and the developing process are quite the same, therefore it can be seen as a routine job. The same design of the data layer is used where the only thing that differs are business object. During personal experience in software companies, we have noticed that managers are often asked to design and develop such a layer for optimal results which correspond to a given normalized database.

Generally the problem is to design a data layer that provides secure connections, efficiency and optimal result. Then develop a generator in order to remove the routine programming of a data layer. An eclipse plug-in should be developed to generate in a structured way the data layer. Most of the current ORM tools provide an environment for developing the data layer and later on the domain model. There exist different tools like Hibernate which facilitate the storage and retrieval of Java objects however the main problem is to generate such a data layer based on ORM concepts. Unlike Hibernate and other ORM tools which facilitate mapping of the main model to database schema, the proposed solution generates a data layer from a predefined database schema. So despite similarities between Hibernate and our solution, there are also many differences.

We proposed an engineering approach where a data layer and small models are generated by a software tool named Generated Intelligent Data Layer (GIDL). The data layer will be used in the future to create the main model of the system. The proposed tool will help and enable programmer to work only with object unless they know SQL. The proposed tool should reduce the number of connections and also have a cache where to store object lists and modify them. Also it should facilitate the development of the main domain model. If it will be used in MVC architecture, programmers will focus on business logic of the system rather than routine work on CRUD (Create, Read, Update and Delete) functionalities [2].

For every table in the database a mini-model is created using Singleton Design Pattern. Data Layer is the bottom layer in the architecture of a system and should maintain persistency of business object. Transaction Manager Layer will use DB Layer and solve the problem of transaction and commit, work only with objects and also have a cache. Controller Layer is the highest layer and will be used by programmers to manipulate the database. The design that has been chosen, should offer a better performance and efficiency.

1.2 Motivation
Based on personal experience in different software companies, it was discovered that Rapid Application Development (RAP) software engineering process focuses on a rapid implementation phase. From one side the systems which are connected to a database will have a data-layer and the generation of such a layer would really facilitate software developers during the software life cycle of a system. On the other side proposing a
secure, reliable and standard data layer for the main model would increase the performance of the systems. Also it would really help programmers that do not have knowledge of SQL language to work only with objects that are the basic concept of Object Oriented Architecture which is the current trend and is widely spread nowadays among software companies [1].

Model View Controller software architecture is widely spread and the generation of data layer for the main model will reduce cost and time. After researching for similar tools and relevant theories presented in Section 2.3, it was discovered that there are similar generator tools like Data Access Object (DAO) and Hibernate but while using them a lot of problems arise like inefficiency, slow performance due to many connections with database, time to set up, adaption for different technologies and time consuming in learning the frameworks [4, 8].

1.3 Aims and Objectives
The main goal and objectives of this project are formally stated as follows. The goal of the thesis project is to summarize the problems and propose a consistent approach to the defined problem in the Section 1.1

In order to achieve the main goal of the thesis, the following objectives are identified:

O1. Literature Review on similar tools and theories.
O2. Define Software Requirements of data layer, pre and post condition.
O3. Design, develop and test the data layer for a given normalized database.
  Database Normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and dependency.
O4. Develop a software tool for generating the data layer.
O5. Comparing with one of the existing similar solutions (Hibernate)

1.4 Structure of the Report
The rest of the report is divided in four main parts: literature review, analysis and design, implementation and conclusion. During the literature review in Chapter 2, similar tools and frameworks are observed and compared with the proposed data layer. In Chapter 3, requirements of the data layer are stated and the design of the layer is shown. In Chapter 4 the generation of the data layer is shown. Furthermore a comparison with Hibernate and testing phase is shortly discussed. Finally the report ends with conclusion where an evaluation of the thesis is done in Chapter 5.
Chapter 2: Literature Review

In this chapter, the literature review and research is presented. Firstly, the relevance of literature review regarding our topic is explained. Then terms and different concepts of software field which are relevant to our thesis are explained in details to clarify any ambiguities. In what follows, ORM theory and tools are reviewed and presented shortly. In the end Hibernate platform is described and compared with our proposed solution.

2.1 Relevance of Literature Review

Designing, developing and generating a data layer is a need research on ORM field and MVC architecture, before proceeding to the next phases of the Software Development Life Cycle (SDLC). Generating a data layer from a normalized database is not new and some ORM tools attempt to solve this problem. Most of ORM tools develop a platform for creating a data layer according to software system requirement. ORM is not new field and a lot of research is done by academy and software vendors. Many platforms and frameworks are developed for many technologies to implement object relational mapping. The overview of ORM and MVC is required during integration phase [1].

2.2 Terminology

New concepts and terms are presented during the rest of the report which should be known in order to have a better understanding. In the following list all these terms are presented.

1. Data Layer is the bottom layer in the architecture of applications which connect with a database and its tasks are to create, read, update, delete database entities and maintain the connections.

2. Model View Controller is predefined computer software architecture that separates the representation of the information from the user’s interactions with it. The model consists of data and business logic. The controller controls the input or changes and converts them in commands for the model or the view. The view is the user interface of application or any representation of data [1].

3. Database Normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and dependency. Every relational database that undergoes through this process is called Normalized Database [7].

4. Domain Model is a conceptual model of all the topics related to a specific problem in software engineering. It describes entities, their attributes, roles, relationship and constrains [14].

5. Design Pattern is a general reusable designed solution to a commonly occurring problem in the context of software engineering [14].

2.3 Object Relational Mapping (ORM)

Many applications consist of business logic, user interfaces, interaction with other systems and data layer which manipulates data and stores them in databases most of the time. In many cases most of these data are stored, retrieved, processed and analyzed by the application in relational databases which store data in tables. Most of the data in tables are identified by primary key, identities or indexes. Meanwhile in Object Oriented programming data and relationship are stored in attributes and pointers respectively and a domain model consist of a set of objects interacting between them [1]. The process of mapping attributes to columns, objects to tables, pointers to foreign key from a domain model to database is called ORM [1, 2].
Many ORM should provide persistency of object which means that the state of the objects should outlive the applications process. Objects must live beyond the virtual machine and the temporary memory. In what follows, present a short overview of existing ORM tools (Table 2.1).

<table>
<thead>
<tr>
<th>Name</th>
<th>Open Source</th>
<th>Programming Language</th>
<th>SQL support</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiteSQL [9]</td>
<td>No</td>
<td>C++</td>
<td>MySQL, PostGreSQL, SQLite</td>
</tr>
<tr>
<td>ODB [10]</td>
<td>No</td>
<td>C++</td>
<td>MySQL, PostGreSQL, SQLite, MSSQL, Oracle</td>
</tr>
<tr>
<td>Hybernate ORM [4]</td>
<td>YES, widely used framework</td>
<td>Java</td>
<td>MySQL, MSSQL, Oracle</td>
</tr>
<tr>
<td>Devart LinqConnect [12]</td>
<td>No</td>
<td>.NET</td>
<td>MySQL, PostGreSQL, SQLite, Oracle</td>
</tr>
<tr>
<td>CakePHP ORM [13]</td>
<td>YES, framework</td>
<td>PHP5</td>
<td>MySQL</td>
</tr>
</tbody>
</table>

Table 2.1: ORM Tools

Although there is a lot of progress in this field, it is still required a lot work because there are a lot of programming languages and software database management tools which leave space for research. The proposed solution is based on ORM concepts and solves the problem of object persistency, CRUD functionalities and transaction commit. However the proposed approach uses the ORM concepts in the inverse way that is from a normalized database it generates a data layer for the domain model. Therefore ORM concepts are highly relevant to this thesis.

2.4 Hibernate

Hibernate is an ORM library for Java. It provides a framework to map a domain model to a relational database. Hibernate was started in 2001 by Gavin King as an alternative to EJB2 entity beans. The first aim of Hibernate was to offer better persistence capabilities than EJB2 [4]. In 2003, Hibernate 2 was improved and released and offered a better environment. Later on, JBoss hired Hibernate developers and collaborate with them in supporting Hibernate [4]. In 2010 Hibernate 3 was released and after in Dec 2011, Hibernate Core 4.0.0 Final was released which is one of the most stable and completed version. Finally in 2012 Hibernate 5 started to develop and it will contain JPA 2.1 support [4].

Hibernate maps classes to database tables and attributes to columns. It works with High Query Language (HQL) for filtering objects in the database. Hibernate does not generate a data layer but it provides the right environment and facilitates the connection between databases and domain model for software developers. It has high level object functions that realize the persistency of object. Hibernate is one of the most used ORM tools among the others nowadays and it works with many databases. Based on personal experience we can argue that, Hibernate shows a great performance with Oracle database but not with MSSQL Server [3, 4].
Hibernate fulfills all CRUD functionalities and allows programmers to store any object easily without much effort spend for choosing the configuration, installation and storage. It helps software developers as an ORM tool to map in any object to a table through XML files. It requires training by programmers in order to use it appropriately. Hibernate is a powerful tool for dynamic systems that tend to change their requirements because it provides flexibility to synchronize the source code with the data storage structure [4].

Hibernate reduces coding time, and it is a powerful ORM tool for supporting big domain models which require a large database. Hibernate is different from the proposed solution because it is a platform rather than a generating tool however both of them have the same aim. The proposed solution looks similar with Hibernate regarding ORM concepts therefore it is required to have an overview of Hibernate framework and its features. Furthermore, Hibernate is used to compare the outcome results that were derived from the proposed approach in Section 4.5.
Chapter 3: Analysis and Design

In this chapter, Analysis and Design of the proposed solution are shown. In first section all the requirements of the proposed tool are defined and stated. Then a use case is used to show the main interaction between the user and the proposed tool. Furthermore technology and methodology used in the project are shown and justified. Object Oriented Analysis and Design Paradigm is used during Analysis and Design phase. Finally in the last section the proposed design is shown as a solution for the data layer.

3.1 Short Description of the Proposed Solution

Shortly the idea for solving the problem is to generate a data layer consisting of three sub-layers which are Controller, Transaction Manager and Database Layer. In the end the software developers may use only the Controller to manipulate the business objects in the main model. Controller should offer CRUD (create, read, update, delete) functionalities, get all business object with or without filtering and reset cache. Figure 3.1 shows an overview of the proposed solutions. Section 3.6 explains the idea in details.

![Diagram of data layer](image)

Figure 3.1: Overview of Data Layer

3.2 Software Requirement

The proposed tool has a list of requirements that should be fulfilled:

1. It should generate a data layer as a project for the given relational database with classes, object and packages. The project should be well structured and easy...
understandable by software developers, so the set of classes and object should be putted in logical packages.

2. It should provide facilities to create an object and save it. For each table entity a corresponding object class should be generated so the programmers might use it to initialize. After initializing the object a class should be used to save the object.

3. It should provide facilities to update an object which is the same as to save the object. The proposed tool should define if the object is new or old and decide for an update or save.

4. It should provide facilities to delete an object. Every object in the system might be deleted so the data layer should provide this ability.

5. It should provide facilities to read an object and retrieve it from databases. The data layer should read all table entities and store them in a data structure which should be easily accessible.

6. It should have a cache to maintain performance. As most of ORM tools, the proposed approach stores all the retrieved objects in a cache to increase the performance of the data layer.

7. It should be flexible if connection’s properties are changed and the generated code should not be touched. All the connection properties should be stored in a different file and if the database changes, the software developers should change only one file.

8. It should guarantee a secured connection. The user password of the database should be encrypted.

9. It should facilitate the construction of the domain model. A list of utility classes, interfaces and abstract classes in the packages should provide more functions for the software developers.

10. It should provide additional functionalities for managing list of objects. An example would be finding object by id, key or code.

11. The generated code should be qualitative. Object-Oriented programming rules should be followed.

12. The generated code should follow Java Convention coding.

13. The data layer should work with MSSQL Server.

### 3.3 Analysis of GIDL tool

GIDL tool has only one critical task which is expressed in the use case below. In the next chapter the generated data layer is explained.

<table>
<thead>
<tr>
<th>Use Case 1: Generate Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors</strong></td>
</tr>
<tr>
<td><strong>Pre-conditions</strong></td>
</tr>
<tr>
<td><strong>Post-conditions</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Action Num.</strong></td>
</tr>
<tr>
<td><strong>Main Success Scenario</strong></td>
</tr>
<tr>
<td>1.</td>
</tr>
</tbody>
</table>
2. User presses button test connection. GIDL checks the connection for the given inputs and enable “Generate” button.

3. User presses Generate. GIDL creates an eclipse project named “DataLayer”.

### Extension

| 2.a | GIDL checks the connection and if it is not valid. It does not enable “Generate” button. |
| 2a1. | It informs user with an Error Message |

| Priority | High |
| Risk     | Critical |

Table 3.1: Use Case Generate Layer

### 3.4 Technology and Methodology

Object Oriented Analysis and Design methodology is a common trend and it is chosen for implementing data layer and GIDL. UML is used as modelling language because is accepted de facto as a common standard for software developers [5]. Objected Oriented Paradigm is chosen for implementation of the project because it is widely used nowadays and is adapt to the features of data layer. For implementing the project, Java programming language was chosen as a suitable and appropriate Object Oriented Language.

MSSQL Server [6] is chosen for implementing a database since a lot of experience and knowledge is owned in this area. Although the data layer has been designed to work with other SQL drivers, currently JTDS driver [15] and MSSQL is appropriate to develop a testing database. Moreover JTDS driver is open source and MSSQL offers a set of tools for creating and managing and testing databases.

### 3.5 Design of Data Layer Model

The data layer is designed to have five main packages. Packages “businessobject” and “filter” are dynamic and contain the mapped objects and filters from the database. Package “com.server” which implements, maintains, handles, and facilitates the mapped objects. Static packages “db” and “util” maintain and facilitate the connection. Package “com” should contain a main class for testing purposes and it is left to build the domain model. Fig 3.2 shows the design and the connections between the packages. The packages in the figure represent the logical components of the data layer while the arrows represent the associations and dependency between packages. In what follows every package is explained in details.

Package “com.server” package have three main classes: Controller, TransactionManager and DBLayer class. Controller maintains the connection between the model and the TransactionManager that is a bridge between the DBLayer and Controller. The overall design is shown in the class diagram at Figure 3.2 where an Student class is given as an example. Controller class provide a set of methods for the business object Student. Controller class is a Singleton Design Pattern so only one instance of this class can run all over the model. Furthermore it retrieves Student objects from the TransactionManager class and stores them in the cache. TransactionManager which has an association with DBLayer class acts as a bridge and
requires a database layer for each committed transaction. **DBLayer** is the bottom class in the data layer as shown in bottom left of Figure 3.3. This class maintain all the generated SQL and Java code for the class **Student**. Furthermore it construct all list needed by the upper classes.

![Figure 3.2: Data Layer Package Diagram](image)

This design was chosen because it is high-cohesion and loosely-coupled. The generated code is more qualitative and the abstract objects are expressed better. The step toward the database was divided into three logical layers based on their behaviour and attributes.
Package “db” is the lowest layer in the package hierarchy and maintains SQL connections between package “com.server” and database. Figure 3.4 shows some of the main classes in “db” package and the relationship between them. ObjectContainer class is inherited by all business objects retrieved by the DBLayer class. StoreList data structure stores all business object lists and has a dependency association with ObjectContainer class. DBAdapter is the main class that adapts to database based on
connection properties. This class is associated with *Transaction* class, is depended with *JDBCConnection* class and inherits from the *DefaultDBAdapter* abstract class. During design, it was chosen to provide default values in abstract classes with predefined attributes and behaviour which could be inherited by sub-classes. The generalization of entities and providing default values is expected to provide a stable layer. The important classes and packages are described in more details in Section 4.3.

Figure 3.4: Package “db” class diagram

One of the basic requirements of ORM tools is the implementation of transaction commit problem [2,6]. Basically, there is a need to execute two SQL queries successfully at the same time otherwise none of them should be executed. Suppose that an object *Student* and its object *Address* need to be stored. First object *Student* will be stored and an object *Address* with its identity will be stored in the database. If the first transaction fails than address should not be stored so either both of them should success or fail in saving the data. In this case the proposed approach attempts to execute both SQL commands with one database layer.
Our solution is shown in Figure 3.5 where the same database layer is used to commit a transaction or is rolled back. Other class diagrams and system sequence diagram are shown in Appendix A.

Figure 3.5: Transaction Problem
Chapter 4: Implementation

In this chapter implementation phase is described and testing results are demonstrated. First development tool and software environment is shown and argued, why they are chosen as appropriate for the development. Then eclipse plug-in and the data layer are described in details. The report continues with describing testing phase used during the development phase. Finally comparison results with Hibernate are demonstrated.

4.1 Development Tools and Environment

The selection of Java platform as the most appropriate programming language has been discussed in Section 3.4. The platform, used for compiling and interpreting the classes is Java Enterprise Edition 6 Software Development Kit (JEE6SDK) [16]. Java 1.7 was excluded since it showed a lot bugs for building an eclipse plug-in based on the personal experience. Eclipse IDE (Integrated Development Environment) [17] platform is used as an editor for building GIDL plug-in. Eclipse IDE was used for debugging and handling different .jar packages and building GIDL tool. Such plug-ins as “org.eclipse.ui” and “org.eclipse.core.runtime” are required for GIDL plug-in to run. Furthermore GIDL plug-in depend on these packages:

1. org.apache.commons.collections,
2. org.apache.commons.io,
3. org.apache.commons.lang,
4. org.eclipse.core.resources,
5. org.eclipse.jdt.core,
6. org.eclipse.jdt.launching,
7. org.eclipse.pde.core.plugin

It was noticed that Java EE 1.7 shows a higher performance than Java EE 1.6 therefore it is used for the generated data layer. To connect with MSSQL [6] database, JTDS [15] driver is used because it shows a higher performance than equivalent JDBC-ODBC [15] driver. Both drivers serve the same purpose however based on personal experience JTDS was chosen because it is stable, fast and open source unlike JDBC-ODBC. Alto JTDS passes J2EE 1.3 Certification and Hibernate test suites [15].

Although MSSQL database server is used to implement a normalized database for testing purposes, the layer is designed to be extended and work with other database management tools. The generated data layer is extended to work with other drivers, but it was not tested and this is obviously a future work. SQL Server Management Studio 2008 R2 [6] is used to design, create and normalize the database.

The object oriented UML diagrams that were shown during the report were created with ObjectAid [18] eclipse plug-in. Finally, the Operating System where the data layer and the plug-in were developed and tested is Microsoft Windows 7 Ultimate.

4.2 GIDL Eclipse Plug-in

The major objective of the plug-in is the generation of the data layer for the given normalized database and on the interface side the objective is to require the entering of the connection properties. This section is divided into two subsections where structure and interface are described seperately.

4.2.1 Structure Description

The implementation of the plug-in is divided in two main packages “basereader” and “gidl.action”. In package “basereader”, all tables are extracted from database in the first step followed by the second one where all columns and data types are extracted from each table that has an identity. In order to maintain persistency, every entity in each
A table should have an identity (ID) otherwise it is possible to have multiple objects with the same value. Almost all the tables in normalized databases have an identity. In Object Oriented paradigm every object has an identity, state and behaviour and for mapping rows to object, it is required an ID for each database entity. The plug-in structure is shown in Figure 4.1. The left part shows the set of classes used to build the plug-in and their respective packages. The right part of the figure shows other required plug-ins and dependencies.

In the second package “gidl.action”, there are multiple actions that were implemented to fulfil the objectives: All classes are shown in Table 4.1.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBFiles</td>
<td>Stores all source code of package “db” of the data layer.</td>
</tr>
<tr>
<td>DBProperties</td>
<td>Stores all connection properties.</td>
</tr>
<tr>
<td>DesEncrypter</td>
<td>Encrypts or decrypts the database password in order to provide a secure connection.</td>
</tr>
<tr>
<td>GIDLAction</td>
<td>Auto-generated class which provides a set of interface commands.</td>
</tr>
<tr>
<td>JDBCConnection</td>
<td>Creates and maintains all SQL Connection.</td>
</tr>
<tr>
<td>MyDialog</td>
<td>Generates a User Interface which is described in details at the following Subsection 4.2.2.</td>
</tr>
<tr>
<td>Project Creator</td>
<td>Creates a project, adds the static java files, creates the dynamic files read from “basereader” package mentioned above and compiles them in the end.</td>
</tr>
<tr>
<td>UtilFiles</td>
<td>Stores all the source code of the package “util” of the data layer.</td>
</tr>
<tr>
<td>Column</td>
<td>An object column is created with column name, attribute name and data type.</td>
</tr>
<tr>
<td>Table</td>
<td>An object table is created with table name and a list of</td>
</tr>
</tbody>
</table>
A Singleton Design Pattern is used to store all tables.

A procedure class where all tables are read and store to the Tables Class.

### Table 4.1: Package “gidl.action”

#### 4.2.2 Interface Description

*MyDialog* class creates and maintains the user interface which is shown in Figure 4.2. The user interface asks for the database connection properties such as “Username”, “Password”, “MSSQL Server Name”, “Instance” and “Database Name”, and then it stores them. The user is asked to test the connection for the input values. If the connection is valid, then user press “Generate” button and a new Java Project named “DataLayer” is created which is described in the following section.

![Figure 4.2: GIDL User Interface](image)

#### 4.3 Data Layer Java Project

In this section, an explanation of Data Layer implementation will be showed on a particular example. As it was mentioned in previous chapter data layer consists of five main packages which will be described in details based on their relevance. For a brief explanation only a database table Student is generated to Java code. The left part in Figure 4.3 shows the generated classes which enable mapping of the Student table to *Student* class. Package “com” has only one main class and is left empty to be used by the software developers.

In the “businessobject” package, *Student* class is generated as it is shown in the right part of Figure 4.3. The table columns such as “name”, “surname”, “course” and “date” are mapped as attributes of the *Student* class by following the Java conventions for writing code. Also the types of columns are transformed in Java types. *UtilDate* class facilitates the transformation of SQL Datetime type to Java type. Student class inherits from *ObjectContainer* which also inherits from the abstract class *DBEntity*. This design was chosen as the optimal solution because it offers flexibility for treating
business object in the main model. The same values and types are seen as database entities and objects while the ObjectContainer is abstractly a bridge and act like a storehouse. As it is shown in the Figure 4.3 each business object is initialized by setting the ID, if the ID is empty then the object takes a default ID value “-1”.

As it is shown in the Figure 4.3 each business object is initialized by setting the ID, if the ID is empty then the object takes a default ID value “-1”.

Figure 4.3: Data Layer Java Project

Package “filter” is simple and contains respective filter object for every business object. Package “com.server” is a dynamic package which consists of three main classes: Controller, DBLayer and TransactionManager.

Controller class is the top class in the hierarchy and Singleton Design Pattern is used to construct it, so wherever Controller is used only one instance of this class will run. Controller class manages cache for all business objects by setting a new identity for each business object. Furthermore, Controller class provide five functionalities for Student object which are:

1. getAllStudent() method returns a StoreList of Student objects from the Cache object. If Cache is empty then it gets fresh Student objects from the database, stores them in the Cache and returns them. Singleton Design Pattern allows only one instance of the class to run so only one copy of Student objects will be provided by the data layer.
2. getAllStudent(StudentFilter filter) method which return a StoreList of Student objects based on the StudentFilter.
3. deleteStudent(Student data) method deletes the specified Student object in the database and resets cache.
4. saveStudent(Student data) method saves the specified Student object in the database and resets Cache.
5. resetStudentCache() method resets the memory in the Cache object that was used to store StoreList of Student object.

Table 4.2 shows the code of the Controller class.

```java
public class Controller {
    private static Controller controller;
    private static Cache cache;
    private static final int ALL_STUDENT = 4000;

    Controller() {
    }

    public static Controller getInstance() {
```
if (Controller == null) {
    Controller = new Controller();
}
return Controller;

private TransactionManager getTransactionManager() {
    return new TransactionManager();
}

private static Cache getCache() {
    if (cache == null) {
        cache = new Cache();
    }
    return cache;
}

// Student
public StoreList getAllStudents() {
try {
    return (StoreList) getCache().getCacheObject(ALL_STUDENT);
} catch (Exception e) {
    return (StoreList) getCache().setCacheObject(ALL_STUDENT, getAllStudentsOfSystem(new StudentFilter()));
}
}

public StoreList getAllStudents(StudentFilter filter) {
    return getAllStudentsOfSystem(filter);
}

private StoreList getAllStudentsOfSystem(StudentFilter filter) {
    return getTransactionManager().getAllStudents(filter);
}

public void deleteStudent(Student data) throws Exception {
    getTransactionManager().deleteStudent(data);
    resetStudentCache();
}

public void saveStudent(Student newData) throws Exception {
    getTransactionManager().saveStudent(newData);
    resetStudentCache();
}

public void resetStudentCache() {
    getCache().resetCache(ALL_STUDENT);
}

//End of Student

Table 4.2: Controller Class

TransactionManager class acts like a bridge between Controller and DBLayer class. It also manages all the transactions and uses only one database layer to commit a transaction. For every business object it provides the same functionalities as Controller except “Save” functionality which is split into update and save. This is done by checking if the new object has an ID or if it is a new instance. Table 4.3 shows how a transaction is committed.
```java
public class TransactionManager extends Transaction {
    public TransactionManager() {
    }

    public TransactionManager(Transaction transaction) {
        super(transaction);
    }

    public DBAdapter createDBAdapter() {
        return new DBLayer();
    }

    public DBAdapter createDBAdapter(DBAdapter dbAdapter) {
        return new DBLayer(dbAdapter);
    }

    private DBLayer getDBLayer() {
        return (DBLayer) getDBAdapter();
    }

    public void saveStudent(Student student) throws Exception {
        try {
            DBLayer dbLayer = getDBLayer();
            if (student.hasDefaultID()) {
                //inserting
                dbLayer.saveStudentData(student);
            } else {
                //updating
                dbLayer.updateStudentData(student);
            }
            attemptToCommit();
        } catch (Exception e) {
            attemptToRollBack(e);
        } finally {
            attemptToClose();
        }
    }
}
```

**Table 4.3: Transaction Manager Class**

*DBLayer* is the last class in the hierarchy and it implements all CRUD functionalities for all business objects. It uses *PreparedStatement* Java classes to structure SQL queries. *StoreList* object are created and filled with business object which are lately used by the domain model. The source code is presented in Appendix-B.

Package "db" consists of static classes which maintain the connection of database and the *DBLayer* class. Some of the main classes are: *StoreList, JDBCConnection, DBAdapter, DBConfiguration*. *StoreList* data structure has some extra functionality like get by key, get by code, caching, get by id for every business object. *JDBCConnection* class acts as a factory to provide successful connection to the *DBLayer* or *DBAdapter* class. *DBAdapter* adapts and synchronizes the connections. *DBConfiguration* sets the database connection properties which are read from the config.xml file.
Package “util” have some utility classes like *DesEncrypter* used for encrypting the password, *Cache* and *UtilDate. Cache* class acts like a storehouse where all objects are stored by an identity. The source code for all important classes is presented in Appendix-B. Finally Table 4.4 presents a summary of all the packages and classes.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>“bussiness object”</td>
<td>This package consists of business object and in our example <em>Student</em> object.</td>
</tr>
<tr>
<td>StudentFilter</td>
<td>“filter”</td>
<td>This package consists of respective filters for the business objects.</td>
</tr>
<tr>
<td>Main</td>
<td>“com”</td>
<td>It consists of main class and is left to be used by software developers for building the main model.</td>
</tr>
<tr>
<td>Controller</td>
<td>“com.server”</td>
<td>It is the upper layer class which provide business object to the main model.</td>
</tr>
<tr>
<td>TransactionManager</td>
<td>“com.server”</td>
<td>It is the middle layer class in the hierarchy of package “com.server” and it handles the transaction-commit problem.</td>
</tr>
<tr>
<td>DBLayer</td>
<td>“com.server”</td>
<td>It is the bottom layer class which constructs and execute SQL queries. It inherits from <em>DBAdapter</em> class.</td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>“db”</td>
<td>It stores constants values.</td>
</tr>
<tr>
<td>DBAdapter</td>
<td>“db”</td>
<td>It is an abstract class that stores all attributes for adapting with a database and committing a transaction.</td>
</tr>
<tr>
<td>DBConfiguration</td>
<td>“db”</td>
<td>It stores all the connection properties the database.</td>
</tr>
<tr>
<td>DBEntity</td>
<td>“db”</td>
<td>It is an abstract class with predefined attributes for database entities. It is a super class for <em>ObjectContainer</em> and all business objects.</td>
</tr>
<tr>
<td>DefaultConfigurationManager</td>
<td>“db”</td>
<td>It reads from config.xml file.</td>
</tr>
<tr>
<td>DefaultDBAdapter</td>
<td>“db”</td>
<td>It inherits from <em>DBAdapter</em> class is rarely used. It consists of five different constructors for adapting the connection properties or checking the <em>DBAdapter</em> object. Also it returns a <em>HashMap</em> object with all connection properties.</td>
</tr>
<tr>
<td>DefaultTransaction</td>
<td>“db”</td>
<td>It inherits from transaction and has a generalized behaviour of transactions.</td>
</tr>
<tr>
<td>JDBCConnection</td>
<td>“db”</td>
<td>It is like factory that generates SQL <em>Connection</em> objects.</td>
</tr>
<tr>
<td>ObjectContainer</td>
<td>“db”</td>
<td>It is like storehouse for all business objects. It inherits from <em>DBEntity</em> class and is a super class for business object classes. <em>StoreList</em> class works with <em>ObjectContainer</em> objects.</td>
</tr>
<tr>
<td>StoreList</td>
<td>“db”</td>
<td>It is a data structure which stores and retrieves business objects. It is more complicated than Java List class.</td>
</tr>
<tr>
<td><strong>Transaction</strong></td>
<td>“db”</td>
<td>It is an abstract class with all attributes for a transaction.</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>“util”</td>
<td>It is like storehouse class where different object might be stored. It has reset functionality.</td>
</tr>
<tr>
<td><strong>DesEncrypter</strong></td>
<td>“util”</td>
<td>It encrypts and decrypts database user password.</td>
</tr>
<tr>
<td><strong>UtilDate</strong></td>
<td>“util”</td>
<td>It facilitates the transformation of SQL Datetime type to Java type.</td>
</tr>
<tr>
<td><strong>UtilJOptionPane</strong></td>
<td>“util”</td>
<td>A utility class to generate JOptionPane dialogs.</td>
</tr>
</tbody>
</table>

Table 4.4: Data Layer Packages and Classes

4.4 Testing

In this section, testing phase of the project is shown. White and Black Box testing are used throughout the Software Development Life Cycle [5]. White Testing is a technique used by software developers to test simple units of the software. The testing is done by programmers for every unit they develop during implementation phase [5]. Black Box testing is used in the end to find possible errors and to check the fulfilment of functionalities. It is done by testers to test the entire system as a box without considering the components [5]. The experimentation was done in a computing device with 2 GB RAM, Intel Pentium Core Duo 2.1 GHz processor and OS is Window 7 Ultimate 32-bit. Eclipse platform and its debugger were used as a testing environment for GIDL tool. Microsoft SQL Server 2008 database management tool was used to design and build testing databases.

White testing was done in parallel with implementation phase of the data layer. During the generation of the data layer a normalized database was used to generate mini-model, in order to test the performance and efficiency of the generated code. A lot of errors were discovered during this phase such as mapping types with loosing the precision of values, missing attributes, mapping wrong attributes and problem with import of classes. A good example during this phase is **UtilDate** classed which is designed to map better SQL Datetime to Java date type. The plug-in classes were tested one by one, until no further errors were found. This process was highly successive because a lot of errors were found and fixed.

Black Box testing technique is used in the end. The plug-in is tested as a Black Box, by checking if all the requirements and functionalities are fulfilled. During these phase, two simple databases were constructed in order to make sure that the testing and the outcome were good. The simple database “Student” was used also for both Hibernate and GIDL tool in order to see an outcome and compare them. Figure 4.4 shows the “Student” Database Table.

![Student Table](image)

Figure 4.4: Student Table
Furthermore the same data layer is used to develop a CRM (Custom Relationship Management System) in Albania. The software developer was asked to design and create a data layer. Due to some months of research, the first layer was developed and adapted to CRM system. Data layer usage and adaption with a real time system proved to be successful. The second database used during testing phase is CRM normalized database model and is shown in Figure 4.5.

Figure 4.5: CRM Database Model
4.5 Experiment Results: Comparison with Hibernate

As it was mentioned in Literature Review Chapter at Section 2.4, a similar tool with GIDL is Hibernate. Hibernate was used and we have tested for similar functionalities with our data layer for speed performance and efficiency. The testing was done under the same software and hardware resources in order to guarantee fair result. Both of them were configured to work JTDS driver and MSSQL Server in order to guarantee the same software environment. The table “Student” showed in Figure 4.8 was used as an example for Hibernate and our data layer. Also in order to provide an accurate result, five tests were done to find average values for completing each of CRUD functionalities. The measurements unit to find average time were done in milliseconds. Read functionality is tested without using the cache while other functionalities do not use the cache. Both Hibernate and our data layer complete Read functionality nearly in the same time if they used the cache. In the first column of Table 4.5 all functionalities are stated. In the second column, the respective result times of the proposed data layer are shown for all functionalities. Third column presents Hibernate result times for completing the respective functionalities. The last column of Table 4.5 shows difference in percentage between our data layer and Hibernate times. The formula used to find the difference in percentage is \( \text{% difference} = \frac{(X-Y)}{X} \times 100 \) where X values are Hibernate times and Y values are our data layer times. The result indicates a better performance of the proposed data layer.

<table>
<thead>
<tr>
<th>Functionalities</th>
<th>Data Layer (milliseconds)</th>
<th>Hibernate (milliseconds)</th>
<th>Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get All</td>
<td>8</td>
<td>12</td>
<td>33.33 %</td>
</tr>
<tr>
<td>Delete</td>
<td>3</td>
<td>13</td>
<td>76.92 %</td>
</tr>
<tr>
<td>Insert</td>
<td>5</td>
<td>94</td>
<td>94.68 %</td>
</tr>
<tr>
<td>Update</td>
<td>4</td>
<td>92</td>
<td>95.65 %</td>
</tr>
<tr>
<td>Get All (Filter/ HQL)</td>
<td>9</td>
<td>11</td>
<td>18.18 %</td>
</tr>
</tbody>
</table>

Table 4.5: Comparison Result

During the comparison with Hibernate, it was noticed that Hibernate need more time in set up and more coding in order to fulfil the same functionality as GIDL. Table 4.6 shows the difference in code between Hibernate and our data layer for Create functionality. Hibernate needs nearly five LOC (lines of code) more than the proposed data layer to complete each of the functionalities. Furthermore the proposed data layer provides more functionality with StoreList data structure compared with simple List that Hibernate uses. StoreList data structure has some extra functionality like get by key, get by code, caching, get by id for every business object. Although Hibernate provides a platform where software developers might work without knowing SQL, it still uses HQL (High Query Language) while the proposed data layer uses filter objects for querying. Also Hibernate needs XML files to map object with database tables and have problems with database relationship [3, 4].
### Data Layer

```java
Student stu = new Student();
stu.setName(name);
stu.setSurname(degree);
stu.setCourse(phone);
stu.setDate(date)

try {
    Student stu = new Student();
    Controller.getInstance().saveStudent(stu);
} catch (Exception e) {
    e.printStackTrace();
}
```

### Hibernate

```java
Student stu = new Student();
stu.setName(name);
stu.setSurname(degree);
stu.setCourse(phone);
stu.setDate(date)

try {
    sessionFactory = new Configuration().configure("com\hibernate.cfg.xml").buildSessionFactory();
} catch (Exception e) {
    System.out.println(e.getMessage());
}

Session s = sessionFactory.openSession();
Transaction tx = s.beginTransaction();
s.save(stu);
tx.commit();
if (s != null) 
s.close();
```

Table 4.6: Comparison Result in Coding
Chapter 5: Conclusion

This chapter presents the conclusion of this thesis. The goal of the thesis project is to propose a consistent approach for the defined problems in Section 1.2. The complete project took 6 months. In what follows there is a summary of objectives and their accomplishments. Then in Section 5.2 future work is shown followed by an evaluation of this thesis.

5.1 Summary

The summary is based on the five main objectives that were defined in Section 1.3.

O1. Literature Review on similar tools and theories.

O2. Define Software Requirements of data layer, pre and post condition.

O3. Design, develop and test the data layer for a given normalized database.

O4. Develop a software tool for generating the data layer.

O5. Comparing with one of the existing similar solutions (Hibernate)

The first objective was to make a Literature Review on similar tools and relevant theories. Chapter 2 presents all the similar tools and ORM theory is described in details. Furthermore Hibernate is described in details. This objective was straightforward and it was achieved by a lot of research in the ORM theories and tools. In the end the most important and relevant material was chosen to be presented. The aim of Literature Review is to help the researchers to have a good background on their research field. In this phase of the thesis a lot of knowledge was gained for defining better design of the data layer during the development phase.

The second objective was to define clearly all the software requirements. During Analysis phase of the project we defined all the requirements with pre and post conditions. Object Oriented Analysis and Design paradigm was chosen for presenting the requirements in a structured way. In the first sections of Chapter 3, all software requirements are presented specifically.

The third objective of this thesis has three sub-objectives which are design, develop and test the data layer but since this is a cycle process it is defined as one. Firstly, the design phase of the data layer is shown in Section 3.4. Then in Chapter 4 we present the implementation of the data layer for a given normalized database. Finally in Section 4.4 we present the testing phase of the data layer. The accomplishment of this cycling process means that the third objective is realized.

The fourth objective is to develop a tool for generating the data layer and it was quite straightforward. Section 3.2 and 4.2 present the work for accomplishing this objective. The idea was to develop an eclipse plug-in for generating a Java project and the results shown in Section 4.2 justify the achievement of this objective.

The last objective is to compare with Hibernate. After we defined the same software and hardware environment the comparison was done. Based on the comparison results shown in Section 4.5 of the proposed data layer compared with Hibernate we might say that CRUD functionalities are done faster. The generated data layer offers a lot of functions and a good environment for developing the domain model while using MVC architecture. This is done by StoreList data structure rather than using simple List. Based on the work we have done, the proposed data layer requires less coding than Hibernate for realizing the same functionality. The usage of Singleton Design Pattern and filtering objects makes our solution more Object Oriented than Hibernate.

Finally all the objectives were accomplished and the main aim of the thesis is realized.
5.2 Future Work

The proposed solution is a dynamic tool for generating data layer from given normalized databases. The characteristics of this thesis leave space for more future work. A highly important and straight forward work would be to extend GIDL to work for other database management tools like MySQL or Oracle. Another work would be to adapt and test GIDL with other drivers that are used nowadays. Up to now the data layer is tested with three databases however more testing could be done.

Secondly it would be recommended to use Observer Design Pattern for managing StoreList data structure. This would be helpful for building the main model as part of MVC architecture. Also using Generics instead of simple business objects for StoreList data structure would be helpful for developers although it did not show a good performance.

Testing phase for the data layer needs time and different databases so designing and building some standard database models would be good. A benchmark might be developed to check speed and connection time for every generated data layer.

Based on personal experience, it is recommended that the GIDL plug-in should be used when database managers have a normalized database with 10-80 tables in order to provide optimum results. Scalability is not done and discussed however we advice that size of the database should be less than 80 tables, because it is not tested for bigger databases. Checking scalability and how the data layer behaves with larger database is a future work.

5.3 Evaluation

The data layer was tested with three different databases and it is robust. GIDL tool is expected to help and facilitate the software development life cycle. Furthermore it can be used by programmers who do not have deep knowledge of SQL.

In Section 1.1 we specified that the proposed data layer should provide secure connections, efficiency and optimal result. These attributes of our solution were implemented and tested during implementation phase. DesEncrypter class enables password encryption for database connection properties. Efficiency was not easy to be defined and a lot of different versions of data layer were tested until we decide on the current solution. The different versions of data layer were built during an iterative cycling process. After the three base components (Controller, Transaction Manager and Database Layer) were designed and developed, extra features and properties were added to the data layer. We think that the proposed solution is efficient in terms of time and LOC (Line of Codes) based on comparison results in Section 4.5. It is hard to find optimal results of data layers because systems are dynamic and their requirements have a tendency to change. However we propose a general solution and add extra features and functions to the data layer explained in Section 4.3 which will facilitate the development of the main model. Our solution seemed to have optimal result for specific problems and normalized databases.

Finally we might say that the proposed data layer is good alternative solution for software developers.
References


Appendixes

In Appendix A all Class and System Sequence Diagrams are presented followed by Appendix B where source code of all important classes is presented. Finally in Appendix C all supervisor-supervisee meetings log are shown.

Appendix A: Class and System Sequence Diagram

Business Object and “com.server” package class diagram is presented in Figure A.1.
“db” package class diagram is presented in Figure A.2 in an extended version.

Figure A.2: Extended Database Layer Class Diagram
“util” package and classes are presented in the following class diagram.

![Util Class Diagram](image)

**Figure A. 3: Util Class Diagram**

Delete functionality is presented through system sequence diagram in Figure A.4.

![Delete System Sequence Diagram](image)

**Figure A. 4: Delete System Sequence Diagram**
Get Functionality is presented through system sequence diagram in Figure A.5.

Figure A. 5: Get System Sequence Diagram
Save Functionality is presented in Figure A.6
Appendix B: Source Code

The source code of all important classes is shown in the following tables.

```java
package bussinessobject;
import db.CONSTANTS;
import db.ObjectContainer;
import util.UtilDate;
import java.text.SimpleDateFormat;

/**
 * Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
 * User: user
 * Date: Aug 1, 2012 2:35:20 PM
 */

public class Student extends ObjectContainer {
    public final SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm");
    private String name;
    private String surname;
    private String course;
    private UtilDate date;
    public Student(int id) {
        setID(id);
    }

    public Student() {
        this(CONSTANTS.MINUS_ONE);
    }

    public String getName() {
        return name;
    }

    public void setName(String param) {
        this.name = param;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String param) {
        this.surname = param;
    }

    public String getCourse() {
        return course;
    }

    public void setCourse(String param) {
        this.course = param;
    }

    public UtilDate getDate() {
        return date;
    }

    public void setDate(UtilDate param) {
        this.date = param;
    }
}
```
```java
package filter;
import db.CONSTANTS;
import db.ObjectContainer;
import util.UtilDate;
import java.text.SimpleDateFormat;

/**
 * Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
 * User: user
 * Date: Aug 1, 2012 2:35:20 PM
 */

public class StudentFilter
{
    private int filterID = ObjectContainer.MINUS_ONE;
    public final SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm");
    private Integer iD;
    private String name;
    private String surname;
    private String course;
    private UtilDate date;

    public StudentFilter(int id) {
        setFilterID(id);
        filterID = id;
    }

    public StudentFilter() {
        filterID = ObjectContainer.MINUS_ONE;
    }

    public void setFilterID(int filterID) {
        this.filterID = filterID;
    }

    public boolean hasIDSet() {
        return (filterID != ObjectContainer.MINUS_ONE);
    }

    public Integer getID() {
        return iD;
    }

    public void setId(Integer param) {
        this.iD = param;
    }

    public String getName() {
```

Table B.1: Student Class

```java
```
Table B.2: StudentFilter Class

```java
package com.server;
import businessobject.Student;
import filter.StudentFilter;
import db.IDGeneratorManager;
import db.StoreList;
import db.DBAdapter;
import db.Transaction;
/**
* Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
* User: user
* Date: Aug 1, 2012 2:35:20 PM
*/
public class TransactionManager extends Transaction {
    public TransactionManager() {
    }
    public TransactionManager(Transaction transaction) {
        super(transaction);
    }
    public DBAdapter createDBAdapter() {
        return new DBLayer();
    }
}
```
public DBAdapter createDBAdapter(DBAdapter dbAdapter) {
    return new DBLayer(dbAdapter);
}

private DBLayer getDBLayer() {
    return (DBLayer) getDBAdapter();
}

//Student
public StoreList getAllStudents(StudentFilter filter) {
    StoreList objects = null;
    try {
        objects = getAllStudentsNoConnectionClosing(filter);
    } finally {
        attemptToClose();
    }
    return objects;
}

private StoreList getAllStudentsNoConnectionClosing(StudentFilter filter) {
    DBLayer umsDBLayer = getDBLayer();
    return umsDBLayer.getStudents(filter);
}

public void saveStudent(Student student) throws Exception {
    try {
        DBLayer dbLayer = getDBLayer();
        if (student.hasDefaultID()) {
            dbLayer.saveStudentData(student);
        } else {
            // updating
            dbLayer.updateStudentData(student);
        }
        attemptToCommit();
    } catch (Exception e) {
        attemptToRollBack(e);
    } finally {
        attemptToClose();
    }
}

public void deleteStudent(Student student) throws Exception {
    try {
        DBLayer dbLayer = getDBLayer();
        checkStudentForDelete(student);
        // delete object
        dbLayer.deleteStudent(student);
        attemptToCommit();
    } catch (Exception e) {
        attemptToRollBack(e);
    } finally {
        attemptToClose();
    }
}

private void checkStudentForDelete(Student student) throws Exception {
    getDBLayer().checkDeletionOfStudent(student);
}

//Student
Table B.3: TransactionManager Class

```java
package com.server;
import bussinessobject.Student;
import filter.StudentFilter;
import db.StoreList;
import db.DBApapter;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import util.UtilDate;
import java.sql.SQLException;
import java.util.ArrayList;

/**
 * Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
 * User: user
 * Date: Aug 1, 2012 2:35:20 PM
 */
public class DBLayer extends DBAdapter {

    private static final String DB_DATE_DD_MM_YYYY = "convert(
datetime, cast(day(getdate()) as varchar(2))+'/'+
cast(month(getdate()) as varchar(2))+'/'+
cast(year(getdate()) as varchar(4)), 103)";

    public DBLayer() {
        this(true);
    }

    public DBLayer(boolean isTransaction) {
        super(isTransaction);
    }

    public DBLayer(DBAdapter parentAdapter) {
        super(parentAdapter);
    }

    public StoreList getStudents(StudentFilter filter) {
        ResultSet rs = null;
        StoreList result = new StoreList();
        PreparedStatement preparedStatement = null;
        String sqlQuery = "SELECT * FROM Student WHERE 1=1";
        try {
            StudentQuery = "SELECT * FROM Student WHERE 1=1";
            if (filter.getID() != null) {
                sqlQuery += " AND " + " ID = " + filter.getID() + " ";
            }
            if (filter.getName() != null) {
                sqlQuery += " AND " + " Name LIKE " + filter.getName() + " ";
            }
            if (filter.getSurname() != null) {
                sqlQuery += " AND " + " Surname LIKE " + filter.getSurname() + " ";
            }
            if (filter.getCourse() != null) {
                sqlQuery += " AND " + " Course LIKE " + filter.getCourse() + " ";
            }
            try {
                StudentQuery = "SELECT * FROM Student WHERE 1=1";
                if (filter.getID() != null) {
                    sqlQuery += " AND " + " ID = " + filter.getID() + " ";
                }
                if (filter.getName() != null) {
                    sqlQuery += " AND " + " Name LIKE " + filter.getName() + " ";
                }
                if (filter.getSurname() != null) {
                    sqlQuery += " AND " + " Surname LIKE " + filter.getSurname() + " ";
                }
                if (filter.getCourse() != null) {
                    sqlQuery += " AND " + " Course LIKE " + filter.getCourse() + " ";
                }
            }
        }
    }
}
```
sqlQuery += " AND " + " Course LIKE '" + filter.getCourse() + "'");
        }
        if (filter.getDate() != null) {
            sqlQuery += " AND " + " Date >= '" + filter.getDate() + "'");
        }
        PreparedStatement preparedStatement = getConnection().prepareStatement(sqlQuery);
        System.out.println(sqlQuery);
        rs = preparedStatement.executeQuery();
        while (rs.next()) {
            student = new Student();
            student.setID(rs.getInt("ID"));
            student.setName(rs.getString("Name"));
            student.setSurname(rs.getString("Surname"));
            student.setCourse(rs.getString("Course"));
            student.setDate(new UtilDate(rs.getDate("Date"))));
            result.add(student);
        }
        try {
            rs.close();
            preparedStatement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return result;
    }

    public void saveStudentData(Student student) throws SQLException {
        String query = "INSERT INTO Student ( "+
                        " Name,"+
                        " Surname,"+
                        " Course,"+
                        " Date"+
                        " ) VALUES (?, ?, ?, ?)"");
        PreparedStatement pstmt = getConnection().prepareStatement(query);
        int i = 1;
        pstmt.setString(i++, student.getName());
        pstmt.setString(i++, student.getSurname());
        pstmt.setString(i++, student.getCourse());
        pstmt.setTimestamp(i++, student.getDate().toTimeStamp());
        pstmt.executeUpdate();
```java
public void updateStudentData(Student student) throws SQLException {
    String query = "Update Student set 
    " +
    " Name = ? ," +
    " Surname = ? ," +
    " Course = ? ," +
    " Date  = ?";

    query = query + " where ID = ?";

    PreparedStatement pstmt = getConnection().prepareStatement(query);
    int i = 1;
    pstmt.setString(i++, student.getName());
    pstmt.setString(i++, student.getSurname());
    pstmt.setString(i++, student.getCourse());
    pstmt.setTimestamp(i++, student.getDate().toTimeStamp());
    pstmt.setInt(i++, student.getID());
    pstmt.executeUpdate();
}
```

```java
public void deleteStudent(Student student) throws SQLException {
    PreparedStatement pstmt;
    String query = "delete from Student WHERE 
    " +
    " ID = ? ";

    pstmt = getConnection().prepareStatement(query);
    pstmt.setInt(1, student.getID());
    pstmt.executeUpdate();
}
```

```java
public void checkDeletionOfStudent(Student student) throws Exception {
    // TODO Auto-generated method stub
    // You have to check before deletion. IF the ID of the
    // object exist in other tables as foreign key.
    // The example shows how to do it if it exist You just
    // throw an exception.
    // PreparedStatement preparedStatement =
    // getConnection().prepareStatement(" Select Count(*) as cnt "+
    // "from o2CRMPotentialContacts where Category = ? ");
    // preparedStatement.setInt(1, contactCategory.getID());
    // ResultSet resultSet = preparedStatement.executeQuery();
    // resultSet.next();
    // if (resultSet.getInt("cnt") > 0) {
    //    throw new Exception("ContactCategory_Has_Been_Used_On_PotentialContact");
    // }
    // resultSet.close();
    // preparedStatement.close();
}
```

Table B.4: DBLayer Class
package util;
import java.io.IOException;
import java.io.PrintStream;
import java.io.UnsupportedEncodingException;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.AlgorithmParameterSpec;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.KeySpec;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.PBEParameterSpec;
import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;

/**
 * Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
 * User: user
 * Date: Aug 1, 2012 2:35:20 PM
 */

public class DesEncrypter
{
    Cipher ecipher;
    Cipher dcipher;
    byte[] salt = { -87, -101, -56, 50, 86, 53, -29, 3 };
    int iterationCount = 19;

    public DesEncrypter(String passPhrase)
    {
        try {
            KeySpec keySpec = new PBEKeySpec(passPhrase.toCharArray(), this.salt, this.iterationCount);
            SecretKey key = SecretKeyFactory.getInstance("PBEWithMD5AndDES").generateSecret(keySpec);
            this.ecipher = Cipher.getInstance(key.getAlgorithm());
            this.dcipher = Cipher.getInstance(key.getAlgorithm());
            AlgorithmParameterSpec paramSpec = new PBEParameterSpec(this.salt, this.iterationCount);
            this.ecipher.init(1, key, paramSpec);
            this.dcipher.init(2, key, paramSpec);
        } catch (InvalidAlgorithmParameterException e) {
        } catch (InvalidKeySpecException e) {
        } catch (NoSuchPaddingException e) {
        } catch (NoSuchAlgorithmException e) {
        }
    }
}
```java
public String encrypt(String str) {
    try {
        byte[] utf8 = str.getBytes("UTF8");
        byte[] enc = this.ecipher.doFinal(utf8);
        return new BASE64Encoder().encode(enc);
    } catch (BadPaddingException e) {
    } catch (IllegalBlockSizeException e) {
    } catch (UnsupportedEncodingException e) {
    } catch (IOException e) {
    }
    return null;
}

public String decrypt(String str) {
    try {
        byte[] dec = new BASE64Decoder().decodeBuffer(str);
        byte[] utf8 = this.dcipher.doFinal(dec);
        return new String(utf8, "UTF8");
    } catch (BadPaddingException e) {
    } catch (IllegalBlockSizeException e) {
    } catch (UnsupportedEncodingException e) {
    } catch (IOException e) {
    }
    return null;
}

public static void main(String[] args) {
    try {
        DesEncrypter encrypter = new DesEncrypter("isolutions");
        String decrypted = encrypter.encrypt("amir");
        System.out.println("decrypted = " + decrypted);
    } catch (Exception e) {
    }
}
```

Table B.5: DesEncrypter Class

```java
package util;
import java.util.HashMap;

/**
 * Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
 * User: user
 * Date: Aug 1, 2012 2:35:19 PM
 */

public class Cache
```
private HashMap objectsHashMap;
private static Cache cache;
public void resetCache(int identifier) {
    setCacheObject(identifier, null);
}
public void resetAllCaches() {
    setObjectsHashMap(null);
}
public Object getCacheObject(int identifier) throws Exception {
    return getCacheObject(identifier + "");
}
public Object getCacheObject(String identifier) throws Exception {
    Object o = getObjectsHashMap().get(identifier);
    if (o == null) {
        throw new Exception("Object_not_found_for_identifier" + " : ":
                        + identifier);
    }
    return o;
}
public Object setCacheObject(int identifier, Object cacheObject) {
    return setCacheObject(identifier + "", cacheObject);
}
public Object setCacheObject(String identifier, Object cacheObject) {
    getObjectsHashMap().put(identifier, cacheObject);
    return cacheObject;
}
public HashMap getObjectsHashMap() {
    if (this.objectsHashMap == null) {
        this.objectsHashMap = new HashMap();
    }
    return this.objectsHashMap;
}
public void setObjectsHashMap(HashMap objectsHashMap) {
    this.objectsHashMap = objectsHashMap;
}

Table B.6: Cache Class

package db;
import db.ObjectContainer;
import util.Cache;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
/**
 * Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
*/
public class StoreList extends ArrayList {
    public static final int descriptionHashMap = 10;
    public static final int codeHashMap = 20;
    public static final int idHashMap = 30;
    private static int keyHashMap = 31;
    public static final int codeArrayList = 40;
    public static final int descriptionArrayList = 50;
    public static final int idArrayList = 60;
    public static final int keyArrayList = 70;
    private transient Cache cache = null;

    public void setDirty() {
        getCache().resetAllCaches();
    }

    public StoreList() {
    }

    public StoreList(int initialCapacity) {
        super(initialCapacity);
    }

    public StoreList(Collection c) {
        super(c);
    }

    public boolean add(Object o) {
        boolean a = super.add(o);
        refreshCashe();
        return a;
    }

    public boolean remove(Object o) {
        boolean a = super.remove(o);
        refreshCashe();
        return a;
    }

    public void refreshCashe() {
        setDirty();
    }

    public ObjectContainer getByCode(Object code) {
        return (ObjectContainer)getCodeHashMap().get(code);
    }

    public ObjectContainer getDescription(Object description) {
        return (ObjectContainer)getDescriptionHashMap().get(description);
    }

    public ObjectContainer getID(int id) {
        return (ObjectContainer)getIdHashMap().get(Integer.toString(id));
    }
}
public ObjectContainer getByKey(Object key) {
    return (ObjectContainer)getKeyHashMap().get(key);
}

public HashMap getDescriptionHashMap() {
    HashMap hashMap;
    Iterator i$;
    try {
        return (HashMap)getCache().getCacheObject(10);
    } catch (Exception e) {
        hashMap = new HashMap();
        for (i$ = iterator(); i$.hasNext(); ) { Object o = i$.next();
            ObjectContainer entity = (ObjectContainer)o;
            hashMap.put(entity.getDescription(), entity);
        }
        return (HashMap)getCache().setCacheObject(10, hashMap);
    }
}

public HashMap getCodeHashMap() {
    HashMap hashMap;
    Iterator i$;
    try {
        return (HashMap)getCache().getCacheObject(20);
    } catch (Exception e) {
        hashMap = new HashMap();
        for (i$ = iterator(); i$.hasNext(); ) { Object o = i$.next();
            ObjectContainer entity = (ObjectContainer)o;
            hashMap.put(entity.getCode(), entity);
        }
        return (HashMap)getCache().setCacheObject(20, hashMap);
    }
}

public HashMap getKeyHashMap() {
    HashMap hashMap;
    Iterator i$;
    try {
        return (HashMap)getCache().getCacheObject(keyHashMap);
    } catch (Exception e) {
        hashMap = new HashMap();
        for (i$ = iterator(); i$.hasNext(); ) { Object o = i$.next();
            ObjectContainer entity = (ObjectContainer)o;
            hashMap.put(entity.getKey(), entity);
        }
        return (HashMap)getCache().setCacheObject(keyHashMap, hashMap);
    }
}

public HashMap getIdHashMap() {
    HashMap hashMap;
    Iterator i$;
    try {
        return (HashMap)getCache().getCacheObject(30);
    } catch (Exception e) {
        hashMap = new HashMap();
        for (i$ = iterator(); i$.hasNext(); ) { Object o = i$.next();
            ObjectContainer entity = (ObjectContainer)o;
        }
    }
}
hashMap.put(Integer.toString(entity.getID()), entity); } 
return (HashMap)getCach()e.setCacheObject(30, hashMap);

public ArrayList toCodeArrayList() 
{ 
ArrayList list; 
try { 
return (ArrayList)getCach()e.getCach()eObject(40); } 
catch (Exception e) { 
list = new ArrayList(size()); 
for (Iterator i$ = iterator(); i$.hasNext(); ) { Object o1 = i$.next(); 
ObjectContainer o = (ObjectContainer)o1; 
list.add(o.getCode()); } 
Collections.sort(list); 
return (ArrayList)getCach()e.setCacheObject(40, list); } 
public ArrayList toKeyIdArrayList() { 
ArrayList list; 
try { 
return (ArrayList)getCach()e.getCach()eObject(70); } 
catch (Exception e) { 
list = new ArrayList(size()); 
for (Iterator i$ = iterator(); i$.hasNext(); ) { Object o1 = i$.next(); 
ObjectContainer o = (ObjectContainer)o1; 
list.add(o.getKey()); } 
Collections.sort(list); 
return (ArrayList)getCach()e.setCacheObject(70, list); } 
public ArrayList toIdArrayList() { 
ArrayList list; 
try { 
return (ArrayList)getCach()e.getCach()eObject(60); } 
catch (Exception e) { 
list = new ArrayList(size()); 
for (Iterator i$ = iterator(); i$.hasNext(); ) { Object o1 = i$.next(); 
ObjectContainer o = (ObjectContainer)o1; 
list.add(Integer.toString(o.getID())); } 
Collections.sort(list); 
return (ArrayList)getCach()e.setCacheObject(60, list); } 
public ArrayList toDescriptionArrayList() { 
ArrayList list; 
try { 
return (ArrayList)getCach()e.getCach()eObject(50); } 
catch (Exception e) {
list = new ArrayList(size());

for (Iterator i$ = iterator(); i$.hasNext(); ) { Object o1 = i$.next();

    ObjectContainer o = (ObjectContainer)o1;
    list.add(o.getDescription());
} 
Collections.sort(list);
} return (ArrayList)getCach().setCacheObject(50, list);

public Cache getCach()
{
    if (this.cache == null) this.cache = new Cache();
    return this.cache;
}

public void setCache(Cache cache) {
    this.cache = cache;
}

public boolean hasCode(String code) {
    return getByCode(code) != null;
}

Table B.7: StoreList Class

package db;

/***
 * Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
 * User: user
 * Date: Aug 1, 2012 2:35:20 PM
 */

public class ObjectContainer extends DBEntity {
    protected String Code = "";
    protected String Description = "";
    protected boolean isActive = true;

    public String getCode() {
        return this.Code.toUpperCase();
    }

    public void setCode(String code) {
        this.Code = code;
    }

    public String getDescription() {
        return this.Description.toUpperCase();
    }

    public void setDescription(String description) {
        this.Description = description;
    }
public Object getKey() {
    return getID() + "";
}

public String toString() {
    return getDescription();
}

public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof ObjectContainer)) return false;
    if (!super.equals(o)) return false;

    ObjectContainer that = (ObjectContainer)o;

    if (!this.isActive.equals(that.isActive)) return false;
    return this.Description.equals(that.Description); // null check
}

class ObjectContainer {

    public boolean isActive() {
        return isActive;
    }

    public void setActive(boolean active) {
        isActive = active;
    }

    // The rest of the class remains unchanged

    // Table B.8: ObjectContainer Class
}

package db;
import util.UtilJOptionPane;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
import java.util.StringTokenizer;

/**
 * Created by GIDL GENERATOR INTELLIGENT DATA LAYER.
 * User: user
 * Date: Aug 1, 2012 2:35:20 PM
 */

public class JDBCConnection {
    public static String SQLDriver = "com.microsoft.jdbc.sqlserver.SQLServerDriver";
    public static String SQLDriver2005 = "com.microsoft.sqlserver.jdbc.SQLServerDriver";
    public static String SQLDriverJDTS =

    // The rest of the class remains unchanged
public static String MySQLDriver = "org.gjt.mm.mysql.Driver";
public static String MSACCESS = "sun.jdbc.odbc.JdbcOdbcDriver";
private String myURL;
private String dbName;
private String serverName;
private String instance;
private String serverNameURL;
private String instanceName;
private String userName;
private String pass;
private String driver;
public static final String INSTANCE_DELIMITER = "\\";
private boolean noDialogs;
public JDBCConnection(String dbName, String pServerName, String userName, String pass, String driver, String instance) {
    setDriver(driver);
    setDbName(dbName);
    setServerNameURL(pServerName);
    StringTokenizer stringTokenizer = new StringTokenizer(pServerName, "\\", false);
    setServerName(stringTokenizer.nextToken());
    if (stringTokenizer.hasMoreTokens()) {
        setInstanceName(stringTokenizer.nextToken());
    }
    setUserName(userName);
    setPass(pass);
    if (driver.equals(SQLDriver))
        setMyURL("jdbc:microsoft:sqlserver://" + getServerNameURL() + ":1433");
    else if (driver.equals(SQLDriver2005))
        setMyURL("jdbc:sqlserver://" + getServerNameURL() + ":1433");
    else if (driver.equals(SQLDriverJDTS))
        setMyURL("jdbc:jtds:sqlserver://" + getServerName() + ":1433/" + dbName + "/:3306");
    else if (!driver.equals(MYSQLDriver)) {
        if (driver.equals(MSACCESS)) {
            setMyURL("jdbc:odbc:o2Store");
        }
    }
    else
        setMyURL("jdbc:mysql://" + getServerNameURL() + "/" + dbName + ":3306");
}

public Connection getDBConnection() {
    try {
        Class.forName(this.driver);
        try {
            Connection conn = getConnectionNoPool();
            return conn;
        }
    } catch (SQLException e) {

showMessageDialog("SQL_SERVER_Connection_FAILED");
getMyURL() + " " + getUserName());
e.printStackTrace();
}
}
catch (ClassNotFoundException e) {
e.printStackTrace();
showMessageDialog(e.toString()+"Driver of Database was not found!");
}
return null;
}

private void showMessageDialog(String msg) {
if (isNoDialogs()) {
return;
}
UtilJOptionPane.showMessageDialog(null, msg);
}

private Connection getConnectionNoPool() throws SQLException {

Properties p = new Properties();
p.setProperty("DatabaseName", getDbName());
p.setProperty("Password", getPass());
p.setProperty("User", getUserName());
p.setProperty("SelectMethod", "cursor");
if (hasInstanceName()) {
p.setProperty("instance", getInstanceName());
}

return DriverManager.getConnection(getMyURL(), p);
}

public String getMyURL() {
return this.myURL;
}

public void setMyURL(String myURL) {
this.myURL = myURL;
}

public String getUserName() {
return this.userName;
}

public String getPass() {
return this.pass;
}

public String getDbName() {
return this.dbName;
}

public void setDbName(String dbName) {
this.dbName = dbName;
}

public String getServerName() {
    return this.serverName;
}

public void setServerName(String serverName) {
    this.serverName = serverName;
}

public String getDriver() {
    return this.driver;
}

public void setDriver(String driver) {
    this.driver = driver;
}

public void setUserName(String userName) {
    this.userName = userName;
}

public void setPass(String pass) {
    this.pass = pass;
}

public String getInstanceName() {
    return this.instanceName;
}

public void setInstanceName(String instanceName) {
    System.out.println("setInstanceName instanceName = " +
    instanceName);
    this.instanceName = instanceName;
}

public boolean hasInstanceName() {
    String pInstanceName = getInstanceName();
    return (pInstanceName != null) &&
    (!pInstanceName.trim().equals(""));
}

public String getServerNameURL() {
    return this.serverNameURL;
}

public void setServerNameURL(String serverNameURL) {
    this.serverNameURL = serverNameURL;
}

public void setNoDialogs(boolean pNoDialogs) {
    this.noDialogs = pNoDialogs;
}

public boolean isNoDialogs() {
    return this.noDialogs;
}
public void setInstance(String instance) {
    this.instance = instance;
}

public String getInstance() {
    return instance;
}
Appendix C: Supervisor-Supervisee Meeting Logs
All meeting logs with the supervisor are presented in the following tables.

<table>
<thead>
<tr>
<th>Meeting Date</th>
<th>Student</th>
<th>Supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>02/04/2012</td>
<td>Amir Buzo</td>
<td>Antonina Danylenko</td>
</tr>
</tbody>
</table>

**Issues Discussed:**
- Presenting idea.
- Guidelines on research, ideas and ORM
- Guidelines to check similar system and compare

Table C-2: Meeting Log 04/02/2012

<table>
<thead>
<tr>
<th>Meeting Date</th>
<th>Student</th>
<th>Supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>02/04/2012</td>
<td>Amir Buzo</td>
<td>Mathias Hendenborg</td>
</tr>
</tbody>
</table>

**Issues Discussed:**
- Presenting project to the class and Mr. Hendenborg?
- Guidelines on writing project

Table C-3: Meeting Log 21/02/2012