The Dell Report

-

Subject	::	2th Semester Project ++ Dell ++
Date	::	08-05-2015
Group	::	05
Authors	::	Bancho Petrov
		Athinodoros Sgouromallis
		Bo Vilstrup Mortensen

Table of Contents

1 Introduction

1.0 Foreword

Default text Default text Default text Default text Default text Default text Default text Default text Default text

2.0 Introduction

The main objective is to create a Web application that is able to handle and document advertisement campaigns that Dell runs in different countries in the Nordic Area, and shrink the workload for Dell’s managers.
It is a platform that partners can go and start a project and request funding, presenting their idea and providing essential information like starting and finishing dates and a description. Then the region manager can read it and make comments if necessary and approve it or reject it.
There is a communication tool in every project that keeps track of the conversation so the user can go back in time and read everything without having to search in emails which is the way it works in the current moment.

There is an upload function in each project so proof-of-execution can be uploaded from the Partners side and be controlled by the supervisor of the project and if the proof that the project was executed as intended the project can be marked as finished and the fund is transferred to the Partner through e-banking or external services.

So the problems solved by using this web application are:

- One place for all users to communicate and act with.
- Book keeping to some extent.
- Easy overview of projects running.
- Partners are now responsible to start a project.
- Better and easier supervision of a vast workload for managers.
- Easier proof of execution handling (No more attached files in a mail :P).

3.0 Case description

4.0 Problem analysis

5.0 Reading guide

2 Product Report

1.0 Product Backlog

2.0 Description of the Program

3.0 Key Features

3.1 Naming Convention Class

At the beginning of our work, we decided that we are going to need some kind of a naming convention interface that would be implemented by most of the classes in the project to prevent bugs, caused by spelling mistakes. Upon further consideration, however, we discovered that this interface was mostly going to be used in the top layer of our program to coordinate the interaction between the Servlet and the .jsp files, and that was a problem, because .jsp files cannot implement interfaces. They can, however, use the import statement to import classes. That is why we agreed that instead of an interface, we should use a plain old class for this purpose. A class, which serves as something like a pseudointerface. The only reason it remains placed in the layer2.domain.interfaces package, instead of being moved to the first layer is because, as I previously mentioned, it is heavily referenced in the majority of the .jsp files, and even if we use NetBeans' refactor move option, we would still have to change all of the import statements in the .jsps by hand, and that is something that we cannot afford to do, due to time constraints.

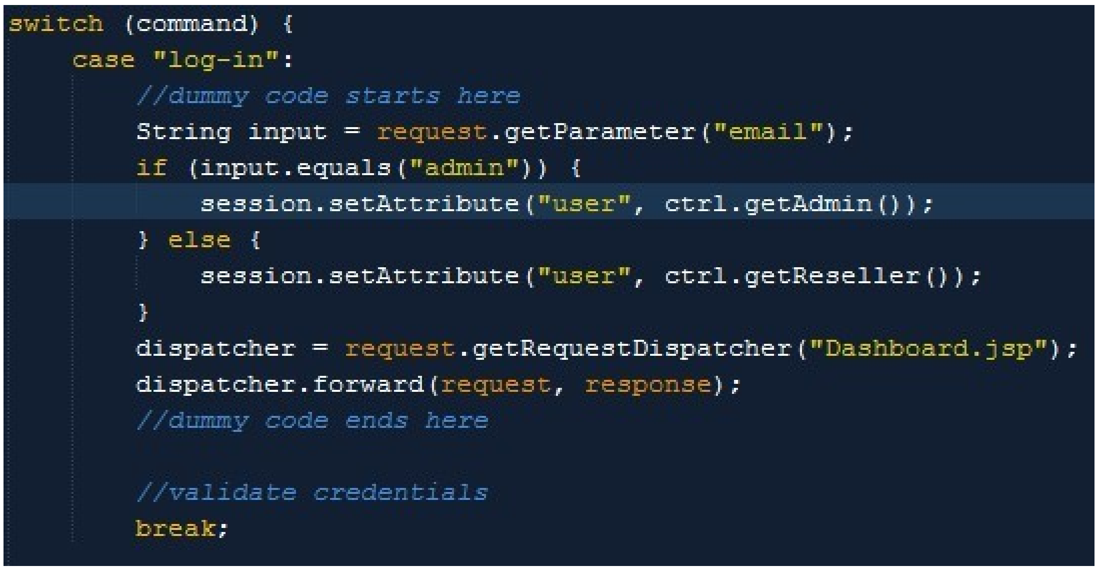
3.2 Log-In and Security

Since we were going to have at least two different types of users, using the product, and since developing a log-in system that actually checks for someone's username and password seemed like a rather lengthy process, we decided that, at least until we're done with the more important user stories first, we would need some kind of a dummy log-in system that would easily allow us to log-in and test the system.

The idea I came up with is the following: In order for us to log-in as an admin (dell user), we would have to type "admin" in the email field [*Note: At this point, we were still using emails for logging-in, which we later replaced with a username] and just some random characters in the password field. And in order for us to log-in as a partner of dell, we would just type random characters in both input fields and get in.

It works the following way. First, we manually populated the database with one tuple of each of the two different types of users we would have, via INSERT INTO queries. Then, we wrote the code, so that once the log-in button on the index page is pressed, the Servlet would only check for the email parameter. If it was “admin”, it would start the chain calling of the dummy getAdmin() methods which went through the Controller class all the way down to the dataSource layer. And if the email parameter was not “admin”, it would call the dummy getReseller() methods. Once in the dataSource layer (the DBFacade, to be more precise), the correct userID’s were hard-coded and passed as arguments for the mappers to use and retrieve the needed user.

*Note: At this time, we were still using the “reseller” terminology for dell’s partners, which was later dropped.



```
switch (command) {
    case "log-in":
        //dummy code starts here
        String input = request.getParameter("email");
        if (input.equals("admin")) {
            session.setAttribute("user", ctrl.getAdmin());
        } else {
            session.setAttribute("user", ctrl.getReseller());
        }
        dispatcher = request.getRequestDispatcher("Dashboard.jsp");
        dispatcher.forward(request, response);
        //dummy code ends here

        //validate credentials
        break;
```

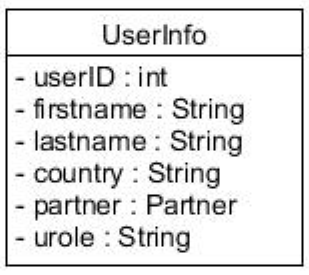
Illustration 1: Dummy Log-In [Note: At the time this screenshot was taken, we were still using the email for logging-in]

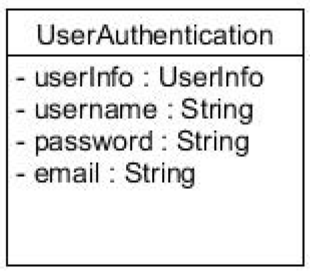
One week before the project deadline, however, we determined that it was time to handle logging-in the proper way, so we dropped this dummy implementation. Now when a user tries to log-in, the following takes place:

1. The program checks if a user with the entered username exists in the database.
2. If so, it checks if the entered password corresponds to the one stored for this user.
3. Then, if all went well, the log-in has been successful.

We are aware of the fact that this is in no way a secure log-in as we are breaking the first rule of password storing, which is never to store non-hashed values of passwords anywhere and that's exactly what we're doing (by storing them in the database). However, time constraints were once more the thing that made us decide not to implement a hashing algorithm for the passwords.

Nevertheless, one of the things we did concerning security was to split the user information into two different tables – one that stores the general information for the user and one that stores his log-in credentials and more sensitive information (e.g. email), called respectively UserInfo and UserAuthentication. That way, when we create an object of the type UserInfo and we use it throughout the program, we don't carry around all the sensitive information with us.





3.3 Java Beans

The way we create and use java beans is through the JSP syntax :

```
<jsp:useBean id=”” class=”” score=”” />
```

- The ID Attribute is the name of the new Object.
- The class Attribute is the import off the JAVA class so path has to be defined there
- And the scope defines the scope you want your object to be set

As the line of code only creates an empty object (Bean), its attributes have to be set.
And the other option is to set them one by one like this :

```
<jsp:setProperty name="" property="title" />
```

- The name attribute has to match the Id of the bean

If the bean has a set method for an attribute called title it will have the name like :

```
setTitle(String title)
```

So when the jsp page compiles to a servlet page, and let's say the name of the bean is book, it will be :

```
Book.setTitle(request.getAttribute("title"));
```

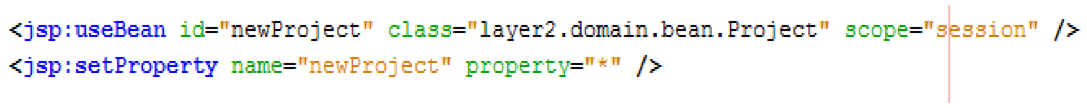
And title has to be the name of the input field in a form for example.

There is the option to set them all in one go like this:

```
< jsp:setProperty name="" property="*" />
```

Then it will go through the attributes in the request and see if there are matching setter methods, and if there are, it will set them all in the same line.

All the forms in this project (excluding the file upload form) are using this syntax to create a new bean and store it in the session object. A good example would be the newProjectHandler (illustration 2) where a lot of information from a form is placed in a bean in just two lines :



```
<jsp:useBean id="newProject" class="layer2.domain.bean.Project" scope="session" />
<jsp:setProperty name="newProject" property="*" />
```

Illustration 2: creating a new bean with the name newProject and storing it to the session)

4.0 Design and architecture

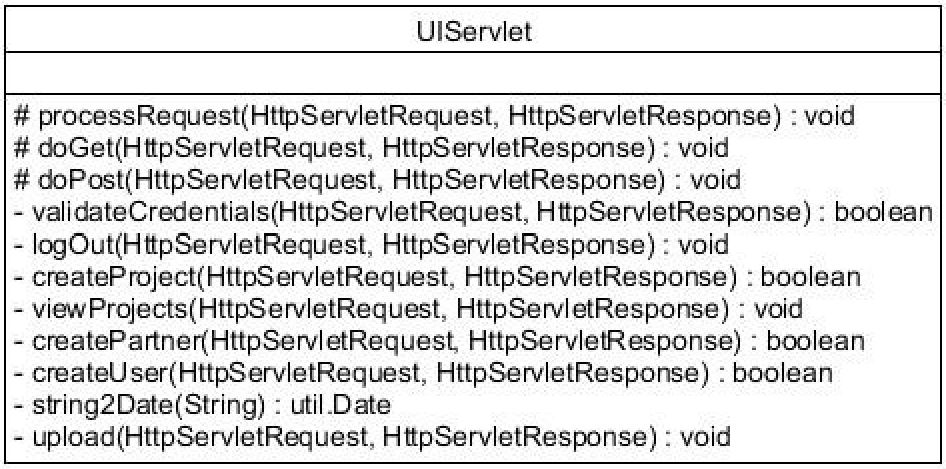
4.1 Three principal layers

Default text Default text Default text Default text Default text Default text Default text Default text Default text

4.2 The GUI - HTML, CSS and JSP

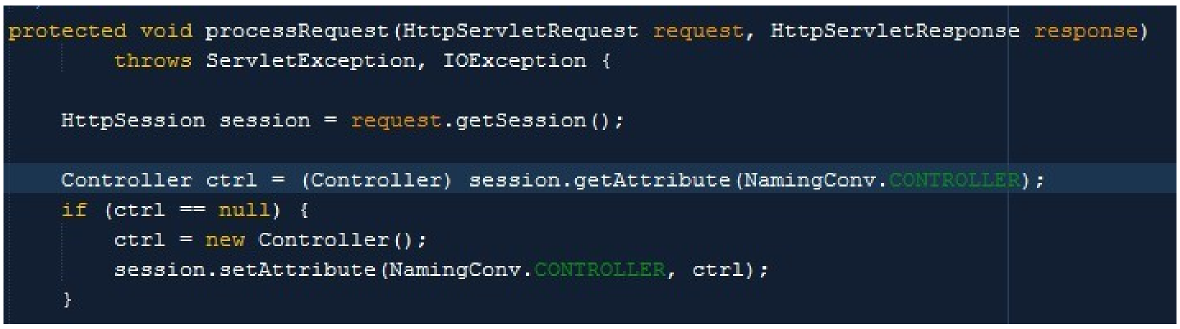
4.3 Servlet

Since our application was not going to be too big, we decided to have one servlet that handles all requests. In this manner, we would have a centralized code-flow, while still keeping the code in the servlet manageable.


UIServlet

processRequest(HttpServletRequest, HttpServletResponse) : void
doGet(HttpServletRequest, HttpServletResponse) : void
doPost(HttpServletRequest, HttpServletResponse) : void
- validateCredentials(HttpServletRequest, HttpServletResponse) : boolean
- logOut(HttpServletRequest, HttpServletResponse) : void
- createProject(HttpServletRequest, HttpServletResponse) : boolean
- viewProjects(HttpServletRequest, HttpServletResponse) : void
- createPartner(HttpServletRequest, HttpServletResponse) : boolean
- createUser(HttpServletRequest, HttpServletResponse) : boolean
- string2Date(String) : util.Date
- upload(HttpServletRequest, HttpServletResponse) : void

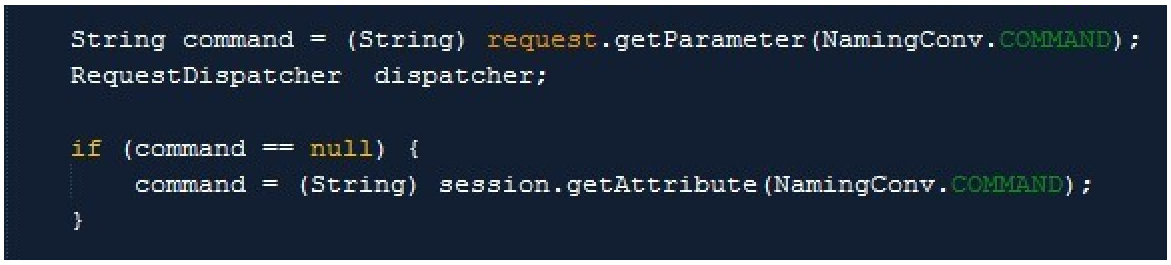
For the implementation, we relied on the examples, given to us before the project start. The first thing we do whenever a request is sent to the servlet and into the processRequest() method is to attempt to retrieve the Controller object from the session. If no such object exists, we create one and set it as an attribute of the session, so we can access it later as long as the same user is still logged-in. Later, we realized that we don't need to do that because we don't actually store any user-related information in the Controller, so we could just make a new one every time the processRequest() is called, but by the time we had this realization, we had already finished coding.


```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    HttpSession session = request.getSession();

    Controller ctrl = (Controller) session.getAttribute(NamingConv.CONTROLLER);
    if (ctrl == null) {
        ctrl = new Controller();
        session.setAttribute(NamingConv.CONTROLLER, ctrl);
    }
```

The next thing that happens is we try to retrieve the command attribute from the request object. If it's not there, then we take it from the session, because we've constructed the .jsp pages in such a way that whenever a call to the Servlet is made, there is always a command attribute either in the request, or in the session.



```
String command = (String) request.getParameter(NamingConv.COMMAND);
RequestDispatcher  dispatcher;

if (command == null) {
    command = (String) session.getAttribute(NamingConv.COMMAND);
}
```

After that, we have two big nested switch-cases. Our logic here is the following. If the user has pressed one of the sidebar menu buttons, then all the application would have to do is reload the main area of the dashboard with a new page. Hence, in this case we set the command to be “reloadMain” and we also add another attribute to the request, called “mainArea”, which holds information on which .jsp should be shown in the main area. Anything else the user wants to do is a different command and is handled by corresponding methods.

4.4 Controller

Default text Default text Default text Default text Default text Default text Default text Default text Default text

4.5 The Facade

The DBFacade class provides a mean to separate the data source layer from the domain layer. The class uses the singleton pattern so only one instance of the class can exist. The idea behind the facade is to provide a separation of the layers so it will be easier to swap out the existing data source layer with a new data source layer, in case of, a new database is chosen for the existing program.

4.6 The Connection

The DBConnection class provides a connection to the database. The class uses the singleton pattern to insure that it always gives back the same connection every time a class or method asks for one. Besides, having the responsibility of providing a connection to the database, the DBConnection also has the responsibility to close a connection, if a user decides to logout of the program. We added one more feature to the class, namely, a possibility to switch databases. This feature was add so we could switch between the production database and the test database. A much needed feature while coding the Junit test.

4.7 Singleton

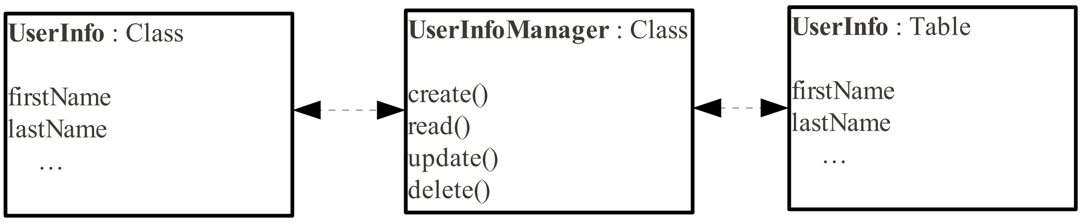
The singleton pattern is a way to ensure that an object can only be instantiated once. when a singleton is instantiated it will not be garbage collected again before the program is terminated. besides the advantage of only having one instance of an object, if that is what you need, a singleton also provides a lazy instantiation. In our program we use the singleton pattern in the DBConnector and the DBFacade, but the singletons are not implemented in the same way in the two classes. One is thread safe the others is not. either way, the two implementations works nearly in the same way.

4.8 Data Mapper

The data mapper pattern is used for mapping or managing data in a way so it is possible to take Plain Old Java Objects from memory and store these objects on a persistent storage like for example a database. Because of Java's way of handling data is different from how a database handles data, we need some kind of mapping between these two. As you know, Java uses object and databases uses tables. But the mapping between objects and tables is not the only problem you are facing while dealing with Java and databases. The problem is that Java does not know anything about databases and databases do not know anything about Java. So to deal or overcome this problem, you have to import a library like JDBC (Java DataBase Connectivity). When the JDBC API (application programming interface) is imported into Netbeans, then it becomes possible for Java to "talk" to a database and visa versa.

SQL (Structured Query Language) is the language databases "talk". A SQL query or statement is a statement which does something in the database, for example deletes or inserts a row of data in a table. In our program, all tables in the database have a corresponding mapper class which in our case is called a manager. The manager class responsibility is to map a specific Java class, for example a UserInfo bean class, to its corresponding UserInfo table in the database. We have 7 tables in our database and 7 manager classes to handle the mapping of these tables. Nearly every manager class has at least the 4 CRUD (create, read, update and delete) methods. The CRUD methods provide the basic functionality we need, so we can handle data between our program's front-end

(GUI) and our database, the back-end. The mapping between Java objects and the tables in the database is illustrated below:



A concrete example of how the mapping is coded and how it is working in our program is shown and described below:

```java
public boolean delete(Connection conn, int userID) {

    String sql = "DELETE FROM userInfo WHERE userID = ?";

    try (PreparedStatement stmt = conn.prepareStatement(sql);) {

        stmt.setInt(1, userID);
        int effected = stmt.executeUpdate();

        if (effected == 1) {
            return true;
        } else {
            return false;
        }

    } catch (SQLException e) {
        DBConnector.processException(e);
        return false;
    }
} // End of method :: Delete()
```

The example snippet above is taken from the userInfoManager class. This delete method takes two parameters, namely, a connection to the database and a unique userID. The userID points to a unique row in the userInfo table in the database, that we want to delete. The sql string “Delete FROM userInfo Where userID = ?” is the actually SQL query we want to execute. The question-mark in the SQL query is a placeholder for the parameter, userID. To prepare the execution of this query we use the prepareStatement() method provided by the JDBC API. The JDBC API will translate the Java code into a “real” SQL query which databases can understand. The PrepareStatement() method is executed inside a try-with-resource block. This insures, first and foremost, that if the call to the database somehow fails, then the database will throw an exception that will be caught by the catch block and the delete() method will return false. Second, the try-with-resource block also insures that any resources which was used or opened by the JDBC API while the communication with the database was ongoing, will be handled in a proper way, for example closed. If everything went well in the try-with-resource block, we end up with a prepare statement object. The prepare statement object stmt has a method called setInt() which takes two parameters. The first parameter in the setInt() method is pointing or referring to the place in the SQL query where there is a question-mark. Because there is only one question-mark in the SQL query, the value (1) points to the first, and in this case the only, question-mark. The second parameter the setInt() method takes the integer value of the userID, and places this value in the SQL query where the first parameter (1) points to. When everything is set up you can execute the executeUpdata() method, which actually and finally executes the SQL query. If everything went well, then the row which the userID points to will be deleted, and the integer variable effected will be 1, because one row was deleted, otherwise the value will be zero.

5.0 Testing

Programmers are often in a situation where they have written some or more lines of code, that they want to test. Do the code work as intended or not?. If you, as a programmer, find yourself in a situation as just described above, where you as a programmer, over and over again, are writing test code or methods in some unappropriated places in your source code just to see if your code is working or not, then it is probably the right time for you to switch your testing habits, and start using a test -environment or -framework, like JUnit or ngTest.

5.1 Unit Testing

Junit is a part of NetBeans' test framework and unit testing is about testing code. To test if some code is working or not, the following 3 steps are normally needed to be set up in the unit test environment before the actual test or test-method is executed.

1. **Setting up preconditions**
   - For example: establish a connection to a database.
2. **Setting up or resetting test environment before every new test**
   - For example: deleting all rows in all tables in a database.
3. **Setting up test data**
   - For example: creating objects to be inserted in a table in a database.

It is a good practice to see if the preconditions are met before evaluating the actual test result. If the preconditions is not met the test should fail.

There are no good excuses not to write tests for your programs, but there can be many good reasons to write a test in different ways and for various reasons, here are some:

1. **Write your test before the code is written.**
   This approach makes you thing about what your code has to do before you actually code the class or method.
2. **Write a test for an existing or newly written class or method.**
   This approach is probably the approach you choose if you don't no how to code a class or method. This gives you the change to understand the problem with the code before you write the actual test.
3. **Write a test for an existing program.**
   This approach is better then no test, but should be avoided.

Premature optimizations is a bad programming practice, and that goes for testing as well. Do not over test your code to begin with, let your tests evolve over time. As your code base starts to settle, you can begin refactoring your test code in the following order:

1. **Refactor the test code for more consistency**
2. **Collect all tests in a test-suite.**
3. **Enhance the test code with stress and false condition tests.**

Not everything about testing is good, here are some pros and cons:

Cons

1. If the code base is not settled, then you will end up spending a lot of time changing or refactoring your test code.
2. If you do not know have to write tests, then you will end up spending a lot of time trying to figure out have to make a good test environment.

Pros

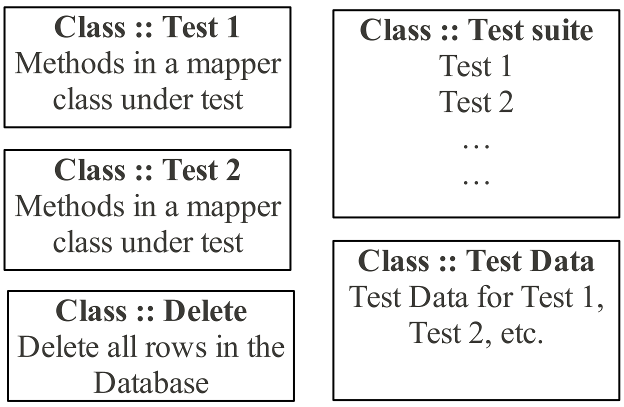
1. If you have a working test you can test if code changes breaks other parts of your program before committing these changes to the main repository.

5.2 Unit testing in action

When we started our work on the project, unit-testing was not a part of the discussion or even in the planing process. But when we began coding the mapper/manager classes for the project we realized that we needed some way to test these mapper classes, for two reason, namely:

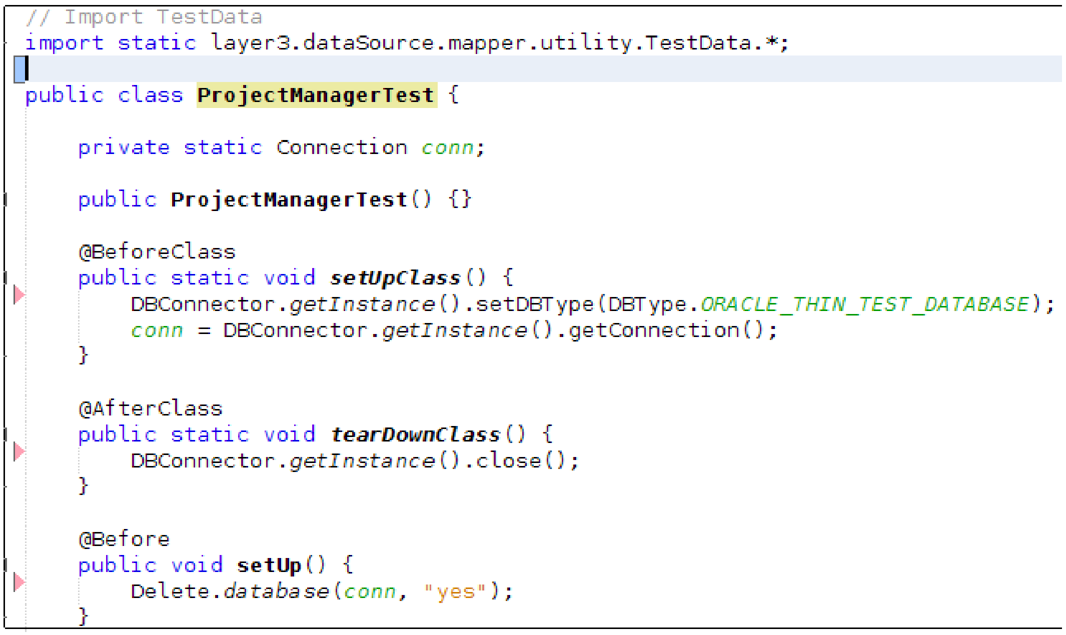
1. Lack of a front end (GUI), that could communicate with these mapper classes
2. lack of feedback from the compiler in case of errors in the mapper classes/database

Setting up a test environment seemed, at that time, to be a good solution to these two problems. By setting up a test environment, we could test the basic CRUD (Create, Read, Update and Delete) method in the mapper classes without having a GUI, and at the same time debug the mappers, if needed, in a more controlled and consistent way. After a lot of "try and errors" we came up with a test environment like this simplified example:


Class :: Test 1
Methods in a mapper class under test
Class :: Test 2
Methods in a mapper class under test
Class :: Delete
Delete all rows in the Database
Class :: Test suite
Test 1
Test 2
…
…
Class :: Test Data
Test Data for Test 1, Test 2, etc.

Every mapper class in the project has a counterpart in the form of a test class. And every test class is placed in the test suite. The test suite can then be run to see if all the tests can pass. The test data is placed in a class for it self so it can be imported into the test classes. The static delete method is placed in a class for it self. The idea of the delete method is to delete all rows in the database before a test is executed, this gives us a “clean” database before every test. A test database was used instead of the production database to make the tests more reliable and because we didn't wanted to interfere with the production database. By adding a test database every test can be executed in a consist manner.

Here is an example of a test set up of the projectManagerTest class. This set up is used for all our mapper test classes:



```
// Import TestData
import static layer3.dataSource.mapper.utility.TestData.*;

public class ProjectManagerTest {

    private static Connection conn;

    public ProjectManagerTest() {}

    @BeforeClass
    public static void setUpClass() {
        DBConnector.getInstance().setDBType(DBType.ORACLE_THIN_TEST_DATABASE);
        conn = DBConnector.getInstance().getConnection();
    }

    @AfterClass
    public static void tearDownClass() {
        DBConnector.getInstance().close();
    }

    @Before
    public void setUp() {
        Delete.database(conn, "yes");
    }
```

The code snippet above shows the following in order:

- **TestData.***

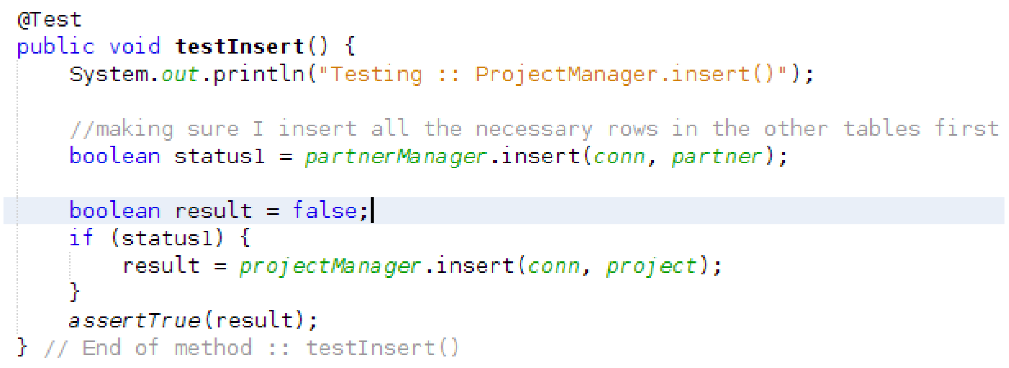
  Import of static test data. For example the Java classes (beans) we want to map and insert in the database.
- **setUpClass()**

  Is executed before any tests are run. The setUpClass sets up the connection to the test database.
- **tearDownClass()**

  Is executed after all the tests are run. The tearDownClass closes the connection to the test database
- **setUp()**

  Is executed before a test is run. The setUp Deletes all rows in the database before a test is run.

The code snippet below shows a simple test method from the projectManagerTest class, where we insert a row (a project) in the table in the database:



```
@Test
public void testInsert() {
    System.out.println("Testing :: ProjectManager.insert()");

    //making sure I insert all the necessary rows in the other tables first
    boolean status1 = partnerManager.insert(conn, partner);

    boolean result = false;
    if (status1) {
        result = projectManager.insert(conn, project);
    }
    assertTrue(result);
} // End of method :: testInsert()
```

The code above “serves”, more or less, as a template for more advanced tests. The code works as follows. First we insert a partner in the database, because a project needs a partner in the database before a project can be inserted, due to the dependency between a partner table and a project table in the database. if the partner table is inserted without errors, a project will be inserted in the database, and if not, the test will fail. if both tables is inserted correctly the test will pass.

5.3 Problems and Shortcomings

While writing the tests we encountered some problems. At first we thought that all test methods in a test class was executed in order, they were not. all tests methods are actually executed out of order. That came as a surprise and we had to rewrite some of the tests that we had already written. Later on, the out of order execution, help us finding a problem with the structure of the database. so in the end we ended up embracing the idea of this out of order execution of the test methods.

We didn't have time to make any enhanced testing, like stress and false condition tests. Neither did we have time to make any real descriptions of have to test the code manually.

6.0 The Database

6.1 Database

6.2 Normalization of the Database

7.0 Conclusion

8.0 Errors and shortcomings

3 The Process

1.0 Scrum

2.0 Git

3.0 Conclusion

This Project was a great way to force students to use all the things that has been taught throughout the semester. It was so stressful, but still so helpful, as the only thing we need is a push to learn new technologies.

Here is some cons and pros:

Cons

- We did not know enough to start the project. We knew a little about everything, but with no deep understanding of the process. We did not have enough confidence in ourselves.
- Answers from teachers on the same question where not matching causing confusion and uncertainty to the group
- The lack of a scrum master was a big problem at least in our group. A scrum master should be assigned by the teachers or a suggestion to rotate the role of a scrum master among the team members could be nice.

Pros

- We did not know enough to start the project, BUT we had been shown the direction and that was good enough. This is a counter point for the first con. : When you figure out a solution on your own you feel ten times better than copying something someone else have shown you in class, and if you do figure out or find a solution to problem by yourself it is most likely that you will never forget it.
- We started reading articles on the net and learn about finding solutions and that is one of the most useful skills we could have when we go out there trying to find a

job, and that skill is to be adaptive and open to new technologies.

- When the group works good even if you fill you are stack you know can rely on your fellow group members.

Suggestion for next go

There should be an agreement between teachers so that everyone will be on the same page and if that is not possible there should be one teacher per subject like : database guidance, html/css, Junit testing, general syntax problems, UML etc. This way less confusion will be caused to students.

More often meetings with the group and the product owner so that we do not get of tracks and lose a lot of precious time.