The Dell Report

-

Subject	::	2th Semester Project ++ Dell ++
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 Default text

2.0 Introduction

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

3.0 Case description

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

4.0 Problem analysis

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

5.0 Reading guide

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

2 Product Report

1.0 Product Backlog

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

1.1 Product backlog 1

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

2.0 Description of the Program

2.1 Description 1

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

2.2 Description 2

Default text Default text Default text Default text Default text Default text Default text Default text Default text.

2.3 Description 3

Default text Default text Default text Default text Default text Default text Default text Default text Default text.

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]

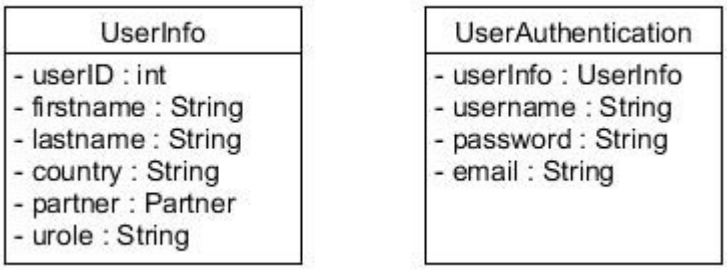
One week before the project deadline, however, we determined that it was time to handle logging-in the proper way, so we dropped this dummy implementation. Now when a user tries to log-in, the following takes place:

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.



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 Default text

4.2 The GUI - HTML, CSS and JSP

4.3 Servlet

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

4.4 Controller

Default text 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

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

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.

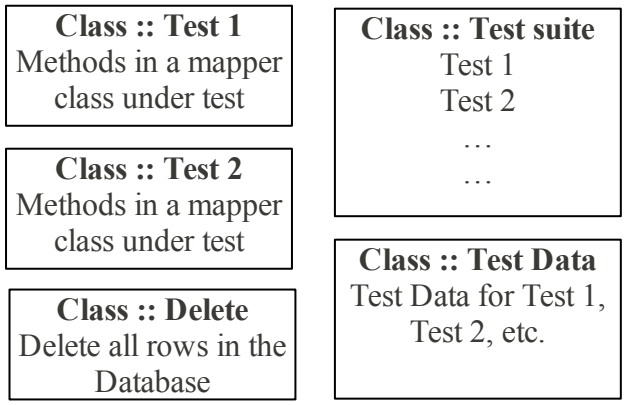
5.2 Unit testing in action

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:



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:

```java
// 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:

```java
@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

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

6.2 Normalization of the Database

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

7.0 Conclusion

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

8.0 Errors and shortcomings

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

3 The Process

1.0 Scrum

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

2.0 Git

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

3.0 Conclusion

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

```
@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()
```