

**Umeå University**  
Department of Computing Science

Application Development in Java 7.5 p  
5DV135

**Unit Test Framework**

Submitted 2016-11-17  
Author: Lorenz Gerber (dv15lgr@cs.umu.se lozger03@student.umu.se)  
Instructor: Johan Eliasson / Jan Erik Moström / Alexander Sutherland / Filip Allberg / Adam Dahlgren Lindström

## Contents

1	Introduction	1
2	Usage	1
2.1	Compiling from Command Line	1
2.2	Program Usage	1
3	System Description	1
3.1	Basic Design	1
4	Testing	3
4.1	Automated Testing	3
4.2	Manual Testing	3
	References	3

## 1 Introduction

The aim of this assignment was to learn building a simple GUI application, using the Java Reflection class and formatting the source code according to a given code convention. The implemented application is a so called Unit Tester. It runs unit tests on a given class. The unit tests themselves are also defined in a specific class. A screenshot of the running application can be seen in *figure 1*.

## 2 Usage

### 2.1 Compiling from Command Line

The source code for the Java application 'UnitTest' was divided into the packages 'model', 'view' and 'controller'. The class containing the 'main' method, the classes to be tested as well as the test classes were contained in the base directory (or default package). The following steps were conducted to build a jar application file:

```
mkdir build
javac -d ./build *.java controller/UnitTestController.java ./
    model/*.java view/*.java
cd build
jar cvfe UnitTest.jar UnitTestMain *.class controller/ view/ model/
```

Note that the 'jar' file is named 'UnitTest' while the 'main' containing class is named 'UnitTestMain'. The source code is provided in a separate tar.gz file.

### 2.2 Program Usage

The 'UnitTest' application is invoked from the command line by `java -jar UnitTest`. Then the application GUI should open with a default test class 'Test1' already chosen. To run the unit tests, the user presses the button 'Run Tests' upon the output shows up in the main text area. An additional button under the main text area 'clear' can be used to empty the main text output area. The application shall be terminated by closing the main application window using the system's window close button.

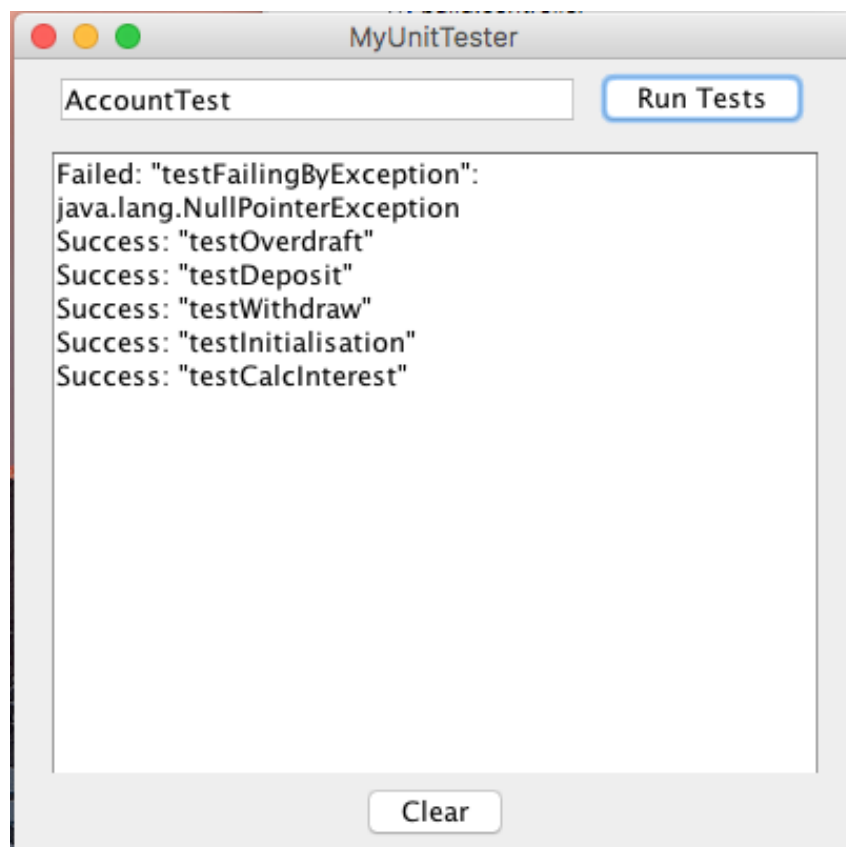
## 3 System Description

### 3.1 Basic Design

The application was designed according to the Model-View-Controller principle. The following classes represent the respective functionality:

- Model: UnitTest
- View: UnitTestGui
- Controller: UnitTestController

The following textual description is also represented in an UML model shown in *figure 2*. The model is represented by the UnitTest class. Plugged into a command line only main program, this class provides all functionality needed to do unit testing according to the



**Figure 1:** This figure shows a typical screendump of the UnitTest applicaiton running on OSX

specifications in the assignment description. Output in this case is obtained through an `ArrayList<String>` that contains all test messages. The 'view' implementation is the 'Swing' based graphical user interface in the class `UnitTestGui`. Here, it was chosen to keep all interfacing parts to the model such as action listeners outside. Instead, methods that export the interactive elements of the GUI were implemented. The final part is the `UnitTestController` class which interconnects model and view. It takes a `gui` object as argument, which provides the needed access to the GUI elements. The controller class contains all the action listeners and controls the general flow of the program such as verification of the test class and running the actual tests. This is achieved by instantiation of the model from within the controller class.

## **4 Testing**

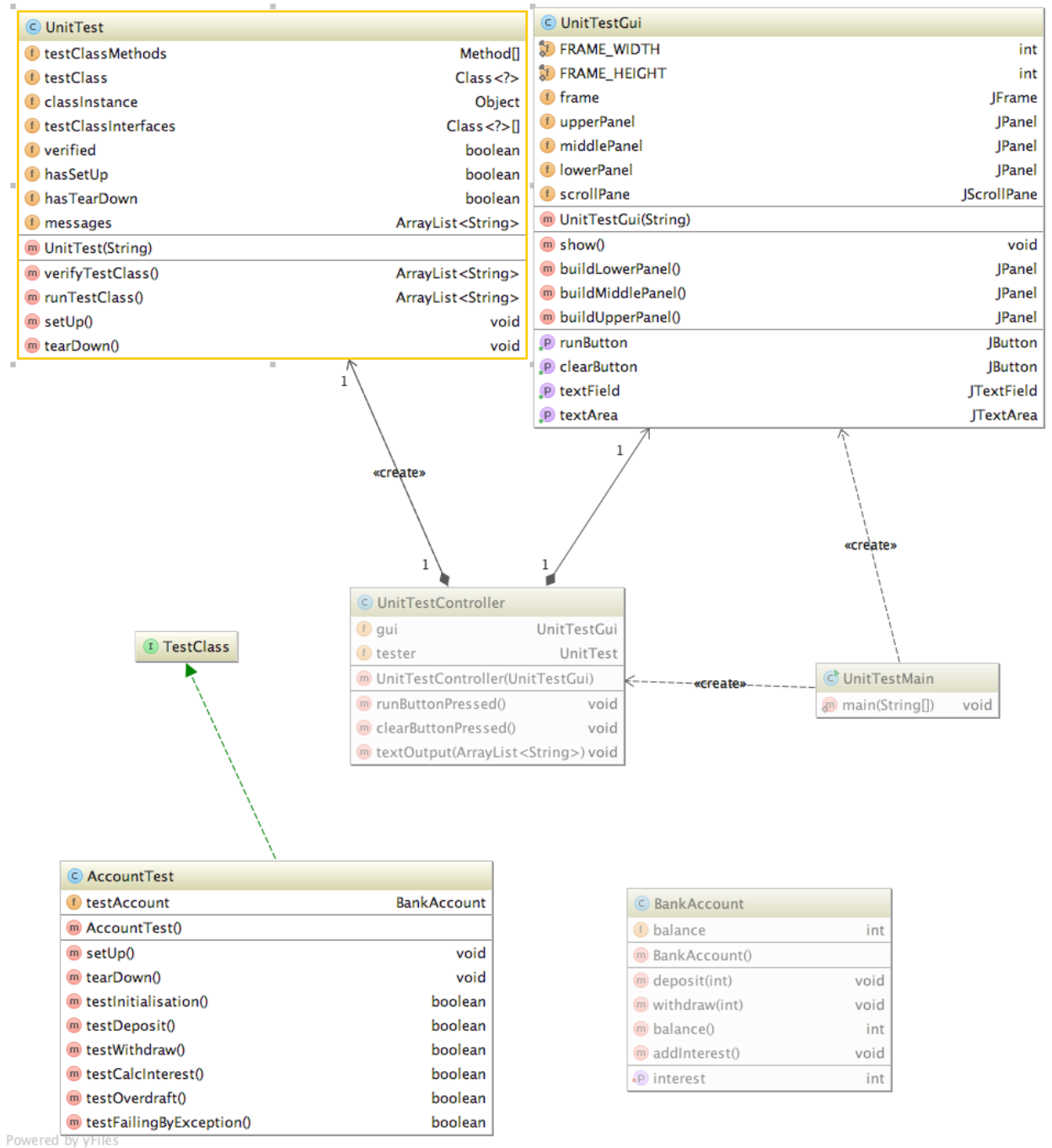
### **4.1 Automated Testing**

A set of JUnit tests for the model class 'UnitTest' was implemented. Due to a misunderstanding, I was not aware that JUnit tests should be included, the tests were built after coding. While building the tests, it was realized that the chosen design for message output was probably not the most convenient as it makes it difficult to test.

The JUnit tests check mostly the `TestClass` verification functionality. They assure that wrong `TestClasses` are detected and reported to the user. A number of test classes (`Test1`, `Test2`, `Test3`) were used for this purpose. It is not clear to me whether this is how it should be done, or whether one should build such objects dynamically within the JUnit code.

### **4.2 Manual Testing**

In manual testing a problem was found related to different operating systems. Running the application on OSX resulted in uncaught exceptions when a correct class name with wrong upper/lowercase letters was entered. Several attempts to fix this issue were unsuccessful. Despite setting the statement in question within a `try/catch` block did not mend the problem. Testing on Linux revealed that this error did not occur there.



**Figure 2:** This figure shows the UML diagram of the UnitTest application.