A Web-based Testing Tool

Yaozhong Li, Bohong Zhou, BianWu,
Mao Zheng and Tom Gendreau
Department of Computer Science
University of Wisconsin - La Crosse
La Crosse, WI 54601
zheng.mao@uwlax.edu

Abstract

Design by contract is a very important concept and method in software development to ensure the quality of the software. This paper describes a web-based testing tool in a teaching environment for instructors and students. The basic concept of design by contract is practiced through specifying pre- and post conditions in each test case.

This paper presents a web-based testing tool used for testing intro-level Java programs. The instructor can create test cases/test suites that will be accessible to all the students. The instructor can also execute the test cases/test suites from programs submitted by the students. Test cases/suites and execution results are saved in a database. Students can use the test cases/suites given by the instructor to test their own programs. Students can also save the execution results and choose to submit one of the results. Any individual student can create his/her own test cases/suites that are only accessible by him/herself.

The test case is created by specifying pre- and post conditions for a method through the graphical user interface. The testing tool will translate the information into a JUnit test class and invoke JUnit to execute.

Currently the testing tool supports unit testing for the following: void and non-void methods with/without primitive and reference type parameters, inheritance, polymorphism and file IO. For the primitive type parameters, boundary value analysis can be used to generate test cases based on a given range. Java inner classes and subfolders are not supported in the current prototype.

1 Introduction

This paper presents a web-based testing tool used by instructors and students in the CSI and/or CSII courses. The tool aims to be used in a teaching and learning environment to test intro-level Java programs.

The instructor can create test cases/test suites that will be accessible to all the students. The instructor can also execute the test cases/test suites from programs submitted by the students. Test cases/suites and execution results are saved in a database. Students can use the test cases/suites given by the instructor to test their own programs. Students can also save the execution results and choose to submit one of the results. Any individual student can create his/her own test cases/suites that are only accessible by himself/herself.

Each test case is created by specifying pre- and post conditions for a method. This also helps students to practice the concept of "Design by Contract"(DBC) [1].

Design by contract is a very important concept and method in software development to ensure the quality of the software. The principal idea behind DBC is that a class and its clients have a "contract" with each other. The client must guarantee certain conditions before calling a method defined by the class and in return the class guarantees certain properties that will hold after the call. In this testing tool, the contracts are defined by using pre- and postconditions in the test cases. When executing the test cases, any violation of the contract that occurs while the program is running can be detected immediately.

2 The Design of the Web-based Testing Tool

2.1 High level Architecture Design

Model-View-Controller ("MVC") is the BluePrints recommended architectural design pattern for interactive applications. MVC, organizes an interactive application into three separate modules: one for the application model with its data representation and business logic, the second for views that provide data presentation and user input, and the third for a controller to dispatch requests and control flow. The testing tool presented in this paper used MVC design pattern. The instructor/student, from a client machine, will use his/her web browser to interact with the system. The web browser will send HTTP requests to the web server. This in turn requests a service and passes parameters to the application model. The application model handles the request. The result will then be passed back to the client browser.

At the highest level, there are four basic types of actions as shown in Figure 1: interpret client requests, dispatch those requests to business logic, select the next view for display, and generate and deliver the next view.

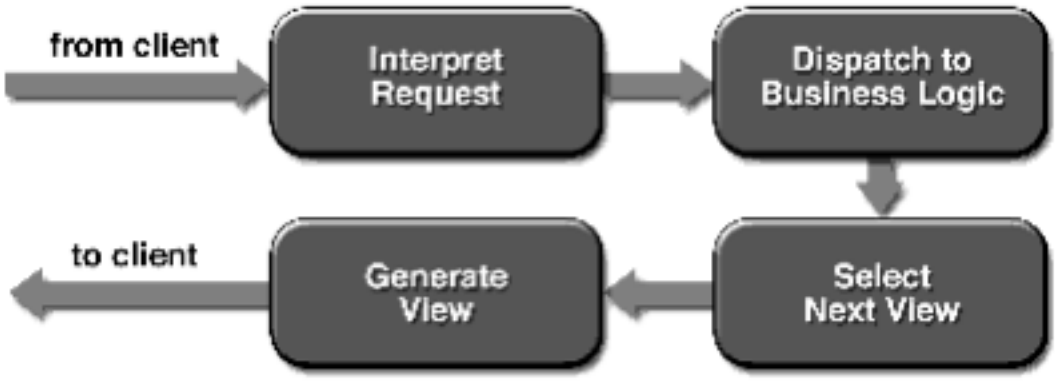

Figure 1 : Service Cycle

Here Figure 2 below describes the high level architecture design of the testing tool.

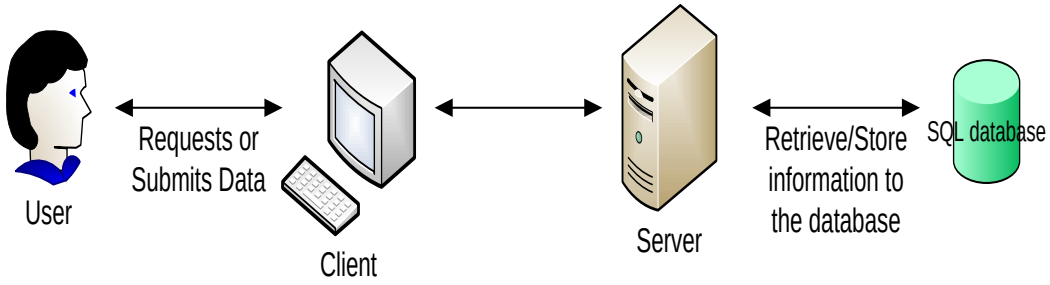

Figure 2 : High Level Architecture Design

Currently the testing tool is designed as a two-tired application. Both the web server and database are on the server machine.

2.2 Web Tier Design

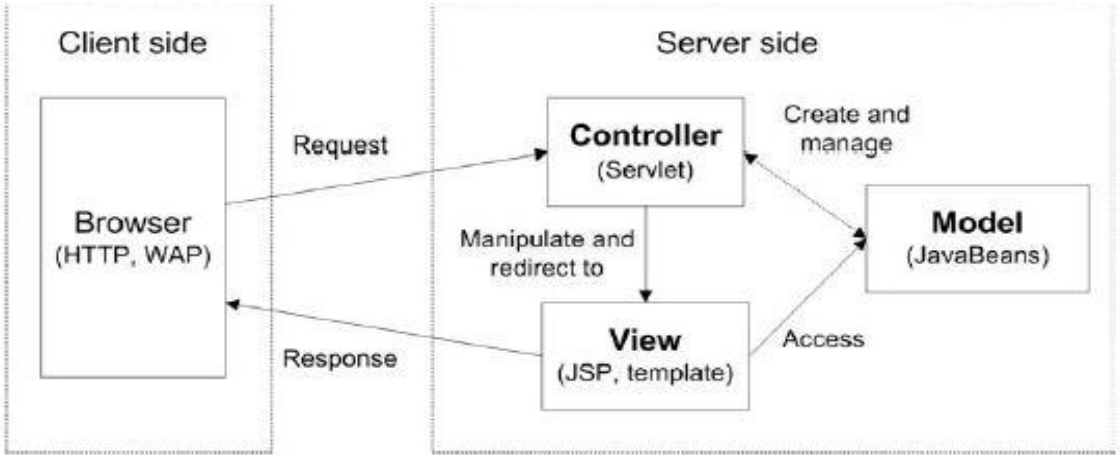

Figure 3 : JSF Web Design [3]

The testing tool uses JavaServer Faces(JSF) [3] to develop web interface as shown in Figure 3. A Java Community Process effort (JSR-12) is currently defining a standardized Web application framework called JavaServer Faces. Current standard Web-tier technologies offer only the means for creating general content for consumption by the client. There is currently no standard server-side GUI component or dispatching model. JavaServer Faces will be an architecture and a set of APIs for dispatching requests to Web-tier model JavaBeans; for maintaining stateful, server-side representations of reusable HTML GUI components; and for supporting internationalization, validation, multiple client types, and accessibility. Standardization of the architecture and API will allow tool interoperation and the development of portable, reusable Web-tier GUI component libraries.

2.3 Business Logic

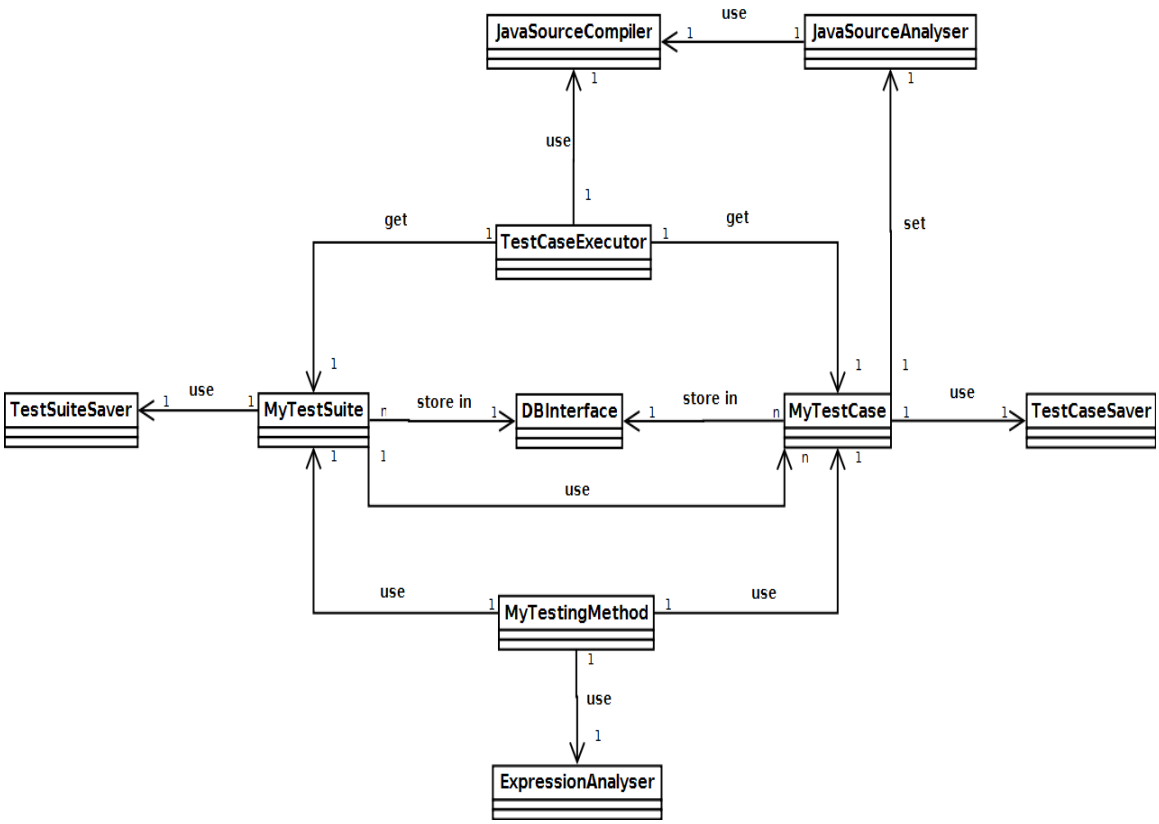

Figure 4 : Class Diagram

Figure 4 shows the business logic class diagram. A user will specify the test case through a graphical user interface. Classes JavaSourceAnalyser, JavaSourceCompiler and MyTestCase will process the information to generate a test case: rewrite/translate into JUnit test class. The class TestExecutor will invoke JUnit to execute test cases.

Java Reflection class is used by JavaSourceAnalyser to get the basic information from a Java source class: the class name, attribute names and types, and the method signatures.

The testing tool uses JUnit 4.8.2 [2] as the backbone: class MyTestCase translate a user's test case into JUnit test class and TestCaseExecutor will invoke the JUnit to execute the test classes.

2.4 Database Design

The user, test case and test result's information are stored in the database as shown in Figure 5.

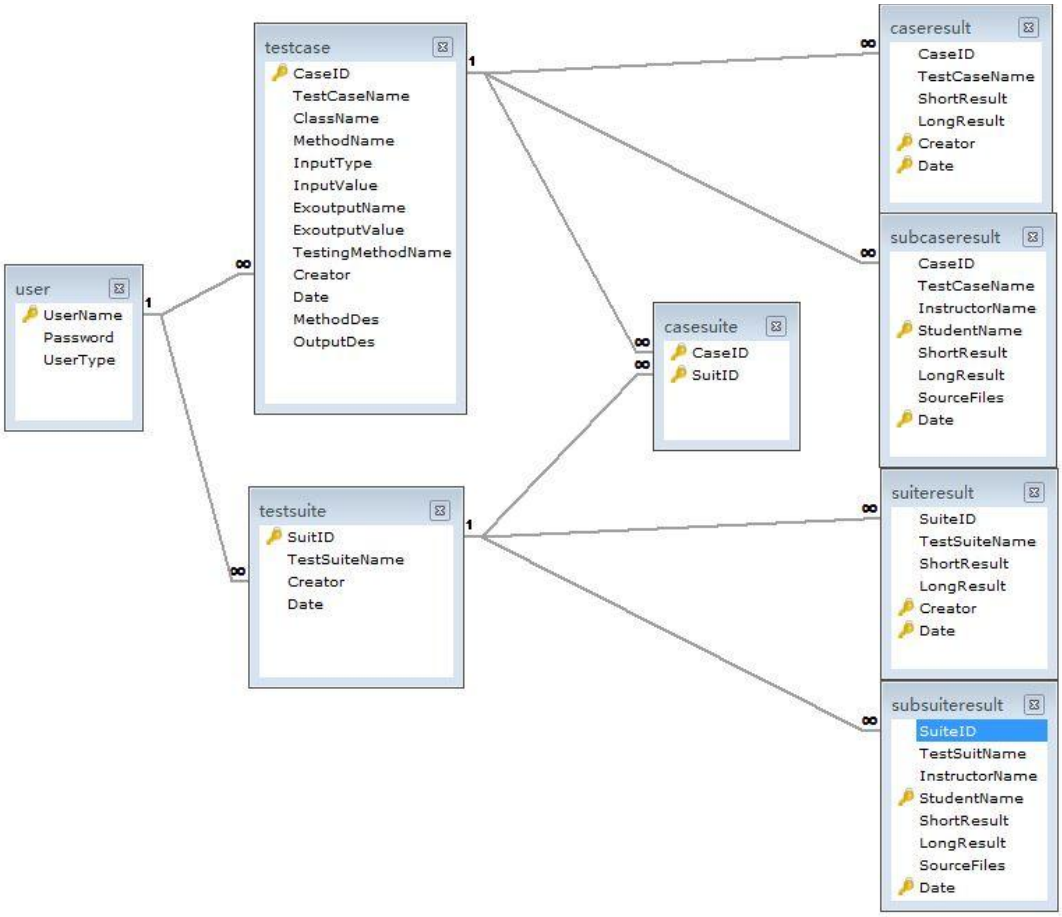

Figure 5 : ER Diagram

3 Implementation Examples

3.1 Test Inheritance

Here is an example of how to use the testing tool to test inheritance. Figure 6 shows the GUI screen used to create a test case.

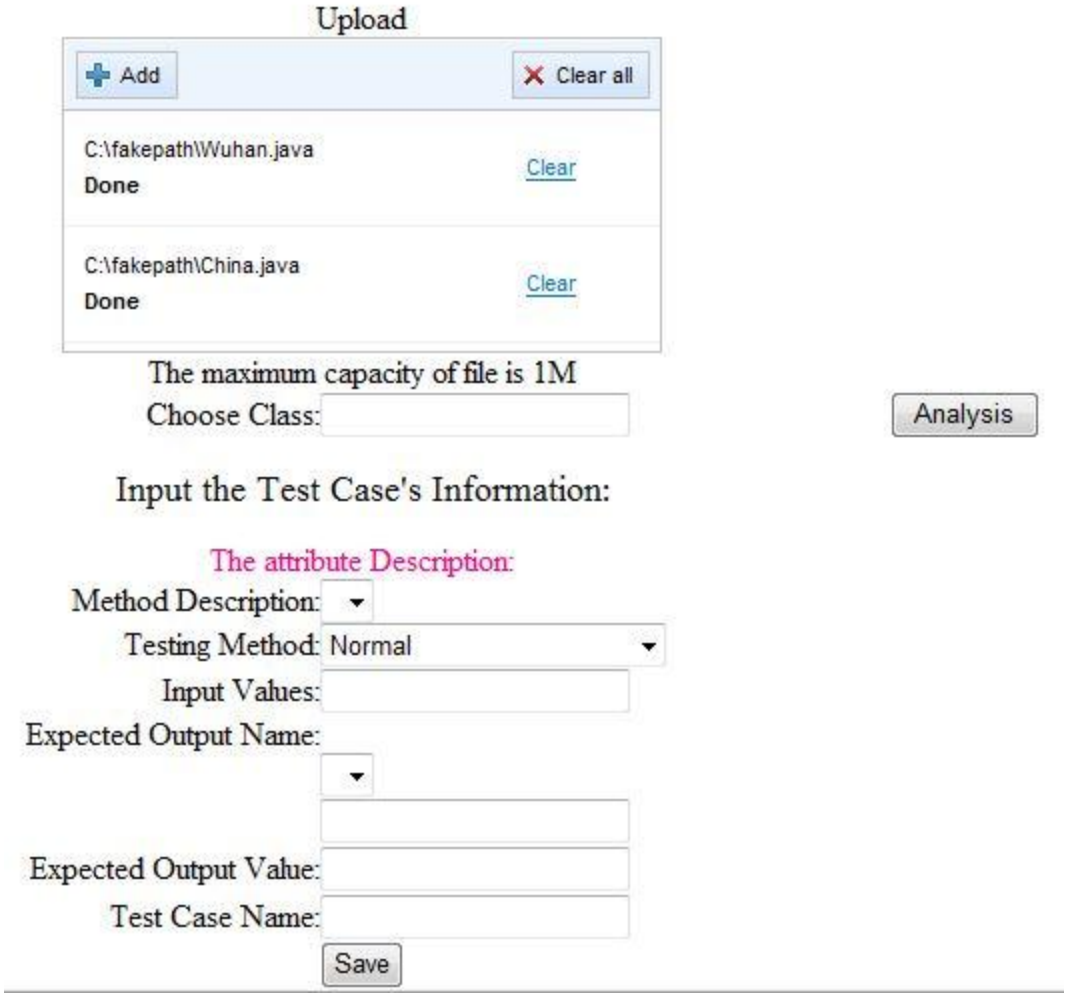

Figure 6 : Create Test Case

The user can upload all the files, then choose the Java class that he/she intends to test. The "Analysis" button will compile the chosen Java class and obtain all the information from the class such as attribute names and types and method signatures. This way, the user can decide which method to test and specify the test case by indicating "Input Values" and "Expect Output Value".

If a method is a void method, or the test case intends to test other properties of the class instead of the method's return value, the user can also specify an attribute's name in the "Expect Output Name".

Figure 7 is the test case created.

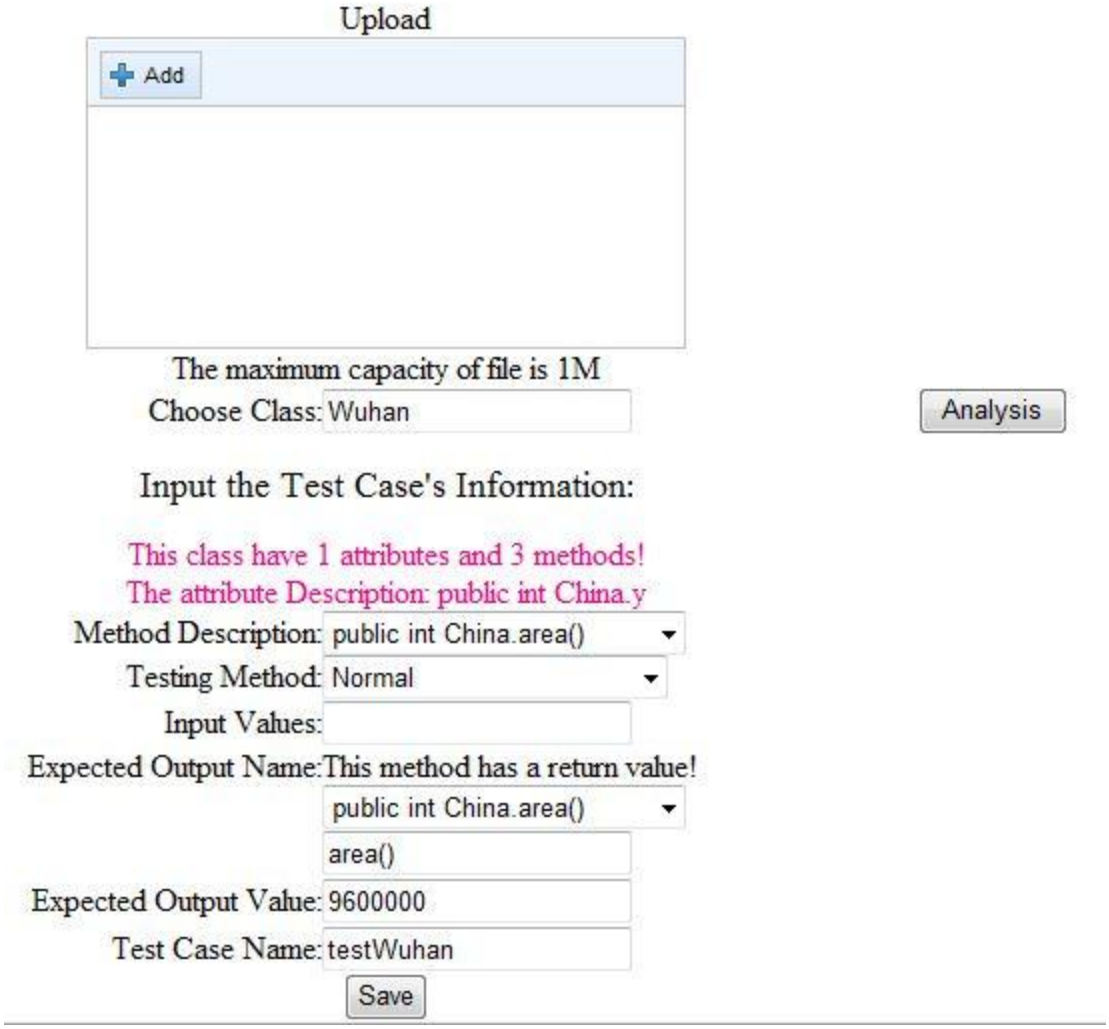

Figure 7 : Inheritance Test Case

In this test case, the purpose is to test the method area() in the class "Wuhan". This class is inherited from class China.

The testing tool translates the test case into a JUnit test class. Below is the source code of the JUnit test class.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class testWuhan{
    private static Wuhan tester = new Wuhan();

    @Test
    public void testsay() {
    tester.area();
    assertEquals(9600000,tester.area(),0);
```

```
    }
}
```

Figure 8 shows the test execution result. This test case is passed (OK).

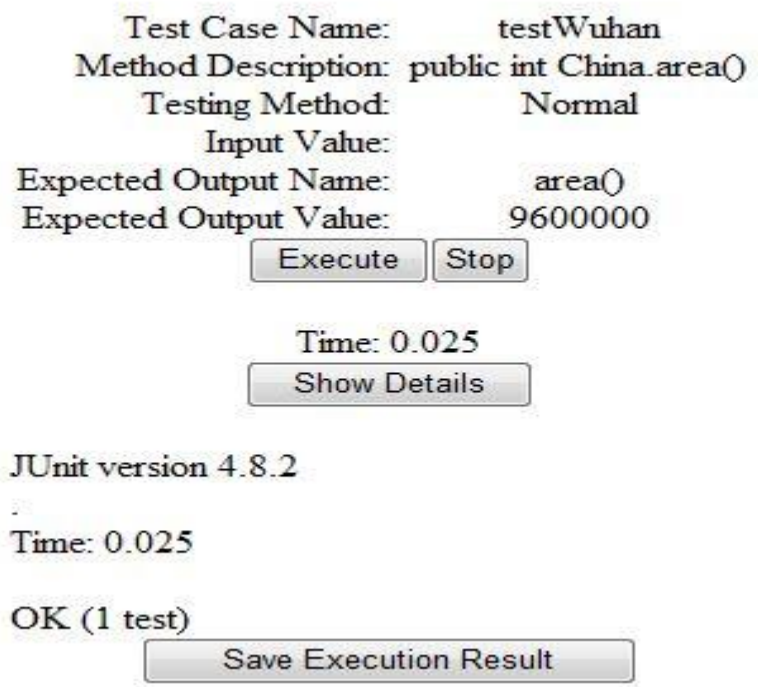

Figure 8 : Test Execution Result

3.2 Boundary Value Analysis

The current testing tool also implements boundary value test case generation. Figure 9 shows the test case created by specifying the input ranges and the boundary value analysis method is chosen as the testing method. The testing tool will generate JUnit test classes based on the chosen test method and finally a JUnit test suite. For example: based on single-fault assumption from reliability theory, from two parameters, 13 test cases are generated. The JUnit test suite source code is shown below.

```
import junit.framework.JUnit4TestAdapter;
import junit.framework.Test;
import junit.framework.TestSuite;

public class TestBoundary{

        public static Test suite() {
                TestSuite suite = new TestSuite(TestBoundary.class.getName());
                //$JUnit-BEGIN$
                suite.addTest(new JUnit4TestAdapter(TestBoundary_1.class));
```

```
        suite.addTest(new JUnit4TestAdapter(TestBoundary_2.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_3.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_4.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_5.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_6.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_7.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_8.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_9.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_10.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_11.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_12.class));
        suite.addTest(new JUnit4TestAdapter(TestBoundary_13.class));
        //$JUnit-END$
        return suite;
    }
}
```

Figure 9 : Boundary Value Analysis

This example illustrates that the testing tool can incorporate multiple test case generation methods in the future.

Figure 10 shows the test execution result. There are four cases out of the range that were specified in the test cases as Figure 9, therefore 4 failures demonstrated in the execution.

Figure 10 : Test Execution Result

4 Summary

Currently the testing tool supports unit testing for the following: void and non-void methods with/without primitive and reference type parameters, inheritance, polymorphism and file IO. For the primitive type parameters, boundary value analysis can be used to generate test cases based on a given range. Java inner class and subfolders are not supported in the current prototype.

In the design and deployment of the testing tool, the assumption is that it is used in a teaching and learning environment. The user is "trusted": they can upload the source file and execute the test case against the Java source files in the server. Future work of this testing tool will include different levels of user access and server security checks.

References

[1] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40-51, October 1992.
[2] **http://www.junit.org/**.
[3] Kito D.Mann, Java Server in Action, Manning Publications Co., 2005, ISBN: 1-932394-11-7