A Testing Framework to Improve Automatic Grading of Student Programming Assignments

Submitted by

Madhuri Gurumurthy

Under the guidance of

Prof. Daniel Bergeron

Table of Contents

Abstract

Grading programming assignments is tedious and time consuming. Extensive testing and evaluation is needed, especially to award partial credits for partially correct results. This is particularly a problem at UNH because of rapidly increasing enrollments and the complexity of the programs we assign. Creating rubrics to award partial credit for complex assignments poses another challenge since it is difficult to predict everything students can do wrong.

We currently have a semi-automated testing tool to evaluate the assignments by comparing their output to the output expected by tests specified by the instructor [Rossi,2016]. Non-trivial work is needed for the instructor to convert existing assignments to utilize the tool effectively.

This research explores how we can use test-driven development methodology to improve assignment design, simplify the process of refactoring existing assignments, and streamline the effectiveness of the grading process. A major goal of the project is to provide a framework and tools to facilitate the conversion and improve support for testing more complex programs, especially those that build interactive GUIs.

We have extended the testing tool to support JUnit suites and parameterized tests and have introduced an entirely new functionality based on the Mockito testing tool that supports the creation of "mock" test objects suitable for both interactive and non-interactive programs. The framework uses advanced features of JUnit to create/support a test suite to perform bulk testing of similar tests on similar methods/classes. Threads and Reflection API are the other features used to develop a simple automated tool, Synchronized Checkpoint Execution. This tool executes the student program and a solution program until a certain checkpoint. The outputs from both the programs are compared at these checkpoints to facilitate synchronous evaluation of the student's code against the solution code.

1. Introduction

Grading assignments of programming intensive courses in Computer Science is time consuming. Extensive testing and evaluation needs to be performed to award partial credits for partially correct results. This is a problem especially for UNH, with the increasing enrollment of students. The introductory CS courses require students to develop graphical and interactive programs. Currently, graders and TAs are relied upon to evaluate these assignments based on functionality, quality, performance, and reliability of the code. Creating rubrics to award partial credit for complex assignments poses another challenge since it is difficult to predict all the things students can do wrong. Also, matching partially correct answers to rubrics is time consuming and is hard to do consistently for all the students.

To facilitate a better grading process, a semi-automated testing tool was developed [Rossi, 2016]. This tool evaluates the assignments by comparing their output to the output expected by tests specified by the instructor. Although effective for some of assignments and labs, this tool has some important limitations:

(a) it has only minimal support for testing interactive GUI programs; and
(b) many non-interactive programs need extensive re-structuring in order to effectively test advanced features such as graph and tree data structures.

2. Unit Testing

2.1 Introduction

Unit is the smallest testable part of an application.

Unit Testing is "Testing of individual units of source code."

It is a white box testing, tests internals of code, i.e. Business Logic in individual method in code is tested.

2.2 Why Unit Testing?

- Detect bad design decisions as early as possible
- Detect implementation errors as early as possible
- Proves that code works
    - Enforces better design
    - Faster way to test
- Regression assurance
    - Changes do not break existing code
    - Changes do not change behavior of existing code

2.3 What Unit Test should do?

- Test boundary conditions, Null Values, Branches (if-else etc.)
- Unexpected errors (Do not put try-catch in unit test)

2.4 Characteristics of a good Unit Test:

- Test one aspect/behavior, avoid verifying too many functionalities in one test.
- Document the expected behavior.
- Good test code is well defined code.

2.5 Different ways of Unit testing

- Manual writing of tests
- Using automation like Selenium, etc.
- Using unit testing framework such as Mockito etc.

2.6 Some Things that can be done while unit testing:

- Dummy objects are created and passed around. But actual objects are not used.
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

2.7 Design factors that can complicate unit testing

- Code in constructor
- Global State
- Singleton classes
- Static methods
- Deep inheritance
- Too many conditionals
- Method providing more than one functionality

2.8 JUnit

JUnit is a simple framework to write repeatable tests [1]. It is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development.

2.8.1 A simple example

Consider the simple "PowerSum" program shown in Code Snippet 1a. It has two methods that need to be tested, sum and powSum. The method powSum (Code Snippet 1c) calls the method (Code Snippet 1b) which sums the values mentioned in the array recursively, and returns the total value to the pow method. The pow method then uses the sum value returned as an exponent and finds the power of 2 (Code Snippet 1c).

```
1 public class PowerSum
2 {
3     public int sum(int[] vals, int n)
4     {
5           ...
6     }
7
8     public int powSum()
9     {
10          ...
11    }
12 }
```

Code Snippet 1a: Program to find the sum and power of the given numbers.

```
1 public int sum(int[] vals, int n)
2 {
3     if (n == 0)
4           return 0;
5     else
6           return vals[n - 1] + sum(vals, n - 1);
7 }
```

Code Snippet 1b: The sum method of PowerSum class.

```
1 public int powSum(int[] vals)
2 {
3     int n = vals.length;
4     int exp = sum(vals, n);
5     if (exp == 0)
6           return 1;
7     else
8           return (int) Math.pow(2, exp);
9 }
```

Code Snippet 1c: The pow method of PowerSum class.

The unit test framework for the class that tests PowerSum class has two methods (Code Snippet 1d), one testing sum method and the other testing powSum method.

```
1 public class PowerSumTest
2 {
3     private PowerSum ps;
4
5     @Before
6     public void setUp() throws Exception
7     {
8           ps = Mockito.mock(PowerSum.class);
9     }
10
11    @Test
12    public void testPowSum()
13    {
14          ...
15    }
16
17    @Test
18    public void testSum()
19    {
20          ...
21    }
22 }
```

Code Snippet 1d: The skeleton of the unit test class of PowerSum class.

```
1 @Test
2 public void testSum()
3 {
4     int[] vals = { 1, 2, 3, 5, 5 };
5     int n = 5;
6     int result = ps.sum(vals, n);
7     assertEquals(16, result);
8 }
```

Code Snippet 1e: The unit test of the sum method of PowerSum class.

```
1 @Test
2 public void testPowSum()
3 {
4     int[] vals = { 1, 2, 3, 4, 5 };
5     int n = 5;
6     Mockito.when(ps.sum(vals, n)).thenReturn(15);
7     int result = ps.powSum(vals);
8     assertEquals(32768, result);
9 }
```

Code Snippet 1f: The unit test of the pow method of PowerSum class.

@Test- The `Test` annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded. [2] (Code Snippet 1e, line 1), (Code Snippet 1f, line 1).

The Test annotation supports two optional parameters. The first, expected, declares that a test method should throw an exception. If it doesn't throw an exception or if it throws a different exception than the one declared, the test fails. (Code Snippet 2b, line 1).

Example of the test method that expects an exception is mentioned in the Code Snippet 2b. The main program is mentioned in the Code Snippet 2a.

```
1 class Summation
2 {
3     public int sum (int[] vals, int n) throws SumException
4     {
5           if (n == 0)
6                 throw new SumException("Invalid Number");
7           else{
8                       return vals[n-1]+sum(vals,n-1);
9                 }
10    }
11}
```

Code snippet 2a: Program to find the sum of the given numbers.

```
1 @Test(expected=SumException.class)
2 public void testSum()
3 {
4     int[] vals = { 1, 2, 3, 5, 5 };
5     int n = 5;
6     s.sum(vals, n);
7 }
```

Code snippet 2b: Unit test program to find the sum of the given numbers.

The second optional parameter, timeout (Code Snippet 3, line 1), causes a test to fail if it takes longer than a specified amount of clock time (measured in milliseconds).

```
1 @Test(timeout = 500)
2 public void timeTest() throws InterruptedException
3 {
4    Thread.sleep(1000);
5 }
```

Code snippet 3: Unit test program to demonstrate timeout() functionality. The method fails as it did not complete the execution within the timeout period specified.[3]

2.8.2 Assertions

JUnit provides overloaded assertion methods for all primitive types and Objects and arrays (of primitives or Objects). The parameter order is expected value followed by actual value [Code Snippet 1e, line 7], [Code Snippet 1f, line 8]. Optionally the first parameter can be a String message that is output on failure. There are various assert tests like assertEquals, assertNotEquals, assertFalse, assertNull, etc[4].

Some assert statements are as follows:

Assert statements	Syntax

assertEquals	assertEquals(long expected, long actual)
assertNotNull	assertNotNull(java.lang.String message, java.lang.Object object)
assertNull	assertNull(java.lang.String message, java.lang.Object object)
assertThat	assertThat(java.lang.String reason, T actual, org.hamcrest.Matcher<T> matcher)
assertTrue	assertTrue(java.lang.String message, boolean condition)
assertArrayEquals	assertArrayEquals(long[] expecteds, long[] actuals)
assertFalse	assertFalse(java.lang.String message, boolean condition)

2.8.2 Junit Test Suite

A TestSuite is a Composite of Tests. It runs a collection of test cases.

Code Snippet 4a and 4c demonstrate a program to find the sum of the given numbers and exponential of the given numbers, respectively. The unit test class to test these class programs are mentioned in the Code Snippet 4b and 4d, respectively.

```
1 class Summation
2 {
3     int sum (int[] values)
4     {
5           int sum=0;
6           for(int i= 0; i< vals.length; i++)
7                 sum=sum+val[i];
8           return sum;
9     }
10 }
```

Code snippet 4a: Program to find the sum of the given numbers.

```
1 public class JunitSummationTest
2 {
3     @Test
4     public void sumTest()
5     {
6           Summation s= new Summation ();
7           int[] vals={1,2,3,4,5};
8           assertEquals(15, s.sum(vals));
9     }
10 }
```

Code snippet 4b: Unit Test class to find the sum of the given numbers

```
1 class ExponentialPower
2 {
3     int power(int exp, int pow)
4     {
5           int result = 0;
6           for (int i = 0; i < pow; i++)
7                 result = result * exp;
8           return result;
9     }
10 }
```

Code snippet 4c: Program to find the exponential power of the given number.

```
1 public class JunitExponentialPowerTest
2 {
3     @Test
4     public void sumTest()
5     {
6           ExponentialPower exp = new ExponentialPower();
7           assertEquals(32, exp.power(5, 2));
8     }
9 }
```

Code snippet 4d: Unit Test program to find the exponential power of the given number.

The test classes in Code Snippet 4b and 4d can be executed simultaneously in a test suite as mentioned in the Code Snippet 4e.

```
1 import org.junit.runner.RunWith;
2 import org.junit.runners.Suite;
3
4 @RunWith(Suite.class)
5 @Suite.SuiteClasses({ JunitSummationTest.class,
6                 JunitExponentialPowerTest.class })
7 public class JunitRecursionTestRunner{}
```

Code snippet 4e: Junit Test Suite to run multiple tests.

An alternative approach to test suite is executing the classes by mentioning as in Code Snippet 4f. In this method, the output can be printed in a user friendly way. Also further code, if any, can be written as part of the program.

```
1 public class JunitTestRunner
2 {
3     public static void main(String[] args)
4     {
5           Result result = JUnitCore.runClasses(JunitSummationTest.class,
6                 JunitExponentialPowerTest.class);
7           for (Failure fail : result.getFailures())
8                 System.out.println(fail.toString());
9           if (result.wasSuccessful())
10                System.out.println("All tests finished successfully.");
11    }
12 }
```

Code snippet 4f: Junit Test Suite to run multiple tests, an alternative approach.

3. Family of Test Tools

3.1 Object mocking

There are many different mocking frameworks in the Java space, however there are essentially two main types of mock object frameworks, ones that are implemented via proxy and ones that are implemented via class remapping. A proxy object is an object that is used to take the place of a real object. In the case of mock objects, a proxy object is used to imitate the real object your code is dependent on. The second form of mocking is to remap the class file in the class loader. In this method you tell the class loader to remap the reference to the class file it will load [5].

Why Mocking?

- Class has dependencies. Those dependencies have to be somehow broken, so we can test one and only one class at a time.
- **Mock Object**: It is dummy object, which can behave as actual in a controlled way.
- By using mock object, the dependency of objects can be cut. You can inject (set) these mock object, and configure the behavior of it.
- Through mocking we can minimize the dependencies of a class, so it can be tested in isolation, without testing the functionality of the dependent classes.

Some Things that are done in unit testing through Mockito:

- **Mock** objects pre-programmed with expectations which form a specification of the calls they are expected to receive.
- Testing involves creating multiple mock objects; in principle covering range of expected values in final system.

3.2 Mockito

Mockito is an open source testing framework for Java, released under the MIT License, it is a "mocking framework", that lets you write unit tests for the purpose of Test-driven Development (TDD) or Behavior Driven Development (BDD). The mock object expects a certain method to be called and it will return an expected result.[6]

3.3 EasyMock

EasyMock is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications. EasyMock is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing [7].

3.4 PowerMock

PowerMock is an "add-on", of sorts, for EasyMock. It adds the ability to mock things that would not normally be considered possible, such as private methods, static methods. It is a separate package from EasyMock, but it "plays well" with EasyMock and other mocking frameworks (e.g. Mockito). [8]

3.5 Annotations and Methods used in Mockito [9]

- when(..). thenReturn(..): This method is used for stubbing (Code Snippet 1f, line 6) (Code Snippet 5b, line 9 and 12).
  An alternative stubbing method is doReturn(..).when(..). It is roughly equivalent to when(..).thenReturn(..). But there are cases when you cannot use when(..).thenReturn(..).
  For example stubbing void method, stubbing same method more than once, etc.
  One thing that when(..)thenReturn(..) gives you, type-checking of the value that you're returning, at compile time [10].

Example: Consider the program to find count of a specified letter in a string (Code Snippet 5a). In this program the source string and the character to be searched in the string are provided. The program returns the count of the character in the string.
There are two conditions that are to be tested. Case 1: When The length of string is 0, the program returns 0. (Code Snippet 5b, line 9) Case 2: When the length of string is greater than 0, the program returns the count of the character in the string (Code Snippet 5b, line 12).

```
1  class LetterCounting
2  {
3      public static int letterCount(String line, char letter)
4      {
5            if (line.length() == 0)
6                  return 0;
7            if (line.charAt(0) == letter)
8                  return 1 + letterCount(line.substring(1), letter);
9            else
10                 return letterCount(line.substring(1), letter);
11     }
12 }
```

Code Snippet 5a: Program to find count of a specified letter in a string.

```
1  String line = Mockito.mock(String.class);
2  LetterCounting lc=Mockito.mock(LetterCounting.class);
3  char letter = 'c';
4  int count = lc.letterCount(line, char);
5
6  //Mock the method values to test the behavior
7
8  //Returns 0
9  case 1: Mockito.when(line.length()).thenReturn(0);
10
11 //Let the length be 5.
12 case 2: Mockito.when(line.length()).thenReturn(5);
13
14 //Since string length is 5, so 5 times charAt(0) is called. So
15 //return method returns 5 characters, once for each call.
16 Mockito.when(line.charAt(0).thenReturn(letter,'a','b',letter,'d');
```

Code Snippet 5b: Unit test program to count the numbers of a specified letters in a string.

- @RunWith - To enable annotations for Mockito tests. When a class is annotated with @RunWIth, JUnit will invoke the class in which is annotated so as to run the tests, instead of using the runner built into JUnit. Also, validates framework usage after each test method. [11] (Code Snippet 6)

```
1  //Initializes mocks annotated with Mock
2  @RunWith(MockitoJunitRunner.class)
3  public class PowerSum{
4  …
5  }
```

Code snippet 6: To enable annotations for mock test.

Alternatively, we can **enable these annotations programmatically** as well, by invoking *MockitoAnnotations.initMocks()* (Code Snippet 7).

```
1 @Before
2 public void init(){
3     MockitoAnnotations.initMock(this);
4 }
```

Code snippet 7: Alternative approach to enable annotations for Mockito.

3.6 PowerMockito Static example

Static methods are often seen in the programs. These methods pose a challenge when un it testing. Following methods and annoatations gives an outline of unit testing such static methods.

- PowerMockito.mockStatic(..)

  PowerMockito.mockStatic(..) is used to enable static mocking for all static methods of a class. [12]

  Following are the steps you need to follow to mock/stub static methods.
  1. Add @PrepareForTest at class level [Code Snippet 5, line2].
  2. Call PowerMockito.mockStatic() to mock a static class [Code Snippet 5, line 9].
     (use PowerMockito.spy(MyStaticClass) to mock a specific method).
  3. Just use Mockito.when() to setup your expectation [Code Snippet 5, line 15].

- @PrepareForTest
  This annotation tells PowerMock to prepare certain classes for testing. This annotation is needed for classes whose byte-code needs to be manipulated. This includes final classes, classes with final, private, static or native methods that should be mocked and also classes that should be return a mock object upon instantiation. [13]

  This annotation can be placed at both test classes and individual test methods. If placed on a class all test methods in this test class are handled by PowerMock (to allow for testability). To override this behavior for a single method just place a @PrepareForTest annotation on the specific test method. [13]

  Consider an example where we test an anagram generating program. An anagram is a word, phrase, or name formed by rearranging the letters of another, such as *cinema*, formed from *iceman*.In this program, the user is prompted to enter the type of dictionary, type of data structure, and the words to be checked. Different user input needs to trigger different tests (Code Snippet 8).

```
1 @RunWith(PowerMockRunner.class)
2 @PrepareForTest({ JOptionPane.class, Scanner.class })
3 public class JunitMockitoAnagramTest extends PowerMockTestCase
4 {
5     @Test
6     public void testAnagram()
7     {
8           AnagramTest test = Mockito.mock(AnagramTest.class);
9           PowerMockito.mockStatic(JOptionPane.class);
10          JOptionPane.showInputDialog(any(), any(String.class));
11          JOptionPane joption = Mockito.mock(JOptionPane.class);
12          File file = new File("opted3.txt");
13          try
14          {
15                Mockito.when(joption.showInputDialog(any(),
16                anyString())).thenReturn("s", "b", "river",river",
17                "elf", "ram", "set","bus", null);
18
19                PowerMockito.whenNew(File.class).
20                withArguments(String.class).thenReturn(file);
21          } catch (Exception e)
22          {
23                e.printStackTrace();
24          }
25          test.findAnagramByOptions();
26    }
27 }
```

Code snippet 8: Example demonstrating the mocking of static methods. Program to test Anagram.

3.7 Mocking private method example

When you've situations where you need to mock private methods, Powermock comes handy and useful. It's better than modifying the code under test. [14]

Example: Consider PowerSum class with sum method being private (Code Snippet 9a). The unit test for the private method is written using the spy method (Code Snippet 9b).

```
1 public class PowerSum
2 {
3     private int sum(int[] vals, int n)
4     {
5           ...
6     }
7
8     public int pow()
9     {
10          ...
11    }
12 }
```

Code snippet 9a: Program to find the sum and power of the given numbers.

```
1 @RunWith(PowerMockRunner.class)
2 @PrepareForTest(PowerSum.class)
3 public class PowerSumTest
4 {
5     PowerSum ps;
6
7     @Before
8     public void setUp() throws Exception
9     {
10          ps = PowerMockito.spy(new PowerSum());
11    }
12
13    @Test
14    public void testPower()
15    {
16          int[] vals = { 1, 2, 3, 4, 5 };
17          int n = 5;
18          PowerMockito.when(ps, "sum", vals, n).thenReturn(15);
19          int result = ps.pow();
20          assertEquals(32768, result);
21    }
22 }
```

Code snippet 9b: Unit test program to find the sum and power of the given numbers when the method is private.

4. SYNCHRONIZED CHECKPOINT EXECUTION

A program can contain many checkpoints, which can be defined as intermediate stages in a program execution. At these checkpoints, certain outputs are formed. The goal of this approach is to use such checkpoints to compare the outputs of the student and solution synchronously. For example, assume that a program *P* has three checkpoints, *a, b,* and *c*, in its execution. This approach executes till the first checkpoint, i.e. a, from student and solution code and gives the control to the comparison program which compares the output. Then the execution in student's and solution code resumes and executes till next checkpoint, and so on. To achieve this goal, we use threads and reflection API.

A thread is an independent path of execution and many threads can run concurrently within a program. Every thread in Java is created and controlled by the java.lang.Thread class. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously. [15]

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the program running in the computer. Each user request for a program or system service (and here a user can also be another program) is kept track of as a thread with a separate identity. As programs work on behalf of the initial request for that thread and are interrupted by other requests, the status of work on behalf of that thread is kept track of until the work is completed. [16]

Multithreading has several advantages over Multiprocessing such as;

- Threads are lightweight compared to processes
- Threads share the same address space and therefore can share both data and code
- Context switching between threads is usually less expensive than between processes
- Cost of thread intercommunication is relatively low that that of process intercommunication
- Threads allow different tasks to be performed concurrently.

The reflection APIs (java.lang.reflect) can be used to dynamically figure out the capabilities of an object without necessarily knowing anything about it in advance [17]. Reflection is commonly used by programs that require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine.

Dynamic Java Class loading is an important feature of the Java Virtual Machine because it provides the Java platform with the ability to install software components at run-time. Dynamic class loading maintains the type safety of the Java Virtual Machine by adding link-time checks, which replace certain run-time checks and are performed only once. Moreover, programmers can define their own class loaders that, for example, specify the remote location from which certain classes are loaded, or assign appropriate security attributes to them [18].

In order to develop a better testing tool, we used the features of both threads and Reflection API.

The implementation of the tool to test the student's assignments is as follows:

- The program takes three parameters as an input- file path where the student's assignment is located, file path where solution and student program is located and the class to be tested.
- Then the current working directory is set to the file path of the solution [Code Snippet 1, Line 1] Using reflection API, the required class (Recursion) is loaded and a new instance of the class is created [Code Snippet 1].

```
1 String solutionPath = System.getProperty("solPath");
2 System.setProperty("user.dir", solutionPath);
3 File solFile = new File(solutionPath);
4 URL solUrl = solFile.toURL();
5 URL[] solUrls = new URL[] { solUrl };
6 ClassLoader solClazz = new URLClassLoader(solUrls);
7 Class solution = solClazz.loadClass("solution.Recursion");
8 Object solutionObj = solution.newInstance();
```

Code Snippet 1: Code to set the current working directory for the solution and also to fetch the required class for the solution.

- Next, the current working directory is set to the file path of the student [Code Snippet 2, line 2]. Using reflection API, the required class (Recursion) is loaded and a new instance of the student's class is created [Code Snippet 2].

```
1 String studentPath = System.getProperty("studentPath");
2 System.setProperty("user.dir", studentPath);
3 File dir = new File(studentPath);
4 File[] directoryListing = dir.listFiles();
5 if (directoryListing != null)
6 {
7     for (File child : directoryListing)
8     {
9           String[] childNameSplit = child.getName().split("-");
10          File file = new File(studentPath + "/" + child.getName()
11                              + "/" + childNameSplit[0]
12                              + "_" + childNameSplit[1]);
13          URL url = file.toURL();
14          URL[] urls = new URL[] { url };
15          ClassLoader clazz = new URLClassLoader(urls);
16          Class recursion = clazz.loadClass("student.Recursion2");
17          Object recursionObj = recursion.newInstance();
18    }
19 }
```

Code Snippet 2: Code to set the current working directory for the student and also to fetch the required class for the student

- Next, we use threads where we create a thread to run the student's code and a thread to run the solution code. Let's name the threads studentThread and solutionThread respectively [Code Snippet 3, line 1 and line 3].

```
1 Thread studentThread = createStudentThread(recursion,
2                        studentRecursionClass, studentRecursionObj);
3 Thread solutionThread = createSolutionThread(recursion,
4                        solutionClass,solutionRecursionObj);
```

Code Snippet 3: Code to create student and solution threads.

- To have a synchronized flow, four locks are created, namely, studentLock - based on studentThread [Code Snippet 4, line 1], solutionLock - based on solution thread [Code Snippet 4, line 2], testLockStudent - mainThread's lock for student [Code Snippet 4, line 3], and testLockSolution - mainThread's lock for solution [Code Snippet 4, line 4].

```
1 static Integer studentLock = new Integer(100);
2 static Integer solutionLock = new Integer(101);
3 static Integer testLockStudent = new Integer(102);
4 static Integer testLockSolution = new Integer(103);
```

Code Snippet 4: Code to the locks.

- The studentThread is started [Code Snippet 5, line 3] and the mainThread waits on studentLock. The studentThread first executes the sum method and then calls the checkPoint method of the mainThread and passes along the codeId and the dataStructure required for comparison. Then the studentLock is notified [Code Snippet 5, line 7] and the studentThread waits on the testLockStudent lock as mainThread takes on the control from here.

```
1 if (!studentThread.isAlive())
2 {
3     studentThread.start();
4 }
5 synchronized (testLockStudent)
6 {
7     testLockStudent.notify();
8 }
9 synchronized (studentLock)
10 {
11    studentLock.wait(2000);
12 }
```

Code Snippet 5: Summary of the working of studentThread and studentLock.

- The mainThread starts the solutionThread [Code Snippet 6, line 3] and waits on solutionLock. The solutionThread first executes the sum method and then it calls the checkPoint method of the mainThread and passes along the codeId and the dataStructure required for comparison. Then the solutionLock is notified [Code Snippet 6, line 7]. and the solutionThread waits on testLockSolution lock as mainThread takes over the control from solutionThread.

```
1 if (!solutionThread.isAlive())
2 {
3     solutionThread.start();
4 }
5 synchronized (testLockSolution)
6 {
7     testLockSolution.notify();
8 }
9 synchronized (solutionLock)
10 {
11     solutionLock.wait(2000);
12 }
```

Code Snippet 6: Summary of the working of solutionThread and solutionLock.

- Now the mainThread has the dataStructures both from the student and the solution and does the comparison between the values [Code Snippet 7a, line 7] [Code Snippet 7b]. The current implementation prints the result of the comparison.

```
1 String[] checkPointIds = { "ordered", "charPair", "sum", "power",
2                       "exchange" };
3 for (String s : checkPointIds)
4 {
5     execute(studentThread, testLockStudent, studentLock);
6     execute(solutionThread, testLockSolution, solutionLock);
7     doCompare(s);
8 }
```

Code Snippet 7a: Summary of the working of solutionThread and solutionLock.

```
1 public static void doCompare(String s)
2 {
3     boolean result = false;
4     Object[] solutionValue = solutionObj.get(s);
5     Object[] studentValue = studentObj.get(s);
6     boolean returnValue = true;
7     if (solutionValue != null && studentValue != null)
8     {
9           for (int i = 0; i < studentValue.length; i++)
10          {
11                if (!(studentValue[i].equals(solutionValue[i])))
12                      result = false;
13          }
14          output.put(s, result);
15    }
16    else
17          result= false;
18    System.out.println("Result= " + result);
19    solutionStr = null;
20    studentStr = null;
21 }
```

Code Snippet 7b: Summary of the working of doCompare() method.

- After the comparison is done, the mainThread notifies testLockStudent lock and the student thread continues execution. The mainThread waits on studentLock [Code Snippet 5, line 11].
- The studentThread executes the power method and then it calls the checkPoint method of the mainThread and passes along the codeId and the dataStructure required for comparison. Then the studentLock is notified and the studentThread waits on the testLockStudent lock as mainThread resumes its execution.
- Next, the mainThread notifies testLockSolution lock and the solutionThread resumes its execution. The mainThread now waits on solutionLock [Code Snippet 6, line 11]. The solutionThread executes the power method and then it calls the checkpoint method of the mainThread and passes along the codeId and the dataStructure required for comparison. Then the solutionLock is notified and the solutionThread waits on testLockSolution lock as mainThread resumes its execution.
- The mainThread compares the values and prints the result of the comparison. It then notifies testLockStudent and testLockSolution.

The process repeats for each student and the grading is done.

5 References

[1] https://cloud.google.com/appengine/docs/java/tools/localunittesting

[2] http://junit.sourceforge.net/javadoc/org/junit/Test.html

[3] http://howtodoinjava.com/testng/testng-timeout-test-tutorial/

[4] https://github.com/junit-team/junit4/wiki/Assertions

[5] http://www.michaelminella.com/testing/the-concept-of-mocking.html

[6] https://en.wikipedia.org/wiki/Mockito

[7] http://www.tutorialspoint.com/easymock/

[8] http://www.thedance.net/~roth/TECHBLOG/powerMock.html

[9] http://www.baeldung.com/mockito-annotations

[10] http://stackoverflow.com/questions/20353846/mockito-difference-between-doreturn-and-when

[11] https://examples.javacodegeeks.com/core-java/junit/junit-runwith-example/

[12] http://stackoverflow.com/questions/10583202/powermockito-mock-single-static-method-and-return-object

[13] https://powermock.googlecode.com/svn/docs/powermock1.3.5/apidocs/org/powermock/core/classloader/annotations/PrepareForTest.html

[14] https://idodevjobs.wordpress.com/2015/01/17/powermock-mock-static-private-methods-powermock-tutorial/

[15] http://www.wideskills.com/java-tutorial/java-threads-tutorial

[16] http://whatis.techtarget.com/definition/multithreading

[17] http://www.oracle.com/technetwork/articles/java/classloaders-140370.html

[18] http://viralpatel.net/blogs/java-dynamic-class-loading-java-reflection-api/