**SOFTWARE TESTING**

**Junit Testing**

This lab introduces you to testing with JUnit, a standard testing framework for Java (and one used in upper-level CS courses). This lab will help you learn JUnit and think about testing in the presence of mutation.

**1****Problem to Test: A Vending Machine**

Start from this simple implementation of a vending machine. The vending machine offers two items: "Candy" and "Gum". The vending machine operations allow someone to deposit money, purchase items, see the current balance in the machine, and restock items. Your job will be to design tests for the vending machine and express them in JUnit.

==============================================================================================

import java.util.HashMap;

class VendingItem {

double price;

int numPieces;

VendingItem(double price, int numPieces) {

this.price = price;

this.numPieces = numPieces;

}

void restock(int pieces) {

this.numPieces = this.numPieces + pieces;

}

void purchase(int pieces) {

this.numPieces = this.numPieces - pieces;

}

}

/\*\*

\* Class for a Vending Machine. Contains a hashtable mapping item names to item data, as

\* well as the current balance of money that has been deposited into the machine.

\*/

class Vending {

private static HashMap<String, VendingItem> Stock = new HashMap<String,VendingItem>();

private double balance;

Vending(int numCandy, int numGum) {

Stock.put("Candy", new VendingItem(1.25, numCandy));

Stock.put("Gum", new VendingItem(.5, numGum));

this.balance = 0;

}

/\*\* resets the Balance to 0 \*/

void resetBalance () {

this.balance = 0;

}

/\*\* returns the current balance \*/

double getBalance () {

return this.balance;

}

/\*\* adds money to the machine's balance

\* @param amt how much money to add

\* \*/

void addMoney (double amt) {

this.balance = this.balance + amt;

}

/\*\* attempt to purchase named item. Message returned if

\* the balance isn't sufficient to cover the item cost.

\*

\* @param name The name of the item to purchase ("Candy" or "Gum")

\*/

void select (String name) {

if (Stock.containsKey(name)) {

VendingItem item = Stock.get(name);

if (balance >= item.price) {

item.purchase(1);

this.balance = this.balance - item.price;

}

else

System.out.println("Gimme more money");

}

else System.out.println("Sorry, don't know that item");

}

}

class Examples {

}

==============================================================================================

**2****Introduction to JUnit**

JUnit is a framework for writing test cases, similar in spirit to Tester, but with some significant differences. Both DrJava and Eclipse have built-in support for creating JUnit test cases (directly under the file menu for DrJava; under the File->New menu for Eclipse). The configuration options (especially under Eclipse) will make more sense if you first write a test suite manually.

If you don’t already have JUnit, [download](https://github.com/junit-team/junit/wiki/Download-and-Install)  to your working directory for this lab. Add the jar file to your classpath.

To use JUnit, you first need to include the following line:

|  |
| --- |
| import junit.framework.TestCase; |

A set of related tests then goes into a class that extends the JUnit TestCase class. If I want to define a class of vending-machine tests, for example, I could create:

|  |
| --- |
| public class VendingTest extends TestCase { |
| } |

The contents of this class are a series of test methods (similar in spirit to those we have done with Tester). All test methods must start with the letters test:

|  |
| --- |
| public class VendingTest extends TestCase { |
|  |
| public void test1() { |
| ... |
| } |
| } |

All that remains is to learn the contents of the test methods. Here are five common test operations you can use (the full list is in [the JUnit documentation](http://www.junit.org/apidocs/org/junit/Assert.html)):

* assertTrue(boolean) \\ passes if the expression is true
* assertFalse(boolean) \\ passes if expression is false
* assertSame(Object, Object) \\ passes if the two objects are in the same location in memory
* assertNotSame(Object, Object) \\ passes if the two objects are not in the same location in memory
* assertEquals(Object, Object) \\ passes if the two objects are identical according to their equals method.

So for example, I could write the following (uninteresting) test suite:

|  |
| --- |
| public class VendingTest extends TestCase { |
|  |
| public void test1() { |
| assertTrue(3==3); |
| } |
| } |

Let’s have you write some tests before we cover additional JUnit features.

**3****Problems, Part 1**

* Set up a JUnit test case to make sure that you can add money to a Vending Machine.
* Set up a JUnit test to make sure that you can buy Candy from a Vending Machine.

Did you notice that these two tests had some common infrastructure? Namely, both had to start with creating a vending machine and adding some money to the machine. Rather than repeat that code, it would be nice to tell JUnit to always set up that infrastructure before running our tests.

**4****JUnit, part 2**

Imagine that we had a method to create and initialize our vending machine:

|  |
| --- |
| Vending v; |
|  |
| public void setUp() { |
| v = new Vending(5,10); |
| v.addMoney(3); |
| } |

We would like Java to run this method for us before it runs the test cases. This is one of the features of JUnit. JUnit looks for a method with this exact name and type signature; if it exists, JUnit runs it before executing each test case. You can also provide a method called tearDown that JUnit will execute after each test runs (to clean up any lingering modifications to your data made during testing).

Each TestCase class can have its own setUp and takeDown methods. So if you have two sets of tests that need different setup and takedown operations, put them in two separate TestCase classes.

**5****Problems, part 2**

Consider the following TestCase:

|  |
| --- |
| public class VendTest extends TestCase { |
| static Vending v = new Vending(5,5); |
|  |
| public void setUp() { |
| v.addMoney(1.5); |
| } |
|  |
| public void test2() { |
| v.select("Gum"); |
| assertTrue(v.getBalance()==1); |
| } |
|  |
| public void test3() { |
| v.select("Candy"); |
| assertTrue(v.getBalance()==.25); |
| } |
| } |

* Run these tests. Comment out test2 and run them again. Now uncomment test2and remove the static from the definition of v.

What do you infer about the behavior of JUnit from these experiments? Write down a sentence or two to explain your reasoning clearly.

* If you had to leave v as static, how could you make this test suite work even if you commented out either of test2 or test3?
* Continue developing a test suite for the vending machine. Develop more sophisticated tests that check the interactions between operations in the vending machine.

**6****What to Turn In**

Turn in your .java files for this assignment.

References:

http://www.tutorialspoint.com/junit/index.htm