**Junit Assignments**

**Observer the following code and perform the given test cases below**

import java.util.Hashtable;

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 Hashtable<String, VendingItem> Stock = new Hashtable<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 {

}

**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.

**Note:** 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.

### 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.

**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.