4.1 Unit testing (classes)

Testbenches

Like a chef who tastes food before serving, a class creator should test a class before allowing use. A ***testbench*** is a program whose job is to thoroughly test another program (or portion) via a series of input/output checks known as ***test cases***. ***Unit testing*** means to create and run a testbench for a specific item (or "unit") like a method or a class.

PARTICIPATION ACTIVITY 4.1.1: Unit testing of a class.

Animation content:

```
Three different programs are shown, each outlined with a box.

The first box contains the following:

SampleClass
  Public item1
  Public item2
  Public item3

The second box contains the following:

User program
  Create SampleClass object
  Use public item 2

The third box contains the following:

SampleClassTester program
  Create SampleClass object
```

```
Test public item1
Test public item2
Test public item3
```

Animation captions:

1. A typical program may not thoroughly use all class items.
2. A testbench's job is to thoroughly test all public class items.
3. After testing, class is ready for use. The tester program is kept for later tests.

The testbench below creates an object, then checks public methods for correctness. Some tests failed.

Features of a good testbench include:

- Automatic checks. Ex: Values are compared, as in `testData.GetNum1() != 100`. For conciseness, only fails are printed.
- Independent test cases. Ex: The test case for GetAverage() assigns new values, vs. relying on earlier values.
- ***100% code coverage***: Every line of code is executed. A good testbench would have more test cases than below.
- Includes not just typical values but also ***border cases***: Unusual or extreme test case values like 0, negative numbers, or large numbers.

©zyBooks 12/08/22 21:49 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 4.1.1: Unit testing of a class.

Class to test: StatsInfo.java

```
public class StatsInfo {

   // Note: This class
intentionally has errors

   private int num1;
   private int num2;

   public void setNum1(int
numVal) {
      num1 = numVal;
   }

   public void setNum2(int
numVal) {
      num2 = numVal;
   }

   public int getNum1() {
      return num1;
   }

   public int getNum2() {
      return num1;
   }

   public int getAverage()
{
      return num1 + num2 /
2;
   }
}
```

Testbench: StatsInfoTest.java

```
public class StatsInfoTest {
   public static void main(String[] args)
{
      StatsInfo testData = new
StatsInfo();

      // Typical testbench tests more
thoroughly

      System.out.println("Beginning
tests.");

      // Check set/get num1
      testData.setNum1(100);
      if (testData.getNum1() != 100) {
         System.out.println("   FAILED
set/get num1");
      }

      // Check set/get num2
      testData.setNum2(50);
      if (testData.getNum2() != 50) {
         System.out.println("   FAILED
set/get num2");
      }

      // Check getAverage()
      testData.setNum1(10);
      testData.setNum2(20);
      if (testData.getAverage() != 15) {
         System.out.println("   FAILED
GetAverage for 10, 20");
      }

      testData.setNum1(-10);
      testData.setNum2(0);
      if (testData.getAverage() != -5) {
         System.out.println("   FAILED
GetAverage for -10, 0");
      }

      System.out.println("Tests
complete.");
   }
}
```



```
Beginning tests.
   FAILED set/get num2
   FAILED GetAverage for 10,
20
   FAILED GetAverage for -10,
```

```
0
Tests complete.
```

PARTICIPATION ACTIVITY 4.1.2: Unit testing of a class.

1) A class should be tested individually (as a "unit") before use in another program.
   - True
   - False

2) Calling every method at least once is a prerequisite for 100% code coverage.
   - True
   - False

3) If a testbench achieves 100% code coverage and all tests passed, the class must be bug free.
   - True
   - False

4) A testbench should test all possible values, to ensure correctness.
   - True
   - False

5) A testbench should print a message for each test case that passes and for each that fails.
   - True
   - False

Regression testing

Regression testing means to retest an item like a class anytime that item is changed; if previously-passed test cases fail, the item has "regressed".

A testbench should be maintained along with the item, to always be usable for regression testing.

Testbenches may be complex, with thousands of test cases. Various tools support testing, and companies employ *test engineers* who only test other programmers' items. A large percent, like 50% or more, of commercial software development time may go into testing.

PARTICIPATION ACTIVITY 4.1.3: Regression testing.

1) Testbenches are typically disposed of after use.
   - True
   - False

2) Regression testing means to check if a change to an item caused previously-passed test cases to fail.
   - True
   - False

3) For commercial software, testing consumes a large percentage of time.
   - True
   - False

Erroneous unit tests

An erroneous unit test may fail even if the code being tested is correct. A common error is for a programmer to assume that a failing unit test means that the code being tested has a bug. Such an assumption may lead the programmer to spend time trying to "fix" code that is already correct. Good practice is to inspect the code of a failing unit test before making changes to the code being tested.

Figure 4.1.2: Correct implementation of StatsInfo class.

```
public class StatsInfo {

   private int num1;
   private int num2;

   public void setNum1(int numVal)
{
      num1 = numVal;
   }

   public void setNum2(int numVal)
{
      num2 = numVal;
   }

   public int getNum1() {
      return num1;
   }

   public int getNum2() {
      return num2;
   }

   public int getAverage() {
      return (num1 + num2) / 2;
   }
}
```


PARTICIPATION ACTIVITY 4.1.4: Erroneous unit test code causes failures even when StatsInfo is correctly implemented.

Animation content:

Two code snippets are given, with the errors written in comments.

The first code snippet is as follows:


```
StatsInfo testData = new StatsInfo();
testData.setNum1(20);
testData.setNum2(30);
if (testData.getAverage() != 35) { // Wrong expected value
   System.out.println("FAILED GetAverage for 20, 30");
}
```

The second code snippet is as follows:

```
StatsInfo testData = new StatsInfo();
testData.setNum1(20);
testData.setNum1(30); // Text object's data not properly set
if (testData.getAverage() != 25) {
   System.out.println("   FAILED GetAverage for 20, 30");
}
```


Animation captions:

1. testData is instantiated and num1 and num2 are properly set to 20 and 30.
2. Whether a typo or miscalculation, the unit test expects 35 instead of 25, and fails. A wrong expected value is one reason a unit test may fail.
3. Calling setNum1 twice and not calling setNum2 is also an error, even if the expected value is now correct.
4. Not properly initializing the test object's data is another common error.

PARTICIPATION ACTIVITY 4.1.5: Identifying erroneous test cases.

Assume that StatsInfo is correctly implemented and identify each test case as valid or erroneous.

1) num1 = 1.5, num2 = 3.5, and the expected average = 2.5
   - ○ Valid
   - ○ Erroneous

2) num1 = 33, num2 = 11, and the expected average = 22
   - ○ Valid
   - ○ Erroneous

3) num1 = 101, num2 = 202, and the expected average = 152
   - ○ Valid
   - ○ Erroneous

Exploring further:

- JUnit testing framework for Java.

CHALLENGE ACTIVITY 4.1.1: Enter the output of the unit tests.

Note: Below, there's always a unit test failure.

422352.2723990.qx3zqy7

Start

Typ

CallRectangle.java | Rectangle.java

```
public class CallRectangle {
   public static void main(String[] args) {
      Rectangle myRectangle = new Rectangle();

      myRectangle.setSize(1, 1);
      if (myRectangle.getArea() != 1) {
         System.out.println("FAILED getArea() for 1
      }
      if (myRectangle.getPerimeter() != 4) {
         System.out.println("FAILED getPerimeter()
      }

      myRectangle.setSize(2, 3);
      if (myRectangle.getArea() != 6) {
         System.out.println("FAILED getArea() for 2
      }
      if (myRectangle.getPerimeter() != 10) {
         System.out.println("FAILED getPerimeter()
      }
   }
}
```

1

Check Next

CHALLENGE ACTIVITY 4.1.2: Unit testing of a class.

Write a unit test for addInventory(), which has an error. Call redSweater.addInventory() with argument sweaterShipment. Print the shown error if the subsequent quantity is incorrect. Sample output for failed unit test given initial quantity is 10 and sweaterShipment is 50:

```
Beginning tests.
   UNIT TEST FAILED: addInventory()
```

```
Tests complete.
```

Note: UNIT TEST FAILED is preceded by 3 spaces.

422352.2723990.qx3zqy7

```
1  // ===== Code from file InventoryTag.java =====
2  public class InventoryTag {
3     private int quantityRemaining;
4
5     public InventoryTag() {
6        quantityRemaining = 0;
7     }
8
9     public int getQuantityRemaining() {
10       return quantityRemaining;
11    }
12
13    public void addInventory(int numItems) {
14       if (numItems > 10) {
15          quantityRemaining = quantityRemaining + numItems;
16       }
17    }
```


Run

©zyBooks 12/08/22 21:49 1361995
John Farrell
COLOSTATECS165WakefieldFall2022