SOFT3202

Adam Ghanem

March 17, 2022

Contents

1 Tutorial 3 EXPORT

1.1 Types of Testing

1.1.1 Black Box and White Box

This has probably been echoed to you all for the hundredth time now but we will go through a quick revision of what these two types of testing mean.

Black-box testing means that the code that we are testing upon is obscured or hidden, in which we cannot be able to view it to find internal components. We only test merely by **passing input and checking for an expected output**. This is primarily used to **check for requirements which can be functional or non-functional.** For instance an example of a non functional test can be what we detailed last week with **timeout testing**.

White-box testing involves knowing the internals of the program and ensuring that it takes the correct path or calls statements in certain orders. We check things like branch and line coverage and if we cover the entire

path of the code. Usually software testers focus on black-box testing and the engineer on white-box testing since they know the internals.

In the below example try to identify which is an black-box test and which is a white-box test.

1. Example

```
@Test
public void testCreateDarkChocolateGetType() {

  WillyWonka wonka = new WillyWonka();
  Chocolate choco = wonka.makeChocolate("dark");
  assertThat(choco.getType(), is(equalTo("dark")));
}

@Test
public void testCheckingChocolateBothBranches() {
  WillyWonka wonka  =  new WillyWonka();
  Chocolate choco         = wonka.makeChocolate("dark");
  choco                             = wonka.makeChocolate("light");
  choco                             = wonka.makeChocolate("milk");
}
```

1.1.2 Regression

Regression testing is the idea of re-testing code after we make changes or additions to either our test suite or actual API. Let me give a scenario:

```
public class Circle {
  public int x;
  public int y;
  public double radius;
  // What happens if we change this value? What would it do our previous tests?
  public static final double PI = 3.1415926535;

  public Circle(int x, int y, double radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
```

```
  }

  public double perimeter() {
    return this.radius * 2 * PI;
  }

  public double area() {
    return this.radius * this.radius * PI;
  }

  public static void main(String[] args) {
    Circle circle = new Circle(0, 0, 4233);
    System.out.println(circle.area());
  }
}
```

A very trivial example, I know, but say we had already constructed our tests for the radius and perimeter, then one day our annoying boss strolls in and demands further accuracy for our Circle class. We extend the values of PI to 10 digits and call it a day. Instead, what we should always do is re-run previous tests, we should work like a pipeline to ensure we are always passing functionality that we have previously implemented.

1.1.3 Scales of Testing

Test	Description
Unit	Most atomic form of testing. Done mainly by developers, we have a lot of white-box te
Integration	Testing endpoints to see if the application is working. Multiple "units" are tested.
System	Testing whether the whole API is working and components are linking correctly.
Acceptance	Based on a normal user interaction, is it working and is it suitable for the user.

These are the four most common types of tests you will encounter and integration testing will be used heavily in your next assignment. *But why do we split these tests up and what is the point of learning about them, shouldn't ensuring functionality at the smallest/most atomic scale result in working code?* Well even if individual units work we do not ensure how their interaction with other components fair when implemented. You may have a working class with all its units verified but once interaction with another class exhibits unique or different behaviour your program ends up in spaghetti prison. Even units that are tested to perfection for a certain class A which

uses class B is not sufficient. It does not ensure that class B is implemented correctly and if it isn't it will conflict and cause issues with class A.

A similar argument applies to System testing and to Acceptance testing. Sadly this is not formal verification and we cannot ensure this theoretically from our unit tests. But also fortunately, for some, we do not have to formally verify our code :).

1.1.4 Code Coverage

Some would have encountered a form of code coverage tool, but for those who are new to this area, code coverage acts as a way to measure **the dgeree as to which your API, program etc has been tested.** Code coverage tools display statistics such that the percent/total coverage of statements, branches, lines etc. Jacoco, a code coverage tool, creates a viewable file that highlights sections of your code that has or hasn't been covered.

To add jacoco to your project make sure to specify:

```
plugins {
  id 'jacoco'
    }

// OPTIONAL: If you want to apply to every test run
test {
  finalizedBy jacocoTestReport // report is always generated after tests run
    }
jacocoTestReport {
  dependsOn test // tests are required to run before generating the report
    }
```

See more at Gradle Jacoco guide. Outputs will generally be saved to `build/jacocoHtml/...`

1.1.5 Performance Testing

Now this is the fun material, not that the previous material wasn't interesting but a bit less suprising. This section covers performance testing and how small optimisations to Java code can greatly influence the runtime of our programs.

1. Timeout Testing Last tutorial went through `@timeout` tests. Unfortunately this method is specific to the JUnit4 method, this week the JUnit5 method will be showcased:

```
public class MyMapTest {
  @Timeout(15)
  void testMapPutGetSpeed() {
    MyMap<String, Integer> map = new MyMap<>();
    map.put("oh hi mark", 10);
    map.get("oh hi mark");
  }
}
```

Timeouts can also be applied to the whole class. But it would also be helpful to test the speed of particular methods, e.g. just trying to `get` something from the map.

```
@Timeout(100)
public class MapTest {

  public void testMapGetSpeed() {
    MyMap<String, Integer> map = new MyMap<>();
    map.put("oh hi mark", 10);
    assertTimeout(
                  Duration.ofMilliSeconds(15),
                  () -> {
                    map.get("oh hi mark");
                  },
                  "This should be much faster.");
  }
}
```

Now for the fun stuff, performance optimisations. Try to go through the ArrayList and the for iterator example. For reference:

- Using `new ArrayList<>();add(first element);add(second element);add (add nth elements)` vs `Arrays.asList(elements)`
- standard for loop vs an iterator for a very long string list.

2. StringBuilder and String We have always been told to use `StringBuilder` when doing multiple `String` concatenations, but what is the point?

   Well the only way I can convince you is through an example. First try comparing the performance/time of using a few concatenations and compare their speeds, then bump this up to a 1000 then increase that by tenfold.

```
import java.lang.StringBuilder;
import java.util.function.Consumer;

public class StringPerformanceTest {

  /**
     Add a number of concatenations to a string.
     @param numConcats - the number of concatenations
  **/
  public void testStringConcatentations(int numConcats) {
    String str = new String();
    for (int i = 0; i < numConcats; i++) {
      str += "are we there yet?";
    }
  }

  /**
     Add a number of concatenations to a stringbuilder
     @param numConcats - the number of concatenations
  **/
  public void testStringBuilderConcatentations(int numConcats) {
    StringBuilder strBuilder = new StringBuilder();
    for (int i = 0; i < numConcats; i++) {
      strBuilder.append("are we there yet?");
    }
    strBuilder.toString();
  }

  public double getTime(Consumer<Integer> consumer, int numConcats) {
    long start = System.nanoTime();
    consumer.accept(numConcats);
    return (System.nanoTime() - start);
  }

  public static void main(String[] args) {
    StringPerformanceTest stringTest = new StringPerformanceTest();
    Consumer<Integer> stringMethod = stringTest::testStringConcatentations;
    Consumer<Integer> stringBuilderMethod = stringTest::testStringBuilderConcatent

    System.out.println("Testing 100 concatenations...");
```

```
    System.out.printf("String: %f\n", stringTest.getTime(stringMethod, 100));
    System.out.printf("StringBuilder: %f\n\n", stringTest.getTime(stringBuilderMet

    System.out.println("Testing 1000 concatenations...");
    System.out.printf("String: %f\n", stringTest.getTime(stringMethod, 1000));
    System.out.printf("StringBuilder: %f\n\n", stringTest.getTime(stringBuilderMet

    System.out.println("Testing 10000 concatenations...");
    System.out.printf("String: %.2f\n", stringTest.getTime(stringMethod, 10000));
    System.out.printf("StringBuilder: %.2f\n\n", stringTest.getTime(stringBuilderM

    System.out.println("Testing 100000 concatenations...");
    System.out.printf("String: %.2f\n", stringTest.getTime(stringMethod, 100000));
    System.out.printf("StringBuilder: %.2f\n\n", stringTest.getTime(stringBuilderM
  }
}
```

As an extension, try taking this `StringBuilder` and `String` outside of the for loops for multiple concatenations of strings and plotting the time differences. What is Java doing behind the scenes to the `String` concatenations and specifically when would you want to make use of `String` over `StringBuilder`?

(a) StringBuffer

When you are creating concurrent applications be wary of using StringBuilder. StringBuffer is a threadsafe and synchronised alternative to StringBuilder.

```
import java.lang.StringBuilder;
import java.lang.StringBuffer;
import java.util.function.Consumer;

public class StringPerformanceTest {

  /**
     Add a number of concatenations to a string.
     @param numConcats - the number of concatenations
  **/
  public void testStringBufferConcatentations(int numConcats) {
```

```
  StringBuffer str = new StringBuffer();
  for (int i = 0; i < numConcats; i++) {
    str.append("are we there yet concurrently?");
  }
}

/**
   Add a number of concatenations to a stringbuilder
   @param numConcats - the number of concatenations
**/
public void testStringBuilderConcatentations(int numConcats) {
  StringBuilder strBuilder = new StringBuilder();
  for (int i = 0; i < numConcats; i++) {
    strBuilder.append("are we there yet?");
  }
}

public double getTime(Consumer<Integer> consumer, int numConcats) {
  long start = System.nanoTime();
  consumer.accept(numConcats);
  return (System.nanoTime() - start) / 1000000000;
}

public static void main(String[] args) {
  StringPerformanceTest stringTest = new StringPerformanceTest();
  Consumer<Integer> stringBufferMethod = stringTest::testStringBufferConcat
  Consumer<Integer> stringBuilderMethod = stringTest::testStringBuilderConc

  System.out.println("Testing 100 concatenations...");
  System.out.printf("StringBuffer: %f\n", stringTest.getTime(stringBufferMe
  System.out.printf("StringBuilder: %f\n\n", stringTest.getTime(stringBuild

  System.out.println("Testing 1000 concatenations...");
  System.out.printf("StringBuffer: %f\n", stringTest.getTime(stringBufferMe
  System.out.printf("StringBuilder: %f\n\n", stringTest.getTime(stringBuild

  System.out.println("Testing 10000 concatenations...");
  System.out.printf("StringBuffer: %.2f\n", stringTest.getTime(stringBuffer
  System.out.printf("StringBuilder: %.2f\n\n", stringTest.getTime(stringBui
```

```
        System.out.println("Testing 100000 concatenations...");
        System.out.printf("StringBuffer: %.2f\n", stringTest.getTime(stringBufferB
        System.out.printf("StringBuilder: %.2f\n\n", stringTest.getTime(stringBuil

        System.out.println("Testing 1000000 concatenations...");
        System.out.printf("StringBuffer: %.2f\n", stringTest.getTime(stringBufferB
        System.out.printf("StringBuilder: %.2f\n\n", stringTest.getTime(stringBuil
      }
    }
```

3. Iterator vs Standard For Loop

```
import java.util.function.Consumer;
import java.lang.Runnable;
import java.util.List;
import java.util.ArrayList;

public class IteratorPerformanceTest {

  private List<String> list;

  /**
     Test standard for loop for a long list of strings
  **/
  public void testStandardForLoop() {
    for (int i = 0; i < this.list.size(); i++) {
      String str = this.list.get(i);
    }
  }

  /**
     Test iterator method for a long list of strings
  **/
  public void testIterator() {
    for (String str : this.list) {
    }
  }
```

```
  /**
     Get the time of a runnable method
     @param runnable - the method to run
     @return the time it took to run the function
     **/
  public double getTime(Runnable runnable) {
    long start = System.nanoTime();
    runnable.run();
    return (System.nanoTime() - start);
  }


  /**
     Make the list of a given size
     @param size - the size of  the list to make
     **/
  public void makeList(int size) {
    List<String> list = new ArrayList<>();
    for (int i = 0; i < size; i++) {
      list.add("over 9000");
    }
    this.list = list;
  }

  public static void main(String[] args) {
    IteratorPerformanceTest iteratorTest = new IteratorPerformanceTest();
    Runnable standardForLoop = iteratorTest::testStandardForLoop;
    Runnable iteratorMethod = iteratorTest::testIterator;

    System.out.println("Testing 100000000 list size...");
    iteratorTest.makeList(100000000);
    System.out.printf("For Loop: %f\n", iteratorTest.getTime(standardForLoop));
    System.out.printf("Iterator: %f\n\n", iteratorTest.getTime(iteratorMethod));
  }
}
```

4. List Add vs AsList

```
import java.util.function.Consumer;
```

```
import java.lang.Runnable;
import java.util.List;
import java.util.ArrayList;

public class AddAsListTest {

  private String[] arr;

  /**
     Test standard for loop for a long list of strings
  **/
  public void testAdd() {
    List<String> temp = new ArrayList<String>();
    for (int i = 0; i < arr.length; i++) {
      temp.add(this.arr[i]);
    }
  }


  /**
     Test as list method for a long list of strings
  **/
  public void testAsList() {
    List<String> temp = new ArrayList<String>(Arrays.asList(this.arr));
  }


  /**
     Get the time of a runnable method
     @param runnable - the method to run
     @return the time it took to run the function
     **/
  public double getTime(Runnable runnable) {
    long start = System.nanoTime();
    runnable.run();
    return (System.nanoTime() - start);
  }


  /**
```

```
     Make the list of a given size
     @param size - the size of  the list to make
     **/
  public void makeList(int size) {
    arr = new String[size];
    for (int i = 0; i < size; i++) {
      arr[i] = "" + i;
    }
  }


  public static void main(String[] args) {
    AddAsListTest addAsListTest = new AddAsListTest();
    Runnable testAdd = addAsListTest::testAdd;
    Runnable testAsList = addAsListTest::testAsList;

    System.out.println("Testing 100000000 list size...");
    addAsListTest.makeList(10000000);
    System.out.printf("Add Test: %f\n", addAsListTest.getTime(testAdd));
    System.out.printf("Add AsList Test: %f\n\n", addAsListTest.getTime(testAsList)
  }
}
```

1.2 EXTRA: Power Mocks

Power mocks came from a legacy riot that people wanted more power with their mocks. Developers wanted a way to test for `private` variables, `final` classes and change the actual class under test. This led to the creation of power mocks. **It treats the code as a white-box from top to bottom.** It works based on Reflections. It will decompile the bytecode using the JVM and then inspect the code to do whatever you want it to, in a way modifying the source file into text.