Testing and debugging software

Author(s): Mihai Pop

last changed 03/14/16

The code you implement must do exactly what the project description requires, otherwise you end up losing points due to failed tests. More importantly, code that you write for a client or employer must do exactly what was specified. Otherwise, in the best case, you may not get paid. In the worst case, your code may lead to costly or even deadly disasters, as you've found out in one of the first reading materials provided to you in the class.

To ensure your code does exactly what it's supposed to do you must **test** it. Specifically, you must provide a series of inputs and make sure that the outputs, and the general behavior of the code (e.g., whether the code reports an error for ill-formed inputs) matches the specified requirements.

Testing is a critical part of the software development process. Without it, software bugs may remain undetected, raising the possibility that they will emerge right when they are most damaging. Testing is so important that many consider that you should start by designing and writing the tests before writing your first line of code.

Below we will go through some examples of the process used to develop effective tests for your code, as well as how you may use these tests to find and fix errors, i.e., to **debug**.

Unit testing in Java

Here we'll mainly focus on **unit testing** – tests developed for specific small segments of code, usually for individual methods. This is in contrast to **end-to-end testing**, which makes sure that the whole software package behaves as expected. While end to end testing is an effective way to ensure your code is correct, errors identified during this process are much harder to debug as they may occur in any of the many components that make up a finished software product. Unit testing allows errors to be found more quickly, and also makes it easier to debug errors identified during end-to-end testing, as you already know all the individual parts work correctly (i.e., errors must involve the way the parts are put together).

The Java language provides a framework for implementing unit tests, framework called JUnit. This framework provides you with the 'machinery' to define new tests, as well as a collection of helper methods that help you check that the code performs as expected.

To get started with testing in Eclipse, simply right-click on the project folder within the "Package Explorer" pane, then select New->JUnit Test Case from the pop-up menu. You can now provide a name to the new test class, and then click 'Finish' to create it. For now let's assume the name is `StudentTests`. A new window will appear containing the newly created file, containing the stub for your first test:

```
@Test
public void test() {
     fail("Not yet implemented");
}
```

The text `@Test` is a 'decorator' and it tells Java that the method following it is a JUnit test.

The method provided to you simply creates a test that is guaranteed to fail due to the statement:
`fail("Not yet implemented");`
which gets executed every time the method `test()` is executed, and it simply tells Java that the test has failed. Clearly this is not particularly useful, and below we'll work through some examples on how to fill in this stub code to actually perform useful tests.

Testing the isOdd method

Imagine that you are asked to write a method called `isOdd` that takes an integer parameter and returns true if that parameter is an odd number, and false otherwise. This description should be very familiar to you, as you've seen a number of such descriptions in your projects.

Before starting to write tests or the method itself, we need to decide on the 'prototype' for the method. Specifically, we must write out the first line in the description of the method, that explicitly writes out the required inputs and return value. We do that in the file where we'll write our actual code, not within the `StudentTests.java` file.

We will start with the keywords `public static` as we have previously done in the course – soon we'll learn their meaning but for now it's not essential. Then we declare the **type** of the method, or more precisely the type of the value returned by the method. Since the description requires us to return values called 'true' and 'false', we know that our method must return a boolean, leading to the prototype:

```
public static boolean isOdd()
```

One thing is still missing – the parameter passed to the method. Also from the specification of the requirements we know that this parameter is an integer, i.e., a variable of type `int`. We

declare this variable within the parentheses next to the name of the method, then we add the braces that will surround the code that we will write:

```
public static boolean isOdd(int num)
{
}
```

Note that we have named the parameter `num` – the value passed to this method will be visible within the body of the `isOdd` method as the variable `num`. This variable cannot be redefined within the body of the method, nor do we need to assign anything to it given that its value will be provided by the code executing the `isOdd` method.

We can now leave the file where we are writing the code and return to the JUnit tests cases we have started. Yes, we have yet to write any code, yet we are already starting to write the tests.

Before writing anything, though, we need to spend a bit of time thinking about how exactly we plan to test the (as of yet unwritten) method `isOdd`. In this case, its definition is straight forward so writing the tests should not be too hard.

First, let us return to our definition of what a test is – it is simply providing the code an input and checking that the output is what it should be. The description of the project already gives us some indication of what the method should do – when provided with an odd number it must return `true`, and when provided with an even number it must return `false`.

This immediately gives us an idea of what inputs to provide the method – we need to provide it with one or more even numbers and check that the output is indeed `false`. We also need to provide it with one or more odd numbers and check that the output is indeed `true`.

Let us break these two cases into two separate test cases. We'll simply cut and paste the original stub of the test (the method called `test()` shown above), and rename one of the methods `testEvens` and the other `testOdds`:

```
@Test
public void testEvens()
{
     fail("Not yet implemented");
}

@Test
public void testOdds()
{
     fail("Not yet implemented");
}
```

Just to make sure everything is OK, you can run these tests by clicking on the white triangle at the top of the Eclipse window, just like you do for regular code. In this case, however, you will see a new panel appear, titled "JUnit" and within it you will see the result of the two tests, both of which will appear as failed.

We can now proceed to fill in the body of the tests. As the name of the test methods indicate, we want to apply the `isOdd` method to a series of even and odd numbers and see if the results make sense. What numbers? At some level it doesn't matter much – we can simply pick several arbitrary ones and see what happens.

```
@Test
public void testEvens()
{
     if (isOdd(2)) {
          fail("2 is not odd");
     } else if (isOdd(16)) {
          fail("16 is not odd");
     }
}

@Test
public void testOdds()
{
     if (! isOdd(3)){
          fail("3 is odd");
     } else if (! isOdd(15)) {
          fail("15 is odd");
     }
}
```

Note that we have made use of the 'fail' method provided by JUnit and we have used conditional statements to check that the `isOdd` method performs the right task for several arbitrary inputs. If the fail methods do not get executed (which is the case for a correct isOdd method), then Java will report the tests as successful.

You can easily see how the code we wrote above can become quite hairy as we try out more and more inputs. To make it easier to write such tests in a more concise manner, the JUnit framework provides several other methods beyond `fail`. These take the form or assertions – statements we make that we believe hold true when the code being tested is correct. There are three such methods that you will find using a lot in your tests:

```
void assertTrue(String text, boolean value);
void assertFalse(String text, boolean value);
void assertEquals(String text, SomeType expected, SomeType actual);
```

The provided text is optional for the three methods.

`assertTrue` causes the test to fail, and outputs the optional text, if the value provided is **not** `true`.
`assertFalse` causes the test to fail, and outputs the optional text, if the value provided is **not** `false`.
`assertEquals` causes the test to fail, and outputs the optional text, if the two parameters `expected` and `actual` are not equal. The text SomeType is not an actual Java construct, just a place-holder to indicate the assertEquals method can take parameters of different types.

It is important to stress again as it can be confusing – when using `assertTrue` the method causes the test to fail if the parameter is **not** `true`. What we are doing here is stating the fact that in the correct code the parameter should be true, and a deviation from this statement indicates an error.

We can rewrite our test methods in a cleaner way using these methods as follows:

```
@Test
public void testEvens()
{
     assertFalse("2 is not odd", isOdd(2));
     assertFalse("16 is not odd", isOdd(16));
}

@Test
public void testOdds()
{
     assertTrue("3 is odd", isOdd(3));
     assertTrue("15 is odd", isOdd(15));
}
```

Or using the `assertEquals` method:

```
@Test
public void testEvens()
{
     assertEquals("2 is not odd", false, isOdd(2));
     assertEquals("16 is not odd", false, isOdd(16));
}

@Test
public void testOdds()
{
     assertEquals("3 is odd", true, isOdd(3));
     assertEquals("15 is odd", true, isOdd(15));
}
```

When running this code, if an error is found and the test fails, the JUnit framework will also

output the text we provided as well as additional helpful information, such as the expected and actual values that led to `assertEquals` to fail, making the debugging process easier.

Now that the mechanics of writing tests are sorted out, we can get back to figuring out what inputs to provide to our code to make sure we haven't made any mistakes. We have to ask ourselves whether the inputs we provided are representative of all the inputs that could be received by our program. In particular, it is useful to think of unusual inputs that would normally not be the first ones that come to mind. What is unusual depends on the actual code being tested. Here, we can wonder what will happen if the value 0 is provided to the code. The value 0 is tricky because mathematicians themselves may argue whether it is an even or an odd number, and the specification text doesn't give us much guidance. Let us assume for now that the definition of 'even' is a number that yields a remainder of 0 when divided by 2, i.e., we'll define 0 to be even. We can test the (as of yet unwritten) code to make sure this is the case by creating a new test:

```
@Test
public void testZero()
{
     assertFalse("0 is even", isOdd(0));
}
```

What other numbers could be unusual? What about negative numbers? The rules of evenness and oddness should still apply and we can verify this with two new tests:

```
@Test
public void testNegativeEvens()
{
     assertFalse("-4 is not odd", isOdd(-4));
}

@Test
public void testNegativeOdds()
{
     assertTrue("-3 is odd", isOdd(-3));
}
```

Coming up with all the different scenarios that may trip up our code may be difficult, thus we can consider creating random numbers to make sure the results hold in a more general setting. To do so we can use the `Math.random()` method – a method that returns a random `double` between 0 and 1 (excluding 1). To increase the range of the numbers created we can simply multiply the result by another number – `Math.random() * 10` creates a random double between 0 and 10.

To convert the resulting `double` to an `int` (the type expected by the `isOdd` method) we can

either cast it: `(int) (Math.random() * 10)`, or use one of the `Math.floor()`, `Math.ceil()`, or `Math.round()` methods in case we want to guarantee a specific convention for narrowing a `double` to an `int` (see the documentation for these methods in the Math class).

One tricky issue is that we want to create numbers for which we already know the answer. We can easily extend the approach above to create random even or odd numbers by multiplying the random number we obtained by 2, then adding a 1 to create a random odd number.

We can now create two or more test methods that make sure the isOdd method behaves correctly for random numbers:

```
@Test
public void testPositiveRandom()
{
     int rand = (int) (Math.random() * 100000);
     assertFalse(2 * rand + " is even", isOdd(2 * rand);
     assertTrue( (2 * rand + 1) + " is odd", isOdd(2 * rand + 1);
}

@Test
public void testNegativeRandom()
{
     int rand = (int) (Math.random() * 100000);
     assertFalse(-(2 * rand) + " is even", isOdd(-2 * rand);
     assertTrue((-2 * rand + 1) + " is odd", isOdd(-2 * rand + 1));

}
```

To sum it all up, we have created 7 tests (`testEvens, testOdds, testZero, testNegativeEvens, testNegativeOdds, testPositiveRandom, testNegativeRandom`) before writing a single line of code in our `isOdd` method.

We can now return to filling in the body of the method, knowing that any coding mistakes will become apparent when we run the tests. We can also be fairly confident that if all the tests are successful our method correctly meets the specification.

Let's go ahead and fill the body of the method and see what happens. To decide whether a number is even or odd, we'll simply check whether the remainder of the division by 2 is 1 (indicating an odd number) or not (indicating an even number). Filling in the prototype code we have started we get:

```
public static boolean isOdd(int num)
{
     if (num % 2 == 1) {
          return true; // it's odd
     } else {
          return false;
     }
}
```

Now we return to the testing code and run it. We'll immediately see that two tests fail: `testNegativeOdds` and `testNegativeRandom`. A quick look at the report produced by JUnit for the failed tests gives us hints that will help us in debugging. For example, the string output by the `testNegativeOdds` method says "`-3 is odd`" and that the failure occurred in the `assertTrue` method, indicating that our `isOdd` method returned the value `false` for the input -3. We can now return to our `isOdd` method and start tracing its execution to see what went wrong. You can do this by hand on paper, or you can use the debugger provided by Eclipse.

```
public static boolean isOdd(int num)  - here num is -3
{
     if (num % 2 == 1) {              - here we compute -3 % 2
          return true; // it's odd    - should return true...
     } else {
          return false;
     }
}
```

Clearly something is wrong, but what? To find out we can use the `System.out.println` statement to check the value of different expressions throughout the code:

```
public static boolean isOdd(int num)
{
     System.out.println("Here num is ", num);
     System.out.println("The modulo is ", num % 2);
     if (num % 2 == 1) {
          System.out.println("I'm returning true");
          return true; // it's odd
     } else {
          System.out.println("I'm returning false");
          return false;
     }
}
```

Running this code will yield:

```
Here num is -3
The modulo is -1
I'm returning false
```

Aha! Mystery solved. Our test `if (num % 2 ==1)` is incorrect for negative numbers, as the remainder itself is a negative number. We can now fix the code:

```
public static boolean isOdd(int num)
{
      System.out.println("Here num is ", num);
      System.out.println("The modulo is ", num % 2);
      if (num % 2 == 1 || num % 2 == -1) {
            System.out.println("I'm returning true");
            return true; // it's odd
      } else {
            System.out.println("I'm returning false");
            return false;
      }
}
```

Then remove or comment out our debugging statements. We can now return to the test code, rerun the tests, and convince ourselves that the code now successfully passes all the tests.

We are now ready to deliver the completed assignment to our client, confident that we have implemented the correct specification.

Testing and debugging – caveats

The testing procedure we have (painfully) slowly developed above can help us identify and correct some errors in the code, however it doesn't eliminate the possibility that errors still remain. Furthermore, the testing code itself is code, and can also contain errors that lead in either false alarms, or worse – in the failure to identify errors. In real software projects substantial efforts are developed to testing, taking up at least as much effort (time/money) as the coding process itself.

Substantial research efforts (including efforts conducted in our department) are also devoted to creating approaches that allow the code to be automatically verified. The Java language itself is an example – the annoying quirks of the language such as the inability to automatically narrow variables, or redefine variables within blocks are actually conventions put in place that allow the Java compiler to detect and report some of the most common coding errors.

In this class you will also get additional help from the FindBugs system developed by Prof. Pugh in our department. This package is integrated with the submit server and highlights regions in your code that either contain bugs or contain code that doesn't follow good coding practices and may contain bugs. You will see these reported in the submit server in addition to the result of the public and release tests.

More on testing strategies

For another example on how to think about testing, let us try to build the tests a function called log10 that takes in a double and returns another double which is the logarithm base 10 of the input parameter.

First, the prototype of this method is:

```
public static double log10(double num)
{
}
```

We won't go into the full details of the implementation of the tests and leave that to you as an assignment. Instead we'll try to figure out what combinations of inputs and outputs we will use.

First, we have to remember a few things about the logarithm function. We know that this function crosses the x axis at position 1 (log10(1) is 0). This becomes our first test:

```
assertEquals("at 1 should be 0", 0, log10(1));
```

We also know that the logarithm takes the value 1 at exactly the value of its base (log10(10) is 1), leading to a second test:

```
assertEquals("at 10 should be 1", 1, log10(10));
```

We can extend this further by noticing that the logarithm of 100 should be 2, of 1000 should be 3, and so on.

But what about other numbers – will the method be correct for a random number? One way to check this is by using a calculator:

```
assertEquals("at 2 should be 0.3010299", 0.3010299, log10(2));
```

But isn't this cheating? Not quite – a common testing strategy involves having different teams write the same code and comparing the outputs. If they match, it's hopefully unlikely that both teams made the same mistake, hence the code is correct.

While the code written above is not quite cheating (we are using a pocket calculator as an **oracle** – a gold standard against which we compare our answer), it will likely fail even for a correct implementation of the `log10` method. Why? It has to do with the limited precision of our representation of numbers within a computer, and the quirks of the floating point convention. When comparing two floating point numbers (of type `double` or `float`) we cannot assume they will match exactly even if they attempt to encode the same number. Rather, they will most likely differ at some low significance digit, such as, for example, two representations of π: 3.141592653589 and 3.14159265359. The difference is hardly significant enough to write home about (or to lose points on a project), yet the `assertEquals` method

will consider the two values to be different. To help us avoid such situations, and to be more precise about the precision to within which we want the answer to be, the JUnit framework provides a special assertEquals method for doubles with the prototype:

`public void assertEquals(double expected, double actual, double delta)`
(the `String` parameter is omitted for clarity) which declares the variables `expected` and `actual` to be the same if the difference between them is less than or equal to the value of `delta`.

In the π example:

```
assertEquals(3.141592653589,
             3.14159265359,
             0.0001)
```

would pass the test as the two values are within 1 in 10,000 from each other, while the statement:

```
assertEquals(3.141592653589,
             3.14159265359,
             0.0000000000001)
```

would fail. Note that wrapping the text as I did above is perfectly OK in Java and it is even preferred as it makes the code easier to read and understand. Compare this to the one line version:

```
assertEquals(3.141592653589,3.14159265359,0.0000000000001)
```

The correct version of our test for the log10 method, thus, is:

```
assertEquals("at 2 should be 0.3010299",
               0.3010299, log10(2), 0.00001);
```

Coming back to the logarithm, we can also try to test it with random numbers. But what should the outputs be. While we may not know what any specific output may be, we know a few properties of the logarithm function that we can use to develop tests. For example:

log(a * b) = log(a) + log(b)

log(a / b) = log(a) - log(b)

leading to the test:

```
double x = Math.random() * 1000; // no need for a cast anymore
double y = Math.random() * 1000;
assertEquals(log10(x) + log10(y), log10(x * y), 0.0000001);
assertEquals(log10(x) – log10(y), log10(x / y), 0.0000001);
```

Note that we used the 'delta' parameter to make sure our tests don't fail simply because of the limited precision of the `double` type.

In essence, what we've done is check the value of the result at several specific inputs and also checked some more general properties of the function implemented by the `log10` method, making it more likely that if all tests pass the method truly implements the correct function.

One thing we've ignored here are the special/unusual inputs. The log function is not defined for numbers less than or equal to 0. We could provide such inputs to the code to make sure it fails appropriately. We'll learn more about how to do that in a correct way when we discuss exceptions.

Code coverage

An important concept related to testing is **code coverage** – the percentage of the code that gets executed when running one or more tests. Coding errors/bugs that occur within segments of code that are not executed cannot be detected by the tests we came up with. Part of the art of creating tests is coming up with inputs that force the code to execute all the statements, ensuring that bugs cannot 'hide'.

As an example, let's take a simple method `convertTemp` that takes two parameters – a `double` and a character – representing a temperature in one of the popular scales ('F' for Fahrenheit, 'C' for Celsius, 'K' for Kelvin, etc.). The method then returns a double representing the corresponding temperature in Celsius (the **right** scale we should all be using).

The method might look like:

```
public static double convertTemp(double temp, char scale)
{
     if (scale == 'F'){
        return (temp - 32) * 5 / 9;
     } else if (scale == 'K'){
       return temp - 273.15;
     } else if (scale == 'C'){
      return temp * 9 / 5 + 32;
     }
}
```

Assume our test methods contain the following statements:

```
assertEquals(0, convertTemp(32, 'F'));
assertEquals(0, convertTemp(273.15, 'K'));
```

Together, these methods only cover part of the code, as shown in the example below. The line marked with a * in the margin is never executed, i.e., it is not covered by the tests, making us miss the fact that the code incorrectly converts Celsius to Fahrenheit instead of simply returning the Celsius input unchanged.

```
public static double convertTemp(double temp, char scale)
{
  if (scale == 'F'){
    return (temp - 32) * 5 / 9;
  } else if (scale == 'K'){
    return temp - 273.15;
  } else if (scale == 'C'){
*   return temp * 9 / 5 + 32;
  }
}
```

The addition of one more test can help us find the bug:

```
assertEquals(15, convertTemp(15, 'C'));
```

For simple programs such as the one discussed above we can figure out whether our tests cover the code properly by tracing the execution of the code by hand. For larger programs such strategies become quickly unmanageable. Instead, software packages exist that can help compute the coverage of the code you are testing and even highlight within the code which lines were missed by a particular testing setup.

Exercises

For all examples below, start by defining the prototype of the method, without implementing its body, then create the JUnit tests that ensure the code correctly follows the specification.

For all of these, think about how you would create inputs for which the output is known. Also, think about what type of inputs may be unusual and cause your code to fail or behave incorrectly.

1. Design and implement the test framework for a method named `squareRoot` that takes a parameter of type `double` and returns another `double` representing the square root of of the input parameter. If the method is provided with a negative number it must return the number -1 indicating that an error has occurred.
2. Design and implement the test framework for a method named `length` that takes a parameter of type `String` and returns an integer representing the length of the string.
3. Design and implement the test framework for a method called `rotateLetter` that takes two parameters, a character `cin` and an integer `n`, and returns a new character `cout`, which is exactly `n` characters from `cin` in lexicographic order, wrapping around the end of the alphabet if necessary. The method must work for both lower case and upper case characters (the output should have the same case as the input) and for both positive and negative input integers.
4. Design and implement the test framework for the assignment in Project 2.
5. Design and implement the test framework for the following Project 3 flags: Q, C, F.