CITS5501 Software Testing and Quality Assurance
Semester 1, 2020
Workshop 3 – BlueJ walk-through

Installing BlueJ

If you are using a lab computer – BlueJ is already installed.

If you are using a laptop or home computer – BlueJ can be obtained from https://www.bluej.org/versions.html.

Select the appropriate download for your OS – Windows, MacOS or Linux.

Creating a skeleton class

- Select 'Project/New Project' - specify a name and location for a project.
- Select 'Edit/New Class' - specify a name for your new class (e.g. "Box").
- Right click on the new class, click "compile" to ensure it compiles without error.

Creating a skeleton test class

- Right click on your class, select "Create Test Class".
- A class with a name like "TestBox" (depending on the name of your class) will be created.
- Right click on the new class to compile it; you can also run the tests, once some tests (see the "Run tests" button on the left) have been created.

JUnit methods and sample code

- See https://github.com/junit-team/junit4/wiki/Getting-started for sample code for tests.

Writing tests

We can start writing our tests in `BoxTest`, before the code for `Box` is even finished – useful when some other developer is writing `Box`, but we are given the task of creating tests.

And even if we are the only developer working on `Box`, it can *still* be useful to think about the specifications and tests first, before we even begin writing any code for `Box`; this is used in the methodology called “Test-Driven Development”.

We know our `Box` class needs a constructor ... though the specification doesn’t say what it should be passed. We might decide the following would be good:

```
public Box(int x1, int y1, int x2, int y2)
```

So let us write skeleton code for the constructor (replacing the existing constructor and class body), and document any decisions we’ve made about the class specification:

```
/** Creates a new Box instance, when given
 * coordinates for 2 points, (x1,y1) and (x2,y2).
 */
public Box(int x1, int y1, int x2, int y2) {
}
```

So our class as a whole will look like this:

```
/**
 * Write a description of class Box here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Box {

  /** Creates a new Box instance, when given
   * coordinates for 2 points, (x1,y1) and (x2,y2).
   */
  public Box(int x1, int y1, int x2, int y2) {
  }

}
```

And we now already have something we can test - if we go to `BoxTest`, we can add the following:

```
@Test
public void constructorSucceeds() {
  Box myBox = new Box(5,5, 10,10);
}
```

This test doesn’t actually assert anything – all it does is check whether the constructor can be called, without raising an exception.

We can compile and run our tests using the “Compile” and “Run tests” buttons on the left of the BlueJ window.

Let’s add a test for the `area` method. First, we will have to decide what the signature will be, and add skeleton code for the method. Something like:

```java
/** Returns the area of the Box.
 */
int area() {
  return 0;
}
```

We need to include a `return 0` statement, else the code won’t compile. Again, the method doesn’t *do* anything, but it’s enough for us to start writing tests against – let’s add the following test to `TestBox`:

```java
@Test
public void calculatesExpectedArea() {
  Box myBox = new Box(5,5, 10,10);
  int expectedArea = 25;
  int actualArea = myBox.area();
  assertEquals(expectedArea, actualArea);
}
```

We can run our tests, and see that one test passes, and that the other test (`calculatesExpectedArea`) fails – click on the failure to see more details about it.

We might now start adding some actual code to our `Box` class, filling in the constructor and the `area` method, to get something like:

```java
/**
 * Write a description of class Box here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Box {
  // instance variables to store x and y
  // coordinates of 2 points
  private int x1, x2, y1,y2;

  /**
   * Constructor for objects of class Box
   */
  public Box(int x1, int y1, int x2, int y2) {
    this.x1 = x1; // specifies that instance variable
```

```
                    // x1 of this object should be
                    // initialized using the parameter
                    // x1 passed in
    this.y1 = y1;
    this.x2 = x2;
    this.y2 = y2;
  }

  /** Returns the area of the Box.
   */
  int area() {
    return (x2-x1) * (y2 - y1);
  }
}
```

And we can work on improving and adding to our tests. For instance, were the points $(5, 5)$ and $(10, 10)$ good values to choose for our `calculatesExpectedArea` test? The height and width are the same, so this wouldn't detect an error where (say) we accidentally calculated area as height $\times$ height.

We could add a better test, and document our reason for adding it:

```
/** Ensure area is calculated correctly
 * when height and width of box are not
 * the same
 */
public void calculatesExpectedArea2() {
  Box myBox = new Box(5,5, 10,20);
  int expectedArea = 75;
  int actualArea = myBox.area();
  assertEquals(expectedArea, actualArea);
}
```

And if our code doesn't pass, we can try to debug it until the tests do pass.

And we can think about what other sorts of tests might be useful:

- What if the point $(x1, y1)$ is actually above the point $(x2, y2)$ – what should `area()` return? Is it sensible to return a negative area? Does our code for `area` need to be improved?
  (Hint: take a look at the `Math.abs` method from the Java standard library.)
- What if the two points are the same – is that allowable?

And we can then move on to thinking about the specification of `intersection`. In future weeks, we'll see ways of measuring how *much* of our code is exercised by our tests ("code coverage").