2 The Variety of Unit Test Scenaria

The goal of this lab is to make you comfortable with designing unit tests using the *tester* library and to become familiar with some of the test case scenaria that the library supports.

2.1 Basic Practice

We will now focus on the design of methods and the design of tests as an integral part of the method design.

Download the file **GraphUSA.zip** into a temporary directory, and unzip it. Start a new project *GraphUSA*. Highlight the project in the *Project Explorer* and *Import* into the project all *java* files from the unzipped folder.

From your *EclipseJars* folder add to your project's path the following library files:

- **tester.jar**
- **draw.jar**
- **geometry.jar**
- **colors.jar**

Now run the project. It will show you a simple map of the 48 capitals of the lower 48 US states, with a highlighted (circuitous) route from Alabama to Maine. (Move the second Canvas over, then use the space bar to see the route legs one at a time.)

We have written the code for drawing of the map. You may look at what it entails - it is quite straightforward. Equally easy is the handling of the interactive route display. What is missing in the code is a number of tests. Your first task is to add at least some of the missing tests.

1. Add the tests for the method `toPosn` for the class `Loc` and for the class `City`. You need to make sample data of the type `Loc`. My suggestion is to take the locations of the first three cities, you can then reuse the results for the tests for the first three cities.

2. Add the (inexact) tests for the method `distanceTo` for the class `Loc` and for the class `City`. Here we can reuse the data for the three locations defined for the previous test.

3. Add tests for the method `listSize` for the classes that implement the `ILoCity` interface.

4. Add tests for the method `concat` for the classes that implement the `ILoCity` interface. Here I would define two new routes, the first chunk of the route from Alabama to Maine, and the second chunk, then check that the `concat` method produces the whole route from the two chunks. Make sure you add tests where either the object that invokes the method or the argument to the method is the empty list.

5. Add tests for the method `advance` for the classes that implement the `ILoCity` interface. Again, start by making three routes, the original (I used one of the chunks from the previous test), then how it looks with the first leg removed, then how it looks with one more leg removed. Again, add a test that shows that nothing happens when we try to get the next leg of an empty list.

6. Finally, add tests for the method `onKeyEvent` in the class `GraphWorld`. At this point you may want to change the settings for running the tests to `Tester.runReport(e, false, false);` so that only the failed tests are reported. Again, we need sample data. Make new `GraphWorld`s using the routes you have defined for the previous test.

There are a few things you may have noticed.

First, we have tested the entire behavior of an interactive program without any additional *scaffolding*. The only part that was not properly tested was the display, but even there, the ability to show the drawing on a standalone `Canvas` allowed us to check that the display contains all desired items and that they are in the desired locations.

Next, we would like you to reflect how much did you learn about the design of this program and its behavior by writing the tests.

Finally, notice that because every method produces new values, we could create quite complex tests without excessive set-up and tear-down steps and we could reuse the data we have defined in several tests, knowing that once defined, the data will not change.

2.2 Special Test Scenaria

Managing Complexity: Using and Testing Helper Methods

We would like to give the user better information about the route. Rather than just writing `From AL to AK` our text would give the distance to travel and the direction of the travel, for example `From AL to AK travel 451 miles traveling NW`.

Design the method `fromHereToThere` that produces a `String` that describes the route from this city to the given one. Replace the text in the last line of the method `drawLineToCity` with the invocation of this method.

We get you started. Here is what the method would look like if we did not make any changes in the text that is displayed:

```
/**
 * Produce a <code>String</code> that describes the route from
 * this city to the given one
 * @param that the given city
 * @return a <code>String</code> that describes the route
 */
public String fromHereToThere(City that){
  return "From " + this.state + " to " + that.state;
}
```

We hope that this example forces you to use helper methods, and forces you to test the method that determines the direction very carefully. If you get hopelessly stuck, ask for help, we may give you the solution (for the method that determines the direction) and ask you to just design the tests for it.

Testing for Exceptions

Our program should not go to the next leg of the path, if we have finished our journey and the path is empty.

Start by adding the method `isEmpty()` to the classes that implement the `ILoCity` interface.

Now modify the method `advance` so that if it is invoked on an empty list it throws a new `NoSuchElementException` with an appropriate message. (You will need to add `import java.util.*;` to your imports.)

Well, now our program fails. The first culprit is the test

```
t.checkExpect(this.mtlocity.advance(), this.mtlocity);
```

We need to replace it with a test that verifies that the invocation of the method `advance` by the empty list indeed throws the expected exception and produces the expected message.

Replace this test case with the following:

```
t.checkException(
       new NoSuchElementException(
       "no next item in an empty list"),
       this.mtlocity, "advance");
```

The first argument is the expected exception instance (with the expected message it should provide), next is the object that invokes the method that throws the exception, next is the name of the method. If the method invocation consumes arguments, they are listed afterwards, separated by commas.

Modify the test case by changing the message, the class of the exception, the object that invokes the method, and the method name. Observe the messages the `tester` library provides.

Now fix the code in the `GraphWorld` class, in the `onKeyEvent` method, so that it checks for empty path before it tries to advance.

Testing Value Within the Given Range

We now want to make sure that the city data is correct. We would like to verify that the given latitude and longitude are indeed somewhere within the bounds of 20 to 50 for the latitude and 65 to 125 for the longitude.

Our first step is to design a method that will adjust the given values so that they would fall within the given bounds. (This is not a very realistic example, and we'll rectify it soon, but I am sure you can think of other examples where the outcome of a given method may not be known exactly, but we expect it to fall within the given range.)

1. Design the method `adjustData` for the `Loc` class that consumes an integer value and two bounds (lower and upper) and produces a value within the given range as follows: if the given value is too low, it produces the lower bound, if the given value is too high, it produces the upper bound, otherwise it just returns the given value.

2. Add new tests for your method that only verify that the resulting value is within the given range. *(Of course, you had the tests for out-of-bounds at each end, the boundary cases, and a case with the value in the middle.)* Use the `checkRange` method in the *tester* library.

Note: This is a helper method that does not have the word `this` in its purpose statement. This means that the method does not use any information provided by the instance of this class. In full-scale Java this would be a `static` method, but we do not burden students with these details early on. We do mention that there is something different about this method.

We will not use this method - instead, we will only worry about the constructors for the `Loc` class.

Testing the Constructors

Now we change the constructor for the `Loc` class, and instead of adjusting the incorrect values, it will throw an exception, indicating which of the values was incorrect.

We can do this directly in the constructor, or we can again use a helper method `checkCoordinates` that just throws a `RuntimeException` if the given coordinates are not valid.

Look up the *tester* method `checkConstructorException`.

Testing One-Of Values

If you still have some time (and energy) left, look at how we can test whether the expected value is one of several choices. We return to the *Shapes* project from Lab 1.

1. Design the method that produces a new `CartPt` moved a random distance from the current location, no further than 1 pixel in any direction. So, the new `CartPt` will have the x coordinate equal to one of $x - 1$, x, $x + 1$, and the y coordinate one of $y - 1$, y, $y + 1$.

   The following should help you — expand it to check that the newly moved `CartPt` is indeed built correctly.

```
import java.util.*;
...
  Random rand = new Random();

  // test that we produce one of possible three numbers
  void testOneOf(Tester t){
    t.checkOneOf(1 - this.rand.nextInt(3), -1, 0, 1);
    t.checkOneOf(1 - this.rand.nextInt(3), -1, 0, 1);
    t.checkOneOf(1 - this.rand.nextInt(3), -1, 0, 1);
    t.checkOneOf(1 - this.rand.nextInt(3), -1, 0, 1);
    t.checkOneOf(1 - this.rand.nextInt(3), -1, 0, 1);
    t.checkOneOf(1 - this.rand.nextInt(3), -1, 0, 1);
  }
```