CITS5501 Software Testing and Quality Assurance
Semester 1, 2020
Week 5 Workshop – fixtures, Input Space Partitioning

Reading

It is strongly suggested you complete the recommended readings for weeks 1-4 *before* attempting this lab/workshop.

A simple class

Consider the following code we wish to test:

```
1  class MyClass {
2     private int x;
3     public MyClass(int x) { this.x = x; }
4
5     @Override
6     public boolean equals(Object obj) {
7        if (!(obj instanceof MyClass)) return false;
8        return ((MyClass) obj).x == this.x;
9     }
10 }
```

Find the Java library documentation for the `equals()` method (it's in the `Object` class), and read what its requirements are.

In Java, all other classes automatically inherit from the `Object` class, and may also *override* methods provided by the `Object` class – this is what the "`@Override`" annotation on the `equals()` method means.

The `equals()` method should test whether the `Object` "`obj`" is "equal to" the receiver object, `this`, where what "equal to" means is decided on by the implementer of the class. (The `equals()` method in Java serves the same purpose as the `__eq__` special method in Python.) The implementer is free to decide for themselves what "equal to" means for their class.

If our class is intended to just represent an `int` (i.e., like the `Integer` class), then the `equals()` method we have used is probably fine.

But consider a class `Fraction`, with instance variables `int numerator` and `int denominator`. Our comparison would be more complex, because we probably want that `Fraction` objects representing $\frac{3}{4}$ and $\frac{6}{8}$ should compare as equal.

In order to avoid suprising behaviour for callers of the method, in general equality should be an *equivalence relation*; for instance, an object should always be equal to itself, and if `a.equals(b)` is true and `b.equals(c)` is true, then `a.equals(c)` should also be true.

The `instanceof` keyword in Java allows us to check whether an object is an instace of some class (or some class that inherits from that class, directly or indirectly). Normally, implementers of `equals` will want to return "`false`" whenever we try to compare with objects not of the same class.

Create a new Java project, create a `MyClass.java` file containing the code above, and check that it compiles.

A test class

Suppose we use the following test code for our `MyClass` class:

```
1  import static org.junit.jupiter.api.Assertions.*;
2  import org.junit.jupiter.api.AfterEach;
3  import org.junit.jupiter.api.BeforeEach;
4  import org.junit.jupiter.api.Test;
5
6  public class MyClassTest {
7     private MyClass mc1;
8     private MyClass mc2;
9     private MyClass mc3;
10
11    @BeforeEach
12    public void setUp() {
13      mc1 = new MyClass(3);
14      mc2 = new MyClass(5);
15      mc3 = new MyClass(3);
16    }
17
18    @Test
19    /* Test the case when, for two objects, the second is null */
20    public void equalsWhenNullRef() { fail("incomplete"); }
21
22    @Test
23    /* Test the case when, for two objects, they are not equal */
24    public void equalsWhenNotEq() {/*...*/}
25
26    @Test
27    /* Test the case when, for two objects, they are equall */
28    public void equalsWhenEq() {/*...*/}
```

```
29
30 }
```

In this case, the instance variables `mc1`, `mc2` and `mc3` are potential *fixtures* for any test.

1. Given the test code above, how many times will the `setUp()` method execute?

   Compile and run the tests and check whether this is the case.

2. It is good practice, when writing new tests, to ensure that at first they *fail*. This is useful as a warning, so that you know the test is not yet complete. (We don't want to accidentally give our code to other developers when it contains tests that are incomplete, or do the wrong thing. Potentially even better is to adopt the *test-driven development* methodology, where we write tests *before* implementing our code – we will discuss this in lectures.)

   Insert code into the test methods that will always fail. What JUnit method have you used? Are there any other ways you can think of (or spot in the JUnit documentation) for writing a test that always fails?

3. Fill in code for the test methods in this class.

4. Are there any other tests you think we should add in order to thoroughly test our class? What are they?

5. Using the material from lectures, and the JUnit user guide, write a "teardown" method. What code should go in it? Is it necessary in this case? Why or why not?

Sample solution

1. Fixtures: three times.
2. Failing: We could use `assertTrue(false, "incomplete")` to make tests always fail. But JUnit also has a method specifically for this purpose called `fail()`, that always fails.
   So we can also use 'fail("incomplete").
4. Other tests: This is a pretty simple class. But we could add a test of the constructor, and we could add some data-driven tests if desired.
5. Teardown: We could write a teardown method that just assigns `null` to our instance variables. However, it's not necessary in a simple case like this, because whenever a value gets assigned to an instance variable (e.g. to `mc1`), any previous contents of the variable will get garbage collected.

Input Space Partitioning

Consider the Javadoc documentation and signature for the following Java method, which searches inside an array of `char`s for a particular value.

(Adapted from the Android version of the Java standard library.)

```
1 /**
```

```
 * Performs a binary search for {@code value} in the ascending
 * sorted array {@code array}, in the range specified by fromIndex
 * (inclusive) and toIndex (exclusive). Searching in an unsorted
 * array has an undefined result. It's also undefined which element
 * is found if there are multiple occurrences of the same element.
 *
 * @param array the sorted array to search.
 * @param startIndex the inclusive start index.
 * @param endIndex the exclusive start index.
 * @param value the element to find.
 * @return the non-negative index of the element, or a negative index which
 *         is {@code -index - 1} where the element would be inserted.
 * @throws IllegalArgumentException if {@code startIndex > endIndex}
 * @throws ArrayIndexOutOfBoundsException if
 *         {@code startIndex < 0 || endIndex > array.length}
 * @since 1.6
 */
public static int binarySearch(char[] array, int startIndex, int endIndex,
    ↪ char value)
```

What are some preconditions of this method? What happens if the caller doesn't meet the preconditions? Is an exception thrown? Why or why not?

Discuss in pairs or small groups how you would go about creating tests using Input Space Partitioning. What steps are involved? What is the input domain? And what characteristics and partitions would you use?

List three different tests derived using this method.

Discuss in pairs or small groups how you would assess whether a set of tests have *base choice* coverage. What would you use for base choices?

Sample solution

Preconditions: An example precondition is that the array must be sorted. No exception is thrown if the array is not sorted – instead, the behaviour just won't be what the caller wants.

Why don't we check whether the array is sorted, and throw an `IllegalArgument` exception of some sort? Because that would be a silly amount of work to go to – to check whether an array is sorted, in the worst case, we need to examine every element of the array (only to discover potentially that it is already sorted). Instead we just document the requirement, and it's up to the caller to ensure it holds.

Input Space Partitioning (ISP) – see the lecture notes for examples.

Some possible characteristics:

- Is the array null or not null? (We probably will not write tests for the null case – but it's worth bearing in mind and potentially documenting.)
- Is the array empty, or not? (true or false)
- Does the array contain: 0 elements, 1 element, more than 1 element? (If using this, we don't need the "is empty" characteristic)
- Are the characters in the array printable (like `a`), or non printable (like `\0`)? (Similarly for `value`).
- Is the startIndex negative, zero, or positive? (Similarly for endIndex).
- Characteristics involving multipe parameters:
  - Is startIndex less than endIndex, equal to, or greater than?
  - If not equal to – is the difference between them 1, 2 or more than 2?
  - Is endIndex less than, equal to, or greater than the length of the array?

Base Choice Coverage: First, we pick some "base choice" – usually a "typical value" – for each characteristic.

Then we go through our characteristics and systematically try all the *other* (i.e., non-typical) values for that characteristic. If we have tests that implement all these (where feasible), then we have base choice coverage.

Here is the body of the method.

```
1  public static int binarySearch(char[] array, int startIndex, int endIndex,
       ↪ char value) {
2      if (startIndex > endIndex) {
3          throw new IllegalArgumentException();
4      }
5      if (startIndex < 0 || endIndex > array.length) {
6          throw new ArrayIndexOutOfBoundsException();
7      }
8
9      int lo = startIndex;
10     int hi = endIndex - 1;
11     while (lo <= hi) {
12         int mid = (lo + hi) / 2;
13         char midVal = array[mid];
14         if (midVal < value) {
15             lo = mid + 1;
16         } else if (midVal > value) {
17             hi = mid - 1;
18         } else {
19             return mid;  // value found
20         }
21     }
22     return lo * -1;  // value not present
23 }
```

When we derive tests (as we did in the previous section) just by looking at the requirements for a method, what is that traditionally called?

Solution

Testing based on requirements: It's traditionally called *white box* testing.

Looking at the method implementation, does this suggest any tests you might not have thought of in the previous section?

Self-study task: Write JUnit tests for the method, implementing some of the tests you have identified.

Solution

This will depend on what tests you identify – feel free to show your tests in labs and ask if they are an appropriate implementation.