CITS5501 Software Testing and Quality Assurance
Semester 1, 2022
Workshop 4 (week 5) – graphs – solutions

1. Graphs and control flow

Consider the following Java method for collapsing sequences of blanks, taken from the `StringUtils` class of Apache Velocity (http://velocity.apache.org/), version 1.3.1:

```
1  /**
2  * Remove/collapse multiple spaces.
3  *
4  * @param String string to remove multiple spaces from.
5  * @return String
6  */
7
8  public static String collapseSpaces(String argStr) {
9    char last = argStr.charAt(0);
10   StringBuffer argBuf = new StringBuffer();
11
12   for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++) {
13     char ch = argStr.charAt(cIdx);
14     if (ch != ' ' || last != ' ') {
15       argBuf.append(ch);
16       last = ch;
17     }
18   }
19   return argBuf.toString();
20 }
```

a. Using the ISP principles we have covered in class, suggest some *characteristics* we could use to partition the `argStr` parameter.

   Try generating some test values from your characteristics.

b. Using the techniques outlined in the last lecture, try to construct a *control flow graph* of the method.

   How many nodes do you end up with?
   How many edges?

You may wish to work with a partner for these exercises, and compare your answers.

Note that when constructing a control flow graph, you may ignore calls to other methods, such as `.charAt()`, for the purposes of this exercise – you only need model the control flow *within* the method.

A typical way of "labelling" your graph nodes needs is to use letters ("A", "B", "C" and so on), and to provide a legend, showing a reader which nodes correspond to which lines (or fragments of lines) of code.

Sometimes there may be multiple nodes representing fragments of code all within the same line (e.g. line 11). As long as you have a clear explanation of what each node represents (e.g. "node D: the line 11 loop condition") then that's fine.

Sample solutions:

a. ISP characteristics

Here are some possible characteristics, and values we might select from each partition:

- Is the string empty, or non-empty?
  - This gives us two partitions.
  - For the "non-empty" option, we might choose as test values a "typical" string, say `"too many assessments :/"`, and maybe some less typical options (perhaps very long strings, strings using non-English Unicode characters, or the string `"not nearly enough assessments :/"`)

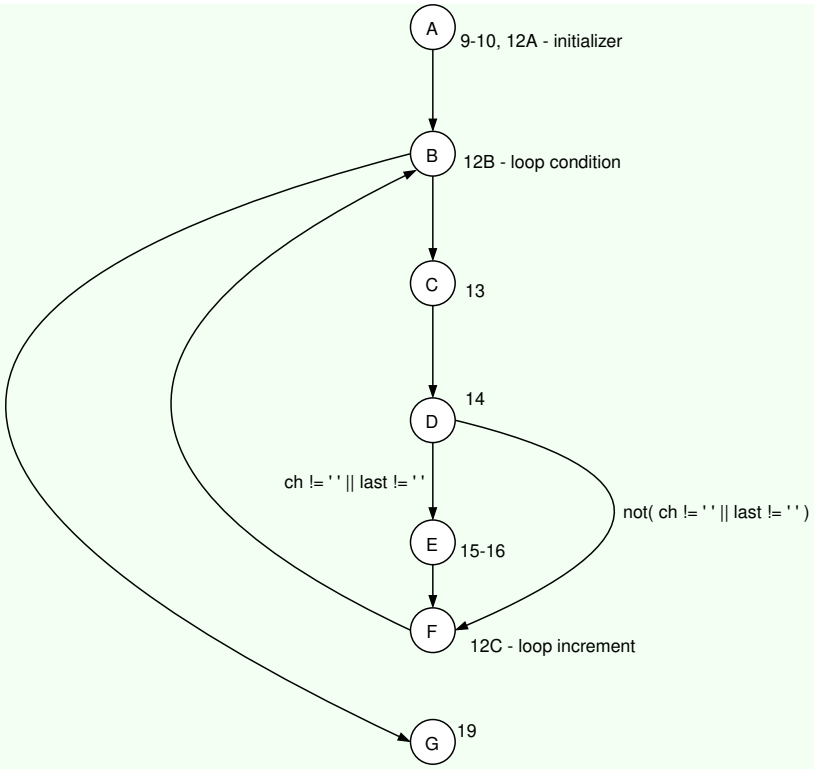
We'll make some *sub*-characteristics for the non-empty option. That is, the following characteristics all assume the string is non-empty:

- Does the string contain spaces? (Gives 2 partitions; strings that do, and strings that don't)
- Does the string contain only spaces? (ditto – gives 2 partitions)
- Does the string contain spaces at the start? (gives 2 partitions)
- Does the string contain spaces at the end? (gives 2 partitions)
- Does the string contain a run of two or more contiguous spaces? (gives 2 partitions)
- Does the string contain a run of two or more tab (`\t`) characters? (gives 2 partitions)
- Does the string contain a run of two or more zero-width space (unicode `U+200B`) characters? (gives 2 partitions)

Sample solutions:

b. control flow graph

Here is one possible control flow graph:



Here, the nodes are labelled with the section of code they represent. Contiguous lines of code (e.g. lines 9–10 and the start of 12) are "collapsed" together to save space – since they must *always* be executed together (in our simple model of the function), there's no real point in giving each line its own node.

c. Given your test cases from part (a), try mentally or on paper "executing" several tests, and see what paths of the graph get exercised by each of your tests.

   How would you subjectively rate the "coverage" of the graph by your tests – good? reasonable? poor?

Sample solutions:

c. subjective graph coverage

The answers here will depend on your suggested tests.

d. Work out whether your tests give the following sorts of coverage:

   i. node coverage
   ii. edge coverage

Sample solutions:

d. node and edge coverage

The answers here will depend on your suggested tests.

e. What are the *prime paths* in your graph? What proportion of the prime paths are exercised by the tests you've given?

Can you construct some tests which exercise prime paths you haven't already covered?

Sample solutions:

e. prime paths

For the graph solution shown earlier, the prime paths are:

ABG, ABCDEF, ABCDF,
BCDEFB, CDEFBC, DEFBCD, EFBCDEF, FBCDEF *(these all execute the left-hand 'if' branch)*
BCDFB, CDFBC, DFBCD *(these all execute the right-hand 'if' branch)*
EFBCDF, *(this takes one left and one right branch)*
CDEFBG, *(takes left branch)*
CDFBG *(take rights branch)*

What proportion of the prime paths your tests cover will depend on what tests you chose. But note that if your tests don't have node or statement coverage, then they certainly won't have prime path coverage.

One useful path is the path ABG, which will get exercised when we pass in the empty. This is a useful test because it reveals a problem with the code – passing in empty strings causes an exception to be thrown when we reach line 9 (the `.charAt` call fails).

We might arrive at this test *either* by applying ISP techniques, or by looking to see what sort of graph coverage we have – as long as we find the bug, either approach is fine!

2. Self-study – test fixtures

It is suggested that you complete the following exercises in your own time to consolidate your understanding of test automation.

Review the material from the textbook on test automation (ch 6), and the JUnit 4 "Text fixtures" documentation (at https://github.com/junit-team/junit4/wiki/Test-fixtures).

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.

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.)

   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?

1. setUp method

As suggested in the question – you should compile and run the tests to find out how many times the `setUp()` method executes.

2. failing tests

Some possibilities are:

- inserting an assertion you know will fail (e.g. `assertTrue(false)`)
- throwing an exception (if your test is *intended* to throw an exception, then throwing an exception other than the expected one). For example:

```
1  throw new RuntimeException("test not implemented yet");
```

- calling the method `fail(String message)` from the `org.junit.jupiter.api.Assertions` class (or one of several similar overloaded `fail` methods). For instance:

```
fail("test not implemented yet");
```

Of these, the last is the best, as it most clearly demonstrates the intention – to fail, not because some assertion is false, but because a test is not complete or has not yet been written.

(You can also check out the answers to this StackOverflow question for some other possibilities.)

Some testing frameworks have special assertions or annotations for marking a test as *pending* (not yet running, for some reason), but JUnit does not yet have this functionality built into it.

3–4 unit test practice

You should gain practise writing tests by doing these exercises yourself – model solutions are not provided. Feel free to show your code to facilitators or the unit coordinator for feedback if you have attempted them.

5. tearDown method

You could write something like

```
@AfterEach
public void tearDown() {
  mc1 = null;
  mc2 = null;
  mc3 = null;
}
```

But this is not actually necessary. The order of events is that for each test, the JVM will

- create an instance of `MyClassTest`
- run the `setUp` method, and
- execute the test

and sometime after this, the instance of `MyClassTest` will get garbage-collected and any memory associated with it will be freed.

We only need to write a `tearDown` method when there are resources (e.g. files on disk, database tables) that are *not* cleaned up by the JVM. In that case, we would write a `tearDown` method that, for instance, deletes any created files.