CITS5501 Software Testing and Quality Assurance
Semester 1, 2020
Workshop 1 – Testing concepts

1. A database has a table for students, a table for units being offered, and a table for enrolments. When a unit is removed as an offering, all enrolments relating to that unit must also be removed. The code for doing a unit removal currently looks like this:

```
1 /** Remove a unit from the system
2  */
3 void removeUnit(String unitCode) {
4   units.removeRecord(unitCode);
5 }
```

Discuss and come up with an answer to the following questions:

a. What preconditions do you think there should be for calling removeUnit()?
b. What postconditions should hold after it is called?
c. Does the scenario give rise to any system invariants?
d. Can you identify any problems with the code? Describe what defects, failures and erroneous states might exist as a consequence.

Sample solutions:

a. Preconditions

Preconditions for `removeUnit` to complete properly, and bring about the postconditions, might include:

- `unitCode` is not `null`
- `unitCode` represents a valid, existing unit code
- The receiver object for `removeUnit()` (i.e., the object reference it is being called on) is not `null`
- A valid database and database connection exist

However, note that we would not necessarily mention all of these in the Javadoc comment for `removeUnit()`:

- Although it may be a precondition that `unitCode` is not `null`, it is true for nearly all Java methods that their arguments must not be null; we are more likely to document the *opposite* case, where a `null` value *is* allowed.
- It is likely that "A valid database and database connection exist" are preconditions for many of the methods in the class we are considering. So we probably would mention this in the Javadoc comment for the class as a whole, to save repeating ourselves.
  (For example, take a look at the documentation for Java's `java.util.TreeMap` ↪ class. It says "This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations", rather than repeating that statement four times.)
- It is a property of the Java language that a method can only be successfully be called when the receiver object is not `null`, else a `NullPointerException` will be thrown. So this need not be mentioned.

Other preconditions we need not document:

- That `unitCode` is of type `String`. If `unitCode` were not of type `String`, the source code couldn't possibly have compiled, and we couldn't be running the program.
  (Or: if, somehow, `unitCode` referred to a spot in memory that did *not* contain a String object, this would be an indication that the Java Runtime Environment had somehow become corrupted.)
  It is a guarantee of the Java language that parameters always have the correct types.

  (In Python, the situation is different. We might require that `unitCode` supports particular string operations, since at runtime, the parameter passed need not be of type `str`.)

Alternative solutions:

- If we adopt the solution above, then that means that if `unitCode` does not refer to a valid, existing unit code, either this method or one of the methods we call should throw an exception. (We would document *that* in our Javadoc as well, so callers know what exceptions can be thrown.)
  But an alternative design is to simply do nothing when the unit code does not exist. In that case, we should *not* throw an exception.

⚠ If you are not clear on what preconditions are from the lecture slides, you might want to read chapter 3 of *Object-Oriented Design and Patterns* (2nd edn) by Cay S. Horstmann. (Password-protected; I'll provide logon details next lecture.)

b. Postconditions

The postcondition here could be stated as:

- "The unit with code `unitCode` does not exist in the database, and no records in the enrolment table exist that refer to it."

c. System invariants

Recall that invariants are assertions that should hold true before and after every method call of a class (or, if we are describing invariants for a whole subsystem or system, for all methods of classes in the subsystem or system).

From what we are told, we can infer that there is at least the following invariant (presumably, for the whole system):

- If an entry in the enrolments table refers to a unit code, then a corresponding entry in the units table must exist.

There might be others as well.

c. Problems with the code

We are not told what `units.removeRecord()` does, exactly.

However, if we make the following assumption:

- `removeRecord()` removes an entry from the units table, and does not alter any other tables

then there *is* a problem with the code: it should have updated the enrolments table, to remove any references to the deleted unit.

So a *defect* in the code is that it does not call whatever methods are necessary to delete records from the enrolments table. (This is a static property of the code.)

The system enters an *erroneous state* immediately after the `removeUnit` method call, because now one of the system invariants is not satisfied – the system is inconsistent.

As for *failures*: recall that these are ways in which the system observably departs from its specification. It is likely that no failure will occur *directly* after the `removeUnit` method call. But the next time someone queries the system to see what units a student was enrolled in: they may be recorded as being enrolled in a non-existent unit, which likely *is* a failure.

⚠ If you are not clear about the difference between *faults*, *failures*, and *erroneous states*, you might want to read chapter 11 of Bruegge and Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java* (3rd edn), particularly section 11.3, "Testing Concepts".

Alternative solution:

- On the other hand, if we make the assumption that `removeRecord` *does* correctly remove all enrolment records which mention the deleted unit, then there is no fault. (This is called "cascading a deletion", in database terminology.)
  But we have to make one assumption or the other, and say which one we are making and why.

2. Discuss whether the following requirements for a system are (a) precise and (b) testable. (It is hard to know if they are consistent and complete in isolation, so we won't consider that.) If they are not, how might you make them precise and testable?

a. The flight booking system should be easy for travel agents to use.
b. The `int String.indexOf(char ch)` method should return a -1 if `ch` does not appear in the receiver string, or the index at which it appears, if it does.
c. Internet-aware Toast-O-Matic toasters should have a mean time between failure of 6 months.

Sample solutions:

a. easy to use

This is not at all precise – "easy to use" is a vague description, and difficult to quantify.

A better requirement might be:

> "Travel agents shall be able to use all the system functions after successful completion of a training course designed by the software provider. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use."

(This is adapted from *Pressman*.)

The resulting requirement is certainly testable, but would probably not be fully tested until the *acceptance testing* phase. Prior to that, the software provider might try to come up with a quicker and cheaper test to act as a *proxy* for the acceptance test – they might test it on non-technical staff in their own organisation, for instance.

⚠ If you are not clear about what makes a good requirement, you might want to review the chapter from the *Pressman* textbook on "Understanding Requirements" (in the 9th edition) or "Requirements Engineering" (in earlier editions).
There is also a quick summary (taken from documentation for an IBM requirements management product) available here.

b. indexOf method

This doesn't specify what happens if the character appears in the string multiple times – it says "the index at which it appears", implying there is only one. It would be better to specify "the *first* index at which it appears".

Once that is done, the method could still be made *more* precise – compare the actual Javadoc for the Java `String.indexOf` method. That documentation clarifies that `ch` represents a Unicode code point, and explains what happens when `ch` falls in various ranges.

Once corrected, the requirement is straightforward to write tests for – we have seen examples using JUnit.

b. mean time between failure

For many purposes, this is probably precise enough (though one might want to add "under normal operating conditions").

As it stands, it is not easy to test, however, until after the toasters have been sold and are in normal operational use.

Again, the provider might make use of some sort of proxy test to assess the resilience of toasters. (Consider testing of car safety, for instance: do manufacturers "test" the safety of cars by simply selling them, and seeing what accidents occur? No – they do things like simulating wear and tear, and the effects of collisions.)

3. Sketch out code for a JUnit test for the `String.indexOf()` method. What does it

need to contain? What are some tests you might include? Use pseudocode if you cannot recall the Java syntax.

Sample solution:

Here is code for *one* test we might run:

```
1  import static org.junit.Assert.assertEquals;
2  import org.junit.Test;
3
4  public class StringTest {
5    @Test
6    public void returnsNeg1OnEmpty() {
7      int result = "".indexOf('a');
8      assertEquals(-1, result);
9    }
10 }
```

It tests that when we look for the position at which the letter "a" appears in the empty string, we get a result of -1.

Some other tests we might try are:

- The result returned when the string has non-zero length, and contains (a) zero or (b) one or (c) two or more instances of the character being looked for.
- The result returned when the string being searched has length one, and contains (a) zero or (b) one instance of the character being looked for.
- We might try a word or character from a non-English alphabet.

⚠ The unit CITS1001 Software Engineering with Java covers an informal approach to coming up with test cases – we will look at more rigorous approaches in coming weeks. If you have not completed CITS1001 (or want to refresh your memory on the topic) you might want to read Chapter 7 ("Well-behaved Objects") of the CITS1001 textbook, *Objects First with Java: A Practical Introduction Using BlueJ*.