SENG275 – Lab 3
Due Monday Feb 7, 5:00 pm

Technical note: You now have a third repository on gitlab, named lab03

Welcome to Lab 3. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm.

Quick summary of what you need to do:

1. Clone your lab03 repository from gitlab.
2. Write tests for the three Boundary Testing exercises in BoundaryTest.java
3. Write tests for the two Specification Testing exercises in SpecificationTest.java. Do this *before* inspecting the code supplied in the main/java/resources folder of the repository.

   Note: DO NOT write your own implementation for these two methods! The idea is that write your tests against the *specification* (the comments) without having access to the code. AN IMPLEMENTATION HAS BEEN WRITTEN FOR YOU in main/java/resources.

4. The supplied code for messageIsValid() has several omissions. Identify one such error and write a bug report following the template in the test/java/resources folder.
5. Commit your changes to your lab03 repository, and push the commits back to gitlab.

   You're done!

Domain Testing

The domain of a function is the set of valid inputs that can be supplied to that function. Many functions have an enormous domain – the set of all integers or floating point numbers, for example. Domain testing is a process by which we select and test particular values from the domain of possible inputs, and thus keep the number of tests we need to write manageable. Our goal is therefore to balance *effectiveness* with *efficiency* – we want to avoid writing redundant tests, while still testing the input values critical to the proper functioning of the code under test.

Bugs in the boundaries of the input domain are pretty common in software systems. We have all made mistakes with using a > operator when it should have been a >= operator! To catch these types of errors, we use a process called Boundary Testing. We refer to the requirements that specify the conditionals (that we are also likely to see in our source code as well). We inspect each of the conditionals that will process our inputs. We identify the ON and OFF points for each conditional, as well as the In and Out points that the conditional implies. We then divide our domain of inputs into equivalence classes, and implement at least one test for each equivalence class.

If we don't have access to the source code, we have to perform Specification Testing - we must rely on the documentation for the unit under test (we can't peak at the code!). Again we try to analyze the domain of inputs, separating them into equivalence classes. Again we write at least one test per equivalence class.

Worked example

Worksafe BC defines ‘unsafe sound exposure’ as long-term exposure to noise of greater than 85 decibals in volume. The specification for a method implementing this logic might look like this (from Boundary.java). We also show the source code here (but we could just as easily infer this from the requirements that there is a condition that uses the > operator).

```
/**
 * Returns true if the sound level in decibels is unsafe - that is, higher than 85 decibels.
 *
 * The WCB defines unsafe sound exposure as long-term exposure to sound in excess
 * of 85 decibels in volume.
 *
 * @param volume the volume of the sound in decibels
 * @return       true if the volume is unsafe, false otherwise
 */
public boolean isUnsafe(int volume) {return volume > 85;}
```

Using our assumption of how this is implemented, first, we identify the ON and OFF points.

The ON point of a conditional is the value that appears in the conditional expression. In this case our conditional is volume > 85, so the ON point is 85. We then look for the OFF point.

The OFF point of a conditional is the value that is closest to the ON point, but which flips the output of the conditional. In our case, at the ON point, our conditional evaluates to false. So the OFF point is the closest value to 85 in which the conditional evaluates to true. At 86, the conditional is true, so 86 is our OFF point.

Note that there is no automatic relationship between true and false and on and off points – they depend on the particular way the conditional has been written. In particular, using >= rather than > will change the ON and OFF points. Also notice that ON and OFF points are specific to individual conditionals. In our “guess” of how the code is implemented, we make some assumptions to help guide our tests! (Optional: if we had assumed a different implementation for the condition, e.g., one that used >= 86, would this change your test cases?)

Next we determine IN and OUT points. The IN points are all the values that result in the expression evaluating to true, while the OUT points are those that evaluate to false. Our IN points are therefore [86, Inf), while our OUT points are (-Inf, 85].

Every continuous range of IN points constitues an equivalence class. Note that we could write tests for 86 decibals, 102 decibals, 250 decibals and so on – each would be treated by the conditional the same, and so each are *equivalent*. If two tests use values in the same equivalence class, one can generally be discarded without reducing the effectiveness of the class.

Finally, note that we can eliminate tests that explore impossible or nonsensical parts of these ranges. For example, there is no such thing as a negative decibal rating, so we can truncate the OUT range to [0, 85].

Now we select a value from each IN range and each OUT range to test. The best values are those that are also either ON or OFF points, since they lie at the boundary where the behaviour of the code changes. Our best values to test are therefore 85 (which is both our ON and OUT point) and 86 (which is both our OFF and IN point):

```
@Test
void isUnsafe() {
   assertTrue(Boundary.isUnsafe(86));
}

@Test
void isNotUnsafe() {
   assertFalse(Boundary.isUnsafe(85));
}
```

More tests could be written, but they would be unnecessary, we can judge that the program would not behave any differently with other input values.

Multiple input values

Some methods take more than one input value. In these cases, you may have multiple boundary points for every independent variable – two input values might produce a 2-d array of testable values, three input variables a 3-d array, and so on. The number of combinations might soon become intractable. Part of your role as tester will be identifying sets of boundary points that need to be tested as a priority, and others that do not.

Parameterized Tests

Junit helps us write large numbers of tests quickly with @ParameterizedTest.

Section 2.2 of our textbook covers the technique. Let’s say we want to test our isUnsafe method with lots of unsafe volumes. We start by writing our test normally, and by including an argument list that covers the arguments we want to pass into the individual tests. Then we need to provide a source for our values of *volume*. Our method takes ints, so we can pass them in from a @ValueSource:

```
@ParameterizedTest
@ValueSource(ints = {90, 120, 128})
void volumesUnsafe(int volume) {
   assertTrue(Boundary.isUnsafe(volume));
}
```

Behind the scenes, Junit actually creates three separate tests for us, one for each value. It's important to recognize that this is not breaking our One-Action-Per-Test rule – we're calling isUnsafe() three times, but doing so in three tests. You can think of @ParameterizedTest as a macro for automatically building tests for us.

@ValueSource only works with one argument per test, though. If we want more than that, we need either @CSVSource or @MethodSource.

For example, if there were different unsafe sound limits in each province, we might have to pass a province abbreviation into our method. If our method signature was:

```
public static boolean isUnsafe(int volume, String provinceCode)
```

then we could handle this with @CSVSource:

```
@ParameterizedTest(name="volume{0}, provinceCode{1}")
@CsvSource({"86, BC", "81, MB", "84, SK", "91, ON"})
void volumesUnsafe(int volume, String provinceCode) {
   // do something with provinceCode
   assertTrue(Boundary.isUnsafe(volume));
}
```

Finally, if we need even more control over the arguments to our test, we could use @MethodSource, which specifies a method that returns a stream of values:

```
@ParameterizedTest(name="volume{0}, provinceCode{1}")
@MethodSource("allTheVolumes")
void volumesUnsafe(int volume, String provinceCode) {
   assertTrue(Boundary.isUnsafe(volume));
}

private static Stream<Arguments> allTheVolumes() {
   return Stream.of(
           Arguments.of(86, "BC"),
           Arguments.of(81, "MB"),
           Arguments.of(84, "SK"),
           Arguments.of(91, "ON"));
}
```

To use these annotations, you'll need to import them. The following imports should get you what you need:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.jupiter.params.provider.ValueSource;
import java.util.stream.Stream;
```

Specification Testing

If we don't have the access to the source code, we're in a Black-Box-Testing situation. We need to rely on the documentation in whatever form it exists – comments, written specifications, whatever we have available to us. Any deviation from the documentation is either an error in the program, or an error in the documentation. Try to establish the boundary points as implied by the documentation and your domain knowledge, and establish equivalence classes. It won’t be as easy to do so without being able to inspect the conditionals and code flow yourself, so you may need many more tests before you achieve same level of confidence as you do for white-box testing.

Your task

1. Write the tests for the three remaining methods in Boundary.java (use BoundaryTest.java to do so). Determine the boundary ON and OFF points and use them to figure out the IN and OUT points, and to establish your equivalence classes. Try to be efficient – avoid testing multiple values in the same equivalence class.
2. Try Specification testing. The methods in the Specification.java file have specifications (comments) but no implementations. Try reasoning about their behaviour and write tests that would check code for adherance to those specifications.
3. Go into main/java/resources, and find the implementation.txt file. This file contains the implementation for these two methods. Paste the code into the Specification file and run your tests. Did they work?
4. The implementation for messageIsValid is buggy and incomplete. Use your tests to identify a bug, and write a bug report in test/java/bugreport.txt. Try to follow the given bug report format, and try to make your report short enough to be easily readable, but detailed enough to be helpful to the developers. Think about what kinds of information the developers might need to solve the bug you’ve found.