Advanced Programming

Lab Sheet 7 – JavaDoc & JUnit

Introduction

This lab focusses on the documentation and testing of some existing code. I have made an eclipse project available to you on Moodle, which contains some classes that form part of the software for a (hypothetical) self-driving car. You will be writing JavaDoc documentation for the provided code, and then writing some unit tests for the code using JUnit.

An autonomous vehicle will frequently estimate its so-called "stopping distance". This is the distance that would be required to bring the vehicle to a complete halt, taking into account the current speed, road/weather conditions, how heavily loaded the vehicle is, and whether the brakes have warmed up and are likely to perform well. If an unexpected obstacle is detected that is within the current estimated stopping distance, the vehicle may need to perform an evasive manoeuvre, otherwise it may instead choose to stop.

The basic calculation for stopping distance is well known, you may have studied it previously in Physics, at school. The formula is: $d = v^2 \,/\, 2\mu g$ where d is the stopping distance (in meters), v is the velocity (in meters per second), μ is the coefficient of friction between the car's tires and the road surface, and g is gravity, with a standard value of 9.806 used on the surface of the Earth.

The BrakingDistanceEstimator class uses estimates of the friction coefficient for road surfaces under various conditions which were adapted from the JavaScript/HTML source of the calculator in [1], and applies the following modifications to the calculated distance:

1. If the brakes are currently cold, an extra 10% is estimated, allowing for reduced braking performance as the brakes come up to temperature.

2. If the driver is currently alone in the vehicle (as detected by seat weight sensors), then a 10% reduction in braking distance is estimated, as the vehicle is lighter than usual.

3. If the driver has both front and rear passengers (as detected by seat weight sensors), then a 20% increase in braking distance is estimated, as the vehicle is heavier than usual.

Importing The Code

The code is available on Moodle as an eclipse project, exported as a zip file. You can import it into your eclipse workspace by selecting File → Import, and then selecting "Existing Projects Into Workspace". You will see a dialog similar to that depicted below.

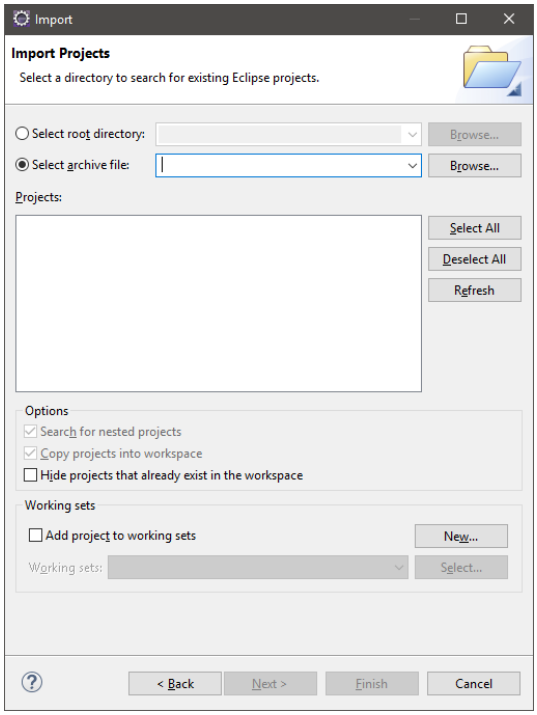

Figure 1: Eclipse Import Projects Dialog

On this dialog, you should make sure "Select archive file" is selected, then browse to the zip you downloaded from Moodle. Make sure that the "CodeQualityLab" project is ticked, before clicking the "Finish" button at the bottom of the dialog. You will see the project in the left-hand panel in eclipse, along with your previous projects.

Task 1: Generating Documentation with JavaDoc

As you saw in the code quality lecture, you can special comments (/** and */) in Java class files, along with the JavaDoc tool (Project → Generate JavaDoc) to create HTML documentation for classes using the standard template, looking similar to the official Java documentation.

There are various JavaDoc annotations (discussed here) that you can use inside JavaDoc comment blocks, with special meaning. Be sure that you're able to use the @return, @param and @throws annotation at least, as these are the ones most commonly used.

Add JavaDoc documentation to each of the classes, properties and methods in the supplied eclipse project, generate the HTML output using the JavaDoc tool, and review the produced web pages in your web browser. The HTML will have been saved in your project's folder, in a subdirectory named doc. This folder isn't normally visible in the package explorer (left-hand panel in eclipse), so you will need to open it via the file system. Open the index.html file to start looking at the output.

Task 2: Writing Unit Tests with JUnit

Add some JUnit test cases to the project, following the procedure demonstrated in the lecture. For some of the test cases, it would be advisable to manually calculate the correct stopping distance and ensure that the correct value is returned. For others, you can write the tests as a simple sanity check. Will it take longer to stop everything else being equal, if the road is wet rather than dry? Because this code would form part of a safety-critical system, you will need to be thorough when devising test cases. Try to think of all the tests necessary for you to be confident the code is working correctly.

Once you have written all of the tests you think are needed, use the "Coverage" tool in eclipse to see whether any of the code in the BrakingDistanceEstimator class wasn't touched by your test cases. If any of the code is red, you didn't test it and need to write some more unit tests!

References

1. Forensic Dynamics, inc. Stopping (Braking) Distance Calculator. Available: http://www.forensicdynamics.com/stopping-braking-distance-calculator