SOFTWARE TESTING COURSEWORK DESCRIPTION

February 2022

SOFTWARE TESTING: PRACTICAL

This page describes the practical for the Informatics Software Testing course. It will be marked out of 100 Points, and is worth 45% of the assessment of the course.

DEADLINE

The coursework comprises 3 tasks with the following issue dates and deadlines:

- Issued: 24th February
- **Deadline : 28th March, 4pm GMT**.

Extension Rule 1 applies to ST coursework.
Rule 1: Extensions are permitted (7 days) and Extra Time Adjustments (ETA) for extensions are permitted.

- **Penalty:** If assessed coursework is submitted late without an approved extension, it will be recorded as late and a penalty of **5% per calendar day will be applied for up to 7 calendar days, after which a mark of zero will be given.**
- For electronic submissions, the last version that has been submitted by the deadline will be the one that is marked (late submission will only be accepted if no submission in time has been made).
- If a student with an approved extension submits late (>7 calendar days from the original deadline) a mark of zero will be given.
- If a student with an approved ETA extension submits late (textgreater7 calendar days from the original deadline) a mark of zero will be given.
- If a student with an approved extension plus ETA(s) extension submits late (>14 or >21 calendar days from the original deadline) a mark of zero will be given.

Coursework will be scrutinised for plagiarism and academic misconduct. Information on academic misconduct and advice for good scholarly conduct is available at `https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`.

TOOLS

You can choose to use the Eclipse IDE, another IDE such as IntelliJ IDEA, or just use JUnit and other appropriate tools of your choice standalone. I have no strong preference – many people find the tools available in Eclipse useful (if you haven't used Eclipse before maybe now is the time to give it a try). You can discuss your choice and problems you come across with the tools (without sharing specifics of your coursework solutions) on Piazza, but note that tool-wrangling is part of what this coursework is intended to help you learn so it is ultimately **your responsibility** to get your chosen tools to work. Don't leave it to the last minute!

The rest of this handout is written mostly as though you were using Eclipse. You will need some of the following:

1. JUnit 4. If necessary you can download JUnit from `https://junit.org/junit4/`. If you are using Eclipse it is probably already installed in the IDE. This article: `https://www.vogella.com/tutorials/JUnit4/article.html` is a reasonable introduction to using JUnit4 with Eclipse.

2. You will need some kind of coverage analysis tool:

   - In Eclipse you can use EclEmma: `https://www.eclemma.org/`. If you use Eclipse on DICE, or if you install the version of Eclipse that is current at the time of writing and select "for Java developers" when asked, this should already be installed, and is accessible as "coverage" from the toolbar or Run menu. If not, it's easy to install through Eclipse's built in software update mechanism. IntelliJ IDEA also has a built-in coverage module.
   - For stand-alone coverage you should consider something like Cobertura: `http://cobertura.github.io/cobertura/`
   - A review of other Open Source code coverage tools for Java is available at `https://java-source.net/open-source/code-coverage`.

Most of the tasks are associated with the tutorials, so that they help you prepare for it. Please prepare in advance for the tutorial to get the most out of the process.

SETTING UP

Preparation

If you don't have Eclipse installed and want to use it, you should download it and install it. You can find Eclipse at `https://eclipse.org/users/`. You want "for Java developers" when asked. This will come with JUnit and EclEmma pre-installed. Eclipse's help menu leads to a lot of information, some of which is outdated: you may wish also to consult the links given above.

Once you install your IDE, make sure to include the given `.jar` file containing the bugs you need to find for task 1 in the build. In eclipse, you can do so by performing a right click on the project, then selecting `Build Path > Add External Archives...`.

Once loaded, in order to set it as the one that the tests will run, right click on the project, select `Build Path > Configure Build Path...` and then raise the `.jar` file on top of the elements order. To execute the tests on the original source code, repeat the same process and lower the `.jar` file at the bottom.

Submission

For each one of the subtasks, create a file as described in the respective subtask. Put all the files in the same folder. Submit your deliverables on Learn. You can do so on `Learn > Assessment > Coursework Submission`.

TASKS

TASK 1: FUNCTIONAL TESTING (40 Points)

TASK1.1: Find bugs applying functional testing (40 Points)

In this task you will implement JUnit tests using the specification provided in the Github repository containing the coursework: `https://github.com/SoftwareTestingUoE/SoftwareTesting2021-22/`. The repository also provides the implementation as a `.jar` file, so you can execute your JUnit tests and observe test results. The specification is described in detail, with helpful examples where necessary, in the `Specification.pdf` file.

Functional testing is a black box testing technique, so use the specification file to derive tests and **NOT** the source code. The `.jar` file can be used to execute the tests derived from the specification. We have also provided a sample JUnit test case, `CommandLineParserTest.java` file, to illustrate a typical test case for the implementation in `ST_Coursework2021_22.jar`. All the files referred to above can be found at the coursework Github repository.

For this task, our ST_Coursework2021_22.jar file contains 20 different bugs. Marking follows this pattern:

- Easy bugs (6): 1 point each.

- Medium bugs (8): 2 Points each.
- Hard bugs (6): 3 Points each.

If you are able to find all of the vulnerabilities, you will get full Points (40) from this task.

"Finding" a vulnerability means writing a test which should pass according to the specification, but fails on the provided implementation.

Deliverable:

- A file `Task1_1_FunctionalTest.java` that contains your JUnit tests.

TASK 2: COVERAGE ANALYSIS (25 Points)

TASK2.1: Analyze Code Coverage (5 Points)

The goal of this task is to measure and analyze the branch coverage of the codebase achieved by executing the test cases developed in Task 1. Use a coverage measurement tool to extract these information automatically.

Deliverables:

- A file `Task2_1.jpg` containing a screenshot of the branch coverage report as shown by the coverage measurement tool. Please make sure that the total branch coverage is clearly visible in the screenshot.

TASK2.2: Increase Branch coverage (10 Points)

Increase branch coverage by adding tests to your suite. If you have uncovered branches, explain why the branches remain uncovered with your tests and whether it is possible to cover the branches.

Deliverables:

- A file `Task2_2_FunctionalTest.java` containing your suite aiming to increase branch coverage.
- A file `Task2_2.txt` containing your explanation.

TASK2.3: Minimize Test Suite (10 Points)

Assume you have a limited number of resources and cannot run all the tests created in Tasks 1 and 2.2. Minimise your test suite (combining tests from Tasks 1 and 2.2) based on your criteria and justify your choices of criteria and why you think it is effective for use in minimisation.

Deliverables:

- A file `Task2_3_FunctionalTest.java` containing your minimised suite.
- A *plain text* file `Task2_3.txt` containing your choice of minimization criteria and why you believe it is effective.

TASK 3: TEST-DRIVEN DEVELOPMENT (35 Points)

Task 3.1: Adding Functionality with Test-Driven Development (35 Points)

A TDD approach is typically interpreted as "A programmer taking a TDD approach refuses to write a new function until there is first a **test that fails** because that function is not present." TDD makes the programmer think through requirements or design before they write functional code. Once the test is in place the programmer proceeds to complete the implementation and checks if the test suite now passes. Please keep in mind that your new code may break several existing tests as well as the new one. The code and tests may need to be refactored till the test suite passes and the specification is fully implemented. In this task, you will need to follow the TDD approach to implement and support a new additional specification in the existing implementation. The new additional specification is described in the last two pages of the specification file, available in the coursework Github repository.

Deliverables: This task will involve a 2-part submission.

1. **Part 1**: Tests (15 Points)

   Submit **only your new JUnit tests** in a file named `Task3_TDD1.java`, for the new additional specification (last 2 pages of the specification document). Please check to make sure all of these new tests fail on the existing implementation available in the `src` folder.

2. **Part 2**: Implementation + Tests (20 Points)

   This part requires that you add source code to `Parser.java` in the `src` folder to support the new specification. Check whether all the tests you developed in Part 1 pass for your modified implementation. If they don't, modify the implementation and/or tests so the entire test suite passes and the new specification is implemented correctly, as breaking past suite tests will have an effect on marking. Name your modified collection of new tests `Task3_TDD2.java` (even if it is identical to the file you submitted as `Task3_TDD1.java` in Part 1). Submit both the modified implementation of `Parser.java` (rename it to `Task3_Parser.java`), and the test suite `Task3_TDD2.java`.

Important Note: For Task 3, modify only the file `Parser.java`. Please do not touch other java files.