

THE UNIVERSITY OF MELBOURNE  
SWEN90006: SOFTWARE TESTING AND RELIABILITY

**Assignment 1**

SECOND SEMESTER, 2019

DUE DATE: 18:00PM, FRIDAY, 6 SEPTEMBER, 2019

## Introduction

The first assignment deals with input partitioning, boundary-value analysis, and control-flow testing. You are given a specification and a program that implements that specification. The aim of this assignment is to test the program using the different techniques, and to analyse the difference between them.

You are expected to derive and compare test cases, but you are not expected to debug the program.

The assignment is part laboratory exercise because you are expected to write a JUnit driver program to run your test cases automatically. Some exploration may be needed here. The assignment is also part analysis exercise as you are expected to apply the testing techniques to derive your test cases, and to compare them. Finally, the assignment is also part competition, as your solutions to various tasks will be evaluated against all other submissions to measure its effectiveness and completeness.

The assignment is worth 20% of your final mark.

## Description: PassBook

PassBook is a (fictional) online password manager. A password manager is a software application that generates, stores, and retrieves login details for users.

A user has an account with PassBook. This account is protected by a master password, or passphrase. Each user can store login details for multiple websites, identified by their URL. A user can add a login details for a given website. The passphrase is used to encrypt all passwords, but for this implementation, we have ignored encryption.

An API specification is available in the source file (see below).

## Source code

To obtain the source for the program, fork the repository at:

<https://gitlab.eng.unimelb.edu.au/tmiller/swen90006-a1-2019.git>

You will have an account created on <https://gitlab.eng.unimelb.edu.au/> that uses your University of Melbourne login details.

Documentation for how to fork, pull, push, merge etc. is available from the Gitlab site: <https://gitlab.eng.unimelb.edu.au/help/user/index.md>.

NOTE: If you find any functional faults in the implementation, please let us know via the discussion board. We will correct the fault and ask that everyone pull changes. There are not intended to be faults in the implementation, but software engineering is hard!

## Building and Running the Program

The source code has been successfully built and tested on JDK 1.12 but should also work with some earlier versions of Java.

The file `build.xml` contains an Ant build script, should you choose to use it. The `README.md` file in the top-level folder has instructions for using this.

There are two JUnit test scripts in `test/swen90006/passbook`. You will need to modify each of these to complete the tasks below. You can run these by compiling and running as a class, but you will need to include the library files in the `lib/` directory.

## Tasks and Questions

### Task 0 (0 marks)

Once you have cloned your repository, set your repository to private. Otherwise, every else can see your mutants and your code. This can be set in Settings → General → Permissions → Project Visibility.

Then, Add ‘Administrator (@root)’ as a developer to your repository, otherwise we will not be able to access your code for submission. To do this, go to Settings → Members, under ‘Select members to invite’, search for ‘Administrator’, then under ‘Choose a role permission’ select ‘Developer’, and finally click ‘Add to project’.

### Task 1

Using the specification, apply equivalence partitioning to derive equivalence classes for the following methods in the API: `addUser`, `loginUser`, `updateDetails`, and `retrieveDetails`.

Document your equivalence partitioning process for each method using only test template trees, listing the assumptions that you make (if any). You should have four tree: one for each method. You will be marked *only* on your test template trees, so ensure that they are clear and concise.

You can omit some nodes to improve readability, provided that it is clear what you intend. For example, in tutorial 2, if I wanted to test all 12 months of the year, I would create nodes for JAN and DEC, and then use “...” in between them to represent the other months.

Note that as part of your input domain, you will have to consider the instance variables. These are not parameters to any of the methods, but they are *inputs*.

Do your set of equivalence classes cover the input space? Justify your claim.

## Task 2

Select test cases associated with your equivalence classes, and implement these in the JUnit test driver named `tests/Partitioning/swen90006/passbook/PartitioningTests.java`. Use JUnit test method for each equivalence class. For each test, clearly identify from which equivalence class it has been selected.

NOTE: As you will find, when implementing tests for one method, you may need to use other methods to check that the first method has worked as expected.

## Task 3

Conduct a boundary-value analysis for your equivalence classes. Show your working for this. Select test cases associated with your boundary values.

## Task 4

Implement your boundary-value tests in the JUnit test driver called `test/Boundary/swen90006/passbook/BoundaryTests.java`.

Note that you can extend/inherit from the JUnit script for your partitioning tests, which will include all tests from the parent class. A JUnit test is just a standard public Java class!

## Task 5

Calculate the coverage score of your two test suites (equivalence partitioning and boundary-value analysis) using *multiple-condition coverage* each of the four of the methods. You should have eight coverage scores: partitioning and boundary scores for each of the four methods.

Show your working for this coverage calculation in a table that lists each test objective (that is, each combination for multiple-condition coverage) and one test that achieves this, if any.

You will receive marks for deriving correct coverage scores and showing how you come to this score. *You will not receive any marks for having a higher coverage score.* If you think your tests are good but do not cover some points, there is no need to add new tests to cover these (note the score calculation below for the competition explicitly penalises larger test suites). The task is testing your knowledge and ability to apply coverage concepts, not to improve the test suite.

NOTE: You do NOT need to draw the control-flow graph for your solution.

## Task 6

Derive five *non-equivalent* mutants for the `PassBook` class using the mutation operators in the notes, and that you believe will be difficult to find using testing. Insert each of these mutants into the files `programs/mutant-1/swen90006/passbook/`, `programs/mutant-2/swen90006/passbook/`, etc.

It is important that these mutants are both non-equivalent AND that each mutant is killed by at least one test in your JUnit script to demonstrate that they are non-equivalent.

Importantly, do not change anything else about the mutant files except for inserting the mutant.

## Question 7

Compare the two sets of test cases (equivalence partitioning and boundary-value analysis) and their results. Which method did you find was more effective and why? You should consider the coverage of the valid input/output domain, the coverage achieved, and the mutants it kills. Limit your comparison to half a page. If your comparison is over half a page, you will be marked only on the first half page.

## Marking criteria

As part of our marking, we will run your *boundary-value analysis* JUnit scripts on everyone else's mutants. You will receive marks for killing other mutants as well as for deriving mutants that are hard to kill. This will contribute 5 marks to the total.

Criterion	Description	Marks
Equivalence partitioning	Clear evidence that partitioning the input space to find equivalence classes has been done systematically and correctly. Resulting equivalence classes are disjoint and cover the appropriate input space	7
Boundary-value analysis	Clear evidence that boundary-value analysis has been applied systematically and correctly, and all boundaries, including on/off points, have been identified	3
Control-flow analysis	Clear evidence that measurement of the control-flow criterion has been done systematically and correctly	2
	There is a clear and succinct justification/documentation of which test covers each objective	2
Discussion	Clear demonstration of understanding of the topics used in the assignment, presented in a logical manner	1
JUnit tests	JUnit scripts implement the equivalence partitioning and boundary-value tests, and find many mutants	2.5
Mutants	Selected mutants are valid mutants and are difficult to find using tests	2.5
Total		20

For the JUnit tests, the score for these will be calculated using the following formula:

$$junit\_score = \frac{\frac{k}{T}}{\ln(N) + 10}$$

in which  $N$  is the number of tests in your test suite,  $k$  is the number of mutants that your test suite kills, and  $T$  is the maximum number of mutants killed by any other JUnit test suite<sup>1</sup>. The entire pool of mutants are the mutants from all other submissions. Therefore, your score is

---

<sup>1</sup>This ensures that equivalent mutants are not counted.

the mutant score, divided by  $\ln(N) + 10$ , which incentivises smaller test suites<sup>2</sup>. The maximum possible score is 0.1, scaled to be out of 2.5.

For the mutants, the score is:

$$mutant\_score = \frac{\sum_i^M \sum_j^N a_{i,j}}{T}$$

in which  $M$  is the total number of your mutants,  $N$  is the total number of other people's test suites,  $a_{i,j} = 1$  if mutant  $i$  is still alive after executing test suite  $j$ , and  $T \leq M + N$  is the highest number of mutants still alive by any student in the class. This is then scaled to be out of 2.5. Therefore, your score is the inverse of the mutant score of all other students' test suites on your mutants, which incentivises you to submit hard-to-find mutants, while  $T$  normalises the score to ensure that everyone is rewarded for good mutants.

**Important note:** We determine that a mutant is found when JUnit contains a failed test. Because of this, if a JUnit fails a test when applied to the original source code, it will fail on everyone else's mutants, giving people a 100% score. As such, *JUnit suites that fail on the original source code emulator will be disqualified from the tournament*. As noted above, if you find any faults in the original source code, please let us know via the discussion board.

## Submission

For the JUnit test scripts, we will clone everyone's Gitlab repository at the due time. We will mark the latest version on the *master* branch of the repository. To have any late submissions marked, please email Tim ([tmiller@unimelb.edu.au](mailto:tmiller@unimelb.edu.au)) to let him know to pull changes from your repository.

Some important instructions:

1. Do NOT change the package names in any of the files.
2. Do NOT change the directory structure.
3. In short: you should be able to complete the assignment without adding any new source files.

JUnit scripts will be batch run automatically, so any script that does not follow the instructions will not run and will not be awarded any marks.

The remainder of the assignment (test template tree, boundary-value analysis working, coverage, and discussion) submit a PDF file using the Turnitin links on the subject LMS. Go to the SWEN90006 LMS page, select *Assessment* from the subject menu.

## Setting Out Your Solution

In the git repository, there are three sample solutions from last year's assignment, which you can use for ideas on how to set out your solution.

---

<sup>2</sup>This incentive is to resist the urge to submit a test suite of thousands of tests with the hope of increasing the score.

Note that these three sample solutions are submissions made by class members of SWEN90006. They are examples of what the staff think are good ways to set out answers. This does not mean that they are free from error. In particular, they all have more information than is necessary to get full marks for this. Please resist the temptation to just trying to map your solution one this. The assignment was very different, and moreso, these were NOT the solutions with the highest marks.

## Tips

Some tips to managing the assignment, in particular, the equivalence partitioning:

1. Ensure that you understand the notes *before* diving into the assignment. Trying to learn equivalence partitioning or boundary-value analysis on a system this size is difficult.
2. Keep it simple: don't focus on what you think we want to see — focus on looking for good tests and then documenting them.
3. Focus on the requirements: as with any testing effort, focus your testing on the requirements, NOT on demonstrating the theory from the notes. If you are worrying about whether to apply a particular part of the theory, look at whether it tests one of the requirements. If not, it is probably not useful.
4. If you cannot figure out how to start your test template tree, simple start listing tests that you think are important. Once you have a list, think about putting them into a tree.

## Academic Misconduct

The University misconduct policy<sup>3</sup> applies. Students are encouraged to discuss the assignment topic, but all submitted work must represent the individual's understanding of the topic.

The subject staff take academic misconduct very seriously. In this subject in the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.

---

<sup>3</sup>See <https://academichonesty.unimelb.edu.au/>