Software Engineering

Testing

3 Ba INF
2023 – 2024

Kasper Engelen
kasper.engelen@uantwerpen.be

1 Practical

- Deadline: **Sunday, October 5, 2023, 22u00**
- Hand in via **Blackboard** as `GroupX_testing.zip`
- The code can be found on Blackboard (`add_course.py`).

2 Context

In this assignment you will learn how to do white box testing using an existing piece of code. More concretely, you will learn how to

- clean up the code, and create a control flow graph,
- compute the independent paths and gain insight into what such paths are,
- do condition testing using "multiple condition coverage",
- do loop testing.

3 Report and evaluation

The evaluation will be **in-person**, where we will discuss your solutions together. You are free to balance the work across your team of 3 students. At the evaluation **every student is expected to understand and explain** the solution.

While making the assignments you should make use of the attached templates. When you have finished the assignments, you can hand them in via **Blackboard as a zip-file**. The templates are essential to ensure a **smooth evaluation** and to ensure you do not **miss out on any grades!**

4 Assignment

4.1 Analysing the algorithm

Before we can do any white-box testing, we must first analyse the code of the module we want to test. All techniques in this assignment will make use of a *control flow graph* (CFG): a graph in which the nodes describe the statements, and the edges describe which statements follow which statements.

On Slide 21 you have seen a number of constructs (if-else, while, boolean expression) and how to translate those into a CFG. In this assignment we will restrict ourselves to only using those constructs. We thus first have to remove any complex boolean expressions, for-loops, complex if-statements, etc. and replace them with the constructs for if-else statements and while loops.

Note: The theory in the slides **makes the assumption that there is a single exit point in the code** (i.e. one single return statement at the end). In practice this is **not true**. You will have to **improvise** here and there to make sure the CFG only has **one exit node**! The formulas and techniques are **not guaranteed to work** with multiple exit nodes! Ask the assistant for help if you are not sure what improvisations to make.

Note: The while-statements in the theory also do not have **continue** and **break** statements. You will have to improvise here as well.

Assignment 1

Complete the following tasks:

1. **Study** the provided code, in particular the `enroll_in_courses` function.
2. Re-write the code so that it only uses **while-loops** instead of for-loops.
3. Replace all assignments that have **boolean expressions** with if-else statements that use assignments of `true` and `false`.
4. Replace all **if-elif-else** statements using only if and else.
5. Replace all if-else statements that have **boolean expressions** in their conditions with **nested** if-else statements.
6. Create two CFGs. **Indicate which node corresponds to which lines of code**.
   (a) One CFG for the code after step 4
   (b) One CFG for the code after step 5

4.2 Basis path testing

A first technique we will apply to construct test cases is called *basis path testing*. Each such a test case corresponds to a path through the CFG. This technique will ensure that for each node in the CFG there exists at least one test case that traverses that node. On Slides 21, 22, 23, 24, 25, and 26 you will find information about basis path testing.

Note: On Slide 24 you will find a heuristic to construct a set of paths. Note that the "pick the most simple entry/exit path" step might not always work. If the chosen initial path is **impossible** (i.e. no inputs exist that trigger this path), then the other paths might also be impossible. **Choose wisely!**

Note: As mentioned earlier, make sure your graph has only one entry and one exit! Otherwise the theory will not work.

Assignment 2

Complete the following tasks:

1. Use the CFG you constructed in **Step 6a** of the previous exercise.
2. Compute the **Cyclomatic Complexity** (Slide 22). Verify that **all formulas** give the same result.
3. Construct a set of **independent paths** (Slide 24).
4. What is the **relation** between the cyclomatic complexity and the independent paths?
5. Come up with **test cases** that trigger these paths. Make sure you have inputs for **as many paths as possible**. Construct a different set of independent paths if needed. Motivate the missing test cases.
6. Construct a **table** similar to Slide 25. You do not have to compute the output. Make sure you **motivate** for every test case why it will trigger the path.

4.3 Condition testing

The next technique we will apply is condition testing. We wish to find test cases to thoroughly test the different conditional statements. To keep things manageable we will restrict ourselves to *multiple condition coverage* (Slides 27 and 28).

In multiple condition coverage we wish to test all possible `true`/`false` combinations of every simple condition. A simple condition is one that does not have sub-conditions. Examples are "`x > y`", "`a == b`", etc. Compound conditions such as "`a > b && a = 2*k`" first have to be split into such simple conditions. If you constructed your CFG in **Step 6b of Assignment 1** correctly, every simple condition should already have its own dedicated node.

Assignment 3

Complete the following tasks:

1. Use the CFG you constructed in **Step 6b of Assignment 1**.
2. Come up with a set of paths such that every `true`/`false` combination of every condition is covered.
3. Come up with inputs that trigger the paths constructed in the previous step. See Slides 27 and 28.
4. Construct a **table** similar to Slide 25. You do not have to compute the output. Make sure you **motivate** for every test case why it will trigger the path.

4.4 Loop testing

Finally, in order to more thoroughly test the behaviour of the loops, we will apply loop testing (Slide 29). In this technique you will analyse the loops in the CFG, and construct test cases that trigger a specific number of iterations of those loops.

Let us take, for example, a loop that should at most have n passes. In that case we want paths and test-cases for the following number of iterations:

- zero iterations (the loop is skipped),
- 1 iteration,
- 2 iterations,
- m iterations, with $2 < m < n$ (i.e. we want an "intermediate" number of iterations),
- $n - 1$ iterations,
- n iterations,
- $n + 1$ (or more) iterations (i.e. we try to "break" the loop).

Assignment 4

Complete the following tasks:

1. Use the CFG you constructed in **Step 6b of Assignment 1**.
2. Construct a set of paths such that each path covers a specific **number of iterations**. See Slide 29 for the various amounts of iterations.
3. Are there certain amounts of passes that **cannot be triggered**? Why?
4. This technique is meant to thoroughly test loops. Do you agree that this technique will always lead to exhaustive tests? Why (not)?
5. Construct a **table** that lists each path and the associated input (Slide 29). You do not have to compute the output. Make sure you **motivate** for every test case why it will trigger the path.

5 Templates

Below you will find a number of templates that you should use for the assignment. You can re-create these templates using a program of choice (LaTeX, MS Word, …).

5.1 Test cases table

Path	**Input**	**Motivation**
The path	The function arguments	Why will it trigger this path?
Another path	…	…