SE465
Software Testing and Quality Assurance
Assignment 1, version 2*

Patrick Lam
Release Date: January 9, 2026

Due: 11:59 PM, Friday, January 30, 2026
Submit: via git.uwaterloo.ca

Getting set up

We will create a copy of the starter repo for you in your `git.uwaterloo.ca` account. You need to log in to `git.uwaterloo.ca` for that to work.

I expect each of you to do the assignment independently. As stated in the course outline, you can ask questions of generative AI, but you cannot submit text or code that comes from GenAI. I will follow UW's Policy 71 for all cases of plagiarism.

Submission instructions:

Commit **and push** your modifications back to your fork on `git.uwaterloo.ca`. It's git, so you can submit multiple times. After submission, **please make a fresh clone of your submission to make sure you have uploaded all necessary files**.

Submission summary

Here's what you need to submit in your fork of the repo. Be sure to commit and **push** your changes back to `git.uwaterloo.ca`.

- (Q1): `a1q1_estimate_size/estimate_size_test_suite.py`, `a1q1_count_tests/count_tests_test_suite.py`
- (Q2): `a1q2a/network_retrieve.py`, `a1q2b/test_suite.py`, `a1q2c/test_suite.py`
- (Q3): `a1q3/coverage_tests.py`, `a1q3/rle_mutant1.py`, `a1q3/rle_mutant2.py`, `a1q3/mutants_kill.py`
- (Q4): `a1-chatgpt.md` in the root of your repository
- (Q5): `coverage_tool/coverage.py`, `coverage_tool/coverage_tool_test_suite.py`
- (Q6): `a1q6/piwigo_metamorphic_test_suite.py`.

*version 3: assign points; version 2: adjusted q3c

Question 1: Statement and Branch Coverage (15 points)

In this question you will write Python test suites that ensure statement and branch coverage. Write test suites that ensure 100% statement and branch coverage on the `estimate_size function` in the `a1q1_estimate_size/` directory (easy) and `count_tests` in the `a1q1_count_tests/` directory (slightly harder). Add them to the `estimate_size_test_suite.py` and `count_tests_test_suite.py` files respectively.

The suites must run with 0 failures and 0 errors.

We will rerun your test suites and count coverage ourselves, but you'll find instructions in the test suite py files about how to generate coverage reports so that you can know if you've achieved the goal yet or not.

Question 2: Test engineering (15 points)

This is meant to be a practical question that has you engaging with various libraries used to support writing tests.

Part 2(a): flaky tests (2.5 points)

One of the reasons for flaky tests is the use of `sleep()` instead of waiting for something to be done. I've created a somewhat contrived example of that, and your task is to add a `wait()` in place of the `sleep()`.

In the skeleton code, you will find `a1q2a/network_retrieve.py` which has an asynchronous `retriever()` function. (You'll see async/await in CS 343, though I think under different names). This function retrieves a resource from the Internet and makes things super flaky by waiting for a random number of seconds, up to 10.

In the same file, you will also find function `network_retrieve()`. I'll skip over some details, but this function calls `retriever()`, which runs in the background (i.e. asynchronously). But, `network_retrieve()` then does the flaky-test thing of waiting 5 seconds. Sometimes that is long enough. Sometimes it isn't. It then retrieves the result from a global variable and returns it to its caller.

There is a test suite, which you can invoke using

```
python -m a1q2a.test
```

from your `a1` repository root. (You may have to install pip packages `aiohttp` and `requests`; see the source code of `network_retrieve.py` for instructions).

Your task is to add code to `network_retrieve.py` to make `network_retrieve()` wait for the result from the `retriever`—I've put three TODOs where I recommend doing so. You should use `threading.Event`. You may edit `settings.py` to set the constant `GRADING` to `True` to eliminate the simulation of flakiness and make things deterministic; we will grade by running the code with `GRADING` set to `True` and by manually diffing your code from the skeleton.

Part 2(b): fake objects (2.5 points)

For this part, I've created a badly implemented `count_characters.py` module. Class `CharacterCounterClass` is bad in a lot of ways, but the way that we're addressing here is that it is reading input from a hardcoded file from the filesystem.

The idea here is to practice inserting a fake object in place of `CharacterCounterClass`. In a more realistic case, the real code might be accessing a database, and your fake object implements an in-memory database. This question aims to give you the flavour of doing that.

The mechanics of inserting a fake object differ. Dependency injection is not part of this course's material, but it would be a fairly common mechanic you can leverage.

You can run the test suite, such that it is, with the command

```
python -m a1q2b.test
```

Your task is to modify `test_suite.py` (and only that file) so that it still tests `count_characters()` from `CharacterCounterClass` but does not perform any disk IO. (`a1q2b/test.py` invokes `test_suite.py`). One could use Python's `unittest.mock` library but the intent of this question is for you to do it manually. Please don't use `unittest.mock`.

We will grade this question by pulling your `test_suite.py` file into a fresh copy of the `a1q2b` directory minus the `lorem-ipsum.txt` file, running it, making sure it passes, and checking that your solution still contains the assertion from the skeleton.

Hints: Python 3 provides `io.StringIO` which creates a file-like object from a string, and you can cut-and-paste the contents of the file into a multiline string with a triple quote (`"""`). You are allowed to subclass `CharacterCounterClass` and use your implementation in the test.

Part 2(c): implementing behavioural testing (10 points)

In directory `a1q2c` you will find a `Model` and a `Controller`. Your task is to create a number of mocks for the `Model` using Python's `unittest.mock`. I've provided skeleton tests for you to fill in as well as directions about what you put in each of these tests. Each test that you write is worth 2.5 points.

The `settings.py` file contains a setting `WHICH_STORY` which determines which story runs in the `Controller`.

Run tests 1-2 with invocation:

```
python3 -m unittest a1q2c.test_suite -k calls
```

and tests 3-4 with invocation:

```
python3 -m unittest a1q2c.test_suite -k three
```

The skeleton gives more specific instructions about tests 3 and 4 and when they should succeed and fail.

wait_once (1) Write a test that creates a mock `Model` that ensures that the `Controller`'s selected `model_story` calls `wait()` exactly once. This test is supposed to succeed for story "zero" but not "one".

wait_four_times (2) Write a test that creates a mock `Model` that ensures that the `Controller`'s selected `model_story` calls `append_to_resource()` exactly four times. This test is supposed to succeed for story "zero" but not "two".

stubbing and faking (3) Write a test that tests the Controller's calls to the Model, but hardcodes the Model's response to `get_resource`. (4) Write another test that creates a mock but maintains a real list for model.resource. These two tests aren't supposed to succeed for story "zero".

Question 3: Mutation Analysis (15 points)

Consider the following program:

```
1  # adapted from https://www.geeksforgeeks.org/python/run-length-encoding-python/
2  def run_length_encoding(inpt):
3    """
4    Issue  python -m doctest rle.py  to run the doctests.
5
6    >>> print(run_length_encoding('aaabb'))
7    a3b2
8    """
9    output = ''
10   count = 1
11
12   if len(inpt) >= 2 and inpt[0] != inpt[1]:
13     output += inpt[0] + '1'
14     inpt = inpt[1:]
15
16   for i in range(1, len(inpt)):
17       if inpt[i] == inpt[i - 1]:
18           count += 1
19       else:
20           output += inpt[i - 1] + str(count)
21           count = 1
22
23   output += inpt[i - 1] + str(count)
24
25   return output
```

You can execute all of the tests for this question from the `a1` directory of your repo with the following command:

```
python -m a1q3.test
```

Our provided test suite achieves 100% statement coverage for `run_length_encoding()`.

(a) (5 points) Manually inject faults into this program. Specifically, propose two non-stillborn and non-equivalent mutants of this program. Put the mutated versions of the code in functions `run_length_encoding_mutant1` and `run_length_encoding_mutant2`. Clearly indicate (with comments in the code) where you've mutated the program and which mutation operator you are applying. Does the initial test suite (`test_provided_one` and `test_provided_two`) kill your mutants? Write your answer ("Y" or "N") in `a1q3/mutants_kill.py`.

(b) (5 points) Write unit tests that kill each of the mutants and put them in `a1q3/coverage_tests.py`. You'll find skeleton tests `test_kill_mutant_1` and `test_kill_mutant_2`. If the provided tests already kill the mutants, you are free to re-use them. We are going to run your tests to verify that they succeed on the original program and fail on the mutant with an assertion failure; the grading script will modify `a1q3/settings.py` to run the different variants.

(c) (5 points) Find a bug in `run_length_encoding` and provide a test case `test_bugfix()` that fails on the provided version of the method but passes on a fixed version of the method. (We are not asking you to submit the fixed version.) Are your mutants from part (a) killed by the test case that also kills the bug? Again, write your answer in `mutants_kill.py`, as well as a one or two-sentence description of the value of your mutant-killing tests from part (b), especially as compared to statement coverage; can you say that one is better?

Question 4: ChatGPT's take on Implicit Oracles (15 points)

I asked ChatGPT this question:

> What are some implicit oracles useful for software testing?

Here's what it told me:

> Implicit oracles are test oracles that don't explicitly define expected behavior in a formal, documented way, but instead rely on inherent properties of the software, system behavior, or domain knowledge. These types of oracles are often used in situations where formal documentation or specifications are unavailable, incomplete, or difficult to construct. They typically rely on a tester's intuition, experience, or observable system characteristics to determine whether the system is behaving as expected.
>
> Here are some examples of **implicit oracles** that can be useful for software testing:
>
> **1. Human Expert Knowledge (Domain Expertise)**
>
> **Description:** Testers or domain experts can leverage their deep understanding of the system or business domain to evaluate whether the software behaves as expected.
> **Example:** A tester with expertise in banking systems can manually review whether a newly implemented feature (e.g., fund transfer) follows the expected logic based on real-world banking rules and principles.
> **Use Case:** This is especially useful when documentation is lacking or ambiguous, but the tester understands the typical behavior of similar systems in the domain.
>
> **2. Behavior Consistency**
>
> **Description:** If the software behaves consistently under similar conditions, you can use its behavior as an implicit oracle. This relies on the principle that software should act predictably and consistently across different runs.
> **Example:** If a web application consistently loads pages in a certain amount of time under normal conditions, a significant deviation in response time can be flagged as a potential issue.
> **Use Case:** This is useful for performance testing or verifying that the system maintains stable behavior over time, especially for non-functional requirements like load time or uptime.
>
> **3. Equivalence Class Partitioning**
>
> **Description:** Rather than comparing results against a documented specification, you can test different equivalence classes or categories of input to see if they produce valid outputs. Implicitly, this is based on the assumption that similar inputs should yield similar outputs.
> **Example:** For an age field that accepts numeric values, you can test different equivalence classes like valid ages (e.g., 18-60), boundary conditions (e.g., 0, 100), and invalid ages (e.g., -5, 150) to check if the software handles these correctly.
> **Use Case:** This works well when you don't have a detailed specification but have a general understanding of what ranges or categories of input are expected to be valid.
>
> **4. Consistency with Previous Versions**
>
> **Description:** You can compare the current version of the software to a previous stable version to verify that the system still behaves correctly. If the previous version behaved well, significant deviations in the new version may be considered failures.
> **Example:** A regression test where the software's output or behavior from an old release is compared with the new one to ensure that existing features haven't been broken.
> **Use Case:** Especially useful in continuous integration and continuous delivery (CI/CD) pipelines to ensure backward compatibility and prevent regressions.

[...]

10. Expected System State Transitions

Description: Implicit oracles can be based on the expected transitions between different states of the system. If the system moves between states in an unexpected way or fails to reach the desired end state, this is considered a failure.
Example: In a login system, if the user successfully logs in and is taken to the dashboard, this is an expected state transition. If the system takes the user to an error page instead, this is a failure.
Use Case: Useful for systems with clear state transitions, such as workflows, state machines, or applications with clear progression (e.g., multi-step forms, user onboarding, or approval processes).

11. Third-Party Integrations and External Systems

Description: When software interacts with external systems or APIs, their behavior can act as an implicit oracle. If an external system is expected to return certain data (e.g., via an API call) and doesn't, this can signal an issue.
Example: A payment gateway that successfully processes transactions should return a success message, while an unexpected error code or delay in response could indicate an issue.
Use Case: Especially useful for testing integrations, such as with third-party APIs, payment gateways, or external databases.

12. End-to-End User Scenarios

Description: Implicit oracles can also be derived from expected user journeys or end-to-end workflows. If the system allows users to complete tasks or workflows in a smooth, uninterrupted fashion, it's likely functioning correctly.
Example: In an e-commerce application, a typical flow of browsing products, adding them to the cart, and completing a checkout without errors is considered a sign of correct system behavior.
Use Case: Useful in functional testing, especially when testing critical business flows like purchases, user registrations, or data submissions.

Conclusion

Implicit oracles are often more flexible and adaptable than formal, documented test oracles. They rely on human intuition, domain expertise, system behavior, and other indirect cues to identify problems. While they can be highly effective, especially when formal documentation is lacking or incomplete, they require skilled testers who have a good understanding of the system and its intended behavior.

What ChatGPT says is an implicit oracle does not match with what the literature and Wikipedia (which follows the literature) say: none of its examples fit the definition of implicit oracle from the literature.

Your task: for each of the categories of oracle { specified, derived, human }, identify one of the examples from the ChatGPT output that better fits this category. Briefly explain (1–2 sentences) why this example fits into your chosen category, and say whether or not the example is either overall correct as a description of an oracle, or incorrect in at least one aspect, explaining why that is.

Put your answer in a file named "a1-chatgpt.md" in your repository. It can be plain text or Markdown.

Question 5: Developing a Coverage Checker (40 points)

The `Coverage` class in the *Fuzzing Book*[1] records the lines that have been executed. In this question, your goal is to extend a modified version of this class to compute (1) the set of executable lines, (2) the set of executed branches, and (3) the set of executable branches (i.e. the skeleton of the program's control-flow graph). Plus, you will write a test suite for your implementation.

You will find the skeleton `coverage.py` in your `a1` repo under the `coverage-tool` directory. You can run the built-in doctests with `python3 coverage.py`. In this code, the type `Location` is a tuple containing a function name (string) and a line number (int).

Executed branches (5 points). Let's start with the method `executed_branches()`. This method returns the set of executed branches, using the set of observed lines in the trace as input. By branch I include the edge from a statement to its successor, as well as edges for loops and conditionals. The *Fuzzing Book* provides a starting point for its `Coverage` implementation, but you will need to adapt it a bit.

Executable lines (5 points). Fuzzing doesn't need the denominator, i.e the number of unexecuted lines or branches, but we might want it for other reasons. In this part, you will compute the complete set of executable lines in the functions that have been visited. I think that the easiest way to do this is using the Python disassembler, in the `dis` module.

My template for `populate_executable_lines()` requests the function bytecode by using `dis` to create a `Bytecode` object for each function that has been seen in the collected trace. (The populate functions are called when tracing is turned off.)

Look through the documentation for `dis`, at `https://docs.python.org/3/library/dis.html`, and find a function that is going to tell you line numbers. You'll want to pass to this function `function_bytecode.codeobj`. You also want to do something (not much) to the results from this function so that you can append tuples `(function_name, lineno)` to `self._lines`.

(I guess you're allowed to ask an LLM for this information, but I recommend reading the documentation. Hopefully you won't hallucinate if you're doing the reading.)

Executable branches (20 points). The largest part of this question is implementing function `populate_succ()`. This function computes the key part of a control-flow graph—the successor relation. For each executable line ℓ, the dict `self._succ` should contain a set of `Locations` containing the successors of ℓ.

In my implementation, I first compute a dict mapping each instruction's bytecode offset (obtainable from the `.offset` field of a bytecode instruction) to its line number. Because not every instruction has a line number (sometimes it is `None`), I track the most recently seen line number as I iterate through the function's instructions, and store the instruction's most recently seen line number for it. (There is also an API call that will compute this dict, so you don't have to do it yourself.)

Next, I get a fresh `get_instructions` and iterate on it. There are two kinds of edges that you need to add: (1) edges from an instruction to its jump target; and (2) edges from an instruction to the one immediately following it. These edges are the same as the edges in the executed branches part, although we are computing a dict here and not a set. This CFG operates in line numbers and not offsets, so you sometimes need to translate.

In theory, the CFG shouldn't contain a successor edge to the immediately-following instruction after an unconditional jump (i.e. case (2) above should only apply when the instruction is not a jump). Ignore that detail for this question.

Test suite (10 points). Finally, write a test suite for the code that you've written, and provide an argument that your test suite is sufficient (as a paragraph-length comment in `coverage_tool_test_suite.py`). You can define "sufficient" in any reasonable manner, but tell us what your definition is. You will need to provide some input for `Coverage` beyond the provided `cd()` method. There is a starter test in the `coverage_tool_test_suite.py` file.

[1] `https://www.fuzzingbook.org/html/Coverage.html`

Question 6: Metamorphic Testing (20 points)

In this question, you will write two metamorphic tests: one which I specify and one which you can choose. The context is the REST API for Piwigo, an image gallery. You are going to write tests for various API calls.

In your skeleton repository, under `a1q6`, you'll find `piwigo_metamorphic_test_suite.py`, where you will add your tests.

I've supplied a sample metamorphic test, which verifies that the set of images tagged "bird" *and* "fauna" is a subset of the set of images tagged "bird" *or* "fauna".

Tag inequality. Another metamorphic property is that the sum of the number of images returned (as the `counter` field) for each of the tags is greater than or equal to the number of tagged images.

For example: let's say that there are two tags, "a" and "b", and four images. Images {1, 2, 3} are tagged "a", while images {1, 2, 4} are tagged "b". If you sum the number of images tagged as "a" and the number of images tagged "b", then you will get 6 images—but certainly at least 4:

$$\#(A) + \#(B) \geq \#(A + B),$$

or, more specifically,

$$3 + 3 \geq 4.$$

(5 points) Write a test that calls `pwg.tags.getList` and sums the number of images reported in the `counter` for each of the tags. Next, the test calls `pwg.tags.getImages`, passing it all of the reported tag ids, and creating an overall set of images which have a tag of any sort. Finally, the test must assert that the sum of the number of images returned by `getList` is greater than or equal to the number of images returned by `getImages`.

Your choice. (15 points) Look through the piwigo API, which you can explore at `https://gallery.patricklam.ca/tools/ws.htm`, and find a different metamorphic relation. You don't have permission to make any changes on my gallery, but there are still other relations that exist between methods that you are allowed to call.

For instance, you can call `pwg.images.search` and pass it some query (a query is just a string that it searches for). Find somewhere else in the API where there should be information that is consistent with the `search` result. (Note that some APIs are "admin only", as indicated in the API explorer, and you can't call them.) There are also some metamorphic relations under `pwg.categories.*`. You don't have to do anything I've mentioned; it just has to combine results from different parts of the Piwigo API.

Implement your test in `test_students_choice_metamorphic` and write a doc comment above this function briefly explaining your metamorphic relation.