CSCI 2134 Assignment 2

Due date: 11:59pm, Saturday, October 12, 2024, submitted via Git

Objectives

- Practice developing effective unit tests
- Practice implementing unit tests with JUnit

Preparation:

Clone the Assignment 2 repository

https://git.cs.dal.ca/courses/2024-fall/csci-2134/assignment2/????.git

where ???? is your CSID.

Problem Statement

Create a set of unit tests using JUnit5 for provided classes to find some bugs and test the correctness of the code.

Background

You have inherited some buggy code for doing Cache simulations. A specification for the entire project is provided, but the code you have now is only a partial implementation. Your boss fired their previous developer because the former developer did not do any testing. Your boss has hired you to write a comprehensive set of tests for part of the codebase. **For now, she just wants you to create the test suite.**

You will be provided with a partial codebase for creating a Cache and Cache Simulator. This codebase includes a specification, a starter JUnit5 test class, and a list of classes for which you are to create unit tests. Your job is to create the test suite and **identify** the bugs. **You do not need to fix the bugs.** You will fix the bugs in a later assignment so do not spend time fixing them now.

Note that the code you are provided does not have a main method. Additional classes will be provided in a later assignment. A specification for the entire project is provided, but the code you have now is only a partial implementation. The other classes in the specification will be provided in a later assignment. **You only need to write unit tests, not code.**

Tasks

1. Read the specification of what the code is supposed to do in `docs/specification.pdf`
2. Create a set of unit tests using JUnit5 for the following classes and methods:
    - `Cache.java`
        - `getSize()`, `getRank()`, `findData()`, `evictData()`, `getContents()`, `fetchData()`
    - `CacheSim.java`
        - `reset()`, `getCacheMisses()`, `cacheToString()`
3. Use a separate test class for each of the above target classes. Some sample empty tests and real tests have been provided. For each test class:
    a. You can decide if you want to use white-box testing, specification testing, or grey-box testing.
    b. Create as many tests for each method of each class as needed. Recall from class that we discussed ways of determining how many tests we would need. By analysing the code (white-box testing) or the specification (black-box testing).
    c. Each test should provide an appropriate message if it fails.
    d. Use good formatting and documentation in your tests, just like for any source code.
4. All the test classes should compile and be runnable in IntelliJ. **If your test classes do not compile, you will receive 0 on the assignment.**
5. Record all detected errors in a file called `errors.txt` in the `docs` directory. You do not need to debug, just record the method and class that failed. Make sure it is not your test that is buggy and causing a failure. Then, report each error in `errors.txt` with the following information:
    a. Class name
    b. Method name
    c. Test name that caught the error
    d. Message that the test method generated

    This information will be used to assess the number of errors found by your tests. An example is provided.
6. Add and commit your tests and updated `errors.txt` file. Push your commit(s) to the remote repository. As the primary branch is called “main” you may need to use the command “git push origin main”. Remember to check that all your files have been submitted using the web interface to git.

Submission

All test classes and an updated error.txt should be committed and pushed back to the remote Git repository.

Grading

The following grading scheme will be used:

Task	4/4	3/4	2/4	1/4	0/4
Thoroughness (40%)	All or nearly all test cases are covered	Most test cases are covered	Some of test cases are covered	Few test cases are covered	No test cases created
Overlap (20%)	Nearly all tests have a purpose. There are very few redundant tests.	Most tests have a purpose. There are a few redundant tests.	At least half the tests have a purpose. Half the tests are redundant.	Most of the tests test the same condition.	All the tests test the same condition.
Error Detection (20%)	5+ distinct errors are detected.	4 of the errors are detected.	3 of the errors are detected.	1-2 of the errors are detected.	None of the errors are detected.
Code Clarity (20%)	Code looks professional, follows style guidelines and has very few issues. Code is very readable.	Code looks ok but has a few inconsistencies. Mostly follows style guidelines. Code is readable.	Code is sloppy with many inconsistencies. Sometimes follows style guidelines and is a little hard to read	Code is very sloppy and does not follow style guidelines. Code is hard to read.	Code is illegible.

Hints

1. There are at least 5 distinct bugs in the provided code. This does not mean that exactly 5 tests will fail. One bug may cause multiple tests to fail. One bug may not be found by any of your tests (until you expand the scope of your test cases).
2. If you are writing a unit test where the expected output is not an exact value but rather a range of values, you can use asserts like: `assertTrue(val >= expectedVal)`.
3. If you are writing a unit test which tests that an exception *is* thrown, you want to manually call `fail()` after calling the method that should throw the exception. Then, leave the catch block empty so that the test can run to completion and pass.
4. In the lab and lectures, we considered unit tests for a Matrix class where the Matrix object was square, tall, short, etc. Similar different "states" a cache might have are how many requests for data were made relative to the cache capacity. For example, an empty cache, a cache which is partially fully, a cache which is full, etc.