CSCI 5308 Assignment 1

Due date: 11:59pm, Monday, June 21, 2020, submitted via Git

Objectives

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

Preparation:

Clone the code on the repository https://git.cs.dal.ca/courses/2021-summer/csci-5308/assignment1/ronnie

Problem Statement

You have three main tasks in this assignment:

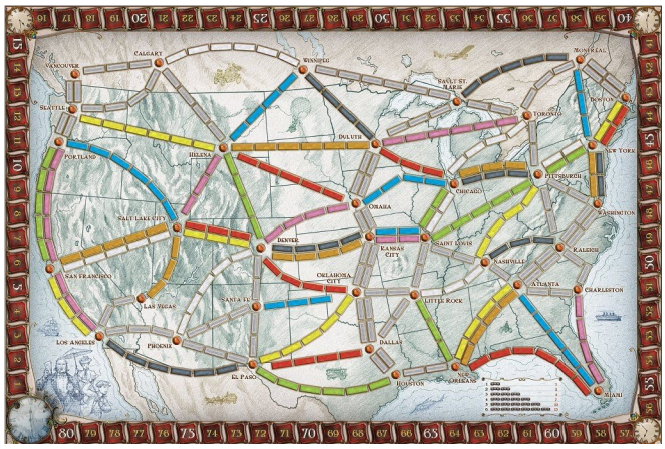
- Create a set of unit tests using for the provided classes which are already developed
- The correctness of the code using your unit tests and if you find any bug fix it.
- Use TDD to design unit tests and develop the parts of code that are missing, use the specification on the comments to do this.

Background

You have inherited some buggy and incomplete code for computing shortest path solutions to the board game Ticket to Ride. The previous developer left the company, so it is your job to finish the software. Your boss has hired you to write a comprehensive set of tests for part of the codebase.

Given a game board of rail segments and a list of routes (pairs of cities), the code is supposed to compute the total cost of building a network between the given routes, assuming that the shortest distance for each route is chosen. This can be computed by computing shortest paths for each route using Dijkstra's shortest path algorithm.

You will be provided with a partial codebase for distance computation, a specification, some JUnit5 test class, and a list of classes for which you are to create unit tests. Your job is to create the test suite, identify and fix the bugs (if any) and finish the methods not implemented in the classes.



Note that the code you are provided does not have a main method and does not have the method implementing Dijkstra's shortest path algorithm. These will be provided in a later assignments. **You only need to write unit tests and finish up the classes already started, not code for the Ticket to Ride problem in this assignment.**

Task

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 using the provided classes. Remember that each class needs to have its own test class, with several tests to check the implemented methods.
    - `City.java`
    - `Link.java`
    - `CityComparator.java`
3. Implement 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. Create as many tests for each method of each class as needed. But remember, each method must have at least one unit test.
    b. Each test should provide an appropriate message if it fails.
    c. 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. Each error should have 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. Commit as you go and add a clear message about what you did, e.g., finished creating the unit test for setCity() method, then commit with the message about this part of your work before moving to something else.

7. Push back your work to **YOUR** remote repository, not to the one that you cloned. Everyone has their own repository to deliver this assignment, e.g., https://git.cs.dal.ca/courses/2021-summer/csci-5308/assignment1/yourscid.

8. Remember to check that all your files have been submitted using the web interface to git.

Submission

All the code 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 (30%)	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%)	All or nearly all errors are detected.	Most of the errors are detected.	Half the errors are detected.	Few 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.
Separate commits (10%)	Used a separate commit to every change made	Used a separate commit to almost all changes made	Used a separate commit to half of the changes made	Sometimes used a separate commit to the changes made	Has not used separate commits.