CSCI 2134 Assignment 2

Objectives

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

Problem Statement

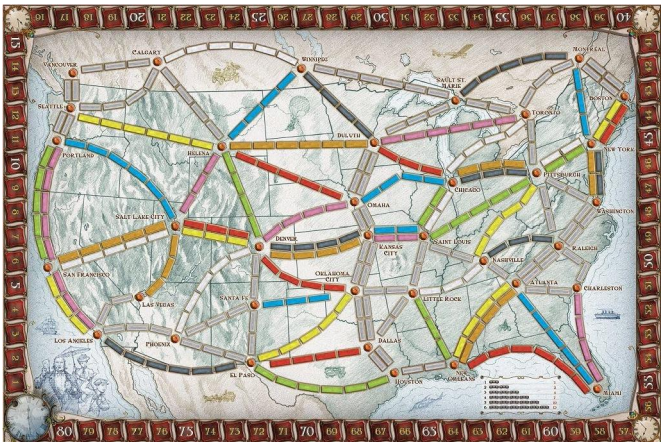
Create a set of unit tests using JUnit5 for provided classes to find some bugs and test the correctness of the code.

Background

You have inherited some buggy code for computing shortest path solutions to the board game Ticket to Ride. 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.**

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, 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 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 and does not have the method implementing Dijkstra's shortest path algorithm. These will be provided in a later

assignment. **You only need to write unit tests, 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 for the following classes:
    - `City.java`
    - `Link.java`
    - `CityComparator.java`
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, black-box 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. 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 and push back to the remote repository. As the primary branch is called “main” you will 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 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%)	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.

Aside about the Ticket to Ride Problem

The code is supposed to compute shortest paths for each given route (pair of cities) and return a rail network that contains each of these shortest paths. This is not necessarily the rail network connecting all those cities with as few rail pieces as possible because it doesn't consider sharing pieces between routes. In the real Ticket to Ride board game you would also want to compute a minimum spanning tree between just the selected cities. This is known as the Steiner Tree problem and is NP-hard so we'll stick to shortest paths between pairs of cities in this assignment!

Image source: https://meepletown.com/wp-content/uploads/2011/03/TTR.jpg (Retrieved on January 28, 2021)