CSCI 2134 Assignment 4

Due date: 11:59pm, Wednesday, April 6, 2022, submitted via Git

Objectives

Extend an existing code-base and perform some basic class-level refactoring in the process.

Preparation:

Clone the Assignment 4 repository

https://git.cs.dal.ca/courses/2022-winter/csci-2134/assignment4/????.git

where ???? is your CSID.

Problem Statement

Take an existing code-base, add required features, test it, and refactor it as necessary.

Background

The Ticket to Ride solver is moving on to version 2. Your boss wants you to add some new features to the program that have been requested by the customer. She has hired you to extend the code. She also mentioned that the original designer of the code did not do a great job and wondered if there was any way to improve the code. She will provide you with (i) the code-base, (ii) the existing requirements, and (iii) the specification of the additions to be made.

Your job is to (i) create a design for the additions, (ii) implement the additions, (iii) create unit tests for the additions, and (iv) identify opportunities for class-implementation and class-interface refactoring, and (v) do some refactoring where appropriate. May the source be with you!

Task

1. Review the old specification (**`specification.pdf`**) in the **`docs`** directory. You will absolutely need to understand it and the code you are extending.
2. Review the extension specification at the end of this document, which describes all the extensions to be done.
3. Design and implement the extensions using the best-practices we discussed in class.
4. Provide a readable, professional looking UML diagram of the updated design. This should be a PDF file called **`design.pdf`** in the **`docs`** directory.
5. For each new class that you implement, you **must** provide unit tests in the form of Junit5 tests. You should design your classes and modify existing classes to facilitate the testing.
6. In a file in the **`docs`** directory called **`refactoring.txt`** list all the class-implementation and class-interface refactoring that you will do and refactoring that you would recommend.
7. Perform any class-implementation and class-interface refactoring that you promised to do.
8. **Bonus**: Research the *Factory* pattern that is used to instantiate classes derived from the same superclass or interface. E.g., when you create different types of links they could implement a Link interface or be subclasses of an abstract Link class and be constructed by a new

LinkFactory class. Implement the Factory pattern to improve the creation of Values in *Value.* Be sure to update the UML diagram and provide unit tests.

9. **Commit and push back** everything to the remote repository.

Grading

The following grading scheme will be used:

Task	4/4	3/4	2/4	1/4	0/4
Design (10%)	Design is cohesive, meets all requirements, and follows SOLID principles	Design meets all requirements and mostly follows SOLID principles	Design meets most of the requirements.	Design meets few of the of requirements.	No design submitted.
Implementation (25%)	All requirements are implemented	Most of the requirements are implemented	Some of the requirements are implemented	Few of the requirements are implemented	No implementation
Testing (25%)	Each new class has a set of unit tests associated with it. All requirements are tested. If implementation is incomplete, the test is still present.	Most of the new classes have an associated set of unit tests. Most requirements are tested.	Some of the new classes have an associated set of unit tests. Some requirements are tested.	Few of the new classes have an associated set of unit tests. Few requirements are tested.	No testing
Refactoring Description (10%)	At least 4 class level refactoring suggestions that follow SOLID principles and make sense.	At least 3 class level refactoring suggestions that follow SOLID principles and make sense.	At least 2 class level refactoring suggestions that follow SOLID principles and make sense.	At least 1 class level refactoring suggestions that follow SOLID principles and make sense.	No refactoring suggestions.
Refactoring Implementation (10%)	At least 2 class-level refactoring suggestions are implemented correctly.	2 class-level refactoring suggestions are implemented, with 1 being done correctly.	1 class-level refactoring suggestion is implemented correctly.	1 class-level refactoring suggestion is implemented.	No refactoring suggestions implemented.
Code Clarity (10%)	Code looks professional and follows style guidelines	Code looks good and mostly follows style guidelines	Code occasionally follows style guidelines	Code does not follow style guidelines	Code is illegible or not provided
Document Clarity (10%)	Documents look professional, include all information, and easy to read	Documents look ok. May be hard to read or missing some information.	Documents are sloppy, inconsistent, and has missing information	Documents are very sloppy with significant missing information	Documents are illegible or not provided.
Bonus [10%]	Factory pattern implemented and tested.	Factory pattern implemented	Factory pattern partially implemented	Factory pattern attempted.	No attempt

Submission

All extensions and files should be committed and pushed back to the remote Git repository.

Hints

1. You can get a large number of marks without writing any code.
2. Do the design first and look at refactoring as you design.
3. The extensions are intended to require minimal code.
4. Testing is as important as implementation
5. The example input in **`system_tests`** has been updated to match the required extensions

Specification of Required Extensions

Background

Our customer has requested that the Ticket to Ride solver software accept new types of input and be more robust to user input errors. You will need to

- Extend the software to support two different players,
- Extend the software to support new player specific links to represent a game partially in progress, and
- Handle improper input in a user-friendly way.

Specification: Changes to Program Input

1. After a link is read there may be a fourth value to read, red or green.
   - For example, “A 42 B red” represents a red link and “B 13 C green” represents a green link.
2. After a route is read there may be a third value to read, red or green.
   - For example, “ A B red” represents a red route and “B C green” represents a green route.

Specification: Functional Changes

1. Colored routes can only use uncolored links or links of the correct color:
   - Red routes can only use uncolored links or red links.
   - Green routes can only use uncolored links or green links.
   - Uncolored routes can only use uncolored links.
2. Output of the rail network must include the color of any colored links
   - Output the links in the same format as they are input
   - E.g. “A 42 B red” for a red link
   - E.g. “B 13 C green” for a green link
   - E.g. “A 5 D” for an uncolored link
3. The program should handle invalid input in a user-friendly way:
   - If the input is invalid the software should output “Invalid line: “ (without the quotes), followed by the invalid line of input.
   - E.g. “Invalid line: A B blue”
   - The software does **not** need to read any more input if a line is invalid
   - Only the first invalid line needs to be indicated
4. Invalid input includes:
   - Too many tokens on one line
   - Too few tokens on one line

- A player color other than red or green
- A non-integer value in place of a distance

Specification: Nonfunctional Changes

1. The design should follow the SOLID principles
2. The customer has informed us that more than 2 players and different kinds of links will be added in the near future, so the design should reflect this.