Homework #2: Polymorphism, Encapsulation, and Testing

Due Thursday, January 29th at 11:59 p.m.

In this assignment, you will implement two different graph representations implementing the same interface. You will then use your graph interface to represent transit data and write a route-planning system that allows a bus rider to find an efficient route (via a sequence of buses) between stops in Pittsburgh's transit system.

Your goals for this assignment are to:

- Understand and apply the concepts of polymorphism, information hiding, and writing contracts, including an appropriate use of Java interfaces.
- Interpret, design, and implement software based on informal specifications and an UML class diagram, demonstrating an understanding of basic software design principles.
- Write unit tests and automating builds and tests using JUnit, Gradle and Travis-CI.
- Understand the benefits and limitations of code coverage metrics and interpret the results of coverage metrics.
- Demonstrate good Java coding and testing practices and style.

This document continues by describing the graph data structures you will implement, the transit planner, and your testing requirements.

Using polymorphism for representational abstraction

You will write two different implementations of a graph interface. You have a lot of design flexibility and can design the interfaces and internals of your graph implementation. You will subsequently use this graph structure to represent transit data which a transit planner uses to find efficient routes between bus stops in the transit system.

You should specify the graph interface and describe all contracts using Javadoc comments. In addition, you should provide two distinct implementations based on the following graph representations:

- **Adjacency list:** Your first implementation must internally represent the graph as an adjacency list. If you are not familiar with the adjacency list representation of graphs, see the Wikipedia page on the adjacency list representation as a reference.

- **Adjacency matrix:** Your second implementation must internally represent the graph as an adjacency matrix. If you are not familiar with the adjacency matrix representation of graphs, see the Wikipedia page on the adjacency matrix representation as a reference.

The transit route planner

After you have completed your graph implementations (and tested them, see below), you should use your graph classes to implement the transit route planner. We have described the design of the route planner in a UML class diagram in Appendix A, and your implementation should be based on our design. All classes and interfaces from the UML class diagram should be located in the `edu.cmu.cs.cs214.hw2` package.

Implementing the route planner consists of four related tasks:

1. Developing a suitable representation of the transit data using your graph data structures and any supplemental data structures necessary for efficient implementation of the `RoutePlanner`'s methods.
2. Construction of a `RoutePlanner` from files containing transit data.
3. Implementation of the `RoutePlanner`'s algorithms using the graph's interface and any supplemental data structures you constructed above.
4. Implementation of a simple front-end (`main` method) that allows a user to select source and destination stops and a departure time, and finds the most efficient (earliest arrival time) route to that destination leaving at or after the departure time.

We describe each of these tasks below.

Using your graph to represent transit data

There are many different representations to store transit data in graphs. The following describes a recommended and common representation.

A vertex in the transit graph typically represents not just a stop but what the transit industry calls a *stop-time*: the location of a bus stop **and** the time of some event (such as a bus arrival/departure) at that location. A bus trip in a transit schedule is typically represented as a sequence of stop-times for that bus.

An edge in the transit graph typically represents a directed path from one stop-time to another. Your transit graph should model two mechanisms by which bus riders can travel

between stop-times: (1) a rider may ride a bus from one stop-time to another, or (2) a rider can wait at a bus stop for up to 20 minutes (1200 seconds) until another bus arrives. Your route planner does not need to consider paths that require a rider to wait more than 20 minutes for a bus or to walk from one bus stop to another. The weight (cost/duration) of an edge is the length of time in seconds from the source stop-time to the destination stop-time along that edge.

Constructing the `RoutePlanner`

To construct an object implementing the `RoutePlanner` interface, you must construct (and populate with appropriate vertices and edges) a graph and any other data structures needed by the `RoutePlanner`. To decouple the `RoutePlanner` from the transit data format, you should use the *Builder design pattern* so that the `RoutePlanner`'s constructor does not need to directly parse and interpret the transit data. To accomplish this, the `RoutePlannerBuilder` provides an interface to build `RoutePlanner`s, but that builder interface may be implemented in different ways. The builder should parse the transit data file and construct any data structures needed by the `RoutePlanner`, passing those data structures as arguments to a class implementing `RoutePlanner` with a trivial constructor that simply stores the data structures. The builder decides which concrete graph representation to use. The corresponding interfaces are described in the UML diagram.

We have provided several transit data files that simply list, for each bus trip, the stop-times for that trip in a CSV-like file format. You must read and parse this file to construct the `RoutePlanner`'s data structures. The file format is described in more detail in Appendix B.

Implementing the `RoutePlanner` algorithms

The `RoutePlanner` interface consists of two required methods:

- `findStopsBySubstring(search)`: Returns a list of all stops whose name contains the provided substring, `search`.
- `computeRoute(src, dest, time)`: Returns an `Itinerary` object describing a trip that minimizes the arrival time at `dest`. The itinerary is composed of trip segments as described in the UML diagram. If there exists more than one path with the same arrival time, any such path may be returned.

You may implement `findStopsBySubstring` using any reasonable algorithm; even with Pittsburgh's transit system containing thousands of stops, it is trivial to find the stops matching the search string in a reasonable amount of time.

For `computeRoute` we strongly recommend that you use Dijkstra's algorithm to find the

shortest path, in time, from the source stop to the destination stop. Although you must develop your own Java implementation of Dijkstra's algorithm, you may base your Dijkstra implementation on any reasonable pseudo-code such as from the Wikipedia article. Be sure to properly cite any external resources you use, as required in the course collaboration policy.

The route planner must work with all graph implementations fulfilling the graph interface's contracts, especially your two concrete implementations.

User interaction with the `RoutePlanner`

Your solution should include a simple user interface (`main` method) that allows a user to interact with the route planner. Your main method should construct the route planner and allow a user to search for efficient routes. We will provide a simple interface in a Piazza post that you can use. You are welcome to develop (and share with the class through Piazza) other front-ends, as long as they rely only on the interfaces specified in the UML diagram. If you want to share your user interface on Piazza, please share it as a private post; the course staff will review your interface and make the post public if it does not reveal implementation details of the rest of your solution.

Testing your implementation

Write unit tests for your homework solution. Your unit tests should:

- Check the correctness of normal cases as well as edge cases of your algorithms.
- Achieve 100% or nearly 100% line coverage of all code *excluding test code, code to load data from files, and the user interface code.* Depending on your solution, full coverage may not be achievable. If you cannot achieve 100% coverage, explain with a short comment in the uncovered area why you can't achieve 100% coverage.

We recommend that you start writing tests for your solution as you complete your solution; do not delay writing unit tests until after your implementation is complete. It is far easier to test (and find any bugs in) your graph implementations and algorithms before implementing algorithms that use your graph implementations. A common strategy is to write your unit tests as you write your code, or to even write your unit tests before you write your code, as you design your software specification.

Apply best practices to writing unit tests. This includes writing many small, independent tests instead few long-running complex ones. The source code for unit tests is often organized in a separate `src/test/java` directory within the software project. By mirroring the

directory structure of the `src/main/java` directory, the `src/test/java` directory allows you to place unit tests in the same Java package as the project source code, without placing the test source code in the same directory as the project source.

Evaluation

Overall this homework is worth 110 points. To earn full credit you must:

- Design your abstractions to maximize information hiding. Prefer interface to class types. Encapsulate your implementation using the most restrictive access level that makes sense for each of your fields and methods (i.e. use `private` unless you have a good reason not to). Instead of manipulating class fields directly, make them `private` and implement getter and setter methods to manipulate them from outside of the class. See Controlling Access to Members of a Class for a reference. Do not use `instanceof` and avoid `static` methods.
- Achieve 100% line coverage of your solution with unit tests. We will use the Cobertura to assess coverage of all implementations except (1) testing code itself, (2) functionality loading transit data from a file, and (3) the user interface.
- Apply best practices to writing unit tests.
- Implement the part of the solution specified in the UML diagram corresponding to the specification. You should follow all names and structures provided, but you may add additional methods and fields. Place all implementations of structures in the UML diagram in the `edu.cmu.cs.cs214.hw2` package.
- Make sure your code is readable. Use proper indentation and whitespace, abide by standard Java naming conventions, and add additional comments as necessary to document your code. You should follow the Java code conventions, especially for naming and commenting. Hint: use **Ctrl + Shift + F** to auto-format your code!

Additional hints:

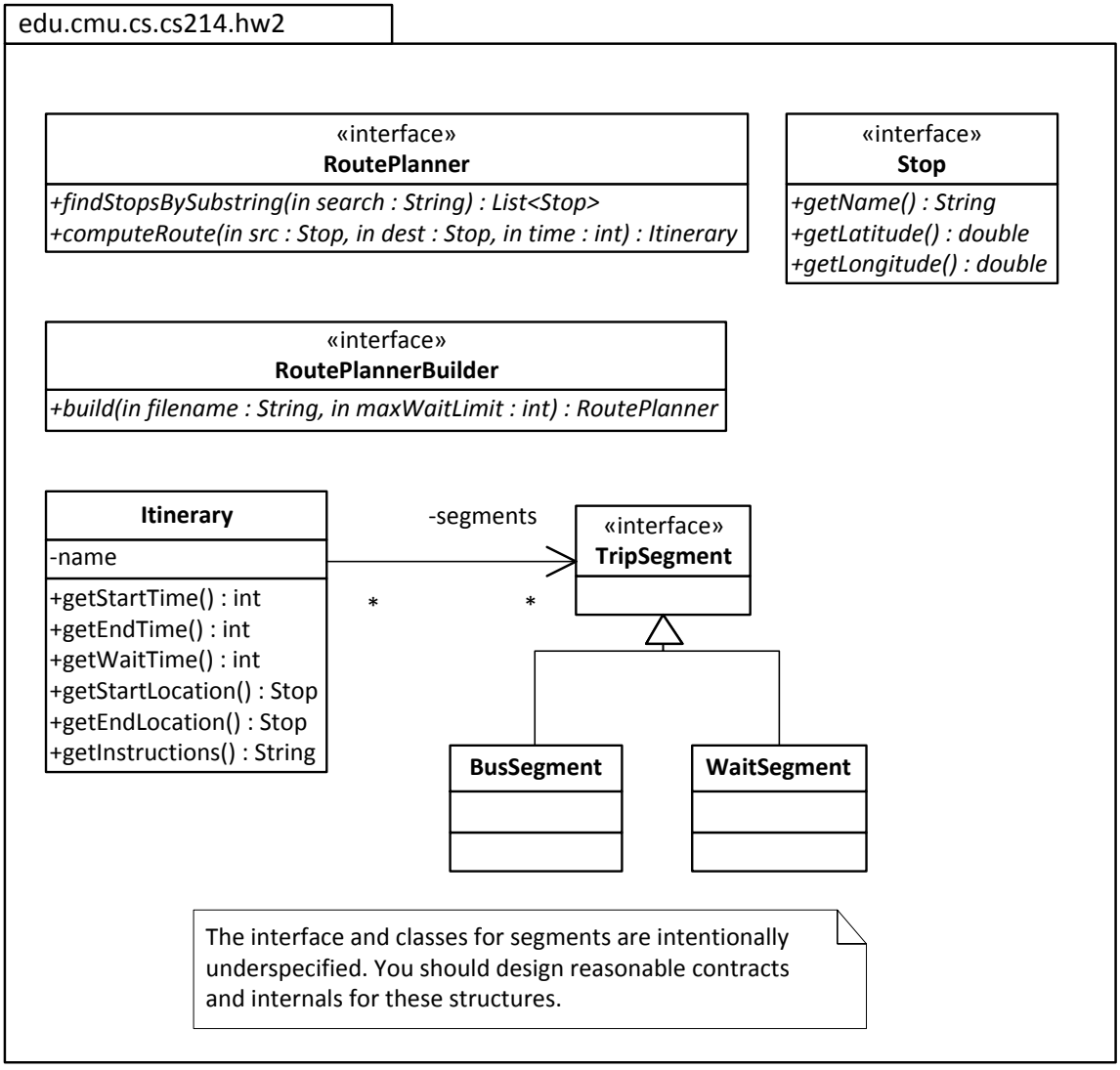
- You may create helper classes, helper methods, and additional data fields to help you with the assignment, as long as your code implements and uses the features of the provided UML class diagrams.
- The tasks and the UML class diagrams are intentionally underspecified in some circumstances. In case of doubt, use your judgment or ask a question on Piazza. If you want to communicate your assumptions, use comments in the source code.

- You may reuse or modify your code from homework 1. If you reuse code, please copy your homework 1 code into your homework 2 directory. Do not import your homework 1 files or classes in homework 2.
- You are not required to throw any exceptions on faulty input arguments for this assignment. Your implementation may behave arbitrarily on incorrect input. You do not have write tests for handling incorrect inputs. You can assume that when we grade your code, our method arguments will conform to the specifications described in this handout.
- As long as your code runs in a reasonable amount of time, and returns the correct values, you do not need to worry about the exact complexity of your algorithms. We will test your code on a variety of inputs, including (possibly) the full schedule for the Pittsburgh transit system.
- Unit tests should be written using JUnit. JUnit is a unit testing framework for Java that is intended to make it easy to implement repeatable tests. Eclipse comes with JUnit installed by default. You can create a new unit test by clicking File → New → JUnit Test Case. You can find additional documentation and starter cookbooks for using JUnit at `http://junit.sourceforge.net/`.
- You do not need to modify the `/.travis.yml` and `gradle.build` files, but you may perform reasonable changes. Do not however change `/settings.gradle` or the 60 seconds time limit in `/.travis.yml`. We enforce a timeout of 60 seconds for all builds to balance Travis-CI capacity for all students.

We will grade your work approximately as follows:

- Correctly applying the concepts of polymorphism and information hiding: 30 points
- Compatibility of your solution with the specification of our UML class diagram: 10 points
- Compatibility of your solution with our informal specification: 30 points
- Unit testing, including adequate coverage and compliance with best practices: 30 points
- Documentation and style: 10 points

Appendix A: Partial UML Class Diagram for a Transit Route Planner



Appendix B: Transit Data File Format

On Piazza we will make available several files containing schedule data from the Port Authority of Allegheny County, which operates the transit system in Pittsburgh. You should download and copy these files to your `homework/2/src/main/resources` directory. The largest file (`all_stop_times.txt`, 40 MB) contains the full transit schedule, while the other files contain smaller subsets of the schedule using just bus routes that pass near Carnegie Mellon.

The file format is a CSV (comma-separated value) variant with blocks of data corresponding to the bus trips in the transit schedule. Here is a snippet of the file:

```
61C,83
5TH AVE AT WOOD ST,40.440785,-80.000814,25620
5TH AVE AT SMITHFIELD ST,40.440027,-79.998816,25680
5TH AVE AT ROSS ST,40.438923,-79.995749,25800
... (80 more lines) ...
61D,54
5TH AVE AT WOOD ST,40.440785,-80.000814,69720
5TH AVE AT SMITHFIELD ST,40.440027,-79.998816,69780
5TH AVE AT ROSS ST,40.438923,-79.995749,69900
... (51 more lines) ...
61C,88
MCKSPT TRANSPORTATION CTR AT BAY #2,40.352264,-79.860522,30540
LYSLE BLVD OPP MCKEESPORT TRANSPORTATION CENTER,40.351798,-79.860919,30600
LYSLE BLVD AT SENIOR CARE PLAZA,40.351967,-79.858497,30600
... (85 more lines) ...
```

The first line of each block contains the route name (e.g. 61C) for a bus trip and the number of stops (e.g. 83) the bus makes for that trip. The rest of the block consists of one line per bus stop, with each line containing the stop's name, latitude, longitude, and the time (in seconds since midnight) the bus visits the stop along the trip. Because each block corresponds to only a single bus trip and each route is traveled by many buses each day, each route will have many blocks in the file.