CSCI 2600 Principles of Software

Homework 7: Model-View-Controller and RPI Campus Paths

Due: Friday, August 21 @ 11:59:59 pm

Introduction

You will be building a route-finding tool. It will take the names or ids of two buildings on the RPI Troy Campus, and generate directions for the shortest route between them, using your graph ADT to represent buildings and pathways between them. For now you will provide a simple text interface. In this homework, you will write a complete application runnable from the command line via a `main` method. You will have to name your main class `CampusPaths.java` for testing on the Homework Server.

In this assignment, you will practice modular design, writing code for reuse, and design patterns. As before, you get to choose what classes to write and what data and methods each should have. Specifically, you will practice the model-view-controller design pattern.

For organization, this assignment contains one "problem" for each logical component you will write. The order of the problems is not meant to suggest an order of implementation. **Carefully design** the whole system before attempting to implement any part. Design with "low coupling" and the "open/closed principle" in mind: you should be able to extend your system, while at the same time the heart of your system (your model) remains insulated from the changes. As always, you should develop incrementally, which may mean repeatedly writing a bit of all the parts and verifying that they work together.

Model-View-Controller

You will design your application according to the model-view-controller (MVC) design pattern described below. MVC is a variation of the Observer design pattern discussed in class.

As you design and implement your solution, please list which parts of your code belong to the model, the view, the controller, or none of the above in `answers/mvc.pdf`. Often this can be on a per-class level, but when a class implements both the view and controller, you must indicate which methods or lines logically represent the view and which represent the controller. *Be sure to list ALL classes you write for Homework 7.* This just should be a list of classes; you don't need to write any sentences of explanation.

The Three Pieces: Model, View, Controller

- The **model** consists of the classes that represent data, as well as the classes that load, store, look up, or operate on data. These classes know nothing about what information is displayed to the user and how it is formatted. Rather, the model exposes observer methods the view can call to get the information it needs.

  In general, functionality of a model includes:

  - Reading data from the data source (text file, database, etc.) to an in-memory representation.
  - Storing data while the program is running.
  - Providing methods for the view to access data.
  - Performing computations or operations involving the data and returning the result.

  - Updating the in-memory state (if the application allows the user to modify data).
  - Writing to the data source (text file, database, etc.)

- The **view** implements the user interface. It should store as little data and perform as little computation as possible; instead, it should rely on the model for data storage and manipulation. The view decides how the user sees and interacts with these data.

  Does the user interact with a text interface or a GUI? What does the user type and/or click to get directions from one building to another? How are those directions formatted for display? What message does the user see upon requesting directions to an unknown building? These are questions the view answers.

- The **controller** listens to user input. Based on the user's keystrokes, mouse clicks, etc., the controller determines their intentions and dispatches to the appropriate methods in the model or view.

  For a simple interface like in this assignment, the view and controller may be somewhat intermingled in code. Don't worry too much about the separation there; the key point for now is that the model is cleanly separated and reusable.

Model-View Interaction

In general, avoid the temptation to create an oversized "god class" that does everything for the model. The model may contain multiple classes, and the view can interact with multiple classes in the model. Most of the time, any class that exists solely to represent data is part of the model. For this assignment, you will likely choose to have one central model class that manages the graph and does most of the heavy lifting, but you may also want some smaller objects that encapsulate related data. Some of these objects might be returned to the view so it can access their data directly, avoiding the "god class" scenario; others might be used only internally within the model.

Your model should be completely independent of the view (UI), which means it shouldn't know or decide how data is displayed. The model does know something about what data and operations the application needs, and it should provide methods to access them; however, it shouldn't return strings tailored to a particular view and definitely shouldn't contain `println`s. Imagine replacing your text UI with a GUI or a Spanish/Mandarin/Klingon text UI (but with the same English building names) and ask yourself: is my model reusable with this new view?

On the flip side, the view doesn't know anything about how the model stores data internally. Someone writing a view for your application should only know that the model somehow stores buildings and paths on campus and should only interact with the data at this level. In other words, the public interface of the model should give no indication that these data are represented internally as a graph. That level of detail is irrelevant to the view, and revealing it publicly means the model can no longer change its implementation without potentially breaking view code.

Problem 1: Parsing the Data

We have added two .csv data files to the `data` folder to be parsed by your application: `RPI_map_data_Nodes.csv` and `RPI_map_data_Edges.csv`. The `.csv` files are comma-separated value files that can be opened in any text editor, just like `marvel.csv`. Their format is described in more detail below.

As usual, your program should look for files using filenames in your data folder e.g., `data/RPI_map_data_Edges.csv`.

You will write a parser to load the data from these files into memory. You may use `MarvelParser.java` as a general example of how to read and parse a file, keeping in mind that the new data files are structured differently from the Marvel data file.

The file `RPI_map_data_Nodes.csv` lists all buildings on campus along with their pixel coordinates on the campus map. (The image of the campus map can be downloaded here or a larger version from here). File `RPI_map_data_Nodes.csv`

has two parts. The first part lists all the buildings on campus, where each line has four comma separated fields:

```
Name,id,x-coordinate,y-coordinate
```

where *Name* is the full name of the building, *id* is the building id, and *x-coordinate* and *y-coordinate* are the coordinates on the map. There may be spaces in the building names.

The second part of file `RPI_map_data_Nodes.csv` shows the intersections on the map. The intersections have no name, hence the name field is empty.

The file `RPI_map_data_Edges.csv` lists pairs of building and intersection ids:

```
id1,id2
```

which means that there is a pathway between the building or intersection denoted by id1 and the one denoted by id2. Pathways are bi-directional: *id1*,*id2* means that there is a path from id1 to id2 and also from id2 to id1.

Your task is to parse these two files and build a graph that represents the RPI campus map. For this assignment, you will compute the length of a pathway from the coordinates of the end points by applying the Euclidean distance formula.

Problem 2: The Model

As described above, the model handles the data and contains the major logic and computation of your program. For this assignment, a key function of the model will be finding the shortest route between two building on campus. This can be accomplished by using Dijkstra's algorithm to find a shortest path in the graph, where edge weights represent the pathway length. Reuse the Dijkstra method by calling the hw6 code. Do not copy and paste your Dijkstra code from hw6 into this homework. Do not re-implement the Graph class in your hw7 code. Reuse the Graph class from hw4. Do not call GraphWrapper from your hw7 code, use your Graph class. If you make changes to code from a previous homework, be aware that the code must still compile.

Problem 3: The Controller and View

In this homework, you will write a simple text interface. When the program is launched through the `main` method, it repeatedly prompts the user for one of the following one-character commands:

- *b* lists all buildings (only buildings) in the form `name,id` in lexicographic (alphabetical) order of name.
- *r* prompts the user for the ids or names of two buildings (only buildings!) and prints directions for the shortest route between them.
- *q* quits the program. (Note: this command should simply cause the main method to return. Calling `System.exit` to terminate the program will break the tests.)
- *m* prints a menu of all commands. Feel free to add functionality. Our tests cover only the functionality specified above.

When an unknown command is received the program prints the line

```
Unknown option
```

Route directions start with

```
Path from Name 1 to Name 2:
```

where *Name 1* and *Name 2* are the full names of the two buildings specified by the user. Route directions are then printed with one line per pathway. Each line is indented with a single tab and reads:

```
    Walk direction to (Name 3)
```

where *direction* is the direction of the pathway and *Name 3* is the name of the building at the pathway destination. If the pathway destination is an intersection, print:

```
Walk direction to (Intersection id)
```

Direction should be one of North, NorthEast, East, SouthEast, South, SouthWest, West, and NorthWest. To determine the direction, compute the angle of the destination from direction north clockwise (e.g, angle of 90 degrees is East). This angle falls into one of eight 45 degree circle sectors corresponding to North, NorthEast, East, etc. For example, if the angle is in the sector [22.5, 67.5), the direction is NorthEast. Finally, print the total distance in pixel units:

```
Total distance: x pixel units.
```

where *x* is the sum of the (non-rounded) distances of the individual route pathways.

The total distance should be rounded to three digits after the decimal point. As in Homework 6, we recommend the use of format strings.

Finally, if one of the two buildings in a route is not in the dataset, the program prints the line:

```
Unknown building: [Name]
```

If neither building is in the dataset, the program prints the line twice, once for the first building and then for the second one. If there is no route between two buildings, your program should print:

```
There is no path from Name 1 to Name 2.
```

To help with formatting, we have provided the output from a sample run of our solution in the file `data/sample_output.txt`. This file reflects the exact appearance of the console window after running the program, and includes both user input and program output. If you run your program with the user input in the file, the state of the console should match the file contents exactly (including whitespace). The sample file and the descriptions above, taken together, should completely specify the output format of your program.

Problem 4: Testing Your Solution

Unlike in previous assignments, the specification is based solely on the output of the complete application, as invoked through the `main` method. This means that your JUnit tests will be different from previous homework assignments.

We provide class `CampusPathsTest.java`. The `runTest` method takes as a parameter the name of the test files, then constructs two file names, one for input and one for output. It temporarily points `System.in` and `System.out` to these files while it runs your main program. The result is that commands are read from the input file and output is printed to the output file. For your tests to run, you simply need to add `@Test` methods that call `runTest` with the appropriate argument. Also, you might have to edit the designated line in `runTest` to invoke your main method.

You will specify the commands for your tests to run in `*.test` files. These files simply contain the series of input a user would have entered at the command line as the program was running. For each test, a corresponding `.expected` file should contain the output your program is expected to print if a user entered that input. Use JUnit to run the tests. `runTest` compares the output in your `.out` file against the corresponding `.expected` file.

Reminder: if a test fails, it is often helpful to look at the `.out` file. These files are written to the `data` directory. It may be helpful to navigate through the file system at the command line rather than in Eclipse.

We have provided one example test in `data`. Note that the `.expected` file only contains newlines printed by the program using `System.out.println`.

It is important that your test data, i.e., `*.test` and `*.expected` files, are in directory `data` and not in directory `src/test/java/hw7/`. If you place your test data in `test` the Homework Server won't grab the files and will produce

FileNotFound Exceptions.

Additionally, you should write JUnit tests for *every* class that is not part of the view or controller. You may not need to write tests for the view and controller. One reason is that they should have very little functionality — they act as glue between the UI (which is hard to test programmatically) and the model. Furthermore, end-to-end behavior of your application is tested through the specification tests. You may write additional tests for your view and controller if you feel there are important cases not covered by your specification tests, but avoid creating unnecessary work for yourself by duplicating tests. You must have at least three different path tests.

Reflection [0.5 points]

Please answer the following questions in a file named `reflection.pdf` in your `answers/` directory. Answer briefly, but in enough detail to help you improve your own practice via introspection and to enable the course staff to improve Principles of Software in the future.

1. In retrospect, what could you have done better to reduce the time you spent solving this assignment?
2. What could the CSCI 2600 staff have done better to improve your learning experience in this assignment?
3. What do you know now that you wish you had known before beginning the assignment?

Collaboration [0.5 points]

Please answer the following questions in a file named `collaboration.pdf` in your `answers/` directory.

The standard academic integrity policy applies to this assignment.

State whether or not you collaborated with other students. If you did collaborate with other students, state their names and a brief description of how you collaborated.

Grade Breakdown

- Quality of test suite, percent of your tests passed: 5 pts (auto-graded)
- Quality of test suite, percent coverage: 5 pts (auto-graded)
- Instructor tests: 16 pts (auto-graded)
- Answers to MVC questions (`answers/mvc.pdf`): 5 pts
- Test data quality (`data/*.test` and `*.expected`): 3 pts
- Code quality (`src/main/java/hw7/*.java`, Principles of Software specs, implementation of Observer/MVC and code reuse): 15 pts
- Collaboration and reflection: 1 pt

Hints

Best Coding Practices

When designing classes, keep the single responsibility principle in mind. Avoid huge "god" classes.

Remember to practice good procedural decomposition: each method should be short and represent a single logical operation or common task. In particular, it can be tempting to implement your entire view and controller as one long method, but strive to keep each method short by factoring operations into small helper methods.

Store your data in appropriate types/classes. In particular, you should *not* pack together data into a `String` and then later parse the `String` to extract the components.

Remember that your graph should be completely hidden within the model. Classes that depend on the model (namely, the view and controller) should have no idea that the data is stored in a graph, not even from the class documentation. If you decided later to switch to a different graph ADT or to do away with the graph altogether (for example, by making calls to the Google Maps API to find paths), you want to be able to change the model without affecting the view and controller, whose job has nothing to do with how the data is stored or computed.

As usual, include an abstraction function, representation invariant, and checkRep in all classes that represent an ADT. If a class does not represent an ADT, place a comment that explicitly says so where the AF and RI would normally go. (For example, classes that contain only static methods and are never constructed usually do not represent an ADT.) You very well may find that you have more non-ADT classes on this assignment than in the past. Please come to office hours if you feel unsure about what counts as an ADT and what doesn't.

Common Issues

Do not call `System.exit` to terminate your program, as it will prevent your specification tests from passing.

If you use `Scanner` to read user input from `System.in`, be sure not to call both `next()` and `nextLine()` on the same `Scanner` object, as the `Scanner` may misbehave. In particular, some students have found that it causes their programs to work correctly at the console but not when they run their tests. Using `Scanner` is neither necessary nor required.

Floating-point precision and numeric comparisons. If you do arithmetic over floating-point values (`float`, `double`), then an exact == may not work as expected. Thus, when comparing computed floating-point values, you should use an approximate comparison, such as that the ratio between the values is very close to 1. However, in this assignment you may use == if you are comparing exact floating-point values that are read from a file, without doing arithmetic over them. Doing an approximate comparison is even wrong, since it would give someone reading the code the impression that you are computing approximate values.

The origin of the coordinate system is the upper left corner of the campus map.

What to Turn In

You should add and commit and push the following files to Submitty. Don't forget to click "Grade My Repository" button on Submitty!

- `src/main/java/hw7/*.java`
- `data/*.test`
- `data/*.expected`
- `src/test/java/hw7/*Test.java` *[JUnit test classes you edit or create]*
- `answers/mvc.pdf`
- `answers/hw7_reflection.pdf`
- `answers/hw7_collaboration.pdf`

Errata

Check the Submitty Discussion Forum for possible errata or other relevant information.

Q & A

None yet.

Parts of this homework were copied from the University of Washington Software Design and Implementation class by Michael Ernst.