CSE 331 Software Design and Implementation

Homework 7: ***Model-View-Controller and Campus Paths***
Due: TODO @ **11pm**

Handout H7

Contents:

- Introduction
- Model-View-Controller [8 points]
  - The Three Pieces: Model, View, Controller
  - Model-View Interaction
- Problem 1: Parsing the Data [55 points with Problems 2 & 3]
- Problem 2: The Model [55 points with Problems 1 & 3]
- Problem 3: The Controller and View [55 points with Problems 1 & 2]
- Problem 4: Testing Your Solution [10 points]
- Reflection [1 point]
- Collaboration [1 point]
- Time Spent
- Nullness Checker [0 points]
- Returnin [25 points]
- Hints
  - A warning about generics and Eclipse
  - Best Coding Practices
  - Common Issues
- What to Turn In
- Errata
- Q & A

Introduction

It is sometimes difficult to navigate the 643 acres of the UW campus. To address this problem, UW Marketing has commissioned you to build a route-finding tool. It will take the names of two buildings and generate directions for the shortest walking route between them, using your graph ADT to represent buildings and pathways on campus. For now you will provide a simple text interface, and on the next homework you will write a graphical user interface (GUI). Unlike in previous assignments, you will write a complete application runnable from the command line via a `main` method.

In this assignment, you will continue to practice modular design and writing code for reuse. As before, you get to choose what classes to write and what data and methods each should have. You will also get experience using the model-view-controller design pattern, and in Homework 8 you will reuse your model with a more sophisticated view and controller.

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. You will certainly want to design the whole system before attempting to implement any part. As always, you should also develop incrementally, which may mean repeatedly writing a bit of all the parts and verifying that they work together.

Model-View-Controller [8 points]

You will design your application according to the model-view-controller (MVC) design pattern described in lecture and below.

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 `hw7/answers/mvc.txt`. 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 this data.

  Does the user interact with a text interface or a GUI? What does the user type and/or click to get directions from CSE to Odegaard? 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 his/her 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 intermingled somewhat 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 will likely 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, a client (such as 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 this data is 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 [55 points with Problems 2 & 3]

We have added four files to `src/hw7/data`: two plain-text data files to be parsed by your application, `campus_buildings.dat` and `campus_paths.dat`, and two images depicting the data from these files, `campus_map.jpg` and `campus_routes.jpg`. The `.dat` files are plain-text files that can be opened in any text editor, like `marvel.tsv`. Their format is described in more detail below. The image `campus_map.jpg` is a standard map of campus, while `campus_routes.jpg` is an image with the same proportions but containing only lines for walking paths and labels for some buildings. You will not directly use these images in your program; they are provided as an optional reference to help you understand what the data represents and verify your path output.

As usual, your program should look for files using relative filenames starting with `src`, such as `src/hw7/data/campus_buildings.dat`. (If you are running your program from the command line rather than from Eclipse, recall that Homework 5 has instructions on how to launch your program so that it finds files at this path.)

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 campus data files are structured differently from the Marvel data file.

The file `campus_buildings.dat` lists all known buildings on campus. Each line has four tab-separated fields:

```
shortName	longName	x	y
```

where *shortName* is the abbreviated name of a building, *longName* is the full name of a building, and *(x,y)* is the location of the building's entrance in pixels on `campus_map.jpg`. Some buildings have two entrances, which are listed as separate entries with unique abbreviated and full names. There may be spaces in the building names (both long and short versions).

The file `campus_paths.dat` defines straight-line segments of walking paths. For each endpoint of a path segment, there is a line in the file listing the pixel coordinates of that point in `campus_map.jpg`, followed by a tab-indented line for each endpoint to which it is connected with a path segment. Each indented line lists the coordinates of the other endpoint and the distance of the segment between them in feet. Thus, the structure is as follows:

$$
\begin{array}{l}
x_1,y_1 \\
\qquad x_2,y_2\text{: distance}_{12} \\
\qquad x_3,y_3\text{: distance}_{13} \\
\qquad \ldots \\
\ldots \\
x_i,y_i \\
\qquad x_j,y_j\text{: distance}_{ij} \\
\qquad \ldots
\end{array}
$$

where *distance*$_{ij}$ is the distance from (x_i,y_i) to (x_j,y_j). There is no relationship among points on consecutive indented lines: they are all connected with a path segment to the point on the previous *non-indented* line, but they are not necessarily connected to each other.

Some endpoints are building entrances and will match the coordinates of an entry in `campus_buildings.txt`, but most are not. The dataset only lists outdoor paths, to ensure (for example) that the paths work even if it's midnight, all the buildings are locked, and you're being chased by a zombie so you really need the shortest path.

Problem 2: The Model [55 points with Problems 1 & 3]

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 walking route between buildings. This can be accomplished by using Dijkstra's algorithm to find a shortest path in the graph, where edge weights represent the physical length of a route segment.

Problem 3: The Controller and View [55 points with Problems 1 & 2]

On 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 in the form `abbreviated name: long name`. Buildings are listed in lexicographic (alphabetical) order of abbreviated name.
- *r* prompts the user for the abbreviated names of two 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 specification tests.)
- *m* prints a menu of all commands.

To support the test scripts you will write later, your program should simply echo empty lines or lines beginning with # to `System.out`. It should not attempt to process these lines as commands and should not reprint the usual prompt for a command after printing the line.

Otherwise, when an unknown command is received the program prints the line

```
Unknown option
```

We have provided the output from a sample run of our solution in the file `hw7/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 below, taken together, should completely specify the output format of your program.

Route directions start with

```
Path from Building 1 to Building 2:
```

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

```
	Walk dist feet direction to (x, y)
```

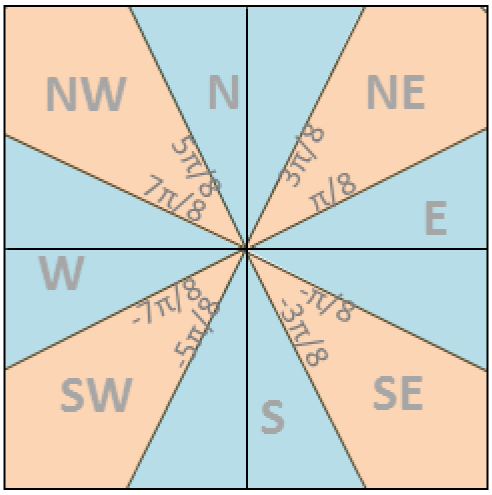
where *dist* is the length of that segment in feet, *direction* is a compass direction, and *(x, y)* are the pixel coordinates of the segment endpoint. Finally, the route directions end with

```
Total distance: x feet
```

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

Distances and coordinates should be rounded to the nearest integer. As in Homework 6, we recommend the use of format strings.

Each compass direction is one of N, E, S, W, NE, SE, NW, or SW and can be determined by comparing the coordinates of the start and end points. Pixel (0,0) is at the top-left corner of the image, and the direction is selected according the eight sectors shown below. Each sector has the same angle. Points that fall exactly on a boundary between sectors should be classified as a single-letter direction (N, E, S, or W). (Hint: the function `Math.atan2` and the constant `Math.PI` are helpful for determining which sector a route segment falls into. When using `Math.atan2`, think very carefully about the directions in which pixels increase as compared to the Cartesian coordinate plane.)



The eight sectors for classifying directions.

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 building. You may assume there is a route between every pair of buildings in the dataset; if not, the behavior is undefined.

For this assignment, the logical view and controller may be part of the same class. In this case you must clarify which parts of the class are the view and which are the controller in `answers/mvc.txt`, as described above. Do so by method - or even by line, if needed, though we recommend against having a single method contain parts of the view and of the controller.

Problem 4: Testing Your Solution [10 points]

As usual, your tests should be organized into specification and implementation tests. 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, in addition to your specification tests, integration tests will be required to fully test your solution [more on this in the final paragraph of this section].

To facilitate your specification tests, we have provided a class `HW7TestDriver` that looks very different from your test drivers for past assignments. The `runTests` method accepts the names of two files, one for input and one for output, and 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 one line to `HW7TestDriver` to invoke your main method.

As usual, 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. Recall from Problem 3 that you can also include blank lines or comment lines (starting with #) for readability and documentation, and your controller should echo these lines to the output. For each test, a corresponding `.expected` file should contain the output your program is expected to print if a user entered that input. As usual, the provided `ScriptFileTests` class will find all the `.test` files in the `hw7/test` directory, use the test driver to run each test (with a 30-second timeout for this assignment), print its output to a `.actual` file, and compare the output against the corresponding `.expected` file.

Reminder: if a test fails, it is often helpful to look at the `.actual` file. These files are written to the `bin/hw7/test` directory under your `cse331` project. This directory is not visible through Eclipse, but you can navigate directly there through the file system or at the command line.

We have provided one example test in `hw7/test`. Note that the `.expected` file only contains newlines printed by the program using `System.out.println`, not newlines that were contained in the input from the `.test` file. So the `.expected` file will look somewhat different from the `hw7/sample_output.txt` file, which does show user input.

Additionally, you should write implementation tests for *every* class that is not part of the view or controller, and add the test class names to `ImplementationTests.java`. You may not need implementation tests for the view and controller. One reason is that they typically 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.

Reflection [1 point]

Please answer the following questions in a file named `reflection.txt` 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 CSE 331 in the future.

a. In retrospect, what could you have done better to reduce the time you spent solving this assignment?
b. What could the CSE 331 staff have done better to improve your learning experience in this assignment?
c. What do you know now that you wish you had known before beginning the assignment?

Collaboration [1 point]

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

The standard collaboration 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.

Time Spent

Tell us how long you spent on this homework via this catalyst survey: https://catalyst.uw.edu/webq/survey/mernst/198079.

Nullness Checker [0 points]

Unlike the previous homework, the code you write for hw7 is not required to be annotated for the Nullness Checker. However a *small* ammount of extra credit will be given to student's whose hw7 code does pass the Nullness Checker. You can run the nullness checker with the `ant check-nullness` target.

Returnin [25 points]

After you receive your corrected assignment back from your TA, you will need to resubmit a corrected version of the code. You will have until Thursday, June 6 at 11pm to returnin your code. The returnin script will be enabled a week earlier, at 11pm Thursday, May 30.

You will only receive credit if you pass all the tests, and you have a fixed number of trials without penalty. See the returnin331 documentation for details.

Hints

A warning about generics and Eclipse

As explained in Homework 6, Eclipse's built-in compiler sometimes handles generics differently from `javac`, the standard command-line compiler. This means code that compiles in Eclipse may *not* compile when we run our grading scripts, leaving us unable to test your code and significantly impacting your grade. You must periodically make sure your code compiles (builds) with `javac` by running `ant build` from the `hw7` directory (which will also build `hw4`, `hw5`, and `hw6` automatically). You can do this in one of two ways:

- **Through Eclipse:** Right-click `build.xml` under `hw6` and click `Run As >> Ant Build...` (note the ellipsis). In the window that appears, select `build [from import ../common.xml]` and hit `Run`.
- **At the command line:** Run `ant build` from the `hw6` directory.

To ensure that you are indeed recompiling your entire program with javac, you may wish to urn `ant clean` before `ant build`; alternatively, just `ant clean build` does both.

Best Coding Practices

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.

What to Turn In

You should add and commit the following files to SVN:

- `hw7/*.java`
- `hw7/test/*.test`
- `hw7/test/*.expected`
- `hw7/test/*.dat`
- `hw7/test/*Test.java` *[Other JUnit test classes you create, if any]*
- `hw7/answers/mvc.txt`
- `hw7/answers/reflection.txt`
- `hw7/answers/collaboration.txt`

Additionally, be sure to commit any updates you make to the following files, as well as any files in `hw4`, `hw5`, or `hw6`:

- `hw7/test/ImplementationTests.java`
- `hw7/test/HW7TestDriver.java`

Finally, remember to run `ant validate` to verify that you have submitted all files and that your code compiles and runs correctly when compiled with `javac` on attu (which sometimes behaves differently from the Eclipse compiler).

Errata

None yet.

Q & A

None yet.

For problems or questions regarding this page, contact: cse331-staff@cs.washington.edu.