Final Project

Introduction

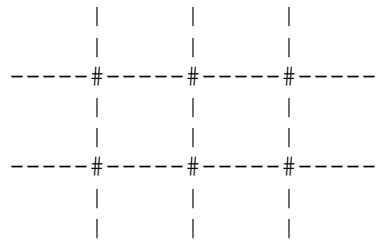
The Final Project gives you the opportunity to apply the object-oriented software development skills you have acquired during the course to a **significant** but **moderately-sized** application. The following sections describe the general behavior of the application, the simulation, and the deliverable requirements for the project.

An example executable is here: file:example-project.jar

Starter code for the project is here: file:myproject.zip

Model Overview

We are modeling a traffic grid with a variable number of rows and columns. For example, a 2x3 grid would look something like this:

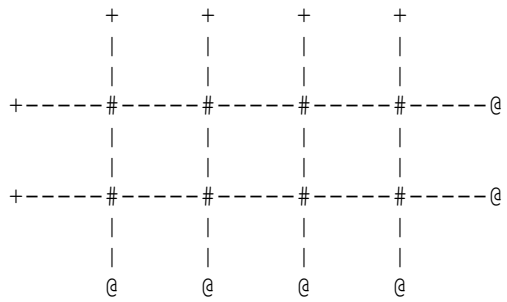


The `#` represent intersections, and the lines represent roads (or road segments).

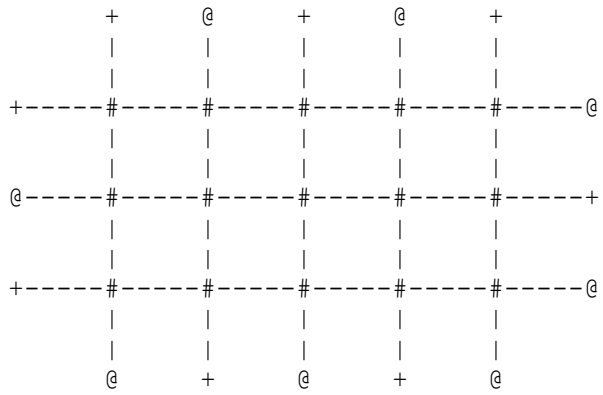
We will assume that each car goes in a consistent direction (north to south, east to west, etc). Cars do not turn. All roads are one-way and have only a single lane of traffic.

We will consider two kinds of traffic patterns: *simple* and *alternating*.

In the simple traffic pattern, all traffic goes in the same direction. In the following picture, `+` represent a car *source* and `@` represents a car *sink*. Cars flow from sources to sinks. Here is a 2x4 simple grid:



In the alternating traffic pattern, roads alternate direction. Here is a 3x5 alternating grid:



Sources generate cars and place them on a road. Cars move in a straight line until it hits the Sink. Sinks then delete cars from the simulation.

Cars

Behavior

The behavior of a car depends up the distance to the nearest obstacle. There are three attributes regulating this behavior, which may vary from car to car.

```
maxVelocity    // The maximum velocity of the car (in meters/second)
brakeDistance  // If distance to nearest obstacle is <= brakeDistance,
               //   then the car will start to slow down (in meters)
stopDistance   // If distance to nearest obstacle is == stopDistance,
               //   then the car will stop (in meters)
```

Cars also have the following attribute, which determines how much space the consume:

```
length  // Length of the car (in meters)
```

As well, they have a Color attribute. This is mostly for identification purposes. You can make it have a random color.

```
private java.awt.Color color = new
java.awt.Color((int)Math.ceil(Math.random()*255),(int)Math.ceil(Math.random()
*255),(int)Math.ceil(Math.random()*255));
```

Car Velocity Calculation

Suppose a car has the following values for the first three attributes.

```
maxVelocity = 30.0
brakeDistance = 10.0
stopDistance = 1.0
```

If the nearest obstacle is 12 meters away, then the car will assume its `maxVelocity`. If the nearest obstacle is 1 meter away, the car will stop. If the nearest obstacle is 5 meters away, the car will consider the distance to the obstacle in computing its velocity. You can use the following formula to compute the next position of the car.

```
double velocity = (maxVelocity / (brakeDistance - stopDistance))
                  * (distanceToObstacle - stopDistance);
velocity = Math.max(0.0, velocity);
velocity = Math.min(maxVelocity, velocity);
nextFrontPosition = frontPosition + velocity * timeStep;
```

Since we do not consider acceleration, you do not need to store the velocity as an attribute.

A student comment: The updateVelocity algorithm allowed cars move at their maximum velocity if there were no obstacles within braking distance. However, it's possible for a slow car to be in front of a fast car, outside the fast cars braking distance but within the fast cars maximum velocity. On the next time step the fast car jumps ahead of the slow car. To clarify, a fast car has position 0, maximum velocity 30, and braking distance 10. A slow car has position 15 and maximum velocity 10. On the next time step the fast car will update its position to 0+30 = 30, leapfrogging the slow car that is now at position 15+10 = 25. To fix this, I changed the updateVelocity algorithm to say that if there's a car outside of our braking distance but inside our max velocity, we adopt velocity of distanceToObject / 2.

Intersections

Behavior

Each intersection has two traffic lights; one for each direction (NS=North/South, EW=East/West). The traffic lights of an intersection are coordinated by a light controller.

Light controllers have four states: GreenNS/RedEW, YellowNS/RedEW, RedNS/GreenEW, RedNS/YellowEW. The rate at which a light controller transitions between these states is determined by two attributes:

```
greenDurationNS  // Duration of the North/South green phase (in seconds)
yellowDurationNS // Duration of the North/South yellow phase (in seconds)
greenDurationEW  // Duration of the East/West green phase (in seconds)
yellowDurationEW // Duration of the East/West yellow phase (in seconds)
```

Light Controller example

Suppose a light controller has the following values for these attributes:

```
greenDurationNS  = 55.0
yellowDurationNS = 5.0
greenDurationEW  = 25.0
yellowDurationEW = 5.0
```

Then the light will make the following transitions:

```
time=0    state=GreenNS/RedEW
time=55   state=YellowNS/RedEW
time=60   state=RedNS/GreenEW
time=85   state=RedNS/YellowEW
time=90   state=GreenNS/RedEW
time=145  state=YellowNS/RedEW
time=150  state=RedNS/GreenEW
time=175  state=RedNS/YellowEW
time=180  state=GreenNS/RedEW
```

Intersection behavior

Intersections also have a length, and therefore may hold cars.

From the point of view of a car, an intersection is an obstacle if any of the following are true:

- The light state in the car's direction is Red.

- The light state in the car's direction is Yellow and the light is at least `brakeDistance` away.
- The light state in the car's direction is Green, but the intersection is occupied by cars travelling in the other direction (gridlock).

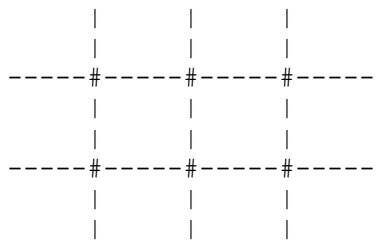
Thus, a car will ignore Yellow lights within its `brakeDistance`.

Simulation Parameters

All spatial units will be given as meters. All time units will be given in seconds. Use the type `double` to store space and time parameters.

The simulation has the following parameters. These are parameters can be modified in the Change Simulation Parameters menu.

- **Simulation time step (seconds)** Default=`[0.1]`. This indicates how much model time elapses between each simulation step. You can use this to adjust the granularity of the simulation.
- **Simulation run time (seconds)** Default=`[1000.0]`. The length of the simulation in model seconds. When the user chooses `Run simulation` from the main menu, this indicates how long the simulation should run.
- **Grid size (number of roads)** Default=`[row=2,column=3]`. This indicates the size of the grid. If there are 2 rows and 3 columns, then the grid looks like this:



- **Traffic pattern** Default=`[alternating]`. This indicates how the direction of roads should vary. The choices are `simple` or `alternating`.
- **Car generation delay (seconds/car)** Default=`[min=2.0,max=25.0]`. Each car *source* generates cars at a fixed rate, but the sources may vary. You should set the delay between car generations by each source to be a random number between the min and max chosen here.
- **Road segment length (meters)** Default=`[min=200.0,max=500.0]`. Each road segment has a fixed length, but the road segment lengths may

vary. You should set the length of each segment to be a random number between the min and max chosen here.

Although road segment lengths vary, they will all appear the same size in the graphical output. Thus, cars will appear to move more slowly on longer segments.

- **Intersection length (meters)** Default=`[min=10.0,max=15.0]`. Each intersection has a fixed length, but the intersection lengths may vary. You should set the length of each intersection to be a random number between the min and max chosen here.
- **Car length (meters)** Default=`[min=5.0,max=10.0]`. Each car has a fixed length, but the car lengths may vary. You should set the length of each car to be a random number between the min and max chosen here.
- **Car maximum velocity (meters/second)** Default=`[min=10.0,max=30.0]`. (You get the idea.)
- **Car stop distance (meters)** Default=`[min=0.5,max=5.0]`. (You get the idea.)
- **Car brake distance (meters)** Default=`[min=9.0,max=10.0]`. (You get the idea.)
- **Traffic light green time (seconds)** Default=`[min=30.0,max=180.0]`. (You get the idea.)
- **Traffic light yellow time (seconds)** Default=`[min=4.0,max=5.0]`. (You get the idea.)

Note that some combinations of simulation parameters may not make sense. For example, if the simulation time step is too large, then cars may disappear before they ever display. You do *not* need to worry about this. Make sure that the simulation works for sensible values.

Program Interaction

There will be two UIs, a Text one and a GUI-based one. The specific UI will be selected via command line arguments: “TEXT”, or “GUI”.

Once the UI is selected, you will create the appropriate AnimatorBuilder, and then run the Simulation. The simulation should run for 1000 seconds (pass in 1000 to the run function as an argument). The Model Parameters will not be set by the user – they will be random numbers as described in the Simulation Parameters section.

Code Requirements

1. The application must be written in Java using the Java2 SDK 1.8 or higher.
2. All input must be text based, using a variation of the classes provided in homework 3. Graphics are used for output only.
3. Only features and capabilities that are part of the Java2 SDK may be used in the application. *No third-party software* such as BlueJay or JBuilder class libraries or COM/CORBA components.
4. You must write unit tests for at least one reasonably sized/complex class (e.g. NOT the settings class), but you are encouraged to write more!!!
5. The application must use at least four different design patterns that we have discussed in class. You will be expected to demonstrate and explain the patterns in your final written description.

Time Recording

Every day that you work on your project, write down the number of hours spent and what was achieved. Record any design problems or solutions you explored.

Keep this electronically. Your name at the top. Start a new paragraph for each day, indicating the date, number of hours spent, followed by comments. This should take no more than a few minutes per day.

Estimate the time you spent in the following activities:

- Design (any activity other than coding and debugging).
- Coding and debugging.
- Dealing with BIG bugs. (If a bug takes more than five minutes to fix, it is worth remembering.)

At the end of each week, compute the total time spent in each of the three areas.

At the end of the project, compute the total time spent in each of the three areas.

Report

Thirty percent of your grade is based is based on a written report. Although no specific style guidelines are being enforced, the report must be presented in a neat, legible, and consistent format.

All text must be typed. Diagrams may be hand-drawn; however, they must be neat, ie, drawn with a ruler. DL students may scan hand-drawn diagrams. The diagrams should conform to the UML notational conventions presented in class.

The written report should be structured as follows:

1. **Title page** with your **name** at the top.
2. **Class diagrams for the Model classes**. Include a design class diagram for the model portion of your project (model as in MVC). This includes classes like Car, TrafficLight, etc. This does NOT include classes like Control, Main, and anything UI-related. Because the Model is separated from your Controller and View, this should be pretty straightforward. When in doubt, ask in D2L. Be sure to include all significant class relationships: realization, specialization, and association. Show associations as dependencies, aggregations or compositions when appropriate. Show attributes and methods *only* if they are crucial to understanding the class relations.

   You may use tools that automatically generate diagrams from your code, but you are responsible for making sure the diagrams they produce are readable.

3. **Sequence Diagram.** Draw a sequence diagram indicating the how a car updates its position. Show all the objects involved.
4. **Time Recording Journal**. As detailed under the Time Recording section above. You do NOT need to include the daily summary of hours spent. This data should be aggregated into the Time Summary (number 5).
5. **Time Summary.** Provide a table breaking down the amount of time (in hours) you spent each week in the three areas. The table should look like this:

```
Week     | 1 | 2 | 3 | 4 | Total
---------|---|---|---|---|------
Design   |   |   |   |   |
Code     |   |   |   |   |
```

```
Bigbug  |   |   |   |   |
```

6. **Notes on patterns**. Indicate the design patterns used in your project. For each pattern, note the specific problem in your project that the pattern solved. Also indicate the classes involved and briefly discuss the implementation of the pattern. These notes should take 1 to 2 pages.
7. **Successes and Failures.** Discuss what went right with your project? What went wrong? Note design issues that arose during development, such as specific decisions, use of design patterns, failures, successes, etc. This should take about 1 page.

Deliverables

1. 2015/11/01: Initial prototype (baseline/high risk). A working program implementing limited functionality.

   Submit a `jar/zip/rar/7z` file of your source code, as for hw 1.

2. 2015/11/08: Release 2.

   Submit a `jar/zip/rar/7z` file of your source code, as for hw 1.

3. 2015/11/15: Release 3.

   Submit a `jar/zip/rar/7z` file of your source code, as for hw 1.

4. 2015/11/24: Final release.

   Submit a `jar/zip/rar/7z` file of your source code, as for hw 1.

   Test your `jar/zip/rar/7z` file by unzipping it into a fresh directory. Make sure everything works. **IF WHAT YOU TURN IN DOESN'T COMPILE, YOU WILL GET A 0% for the code portion of the final project!**

   You will submit your report on D2L. You must submit a *single* file in PDF or DOC format. **Do not submit a ZIP or other archive for your report.** This requires that you somehow get diagrams into the report. Microsoft word and OpenOffice are very good at this. Please use one of them. In the worst case, print out your report and scan it in as a single PDF.

Grading

Grading will follow these guidelines:

- **Application: 70%**
  - Satisfies requirements
  - Correctness (compilation & execution)
  - Quality of code – follow the SOLID principles!
  - Complete and quality unit tests for at least one reasonably-sized/complex class.
- **Report: 30%**

Academic integrity

I do encourage collaboration; however, all submitted work must be your own. If the work was duplicated, it will be reported to the university as an Academic Integrity violation. I don't mind copying and pasting of utility functions or copying of overall designs.

As general rules of thumb, I would say the following guidelines dictate whether sharing is allowed or not:

- It is functionality you would reasonably expect to find on Stack Overflow (like how to format a string output)
- It is some sort of mathematical algorithm (like the car velocity calculation, above). It is an abstract description of how your code is working (like how the car decides whether or not it can enter an intersection, above).

I want to avoid being heavy-handed about this. I would prefer you collaborate than not due to being afraid to violate these rules. When in doubt about whether you can share something or not, send me an email. In 99% of cases where you aren't sure, I will likely say it's acceptable.

Further information and hints

http://fpl.cs.depaul.edu/jriely/450/notes/notes-final-project-012.html

Checklist

Code

- ☐ Compiles

- ☐ Complete unit testing coverage for at LEAST one substantial class (e.g. Car/road/traffic light). Complete would mean coverage for any public/package private method with full test cases (like passing in null, for instance)
- ☐ Ability to run both CUI and GUI by specifying command line switches (see homework 3)
- ☐ Code is of a reasonable quality – Encapsulation is used, SOLID principles are being used, good variable/method names are being used.
- ☐ 4 design patterns are used – CORRECTLY!
- ☐ Cars do not turn
- ☐ Car source and sink for each road
- ☐ Traffic light at each intersection
- ☐ Car dies when it hits end of road
- ☐ Cars do not pass each other
- ☐ Cars stop at yellow lights, if they can. Otherwise, they go through the intersection
- ☐ Cars have random colors
- ☐ Cars stop at red lights
- ☐ Cars stop at intersections when there's gridlock
- ☐ Cars move from stop when the light turns green (assuming no gridlock)
- ☐ Traffic lights have green/yellow/red for both directions (lights for only one direction will be displayed)
- ☐ Traffic lights display green/yellow/red for E/W roads (or N/S roads as long as each light is CONSISTENT)
- ☐ Program runs for 1000 seconds.

REPORT

- ☐ Report typed
- ☐ Diagrams done on computer or drawn NEATLY
- ☐ Title Page with your name and date
- ☐ Class diagrams for model package including SIGNIFICANT class relationships. Aim for specificity in your relationships. Do NOT show attributes/methods unless they are relevant to understanding class relationships
- ☐ Sequence diagram detailing how a car updates its position
- ☐ Time recording journal (dated blurbs about what you worked on that day)
- ☐ Time summary
- ☐ Notes on which patterns you used and the problem they solved (~1-2 pages)
- ☐ Successes and failures (~1 page)