Nicholas Ferrara
2352426

Advanced Programming Assessed Exercise

Assumptions - what have you assumed that is missing in these specs.
I have assumed that:

1. Multiple cars can spawn in the same slot, however, the later have to wait until the former move before they can occupy it.
2. The application should end if no more cars are present.
3. Time statistics can be floored.
4. That cars made with a generator do not have to be integrable into APSpec1.

Class design - description of each class (can be commented UML if you like, plain text if you prefer).

Car.java Stores data about positioning and movement for each of the blips on the grid. The class is its own thread so the movements occur continuously, independent of the grid's refresh rate. It only contains one constructor because there was no need in the current implementation for multiples. It holds a reference to the grid object it is on, so without that reference the class does not work. Its run method is called when the thread is started and then is configured to continue looping until the car navigates off the grid.

CarGenerator.java An easily customizable facility to mass produce cars. Its properties contain the currently foreseeable modifications that one may want to make to a car. Each, with the exception of a grid which must be specified, has a default value which is used to produce cars unless a subclass intervenes to modify that value. The generate method is designed to be called at the conclusion of subclass constructors to undergo the actual creation of the cars. It also includes various setter methods primarily designed to give subclasses a mechanism to change its values before calling the generate method. Finally, it also includes methods relating to time calculations for its cars which is then used by the Statistics class. It has the following subclasses. Although they are organized by direction other implementations could organize them by a different property.

> **EastGenerator.java** An example simple generator for eastbound cars.
>
> **NorthGenerator.java** An example simple generator for northbound cars.
>
> **SouthGenerator.java** An example customized generator for southbound cars.
>
> **SouthGeneratorFast.java** An example simple generator for fast southbound cars.

WestGenerator.java An example simple generator for westbound cars.

Grid.java Organizes the Slot objects and handles graphical output. It is meant to be a unifying class across the application since it holds the slots and is held by cars and simulations. Its most important method is refresh which is what executes the constant screen refreshes and detects the end of the simulation. Print follows which actually calculates the String that refresh flushes to the terminal. It also contains methods for Car to access and manipulate slots in a sensible manner.

Simulation.java This abstract class only exists to allow Grid to call back whichever simulation called it once the simulation ends for a custom ending message. That message is currently the statistical output in APSpec2. It has the following subclasses:

APSpec1.java The original spec which has since been modified to conform to bug fixes and refactoring that occurred in the process of implementing APSpec2. Unlike in APSpec2 this file actually contains the Car generation logic since it does not use generators. In general, the file contains everything one would likely want to change if they wanted to make minor changes to the simulation with the first specification. Most of that control comes in the form of its properties which handle things like refresh rates and the number of cars to be generated.

APSpec2.java The greatly improved version of the class which outsources most of its logic to the car generators. It is mostly reduced to a data structure holding the simulation's configuration. The exact generators implemented are meant to be representative of what one could do with them, but implementing more or changing their parameters would only require changing one line of code.

Slot.java Holds the actual data about a given slot in the grid. Its method SetCar is used to modify the car associated with it, but a lock and condition is used to prevent another car from taking over until the previous one has left. It also includes a reserve method, which is used to prevent double booking a slot when not using a generator.

Statistics.java Handles the tabulation of statistics about car journeys needed for specification 2. It calls on the car generators to directly access their data on their cars.

Velocity.java Holds data pertaining to motion. It is abstract enough that it could foreseeably be used in entirely different applications and is not coupled to any other classes. Instead of passing speed and direction variables through numerous method calls I created this class to abstract the data and simplify the code. It is specifically designed to be clonable so that when it is applied to different cars its exact values can be changed for each car. Since the current application is relatively simple it does not contain many methods, but if more advanced movement were ever to be introduced this would start as a useful foundation to implement that functionality.

Testing - how have you tested your code? The bar is a bit higher than in Programming last semester.

Upon completion I first reviewed all code line by line to ensure no errors were present. I successfully removed a few minor flaws in the process. After that I tested classes independent of their given context to ensure they worked well in their own right. For instance, I extended Velocity and in the process realized I needed to improve its constructors so it would be more usable.

I ran the application numerous times at extremely slow speeds to watch for any amoralities in how the cars moved. I caught one error where a generator's cars were moving significantly faster than expected because of a mathematical error in the code.

Until immediately before submission I kept numerous System.out.println statements for most complicated variables throughout the application which detailed their state at any given time. I would then stop the simulation midway through and examined those values to compare them with their expected result. Once again, a few minor deviations were found, but by the time I completed the code and debugged obvious issues most of the minor problems had also been resolved.

Most importantly, I modified the main classes to provide different input parameters to the application. Those classes are designed to contain fully customizable elements so I put them to the test to make sure they worked as intended.

By far my favorite part of the application to test was the generators. I provided different inputs to each and created others. I still found them to work as expected.

Q1: Critique: one of the key characteristics of good object oriented programming is the ability to make code easily extensible. Describe an extension of the system that your current design could not handle (you may not use the example given in point 3 below)

If the grid were to become three-dimensional a significant reworking of most classes would become necessary. Since coordinates are modified in most classes and given in two-dimensional space and the grid is finally printed in two dimensions a lot of code would have to be modified.

That said, the general logic of the present design would still be salvageable. One way to make it more modular if such an idea were seriously considered would be to refactor coordinates into a coordinates class instead of giving raw numbers. That would be much less cohesive, but would reduce the liability of the coordinates system changing. For instance, cord.modifyM(5) could take the place of mSlot = mSlot + 5. Cord could also be passed to other classes like Grid and Velocity without the method passing it having to modify it. That way any number of new axes could be added without coupling intermediary methods. I considered implementing that in the current application but decided it would be too much additional code without an immediate need.

Q2: As 1., but for an extension that it can handle (you may not use the example given in point 3 below)

The Velocity class is considerably underutilized right now compared to its full potential. It presently supports jumping multiple spaces per iteration. With slight modifications that could be used to allow certain cars to hop over other cars so long as they have a place to land on the other side light a knight in chess.

It would also be easy to create other accessor methods that provide less predictable data, such as by returning slightly randomized changing speeds over time.

Q3: One possible extension is the stringing together of multiple intersections to produce a model of a larger city area. Describe how you would do this, giving details of any changes in your classes that would be required. You do not have to implement these changes.

I would create a master grid class that the other grids would belong to. That class would also include the parts of the present Grid that assume there is only one grid. The main example of this is the refresh method, which draws the entire screen and would not need to be duplicated among the different grids.

If cars were to move between grids having that master grid would also be useful to facilitate cars using the grids interchangeably. Car itself would not need to be changed except to add a setter method for its grid variable. The actual changing grids would take place in Grid.updateCarSlot.

I would also need to modify the CarGenerator class and the APSpec1 class to reflect the new starting points since they would have to pick a grid to start in.

While new code would be necessary most existing code would not need to be modified.

Q4: Reflection: write a short summary of your experience of this project. How did you tackle the problem, how might you do things differently if you did it again?

Although the usefulness of my prior programming experience has been considerably less this semester in compared to last semester I still am able to look at programming problems and see solutions without much effort. I cannot say I followed any conscious design pattern, but instead more so just imitated elements that I have seen better programmers than me use in the past.

The obvious starting point was in APSpec1.java and then later Grid.java and Car.java. Neither Slot.java or Velocity.java originally existed, but I eventually found it useful to decouple their respective features from Grid.java and Car.java respectively. In any future addition to the code both of those classes would probably be significantly expanded to realize many features they could potentially bring.

While implementing level 2 I misread the specification and started implementing CarGenerator.java without its subclasses. The functionality of the subclasses was instead accomplished through numerous different available constructors. At the present scale of the application this actually brought about welcome cohesion, but would not be as extendable as the present solution.

Shortly before finishing I refactored the two main classes to significantly reduce their sizes and move the excess code to Grid.java. At the same time I also created the abstract class Simulation.java to facilitate Grid accessing the main classes. Doing so successfully hides universal logic while only leaving code that future conceivable simulations would likely want to modify.

This project proved surprisingly more challenging than I originally expected. Professor Poet said in Project Management last semester that as a rule of thumb always assume a development project will take twice as long as you think it will. I would have been wise to heed those words, although given my other commitments I am not entirely sure how I could have.