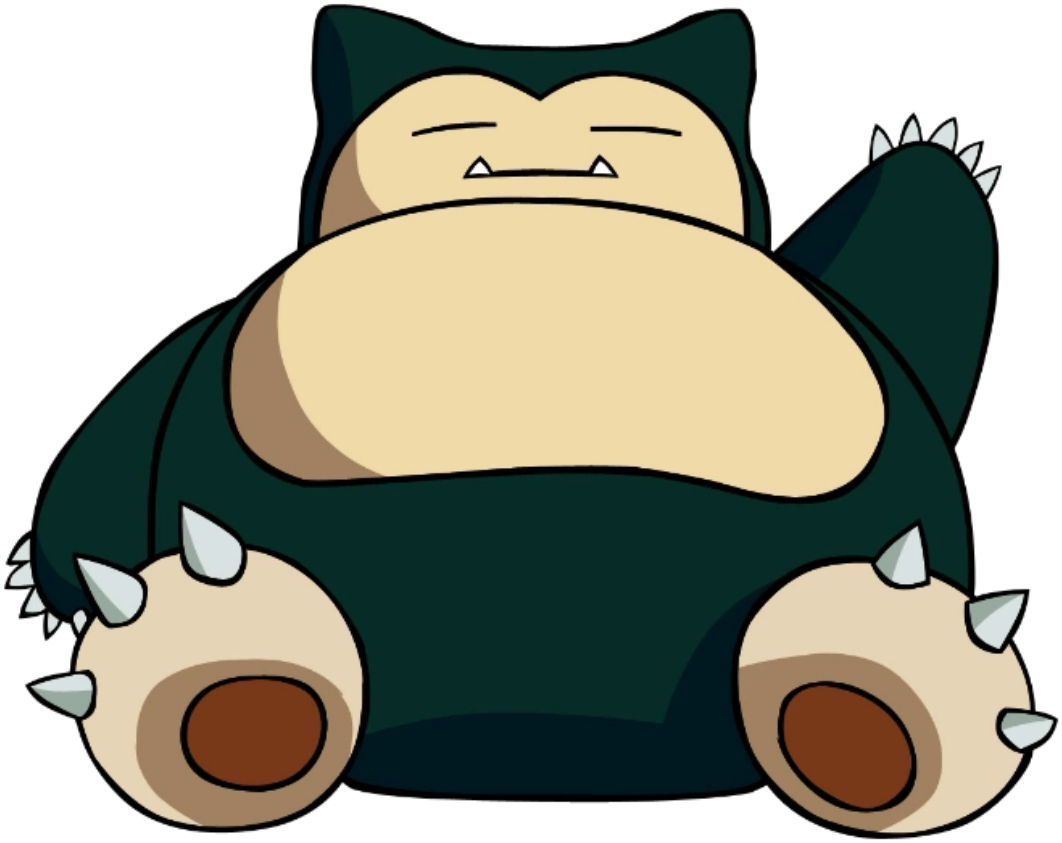
TEAM ABSENTEE

Software Engineering Final Report



Andrew Paek, Nick Smith, Yaoxian Qu, and Doug Schmieder

12.14.2017

VISION STATEMENT

Team Absentee vows to make a well thought out, interesting game full of safe drivers and sprites we stole from Pokemon!

INTRODUCTION

For our project we were tasked with creating a (somewhat realistic) simulation of a traffic system in a town. For our town we decided to scrape all of the building sprites from the popular game *Pokemon* so we named our city Celadon City! In our city we chose to only support one-way roads, but we designed our map so that even after closing a random road (Snorlaxes have a habit of falling asleep in the middle of our roads) in our set of closable roads the cars will still be able to get from any entry point to any landmark or destination. We have a wide variety of landmarks in our city, from Gyms to the famous Celadon City Department store! We're happy to say that the drivers in our fair city are very very good at not crashing into each other and obeying the rules of the road!

TEAM MEMBERS AND CONTRIBUTIONS:

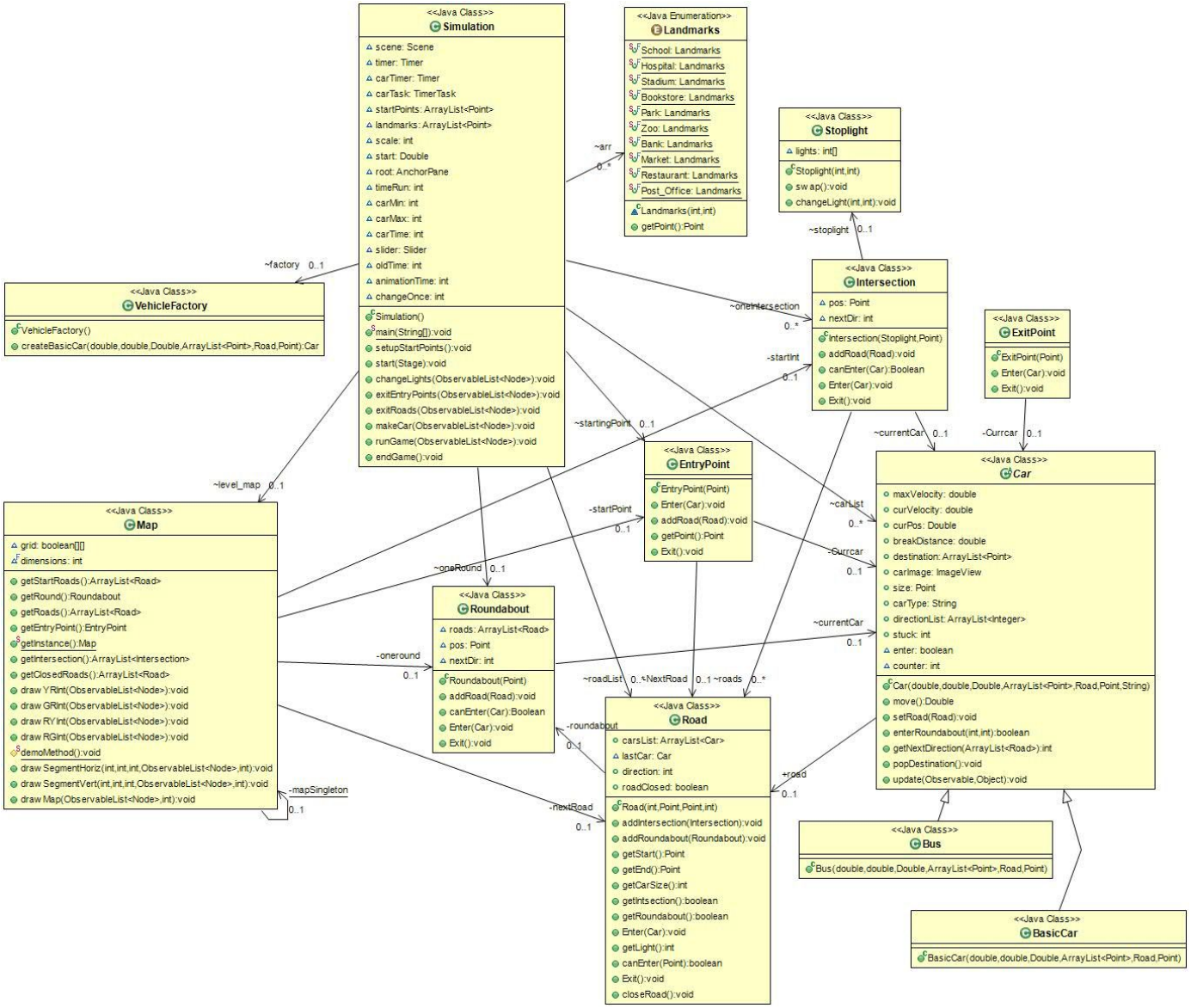
Andrew Paek: I made the car class and road class. In the Car class I created the functionality so that cars could observe other cars and properly move and turn. I made it so that the cars would slow down if they approached an obstacle (primarily cars) and sped up to a max speed if possible. For the road class, I ensured that cars would properly enter roads (adding observers if a car was already in the road) and properly exit by communicating with the intersection. In addition, I made sure that cars and roads could

communicate by creating an instance of the road that the car was located within and a list of all cars that were located in a road.

Douglas Schmieder: worked primarily on the Simulation.java class. He utilized different timers to help run the game. JavaFX was used to handle the simulation as a whole while two other types of timers were used to control different aspects of the simulation. These include changing the lights at set intervals and spawning cars with relation to a slider that is present in the simulation.
Qu Yaoxian: Proposed the initial design of the project. Worked on intersection, roundabout, and stoplight classes.

Nick Smith: Was in charge of all things involved in the map, which included functionality, layout, how it looked and integrating all of the other team members pieces into the map. Given this role I spent a lot of time doing integration tests because the first time we would actually see our individual pieces at work would be when we inserted them into the current map.

FINAL UML DIAGRAM



Our rationale for our Map design is that we wanted to approach this problem of laying out the backend of the map in the most Object-Oriented way as Java lends itself well to this paradigm. Because of this we broke our map up into smaller sub controllers which worked along with each other to build the backend of our map. Our idea was that we wanted to iterate through these pieces of our map and call the Exit() function on them to

facilitate the passing of cars through our road system. In the end this worked quite well, we set up our map and pass the map pieces (roads, intersections,roundabout) to the simulation which took care of calling Exit() on them on every tick of the game.

PROCESS AND TIMELINE

We used Jira to help us organize which tasks we needed to have completed by the end of our two sprints. At the beginning, we created user stories that properly covered each of the requirements that were expected to be met by the end of the project. For each of our sprints, we then gathered the stories that were related to what we needed to have done and put them in the sprint. From there, each member was assigned a task based on which part of the project they were going to be responsible for.

In Sprint 1, we decided to create a basic map that we could populate with cars. We also worked to have the cars be able to see each other to avoid collisions between them. We set up roads and intersections to handle car movement throughout our map. This Sprint was focused mainly on getting the basic functionality in place so that for Sprint 2 we could focus more on the fine tuning of our simulation.

For Sprint 2, we modified the map to include more intersections and roads, along with landmarks and the roundabout. We also included more functionality for our traffic lights. We added landmarks to our cars and created functionality for them to dynamically create their own route as they proceeded through the map.

Our implementation of SCRUM consisted of weekly meetings to make sure that everyone was on the same page as far as what needed to be done and what they should be working on. We used these meetings to voice our concerns about issues that we

foresaw becoming a problem after looking at our code and gave us a chance to prevent them from ever becoming an issue by talking it through with the whole team.

REQUIREMENTS

Vehicle Generation:

- Different Car Generation
- Different car types
- Handle different car sprites and their direction
- Car behavior

Layout/Display Map:

- Create fixed map with 4 way-intersection and roundabouts
- Make final map
- Incorporate roundabouts into Map and give it a special controller

Car perception / vehicle management:

- Should update position of cars every x seconds
- Stop at lights
- Avoid Collisions
- Turning at an intersection
- Set different car speeds and break distances
- collision avoidance 2.0

DESIGN PATTERNS USED

Singleton Pattern: We utilized the singleton design pattern for our map. We did this because we only wanted one instance of our map at any given time and we did not want different classes to access different maps. This let us make sure that there would be no issues when accessing the map from different classes.

Factory Pattern: We used this pattern to make creating cars as easy as possible.

Observer Pattern: This pattern lends itself well to addressing the problem of cars crashing into each other, the cars would observe the car that was directly in front of it so as to not run into it.

TESTING STRATEGY

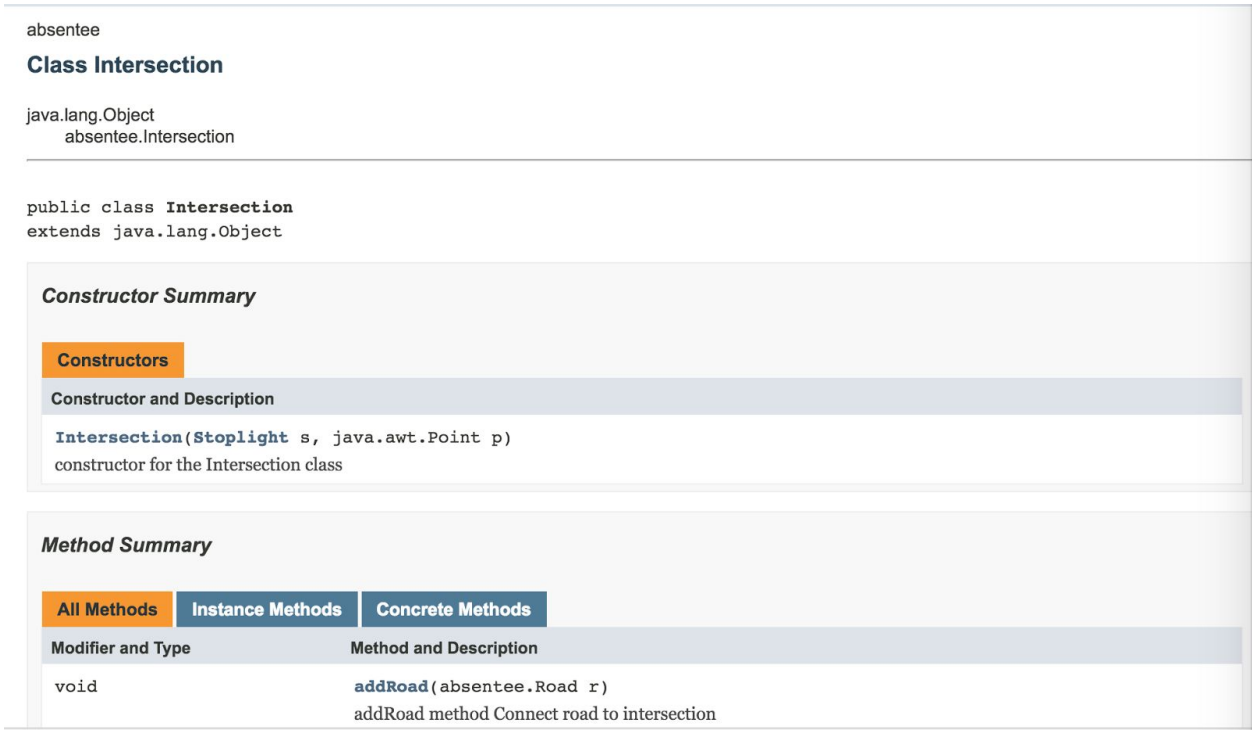
We used a number of different methods to test our simulation. First it started with every 'piece of the puzzle' so to speak, our model consisted of small parts that we fit together to construct the road structure of our beautiful town. Each of these pieces needed to be tested individually to prove beyond a reasonable doubt that they were working as intended. After each aspect of our map was tested individually we were ready to start putting the pieces together to construct our map. This was during sprint 1 when we began this process and for the first sprint we had incorporated roads, intersections, and stop lights together in a segment of our map (only the leftmost portion of the map was actually done at sprint 1). For sprint 2 we had to expand the map to reflect the complete map that we had laid out. This posed a challenge as our map system was well thought out in that it worked really well, but adding more roads and intersections to the map was tedious and time-consuming. At this point when we had a map that was completely set up, testing became trivial because we would add functionality to the map and if everything broke, we did something wrong. On the tail-end of our project this is how

most of our testing went. This would be classified as integration testing.

JAVADOC AND UNIT TEST

Below we provide the javadoc and unit test for the intersection class:

The javadoc is an html file generated by the javadoc tool.



absentee

Class Intersection

java.lang.Object
    absentee.Intersection

```
public class Intersection
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description
`Intersection(Stoplight s, java.awt.Point p)` constructor for the Intersection class

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
void	`addRoad(absentee.Road r)` addRoad method Connect road to intersection

java.lang.Boolean	**canEnter**(absentee.Car car) canEnter allows car to enter if intersection does not have a car in it and current stoplight is green
void	**Enter**(absentee.Car car) Enter method enters a car into intersection
void	**Exit**() Exit method makes a car exit inersection

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Intersection

```
public Intersection(Stoplight s,
                    java.awt.Point p)
```

constructor for the Intersection class

Parameters:

s - a Stoplight object

p - a Point object that represents the position of the intersection

Method Detail

addRoad

```
public void addRoad(absentee.Road r)
```

addRoad method Connect road to intersection

Parameters:

`r - a Road object`

canEnter

```
public java.lang.Boolean canEnter(absentee.Car car)
```

canEnter allows car to enter if intersection does not have a car in it and current stoplight is green

Parameters:

`car - a Car object`

Returns:

`a boolean indicating whether the car can enter or not`

Enter

```
public void Enter(absentee.Car car)
```

Enter method enters a car into intersection

Parameters:

`car - a car object`

Exit

```
public void Exit()
```

Exit method makes a car exit inersection

Below is the screenshot for the unit test:

```
package absentee;

import static org.junit.Assert.*;

import java.awt.Point;
import java.awt.geom.Point2D;
import java.util.ArrayList;

import org.junit.Test;

public class TestIntersection {

    @Test
    public void test() {
        Stoplight s = new Stoplight(0, 0);
        Road r = new Road(3, new Point(0, 0), new Point(0, 3), 0);
        Road[] roads = new Road[4];
        roads[0] = r;
        roads[1] = r;
        roads[2] = r;
        roads[3] = r;
        Intersection i = new Intersection(s, new Point(0, 4));
        i.addRoad(r);
        i.addRoad(r);
        i.addRoad(r);
        i.addRoad(r);
        ArrayList<Point> des = new ArrayList<Point>();
        des.add(new Point(0, 1));
        Car c = new BasicCar(0, 0, new Point2D.Double(100, 0), des, r, new Point(0, 0))
//      cannot enter because red light
        assertFalse(i.canEnter(c));

        //change to green light
        s.changeLight(1, 1);
        assertTrue(i.canEnter(c));

        //enter car
        i.Enter(c);
        Car c2 = new BasicCar(0, 0, new Point2D.Double(100, 0), des, r, new Point(0, 0)
        assertFalse(i.canEnter(c2));
    }

}
```

ACCEPTANCE TEST

1. When the car drives towards the end of the road and sees a red light at the intersection, the car stops.

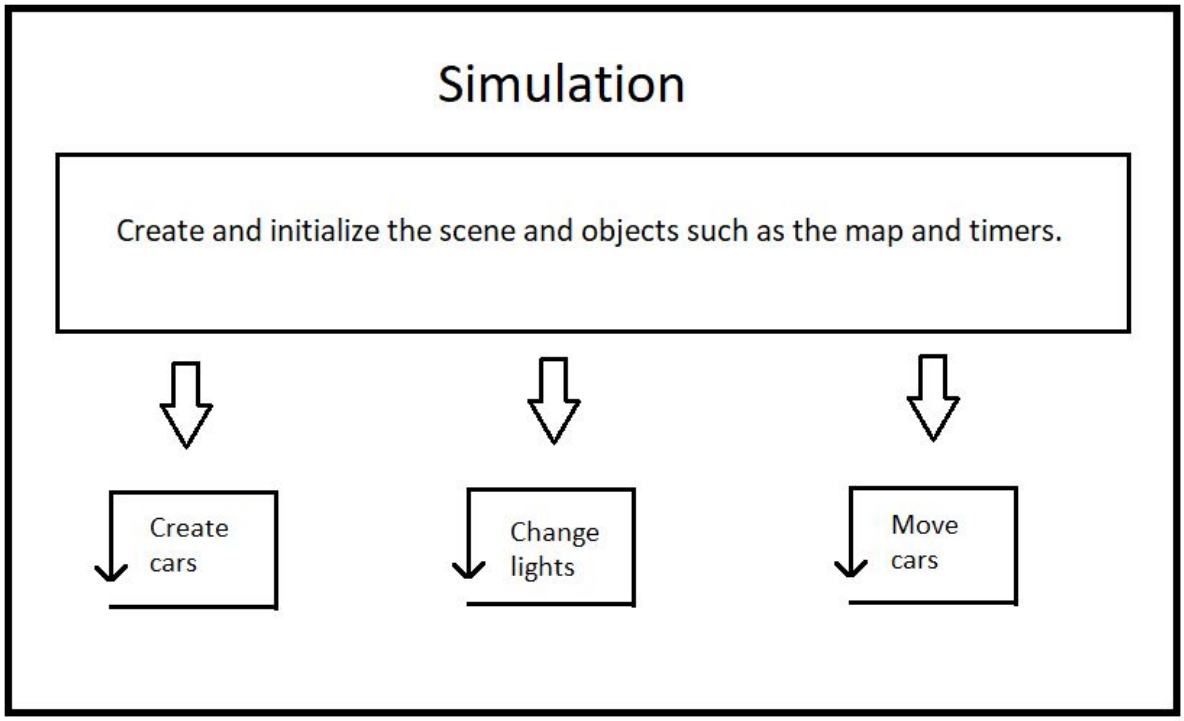
   When the light turns green and the intersection is empty, the car continue driving.

2. When the car enters an intersection and the next road that it is driving towards is full, the car stops and waits.

   When there is extra space in the next road, the car continue driving into the road.

ARCHITECTURAL DIAGRAM

We did not implement our project in the expected MVC format. However, we still had a baseline architecture based on a main controller, Simulation, that ran all aspects of the project. Simulation initially started by creating the javafx scene, the map, and initializing timers. The map then controlled fixed objects such as roads, intersections, and stop lights. Then, through the use of timers and functions, it created cars and modified objects such as cars and stop lights in the map. These timers ran concurrently as different threads, creating a smooth flowing program.


Simulation
Create and initialize the scene and objects such as the map and timers.
Create cars
Change lights
Move cars