High Level Summary of Changes

The bug that was found was taxis not dropping off or picking up people and people not stopping at the taxi rank to be picked up. This was fixed by first adding a 'StoppingArea' just below where the taxi rank spawns to force pedestrians to stop and wait at the taxi rank for pickup. The taxis also now have an occupancy value (up to 4) which determines how many people they will and functionality was added to actually allow for these people to be dropped off (exit the taxi and walk into the lane). The taxi rank was also changed to have functionality to allow the current taxi to pick up a random amount of people (up to 4) once it parks.

Executing Instructions

The tests can be run in multiple ways. They can be run manually from the command line using parameters.

> -testmode: runs the simulation and outputs the results of all tests to std output
>
> -testmode2: runs the simulation with an extra lane in North Terrace and repeats the same tests as in testmode
>
> -visible: this command is only necessary if using a test parameter; it will set the window to visible so the simulation can be observed

The tests can then be run as follows

```
java -jar simtersection_a1647264.jar -testmode | sort -n > test1_results.txt
```

The sort is necessary as due to the random nature of the simulation the tests will sometimes run in a different order (rather than forcing them to run only when others have we can have the sort tool sort the file numerically).

Then the diff command can be run

```
diff -s test1_results.txt expected_results.txt
```

If all is well the output of diff will return

```
Files test1_results.txt and expected_results.txt are identical
```

Otherwise I have included a bash script called test_script, so instead of manually calling the tests you could make the test script executable through chmod +x test_script then simply ran using ./test_script which will run all the tests automatically and compare outputs.

Software Testing and Bug Fixing Process

Finding bugs in a simulation such as Simtersection can be quite difficult, due in part to its random nature and in part with the lack of documentation about what the 'expected results' should be. It was also difficult to find the bugs as it was not immediately obvious that anything was wrong with the simulation, in fact a cursory view of the program in motion does not highlight the bug. Only when the taxi rank is enabled does the bug become quite obvious (the taxi rank essentially does not function at all). Running static testing in this specific scenario, such as a software inspection, does not provide good results as it is extremely difficult to find the bugs in the code without actually running the code.

My method for finding bugs was quite a simple one, first I ran the program with different settings (taxi rank enabled/disabled, different vehicle speeds, night/afternoon models) to see if there was any obvious bugs, especially any that would cause the entire system to crash. Whilst running through the program I was comparing its execution with the requirements of the program as specified in the description of its intended functions (as given in the opening paragraph to the assignment handout). From reading these requirements it became obvious which part of the code was broken (specifically there seemed to be no functionality at all), the taxi ranks. This testing method was a form of black-box visual verification testing as I was running the code, watching its execution and was able to evaluate that the program did not comply with the specifications that had been set. This is not necessarily an effective way of testing but can be quite useful in a simulation such as Simtersection as you cannot feed in traditional test cases or find bugs by inspecting the code.

The most helpful materials provided by the course to me were the lectures on software testing. More specifically the lecture slides relating to unit/component testing, verification, validation testing, and test driven development were the most helpful when designing tests for Simtersection. Another aspect of the course that was useful was the tutorial on jUnit testing and opening a java project inside an IDE (integrated development environment). Whilst it is not entirely necessary to run/debug this code in an IDE it increases efficiency and helps with finding out exactly what variables and methods a class has. All of the bugs could have been fixed using a simple text editor and then compiling from the command line, but using an IDE massively speeds up the process as you have immediate access to all the classes and are able to trivially execute the program from inside the IDE. I found certain resources to be quite helpful, and others to be not so useful, but it is important not to be dismissive of certain testing methods just because they may not be applicable in this specific scenario.

When making any change to the code it was important to test immediately after making the change (regression testing) as due to the interwoven nature of the system if one change caused an error it was likely another part of the system may fail. This meant that after making a change I would run the program, first to see that no actual system crashes occurred and then to see if the

fix achieved its goal. To keep track of these changes I made sure to comment anything beyond a very minor change with a comment beginning with my name, so that whenever I went into a class I could see exactly what I had added or changed. This method worked because only a small number of classes needed to be changed, but this method may not be appropriate if a larger number of classed needed to be changed. In a scenario like this it would be important to have a changelog so all changes can easily be identified without having to trawl through the code to find them. In addition to this it is important to keep a copy of the unedited source code so that if a change is made which breaks the system one can go back to the original code and replace the change with the original code. When making changes to an already established system it is important to test these changes immediately and have them well documented or you may be prone to losing track of changes and cause system-breaking bugs which you can no longer find.

The main method I used for ensuring that my changes worked was to set outputs throughout the program which indicated whether a change was successful, then to confirm visually that the change worked. One example of this is when I was implementing a fix to allow taxis to pick up pedestrians. I set outputs which printed the amount of pedestrians in the lane before the taxi parked, the amount of pedestrians the taxi should have picked up, then the amount of pedestrians the taxi did pick up and finally the amount of pedestrians that were in the lane once the taxi had picked up pedestrians. Outputs like this are very important, in this specific example I found through these outputs that the taxi seemed to always pic up one less passenger than expected (something that would have been hard to confirm visually). Through knowing this I was able to determine that this issue was being caused by a for loop ‘skipping’ over indexes because one had been deleted and the fix was a simple decrement of the iterator every time a pedestrian was deleted.

The test harness I developed to test my program contains the following automated tests. Test one confirms that a stopping area was created. Test two confirms that the occupancy value of a taxi can be set. Test three confirms that a pedestrian can stop at the taxi rank. Test four confirms that a taxi can drop people off. Test five confirms that the created pedestrian walks into the road at an angle of 240 degrees (Northwest). Test six confirms that the taxi can pick up pedestrians. Test seven confirms that the pedestrians turn left (west) once they have walked up into the lane. These automated tests are then run on the simulation, then again on the simulation when an extra lane is added to North Terrace (to show that the changes are dynamic in nature). The output from the tests is then compared to an expected output to demonstrate that the changes were successful.

Lessons Learned

Reading through and attempting to debug this codebase was a very enlightening process. It revealed the difficulty of entering a project once it has been fully developed without any contact or explanation from the project team that developed and designed the project. This is the type of scenario which will appear once in the workforce as you may be hired to update or maintain a large project, which was designed by ex-employees, with poor documentation and poor commenting. One cannot expect that the person who originally commissioned the project has an extensive knowledge of how it works so you must rely on documentation and analyzing the code to gather an understanding.

One of the key things analyzing and debugging this codebase taught me was the importance of good documentation and commenting. Many of the classes in the Simtersection code were poorly commented, whilst some methods were not commented at all. There were often times where I found myself not having a full understanding of what a method did or what a variable was used for. This problem was compounded by the disjoint commenting style (or the lack thereof) followed by the members of the development team. It appeared that some methods were commented quite well but in others the comments appeared to be simply messages to other members of the team, such as found in the pedestrianLane stop() method where the comment reads "THIS SHOULDNT GET CALLED -> Adam: why shouldnt this get called, we want to tell the pedestrian to stop when they don't have a green light." Comments such as these do not provide any insight into what a function does and have no place in a finished project. This project did not come with any other forms of documentation, such as flow diagrams or class hierarchy diagrams which would have been helpful in understanding how classes interact with each other. I learned through experiencing a less-than-adequately documented codebase that including clear, descriptive but concise comments are an integral part of writing code.

Another important field of software engineering I increased my knowledge in was software structure and architecture. It is important when developing software to follow a predefined structure or you may find classes with illogical interactions that aren't linked in a clear coherent manner. The way the Simtersection codebase is structured makes finding classes and their related classes quite easy. The code is split into eight major 'packages' as they are called in Java. These packages each have one overarching goal. Sim deals with setting everything in motion, it can be considered somewhat of a 'driver' package, Entity deals with the abstract 'Entity' class and everything derived from it (Basically anything in the simulation which moves), Event deals with any and all events (all reactions to specific qualifiers), GUI deals with the construction and functionality of the graphical user interface, Light manages the traffic and pedestrian lights, Results manages the collection and interpretation of data, Model with the modeling of the actual North/Frome intersection and Road with the construction of the intersection, roads and their associated lanes. This logical structure is a good example of how to group classes with related roles so that classes are not spread in a haphazard manner making it difficult to find which

classes are related. Having previously never dealt with codebases so large that classes needed to be defined into groups based on their roles I had never put much emphasis on defining and grouping classes based on their roles in the software.

Furthermore I learned how challenging it is to jump into a completed project and attempt to make sense of it all. I have developed an appreciation for just how difficult it can be to understand code that other people have written, especially if it was never really intended to be debugged by anyone but the developers themselves. What helped me in first understanding this code was to 'follow the calls.' By this I mean rather than seeing a method such as stop() and assuming that it just stopped the entity that it was being called on I went into the actual stop() method and attempted to really understand what it was doing rather than just making assumptions (which inevitably leads to making incorrect assumptions about what certain aspects of the program do). Having never written code in Java before (it is however very similar to C++) 'following the calls' also increased my understanding of the Java language itself. This was very helpful when I began writing my own code as I was able to use my understanding of the methods that were already in the code to implement my own changes.

The final and possibly most important area in which I increased my knowledge was software testing. Prior to writing tests for the Simtersection simulation much of my code was tested by simply observing the results as the code was being written, with no clear testing structure in place. Rather than thoroughly testing the entire project I would simply be observing results, this would mean that whilst the correct result may be produced, it does not necessarily mean it will for all sets of input. Writing tests for Simtersection also taught me how to be smart about what, where and when to test. In a program which has a 'random' nature one must be smart about how test cases are written, to ensure that the program meets specifications, which can be a challenge when input cannot necessarily be strictly controlled. This issue made me think more about how I structured my tests when the traditional lower bound, upper bound, value in between boundary testing was not appropriate.