Singleton Design Pattern Implementation

With the growth of apps like Uber and Lyft, more and more people are starting to rely on these services rather than even owning their own vehicle. Because of this, these companies have been able to make higher and ever-growing profits. A big part of these apps relies on calculating the cost of a trip based on the distance traveled and the time the trip takes (for instance, rides during rush hour are bound to be more expensive than rides at other times). For our project, we decided to implement the Singleton design pattern to mimic this cost calculation and see which path algorithm is the most cost efficient. By using the Singleton design pattern, we are able to easily export position data about the robot for each movement it makes. We are also able to keep track of the cumulative distance and cost traveled throughout. With this data, we were then able to calculate the total cost of a trip and determine which path-finding algorithm was the best.

Path-finding algorithms have always been interesting because of how varied their results can be. Each algorithm tends to have different pros and cons, and it can be difficult to decide which is truly the best algorithm. After all, there is no truly best algorithm, it all depends on the scenario. In terms of finding a cost-efficient algorithm for our project's robot movement, we only considered the time and distance variables. So, while the best algorithm is still based on the scenario because of how the road structure of our map is, it is easier to find a single algorithm that works better than others. Of course, it also has to be a realistic algorithm. For instance, the beeline algorithm is generally the fastest, but it is not realistic as it allows the robot to just move through buildings and not follow the road/sidewalk paths. By simulating travel around campus and calculating the costs, we can get a better understanding of how apps like Uber work and maybe even get insight into the algorithms they may be using. Not only Uber, but this simulation

allows us to better understand how a GPS works and even gives us insight into potential path-finding algorithms that a GPS may use when calculating routes.

Our extension of the project allows us to collect data related to distance and time based on the drone and robot movement. We were able to use an already provided function from the vector3 class to calculate the distance between every point that the drone and robot traveled to. The variable “dt,” which is passed in as a parameter to the “update” function, was used as our measurement of time for sake of simplicity instead of having to keep track of a real-time clock. In order to export this data to a csv file, we used the Singleton design pattern and modified the drone class by sending the drone’s position, accumulated distance, time, and cost every time the update function was called and the drone/robot had not yet reached the destination. The data was only collected from the point where the robot was picked up until the point where the robot was dropped off. We did not consider any travel time or distance calculations of the drone moving from one robot’s dropoff point to the next robot’s pick up point in our calculations. After one whole trip/ride had been completed, we were then able see the final cost of the trip in the last line of our csv file as well as display it to the user in the terminal.

We made sure to write our code in a way where it would follow SOLID principles. Originally we had written our code in two different formats. One of the ways was defining a new function and calculating the distance/cost within each route strategy file, and the other was just calculating the cost and distance in the drone’s update function. Both ways gave us the same output in our csv files when used, however, one would cause us to update code in eight files, whereas the other would just have us add a block of code to one file in a pre-existing function. We were able to narrow down which format to use by figuring out which one was more open to extension and closed to modification. This led us to choosing the second format of just updating

the update function drone file. If another routing strategy were added to the project, there wouldn't be any need to modify that file to calculate its distance or cost, the code in the drone class would still work perfectly.

We chose to use the Singleton design pattern to collect and analyze data using the following variables: position of the drone, distance, time, and cost. The Singleton pattern is a creational design pattern, which ensures that only one object of its kind exists and provides a global point of access to that instance. The purpose of the Singleton class is to control object creation and limit the number of objects to only one. One reason that leads us to use the Singleton pattern is that a Singleton can be used repeatedly, instead of a single instance of a class. When a client schedules a trip, the update function gets called repeatedly. For example, for a 0.2-mile distance, AStar, DFS, and Dijkstra algorithms each call the update function about 1000 times. So, instead of defining an object of file class, which is the Singleton class in our program 1000 times, we used the Singleton design pattern and created a static object of Singleton class when the drone is created. This made it so we only needed one instance of the class that would output to one common csv file that could be globally accessed in any file where data could be extracted from, in our case, only the drone class.

Through using the Singleton design principle, we were able to collect and analyze data regarding path movement. With that, we were able to look into which path-finding algorithms seemed to have a lower cost than others. From our testing, it seemed that between DFS, Dijkstra, and AStar, the AStar algorithm had the lowest cost, and DFS had the highest cost. These results seem accurate, but, as mentioned earlier, the cost may rely on the scenario. Overall, being able to see which strategy takes the least amount of time and has the lowest cost can be beneficial for

both the customer and business. With this cost information, they are able to figure out which strategy will result in the greatest profit.

UML (**bold** is what we added to the pre-existing file/code “Drone”):