TEAM MEMBERS
1) Abdullah Meneese menee008
2) Omonigho Egi egi00002
3) Nicholas Normandin-Parker norma484
4) Jacob Thompson thom4762

Video Presentation Link (download the pdf to use hyperlinks)
https://drive.google.com/file/d/1sl1-GRB71JkrWc6KZRIHsPxsywUvusrm/view?usp=share_link

What is the project about?

The Big picture of this project was to incorporate Design and Development into an applied "Big" project. This semester's task was to work on Autonomous Vehicle Transportation (like Uber ++). Our Project Planning involved Designing and Building a system over the course of the semester, that is both flexible and efficient.

How to run the simulation? (TODO add Docker instructions)

1) **Docker**

Download docker and then you would pull are image via
docker pull jacobthompson12/homework4
Follow by
docker run jacobthompson12/homework4
The simulation will be visually projected and can be accessed at the following links (if you are using VS CODE SSH, you will need to create a port or use VOLE)

- http://127.0.0.1:8081/
- http://127.0.0.1:8081/schedule.html

1) From the *first link,* you will see the visual simulation along with all of the entities, such as the Drone, Helicopter, Rat, and Charging Stations.
2) You'll notice that there are no robots (people) in the simulation; this can be added in *schedule.html,* where you can make your uber trip.
3) You can add as many robots as possible and watch from the simulation (*first link*) as they are delivered.

2) **Source Code**

If you have access to the source code, start with the following:
git clone git@github.umn.edu:umn-csci-3081-F22/Team-001-39-homework04.git
Open a terminal and locate
umn-csci-3081-F22/Team-001-39-homework04
followed by running

make -j; make run

The simulation will be visually projected and can be accessed at the following links (if you are using VS CODE SSH, you will need to create a port or use VOLE)

- http://127.0.0.1:8081/
- http://127.0.0.1:8081/schedule.html

4) From the *first link,* you will see the visual simulation along with all of the entities, such as the Drone, Helicopter, Rat, and Charging Stations.
5) You'll notice that there are no robots (people) in the simulation; this can be added in *schedule.html,* where you can make your uber trip.
6) You can add as many robots as possible and watch from the simulation (*first link*) as they are delivered.

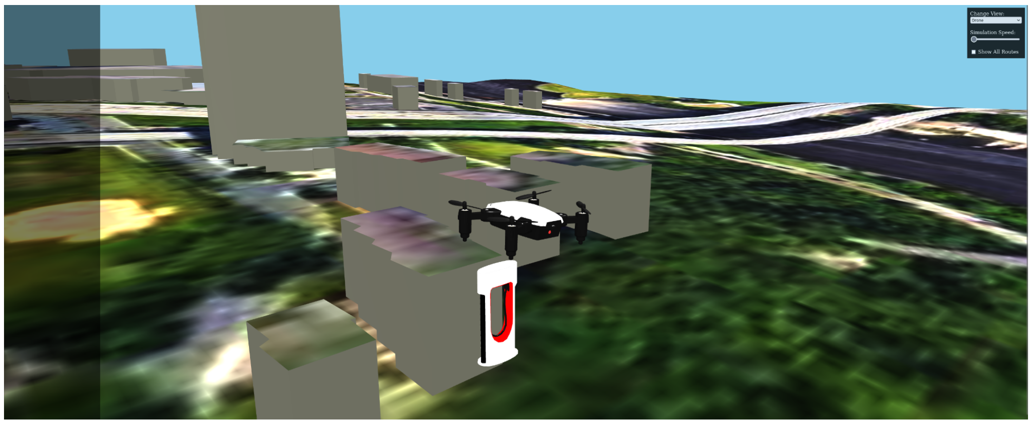
3) Hosted Local (may close)

- http://place.org:23456/
- http://place.org:23456/schedule.html

No need to pull docker or github, follow step 4-6 from above

What does the simulation do?

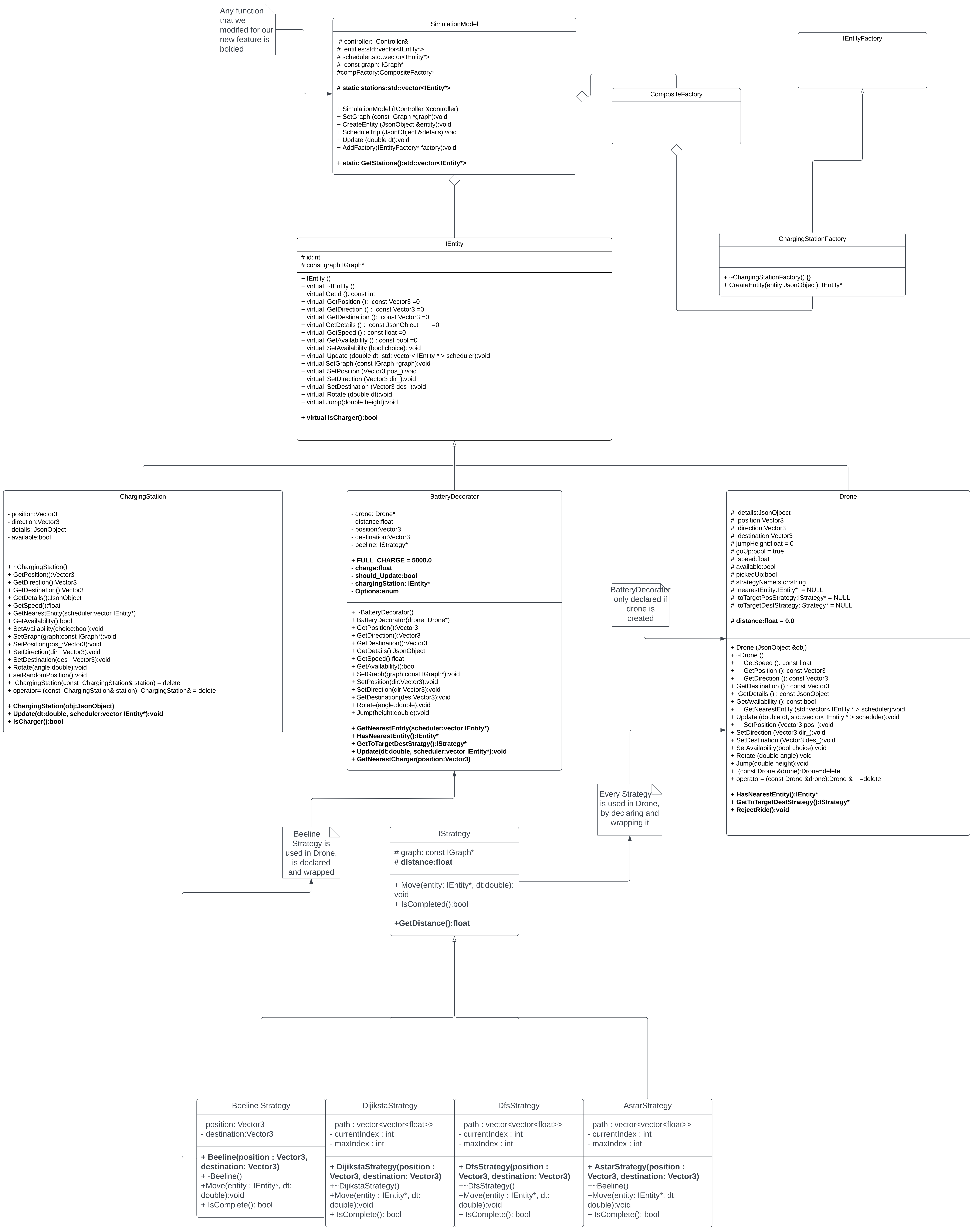
This project will be able to simulate the behavior of drones and robots. You will be able to set the pickup location and the robot's final destination; afterward, the drone will come and pick up the robot toward its goal. Not only the transportation simulation, but you will be able to decide what to do with the robot's behavior when the passenger arrives at their destination or if the drone is too late to pick up.



New Feature

Our group implemented a battery system using a Decorator and Factory Design Pattern. We implement a new Factory Asset as a Tesla Charging Station and a Battery Decorator around the Drone to represent what the Drone would do if it had a rechargeable battery.

UML DIAGRAM (at the bottom of README or available as a .pdf in our GitHub repository)


Any function that we modifed for our new feature is bolded
SimulationModel
# controller: IController&
# entities:std::vector<IEntity*>
# scheduler:std::vector<IEntity*>
# const graph: IGraph*
#compFactory:CompositeFactory*
# static stations:std::vector<IEntity*>
+ SimulationModel (IController &controller)
+ SetGraph (const IGraph *graph):void
+ CreateEntity (JsonObject &entity):void
+ ScheduleTrip (JsonObject &details):void
+ Update (double dt):void
+ AddFactory(IEntityFactory* factory):void
+ static GetStations():std::vector<IEntity*>
CompositeFactory
IEntityFactory
ChargingStationFactory
+ ~ChargingStationFactory() {}
+ CreateEntity(entity:JsonObject): IEntity*
IEntity
# id:int
# const graph:IGraph*
+ IEntity ()
+ virtual ~IEntity ()
+ virtual GetId (): const int
+ virtual GetPosition (): const Vector3 =0
+ virtual GetDirection () : const Vector3 =0
+ virtual GetDestination (): const Vector3 =0
+ virtual GetDetails () : const JsonObject =0
+ virtual GetSpeed () : const float =0
+ virtual GetAvailability () : const bool =0
+ virtual SetAvailability (bool choice): void
+ virtual Update (double dt, std::vector< IEntity * > scheduler):void
+ virtual SetGraph (const IGraph *graph):void
+ virtual SetPosition (Vector3 pos_):void
+ virtual SetDirection (Vector3 dir_):void
+ virtual SetDestination (Vector3 des_):void
+ virtual Rotate (double dt):void
+ virtual Jump(double height):void
+ virtual IsCharger():bool
ChargingStation
- position:Vector3
- direction:Vector3
- details: JsonObject
- available:bool
+ ~ChargingStation()
+ GetPosition():Vector3
+ GetDirection():Vector3
+ GetDestination():Vector3
+ GetDetails():JsonObject
+ GetSpeed():float
+ GetNearestEntity(scheduler:vector IEntity*)
+ GetAvailability():bool
+ SetAvailability(choice:bool):void
+ SetGraph(graph:const IGraph*):void
+ SetPosition(pos_:Vector3):void
+ SetDirection(dir_:Vector3):void
+ SetDestination(des_:Vector3):void
+ Rotate(angle:double):void
+ setRandomPosition():void
+ ChargingStation(const ChargingStation& station) = delete
+ operator= (const ChargingStation& station): ChargingStation& = delete
+ ChargingStation(obj:JsonObject)
+ Update(dt:double, scheduler:vector IEntity*):void
+ IsCharger():bool
BatteryDecorator
- drone: Drone*
- distance:float
- position:Vector3
- destination:Vector3
- beeline: IStrategy*
+ FULL_CHARGE = 5000.0
- charge:float
- should_Update:bool
- chargingStation: IEntity*
- Options:enum
+ ~BatteryDecorator()
+ BatteryDecorator(drone: Drone*)
+ GetPosition():Vector3
+ GetDirection():Vector3
+ GetDestination():Vector3
+ GetDetails():JsonObject
+ GetSpeed():float
+ GetAvailability():bool
+ SetGraph(graph:const IGraph*):void
+ SetPosition(dir:Vector3):void
+ SetDirection(dir:Vector3):void
+ SetDestination(des:Vector3):void
+ Rotate(angle:double):void
+ Jump(height:double):void
+ GetNearestEntity(scheduler:vector IEntity*)
+ HasNearestEntity():IEntity*
+ GetToTargetDestStratgy():IStrategy*
+ Update(dt:double, scheduler:vector IEntity*):void
+ GetNearestCharger(position:Vector3)
BatteryDecorator only declared if drone is created
Drone
# details:JsonOjbect
# position:Vector3
# direction:Vector3
# destination:Vector3
# jumpHeight:float = 0
# goUp:bool = true
# speed:float
# available:bool
# pickedUp:bool
# strategyName:std::string
# nearestEntity:IEntity* = NULL
# toTargetPosStrategy:IStrategy* = NULL
# toTargetDestStrategy:IStrategy* = NULL
# distance:float = 0.0
+ Drone (JsonObject &obj)
+ ~Drone ()
+ GetSpeed (): const float
+ GetPosition (): const Vector3
+ GetDirection (): const Vector3
+ GetDestination () : const Vector3
+ GetDetails () : const JsonObject
+ GetAvailability (): const bool
+ GetNearestEntity (std::vector< IEntity * > scheduler):void
+ Update (double dt, std::vector< IEntity * > scheduler):void
+ SetPosition (Vector3 pos_):void
+ SetDirection (Vector3 dir_):void
+ SetDestination (Vector3 des_):void
+ SetAvailability(bool choice):void
+ Rotate (double angle):void
+ Jump(double height):void
+ (const Drone &drone):Drone=delete
+ operator= (const Drone &drone):Drone & =delete
+ HasNearestEntity():IEntity*
+ GetToTargetDestStrategy():IStrategy*
+ RejectRide():void
Beeline Strategy is used in Drone, is declared and wrapped
Every Strategy is used in Drone, by declaring and wrapping it
IStrategy
# graph: const IGraph*
# distance:float
+ Move(entity: IEntity*, dt:double): void
+ IsCompleted():bool
+GetDistance():float
Beeline Strategy
- position: Vector3
- destination:Vector3
+ Beeline(position : Vector3, destination: Vector3)
+~Beeline()
+Move(entity : IEntity*, dt: double):void
+ IsComplete(): bool
DijikstaStrategy
- path : vector<vector<float>>
- currentIndex : int
- maxIndex : int
+ DijikstaStrategy(position : Vector3, destination: Vector3)
+~DijikstaStrategy()
+Move(entity : IEntity*, dt: double):void
+ IsComplete(): bool
DfsStrategy
- path : vector<vector<float>>
- currentIndex : int
- maxIndex : int
+ DfsStrategy(position : Vector3, destination: Vector3)
+~DfsStrategy()
+Move(entity : IEntity*, dt: double):void
+ IsComplete(): bool
AstarStrategy
- path : vector<vector<float>>
- currentIndex : int
- maxIndex : int
+ AstarStrategy(position : Vector3, destination: Vector3)
+~Beeline()
+Move(entity: IEntity*, dt: double):void
+ IsComplete(): bool

What does it do?

When you run our simulation, you will notice that a drone cannot make a delivery, meaning it doesn't have enough charge; it will go to one of the tesla chargers and fill its battery to 100%.

Why is it significantly interesting?

This is important in simulating an autonomous uber system, as a car shouldn't make delivery if it doesn't have enough battery. In a realistic approach, it shouldn't accept delivery if it can't fulfill it. This added to the realistic simulation that we are trying to develop.

How does it add to the existing work?

Our work adds a seamless integration of learned design patterns, providing more accuracy on how an Autonomous Vehicle Transportation system would work. We added a battery and recharge system with built-in logic representing a real-world system. In a general sense, any autonomous system needs to calculate whether or not it has the capacity to make a delivery. Whether it's drones or cars etc. our system implements logic that would be important to a company in this field both financially and operationally. Failed deliveries would have a negative financial impact on any company within this field. But Also by calculating whether it can make a trip and make it to a charging station with 2-3% battery, which allows for optimal productivity, thus profit.

Which design pattern did you choose to implement and why?

To implement a battery system.

1) We used a factory pattern to represent the Tesla charging station; we used a factory pattern as it is best for creating new objects or entities.
2) We used a Decorator to describe the battery system, which creates flexibility. We could un-wrap the drone, and the drone would function without a battery representation. This allows for flexibility, as any new systems we want to implement on our drone can be added through a new or modified Decorator.

Furthermore, this new feature which we implemented demonstrates the power and simplicity of various design patterns. These design patterns allow for a flexible system which upgrades and changes based on business needs. We can create a battery decorator for different seasons which addresses issues within each season. Perhaps a winter battery decorator can accommodate for snow storms or blizzards which prevent charging station accessibility.

How to use this new feature

You can use the new feature by adding multiple robots into the simulator (use schedule.html) and watching both the simulation and terminal to see how the drone uses the charging station and update the terminal with the battery, and total distance traveled. This is an implementation feature, so a lot of work can be seen through the simulation. We added multiple drones and charging stations to show a real world application.