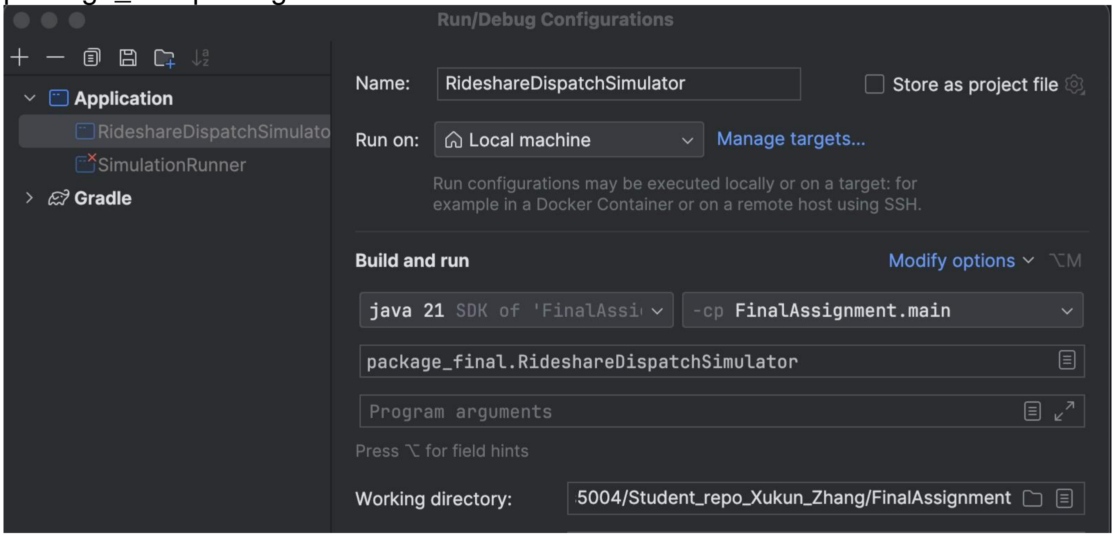
Project write-up

Name: Xukun Zhang
Option 1: RideshareDispatchSimulator

1.How to run the program:

(1) Run the 'RideshareDispatchSimulator' configuration in your IDE, it will execute the main method of the RideshareDispatchSimulator class that is located in the package_final package.



Name: RideshareDispatchSimulator
Run on: Local machine
Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.
Build and run
java 21 SDK of 'FinalAssi
-cp FinalAssignment.main
package_final.RideshareDispatchSimulator
Program arguments
Press ⌥ for field hints
Working directory: 5004/Student_repo_Xukun_Zhang/FinalAssignment

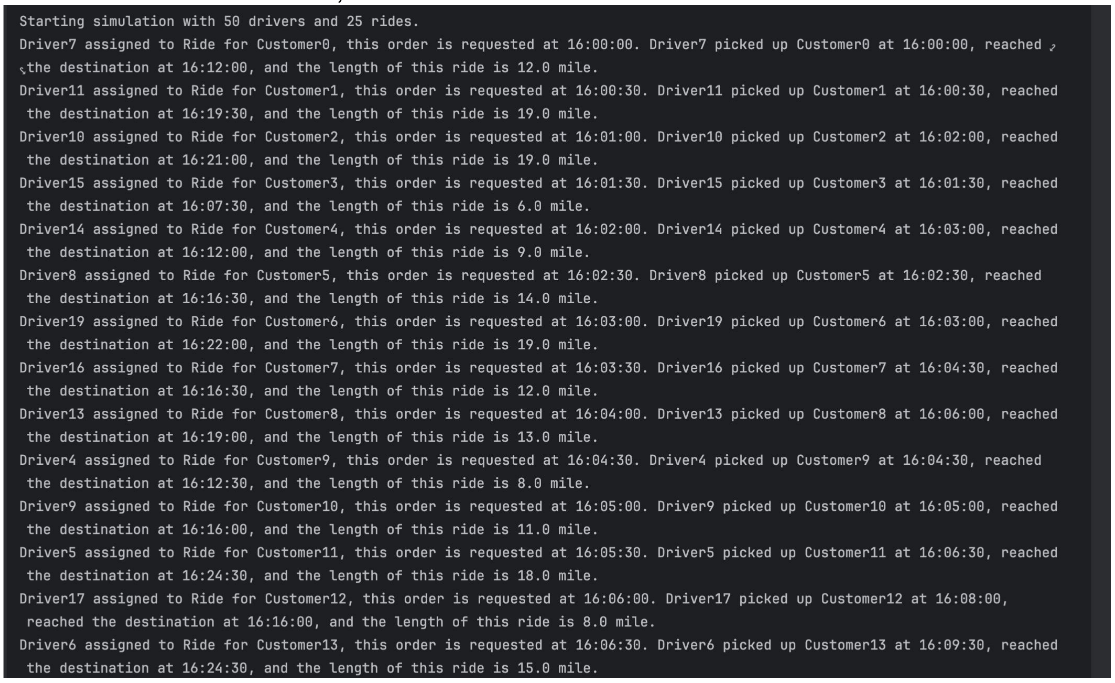
The program will start running simulations based on predefined configuration instances (config1, config2, config3), as the question requested.
Each simulation will display information about the number of drivers and rides, and the statistics of the simulation will be printed at the end.
(2) Adjust Simulation Parameters:
If you wish to adjust the simulation parameters, you can modify the values of numberOfDrivers and numberOfRides in the SimulationConfig instances (config1, config2, config3) inside the main method.
(3) Observe Results:
The program will simulate ride requests, driver assignments, and ride finishes based on the specified configurations.
The statistics printed at the end of each simulation will provide insights into average wait time, average number of rides per driver, and average ride time.

2.Result:

(1) For questions:
What was the average wait time for a ride?
What was the average number of rides a driver has handled?

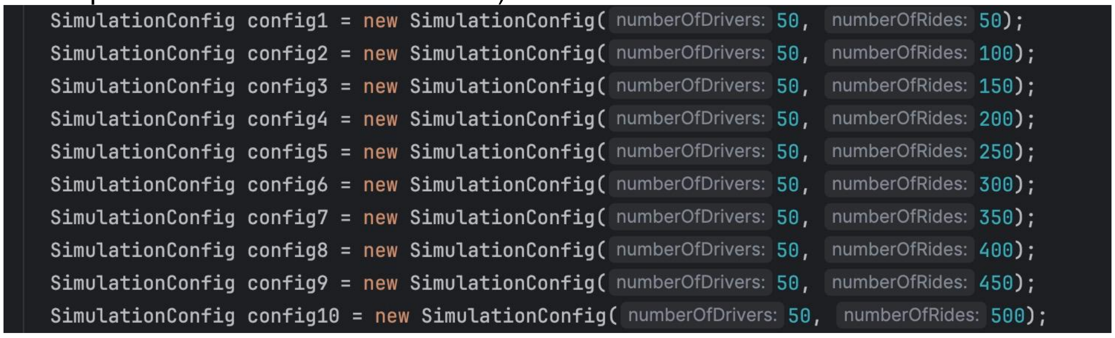
Under the assumption of a **90-minute work time** per driver, I calculate the answers for the specified simulation scenarios:
a.50 drivers and 25 rides:
Average wait time for a ride: 3.28min
Average number of rides per driver: 0.5
Explain: The average wait time for a ride is 3.28 minutes. **This short wait time can be attributed to the initial assignment of drivers**, where they were distributed evenly from positions 1 to 50. Consequently, many of the initial passengers were **quickly matched with drivers located nearby.**
b.50 drivers and 100 rides:
Average wait time for a ride: 9.32min

Average number of rides per driver: 2.0
c.50 drivers and 250 rides:
Average wait time for a ride: 11.94min
Average number of rides per driver: 3.16
Explain: **However**, as subsequent drivers' positions were determined by the destinations of the rides they accepted, they often needed to travel back to the pickup locations after completing their previous rides. **This additional travel time contributed to longer wait times** for passengers as the simulation progressed.
Results shows in terminal, like:



```
Starting simulation with 50 drivers and 25 rides.
Driver7 assigned to Ride for Customer0, this order is requested at 16:00:00. Driver7 picked up Customer0 at 16:00:00, reached
 the destination at 16:12:00, and the length of this ride is 12.0 mile.
Driver11 assigned to Ride for Customer1, this order is requested at 16:00:30. Driver11 picked up Customer1 at 16:00:30, reached
 the destination at 16:19:30, and the length of this ride is 19.0 mile.
Driver10 assigned to Ride for Customer2, this order is requested at 16:01:00. Driver10 picked up Customer2 at 16:02:00, reached
 the destination at 16:21:00, and the length of this ride is 19.0 mile.
Driver15 assigned to Ride for Customer3, this order is requested at 16:01:30. Driver15 picked up Customer3 at 16:01:30, reached
 the destination at 16:07:30, and the length of this ride is 6.0 mile.
Driver14 assigned to Ride for Customer4, this order is requested at 16:02:00. Driver14 picked up Customer4 at 16:03:00, reached
 the destination at 16:12:00, and the length of this ride is 9.0 mile.
Driver8 assigned to Ride for Customer5, this order is requested at 16:02:30. Driver8 picked up Customer5 at 16:02:30, reached
 the destination at 16:16:30, and the length of this ride is 14.0 mile.
Driver19 assigned to Ride for Customer6, this order is requested at 16:03:00. Driver19 picked up Customer6 at 16:03:00, reached
 the destination at 16:22:00, and the length of this ride is 19.0 mile.
Driver16 assigned to Ride for Customer7, this order is requested at 16:03:30. Driver16 picked up Customer7 at 16:04:30, reached
 the destination at 16:16:30, and the length of this ride is 12.0 mile.
Driver13 assigned to Ride for Customer8, this order is requested at 16:04:00. Driver13 picked up Customer8 at 16:06:00, reached
 the destination at 16:19:00, and the length of this ride is 13.0 mile.
Driver4 assigned to Ride for Customer9, this order is requested at 16:04:30. Driver4 picked up Customer9 at 16:04:30, reached
 the destination at 16:12:30, and the length of this ride is 8.0 mile.
Driver9 assigned to Ride for Customer10, this order is requested at 16:05:00. Driver9 picked up Customer10 at 16:05:00, reached
 the destination at 16:16:00, and the length of this ride is 11.0 mile.
Driver5 assigned to Ride for Customer11, this order is requested at 16:05:30. Driver5 picked up Customer11 at 16:06:30, reached
 the destination at 16:24:30, and the length of this ride is 18.0 mile.
Driver17 assigned to Ride for Customer12, this order is requested at 16:06:00. Driver17 picked up Customer12 at 16:08:00,
 reached the destination at 16:16:00, and the length of this ride is 8.0 mile.
Driver6 assigned to Ride for Customer13, this order is requested at 16:06:30. Driver6 picked up Customer13 at 16:09:30, reached
 the destination at 16:24:30, and the length of this ride is 15.0 mile.
```

(2) For question:
What would be the optimal number of drivers on the roads, to balance your business operating costs, with the customer convenience?
I made some assumptions to observe the performance between **50 drivers and varying numbers of orders (more than teacher requested)**. (under the same assumption of work time and rest time)



```
    SimulationConfig config1 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 50);
    SimulationConfig config2 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 100);
    SimulationConfig config3 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 150);
    SimulationConfig config4 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 200);
    SimulationConfig config5 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 250);
    SimulationConfig config6 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 300);
    SimulationConfig config7 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 350);
    SimulationConfig config8 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 400);
    SimulationConfig config9 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 450);
    SimulationConfig config10 = new SimulationConfig( numberOfDrivers: 50,  numberOfRides: 500);
```

Firstly, I assumed that the drivers' work time starts from the first order (midnight) and that each driver stops accepting new orders after 90 minutes to ensure they have enough rest time. This was done to address the question posed about balancing operational costs and customer convenience.

I've created the Excel file with your simulation results.

Number of Drivers	Number of Rides	Average Wait Time (min)	Average Number of Rides per Driver	Average Ride Time (min)
50	50	4.06	1	23.72
50	100	8.56	2	23.72
50	150	9.6	3	24.81
50	200	10.43	3.12	26.19
50	250	9.79	3.28	25.3
50	300	10.16	3.28	25.67
50	350	10.55	3.14	24.89
50	400	10.77	3.08	25.29
50	450	10.88	3.22	26.3
50	500	10.37	3.12	25.27

By examining the simulated results, including average wait times and the number of orders handled per driver, we can draw some conclusions.
For instance, as the number of orders increases, the average wait time also increases, indicating a need for more drivers to meet customer demand and reduce wait times. On the other hand, while the average number of orders handled per driver increases, the rate of increase is relatively small. This could be attributed to the limited work time of drivers, preventing them from accepting more orders.
Based on these assumptions and observed results, it can be inferred that, **under the given work time conditions, 50 drivers and 100 orders may be a suitable choice.** This ensures drivers have sufficient rest time while maximizing their work time to meet customer demands, thus balancing operational costs and customer convenience.

3.Overview of the chosen problem, and high-level overview of the solution:

(1)Overview of the chosen problem:

The chosen problem 'Option 1' is to develop a rideshare dispatch simulator, which aims to simulate the process of dispatching drivers to serve ride requests from customers.
My project involves creating a rideshare dispatch simulator, which simulates the process of dispatching drivers to serve ride requests from customers. The simulator aims to provide insights into various aspects of the rideshare experience, such as wait times for customers, average ride durations, and the optimal number of drivers required to balance business operating costs with customer convenience. The system generates ride requests with specified attributes such as starting and destination locations, anticipated ride distance, request time, and ride type. It then assigns priority to each customer based on ride type and distance. Drivers are allocated to ride requests, and if no drivers are available, customers are placed in a waiting queue. The simulation tracks relevant data such as customer and ride information, and it supports events for ride requests and ride completions. The system evaluates different simulation scenarios with varying numbers of drivers and rides to analyze average wait times and the average number of rides per driver. Additionally, it incorporates assumptions regarding driver work hours and rest times to optimize driver utilization and customer satisfaction.

(2)high-level overview of the solution:

The high-level overview of the solution involves several key components:
a.Simulation Environment Setup: The program initializes simulation configurations such as the number of drivers, the number of rides, and other parameters required for the simulation.
b.Driver and Ride Representation: The system models drivers and rides using appropriate data structures and classes. Each driver has attributes such as driver ID, current location, and availability status. Similarly, each ride consists of a customer, pickup, and drop-off locations, and ride type.
b.Event-Based Simulation: The simulation operates on an event-driven architecture, where events such as ride requests and ride completions trigger corresponding

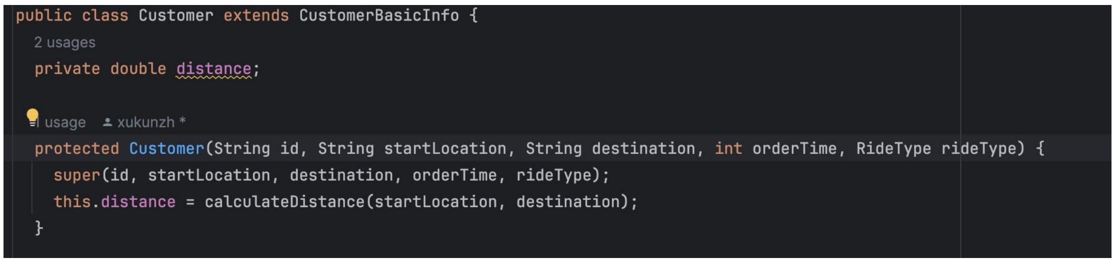
actions in the system. Events are processed sequentially based on their timestamps, simulating the real-time nature of ride dispatching.
d.Dispatching Logic: The core logic of the system revolves around dispatching drivers to incoming ride requests. This involves finding the nearest available driver to each ride request and assigning the ride to that driver if available. If no driver is available, the ride is added to a waiting queue.
e.Driver and Ride Management: The system manages driver availability, ride assignment, and processing of waiting rides. It updates driver status after each ride assignment and handles unassigned rides by recovering them when drivers become available.
f.Simulation Execution: The simulator executes multiple simulations based on different configurations to evaluate system performance under varying conditions, such as different numbers of drivers and ride requests.

3.Description of key challenges encountered during design and implementation, and how those challenges were addressed.
(1) Efficient Driver Assignment: One challenge was efficiently assigning drivers to ride requests based on factors such as driver availability, proximity to the customer, and ride type. To address this, an algorithm was implemented to find the nearest available driver to each ride request, optimizing both time and distance metrics.
(2)Handling Dynamic Driver Availability: Managing driver availability in real-time posed a challenge, as drivers could become available or unavailable during the simulation due to completing rides or other factors. To address this, the system continuously updated the availability status of drivers and adjusted ride assignments accordingly.
(3)Optimizing Ride Queue Management: Efficiently managing the waiting queue of unassigned rides required careful consideration. To address this, a priority-based approach was adopted, where rides were prioritized based on factors such as waiting time and ride urgency, ensuring fair and timely assignment to available drivers.
(4)Scalability and Performance: As the number of drivers and ride requests increased, the system's performance and scalability became critical. To address this, the code was optimized for performance, and data structures were chosen to minimize computational overhead, allowing the simulator to handle large-scale simulations efficiently.

4.The sources and resources used in the project.
The knowledge gained from our classes was useful. Concepts like regular expressions proved useful for parsing and validating input data, while discussions on data and stamp coupling were pivotal in shaping the design of modular and decoupled components in the system.
Furthermore, I utilized online resources and repositories, particularly platforms like GitHub, to explore existing solutions and best practices concerning dispatching problems. By studying code examples and algorithms implemented by other developers, I gained valuable insights into effective strategies for driver allocation, ride scheduling, and queue management.

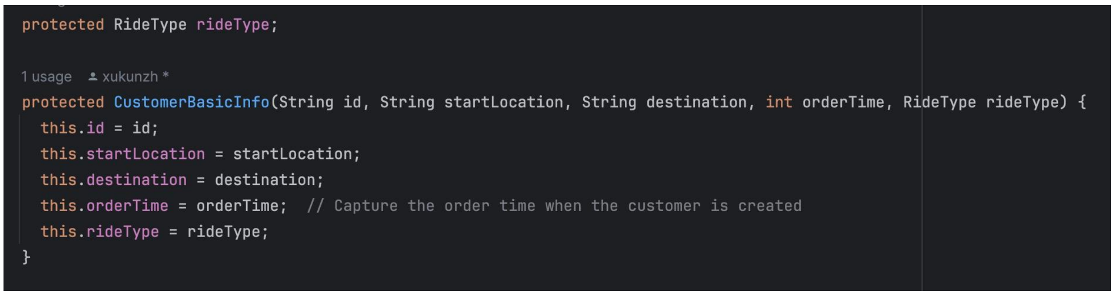
5.Questions about the software system.
Project：Option 1
1. Please include a code snippet showing how have you used inheritance and composition in your code.

Inheritance: The Customer class extends the CustomerBasicInfo class.



```
public class Customer extends CustomerBasicInfo {
  2 usages
  private double distance;

  usage   xukunzh *
  protected Customer(String id, String startLocation, String destination, int orderTime, RideType rideType) {
    super(id, startLocation, destination, orderTime, rideType);
    this.distance = calculateDistance(startLocation, destination);
  }
```

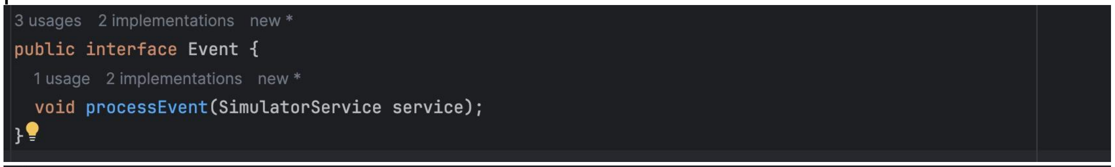
Composition: The Customer class uses composition by having a RideType object as a part of its state, indicating the type of ride a customer requests. This relationship is initialized in the constructor and maintained as a field in the CustomerBasicInfo class.



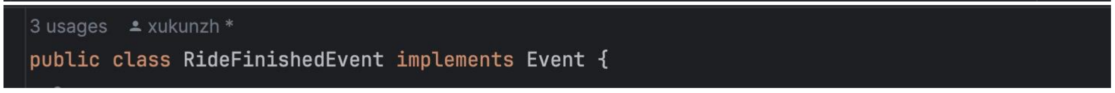
```
protected RideType rideType;

1 usage   xukunzh *
protected CustomerBasicInfo(String id, String startLocation, String destination, int orderTime, RideType rideType) {
  this.id = id;
  this.startLocation = startLocation;
  this.destination = destination;
  this.orderTime = orderTime;  // Capture the order time when the customer is created
  this.rideType = rideType;
}
```

2. Please include a code snippet showing how have you used an interface or an abstract class in your code.

Interface: This example defines an Event interface that any event type in my simulation must implement, ensuring they provide their own implementation of processEvent.



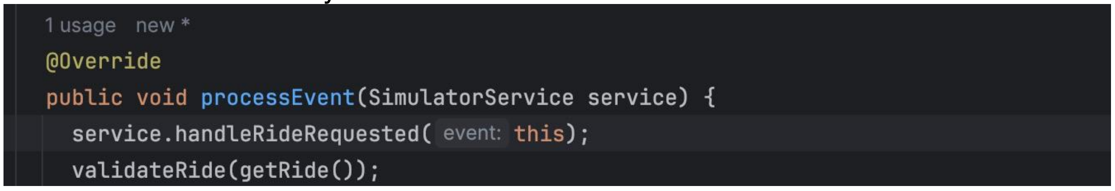
```
3 usages   2 implementations   new *
public interface Event {
  1 usage   2 implementations   new *
  void processEvent(SimulatorService service);
}
```



```
3 usages   xukunzh *
public class RideFinishedEvent implements Event {
```

3. Please include code example of a method overriding and method overloading from your code, or explain why you have not used any overloading or overriding.

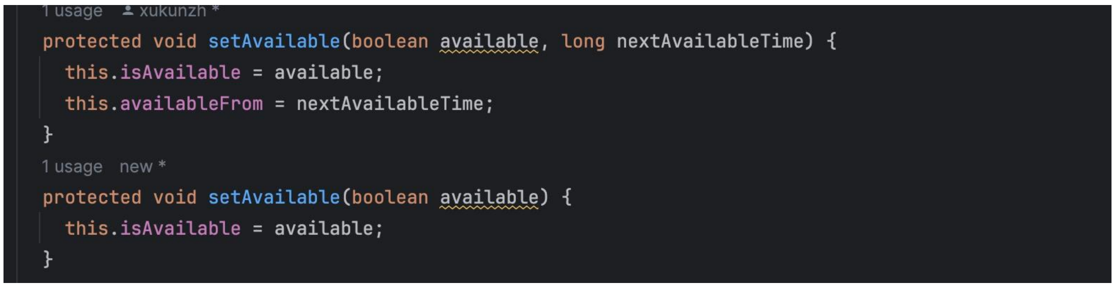
Method Overriding: I've also used method overriding, which involves redefining a method in a subclass that has been defined in the superclass or interface. I've overridden the processEvent method provided by an interface, to handle specific event-related functionality in the SimulatorService.



```
1 usage   new *
@Override
public void processEvent(SimulatorService service) {
  service.handleRideRequested( event: this);
  validateRide(getRide());
```

Explain: This method in an event handling class likely overrides a method from a base Event interface or class, providing specific implementation details for processing ride request events. This is an excellent example of polymorphism, where the system can process different types of events in different ways using a unified interface but specific implementations.

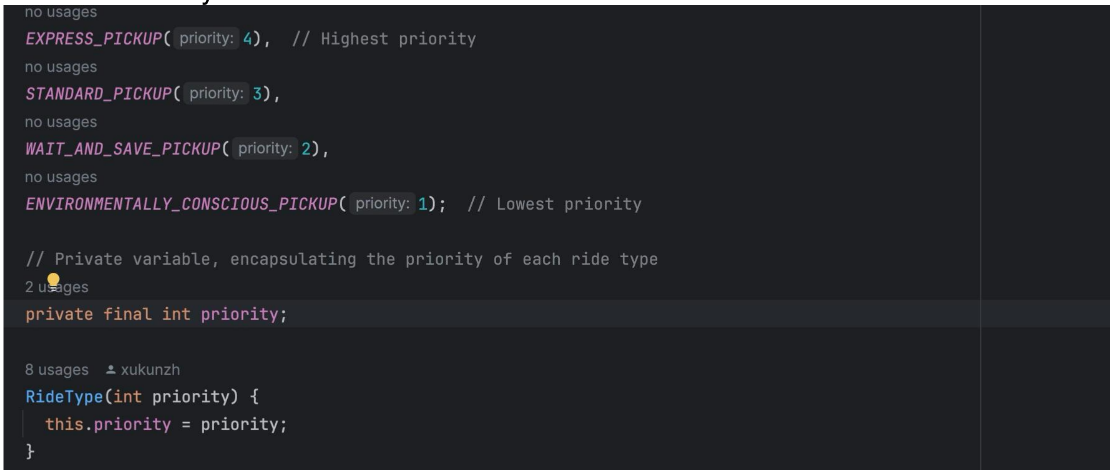
Method Overloading: I've implemented method overloading with the setAvailable method in the Driver class. Method overloading allows a class to have more than one method with the same name but different parameters.



```
1 usage   xukunzh *
protected void setAvailable(boolean available, long nextAvailableTime) {
  this.isAvailable = available;
  this.availableFrom = nextAvailableTime;
}
1 usage   new *
protected void setAvailable(boolean available) {
  this.isAvailable = available;
}
```

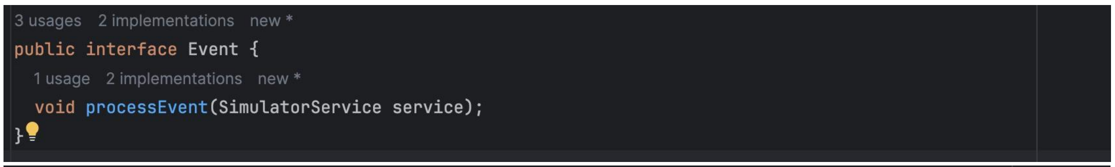
Explain: setAvailable(boolean available) sets only the availability status, allowing for quick updates where the exact time of next availability isn't crucial. On the other hand, setAvailable(boolean available, long nextAvailableTime) allows for setting both the availability and the exact time when the driver will next be available, useful for scheduling and planning.

4. Please include a code example showing how have you used encapsulation, or explain why you did not need encapsulation in your code.
Encapsulation: Encapsulation is demonstrated in RideType enum definition. The field is private, meaning it cannot be accessed directly from outside the enum. Access to it is mediated through the getPriority method, which can be designed to provide controlled and safe access to the value of priority. This is a clear implementation of encapsulation, ensuring that the internal representation of priority cannot be seen or modified directly from outside the enum.

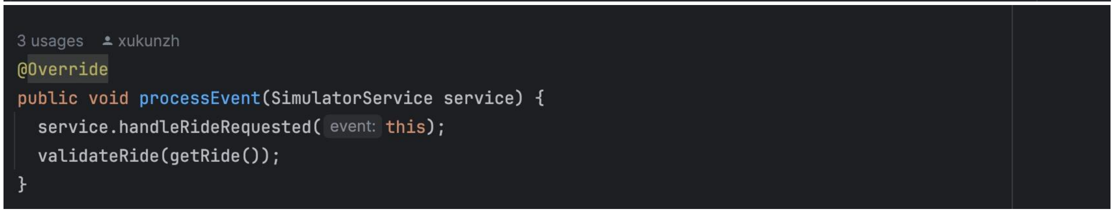


```
no usages
EXPRESS_PICKUP( priority: 4),  // Highest priority
no usages
STANDARD_PICKUP( priority: 3),
no usages
WAIT_AND_SAVE_PICKUP( priority: 2),
no usages
ENVIRONMENTALLY_CONSCIOUS_PICKUP( priority: 1);  // Lowest priority

// Private variable, encapsulating the priority of each ride type
2 usages
private final int priority;

8 usages   xukunzh
RideType(int priority) {
  this.priority = priority;
}
```

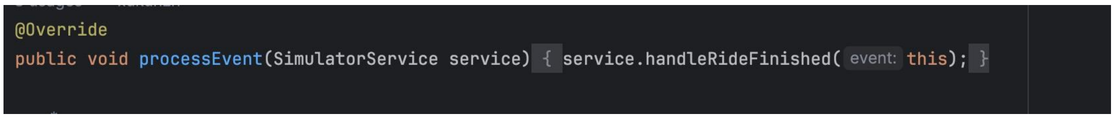
5. Please include a code example of subtype polymorphism from your code, or explain why you did not need subtype polymorphism.
Subtype polymorphism: Although I don't directly use inheritance, the RideRequestedEvent and RideFinishedEvent classes are polymorphic with respect to the Event interface.



```
3 usages   2 implementations   new *
public interface Event {
  1 usage   2 implementations   new *
  void processEvent(SimulatorService service);
}
```



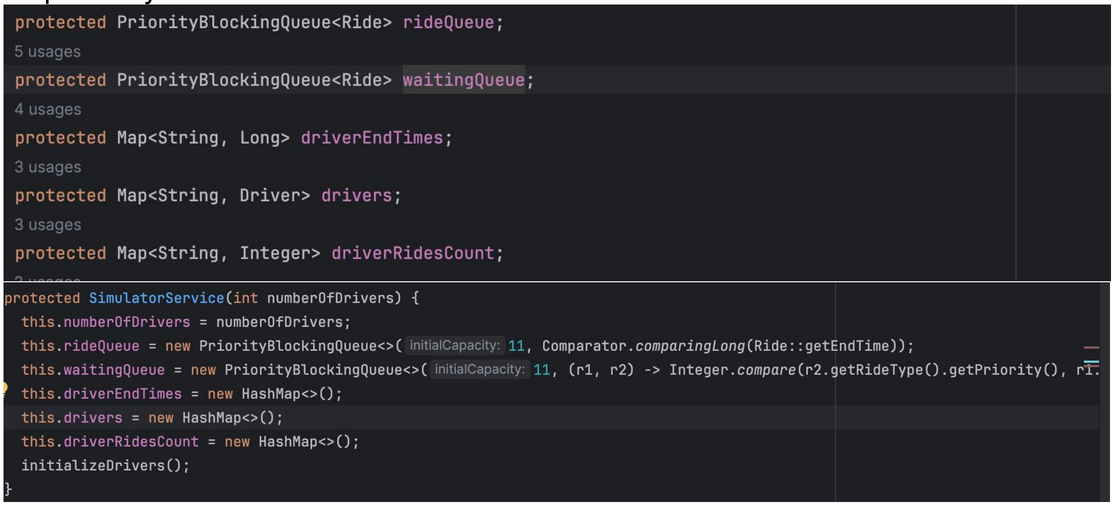
```
3 usages   xukunzh
@Override
public void processEvent(SimulatorService service) {
  service.handleRideRequested( event: this);
  validateRide(getRide());
}
```



```java
@Override
public void processEvent(SimulatorService service) { service.handleRideFinished( event: this); }
```

The Event interface defines a common behavior processEvent(), which is implemented by various event classes such as RideRequestedEvent and RideFinishedEvent. Each implementing class provides its specific behavior within the processEvent() method.

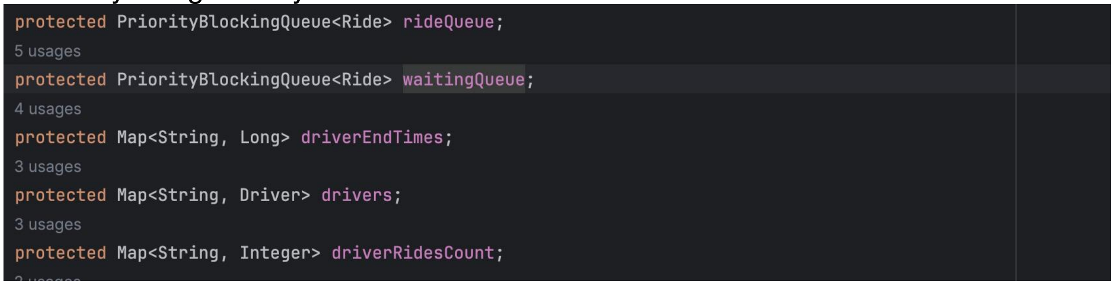
6. Please include a code snippet of generics from your code.
Generics: PriorityBlockingQueue<Ride> and HashMap<String, Driver> utilize generics to specify that they hold Ride objects and map String IDs to Driver objects, respectively.



```java
protected PriorityBlockingQueue<Ride> rideQueue;
5 usages
protected PriorityBlockingQueue<Ride> waitingQueue;
4 usages
protected Map<String, Long> driverEndTimes;
3 usages
protected Map<String, Driver> drivers;
3 usages
protected Map<String, Integer> driverRidesCount;
```

```java
protected SimulatorService(int numberOfDrivers) {
  this.numberOfDrivers = numberOfDrivers;
  this.rideQueue = new PriorityBlockingQueue<>( initialCapacity: 11, Comparator.comparingLong(Ride::getEndTime));
  this.waitingQueue = new PriorityBlockingQueue<>( initialCapacity: 11, (r1, r2) -> Integer.compare(r2.getRideType().getPriority(), r1.
  this.driverEndTimes = new HashMap<>();
  this.drivers = new HashMap<>();
  this.driverRidesCount = new HashMap<>();
  initializeDrivers();
}
```

7. Please include a code snippet showing how have you used some of the built-in data collections from the Java Collections Framework, or explain why you had no need for any data collections.
Java Collections Framework: Map<String, Driver> and Map<String, Integer>: Specifically, I've used HashMap, which provides a basic implementation of the Map interface. A HashMap offers a way of storing key-value pairs and retrieving them efficiently using the key.

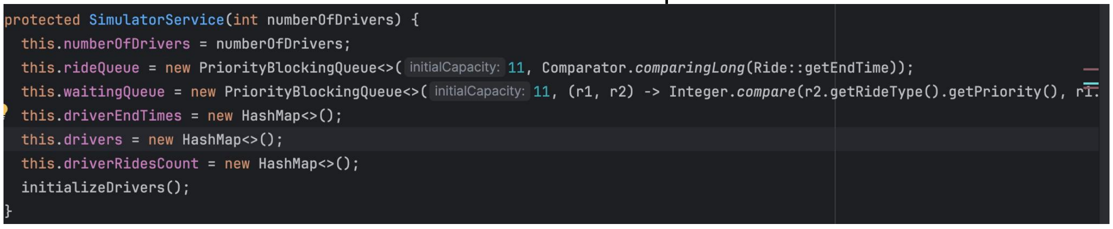


```java
protected PriorityBlockingQueue<Ride> rideQueue;
5 usages
protected PriorityBlockingQueue<Ride> waitingQueue;
4 usages
protected Map<String, Long> driverEndTimes;
3 usages
protected Map<String, Driver> drivers;
3 usages
protected Map<String, Integer> driverRidesCount;
```

8. Please include a code snippet showing how have you used interfaces Iterable and Iterator, or explain why you had no need for these two interfaces.
Explain why you had no need for these two interfaces:
I haven't implemented any data structures or collections that require iteration and I didn't have a specific need for custom iteration logic.
For example, each driver is created and stored in a map. There's no need for iteration here because you're directly creating drivers based on the loop index.
Overall, by directly handling events and their associated logic within specific methods, I've effectively avoided the need for iteration in these parts of the code. Each event is processed individually as it occurs, without the need to iterate over collections of

events.

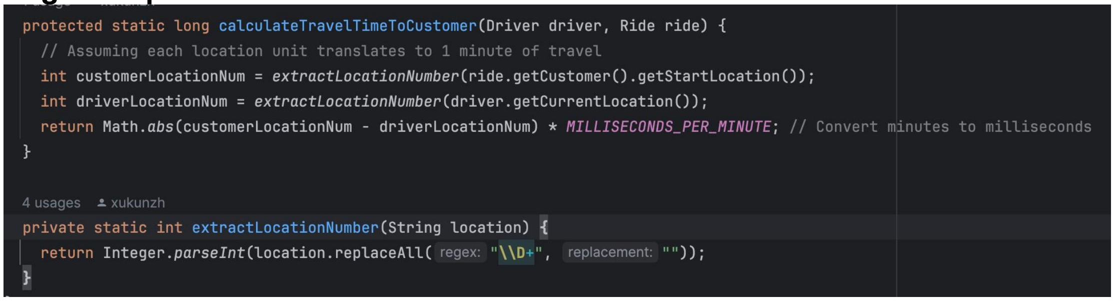
9. Please include a code snippet showing how have you used interfaces Comparable and Comparator, or explain why you had no need for these two interfaces.
Comparator: For rideQueue and waitingQueue within SimulatorService, I pass comparator lambda expressions to the constructor of PriorityBlockingQueue. This establishes the order in which the elements will be processed.



```
protected SimulatorService(int numberOfDrivers) {
  this.numberOfDrivers = numberOfDrivers;
  this.rideQueue = new PriorityBlockingQueue<>( initialCapacity: 11, Comparator.comparingLong(Ride::getEndTime));
  this.waitingQueue = new PriorityBlockingQueue<>( initialCapacity: 11, (r1, r2) -> Integer.compare(r2.getRideType().getPriority(), r1.
  this.driverEndTimes = new HashMap<>();
  this.drivers = new HashMap<>();
  this.driverRidesCount = new HashMap<>();
  initializeDrivers();
}
```

Explain why not use Comparable: By using Comparator for your queues, I provide the flexibility to order the elements in the queue as per the simulation's current needs without tying the Ride objects to a single natural ordering that Comparable would impose.

10. Please include a code snippet showing how have you used regular expressions, or explain why you had no need for it.
Regular expression: It is used within the DriverUtils class.

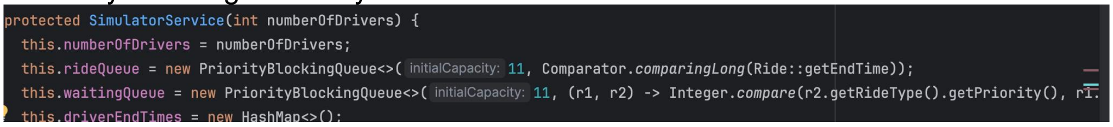


```
protected static long calculateTravelTimeToCustomer(Driver driver, Ride ride) {
  // Assuming each location unit translates to 1 minute of travel
  int customerLocationNum = extractLocationNumber(ride.getCustomer().getStartLocation());
  int driverLocationNum = extractLocationNumber(driver.getCurrentLocation());
  return Math.abs(customerLocationNum - driverLocationNum) * MILLISECONDS_PER_MINUTE; // Convert minutes to milliseconds
}

4 usages  xukunzh
private static int extractLocationNumber(String location) {
  return Integer.parseInt(location.replaceAll( regex: "\\D+",  replacement: ""));
}
```

In this method, \\D+ is a regex pattern that matches one or more non-digit characters. The replaceAll function uses this pattern to find all sequences of non-digits and replace them with an empty string. This allows the method to parse the remaining string into an integer, which represents a location number.

11. Please include a code snippet showing how have you used nested classes, or justify why you had no need for nested classes.
There does not seem to be any explicit use of **nested classes**. **Because:**
(1) Sticking with top-level classes keeps the design simple and clear.
(2) Keeping classes separate and not nested supports modularity.
(3) Since SimulatorService uses concurrent data structures, the concurrency control is already being handled by these classes. Nested classes could potentially complicate the concurrency management if not handled carefully.

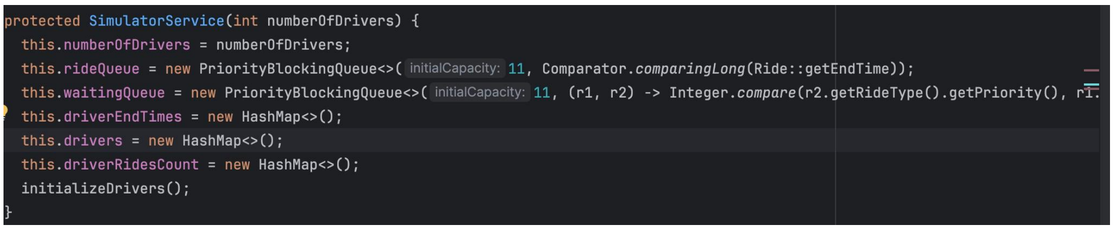
However, Create a PriorityBlockingQueue with a custom comparator via lambda may internally be using an anonymous class:



```
protected SimulatorService(int numberOfDrivers) {
  this.numberOfDrivers = numberOfDrivers;
  this.rideQueue = new PriorityBlockingQueue<>( initialCapacity: 11, Comparator.comparingLong(Ride::getEndTime));
  this.waitingQueue = new PriorityBlockingQueue<>( initialCapacity: 11, (r1, r2) -> Integer.compare(r2.getRideType().getPriority(), r1.
  this.driverEndTimes = new HashMap<>();
```

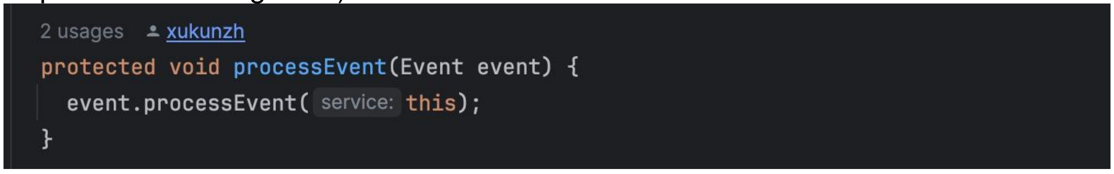
This lambda expression could be considered an anonymous class because under the hood, the Java compiler translates lambda expressions into private, static, nested classes unless they capture variables from the enclosing scope.

12. Please include code example showing how have you used components of functional programming, such as lambdas and streams, or explain why you had no need for it in your code.
Lambda expressions: They are used as the second argument in the construction of PriorityBlockingQueues. They define comparators for the queue's ordering without the need to create an anonymous inner class or implement a separate Comparator class.



```
protected SimulatorService(int numberOfDrivers) {
  this.numberOfDrivers = numberOfDrivers;
  this.rideQueue = new PriorityBlockingQueue<>( initialCapacity: 11, Comparator.comparingLong(Ride::getEndTime));
  this.waitingQueue = new PriorityBlockingQueue<>( initialCapacity: 11, (r1, r2) -> Integer.compare(r2.getRideType().getPriority(), r1.
  this.driverEndTimes = new HashMap<>();
  this.drivers = new HashMap<>();
  this.driverRidesCount = new HashMap<>();
  initializeDrivers();
}
```

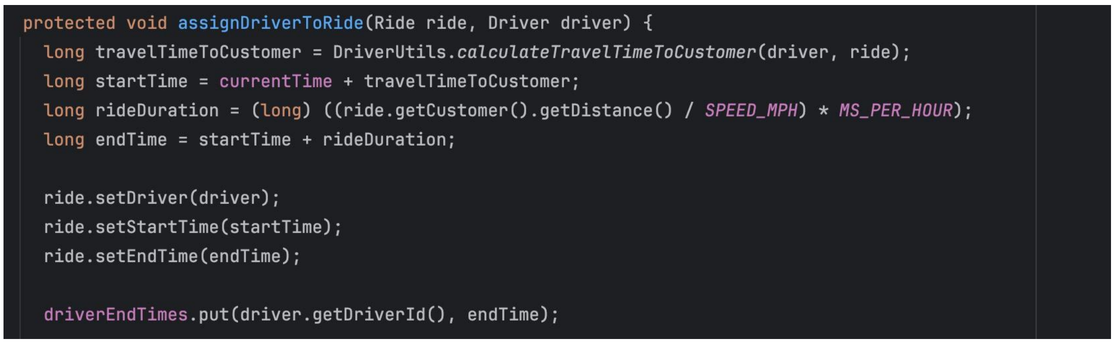
13. Please include code snippet(s) showing how have you used creational, structural and/or behavioral design patterns. Please list which design patterns have you used, or explain why you had no need for design patterns in your solution.
Creational patterns: Not used. My object creation is straightforward and doesn't involve a complex setup or configuration.
Structural patterns: Not used. My application doesn't have a complex hierarchy and no need to change the interfaces of classes and objects to different clients.
Behavioral Patterns: I used the behavioral pattern, specifically the Command pattern, through the processEvent method, where Event can be considered a command object. Each Event encapsulates a request to perform an operation (like handling ride requests or finishing rides).



```
2 usages  xukunzh
protected void processEvent(Event event) {
  event.processEvent( service: this);
}
```

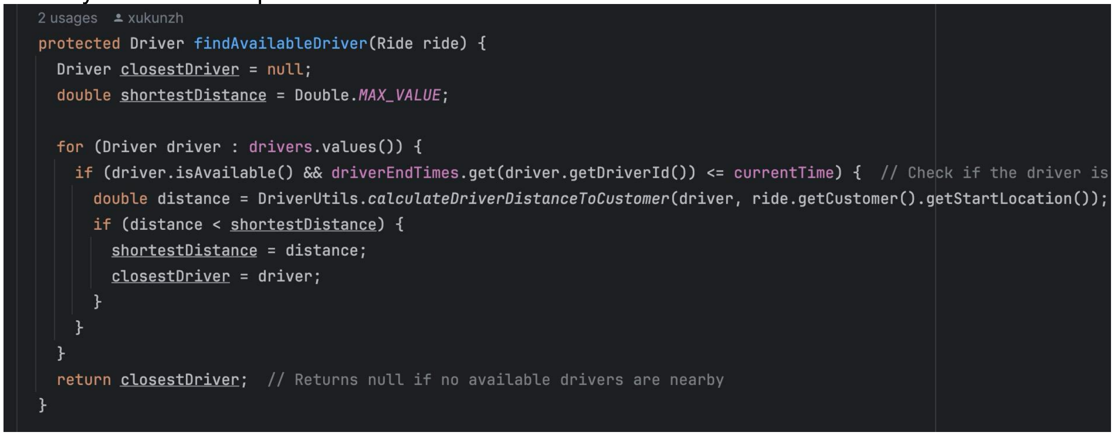
14. Please include code snippets showing examples of MVC architecture, or justify why you had no need for MVC architecture in your design.
Why MVC Might Not Be Used:
(1) No User Interface: There's no graphical interface that requires updating or interaction, which eliminates the need for the View and Controller components of MVC.
(2) Batch Processing: The simulator handles tasks automatically based on predefined scenarios, so there's no real-time interaction that would require a Controller to manage user inputs.
(3) Complexity: Implementing MVC might add unnecessary complexity if it doesn't require a clear separation of concerns.

15. Please include code examples showing data and stamp coupling in your code.
Data Coupling: assignDriverToRide takes Ride and Driver objects as parameters and uses them to set up a ride. This is a classic example of data coupling, where the function directly uses data passed to it through its parameters to perform its operations.



```
protected void assignDriverToRide(Ride ride, Driver driver) {
  long travelTimeToCustomer = DriverUtils.calculateTravelTimeToCustomer(driver, ride);
  long startTime = currentTime + travelTimeToCustomer;
  long rideDuration = (long) ((ride.getCustomer().getDistance() / SPEED_MPH) * MS_PER_HOUR);
  long endTime = startTime + rideDuration;

  ride.setDriver(driver);
  ride.setStartTime(startTime);
  ride.setEndTime(endTime);

  driverEndTimes.put(driver.getDriverId(), endTime);
```

Stamp Coupling: findAvailableDriver receives a Ride object and uses only the getCustomer().getStartLocation() part of it to find the nearest available driver. This is an example of stamp coupling because the function requires a complex object (Ride), but only actually uses a small piece of the information contained within it.



```
2 usages  xukunzh
protected Driver findAvailableDriver(Ride ride) {
  Driver closestDriver = null;
  double shortestDistance = Double.MAX_VALUE;

  for (Driver driver : drivers.values()) {
    if (driver.isAvailable() && driverEndTimes.get(driver.getDriverId()) <= currentTime) {  // Check if the driver is
      double distance = DriverUtils.calculateDriverDistanceToCustomer(driver, ride.getCustomer().getStartLocation());
      if (distance < shortestDistance) {
        shortestDistance = distance;
        closestDriver = driver;
      }
    }
  }
  return closestDriver;  // Returns null if no available drivers are nearby
}
```