SWEN30002 Project 1 Report

Overview of the design

The key design point for our team to adjust this project is adding new robot types, charges, fees and maintenance to the log. In the meanwhile changed the structure of the system and applied the GRASP pattern during the design process to make feature design for this system as simple as possible.

Design Processing:

Our team believes that Simulation is the main class that will trigger the entire operating system. Simulation class has the responsibility of the active maillGernerator to keep generating mail, also the active Automaill class and mailPool class let them add robots to the system and load mail to the robot.

During the adding of the new robot types, we decided to apply the polymorphism pattern and create an abstract class called Robot, and three subclasses, RegularRobot, FastRobot, and BulkRobot. As they share most of the attributes and methods, we only need to override some methods within different robots to achieve their special functionality.

During the fee generation process, Firstly we were thinking of writing a fee calculation in the automail class as it contains the robot list and is associated with the simulation class. However, this might cause high cohesion of the automail class as it needs to deal with multiple tasks now and even more tasks when adding more classes like wifiModem in the future. To reduce high cohesion and to achieve protected variation, we create a special class called FeeCalculator to generate fees needed for calculation and share and reduce the work of the automail class. Nevertheless, this would also result in high coupling as the simulation class is connected with excessive classes, fortunately, simulation only initializes and assigns jobs to each of these classes, it doesn't process overmuch work inside.

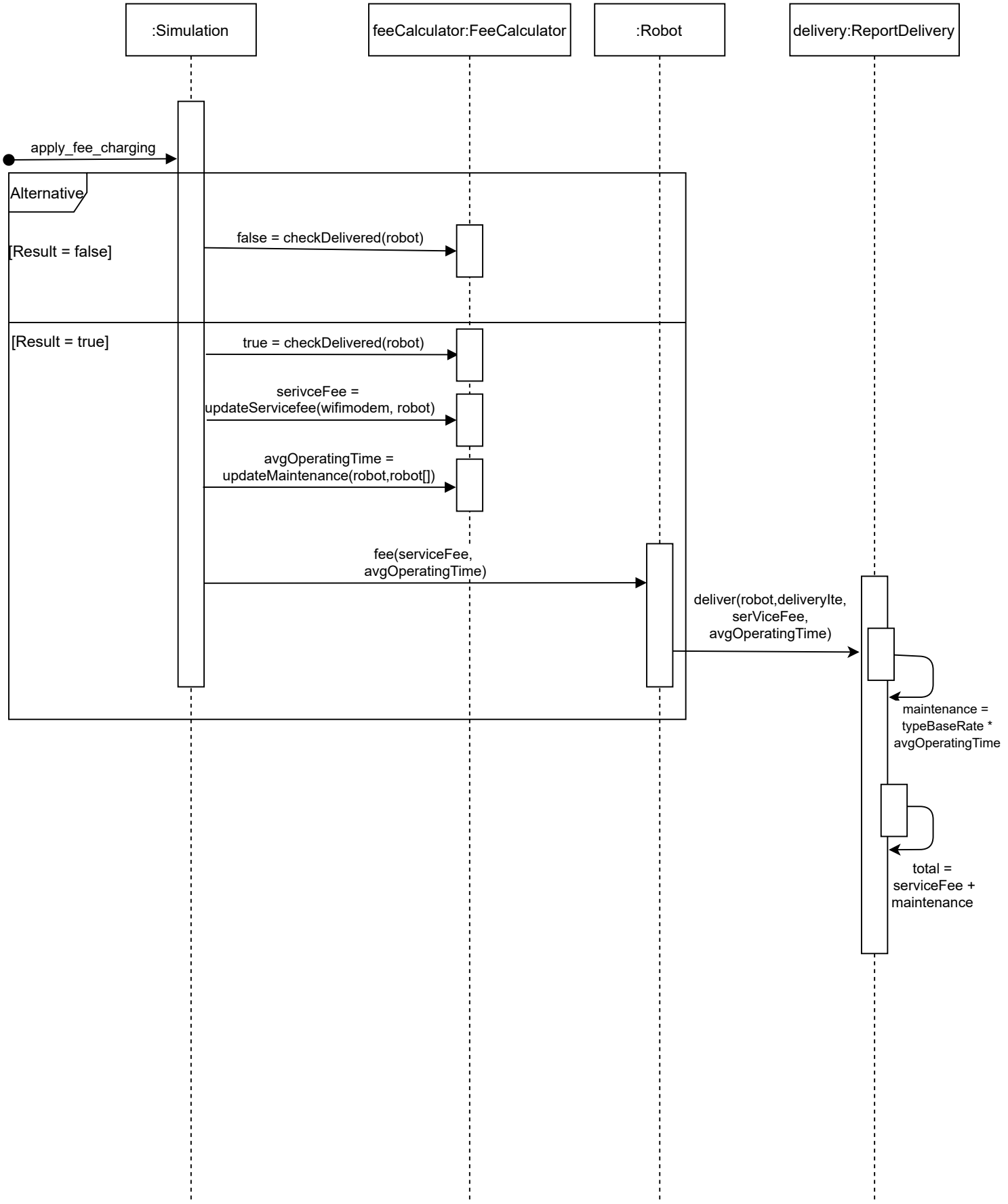
Diagram:

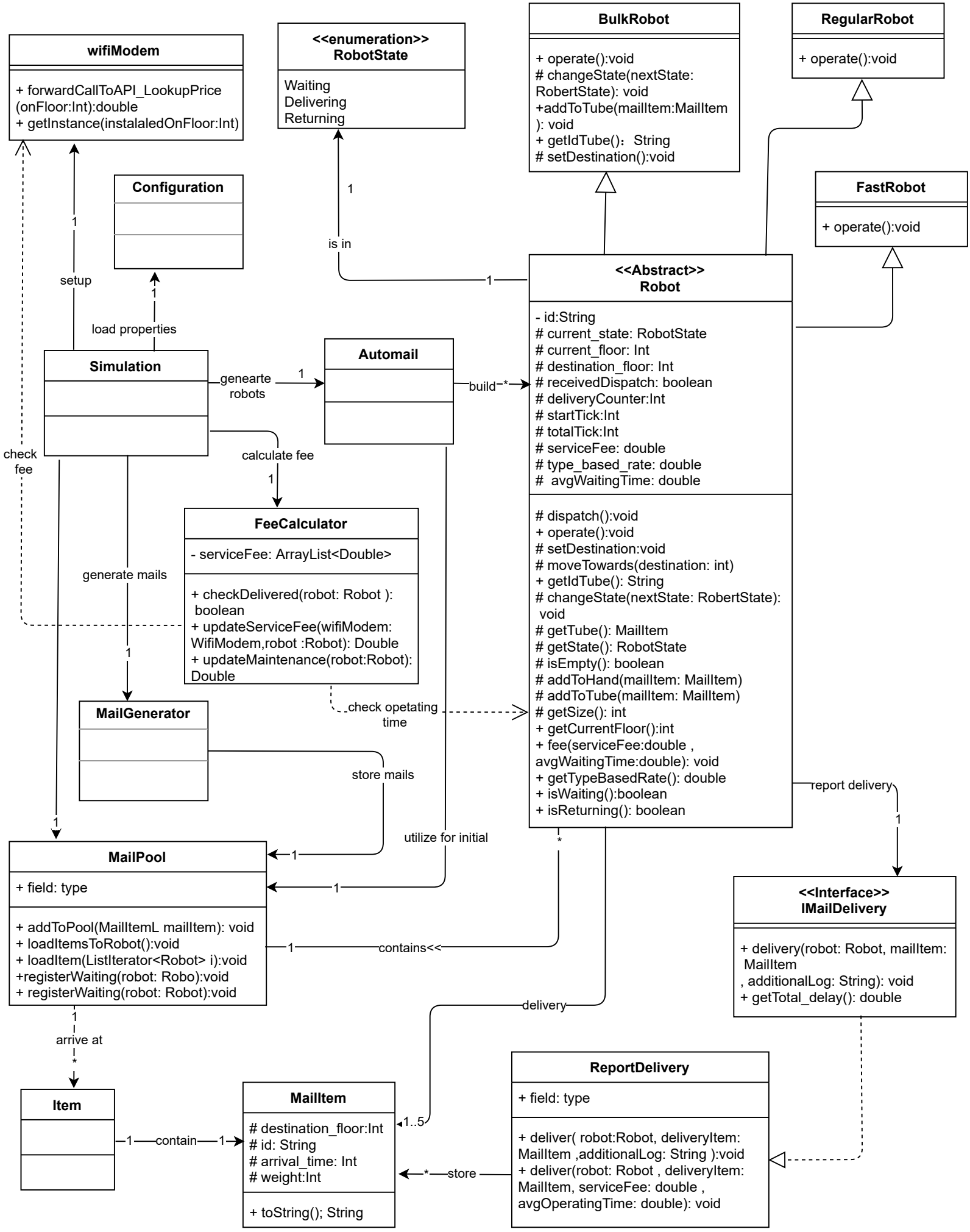
For the Design Class Diagram, we included most of the classes within the src file. Some classes are empty with no attribute and method as we didn't change it at all, the existence of those classes is to show the multiple associations with the simulation class. All the attributes and classes that were created for the implementation, including methods that need to be overwritten(same name as the method in the parent class), are displayed on the graph.

For the Design Sequence Diagram, we illustrate how systems make use of different classes to generate total fees after delivery by using an alternative frame to demonstrate each step of the calculation.

Code change:
For the code implementation, our team tried to minimize the number of new classes added. We made robot class as an abstract class, and then we can extend any type of robot from the Robot class. The reason why our team chose abstract class instead of interface is that different types of robots will share some behaviours (methods). for example move, AddToHand, etc. so we only need to override those methods which need to be changed based on their type. But for the interface, every child class needs to implement the method in the Parent class, once a new feature is added to one type of robot, the other type has to add this feature as well, even those robots do not have this feature. In the future, if any new type of robot is added, we only need to extend the Robot class and add specific behaviours which are not shared with other robots to this new type of robot.

In terms of the charge capability, our team created a new class named as FeeCaculator to handle all the operations regarding the fee calculation. We designed the updateServiceFee method and updateMaintenance method. In updateSeriveFee WifiModem and robot will be inputted into this method. Firstly make an ArrayList length of 13(number of floors + 1) therefore index 1 stands for first-floor index 12 stands for the highest floor and we initialize all the values in this list as 0.0 which means the previous fee is zero. Later we check if the robot is on the destination floor with case ‘delivering’ If it is then we use the forwardCallToAPI_LookupPrice method to get the current service fee from the destination floor. Because service fee will only be increased but not decreased for the same floor if the current service fee is greater than the previous fee then the current service fee should replace the old service fee in the list and new service fee should be returned. If WifiModem fails to connect and gives -1 to the new service fee then the list should not be updated and the old service fee should be returned.
In updateMaintenance two parameters robot and robots(the list of robots) will be passed into the function. We use getTypeBasedRate() method to find the type of the current robot. For all robots with the same type we count the total number of time ticks they moved. At the end returns the total number of time ticks divided by the number of same type robots which is the average operating times. After got both average operating times and service fee, total fee can be calculated as total fee = service fee + average operating times * the base rate of the robot(the rate value is stored in each robot class).
This also leads to much more convenience for future change or add on.


:Simulation
feeCalculator:FeeCalculator
:Robot
delivery:ReportDelivery
apply_fee_charging
Alternative
[Result = false]
false = checkDelivered(robot)
[Result = true]
true = checkDelivered(robot)
serivceFee =
updateServicefee(wifimodem, robot)
avgOperatingTime =
updateMaintenance(robot,robot[])
fee(serviceFee,
avgOperatingTime)
deliver(robot,deliveryIte,
serViceFee,
avgOperatingTime)
maintenance =
typeBaseRate *
avgOperatingTime
total =
serviceFee +
maintenance


wifiModem
+ forwardCallToAPI_LookupPrice (onFloor:Int):double
+ getInstance(instalaledOnFloor:Int)
<<enumeration>> RobotState
Waiting
Delivering
Returning
BulkRobot
+ operate():void
# changeState(nextState: RobertState): void
+addToTube(mailItem:MailItem): void
+ getIdTube(): String
# setDestination():void
RegularRobot
+ operate():void
FastRobot
+ operate():void
Configuration
1
setup
1
load properties
1
is in
1
<<Abstract>> Robot
- id:String
current_state: RobotState
current_floor: Int
destination_floor: Int
receivedDispatch: boolean
deliveryCounter:Int
startTick:Int
totalTick:Int
serviceFee: double
type_based_rate: double
avgWaitingTime: double
dispatch():void
+ operate():void
# setDestination:void
# moveTowards(destination: int)
+ getIdTube(): String
# changeState(nextState: RobertState): void
# getTube(): MailItem
# getState(): RobotState
# isEmpty(): boolean
# addToHand(mailItem: MailItem)
# addToTube(mailItem: MailItem)
# getSize(): int
+ getCurrentFloor():int
+ fee(serviceFee:double , avgWaitingTime:double): void
+ getTypeBasedRate(): double
+ isWaiting():boolean
+ isReturning(): boolean
Simulation
genearte robots
1
Automail
build
*
check fee
calculate fee
1
FeeCalculator
- serviceFee: ArrayList<Double>
+ checkDelivered(robot: Robot): boolean
+ updateServiceFee(wifiModem: WifiModem,robot :Robot): Double
+ updateMaintenance(robot:Robot): Double
generate mails
1
check opetating time
MailGenerator
store mails
1
MailPool
+ field: type
+ addToPool(MailItemL mailItem): void
+ loadItemsToRobot():void
+ loadItem(ListIterator<Robot> i):void
+registerWaiting(robot: Robo):void
+ registerWaiting(robot: Robot):void
1
utilize for initial
1
1
contains<<
*
report delivery
1
<<Interface>> IMailDelivery
+ delivery(robot: Robot, mailItem: MailItem , additionalLog: String): void
+ getTotal_delay(): double
1
arrive at
*
delivery
Item
1
contain
1
MailItem
destination_floor:Int
id: String
arrival_time: Int
weight:Int
+ toString(); String
1..5
*
store
ReportDelivery
+ field: type
+ deliver(robot:Robot, deliveryItem: MailItem ,additionalLog: String):void
+ deliver(robot: Robot , deliveryItem: MailItem, serviceFee: double , avgOperatingTime: double): void