Project 1: Robomail Revision — Group 35

Glenn Phillips 820624 — Eldar Kurmakaev 1028326 — Michael Lowe 911365
SWEN30006 - Software Modelling and Design
University of Melbourne

May 8, 2020

Introduction

The task is to implement an updated system for Robotic Mailing Solutions Inc. (RMS) so that they can make use of the new robot capability. This new capability will allow robots, when in caution mode, to have a new pair of hands specially designed to carry fragile items.
In this implementation we wanted to add in the new concept that RMS has envisioned into the system whilst also maintaining a concise and coherent system that follows the main software modelling patterns. Our solution involves us adding a few classes and refactoring original classes whilst incorporating common software modelling patterns to keep classes and methods focused and easy to use.
We first created a Domain Class Diagram in order for us to get a better understanding of the system and allow others to easily understand exactly how we have gone about implementing the new system capabilities. The Domain Diagram shows all the key concepts and real-situation objects that will be used in the system, whilst keeping it easy to read which supports better communication and understanding. From the Domain Class, we formed a Design Class Diagram which goes into greater detail and describes all the relationships between the classes, as well as the methods and attributes stored in each one. Following this, the System Sequence diagram that we have designed clearly shows the process that is involved when delivering a fragile item. This diagram shows the input and output of the system under discussion along with when and where we call methods.

Discussion

In this section, we will go over the design problems that we faced and the patterns that we used to then overcome each problem. Below is a discussion of these problems and our decision making process in choosing a solution for each.

Robot based problems

Putting a robot into caution mode whilst preserving existing functionality

Firstly, we looked for an information expert that already has information necessary to put robots into the mode in order to assign responsibility. To put robots into a caution mode it is necessary to: a) have access to instances of robots - Automail class has it; and b) have information of whether caution mode is activated - Simulation class has it. Because there are two equal information experts we had to resolve the issue by looking for a high-cohesion and low-coupling solution. For cohesion we saw that the Simulation class would be impacted negatively, as setting up robots would dilute its focus on creating simulation. On the other hand, the Automail class only has one existing function that initialises robots, which is related to setting up the mode of robots and creating it. Therefore

the Automail was chosen by using the information expert, low-coupling and high-cohesion design patterns..
Following this, it was necessary to extend functionality in accordance with the requirement of variations. For this purpose the polymorphism pattern is appropriate. We introduced a new class CautiousRobot that extends class Robot. This gives benefit, in terms of possibility, to extend the number of modes in the future by extending class Robot without negatively impacting existing functionality while reusing large parts of code. Additionally, new functionality can be changed and updated without impact on the base functionality because it is encapsulated within a separate class. An alternative approach would be to use the strategy pattern and separate existing behaviour and new behaviour in new classes that implements a common interface, which would allow for greater flexibility at the expense of greater architectural complexity. Our team judged that this approach would not be justified and would put our team in a common beginners pitfall of over future proofing.

Implementing a pair of special arms in addition to the existing arms and tube

We tried two options for the functionality of the installation of two pairs of hands. The first one was to use the polymorphism pattern and create an abstract Limb class and subclasses Hand, SpecialHand, Tube. This would allow for the extensibility of future design but result in greater complexity. As previously mentioned, a future extension/ evolution points will come mainly in the behavior of the system and not hardware. Our team evaluated that this use of polymorphism belongs outside of the scope of the brief and would increase coupling unnecessarily. Therefore we implemented an extra pair of hands as a new specialHand attribute of MailItem type in the Cautious Robot class. This is less extensible but results in lower coupling due to lesser number of visible classes and is in accordance with the brief.

MailItem based problems

Loading fragile items into special arms

We looked for an information expert to assign the responsibility of loading special hands while the cautious mode is activated. This functionality needs two pieces of information: a)Whether an item is fragile - MailPool and Robot have this information; and b) Whether the system is in the caution mode - this information is within Simulation. Since there is no clear information expert we decided to evaluate options from the perspective of cohesion and coupling. Implementing new functionality to the Simulation class would decrease its cohesion and create new dependencies to Mailpool and Robot classes thus increasing coupling. Robot class is a more viable candidate as it would need to have only one more dependency on Simulation to know which mode is activated. On the other hand, introducing the discussed functionality would decrease cohesion as Robot is not currently responsible for the process of loading mail from the mail pool. We implemented the new functionality in the MailPool class, as it would require only one more dependency to the Simulation class. In addition it would continue to maintain high cohesion, since it is already responsible for deciding which robot limb the item has to be placed in. However, the loadRobot method became quite large so to increase readability and keep the code neat, we split the loadRobot into two additional methods with each one focusing on either handling a fragile item or a normal item.
An alternative approach that we considered was creating a new FragileMailPool class that would extend the original MailPool and would focus on organising and loading robots when fragile mode is enabled. We decided not to continue with this model however, as it proved to be unnecessarily complex compared to the implementation we went with.

Preserving the behaviour of carrying normal items regardless of caution mode

In the case of delivering a non-fragile item with caution mode turned off the Robot class is able to execute the pre-existing delivering method with unchanged behaviour. If instead delivering a fragile item with caution mode turned on the CautiousRobot class calls upon the delivering method within the Robot superclass but implements the appropriate overrides from within the CautiousRobot subclass. For example, whilst the delivering method from the superclass is executed, the instance of the moveTowards method that is called within this is actually the overridden method from the CautiousRobot subclass. This implementation follows the polymorphism design pattern which enables this “plug and play” functionality. An alternative implementation entirely within a single Robot class would not have this flexibility and would result in many lines of repeated code.

Wrapping and unwrapping with the special arms

Prior to implementing this functionality we identified key information and classes which are partial information experts, so that we could then decide on a final information expert. It is necessary to know: a) whether item is fragile and it’s destination - boolean attribute within MailItem class; b) if it is time to wrap/unwrap - stored in an enum state variable in the CautiousRobot class; and finally, c) Whether the cautious mode is on - stored in the Simulation class. We disregarded the Simulation class as a candidate to be considered, as the information expert as it has least of information. Since Robot and MailItem are both equal partial information experts we resolved this tie by evaluating possibilities from the perspective of the impact on coupling. Assigning responsibility to MailItem class would result in necessity to provide backward visibility to Robot which would result in increased coupling. While assigning responsibility to Robot would not result in a new dependency as it already has visibility to MailItem class. Thus we assigned responsibility to the CautiousRobot class, which is reflected in it’s wrap() and unwrap() methods and enum states.

Handling how floors are stored and used within the system

Questions raised for this section included: Should buildings be a part of the Automail system? Should floors be an attribute in Building? Or should it be a seperate class? We came to the conclusion that floors needed to be a new class which would be initialised and stored as an array in Building class. Building class was identified to be a creator since it has necessary information for the initialisation of Floor; It has the lowest floor, mail room location and number of floors. An alternative idea was to initialise the Floors instance in an Automail instance, as it would closely use floors to provide information to robots. However, as it does not have necessary information and this approach would negatively impact coupling and cohesion we ruled out this option.

Maintaining sole access to the delivery floor while unwrapping and delivering a fragile item

It was decided that the CautiousRobot class would control the movement of robots to and from an individual floor. The Floor class has information about the number of robots on the floor, and whether there is active unwrapping. The CautiousRobot class has key information whether there is a fragile item to be delivered to the floor and has existing visibility to the Floor class. Since CautiousRobot had more information than the Floor class it was assigned responsibility according to the information expert pattern. On the other hand, a negative side effect was that this solution impacted the coherence of Robot by introducing a new responsibility that is only moderately related to the primary function of Robot class of delivering items. We achieved this implementation through a booking scenario, as seen in the System Sequence Diagram, where a Robot would book a floor if available once it has reached its destination, which would then close the floor to all other robots. Once the robot has fully finished the delivery of the fragile item, it would then call the floor again to remove the booking on the floor, allowing other robots to continue passing through.

Statistic based problems

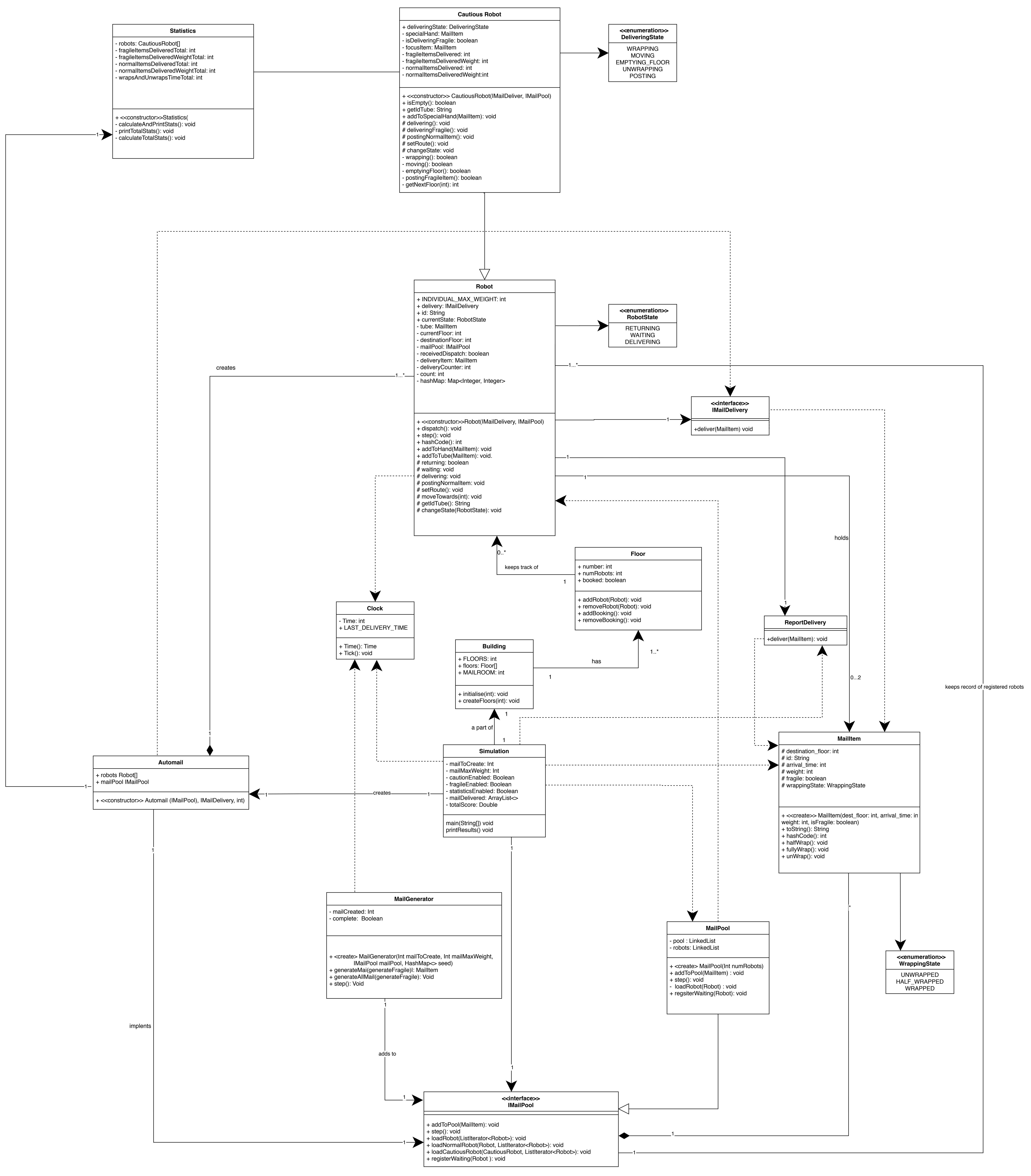
Implementing recording, calculating and printing of statistics

As required by the brief, our implementation of the functionality has to record stats if the robot is in caution mode. It must also allow for additional data gathering in future, related to robots in caution mode or potentially other classes. These requirements immediately ruled out any implementation involving the Robot class and narrowed our options down to the CautiousRobot and ReportDelivery classes. In considering the information expert design pattern, it was logical that calculating an individual CautiousRobot's statistics should be performed by the CautiousRobot class as it has all necessary information. While the ReportDelivery class does have information about delivered items, it does not hold information about robots.
After deciding this, we then had to decide which class should be responsible for collating these individual statistics. The class that aggregates CautiousRobots is Automail, thus has access to all CautiousRobots instances and its data. Therefore, Automail was given the responsibility to record and print stats. As we evaluated the impact of this change on cohesion, it became clear that due to the expectation of introducing new features, as mentioned in the brief, Automail would become bloated with functionality that is not cohesive, as it would become responsible for the logic of calculating the stats. Taking into consideration that data must be also printed using I/O, it gave us additional reason to separate statistics in a separate class called Statistics with high representational decomposition and thus not dilute coherence of the Automail class. We also created a Loggable interface that was to be used as input for Statistics class in order to future proof and allow for tracking robots of any type that implement the interface. However we dismissed this Loggable approach as the brief emphasises that the system should focus only on cautious robots. Thus the Statistics class only takes an input of robot instances of CautiousRobot class.

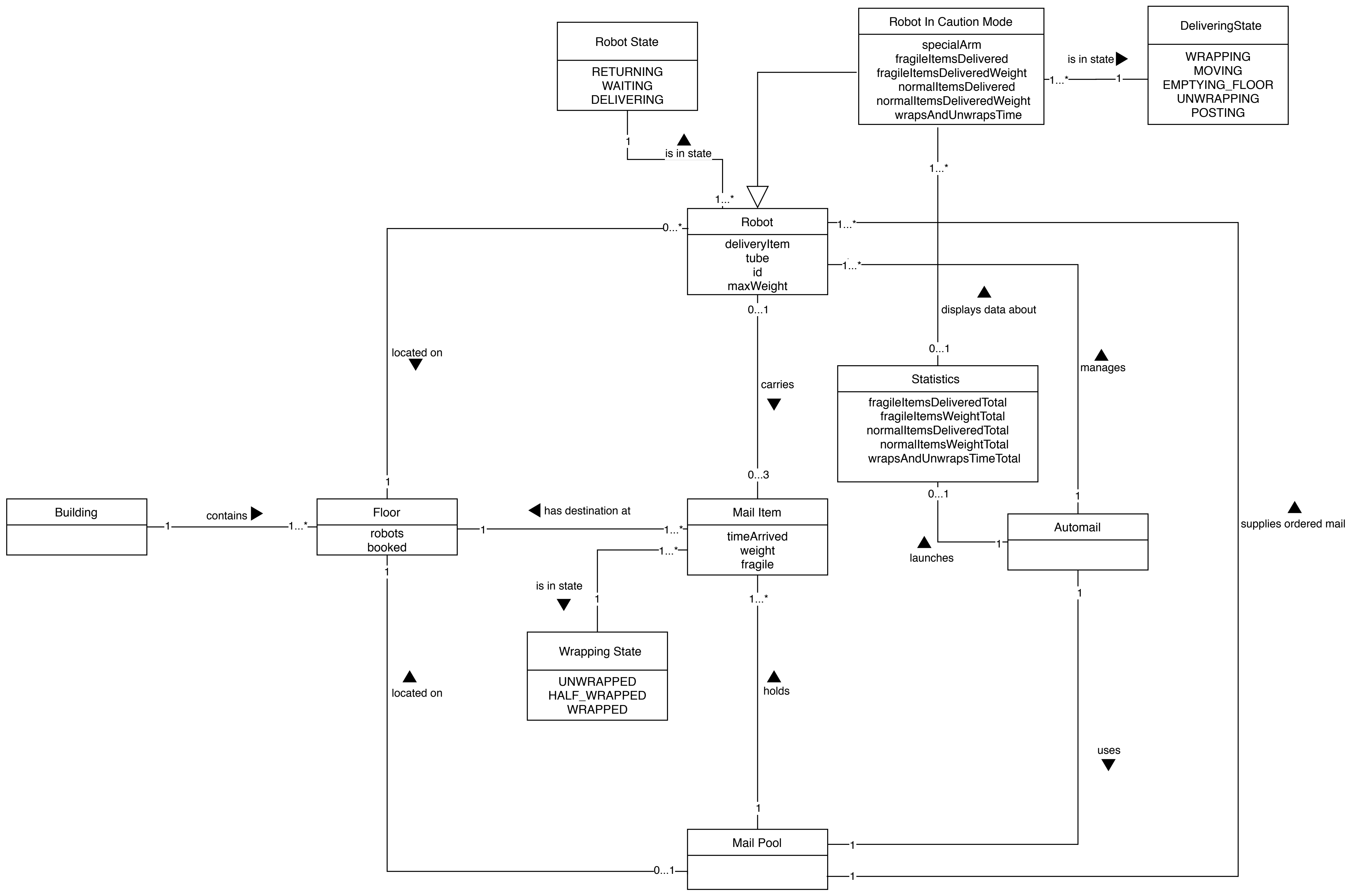
Conclusion

After carefully working through each problem we have incorporated a working system into the original software. This was achieved by working with common design patterns to come up with a solution to the problem initially asked by RMS.

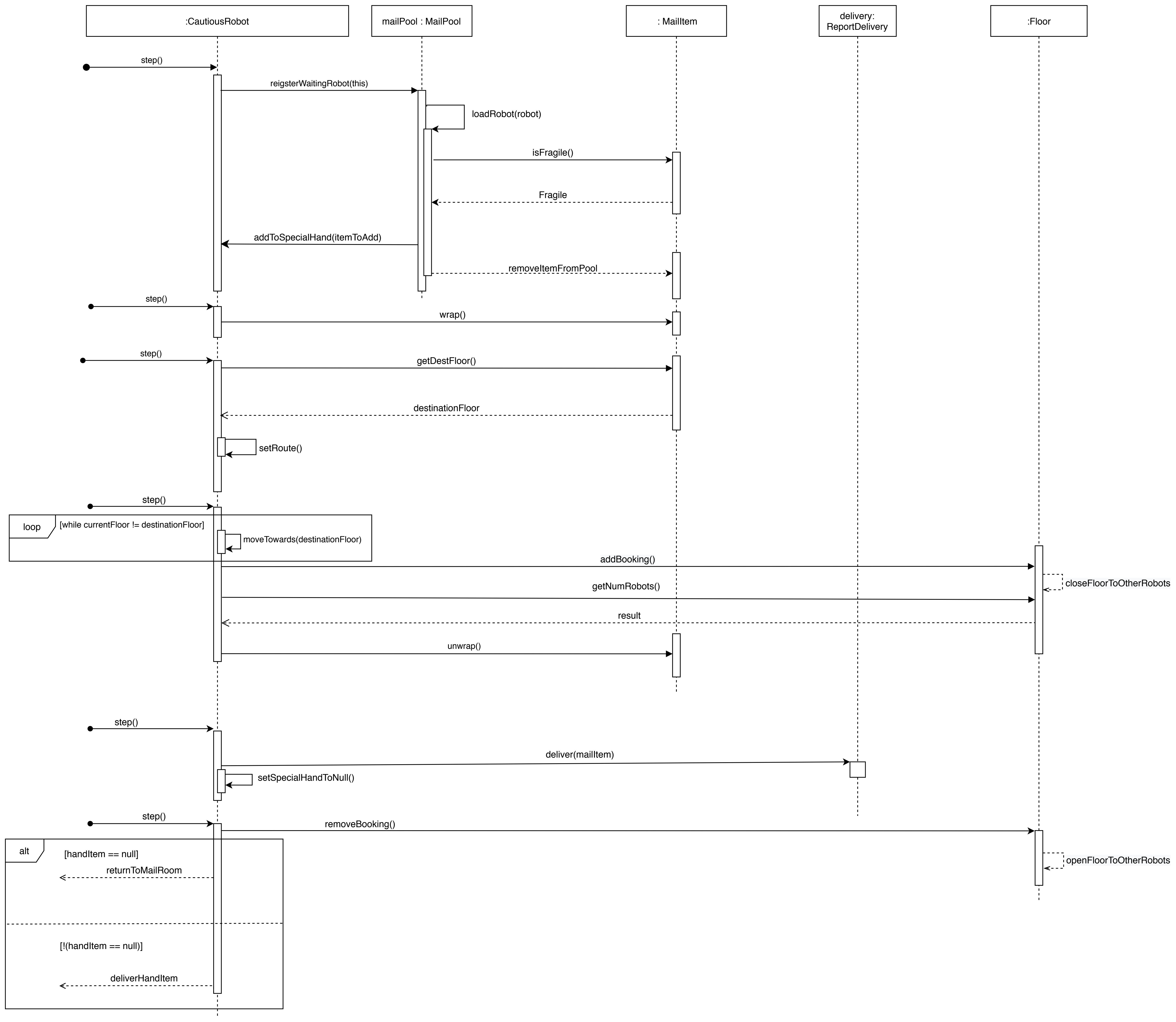
Automail Design Class Diagram - Group 35


Statistics
- robots: CautiousRobot[]
- fragileItemsDeliveredTotal: int
- fragileItemsDeliveredWeightTotal: int
- normalItemsDeliveredTotal: int
- normalItemsDeliveredWeightTotal: int
- wrapsAndUnwrapsTimeTotal: int
+ <<constructor>>Statistics(
- calculateAndPrintStats(): void
- printTotalStats(): void
- calculateTotalStats(): void
Cautious Robot
+ deliveringState: DeliveringState
- specialHand: MailItem
- isDeliveringFragile: boolean
- focusItem: MailItem
- fragileItemsDelivered: int
- fragileItemsDeliveredWeight: int
- normalItemsDelivered: int
- normalItemsDeliveredWeight:int
+ <<constructor>> CautiousRobot(IMailDeliver, IMailPool)
+ isEmpty(): boolean
+ getIdTube: String
+ addToSpecialHand(MailItem): void
# delivering(): void
# deliveringFragile(): void
# postingNormalItem(): void
# setRoute(): void
# changeState: void
- wrapping(): boolean
- moving(): boolean
- emptyingFloor(): boolean
- postingFragileItem(): boolean
- getNextFloor(int): int
<<enumeration>>
DeliveringState
WRAPPING
MOVING
EMPTYING_FLOOR
UNWRAPPING
POSTING
Robot
+ INDIVIDUAL_MAX_WEIGHT: int
+ delivery: IMailDelivery
+ id: String
+ currentState: RobotState
- tube: MailItem
- currentFloor: int
- destinationFloor: int
- mailPool: IMailPool
- receivedDispatch: boolean
- deliveryItem: MailItem
- deliveryCounter: int
- count: int
- hashMap: Map<Integer, Integer>
+ <<constructor>>Robot(IMailDelivery, IMailPool)
+ dispatch(): void
+ step(): void
+ hashCode(): int
+ addToHand(MailItem): void
+ addToTube(MailItem): void.
# returning: boolean
# waiting: void
# delivering: void
# postingNormalItem: void
# setRoute(): void
# moveTowards(int): void
# getIdTube(): String
# changeState(RobotState): void
<<enumeration>>
RobotState
RETURNING
WAITING
DELIVERING
creates
1...*
1...*
<<interface>>
IMailDelivery
+deliver(MailItem) void
1
1
1
holds
0..*
keeps track of
1
Floor
+ number: int
+ numRobots: int
+ booked: boolean
+ addRobot(Robot): void
+ removeRobot(Robot): void
+ addBooking(): void
+ removeBooking(): void
Clock
- Time: int
+ LAST_DELIVERY_TIME
+ Time(): Time
+ Tick(): void
Building
+ FLOORS: int
+ floors: Floor[]
+ MAILROOM: int
+ initialise(int): void
+ createFloors(int): void
has
1..*
1
1
ReportDelivery
+deliver(MailItem): void
0...2
keeps record of registered robots
a part of
1
1
Simulation
- mailToCreate: Int
- mailMaxWeight: Int
- cautionEnabled: Boolean
- fragileEnabled: Boolean
- statisticsEnabled: Boolean
- mailDelivered: ArrayList<>
- totalScore: Double
main(String[]) void
printResults() void
Automail
+ robots Robot[]
+ mailPool IMailPool
+ <<constructor>> Automail (IMailPool), IMailDelivery, int)
1
creates
1
1
1
MailItem
# destination_floor: int
# id: String
# arrival_time: int
# weight: int
# fragile: boolean
# wrappingState: WrappingState
+ <<create>> MailItem(dest_floor: int, arrival_time: in
weight: int, isFragile: boolean)
+ toString(): String
+ hashCode(): int
+ halfWrap(): void
+ fullyWrap(): void
+ unWrap(): void
1
1
MailGenerator
- mailCreated: Int
- complete: Boolean
+ <create> MailGenerator(Int mailToCreate, Int mailMaxWeight,
IMailPool mailPool, HashMap<> seed)
+ generateMai(generateFragile)l: MailItem
+ generateAllMail(generateFragile): Void
+ step(): Void
*
MailPool
- pool : LinkedList
- robots: LinkedList
+ <create> MailPool(Int numRobots)
+ addToPool(MailItem) : void
+ step(): void
- loadRobot(Robot) : void
+ regsiterWaiting(Robot): void
<<enumeration>>
WrappingState
UNWRAPPED
HALF_WRAPPED
WRAPPED
implents
1
adds to
1
1
1
<<interface>>
IMailPool
+ addToPool(MailItem): void
+ step(): void
+ loadRobot(ListIterator<Robot>): void
+ loadNormalRobot(Robot, ListIterator<Robot>): void
+ loadCautiousRobot(CautiousRobot, ListIterator<Robot>): void
+ registerWaiting(Robot): void
1
1
1

Automail Domain Class Diagram - Group 35



Assigning Fragile Item to Robot & Delivering Sequence Diagram - Group 35


:CautiousRobot
mailPool : MailPool
: MailItem
delivery: ReportDelivery
:Floor
step()
reigsterWaitingRobot(this)
loadRobot(robot)
isFragile()
Fragile
addToSpecialHand(itemToAdd)
removeItemFromPool
step()
wrap()
step()
getDestFloor()
destinationFloor
setRoute()
step()
loop
[while currentFloor != destinationFloor]
moveTowards(destinationFloor)
addBooking()
closeFloorToOtherRobots
getNumRobots()
result
unwrap()
step()
deliver(mailItem)
setSpecialHandToNull()
step()
removeBooking()
openFloorToOtherRobots
alt
[handItem == null]
returnToMailRoom
[!(handItem == null)]
deliverHandItem