To begin our modelling, we created a Domain Model which encapsulated the required conceptual classes for the system, as seen in Figure 1. The requirements for the new system appeared to be extending the functionality of the current system, by adding statistics tracking of mail items, extending the existing functionality for modem lookups, and adding charge calculation capability.

For the Design Model, new software classes were necessary to maintain High Cohesion and Protected Variation and to allow for code reuse. Within our Design Model phase, it was clear that additional classes would be necessary to:

- Avoid ReportDelivery becoming too bloated for calculation of the charge at delivery time
- Allow code to be reused for both actual charge calculation at delivery and the expected charge during sorting of the mail pool
- Calculate output statistics
- Provide an interface around the Wifi Modem for Protected Variation

Charge calculation has a specific set of logic in this specification, to be done for every delivery, however as this must be undertaken in the mail pool for expected charge as well as upon delivery, neither MailPool nor ReportDelivery are the Information Expert for this responsibility. Although Simulation could serve as the Expert here, this would lead to large amounts of bloating and Low Cohesion in the Simulation class.

As no other solution seemed appropriate to allow the reuse of code between expected and delivery-time charge calculation, we opted to create a new class by Pure Fabrication. We created the software class DeliveryCharge to handle all calculations relating to the delivery charge. The use of Pure Fabrication allowed DeliveryCharge to be Highly Cohesive, prevented bloating in ReportDelivery and allowed better code reuse between MailPool and ReportDelivery.

Had Simulation taken responsibility for charge calculations by the Expert pattern, Simulation would have had Low Cohesion, bloating and increased Coupling.

For the responsibility of modem lookups, it was stated that Robot is responsible for this task upon delivery, however we abstracted this to a new class inside DeliveryCharge for multiple reasons. Firstly, due to interacting with the potentially unstable external system Wifi Modem, we created a ModemLookup class by Indirection, Pure Fabrication and Protected Variation. This ModemLookup class is responsible for calling lookups to the modem, the external system, and hence creates an Indirection layer between the systems. ModemLookup class also maintains High Cohesion for itself and DeliveryCharge, as DeliveryCharge does not have to also do modem lookups. This interaction between ReportDelivery, DeliveryCharge and ModemLookup is shown in the Design Sequence Diagram on page 3.
This class is also responsible for tracking the successful and failed lookups, as it is the Expert in this case, working closely with WifiModem. The alternative to this was tracking these lookups in Simulation or DeliveryCharge, however that would unnecessarily bloat these classes.
Finally, ModemLookup was assigned as a nested static class within DeliveryCharge. This is by the Creator pattern, as by the design of our system, ModemLookup is only called within the context of DeliveryCharge, meaning DeliveryCharge compositely aggregates ModemLookup, and therefore

should be created by DeliveryCharge. This also serves to reduce the Coupling between Simulation and ModemLookup, and maintain High Cohesiveness in Simulation.

In terms of toggling charge display, as ReportDelivery is responsible for calling DeliveryCharge when the charge must be calculated, it logically follows that ReportDelivery should be responsible for the flow of control when charge display is toggled. ReportDelivery interprets the state of the system, i.e. the charge display toggle, where if the charge display is on, the charge calculations are delegated to DeliveryCharge. Then ReportDelivery need only output the log, with the extra charges if necessary, maintaining High Cohesion in the class.

Finally, for the responsibility of statistics calculation, as DeliveryCharge was the Expert for all the necessary statistics such as modem lookups, service fees etc., it only follows that DeliveryCharge should be responsible for this tracking. Being a static class, DeliveryCharge can incrementally track these statistics upon each delivery, to then be printed by the outputStats() function at the end of the Simulation, maintaining the separation of calculation and printing. This maintains the High Cohesion of Simulation and ReportDelivery, by delegating all logic for statistics tracking to DeliveryCharge.

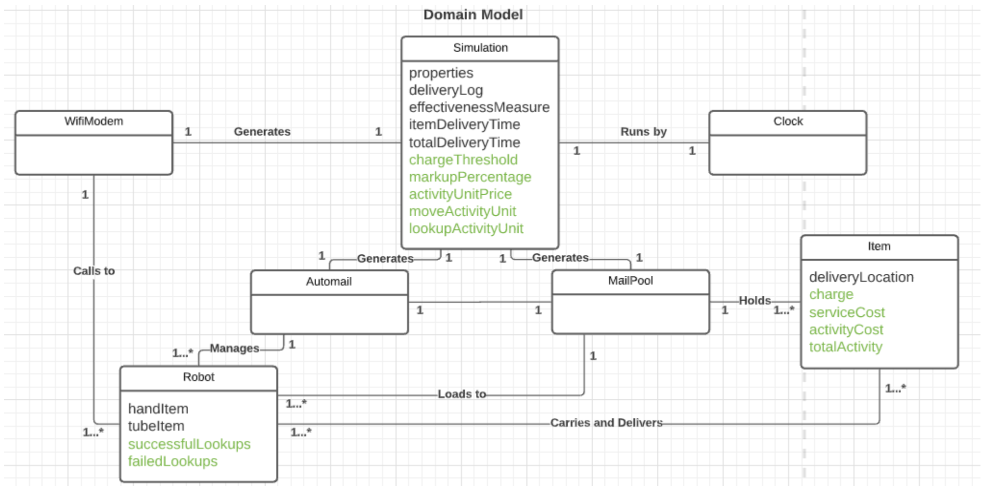
Unfortunately, adding this responsibility may lead to DeliveryCharge becoming slightly less cohesive, as it now takes on additional responsibility for tracking. This shows mostly when expected charges are calculated, as this is handled by specifying if the call is for an estimated charge. This was deemed preferable to having ReportDelivery track these statistics, as ReportDelivery is not the Expert for these statistics. Because they are calculated individually within DeliveryCharge, using ReportDelivery would increase its coupling significantly with DeliveryCharge, whereas the current implementation maintains Low Coupling for both classes.

In order to provide a reusable interface for calculating the delivery charge as used by both the DeliveryCharge and MailPool classes, the IMailCharge interface was created. IMailCharge exists as a way to pass a reference to DeliveryCharge into the constructor of MailPool.

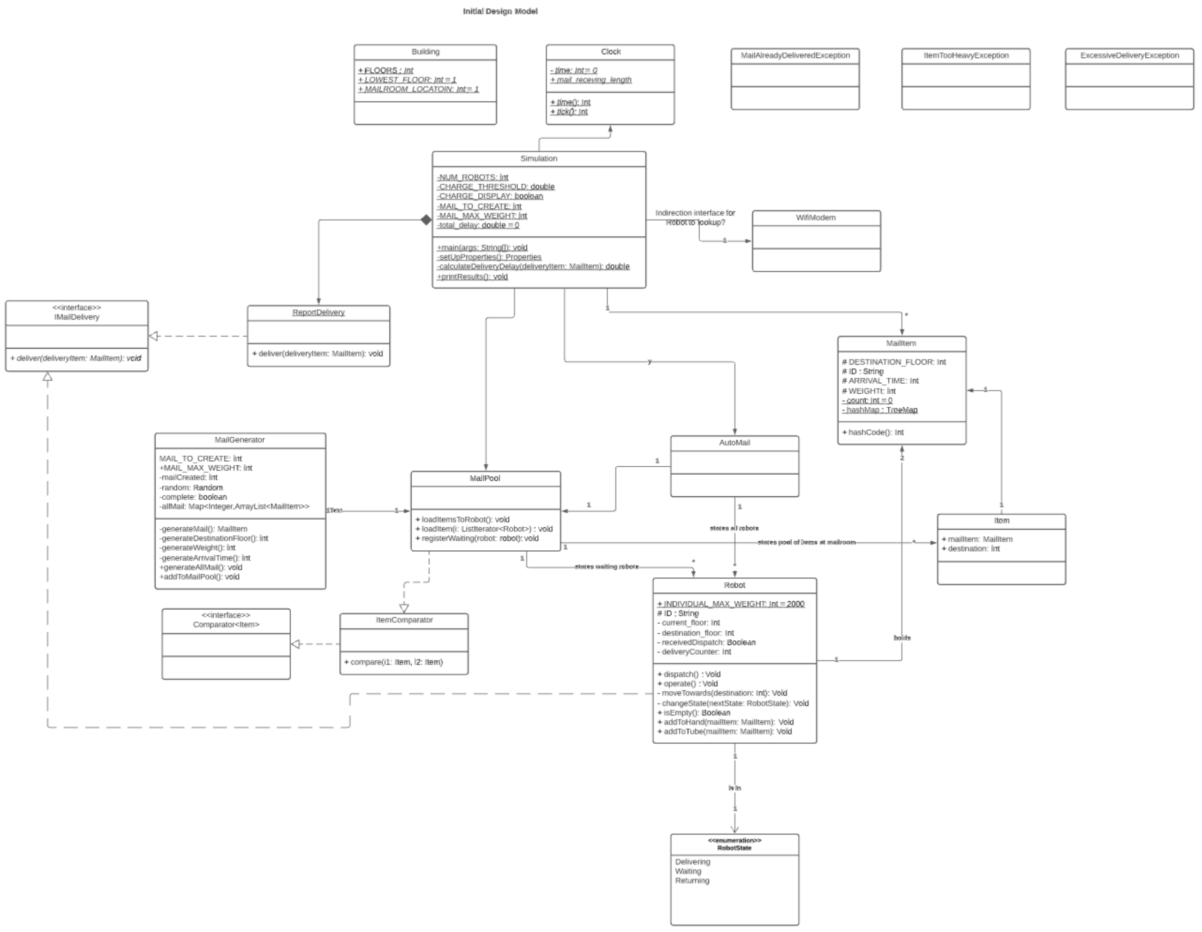
For the prioritisation of items, the expected charge of an item is stored in its Item class, and the ItemComparator class was extended to add functionality for this while maintaining MailPool's High Cohesion. To account for future variations of the system, we decided to assign the priority as an integer value. For the new system in the specification, the maximum priority is 1 as there is only one charge threshold. However the system may be extended in future to also include, for example, a weight threshold which would make the maximum priority 2. As arrival time and weight were included as evolution points for future versions of the system, by Protected Variation we included these variables in the Item class for extendability.

Group 3: Daniel Digby, Jasper Robbins, Fiona Zhu
Green Vo's Workshop 11

Domain Model:

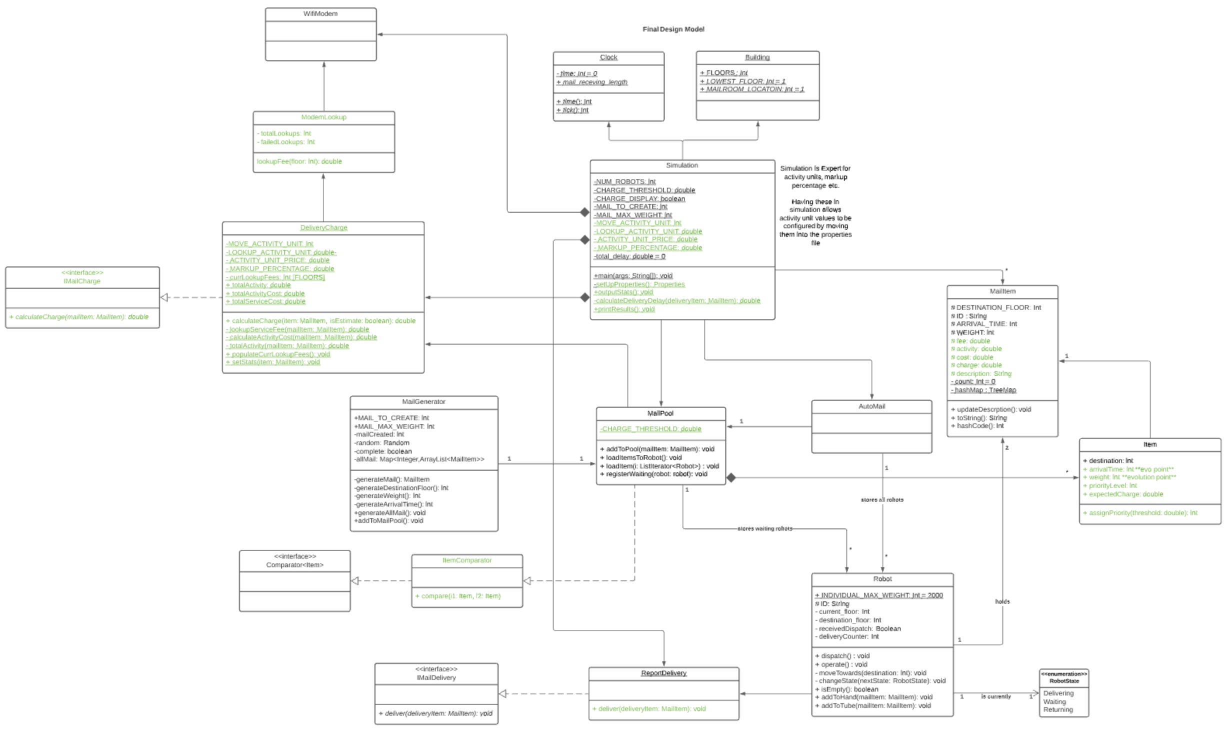


Initial Design Model:

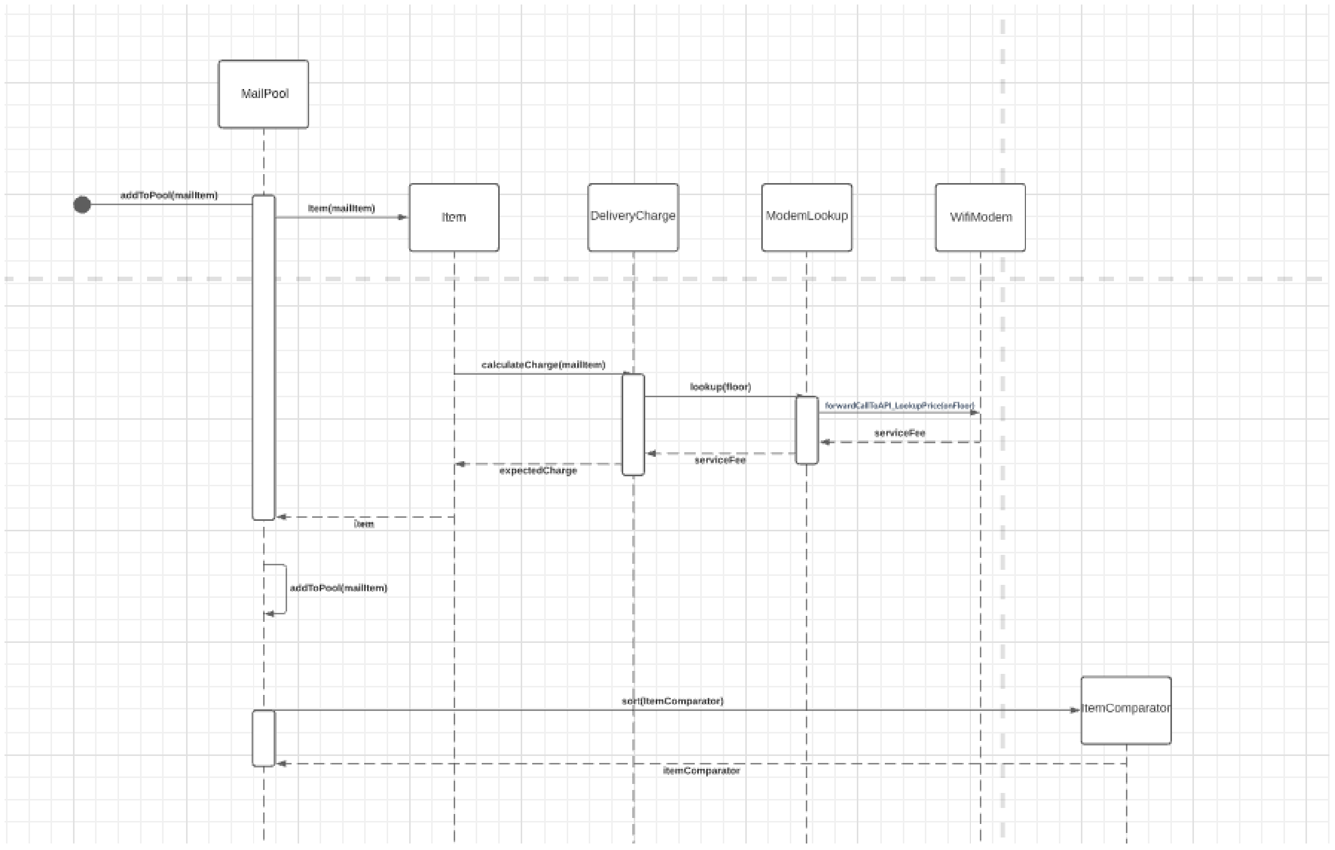


Group 3: Daniel Digby, Jasper Robbins, Fiona Zhu
Green Vo's Workshop 11

Final Design Model:

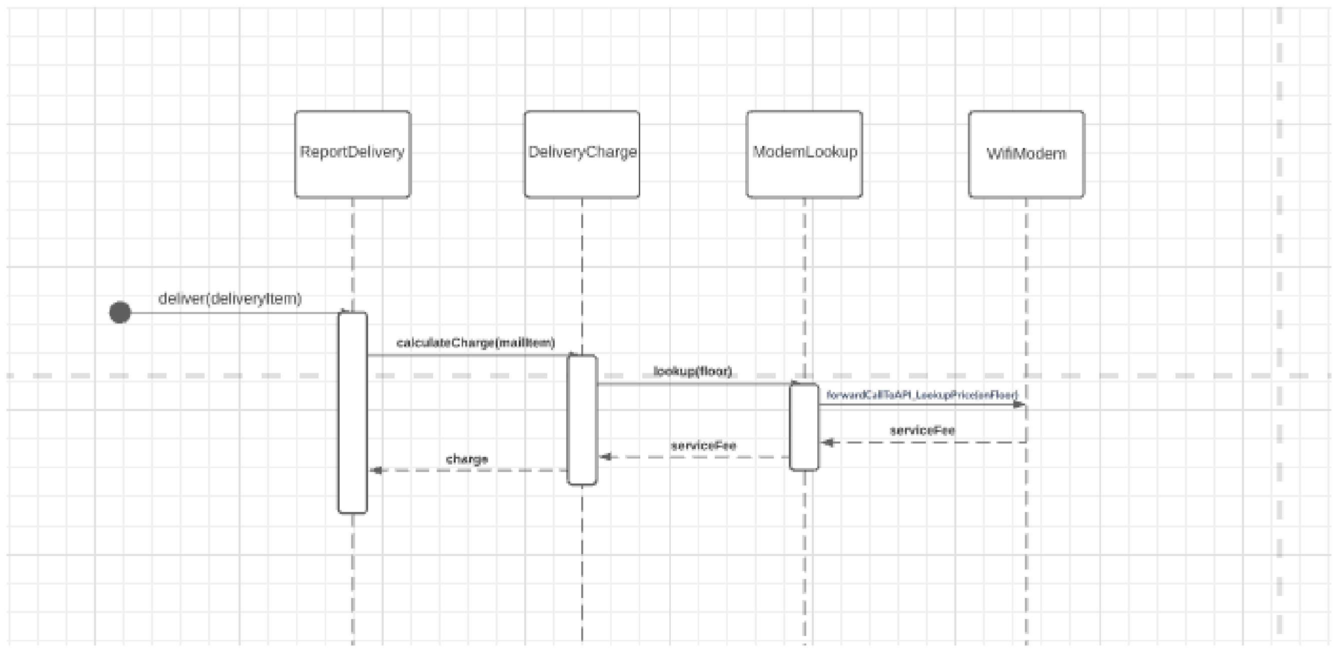

Final Design Model
WifiModem
ModemLookup
- totalLookups: Int
- failedLookups: Int
lookupFee(floor: Int): double
DeliveryCharge
-MOVE_ACTIVITY_UNIT: Int
-LOOKUP_ACTIVITY_UNIT: double-
- ACTIVITY_UNIT_PRICE: double
- MARKUP_PERCENTAGE: double
- currLookupFees: Int [FLOORS]
+ totalActivity: double
+ totalActivityCost: double
+ totalServiceCost: double
+ calculateCharge(item: MailItem, isEstimate: boolean): double
- lookupServiceFee(mailItem: MailItem): double
- calculateActivityCost(mailItem: MailItem): double
- totalActivity(mailItem: MailItem): double
+ populateCurrLookupFees(): void
+ setStats(item: MailItem): void
<<interface>>
IMailCharge
+ calculateCharge(mailItem: MailItem): double
Clock
- time: Int = 0
+ mail_receving_length
+ time(): Int
+ tick(): Int
Building
+ FLOORS,: Int
+ LOWEST_FLOOR: Int = 1
+ MAILROOM_LOCATOIN: Int = 1
Simulation
-NUM_ROBOTS: Int
-CHARGE_THRESHOLD: double
-CHARGE_DISPLAY: boolean
-MAIL_TO_CREATE: Int
-MAIL_MAX_WEIGHT: Int
-MOVE_ACTIVITY_UNIT: Int
-LOOKUP_ACTIVITY_UNIT: double
- ACTIVITY_UNIT_PRICE: double
- MARKUP_PERCENTAGE: double
-total_delay: double = 0
+main(args: String[]): void
-setUpProperties(): Properties
+outputStats(): void
-calculateDeliveryDelay(deliveryItem: MailItem): double
+printResults(): void
Simulation is Expert for activity units, markup percentage etc.
Having these in simulation allows activity unit values to be configured by moving them into the properties file
MailItem
# DESTINATION_FLOOR: Int
# ID : String
# ARRIVAL_TIME: Int
# WEIGHT: Int
# fee: double
# activity: double
# cost: double
# charge: double
# description: String
- count: Int = 0
- hashMap : TreeMap
+ updateDescrption(): void
+ toString(): String
+ hashCode(): Int
Item
+ destination: Int
+ arrivalTime: Int **evo point**
+ weight: Int **evolution point**
+ priorityLevel: Int
+ expectedCharge: double
+ assignPriority(threshold: double): Int
MailGenerator
+MAIL_TO_CREATE: Int
+MAIL_MAX_WEIGHT: Int
-mailCreated: Int
-random: Random
-complete: boolean
-allMail: Map<Integer,ArrayList<MailItem>>
-generateMail(): MailItem
-generateDestinationFloor(): Int
-generateWeight(): Int
-generateArrivalTime(): Int
+generateAllMail(): void
+addToMailPool(): void
MailPool
-CHARGE_THRESHOLD: double
+ addToPool(mailItem: MailItem): void
+ loadItemsToRobot(): void
+ loadItem(i: ListIterator<Robot>) : void
+ registerWaiting(robot: robot): void
AutoMail
stores all robots
stores waiting robots
holds
<<interface>>
Comparator<Item>
ItemComparator
+ compare(i1: Item, i2: Item)
Robot
+ INDIVIDUAL_MAX_WEIGHT: Int = 2000
# ID: String
- current_floor: Int
- destination_floor: Int
- receivedDispatch: Boolean
- deliveryCounter: Int
+ dispatch() : void
+ operate() : void
- moveTowards(destination: Int): void
- changeState(nextState: RobotState): void
+ isEmpty(): boolean
+ addToHand(mailItem: MailItem): void
+ addToTube(mailItem: MailItem): void
<<enumeration>>
RobotState
Delivering
Waiting
Returning
is currently
<<interface>>
IMailDelivery
+ deliver(deliveryItem: MailItem): void
ReportDelivery
+ deliver(deliveryItem: MailItem): void
1
*
2

Add to Mail Pool SSD:



Group 3: Daniel Digby, Jasper Robbins, Fiona Zhu
Green Vo’s Workshop 11

Report Delivery SSD


ReportDelivery
DeliveryCharge
ModemLookup
WifiModem
deliver(deliveryItem)
calculateCharge(mailItem)
lookup(floor)
forwardCallToAPI_LookupPriceIonFloor()
serviceFee
serviceFee
charge