SWEN30006 Project 1
Workshop [Tues16:15] Team 04

Tony Lang (1263784), Alex Nguyen (1356649)

1: Refactoring and Analysis of Current Design

The current design of the DS system presents several concerns based on GRASP principles, which leads to scalability and maintainability issues. This is most notably seen through 2 factors, the bloating of MailRoom and the rigidity in logic through both Robot and Letters. First, in the provided source code, MailRoom has the responsibilities of both controlling the robot movement as well as dealing with the letter logic; the combination of these two different purposes violates the high cohesion principle. Furthermore, the logic and the implementation is designed strictly for the classes of robots and the letters under the Cycling mode.

To begin, we started off by making sure that MailRoom’s sole purpose was to manage the MailItems, moving the robot control logic to both Simulation and the Robot itself. This defines the roles of our classes quite nicely, leading to high cohesion. Simulation is in charge of initialising the properties and establishing the environment and calling the tick function from all simulation units. Robot is in charge of data strictly related to the robot and the movement of itself and Building and Building Grid are already fine by themselves. Furthermore, by dividing the data/methods like this, we follow the Information Expert Pattern. Robot has information about its own position, so it should contain the responsibilities for moving itself.

Through doing this, Simulation is able to serve as the hub of the system. It is the class running the simulation, so by extension it directly aggregates its actors. Thus by the Creator Pattern, Simulation maintains its responsibility for instantiating actors. As Simulation points to both mailroom and lists of robots, as per the Indirection pattern, we are able to place our robotDispatch method in Simulation, in order to reduce direct coupling between Robot and MailRoom. Furthermore, by giving each robot a reference to Simulation, the robots are always able to find the information they need to make decisions.

In anticipation of implementing the additional mail items and needing different types of robots depending on future delivery modes, we have created the abstract superclasses Robot and MailItem in order to promote polymorphism and code reuse. Robot contains all the general movement methods for any robot, such as move, place and return. As such, we are able to create the subclass CyclingRobot extending Robot, which has the logic required to distribute mail items in the cycling mode. Likewise for MailItem, we can create the subclass letter

which currently does not contain any more information (Letters are quite basic). It is also worth noting that Letters has the function getWeight() which returns 0.
We mentioned prior that the logic for each tick of each Simulation Unit (just robot) was moved to the simulation unit themselves. As such, the Robot class implements the interface Tickable. As it stands tickable only has one method which is called tick. It is very important that we are able to implement ticking differently for different types of robots, since the behaviour of a robot varies drastically depending on what it is built for.

An example of this can be seen in Figure 2, where a FlooringRobot is calling the tick method. Whilst this is not a robot in the simple design, the general logic will also apply to CyclingRobot, as there are similar checks required to work out what state the robot is currently in. The method tick involves several checks to see what state the robot is in. For example, it begins by calling (!getItems().isEmpty()) in order to understand if the robot currently holds any items. If the FlooringRobot is holding items, then it proceeds to iterate through the held items to see if there are any items deliverable on this floor. If there are, then the robot delivers them, if there are not then the robot continues moving in a certain direction depending on which side it picked the items up from. If the robot does not hold any items, the system then runs more checks to determine if the Flooring Robot is in the process of collecting items or is simply idle, which changes the action taken by the robot at the tick.

By implementing the Tickable interface, variations in behaviour based on class can be maintained whilst also maintaining a consistent method signature. For instance, a CyclingRobot behaves differently to a FlooringRobot which also behaves differently from a ColumnRobot. Notably, if the system wanted to introduce more simulation units, those could implement Tickable as well. Thus, we have encapsulated the logic for simulation units to tick without violating the interface contract, which in turn follows the pattern of polymorphism and high cohesion.

Finally, in order to differentiate between the modes, we have moved the Mode enum from MailRoom into Simulation. Whilst we could have made a separate class entirely for this enum, Simulation is the only class that uses it, so we have left it in Simulation as Information Expert. The behaviour of Simulation changes two ways based on the Mode. First, different types of robots need to be initialised depending on the mode and then robots are distributed from the MailRoom in different ways also depending on the mode. To deal with this, the constructor of Simulation has switch-cases to initialise robots based on the mode and dispatchRobot() also contains these. Whilst this does bloat Simulation a bit, this is all vital information that Simulation needs in order to simulate the system.

2 - Feature Extension

Feature 1 (Parcels):

The MailItem superclass makes it very easy to implement Parcels. The default constructor for Parcel calls the superclass constructor, and adds the attached weight to the parcel. Furthermore, the previously mentioned getWeight() method is now overridden to return the weight of the parcel. By having the add method be in ItemList as opposed to Robot, we are able to override this method to add to a robot differently depending on the type of Robot as per Protected Variation. As all arrays of MailItems are in the form of the superclass MailItem, no additional logic need be added to the robots.

Feature 2 (Flooring Mode):

The simulation is in charge of the enum Mode that dictates what mode the system will run. The constructor for Simulator has switch-cases for initialising robot types based on what robots are required for the mode. As such, in order to introduce Flooring Mode, subclasses FlooringRobots and ColumnRobots need to be created which implement the behaviour that occurs at each tick of time that has been outlined in the spec.

In particular, ColumnRobot has both an enum Side, which initialises the Column robot to a specific side of the building and an isWaiting attribute which triggers when the column robot has reached its floor that it is delivering items to.

In contrast, FlooringRobot contains pointers to both column robots and an attribute state which gives it a state from the enum State. All FlooringRobots start idle and can be either delivering left and right or moving to collect left or right. These are the 5 states that the robot can be in.

Having established our two subclasses, we can then add the logic for initialisation and dispatching the robots from the mailroom. Since both activeRobots, idleRobots and deactivatedRobots are all Robot arrays, rather than any of the subclasses, we are able to simply tick the activeRobots regardless of what subclasses the robots are.

3: Future Development Plan

Feature 1 (More Mail Items):

The new design makes no assumptions over what type of MailItem any mail item is. As such, when implementing a new MailItem, all you would have to do is extend the superclass MailItem, then add any additional logic or attributes that the item has.

Feature 2 (More Modes):

As outlined above, in order to add more modes, we need to create or use subclasses of Robot, then add the mode to the Mode enum. Following this, we can simply just add the logic to the switch statements in Simulation. Since the robots are stored in a Robot array, as long as the tick logic is implemented correctly for each type, through polymorphism, we are simply able to call the tick method in each active robot.

Feature 3 (Further Extension):

By having Tickable as an interface, in case more entities were to be introduced to the system, we are able to introduce more simulation units that implement Tickable.

Figure 1: A static design model displaying the DS system.

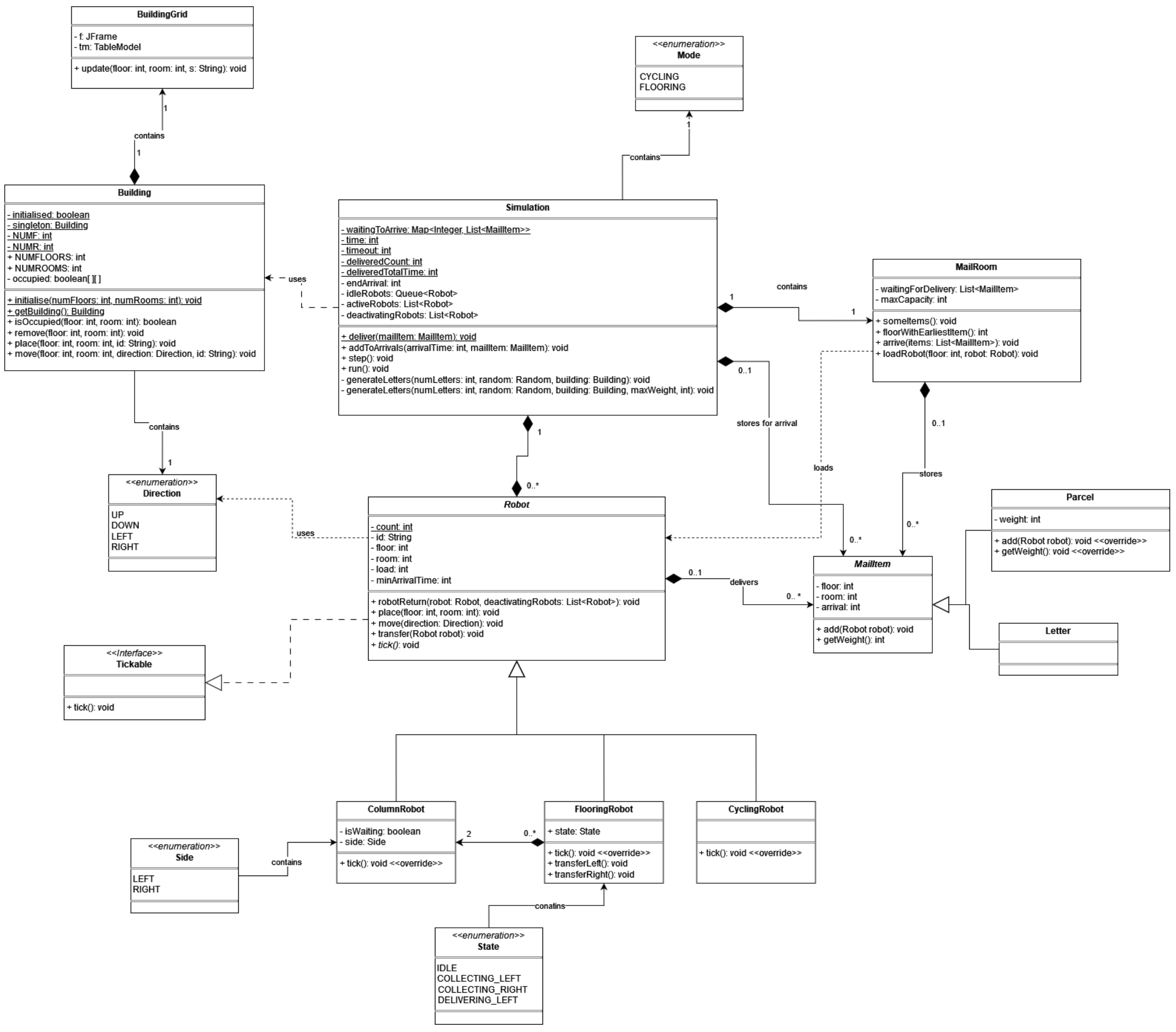
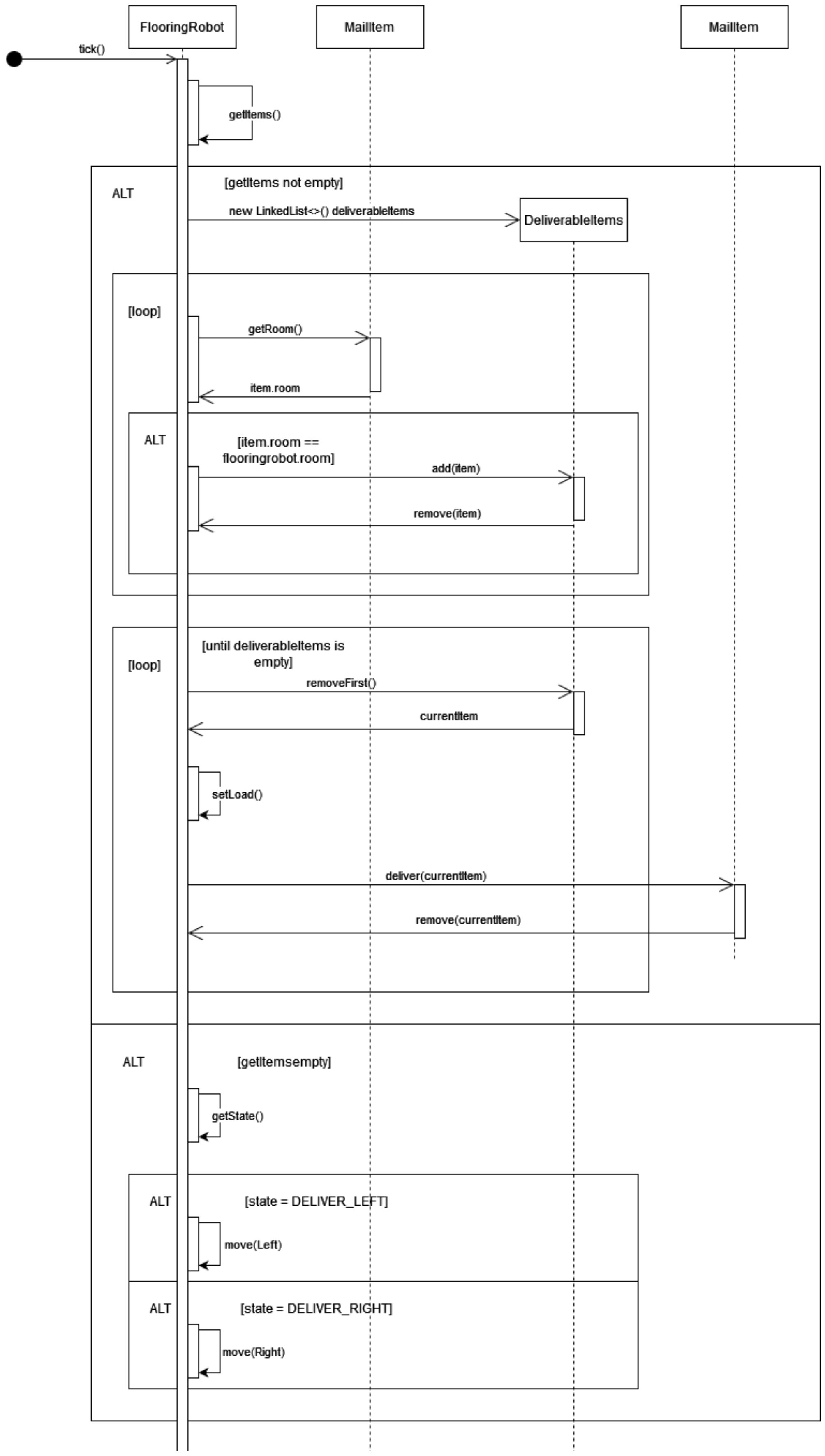

BuildingGrid
- f: JFrame
- tm: TableModel
+ update(floor: int, room: int, s: String): void
1
contains
1
<<enumeration>>
Mode
CYCLING
FLOORING
1
contains
Building
- initialised: boolean
- singleton: Building
- NUMF: int
- NUMR: int
+ NUMFLOORS: int
+ NUMROOMS: int
- occupied: boolean[ ][ ]
+ initialise(numFloors: int, numRooms: int): void
+ getBuilding(): Building
+ isOccupied(floor: int, room: int): boolean
+ remove(floor: int, room: int): void
+ place(floor: int, room: int, id: String): void
+ move(floor: int, room: int, direction: Direction, id: String): void
uses
Simulation
- waitingToArrive: Map<Integer, List<MailItem>>
- time: int
- timeout: int
- deliveredCount: int
- deliveredTotalTime: int
- endArrival: int
- idleRobots: Queue<Robot>
- activeRobots: List<Robot>
- deactivatingRobots: List<Robot>
+ deliver(mailItem: MailItem): void
+ addToArrivals(arrivalTime: int, mailItem: MailItem): void
+ step(): void
+ run(): void
- generateLetters(numLetters: int, random: Random, building: Building): void
- generateLetters(numLetters: int, random: Random, building: Building, maxWeight, int): void
contains
1
1
MailRoom
- waitingForDelivery: List<MailItem>
- maxCapacity: int
+ someItems(): void
+ floorWithEarliestItem(): int
+ arrive(items: List<MailItem>): void
+ loadRobot(floor: int, robot: Robot): void
0..1
stores for arrival
loads
0..1
stores
contains
1
1
0..*
<<enumeration>>
Direction
UP
DOWN
LEFT
RIGHT
uses
Robot
- count: int
- id: String
- floor: int
- room: int
- load: int
- minArrivalTime: int
+ robotReturn(robot: Robot, deactivatingRobots: List<Robot>): void
+ place(floor: int, room: int): void
+ move(direction: Direction): void
+ transfer(Robot robot): void
+ tick(): void
0..1
delivers
0.. *
0..*
0..*
MailItem
- floor: int
- room: int
- arrival: int
+ add(Robot robot): void
+ getWeight(): int
Parcel
- weight: int
+ add(Robot robot): void <<override>>
+ getWeight(): void <<override>>
Letter
<<Interface>>
Tickable
+ tick(): void
ColumnRobot
- isWaiting: boolean
- side: Side
+ tick(): void <<override>>
2
0..*
FlooringRobot
+ state: State
+ tick(): void <<override>>
+ transferLeft(): void
+ transferRight(): void
CyclingRobot
+ tick(): void <<override>>
<<enumeration>>
Side
LEFT
RIGHT
contains
conatins
<<enumeration>>
State
IDLE
COLLECTING_LEFT
COLLECTING_RIGHT
DELIVERING_LEFT

Figure 2: A partially complete Dynamic System Model of FlooringRobot.tick()


FlooringRobot
MailItem
MailItem
tick()
getItems()
ALT
[getItems not empty]
new LinkedList<>() deliverableItems
DeliverableItems
[loop]
getRoom()
item.room
ALT
[item.room == flooringrobot.room]
add(item)
remove(item)
[loop]
[until deliverableItems is empty]
removeFirst()
currentItem
setLoad()
deliver(currentItem)
remove(currentItem)
ALT
[getItemsempty]
getState()
ALT
[state = DELIVER_LEFT]
move(Left)
ALT
[state = DELIVER_RIGHT]
move(Right)