SWEN30006 Project 1

Workshop 1 Team 13

Name1: Yi PU (1518795) Name2: Ruijia Lu(1506676)

1- Refactoring and Analysis of Current Design

In this report, we will analyze the current design of the Automail system and refactor the system based on GRASP (General Responsibility Assignment Software Patterns). There are several concerns based on GRASP principles, leading to scalability and maintainability issues.

1.1 MailRoom class

The MailRoom class takes on many responsibilities. It is not only responsible for robot dispatch but also for managing item information and directly controlling robots to perform tasks, which may violate High Cohesion, Low Coupling and Controller principles.

For example, someItems(), floorWithEarliestItem(), arrive(List<Letter> items), robotReturn(Robot robot), loadRobot(int floor, Robot robot), I think these methods are not the responsibilities of the MailRoom class, we should ensure it only takes its responsibility: dispatch robots according to GRASP principles through assigning these tasks to other classes, such as the Letter class and Robot class.

1.2 Letter class

The Letter class only stores information about letters including the floor number, room number and arrival time. However, according to the Information Expert principle, it should be responsible for processing all information related to letters, and not rely on external classes to obtain or process this information. We should put some of the methods from the MailRoom class into the Letter class, such as someItems(), floorWithEarliestItem(), and arrive(List<Letter> items). In the Letter class, we can find the earliest arriving letter, corresponding floor number, and process letters.

Moreover, according to the Polymorphism principle, the Letter class lacks scalability. In cycling mode, we only have letters to deliver, but in flooring mode, we have parcels, and in the future, we may have more different items to deliver. The best way to achieve it is to make it a parent, abstract class or interface so that other robots can extend or implement it.

1.3 Robot class

According to the Information Expert principle, the Robot class mainly controls the robot's movement, but not the other responsibilities he is supposed to do. For example, the Robot class should control his return to the MailRoom and load letters.

Meanwhile, according to the Polymorphism principle, the Robot class should be easily scalable. In cycling mode, we only have one type of robot, but in flooring mode, we have two different types of robots, and in the future, we may have more different kinds of robots. So, we can’t make the Robot class so specific. We should make it a parent, abstract class or interface so that other robots can extend or implement it.

2- Feature Extension

2.1 Item extension

To better implement the Automail system, we create a superclass Item with attributes floor, room, rival, and weight, and two subclasses: Parcel and Letter. The Item class implements the Comparable<Item> interface for sorting by floor and room. The Parcel class inherits from Item and calls its constructor to initialize properties. Both Letter and Parcel inherit the weight attribute; however, Letter always has a weight of 0, while Parcel has a weight greater than 0. When loading a robot, only items with weight greater than 0 (Parcels) need to be checked against the robot's remaining capacity.

2.2 Operating mode extension and redesign

2.2.1 Operating mode extension: Flooring mode
In flooring mode, the MailRoom system efficiently distributes items using FloorRobots and ColumnRobots. FloorRobots deliver items on specific floors and check for ColumnRobots on either side when they have no items to deliver. ColumnRobots transport items from the MailRoom to each floor and hand them over to the appropriate FloorRobot. The system periodically calls the flooringTick method to update statuses and assign tasks, ensuring timely and accurate deliveries throughout the building.

2.2.2 Redesign
The MailRoom class is the core component of an Automail system, responsible for managing and coordinating different types of robots for efficient delivery of items:

(1) Initialise Parameters: initialise the system parameters including the number of robots, mode (CYCLING or FLOORING), capacity and number of rooms through the constructor and also initialise the list of items waiting to be dropped based on the number of floors. Finally, the initializeRobots method is called to initialise different types of robots according to mode (CYCLING or FLOORING).

(2) Time step processing: the advancement of the time step is simulated by the tick method, and then the appropriate processing method (cyclingTick or flooringTick) is called to update the robot's state and tasks according to the current mode. In the cyclingTick method, the active CyclingRobot is processed and the cyclingDispatch method is called to distribute the robot when conditions are met. In the flooringTick method, the status of FloorRobot and ColumnRobot is handled and the flooringDispatch method is called to distribute the robot.

(3) Dispatch of robots: cyclingDispatch method distributes idle CyclingRobot according to the floor of the earliest arriving items and places it at the start position. flooringDispatch method distributes idle ColumnRobot according to the floor of the earliest arriving items and places it at the corresponding position. At the same time, the position of the FloorRobot is initialised.

The Robot class serves as the base class for Robot and provides basic properties such as robot ID, floor, room, load, remainingCapacity, etc., as well as basic Robot operation methods such as moving, loading, sorting, and adding items. In addition, the Robot class defines the logic for the robot to return to the MailRoom, i.e., add the robot to the deactivatingRobots list when it arrives at the MailRoom.

The ColumnRobot class inherits from the Robot class and is primarily responsible for moving between floors and transporting items from the MailRoom to the floors. processColumnRobots method determines whether the robot moves up or down depending on whether the robot is carrying items or not and the floor it is currently on. The processDeactivatingRobots method handles robots that need to be deactivated, removing them from the active list and adding them to the inactive list.

The FloorRobot class, inheriting from Robot, gets items from a ColumnRobot and delivers them on a specific floor. The processEmptyFloorRobot method directs idle FloorRobots to move or transfer items based on nearby ColumnRobots. The processLoadedFloorRobot method manages delivery to rooms or moves the robot accordingly. transferItemToRobot handles item transfers from ColumnRobots to FloorRobots, while checkWaitingPosition checks for waiting ColumnRobots nearby.

The CyclingRobot class inherits from the Robot class and is primarily responsible for delivering items in Cycling mode. The tick method simulates the progression of time steps and decides whether to return to the MailRoom or deliver items based on whether there are items to deliver. The returnToMailRoom method moves the robot back to the MailRoom. The deliverItems method delivers items when the robot reaches the target room; otherwise, it continues moving toward the target floor.

All robot classes rely on the MailRoom class to manage their states and task assignments. The MailRoom class updates the states and tasks of each robot by periodically calling their methods, ensuring that items are delivered to their destinations accurately and on time. The Simulation class acts as the top-level controller, responsible for advancing the simulation time steps. It uses the tick method to call the tick method of the MailRoom class to update the system's state.

The Item class manages item states and their delivery process. It initializes items with floor, room, arrival time, and weight. The compareTo method sorts items by floor and room. The deliver method updates delivery stats, while addToArrivals adds items to a waiting list. The someItems method checks if items are waiting in the MailRoom, and floorWithEarliestItem returns the floor of the earliest item. The arrive method moves items to the MailRoom's delivery list.

2.2.3 The design that satisfies the GRASP principles.

(1) Information Expert

Each method is implemented by the class that has the most knowledge about the relevant information. The MailRoom class is responsible for managing and dispatching the robots to ensure efficient item delivery. The Robot class and its subclasses are responsible for specific robot operations and state management. The Item class is responsible for managing the state and delivery of items.

(2) Creator

Objects are created by the class that best understands the needs for using those objects. The MailRoom class is responsible for creating and initializing Robot instances, while the Simulation class is responsible for creating and initializing Item instances.

(3) Controller

The MailRoom class acts as the system's controller, responsible for advancing time steps and managing and dispatching the robots. The Simulation class, as the top-level controller, is responsible for advancing the simulation time steps and uses the tick method to call the tick method of the MailRoom class to update the system's state.

(4) Low Coupling

Each class has a clear responsibility, avoiding excessive dependencies between classes. The MailRoom class manages robots by calling their public methods without needing internal details. Robot classes handle their specific operations and states, while the `Item` class focuses on item management without robot logic.

Inheritance among ColumnRobot, FloorRobot, and CyclingRobot from the Robot class allows for the reuse of core methods, reducing code duplication and enhancing maintainability and extensibility.

(5) High cohesion
Each class has a clear responsibility and focuses on completing specific tasks. For example, the MailRoom class is responsible for managing and dispatching the robots, the Robot class and its subclasses are responsible for specific robot operations and state management, and the Item class is responsible for managing the state and delivery of items. Each method focuses on completing a specific task, avoiding complex logic within the method. For instance, the tick method in the MailRoom class is solely responsible for simulating the advancement of time steps, while the specific robot operations are handled by their respective methods (such as cyclingTick and flooringTick).

3- Future development plan

3.1 More items to deliver

We have considered the delivery of letters and parcels, but many more items will need to be delivered in the future. In this system design, we have considered the item's weight, but many other factors should be taken into account, such as the item's size.

In addition, we can prioritize the delivery of items, not just by the arrival time of delivery. For example, if some essential documents or items need to be delivered, we should deliver these high-priority items first.

3.2 More delivering modes

We have completed the cycling and flooring modes in this AutoMail system design, but there may be more in the future. We have only two ladders in the building for the flooring mode, and we design two column robots. However, in a more complex building environment, we must plan the best robot delivery path.

Moreover, the robot in our design only considered the delivery of items on the same floor at a time; in future development, we can also design the robot to simultaneously deliver items on different floors.

As mentioned above, the items to be delivered may have a priority, so we should design a mode that delivers items according to their priority.

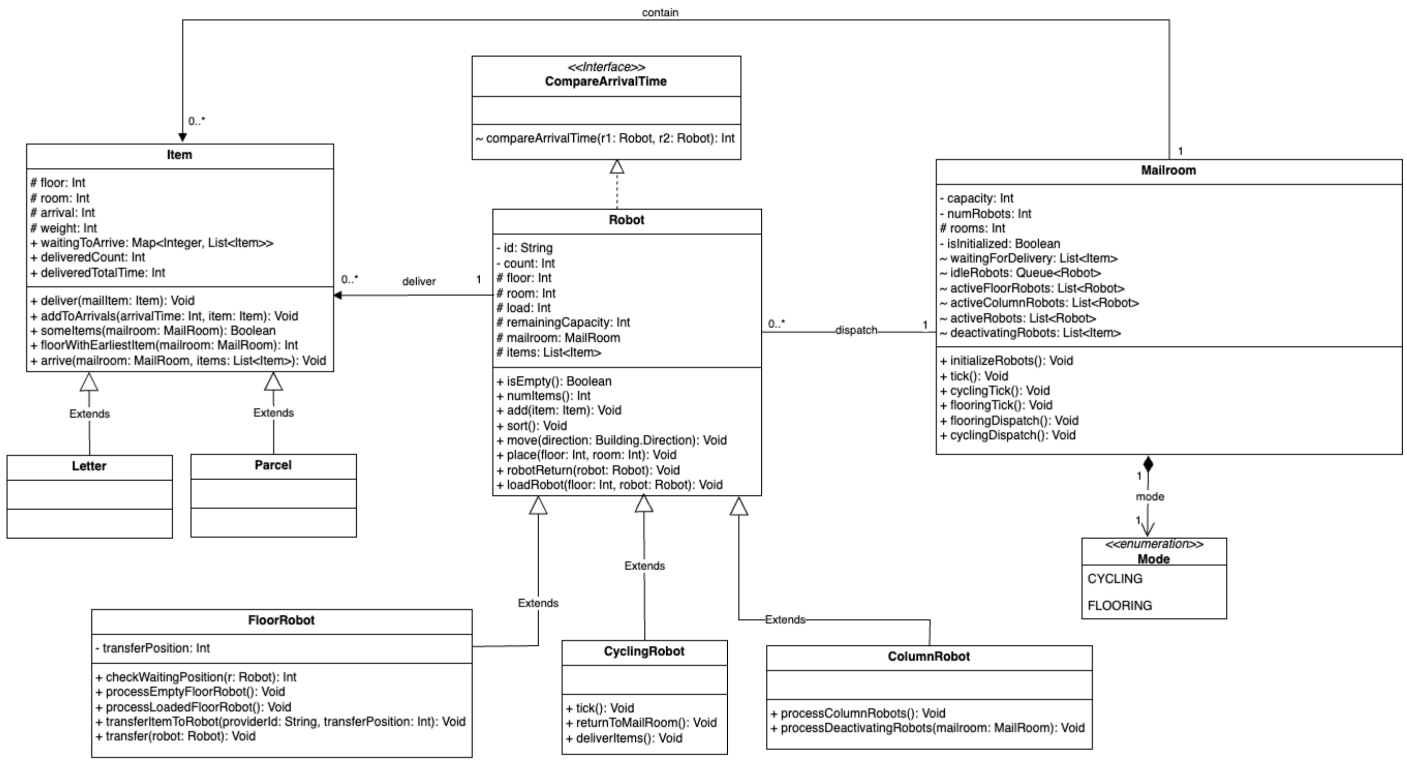
contain
0..*
1
<<Interface>>
CompareArrivalTime
~ compareArrivalTime(r1: Robot, r2: Robot): Int
Item
# floor: Int
# room: Int
# arrival: Int
# weight: Int
+ waitingToArrive: Map<Integer, List<Item>>
+ deliveredCount: Int
+ deliveredTotalTime: Int
+ deliver(mailItem: Item): Void
+ addToArrivals(arrivalTime: Int, item: Item): Void
+ someItems(mailroom: MailRoom): Boolean
+ floorWithEarliestItem(mailroom: MailRoom): Int
+ arrive(mailroom: MailRoom, items: List<Item>): Void
0..*
deliver
1
Robot
- id: String
- count: Int
# floor: Int
# room: Int
# load: Int
# remainingCapacity: Int
# mailroom: MailRoom
# items: List<Item>
+ isEmpty(): Boolean
+ numItems(): Int
+ add(item: Item): Void
+ sort(): Void
+ move(direction: Building.Direction): Void
+ place(floor: Int, room: Int): Void
+ robotReturn(robot: Robot): Void
+ loadRobot(floor: Int, robot: Robot): Void
0..*
dispatch
1
Mailroom
- capacity: Int
- numRobots: Int
# rooms: Int
- isInitialized: Boolean
~ waitingForDelivery: List<Item>
~ idleRobots: Queue<Robot>
~ activeFloorRobots: List<Robot>
~ activeColumnRobots: List<Robot>
~ activeRobots: List<Robot>
~ deactivatingRobots: List<Item>
+ initializeRobots(): Void
+ tick(): Void
+ cyclingTick(): Void
+ flooringTick(): Void
+ flooringDispatch(): Void
+ cyclingDispatch(): Void
1
mode
1
<<enumeration>>
Mode
CYCLING
FLOORING
Extends
Extends
Letter
Parcel
Extends
Extends
Extends
FloorRobot
- transferPosition: Int
+ checkWaitingPosition(r: Robot): Int
+ processEmptyFloorRobot(): Void
+ processLoadedFloorRobot(): Void
+ transferItemToRobot(providerId: String, transferPosition: Int): Void
+ transfer(robot: Robot): Void
CyclingRobot
+ tick(): Void
+ returnToMailRoom(): Void
+ deliverItems(): Void
ColumnRobot
+ processColumnRobots(): Void
+ processDeactivatingRobots(mailroom: MailRoom): Void

Figure 1 Design Class Diagram

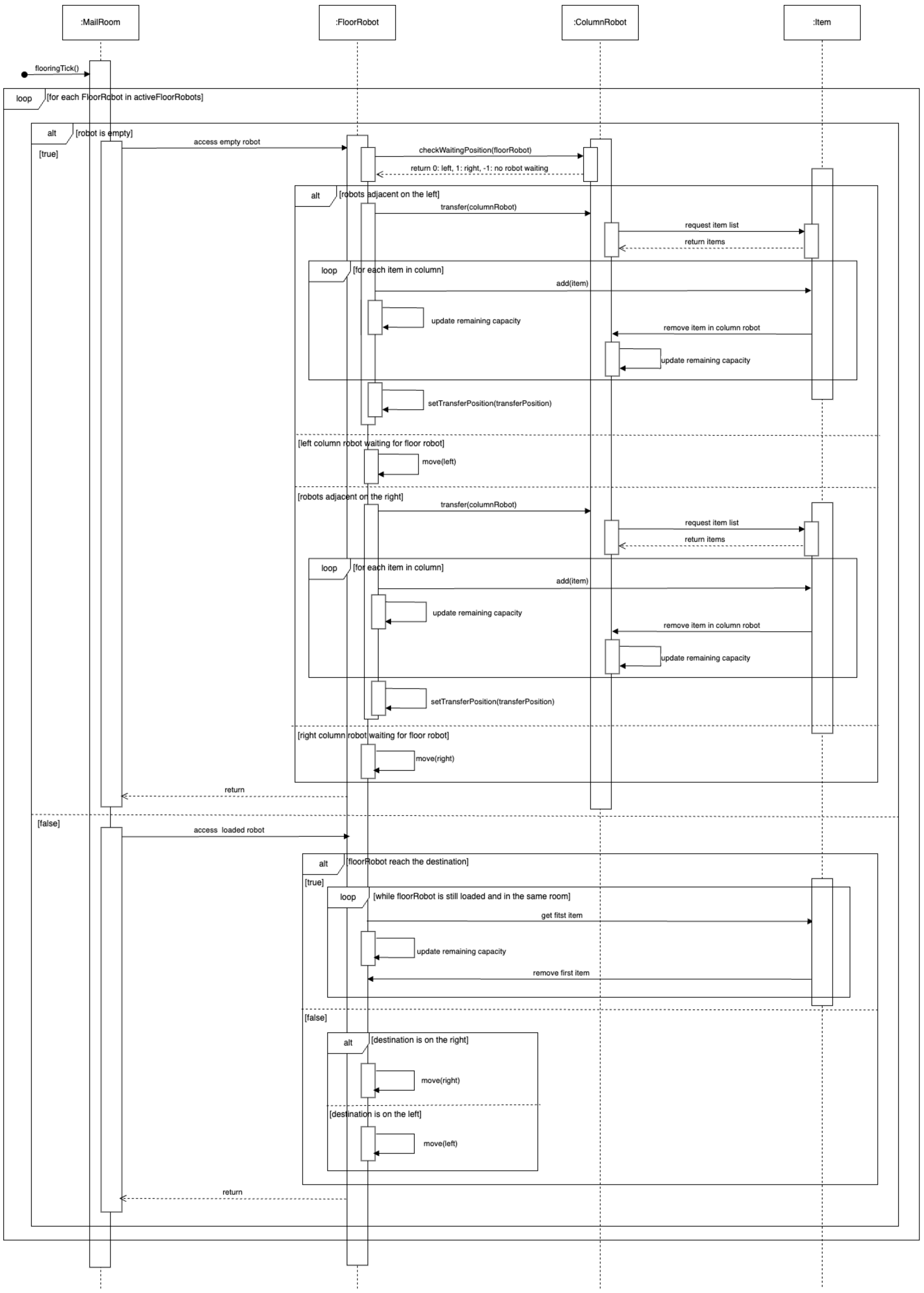
:MailRoom
:FloorRobot
:ColumnRobot
:Item
flooringTick()
loop [for each FloorRobot in activeFloorRobots]
alt [robot is empty]
[true]
access empty robot
checkWaitingPosition(floorRobot)
return 0: left, 1: right, -1: no robot waiting
alt [robots adjacent on the left]
transfer(columnRobot)
request item list
return items
loop [for each item in column]
add(item)
update remaining capacity
remove item in column robot
update remaining capacity
setTransferPosition(transferPosition)
[left column robot waiting for floor robot]
move(left)
[robots adjacent on the right]
transfer(columnRobot)
request item list
return items
loop [for each item in column]
add(item)
update remaining capacity
remove item in column robot
update remaining capacity
setTransferPosition(transferPosition)
[right column robot waiting for floor robot]
move(right)
return
[false]
access loaded robot
alt [floorRobot reach the destination]
[true]
loop [while floorRobot is still loaded and in the same room]
get fitst item
update remaining capacity
remove first item
[false]
alt [destination is on the right]
move(right)
[destination is on the left]
move(left)
return

Figure 2 Design Sequence Diagram