SWEN30006 Project 1

Workshop Monday 15:15 - 17:15 Team 05

Yee Kiu Yeung (1568135), Adam Eldaly (1353834), Rhea Iai (1398804)

1 - Critical Analysis of Current Automail System

The current Automail system displays a sufficient use of GRASP principles for the implemented design, however, the proposal of the extended Automail system may quickly cause challenges with scalability, cohesion and coupling.

The *MailRoom* class in the original system was designed as the central controller for all activities related to robot management and mail deliveries, leading to several scalability issues. While the *MailRoom* class holds the necessary information to perform their tasks, following the information expert principle, the responsibility of the assignment are not well-separated and the robot behaviours are not delegated effectively. It combines the responsibilities of robot dispatching, mail handling, and operational logic, requiring the management of both robots and mail simultaneously. The *MailRoom* and *Robot* classes are tightly coupled and are susceptible to being overloaded with responsibilities as the system scales. Changes in robot behaviour would require changes to the *MailRoom*, making the incohesive design difficult to modify and extend.

The *Robot* class in the original system lacked autonomy and hindered the ability to create an extendable system. All robots behaved the same in the original system, regardless of the environment they were operating in, inherently limiting the flexibility of the system and making it difficult to extend. The lack of abstract classes and interfaces for the robot behaviour makes it difficult to introduce different robot types with different strategies for delivering mail, causing any changes to robot behaviour to require modifications to the *MailRoom* class. Although the creator principle is utilised in the original programme, it misses opportunities to apply it in a scalable way. The rigid and uniform creation of robots will create challenges with the implementation of the flooring mode, where new robot types and behaviours are introduced.

Similarly, the *Letter* class follows a simple structure without leveraging polymorphism for specialised behaviours. In the original system, *Letter* is the only mail item, which has basic properties such as its destination floor and room. With no way of distinguishing between weightless letters and heavier parcels meant that capacity management was difficult to implement.

2 - New Design Proposal

Feature 1 (additional MailRoom classes): To preserve the information expert and creator principle rich designs of the *MailRoom* class, we introduced polymorphism by removing dependency on the singular *MailRoom* class and distributing responsibilities between the proposed modes: cycling and flooring. By refactoring the *MailRoom* class into *FlooringMailRoom* and *CyclingMailRoom* subclasses, the creator pattern is better utilised and promotes greater cohesion, where each mailroom type is responsible for creating and managing the robots relevant to their strategy.

The overall flow of robot dispatch and mail delivery is managed by the *MailRoom* subclasses. Each *MailRoom* type acts as a localised controller for their respective operations, deciding when to dispatch robots and manage deliveries. This maintains a simpler and modular design allowing each component to focus on their designated operations. The *MailRoom* no longer deals directly with how robots deliver items; that responsibility is handed over to the robot subclasses, reducing the coupling between the *MailRoom* and *Robot* classes. *MailRoom* dispatches robots and updates their status but does not need to know how each robot implements its movement logic. This decoupling makes the system more modular and easier to extend.

Other considerations: A centralised *SystemController* was considered as part of the design to manage simulation flow and decide when to dispatch robots in a global context, however, as too many decisions become centralised, the program may become overly complex and less cohesive. Considering the simplicity of the operating mode and interaction between the *ColumnRobot* and *FloorRobot*, it would be more efficient to maintain a decentralised system, allowing the *MailRoom* subclasses to act as localised controllers of their respective operations.

Another proposed design was to use Factory Pattern to create different types of *MailRoom* (or *Robot*). Under this pattern, a Factory class will take the type of operation as an argument (*Flooring*, *Cycling*) and then generate an instance of a specific type of *MailRoom* based on the operation system. This method can create objects without specifying the exact class of the object that will be created and is useful when needed to generate different types of objects based on input or conditions. The intention of using this pattern is to encapsulate the *MailRoom* creation process in the factory class, create indirection between *Simulation* and *MailRoom*, and increase cohesion of *Simulation*. However, we decided to leave it out, as it does not effectively reduce its coupling, and we agreed that creating *Building*, *MailRoom*, and *MailItem* is part of the *Simulation* class' job. Also, the structure will be overcomplicated if adding too many classes. Therefore, we have not adapted this idea.

Moreover, we tried making the *Building* class to create robots inside each floor of the building. The reason for this is to separate the *ColumnRobot* and *FloorRobot* objects. Nevertheless, this method increases coupling between *Robot* and *Building* and reduces the cohesion of *Building*. Therefore, we decided to keep *MailRoom* to be responsible for creating all types of robot.

Feature 2 (additional Robot classes): The new system introduces specialised robot subclasses – *CyclingRobot*, *FloorRobot*, and *ColumnRobot* – each type of robot is an information expert on how to execute its own operations such as *tick()*, *move()*, and *transfer()*. This specialisation is implemented through polymorphism, where the Robot class serves as an abstract base class, and each subclass implements its own specific behaviour. Robots can be treated polymorphically allowing both *MailRoom* types to interact with different robot classes without needing to know details about the behaviour of each robot. This helps to reduce coupling between components and dependency between different types of robots and mailroom operations.

The use of an abstract *Robot* class with specialised subclasses allows different robot behaviours to be encapsulated without changing the main *MailRoom* logic. Each robot type uses its own internal state such as floor and room and implements its own *tick()* method, deciding how it behaves in each simulation step and when to deliver mail or return to the mailroom. For example, a *CyclingRobot* decides how to move through the building based on its own behaviour pattern and a *ColumnRobot* knows how to transfer mail to a *FloorRobot* which handles deliveries across a specific floor, only interacting with the column robots when necessary. This separation of responsibilities makes each class easier to understand and maintain, enabling high cohesion.

The new system introduces capacity management within the *Robot* class with each robot responsible for tracking its own capacity and rejecting items if *Parcel* objects exceed the robot's maximum capacity. Robots behave more autonomously, deciding independently whether they can accept more mail based on their current load and capacity constraints.

Other considerations: A centralised implementation of a *RobotManager* was considered to monitor the state of all robots and make the decision to dispatch each robot, which tasks to prioritise and how robots interact based on the overall system conditions. However, the decentralised control where each *MailRoom* type is an expert in managing its own robot types is more maintainable for simple routing requirements in distinct modes. For the current system with the proposed modes, the use of a centralised *RobotManager* may over complicate the system and increase coupling and dependence among classes when it is not necessary.

Feature 3 (refactoring Letter class): The system initially treated all mail items similarly without leveraging polymorphism for specialised behaviours. Designing the *Parcel* class to extend the abstract *Letter* class was a valid design choice for the simplicity of the proposed programme, however, could very easily be the bottleneck for extensibility. By refactoring the *Letter* class to have an abstract *MailItem* class which handles different types of mail, each subclass has specialised attributes and behaviours. This differentiation allows the system to extend easily and allows each class to be an information expert for its specific type of mail item where a *Parcel* knows its weight and *Letter* knows that it is weightless.

3 - Future Development Plan

Feature 1 (adding further mail items): The new system has a flexible *MailItem* hierarchy with polymorphism, which promotes easy extensibility and allows future additions of mail item types without changing the core system.

Feature 2 (adding further delivery modes): The use of polymorphism in the *Robot* and *MailRoom* classes is applied for future extensibility. For example, if different mode types are introduced to the existing *Cycling* and *Flooring* modes, new *Robot* and *MailRoom* classes can extend their respective abstract classes to achieve their new modes. However, if there are plans to implement more complex coordination strategies, requiring dynamic routing arrangements in a larger building with multiple robots per floor, it may be beneficial to implement a centralised system to ensure efficient resource usage and load balancing so that no single robot is idle or overloaded and shared resources are used effectively.

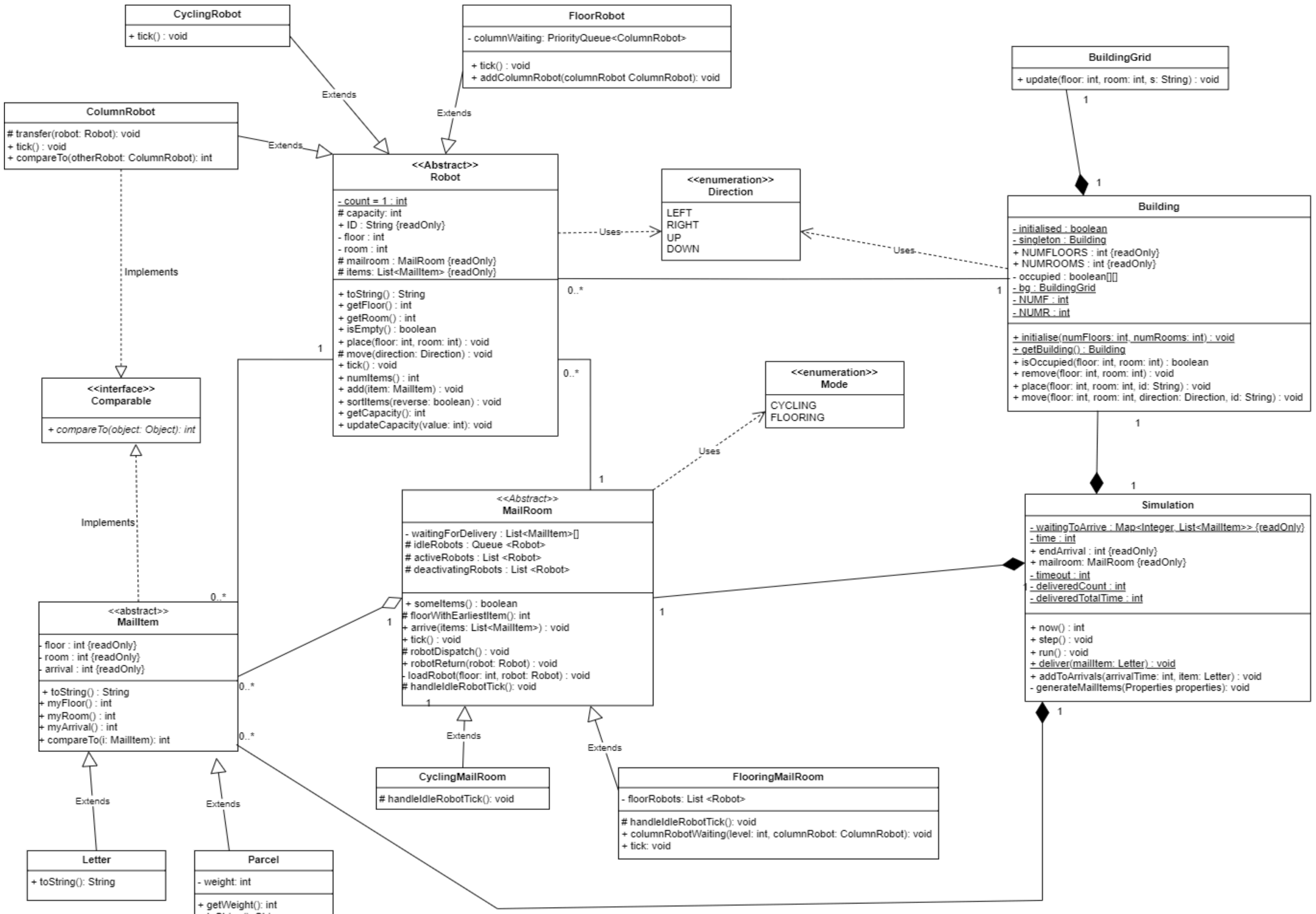
CyclingRobot
+ tick() : void
FloorRobot
- columnWaiting: PriorityQueue<ColumnRobot>
+ tick() : void
+ addColumnRobot(columnRobot ColumnRobot): void
BuildingGrid
+ update(floor: int, room: int, s: String) : void
ColumnRobot
# transfer(robot: Robot): void
+ tick() : void
+ compareTo(otherRobot: ColumnRobot): int
Extends
Extends
Extends
<<Abstract>>
Robot
- count = 1 : int
# capacity: int
+ ID : String {readOnly}
- floor : int
- room : int
# mailroom : MailRoom {readOnly}
# items: List<MailItem> {readOnly}
+ toString() : String
+ getFloor() : int
+ getRoom() : int
+ isEmpty() : boolean
+ place(floor: int, room: int) : void
# move(direction: Direction) : void
+ tick() : void
+ numItems() : int
+ add(item: MailItem) : void
+ sortItems(reverse: boolean) : void
+ getCapacity(): int
+ updateCapacity(value: int): void
Uses
<<enumeration>>
Direction
LEFT
RIGHT
UP
DOWN
Uses
1
1
Building
- initialised : boolean
- singleton : Building
+ NUMFLOORS : int {readOnly}
+ NUMROOMS : int {readOnly}
- occupied : boolean[][]
- bg : BuildingGrid
- NUMF : int
- NUMR : int
+ 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
0..*
1
Implements
1
0..*
<<interface>>
Comparable
+ compareTo(object: Object): int
<<enumeration>>
Mode
CYCLING
FLOORING
Uses
1
1
1
Implements
<<Abstract>>
MailRoom
- waitingForDelivery : List<MailItem>[]
# idleRobots : Queue <Robot>
# activeRobots : List <Robot>
# deactivatingRobots : List <Robot>
+ someItems() : boolean
# floorWithEarliestItem(): int
+ arrive(items: List<MailItem>) : void
+ tick() : void
# robotDispatch() : void
+ robotReturn(robot: Robot) : void
- loadRobot(floor: int, robot: Robot) : void
# handleIdleRobotTick(): void
Simulation
- waitingToArrive : Map<Integer, List<MailItem>> {readOnly}
- time : int
+ endArrival : int {readOnly}
+ mailroom: MailRoom {readOnly}
- timeout : int
- deliveredCount : int
- deliveredTotalTime : int
+ now() : int
+ step() : void
+ run() : void
+ deliver(mailItem: Letter) : void
+ addToArrivals(arrivalTime: int, item: Letter) : void
- generateMailItems(Properties properties): void
0..*
1
1
1
<<abstract>>
MailItem
- floor : int {readOnly}
- room : int {readOnly}
- arrival : int {readOnly}
+ toString() : String
+ myFloor() : int
+ myRoom() : int
+ myArrival() : int
+ compareTo(i: MailItem): int
0..*
0..*
1
1
Extends
Extends
Extends
Extends
CyclingMailRoom
# handleIdleRobotTick(): void
FlooringMailRoom
- floorRobots: List <Robot>
# handleIdleRobotTick(): void
+ columnRobotWaiting(level: int, columnRobot: ColumnRobot): void
+ tick: void
Letter
+ toString(): String
Parcel
- weight: int
+ getWeight(): int
+ toString(): String

Figure 1. Design Class Diagram describing the Extended Automail system based on requirements in the project specifications

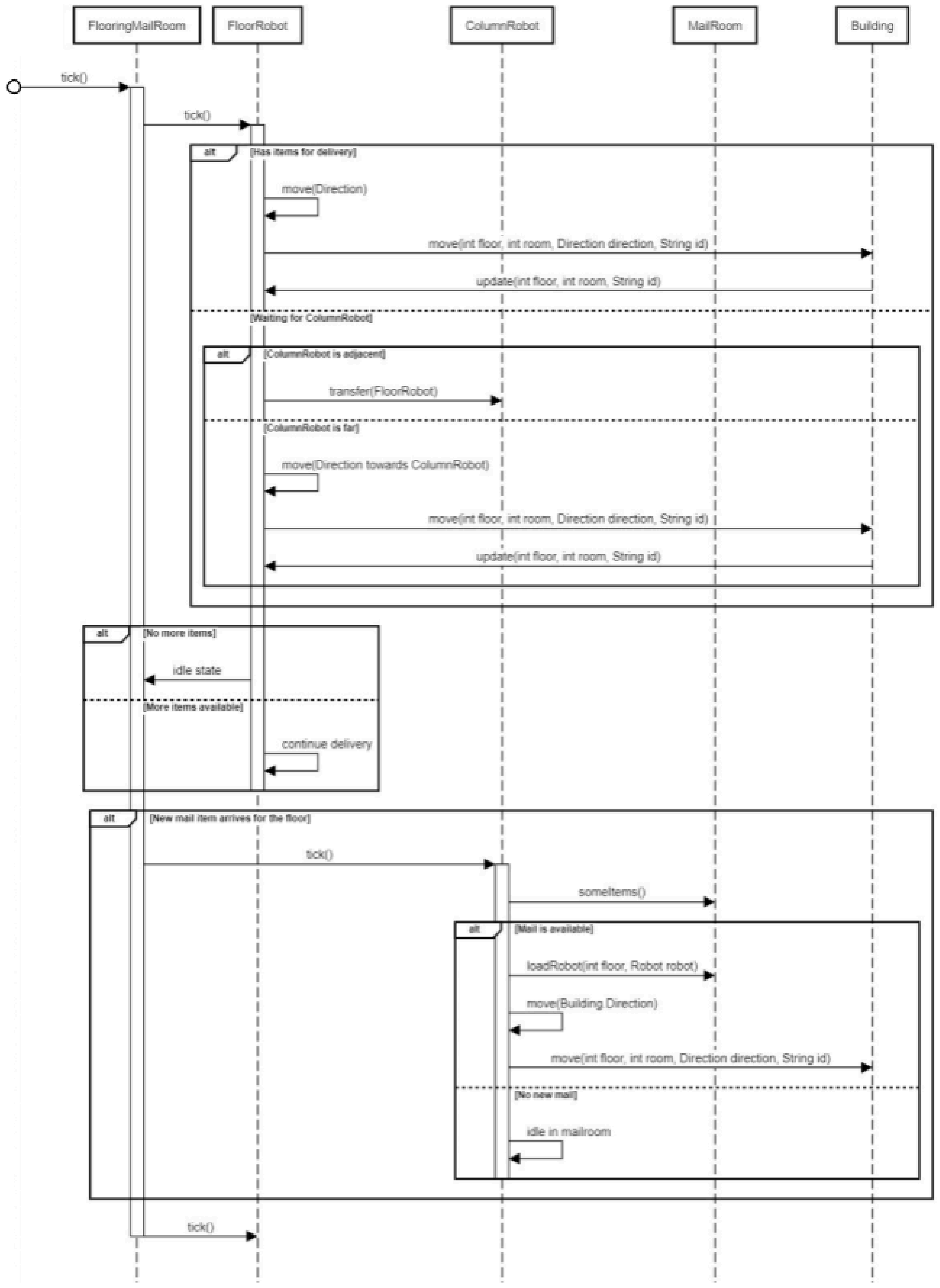
FlooringMailRoom
FloorRobot
ColumnRobot
MailRoom
Building
tick()
tick()
alt [Has items for delivery]
move(Direction)
move(int floor, int room, Direction direction, String id)
update(int floor, int room, String id)
[Waiting for ColumnRobot]
alt [ColumnRobot is adjacent]
transfer(FloorRobot)
[ColumnRobot is far]
move(Direction towards ColumnRobot)
move(int floor, int room, Direction direction, String id)
update(int floor, int room, String id)
alt [No more items]
idle state
[More items available]
continue delivery
alt [New mail item arrives for the floor]
tick()
someItems()
alt [Mail is available]
loadRobot(int floor, Robot robot)
move(Building.Direction)
move(int floor, int room, Direction direction, String id)
[No new mail]
idle in mailroom
tick()

Figure 2: Design Sequence Diagram snippet for the FloorRobot following the calling of tick from the simulation