# Automail System

Miles Li: liyueming828@gmail.com
Skylar Khant: kyishink@student.unimelb.edu.au
Ngoc Thanh Lam Nguyen: ngocthanhlam@student.unimelb.edu.au

September 3, 2024

**Abstract**

This report outlines the refactoring, extension, and analysis of the Automail System, with a particular focus on the implementation of the new FLOORING mode. Through the application of key software design principles, including GRASP (General Responsibility Assignment Software Patterns/Principles), and the Factory Pattern, we have enhanced the system's flexibility, maintainability, and scalability. The improvements address the initial design's issues of high coupling and low cohesion while ensuring that the system is prepared for future expansion. The report also details the practical application of design patterns and principles in achieving a more modular and robust Automail System.

## Contents

## 1 Introduction

The Automail System, developed by Delivering Solutions Inc. (DS), is an automated mail sorting and delivery system designed for large buildings with a dedicated mailroom. Originally, the system was limited to delivering letters using a single operational mode where every robot behaved identically. The need for parcel delivery and more specialized robot behaviors highlighted the limitations of the original design, particularly in its ability to scale and accommodate new modes of operation. This project aims to address these limitations by refactoring the existing design and implementing the FLOORING mode, which introduces a more complex and flexible robot behavior. By applying key design principles and patterns, we have restructured the system to improve its overall cohesion, reduce coupling between components, and prepare it for future development.

## 2 Original Design Analysis

The UML diagram below illustrates the original design of the Automail System before any improvements were made and before the implementation of the FLOORING mode. While there are several advantages in the existing design, there are also notable flaws that need to be addressed:

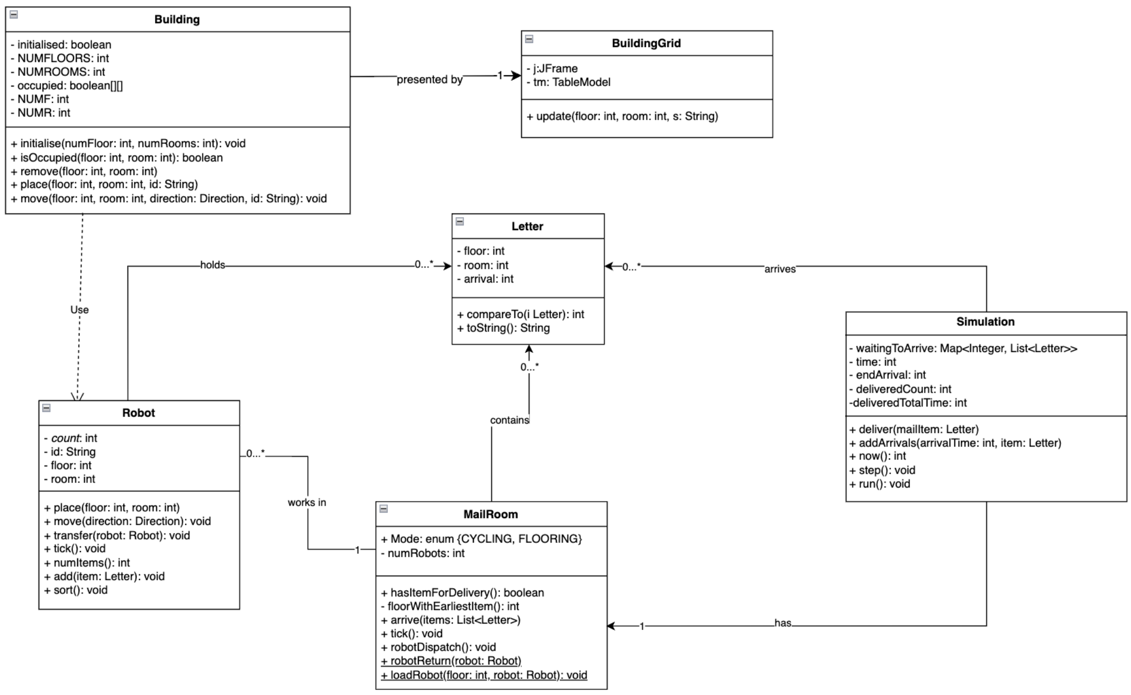

Building
- initialised: boolean
- NUMFLOORS: int
- NUMROOMS: int
- occupied: boolean[][]
- NUMF: int
- NUMR: int
+ initialise(numFloor: int, numRooms: int): void
+ isOccupied(floor: int, room: int): boolean
+ remove(floor: int, room: int)
+ place(floor: int, room: int, id: String)
+ move(floor: int, room: int, direction: Direction, id: String): void
presented by
1
BuildingGrid
- j:JFrame
- tm: TableModel
+ update(floor: int, room: int, s: String)
Use
holds
0...*
Letter
- floor: int
- room: int
- arrival: int
+ compareTo(i Letter): int
+ toString(): String
0...*
arrives
0...*
contains
Simulation
- waitingToArrive: Map<Integer, List<Letter>>
- time: int
- endArrival: int
- deliveredCount: int
-deliveredTotalTime: int
+ deliver(mailItem: Letter)
+ addArrivals(arrivalTime: int, item: Letter)
+ now(): int
+ step(): void
+ run(): void
Robot
- *count*: int
- id: String
- floor: int
- room: int
+ place(floor: int, room: int)
+ move(direction: Direction): void
+ transfer(robot: Robot): void
+ tick(): void
+ numItems(): int
+ add(item: Letter): void
+ sort(): void
0...*
works in
1
MailRoom
+ Mode: enum {CYCLING, FLOORING}
- numRobots: int
+ hasItemForDelivery(): boolean
- floorWithEarliestItem(): int
+ arrive(items: List<Letter>)
+ tick(): void
+ robotDispatch(): void
+ robotReturn(robot: Robot)
+ loadRobot(floor: int, robot: Robot): void
1
has

Figure 1: The original Design of AutoMail System

## 2.1 Advantages of Original Design

1. **Singleton Design** for the Building Class:

   (a) **Global Access Point**: The Singleton pattern provides a global access point to the Building instance. This makes it easier to manage and coordinate activities within the building.

   (b) **Efficient Resource Use**: By ensuring only a single instance of the Building exists, the system avoids unnecessary duplication of resources, thereby saving memory and processing power that would otherwise be consumed by multiple instances.

   (c) **System-Wide Consistency**: The Singleton pattern guarantees that all components of the system operate on the same Building instance, thereby preventing inconsistencies that could arise from having multiple instances (e.g., conflicting room assignments or occupancy data).

## 2.2 Disadvantages of Original Design

1. **High Coupling** Between Classes:

   (a) **Robot and Building**: The Robot class has a direct dependency on the Building class, leading to high coupling. This high coupling reduces the system's flexibility and maintainability. For instance, any modifications to the Building class may necessitate corresponding changes in the Robot class. Additionally, the Building class contains methods like *move*, which may overlap with the responsibilities of the Robot class, potentially leading to redundant code and further complicating maintenance.

   (b) **MailRoom and Simulation**: The MailRoom class is directly associated with both the Robot and Simulation classes. While some interaction is necessary, the direct and potentially complex relationships between these classes can make the system more difficult to understand, maintain, and extend.

2. **Low Cohesion** Within Classes:

   (a) **Building** Class: The Building class is burdened with a wide range of responsibilities, including managing robot movements, which could be seen as unnecessary. This broad scope reduces cohesion, making the Building class harder to manage and more prone to errors.

   (b) **MailRoom** Class: According to the system's specification, the MailRoom should primarily manage items. However, in the original design, the MailRoom class is also responsible for managing robots, which dilutes its cohesion and deviates from its intended role.

3. **Scalability** Concerns:

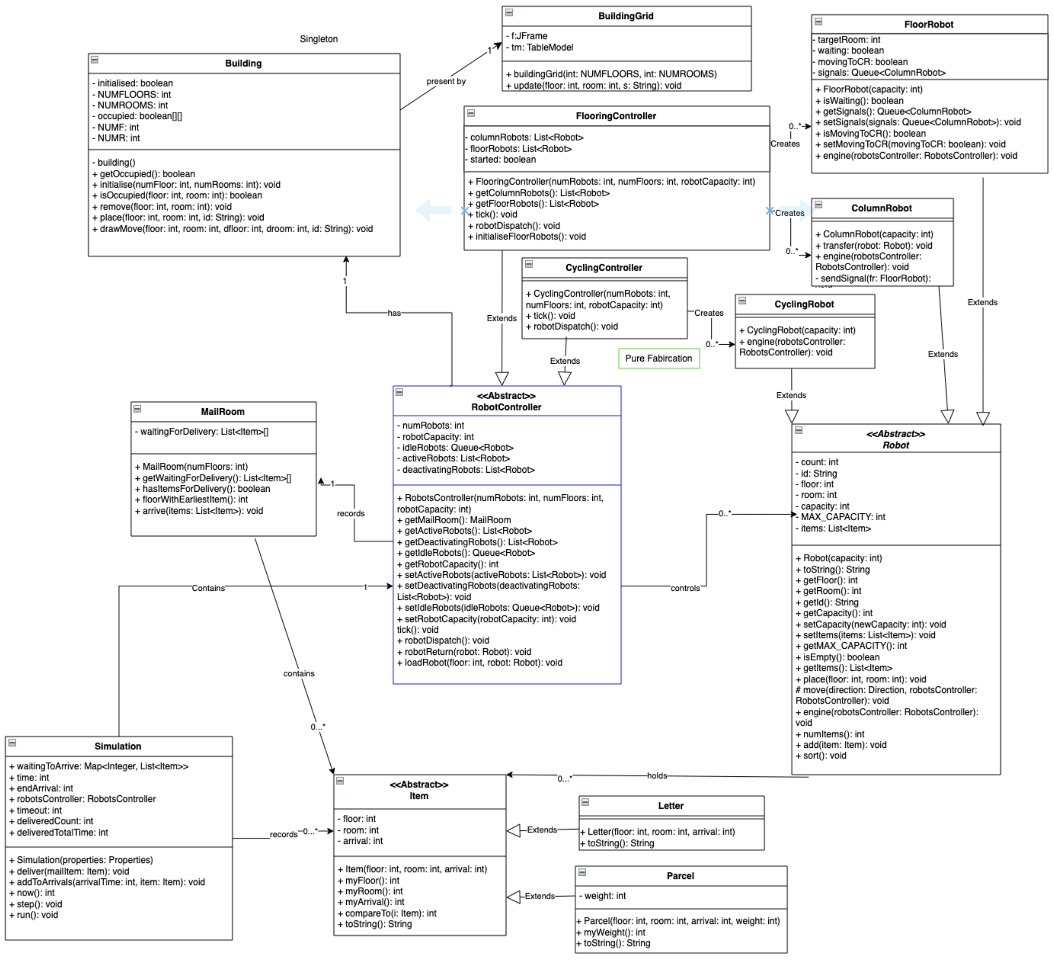
   (a) **Handling New Modes** (e.g., FLOORING Mode): The original design does not easily accommodate the addition of new modes such as the FLOORING mode. For example, since the MailRoom class directly manages robot operations, introducing new behaviors or modes would likely require significant changes to the existing code, making the system less scalable.

4. **Inheritance vs. Composition**:

   (a) **Robot and Item (Letter/Parcel)** Classes: The original design lack of the use of inheritance to manage different types of robots or items (e.g., Letter, Parcel), this leads to less flexible if there are more types of items or robots are introduced in.

## 3 Refactoring, Improvements and Extension

As shown in Figure 1, the original design has been significantly improved by applying GRASP (General Responsibility Assignment Software Patterns/Principles).



Singleton
Building
- initialised: boolean
- NUMFLOORS: int
- NUMROOMS: int
- occupied: boolean[][]
- NUMF: int
- NUMR: int
- building()
+ getOccupied(): boolean
+ initialise(numFloor: int, numRooms: int): void
+ isOccupied(floor: int, room: int): boolean
+ remove(floor: int, room: int): void
+ place(floor: int, room: int, id: String): void
+ drawMove(floor: int, room: int, dfloor: int, droom: int, id: String): void
present by
BuildingGrid
- f:JFrame
- tm: TableModel
+ buildingGrid(int: NUMFLOORS, int: NUMROOMS)
+ update(floor: int, room: int, s: String): void
FlooringController
- columnRobots: List<Robot>
- floorRobots: List<Robot>
- started: boolean
+ FlooringController(numRobots: int, numFloors: int, robotCapacity: int)
+ getColumnRobots(): List<Robot>
+ getFloorRobots(): List<Robot>
+ tick(): void
+ robotDispatch(): void
+ initialiseFloorRobots(): void
FloorRobot
- targetRoom: int
- waiting: boolean
- movingToCR: boolean
- signals: Queue<ColumnRobot>
+ FloorRobot(capacity: int)
+ isWaiting(): boolean
+ getSignals(): Queue<ColumnRobot>
+ setSignals(signals: Queue<ColumnRobot>): void
+ isMovingToCR(): boolean
+ setMovingToCR(movingToCR: boolean): void
+ engine(robotsController: RobotsController): void
ColumnRobot
+ ColumnRobot(capacity: int)
+ transfer(robot: Robot): void
+ engine(robotsController: RobotsController): void
- sendSignal(fr: FloorRobot):
CyclingController
+ CyclingController(numRobots: int, numFloors: int, robotCapacity: int)
+ tick(): void
+ robotDispatch(): void
CyclingRobot
+ CyclingRobot(capacity: int)
+ engine(robotsController: RobotsController): void
Creates
Creates
Creates
0..*
has
Extends
Pure Fabircation
<<Abstract>>
RobotController
- numRobots: int
- robotCapacity: int
- idleRobots: Queue<Robot>
- activeRobots: List<Robot>
- deactivatingRobots: List<Robot>
+ RobotsController(numRobots: int, numFloors: int, robotCapacity: int)
+ getMailRoom(): MailRoom
+ getActiveRobots(): List<Robot>
+ getDeactivatingRobots(): List<Robot>
+ getIdleRobots(): Queue<Robot>
+ getRobotCapacity(): int
+ setActiveRobots(activeRobots: List<Robot>): void
+ setDeactivatingRobots(deactivatingRobots: List<Robot>): void
+ setIdleRobots(idleRobots: Queue<Robot>): void
+ setRobotCapacity(robotCapacity: int): void
tick(): void
+ robotDispatch(): void
+ robotReturn(robot: Robot): void
+ loadRobot(floor: int, robot: Robot): void
MailRoom
- waitingForDelivery: List<Item>[]
+ MailRoom(numFloors: int)
+ getWaitingForDelivery(): List<Item>[]
+ hasItemsForDelivery(): boolean
+ floorWithEarliestItem(): int
+ arrive(items: List<Item>): void
records
Contains
contains
controls
<<Abstract>>
Robot
- count: int
- id: String
- floor: int
- room: int
- capacity: int
- MAX_CAPACITY: int
- items: List<Item>
+ Robot(capacity: int)
+ toString(): String
+ getFloor(): int
+ getRoom(): int
+ getId(): String
+ getCapacity(): int
+ setCapacity(newCapacity: int): void
+ setItems(items: List<Item>): void
+ getMAX_CAPACITY(): int
+ isEmpty(): boolean
+ getItems(): List<Item>
+ place(floor: int, room: int): void
# move(direction: Direction, robotsController: RobotsController): void
+ engine(robotsController: RobotsController): void
+ numItems(): int
+ add(item: Item): void
+ sort(): void
holds
Simulation
+ waitingToArrive: Map<Integer, List<Item>>
+ time: int
+ endArrival: int
+ robotsController: RobotsController
+ timeout: int
+ deliveredCount: int
+ deliveredTotalTime: int
+ Simulation(properties: Properties)
+ deliver(mailItem: Item): void
+ addToArrivals(arrivalTime: int, item: Item): void
+ now(): int
+ step(): void
+ run(): void
records
<<Abstract>>
Item
- floor: int
- room: int
- arrival: int
+ Item(floor: int, room: int, arrival: int)
+ myFloor(): int
+ myRoom(): int
+ myArrival(): int
+ compareTo(i: Item): int
+ toString(): String
Letter
+ Letter(floor: int, room: int, arrival: int)
+ toString(): String
Parcel
- weight: int
+ Parcel(floor: int, room: int, arrival: int, weight: int)
+ myWeight(): int
+ toString(): String

Figure 2: Design After Improvement

### 3.1 Use of Pure Fabrication and Indirection:

1. **RobotController Class**: To reduce direct coupling between key classes in the system, such as Robot and Building, and MailRoom and Simulation, we introduced the RobotController class. This class serves as an indirection layer and is a prime example of Pure Fabrication. The RobotController manages the responsibilities that were previously spread across the Building and MailRoom classes, leading to poor cohesion in those classes.

2. **Improved Cohesion**: By transferring these responsibilities to RobotController, we improved the cohesion of both the Building and MailRoom classes. Now, each

class has a clearer, more focused set of responsibilities, enhancing the system's maintainability and scalability.

## 3.2 Application of the Creator Principle

1. **RobotController as the Creator**: In our redesigned system, the RobotController class is responsible for creating instances of Robot. This responsibility assignment aligns with the Creator principle and significantly increases the cohesion within the RobotController. By centralizing robot creation, RobotController can efficiently manage and coordinate robot behaviors without relying on external classes, thereby reducing coupling and enhancing the overall structure.

2. **Simulation as the Creator of Items**: Similarly, the Simulation class is tasked with creating instances of Items, such as Letter and Parcel. This design choice ensures that the Simulation class maintains control over the lifecycle of items within the system, leading to higher cohesion within the Simulation class. By adhering to the Creator principle, the system becomes more modular and easier to extend, as each class has clear responsibilities for creating and managing specific entities.

## 3.3 Adherence to the Information Expert Principle:

1. **Encapsulation of Responsibilities**: By introducing the RobotController as an intermediary class, we ensured that each class in our redesigned system adheres to the Information Expert principle. For example, in our original design, the MailRoom class handled not only mail item management but also some aspects of robot coordination, which diluted its focus. By shifting robot-related responsibilities to the RobotController, the MailRoom class now exclusively manages mail item deliveries, ensuring it handles only the information it is directly responsible for. Similarly, the Building class no longer manages robot movements directly, which is now within the scope of the RobotController, further streamlining the responsibilities across the system.

2. **Focused Responsibility**: In the redesigned system, the RobotController handles all robot-related operations, such as robot dispatch, return, and task assignment, which were previously scattered across different classes. This change enhances the cohesion of the RobotController by centralizing all robot-specific logic within one class. The Building, MailRoom, and Simulation classes now focus solely on their core responsibilities: Building manages structural aspects like floors and rooms, MailRoom oversees the queue and sorting of mail items, and Simulation manages the overall system flow and timing. This clear separation ensures that each class

operates independently and efficiently, reducing the likelihood of cross-class dependencies and making the system more maintainable and extendable.

## 3.4 Application of Polymorphism

1. Polymorphism in **Item**, **Robot**, and **RobotController** Classes: To effectively manage the different cases and modes within the system, we employed polymorphism and inheritance across the Item, Robot, and RobotController classes. This design choice allows for a more flexible and extensible system, where behaviors can vary dynamically based on the specific subclass being used.

2. **Item Class Hierarchy**: The Item class serves as an abstract base class for both Letter and Parcel. By utilizing inheritance, we can encapsulate shared attributes and methods within the Item class while allowing Letter and Parcel to extend and customize their behavior as needed. This approach reduces code duplication and promotes code reuse, ensuring that any changes to the common behavior are centralized in the Item class, making the system easier to maintain.

3. **Robot Class Hierarchy**: Similarly, the Robot class is an abstract base class for various types of robots, such as CyclingRobot, FloorRobot, and ColumnRobot. Each robot subclass can implement specific behaviors and responsibilities, such as how they move or interact with items. Polymorphism allows the RobotController to interact with any type of robot without needing to know the specifics of each type, thus promoting flexibility and reducing the complexity of the RobotController class.

4. **RobotController and its Subclasses**: The RobotController class itself is abstract, with specific controllers like CyclingController and FlooringController extending it. This design allows each controller to manage a specific mode or type of robot, encapsulating the logic required for that mode. This not only makes the code more modular but also simplifies the addition of new modes in the future, as new controllers can be created without modifying the existing ones.

## 3.5 Implementation of the Flooring Mode:

1. Using the design principles discussed above, we successfully implemented all the requirements for the Flooring mode. The following sequence diagram illustrates the process where a ColumnRobot loads items from the MailRoom, sends a signal to a FloorRobot, transfers the items to the FloorRobot, and finally, the FloorRobot delivers the items.

# 4 Future Development Plan

To ensure our design remains extendable and adaptable for future development, we have incorporated the following:

## 4.1 Application of the Factory Pattern:

Discuss how the system can be extended to support multiple machines of each type, along with any architectural changes required.

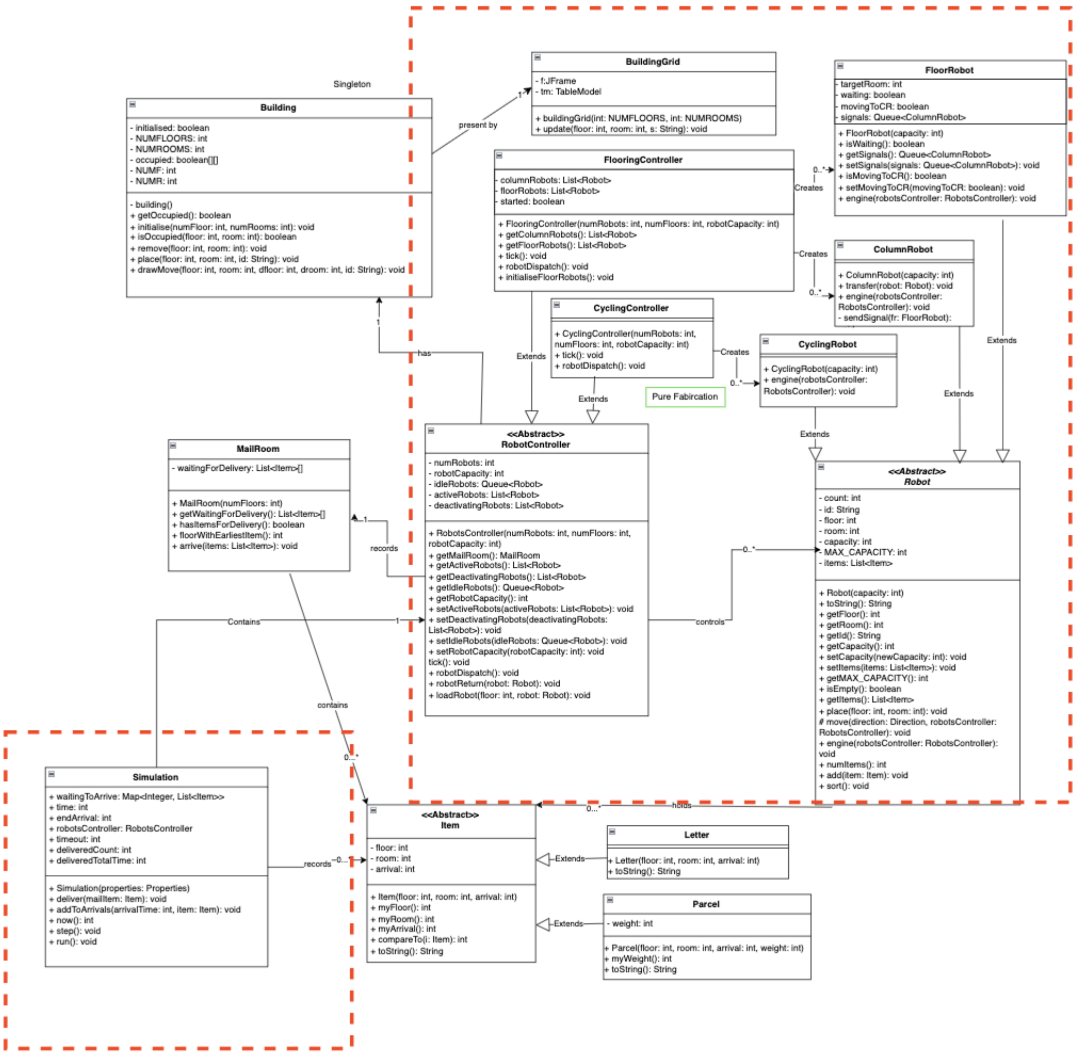


Figure 3: Factory Pattern Applied in Our Design

1. **Encapsulating Object Creation**: The Factory Pattern is utilized to create objects—specifically, the RobotController and its associated Robot types—without specifying the exact class of object to be instantiated. This pattern allows us to encapsulate the object creation process, enabling the system to produce different types of objects based on the input criteria, such as the operational Mode.

2. **Switch Statement in the Simulation Class**:
   (a) Within the Simulation class, a switch statement is employed to determine the operational Mode and instantiate the appropriate RobotController, either CyclingController or FlooringController.
   (b) This aligns with the Factory Pattern as it abstracts the creation logic of the RobotController objects, making the Simulation class more modular and adaptable to future changes.
3. **RobotController Creating Relevant Robots**:
   (a) Once the specific RobotController is instantiated, it is responsible for creating the relevant types of robots, such as CyclingRobot, FloorRobot, or ColumnRobot. This setup allows each RobotController to act as a factory, producing the necessary robot instances required for its respective mode.
   (b) This design is a clear application of the Factory Pattern, where the factory (the RobotController) creates the objects (Robots) needed by the client (the Simulation class).
4. **Benefits of this Design**:
   (a) With this implementation, the Simulation class functions as a client that leverages a factory (the switch block) to generate different types of RobotController. These controllers, in turn, create the specific Robot instances needed for the selected mode.
   (b) This approach encapsulates the object creation process, providing the flexibility to add new modes or robot types in the future without requiring significant changes to the client code. This makes it a robust example of the Factory Pattern in action, contributing to the system's extendability and maintainability.

## 5 Conclusion

In conclusion, the refactoring and extension of the Automail System have significantly improved its design, making it more cohesive, flexible, and ready for future enhancements. By implementing the FLOORING mode, we demonstrated how the system could be extended to accommodate more complex behaviors while maintaining maintainability and scalability. The application of design patterns like the Factory Pattern and GRASP principles has not only solved existing design issues but also laid a strong foundation for future development. Moving forward, the system is well-positioned to adapt to new requirements, such as the introduction of additional robot types or operational modes, with minimal impact on the existing codebase.