Design Analysis Report – SWEN30006
Group 17: Fat Controllers

By Hao Le (695540), Daniel Porteous (696965) and David Stern (585870)

Introduction

The GRASP patterns are object-oriented software patterns that are used to guide good object-oriented design and apply responsibility-driven design - that is, these patterns guide the assignment of the responsibilities of classes. We can use these patterns to guide our analysis of the provided Robot Delivery Simulation, which demonstrates both elements of good object-oriented design, and elements that may be improved.

Good Design Elements

Observation 1: The Controller pattern was implemented quite well by the existing design.
Implementation: The `Simulation` class is the façade controller of the simulation of the Robot Mail Delivery system, in that it manages the coordination of the `Automail` system, the created mail, and the progression of the system through the simulation (and the resulting robot delivery scores). It delegates most of the other possible responsibilities appropriately.

Observation 2: The Creator Pattern has been used correctly.
Implementation: The `MailGenenerator` is a specialised creator of Mail Items, delegated to it by the `Simulation` class. The Simulation class also creates a robot via the `Automail` instance; given that it feeds this robot state updates in the form of new mail deliveries and steps, and given that records the performance of the robot, it seems appropriate that it creates the Robot.

Observation 3: The Information Expert pattern was implemented mostly correctly.
Implementation: This can be seen in that the `Building` class knows its Floors, Lowest floor, and Mailroom Location. This can be seen in that the Clock is responsible for keeping track of the time, and thus knows the last delivery time, and the current time. This can be seen in that the `MailItem` is responsible for knowing its destination floor and arrival time, just as a real mail item would. The `StorageTube` knows its maximum capacity, the number of mail items in itself, etc. The list goes on, and this pattern is *generally* well implemented.

Observation 4: `MailItem` and `PriorityMailItem` have been designed in accordance with the Polymorphism pattern; that is, default behaviour has been managed by the superclass, and the sub-class manages more specific sub-class behaviour. This also makes sense in that it makes the design model more reflective of the domain model. For example, Priority Mail Items *are* a type of Mail Item, so they are, in the problem domain, a sub-type of mail items.

Without the requirement for different robot configurations, the different Robot Behaviours have been managed well by the `IRobotBehaviour` interface; there's no default robot behaviour, so this manages the different types well.

Bad Design Elements

Observation 1: Cohesion for the robot behaviours is a future concern - Polymorphism should be better implemented.
Implementation: Given that different robot configurations will be used in the future, if these were to be implemented in accordance with the existing design, it would result in relatively low cohesion for these classes, in that they will end up having access to more information (methods and attributes) than they need to execute their behaviour. A better design would use inheritance to define subtypes of robot configurations from which different robots can inherit. An example is a subset of robots that have the ability to communicate with the mailroom.

Observation 2: The Information Expert wasn't perfectly implemented and should be changed to lower coupling.
Implementation: `MailGenerator` is an information expert on `MAIL_TO_CREATE`, yet `Simulation` is the class that initialises `MAIL_TO_CREATE`. `Simulation` doesn't use its own version of this, aside from initialising it and passing it into the `MailGenerator` constructor. The only time it does use this is when it takes this data from the `MailGenerator` instance. Thus, this is a diversion from the information expert pattern. `Simulation` is the information expert on mail arrival and priority mail items arriving, and it manages the system output. Yet, `IRobotBehaviour` classes are printing out the arrival of priority items. `Simulation` is the Information expert on mail arrival, and thus should have responsibility for this action. This is made even more clear when considering that robots without communications attached won't be receiving messages of priority arrivals - having this behaviour in the `RobotBehaviour` classes makes for more delicate and hard-to-maintain classes. Thus, placing this behaviour in the `Simulation` class leads to lower coupling.

Observation 3: System output is strewn all over varying classes, decreasing cohesion.
Implementation: Classes like `Robot` and `MailGenerator` are information experts on events like the `Robot` changing state, and the `MailGenerator` is an information expert on Mail Arrival. However, their responsibilities aren't focused enough when these classes are responsible for system output. To increase cohesion, System output should be managed by a class that is purely responsible for that, while each class is still responsible for actually creating the message, maintaining the information expert pattern. While this would result in higher coupling, it's not higher coupling that is the problem. Instead, the problem occurs when the coupling is with elements that are 'unstable', or have a likelihood to change in a future iteration / implementation. System output in these circumstances is not likely to change in any significant way, and as such this higher coupling is not an issue.

Observation 4: Visibility Modifiers are inappropriate.
Implementation: Throughout the system there are visibility modifier issues. `MailItem` has three constants (`DESTINATION_FLOOR`, `ID`, and `ARRIVAL_TIME`) which have the `protected` visibility modifier. From a programming practice perspective, `protected` is dangerous in Java because it allows anyone to subclass your class and modify the protected attributes of the superclass. Beyond this, there is no reason for another class to be able to see these attributes directly, and as such they should all be `private`. `Robot` has some instance variables (`tube`, `delivery`, and `current_state`) which should also be `private`, as well as `MailItem` with its `tube` instance variable. These are all examples of breaking of encapsulation.

Observation 5: A great number of comments are missing from classes throughout the system. `MailPool` lacks JavaDoc comments for all of its methods, as does `SmartRobotBehaviour`, `SimpleRobotBehaviour`, `Simulation`, etc. This makes reading and understanding the code more difficult than necessary and is bad practice. Better, more articulate comments, and complete compliance with JavaDoc commenting would be an expected improvement should this code be shipped to the client, so that the client may more easily make future adjustments.

Summary

This design implements a lot of the GRASP patterns relatively well, however it is indeed flawed, and like all software can be improved. The Simulation software achieves its goal of simulating Robotic Mail Delivery Systems, however it is not built in a way that is easily extensible, and the classes are overall not as cohesive as they could be (their cohesion is perhaps too low). While the classes are overall highly coupled with one another, this high coupling is mostly not likely to cause issues, as the elements that are coupled are all quite stable in nature. The improvements specified in this report, if implemented, should improve the overall design and lead to better extensibility.