Design Analysis Report
Group 52

The design analysis report will discuss the changes our group has made to improve the current software design. In addition, it will also suggest a few possible future changes which would make it easier to manage the system.

1. Design Improvement

1.1 Robot subclass - Polymorphism

In the original design, the robot class was only responsible for representing standard and weak robots. In order to extend the specification, we have made two design changes to it. Firstly, two new attributes were introduced to be used in constructor, namely *careful* (whether the robot can take fragile mailItem), and *capacity* (maximum tube capacity). This provides flexibility for extending current design to four robot types. Secondly, we used inheritance to achieve **polymorphism**. Four subclasses are created (*WeakRobot, StandardRobot, BigRobot, CarefulRobot*) to represent all types of robots. Robot creation is performed in subclasses with appropriate *super()* method. With subclasses, robots can easily be created with their respective types. We also decided to make Robot *abstract*, since it makes no sense to create a robot without knowing its type.

1.2 Reduce coupling – MailGenerator, IMailPool and Simulation

From the initial design, it is clear that there has been high coupling among **MailGenerator**, **IMailPool** and **Simulation** classes. MailGenerator was designed for generating all mails. It also has an attribute called *mailPool* for automail creation. However, inside Simulation *main()* function, *mailGenerator* and *mailPool* were created for simulation. This makes a high coupling relation. In addition, MailGenerator has a *step()* function to return *priorityMailItem*. This violates the principle of **information expert**, since MailGenerator should only be used for generating *mailItem*.

To lower coupling, we decided to remove *mailPool* attribute and *step()* method from *mailGenerator*. Instead, the simulation will perform the action. Where it needs information for all mailItems, it calls the getter in *mailGenerator*. As a result, we got rid of the relationship between *MailGenerator* and *IMailPool*, and reassigned responsibility for low coupling.

1.3 Reduce coupling – Robot, MailItem and StorageTube

In the original design, there is unnecessarily high coupling among **Robot**, **MailItem** and **StorageTube** classes. Robot class contains an attribute called *tube*, which is a stack of mail items. On the other hand, Robot class stores *deliveryItem*, which is the next mailItem to be delivered by this robot. This means that the three classes form a triangle relation, violating the principle of **low coupling**.

Since *tube* contains all mails to be delivered, it is unnecessary to have the attribute *deliveryItem* in Robot. To reduce coupling, we remove *deliveryItem* from Robot. Wherever this robot needs to know which item should be delivered next, we use *tube.peek()* to obtain

the first item in the storage tube (without removing it). When we need to remove it from tube, simply pop it from the stack. As a result, *Robot* and *MailItem* are no longer related, which reduces coupling and adds cohesion to the design.

1.4 Reduce coupling – Automail, Robot and IMailPool

The final place where we noticed a significantly high coupling is among **Automail**, **Robot** and **IMailPool** classes. Both Automail and Robot classes have an attribute *IMailPool mailPool*, which stores all mails shared among robots. Automail also initializes all robots. This relation makes coupling high among three classes.

Therefore, in order to reduce coupling, notice that mailPool is public and the same for all robots, we decided to remove this attribute from Robot, and use *public static IMailPool mailPool* in Automail. We no longer need *mailPool* in Robot constructors (also in Adaptors during robot creation). We call *Automail.mailPool* whenever robot needs to access *mailPool*. By doing this we lowered coupling for *Robot* and *IMailPool*.

1.5 Robot Creation – Adaptors, Factory and Singleton

One of the biggest changes our group has made to strategies is related to robot creation. Originally, robot creation was hard-coded in *Automail* class. This makes future modification difficult, and is against the idea of **Creator** in GRASP, since Automail should just be used to add robots to ArrayList rather than actually create them. To solve the problem, we decide to use **RobotAdaptor** together with **RobotFactory**.

RobotAdaptor is an interface for robot creation. Four types of adaptors implement it (*BigRobotAdaptor* etc.) to directly create robots. It adds a level of indirection to varying APIs in other components. Using adaptors provides multiple benefits such as component hiding and **protected variation**. Adaptor is also an example of **polymorphism**, and uses **pure fabrication** to finally achieve **low coupling** and **high cohesion**.

We created a pure fabrication object, robot factory, to handle the creation of adaptors. It is an **abstract factory** pattern. It allows responsibility separation and potential strategy improvements in the future such as object caching. We decided to use a **singleton** as only one instance should be allowed to be created. Current robot specification is simple, therefore the factory is relatively small. However, with these design patterns, we delegate creation of objects to Abstract Factory and make it much easier to handle future changes.

1.6 Misplaced Constant – Information Expert

Constants should be placed into appropriate classes, to reflect the idea of **information expert**. In the original design, *LAST_DELIVERY_TIME* (The threshold for the latest time for mail to arrive) attribute is put in the *Clock* class, which is inappropriate because the class is only a representation for clocking. This is clearly a responsibility issue. To solve that, we placed the attribute inside *MailGenerator* class as *LAST_DELIVERY_TIME* is only used in mail creation. This assigns responsibility to the information expert.

2. Further Changes / Considerations

2.1 More Robot Types

In terms of more robot types, we can create more subclasses to inherit from Robot. For instance, after introduction of a *JunkRobot*, which detects junk mails, adds them to *junkPool* and discards them, we need another subclass of *JunkRobot* that inherits from Robot. If there are too many robot types, we might add an interface for the Robot superclass to manage Robot types more flexibly.

2.2 Robot Behaviour Extension

Our current design for robot creation makes it easier for robot behaviour extension in the future. With the *RobotFactory*, any further robot actions / requirements can be added to the factory, and adaptors are used to separate responsibility of complex creation logic. For example, robots might need to be created with different responsibilities, such as assigning a floor range to deliver mails. This can be implemented with ease by expanding *RobotFactory* with additional components.

2.3 Property Extension

In the original design, *automail.properties* was read in the main function in *Simulation*. In the future, this could be a problem when more properties are added to the specification. For example, we may want to specify the speed of each robot, or the total number of robots needed (e.g. given a limited budget and price for each robot, we wish to maximise efficiency). This can make our main function hard to manage.

Therefore, we could design a new class called **PropertyManager** to handle property inputs uniformly, rather than do it in main function. The class will contain all default properties and they will be overwritten if there are new parameters. *PropertyManager* is designed to solely handle automail properties, which follows the idea of **pure fabrication** and **information expert**. It helps us manage file input more easily.