SWEN30006 Design Analysis Report

Mingfeng Li 877347

Zhe Ji 877384

Elijah Foster-Mclachlan 833460

1. Introduction

The original mail delivering system implemented by Robotic Mailing Solution Inc. includes two primary components, delivery robots and a Mailpool subsystem. The limitation of this system is that the current system only has a weight limit-based task for each robot, and the requirement of this system is to implement a team task for improving capacity of delivery item weight. In detail, for each delivery item which exceeds the single robot capacity requires pairs of triples robots to coordinate for improving effectiveness of the system, but for the lighter mail items, the system requires to preserver original capacity. In addition, for implementing this requirement, the move speed of a team robot will be slower than individual robot. Therefore, the new version of this mail delivering system will refactor and extend from the original system. This report will describe the modifications of the Automail system, and critically discuss the new design and other optional designs.

2. Original System

For delivering heavier mail items, we need to use two or three robots to increase the performance of this system. It can just change the attributes or methods and don't change the structure of original system to achieve.

In the original system, there are two classes implemented to allocate mail items and to deliver them, **MailPool** and **Robot**. Firstly, the **MailPool** class implements three methods from **IMailPool** interface including **addToPool**, **step** and **registerWaiting**. The **step** method assigns each available robot which is added by **registerWaiting** method to carry the mail items by invoking **loadRobot** function. In the **loadRobot** function, the mail pool puts the mail item into a robot's hand or tube. Then, if the weight of the added item exceeds to the maximum carrying capacity of individual robot, the robot will throw an exception called **ItemTooHeavyException**. Secondly, the move speed is defined at **moveTowards**. In detail, the **Simulation** calls the **step** method of each robot instance every software defined clock, and the **step** method determines the next behaviour according to the **current_state**. For both DELIVERING and RETURNING states, the **moveTowards** will called for walking up or down floor. Furthermore, the behaviour about moving of individual robot is defined as one floor for each clock time.

According to the description of original system, there are several considerations of improvement to implement the requirements. Firstly, the problem about which class has responsibility for assigning the delivery items to a single robot or a team robot should be discussed. Secondly, the problem about how to implement a team behaviour in the new system should be considered.

3. Analysis and Solution

3.1 Assigning Delivery Item

In the original system, the **Robot** responses whether a mail item is overweight. However, for the requirement of a team behaviour, the important consideration is to refactor this responsibility.

3.1.1 First Optional Design

The first optional design proposes to assign this task to the **MailPool**, namely the **MailPool** allocates different items to different robots according to the weight. The advantage of this refactoring is that the responsibility of determining whether an individual or team behaviour is invoked has a clear reason. The reason is that a single robot only focusses on the added item, and it decides to throw an **ItemTooHeavyException** or to deliver the input item. If a single robot takes the responsibility for assigning heavier items, the system will reduce cohesion. Therefore, for implementing a new team robot behaviour of the system, the **MailPool** or other relevant classes except the **Robot** should take the responsibility for assigning delivery items according to the value of weight. Furthermore, if the **MailPool** assigns mails, it will take responsibility for creating a team behaviour which can carry heavier items by pair or triple single robots. However, this approach reduces the cohesion of the system as well.

3.1.2 Second Optional Design

The second optional design is similar to the first discussion, but the difference is that the responsibility of creating a team behaviour is refactored by extending a creator called **RobotSimpleFactory**. In detail, the **loadRobot** method of **MailPool** puts a mail item into the creator, and the creator will determine to assign the mail to a single robot behaviour or a team behaviour. Therefore, the creator can improve the cohesion of the system. To implement the creator discussed above, the factory is a suitable design pattern. The **RobotSimpleFactory** is created as a pure fabrication object to response different types of robots to **MailPool**.

Based on the discussion, the second optional design is better than the first one, and it will take into development.

3.2 Design of Team Behaviour

In the requirement, the team behaviour is similar to a single robot behaviour. However, there are some considerations to discuss.

3.2.1 First Optional Design

The first optional approach is to aggregate pair or triple robots as a team directly, namely the system will not create a new object to take a team behaviour. For example, the **MailPool** assigns an item which is in the PAIR_MAX_WEIGHT to two available robots. Then, these robots are marked as a team, and their original method **moveTowards** will refactor to a team speed (one third of individual robot). The disadvantage of this design is that it reduces cohesion and improves coupling of the system. Particularly, the state of a robot in a team depends on another robot. Therefore, this approach will cause a difficulty to reuse and understand.

3.2.2 Second Optional Design

The second method is to create a new class named **Team**, which completes behaviours of delivering heavier mail items with two or three robots. In fact, **Robot** and **Team** have some common methods and attributes. Thus, creating an abstract base class named **SuperRobot,** which contains common attributes and methods of **Robot** and **Team**, for avoiding repetition. And it declares some abstract methods, which should be overwritten in **Robot** and **Team** respectively to complete the different behaviours of them. **Robot** and **Team** will extend from **SuperRobot**, they are child classes of **SuperRobot**. Other classes can just use the base class, **SuperRobot**, to replace the class of Robot and Team, which is the basic characteristics of Java polymorphism. The main purpose of this approach is that a creator described above can take the place of **Robot** in the **loadRobot** of **MailPool**, because both **Robot** and **Team** extend from **SuperRobot**.

3.2.3 Third Optional Design

From discussed above, **Robot** and **Team** have the common methods that it can define an interface **RobotBehaviours**. This approach provides a stable interface for implementing the pure fabrication principle. **Robot** and **Team** both implements this interface. It is better to extend new classes and new functions in the future. But there are still some disadvantages of it. These two class have many similar attributes to indicate the status of them and same behaviours. If using this structure, **Robot** and **Team** will have replicate code of this part.

3.2.3 Fourth Optional Design

The Fourth option is the combination of the second and third, to create a base class containing the common attributes and methods of **Robot** and **Team**, and an interface defining different methods of them. **Robot** and **Team** both extend from this base class and implements their own methods from this interface. It seems to be best option to overcome all disadvantages. In fact, methods defined in the interface must be public. However, some of these methods should be secure that only can be accessed by itself. So, this option does not the better option for the security of code.

3.3 Other modification

In the new mail delivering system, the **Automail** is refactored to Singleton pattern. Therefore, the factory can put available robots from the list of robots into a **Team** instance.

4. Conclusion

To conclude, the new system extends a new class called **SuperRobot** for generalisation that takes common behaviour from **Robot** and **Team** into another object. Therefore, the refactored **MailPool** can assign a mail item to **RobotSimpleFactory**, and this creator will response a specific object, **Robot** or **Team**, according to the weight of an item.