Project 1 Automail – Design Analysis

SWEN30006 Software Modelling and Design

Author1(Hongyu Su), Author2(Zexu Huang), Author3(Xubin Zou)

Introduction:

This report will focus on the pattern we have applied in the project. Also, we will go through the operation process of the program.

There are two main steps we have taken to finish this project:

1. We have drawn the static design model to analyze the problem and design our solution.
2. We have written the code part in java according to the design model we draw before.

Assumption:

1. There is at least one robot in the building.
2. One robot can carry 0 – 2 mail items.

Summary:

This project used the ***Factory pattern*** and **General Responsibility Assignment Software Patterns (GRASP)** to apply to our design. Also, the **Open-Closed Principle** is the central concept we would like to meet. We have a *MailFactory* class to handle the creation of all the *mailItem*. Also, we create the *Charger, ServiceFee,* and *ActivityCost* to handle the fees we have in the current situation. We create the *Robot* class for the robot, which is abstract, then the *NormalRobot* class inheritance from the *Robot* class. Finally, we have the *feeFinder* class to handle the *ServiceFee* and *StatisticRecorder* class to do the statistic printing.

Operation Process:

First of all, we use the *propertyReader* to read all the information from the property file. If the *ChargeDispaly* is True, we will need to show the charge information at this time. *MailGenerator* will create a *MailFactory*. The MailFactory will create all the mail we received this time. After that, we add these *MailItem* into *MailPool*. In the *MailPool*, they will be sorted by their estimated charge when they add to *MailPool*. Then the *MailPool* will check if the robot in the *MailPool* has a space to load the MailItem. If yes, they will load the MailItem and start to deliver. Until the robot arrives at the destination floor, the robot will use the *FeeFinder* to find the ServiceFee of the current floor. There are two situations. If the robot finds the ServiceFee successfully, we will store the ServiceFee and this floor into the HashMap and then return it to the robot. In the last, we add this into total ServiceFee and ActivityUnit + 1. If the robot cannot find the ServiceFee, the robot will go to check the HashMap. If the HashMap already contains the ServiceFee of this floor, the robot repeats the success situation. If there is no match in the HashMap, the robot will set the ServiceFee to 0. Then the robot will calculate the returning fee from the current floor. In the meantime, the robot records the information that StatisticsRecorder needs and sets ActivityUnit and all the fees to 0. In the last, we deliver this MailItem, and check does the robot has the next MailItem need to send. If it has, repeat the process. If it has not, the robot will go back to MailPool for waiting.

Patterns and Principles:

As the summary above, we have the *MailFactory* class to create the *mailItem* (See Graph 1 in Reference). The source code only has

MailGenerator to produce the *MailItem* and put them into the *MailPool*. It let the MailGenerator has two responsibilities. So, we separate the responsibility of production of *mailItem* by creating the MailFactory class. Thus, we can let the MailFactory handle all the creation of the mailItem, and other classes do not need to care about their creation. Also, because we have the *mailItem* class and the *normalMailItem* class inherits from the *MailItem*. This inheritance and *MailFactory* meet the **Open-Closed Principle** to allow any other types of *mailItem* add into our system without any change in the source code. Also, it decreases the coupling.

We have *NormalRobot* class and *Robot* class for the robot, the *NormalRobot* class's parent class. We can add any new type of robot into our system without changing our source code because of these two classes. For example, if we want to add a new type of robot with different functions, we can add a new type of robot and inherit the *Robot* class. This design meets the **Open-Closed Principle** and the **Polymorphism**. There is only a *Robot* class in the original design, which is hard to add any new type of robot in the future.

For the new feature of Automail system, charge, we said we had created three classes (*Charger, ActivityCost, Service Fee*) to complete it (**See Reference Graph 2**). We separate the *ActivityCost,* and *ServiceFee* from Charger allows us to add any other fees in the future without significant change in the *Charger* class code. *Charger* class is the class that tries to meet the **Pure Fabrication** of **GRASP**. Because of this class, the *Robot* class and *MailPool* class only need to connect to the *Charger* but do not need to connect to the *ActivityCost* and *ServiceFee*. So, we can reduce the coupling by this. This design also meets the **Open-Closed Principle** and the **High Cohesion** of **GRASP**. For example, if the *ActivityCost* has any problem happened, it will not affect the *ServiceFee*. Therefore, it can improve the efficiency of fixing and adding a new type of fee.

In the *ServiceFee*, we need a feature that can get the fee from *WifiModem*. Therefore, we create a *FeeFinder* class to handle this feature. Without this class, the simulation class will directly connect to *WifiModem*. In original design, if any problem happened with the WIFI system, our simulation system will be affected. By creating the *FeeFinder* class, *Simulation* can use this class to connect to *WifiModem*. Therefore, even though the WIFI system has problem, the *Simulation* class will not be interrupted. This class tries to meet **the Indirection** of **GRASP**. It can prevent the direct coupling between *WifiModem* class and *the Simulation* class. Because of this, we can decrease the coupling of the whole system.

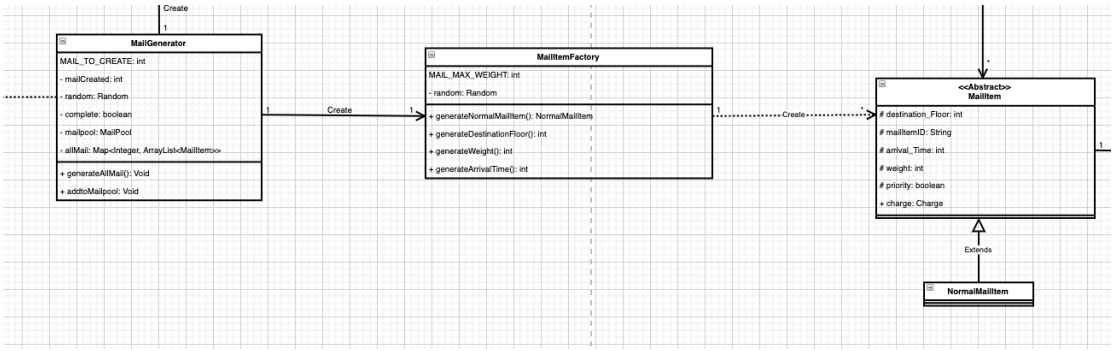
For *StatisticsRecorder*, this class is used to record all the information we need in the current situation. This class is trying to achieve the **High Cohesion** of **GRASP**. This design can help the programmer easily add new data that want to be recorded into the system without changing the whole system. In the source code, the record has been done in the *Simulation* class. We think the source code design will affect the *Simulation* class when the information cannot be found. This is the reason why we separate *StatisticsRecorder* from *Simulation*.

For the last, we have *PropertiesReader* to handle the information of property file. This class is an example of **Pure Fabrication** of **GRASP**. *PropertiesReader* is a class that does not exist in the real world. This class can create the link between the property file and the simulation system. It will not increase the coupling.

Reference:

The complete static design model is in the Automail file.

Graph 1:



Graph 2: