**Project 1 Automail – Report**

**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 done to finish this project:

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

**Assumption:**

**Summary:**

In this project, we used the ***Factory pattern*** and **General Responsibility Assignment Software Patterns (GRASP)** to apply to our design. In addition, **Open-Closed Principle** is the main 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 current situation. For the robot, we create the *Robot* class, which is abstract, then the *NormalRobot* class inheritance from *Robot* class.

At the last, we have *feeFinder* class to handle the *ServiceFee* and *StatisticRecorder* class to do the statistic printing.

**Operation Process**

**Patterns and Principles:**

As the summary above, we have the *MailFactory* class to do the creation of the *mailItem*(See the Graph 1 in Reference)*.* In the source code, it only has *MailGenerator* to produce the *mailItem* and also put them into the *MailPool*. It let the MailGenerator has two responsibilities. So, we separate the responsibility of production of *mailItem* by create the MailFactory class. Thus, we can let the MailFactory to handle all the creation of the mailItem and other classes do not need to care about the creation of them. Also, because we have the *mailItem* class and the *normalMailItem* class inherit from the *mailItem*. This inheritance and *MailFactory* meets the **Open-Closed Principle** to allow any other types of *mailItem* can easily added into our system without any change in the source code. In addition, it decreases the coupling.

For the robot, we have *NormalRobot* class and *Robot* class which is the *NormalRobot* class’s parent class.

Because these two classes, we can add any type of robot into our system without changing our source code. For example, if we want to add a new type of robot which has different function, we can add a new type of robot and have an inheritance of *Robot* class. This design meets the *Open-Closed Principle* and the *Polymorphism*. In the original design, there is only a *Robot* class which is hard to add any new type of robot in the future.

For the new feature of this design, charge, we said we have created three classes (*Charger, ActivityCost, Service Fee*) to complete it (See Reference Graph 2). The reason we separate the *ActivityCost,* and *ServiceFee* from Charger is allowing us to add any other fees in the future without significant changing in the code in *Charger* class. *Charger* class is the class which try to meet the **Pure Fabrication**of **GRASP***.* Because of this class, the mailItem can only 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 new type of fees.

In the *ServiceFee*, we need a feature which can get the cost from *WifiModem*. Therefore, we create a *FeeFinder* class to handle this feature. Without this class, simulation class will directly connect to WIFI Modem. However, if there are 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 WIFI Modem. Therefore, even the WIFI Modem has any problem, the *Simulation* class will not be interrupted. This class try to meet **Indirection** of **GRASP**. It can prevent the directly coupling between *WifiModem* class and *Simulation* class. Because of this, we can decrease the coupling of whole system.

For *StatisticsRecorder*, this class is used for doing the record of 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 record things into the system without changing the whole system. In the source code, record is been done by *Simulation*. We think the source code design will affect *Simulation* class when the information cannot be found. This is the reason why we separate *StatisticsRecorder* from *Simulation*.

**Reference:**

Graph 1:

图片包含 室内, 门, 钟表

描述已自动生成

Graph 2:

图示

描述已自动生成