SWEN30006 Software Modelling and Design

Automail Project B – Design Analysis Report

The University of Melbourne

Team 35

Shuai Wang: shuaiw6@student.unimelb.edu.au

Zhenxiang Wang: zhenxiangw@student.unimelb.edu.au

Tzu-Tung Hsieh: tzutungh@student.unimelb.edu.au

1 Introduction

The original Automail system framework did not consistently apply good design principles. The aim of this project is to extending the design and associated implementation of the system to support the additional robot types (see Figure 1), with consideration for future robot types. This report will firstly identify some good design choice of the original system, then illustrate the implementation of extending the system to support new robot types, and finally discuss the refactoring recommendations of the system according to "General Responsibility Assignment Software Patterns" (GRASP) and other design principles.

Figure 1. Artistic representation of our robot

2 Good Design Choices

Some design choice of the original code meets the GRASP patterns well.

2.1 MailGenerator Class

Firstly, the design of MailGenerator class is in line with the "Creator" pattern. The MailGenerator class has all the initialization information of MailItem class that will passed to MailItem when it is created, so the MailGenerator class is an expert on creating MailItem. Secondly, the MailGenerator class also conforms to the "Pure Fabrication" pattern. This class is not a real-world concept, but an artificial or

convenience class. Introducing this fabricating class can help support high cohesion, low coupling and reusability.

2.2 IMailPool Interface
The IMailPool interface meets the pattern of "Polymorphism". When MailPool class alternatives vary by type in the future, this design is easy to add extensions required for new variations. It reduces coupling and prevent the system from variation.

3 Extending for New Robots

There are two types of robots in project part A: Standard and Weak robots. In project part B, the Automail system is required to support additional robot types: Big robot, Careful robot, and future robot types. The information required to model a robot is in the Robot class. For additional robot types, they contain common attributes and methods with Robot class. Obviously, they just need to inherit these attributes and methods from Robot class and add their own features. This way is convenient and cheap to support future robot types. It follows "Polymorphism" pattern.

What we have changed:

(1) Change the visibilities of most attributes and moveTowards method in Robot class to protected in order to make each type of robot can inherit those attributes and methods.

(2) The differences between those four robot types or future robot types are:
- maxTake: the number of mails the robot can carry.
- strong: if the robot can take heavy mail (>2000).
- canCarryFragile: if the robot is able to carry fragile mail.
- moveTowards method: The movement towards the destination. Standard, Weak and Big robots go two floors, while a careful robot just goes one floor. For Careful robot, it has to override this method to implement half speed of movement.
- To support different Careful robots with different speed of moving, we add a new attribute for Careful robot: SPEED.
- The settings for those attributes above are showed in table 1.

(3) For a future robot type New robot, we can create a class like NewRobot derived from Robot class. According to specific requirements, set these three attributes and override moveTowards method.

RobotType	maxTake	strong	canCarryFragile
Standard	4	true	false
Weak	4	false	false
Big	6	true	false
Careful	3	true	true

Table 1. Default attributes values for robots

4 Refactoring

Some design choices of the original code need to be refactored.

@automail package

4.1 Clock Class
All the variables and functions in the Clock class are static, which are not polymorphic and cannot be overridden (for most languages) in the future. In addition, most remote communication mechanism does not support remote-enabling of static methods. It is better to change this class to a "Singleton" pattern. The name of two functions are changed (Tick to doTick, Time to getTime) to increase the readability of the code.

4.2 Item Class
To develop a better strategy for sorting and selecting the mail for delivery, use Item class to encapsulate mailItem. Even it is just an inner class for MyMailPool, we think it should be taken out as a separate class. This is because it can make the design structure more clearly and simply to understand, and also make design diagram clearer.

4.3 ItemComparator Class
As the same reason with MailItem class, we take ItemComparator class out MyMailPool class. This class is used to sort mails in mailPool.

4.4 MailGenerator Class
To protect privacy, the "MAIL_TO_CREATE" variable should be "private" instead of "public". According to "Open-close Principle", variables should be open for extension, but closed for modification (Meyer, 1988). Therefore, a getter function of the "MAIL_TO_CREATE" variable is added to enable other class to invoke this variable but there is no setter function so that it cannot be changed by other class.

The "HashMap<Boolean, Integer> seed" parameter in the constructor of "MailGenerator" is too complex and unnecessary. To improve the code readability, we can replace it with a nullable primitive type "Integer", using "null" to represent that there is no explicit seed and using a non-null value to represent a seed. After modifying this parameter, the "HashMap<Boolean, Integer> seedMap" variable in the Simulation class and the relative part is no more need. Such changes simplified the code and improves readability.

4.5 PropertiesHelper Class
In order to further increase the cohesion of the Simulation class, it is better to make a PropertiesHelper class to handle all the properties' reading work. It follows the pattern of "Pure Fabrication". This is also consistent with the principle of "Separation of Concerns". Those information can be obtained to configure system by getProperty method.

4.6 ReportDelivery Class
To reduce the amount of code in main method in Simulation class, we create ReportDelivery class to print delivered Item information and calculate a score for an Item and total score. It follows "Pure Fabrication" pattern and "Separation of Concerns" principles.

4.7 Simulation Class
The ReportDelivery class should not be regarded as an internal class in Simulation class since the class is too bloated and does not conform to the pattern of “High Cohesion”. Instead, it should be put under the automail package. In addition, the “ArrayList<MailItem> MAIL_DELIVERED” variable and “double totalScore” variable should be put into the ReportDelivery class. The calculateDeliveryScore function and printResults function should also belong to the ReportDelivery class since they are only used in this class.

The “MAIL_TO_CREATE” variable should not be a member variable of the Simulation class since it doesn’t describe the attribute of the Simulation class. It should be defined as a local variable, read from property file, and passed to initialize the MailGenerator object. The “MAIL_TO_CREATE” variable should be an attribute of the MailGenerator class.

4.8 Storage Tube Class
A different robot type can load up different number of mails in its storage tube, so change the type of maximumCapacity from constant to variable. A storage tube also should have the canLoadFragile to indicate if it can load fragile mail.

@Exception package

We add three more exceptions.
4.9 FragileItemNoCarefulRobotException
This exception will be thrown when there is no careful robot in automail system but mailPool received fragile mail(s). This is because no robot can deliver those fragile mail(s).

4.10 NoRobotTypeException
This exception will be thrown when a robot type in “automail.properties” file does not exist.

4.11 TooManyFragileItemException
This exception will be thrown when a careful robot takes more than noe fragile mail.

@strategies package

4.12 Automail Class
The robots used by Automail Class is hard-coded, which is hard for extension in the future. To solve this problem, we firstly read the robot types from the property file and transfer it as a parameter to the Automail constructor. Then, we generate each robot according the robot types from the property file.

4.13 MyMailPool Class
If only according to “Polymorphism” and “Information Expert” pattern, fillStorageTube function should be put into the Robot class. Each subclass of Robot overrides this method to allow different types of robots to perform different implementations. Such a design would reduce a lot of conditional judgments about robot types and be more polymorphic. However, this design is not in line with the responsibilities of mailPool

and robot. The responsibilities of mailPool is to decide how many mail items, which mail items, and the order of the mail items given to the robot, while the responsibilities of robot is to simply deliver mail items in the provided order. Putting fillStorageTube function into Robot class will split the “Strategies” responsibility, which is not conform to the principle of “Separation of Concerns”, so we finally did not choose to do so.

5 Conclusion

In conclusion, this report demonstrates the good design choice and limitations of the original code and extends the system to support the additional robot types based on “Polymorphism” and “Protected Variation” patterns. In addition, some refactoring implementations are made to make the system flexible, extensible and modifiable. Though our design is far from perfect, this process helps us gain a better understanding of how to improve software modeling and design.

6 Reference

Meyer, B. (1988). *Object-oriented software construction* (Vol. 2, pp. 331-410). New York: Prentice hall.