Project 1: Automail – Design Analysis

SWEN30006 Software Modelling and Design

Ted Marozzi (910193), Fei Yuan (913503), Jake Hum (914666)

1 Introduction

Automail provides clients with an automated mail sorting and distribution system. Originally designed solely for mail, Delivering Solutions Inc. seeks to expand their addressable market to include food delivery services within buildings. Our team has been brought in to extend the functionality of the current system, while ensuring the system retains its existing functionality. We also had a concurrent focus in improving the software design of the existing system to be extensible and allow for continued improvements in the future. This was achieved through an iterative process of modelling and refactoring of the system, and the application of established software design principles and patterns. This report will outline our final solution, the patterns and principles that have been applied, as well as the various alternatives that were considered throughout this process.

2 Solution Design

2.1 Summary

The main changes to the existing software are as follows:

- Created an interface for the robot delivery attachments, which the mail delivery attachments and the food delivery attachments implement to support desired behaviours.
- Created an abstract class, DeliveryItems, which food and mail items inherit from.
- Assigned robot the task of inspecting items before loading, and the responsibility of choosing which delivery items to accept.
- Implemented a floor manager class to support food delivery contamination prevention.
- Implemented a robot statistics class to keep track of relevant statistics for system improvement.

These changes will be discussed in the following section along with the relevant patterns and principles that have guided these design choices.

2.2 Patterns and Principles

When designing our system we had a strong focus on extensibility, low coupling, high cohesion and information hiding. We achieved this through the use of many GRASP (General Responsibility Assignment Software Patterns) principles. The main principles that we utilised were the **Creator, Information Expert, Low Coupling, High Cohesion, Pure Fabrication** and **Polymorphism** design patterns.

Interface for robot delivery attachments

In order to deliver food, a new delivery apparatus, the food tube, was introduced into the system. This brought about increased complexity into the system as the robot could only have one apparatus attached at a time. The food tube also needed to be heated up before it was able to move out for delivery and would deliver in a last in first out order, as opposed to the first in first out order of the existing apparatus. In the old system, the apparatus was simply an attribute in the robot class which held mail items. Congruently, in our initial implementation of the new system, the food tube was simply another attribute in the robot class. As such, the robot had to undertake all the new functionality associated with heating up the tube and delivery order while maintaining the behaviour of the mail attachments, which significantly decreased the level of cohesion. This also affected the extensibility of the software for future needs as the robot class had to be greatly changed to accommodate these behaviours.

Due to these concerns, we created separate classes for each apparatus that implemented a delivery attachment interface. This allowed for the differing needs of each item type to be handled outside the robot class, greatly increasing the **cohesion** of the robot class. This also reduced the **coupling** of the robot class as it no longer stored information of the items being delivered and no longer implements specific behaviours for each type being delivered, as such, it is now only weakly related to the delivery items. This implementation also greatly improves the extensibility of the system as new item types that require new behaviours can be introduced by simply adding new apparatus classes to the system to support them with minimal changes to the robot class due to the use of **polymorphism**.

Applying the **information expert** pattern, we also assigned the responsibility of determining if the robot can be sent out for delivery to the apparatus as it has the information on the heating status as well as the information on what has been loaded.

The task of creating the delivery attachments was assigned to the robot class, which is in line with the **creator** pattern. The robot both closely uses and aggregates the attachments, hence should be in charge of instantiating them. We decided that each robot should have its own instances of both the apparatus as they would likely have a set in real life as well, which would also be useful for flexibility in the tracking of statistics.

Abstracted delivery items

When trying to adapt the DS system to deliver food, it became clear that a large portion of the information that the original mail item contained would also be contained by the food items to be delivered. Furthermore many things that interact with the mail items would also need to interact with the food items in a similar manner. For these reasons we decided that an abstract class delivery item would help us avoid repeated code and allow for a more extensible system. We initially implemented this by introducing a delivery item type attribute in a general delivery item class, but decided to make separate classes to improve the extensibility of the system.

This class became the **information expert** on all things that delivery items shared in common, this allows for the two classes that **polymorphically** inherit from delivery items (food items and mail items) to be very small and easily implemented.

This enables the client (DS Inc.) to easily extend their system in the future if they want to change the system to deliver even more types of items (e.g chemicals). The design choice promotes **low coupling** by removing the coupling from the robot to each type of item it can deliver. Instead the robot is only coupled to one abstract delivery item class.

The **cohesiveness** of the system is also improved as the robot can now deliver a wide range of items instead of a strict mail items only system.

Assigning responsibility for choosing which items to accept from the mail pool to the robots.

In the original system the Mail Pool had the responsibility of determining whether or not a robot should accept a given mail item in the mail pool. This logic was flawed as in reality the robot should be responsible for determining if it can accept an item. This also lowered the **cohesion** of the mailpool class. We thus decided to move that logic into the robot class by creating an inspect item method.

This also reduced the **coupling** of the system by reducing dependency between Mail Pool and Robot.

Furthermore if another type of delivery item was added to the system the mail pool class would not need to be modified, only the robot's inspect item method. This is a benefit of the robot being an **information expert** on the handling of delivery items.

The inspect item method accepts delivery items which can be either food items or mail items so once **polymorphism** is used and is easily extensible if further items to be delivered are added.

Implemented a Floor Manager class

When implementing the food delivery system we faced the challenge of avoiding contamination. This required us to ensure that when a robot was delivering food to a floor, no other robots were

allowed to make any deliveries to the same floor until its delivery was complete. To achieve this we created a Floor Manager class to keep track of locking and unlocking floors. This **purely fabricated** class became the **information expert** on contamination prevention. We decided to make the floor manager class be shared among all robots to ensure cooperation was possible.

Another solution we considered was making the floor manager methods all static or making the Floor Manager class a singleton. Both of these methods are flawed as they make the list of locked floors globally accessible and modifiable, meaning that classes that should not have access to the floor manager could unlock a floor and compromise the safety of the system. In this way we used the principle of **information hiding** to increase the safety of our delivery system.

Implemented a Robot Statistics Class

When implementing the robot statistics tracking functionality, it was determined that a new **purely fabricated** class was necessary in order to preserve the **cohesion** of other classes.

Furthermore by choosing to create the robot statistics class and pass it into the robots constructor we prevented the need for Robot statistics to have globally accessible methods. This could lead to incorrect statistics tracking and could negatively affect the company's decision making process.

By **fabricating** the class, we allow other classes which may need access to the statistics, the opportunity to get the information needed without exposing all of Robots public methods and attributes. This is a benefit of Robot Statistics being an **information expert** on the robots' statistical behaviour.

Robot performance analytics can be easily extended by adding new methods to the robot statistics class without ever having to modify the robot class.