Project 1: Automail – Design Analysis

SWEN30006 Software Modelling and Design

Benjamin Yi (1152795), Victor Lee (940683), Ying Wang (995836)

1 Introduction

The 2020 Semester 2 SWEN30006 project surrounds a recently developed Robotic Mail Delivery system by Delivering Solutions Inc (DS) called Automail, which entails an "automated mail sorting and delivery system designed to operate in a large building" with "dedicated mail rooms". The system consists of a `MailPool` subsystem which contains mail delivered to the building's mailroom, and `DeliveryRobots` which obtains mail items from the mailroom and subsequently delivers them to their destinations. A simulation subsystem is also provided which when run, emulates the processes and produces the "delivering log of robots and delivering statistics". Due to the COVID-19 social distancing restrictions, DS had sought to expand to include a new food delivery system.

The task for the project encompassed incorporating the new food system into the current design, utilizing DS's new heated food tube with the capability to hold up to three food items. Due to the abundance of restrictions and limitations affecting the robots when carrying food (such as robot hands needing to be detached when carrying the food tube, and a wait time of 5 units for heating), a new `FoodTube` class held by the existing `Robot` class was implemented, which worked around the new `MailTube` and `Hands` classes and with the `Robot` class to meet the requirements.

The report will explore the chosen solution design, including justifications for selected patterns and principles used, followed by further discussion involving alternative solutions and explanations as to why they were not adopted for the final solution.

2 Solution Design

2.1 Summary

To implement the desired capability of delivering food, additional interfaces and classes have been created, and alterations to the previous classes to work alongside the new ones have been made. To illustrate the difference between ordinary mail items and the new food items, the concrete `MailItem` class was converted into an abstract class as depicted in the design class diagram, and extended into the `FoodItem` and `RegularItem` classes, as both contain similar properties. This allows each type of mail to be placed accurately into their corresponding tubes (or hands), and interact with the tubes or hands as opposed to the `Robot` in the previously provided design.

To hold the separate food and mail items, tubes designated for each type is implemented, and hands remain to hold the mail. As both food and mail tubes along with the hands all behave similarly (hold items), a `Holdable` interface was created to declare a set of methods that capture the common actions as seen in the design class diagram with the `addItem(mailItem: MailItem): void` and `removeItem(): void`. Consequently, there are three concrete classes, `FoodTube`, `MailTube`, and `Hands` which implement the interface, each containing relevant attributes in regards to content held and limits (such as a maximum

of three food items). As the addition of food items to the system is applied, the restriction of only one robot per floor must be addressed.

A `Building` class is refactored with the aim of aiding the `Robot` class in reserving a floor for `Robots` delivery food items. As seen in the design class diagram, an unreserved floor in a building can have any number delivery robots, however only a food robot is able to reserve a floor to itself.

To avoid food contamination, a robot is able to reserve a floor if it's delivering food with the method `reserveFloor(floorNumber: Int, robotID: String): void` in the design class diagram. In contrast to the food/mail item and tubes/hands classes which are heavily relied on by the robot as items are added and removed, the floor class only interacts with the `Robot` class to reserve floors, as portrayed in the second system sequence design. The design currently chosen is based on decisions made through discussions on the patterns and principles to achieve the best balance of high cohesion and low coupling, whilst also easily allows extensions in the future.

2.2 Patterns and Principles

The behaviour of the new food item mail and food tubes can be described as small variations of the already existing mail items and mail tubes. To exploit this similarity, the principle of polymorphism is utilized to encapsulate the mail items into an abstract `MailItem` class, whereas robot hands and tubes were refactored to extend the `Holdable` interface.

In regards to mail items, the original `MailItem` class has been made abstract, and regular and food items are split into `RegularItem` and `FoodItem` classes respectively. This allows the mail pool to sort and assign incoming mail via the `MailItem` abstract class instead of duplicating code for each type. The mail generator also works primarily with the `MailItem` abstract class, only differentiating between the two at the end of the code logic. The advantages of using an abstract `MailItem` class includes future proofing further mail item types and ensuring the core behaviour of mail items remains consistent between mail item types.

As opposed to the original design where the robot hands and tube were part of the `Robot` class, the current design separates hands, mail tubes, and food tubes into individual classes `Hands`, `MailTube`, `FoodTube` respectively. Each class implements the `Holdable` interface, which defines the standard methods for holding items including picking up an item or counting the number of items currently being held. Individual classes then enforce their own behaviour according to the specification such as `FoodTube` holding up to three items as opposed to `MailTube` holding only one item at a time. By using the `Holdable` interface, code duplication is prevented, and it is easy to extend the `Robot` class to accept more tubes/hands of any type.

An advantage of using assigning the classes as stated above is the low coupling and high cohesion it brings to each class. By separating the `Robot` class from directly handling the `MailItem` classes, coupling is reduced between the two. Due to the `Robot` class previously responsible for handling the logic of picking up, storing, and delivering items, the addition of food items, mail items, and food tubes would have bloated the `Robot` class with more item handling logic, lowering cohesion. The introduction of `Holdable` classes which contain the

item handling logic decouples the `Robot` and `MailItem` classes as well as increasing the cohesion of the `Robot` class.

The principle of indirection is applied in the process of creating the `Holdable` classes that separates the Robot class and the mail classes. By having the `FoodTube`, `MailTube`, and `Hands` as intermediary classes that each have the responsibility of holding a specific type of mail, it assigns the responsibility of holding items to these classes rather than directly in the `Robot` class (from the original design). This avoids direct coupling between the `Robot` and `MailItem` classes, as information regarding mails will be stored within the tubes and hands as attributes, and obtained through getters. This design also encourages further expansions in the future if DS were to support delivering a variety of mail types, as more mail classes can be created easily without too many changes in the `Robot` class. The current model is the most comprehensive design as it utilizes indirection which supports both low coupling and high reuse potential, as well as all the benefits of previously discussed patterns, however there was also a range of alternative options which were not chosen due to limitations in the designs.

By information expert principles, the amount of times regular and food items are delivered, and the accumulated weight of mail and food items are tracked in the robot class by each robot instance because the robot making a delivery contains the necessary information about the items to update the statistics upon delivery. Furthermore, when the mail pool successfully assigns a food item to a robot, it can infer a food tube is attached. So the mail pool is assigned to track the total time food tube is attached. The reason each instance of robot is assigned to keep track of their delivery statistics is to allow DS to track additional statistical information to each robot performance in the future should the need arise.

3 Alternative Solutions

As an alternative for implementing the `Holdable` interface for `Hands`, `FoodTubes`, and `MailTubes`, extending the `FoodTube` and `MailTube` classes from the abstract `Tubes` class was considered, as well as the standalone `Hands` class. While this would still bring the low coupling and high cohesion advantages explained previously, this solution would not exploit polymorphism and its advantages to the fullest. The functionality of the hands is identical to the functionality of the tubes in terms of item handling, and differences only come into play for delivery order, which is not a responsibility of the `Holdable` classes. This means all the holdable classes can inherit from the same abstract class or implement the same interface with no loss in functionality. The reason an interface was chosen to group them instead of an abstract class was to account for potential future variations. For example, implementing `Heatable` and `Coolable` interfaces in the future for `FoodTube` variations would be easier with the interface solution, and may be difficult with the alternative abstract class solution.

Using a `Statistic` class was a considered alternative to track all the necessary statistics. Having a singular instance of a statistics class to update statistics upon every successive robot delivery can improve the overall cohesion of the robot class but would also increase the coupling of the `Robot` class and the `Mailpool` where the counter for the amount of time food tube is attached is tracked.

A `Floor` class could have been implemented to make the `Building` class a composition of floor objects. The floor instances would then hold information on whether it is locked or open and which robot reserved the floor. This implementation however would create a high level of coupling between the `Robot` class, `Building` class and `Floor` class. It results in a system

where the robot would have to access a `Building` instance to get the necessary destination floor data. Furthermore, unused floors would be instantiated to serve no purpose. For larger simulations this could result in significant processing overhead.

With the addition of the newly created classes along with the refactoring of original classes, the functionality of the `Automail` simulation has been extended to include food mail items and food tubes. The new classes `FoodItem` and `FoodTube`, alongside the refactored classes `MailItem`, `Robot` and others incorporate GRASP patterns and principles to create a cohesive, easily extensible system. This new design handles food items as instructed in the specification, and performs identically to the previous system when food items are not enabled. The final design combines multiple iterations and alternative designs to provide the best possible solution to Delivering Solutions Inc while adhering to the given specifications.