SWEN30006

Project 1: Report

Team Members
Andrew Shen 982054
Khant Thurein Hain 1028138
Kaif Ahsan 1068214

Introduction

In our project, we have implemented food delivering capability for the AutoMail delivery system extending upon its existing functionality. In order to achieve this, we first carefully analysed and created a domain and design model. Next, keeping object-oriented and GRASP principles in mind, we meticulously incorporated a food delivery system, first in the domain model and later in the design model. Furthermore, to evaluate our design, we implemented a system design diagram mapping out the sequential method calls, instance generation. Finally, based on our design, we refactored the relevant part of the existing code and developed the required extension.

Domain Model

Our domain model was designed to map all the business entities to demonstrate the concept on a high level. In AutoMail, there are two different types of delivery items, mail and food. Because both share common attributes and are logically similar, we made a common parent concept Item. Similarly, we created a common parent Tube for Food Tube and Mail Tube.

Because the Automail system can be deployed in different buildings, which will have different numbers of floors and different designated mailroom locations, we included it in the domain model.

To represent an instance of the Auto Mail system and the different configurations possible, we included it as a concept.

We discussed whether robots should be connected to the Item Pool. We decided that because both are already connected to Auto Mail, there is no need for a connection. Where the item is stored should not matter for Robot.

Design Model

Responsibility for storing Items held by a robot: Container
Since we wanted the item with higher priority to be delivered first, and the ItemPool distributes the items with higher priority first, we have decided to use a queue. If a robot is holding mail, it is equivalent to having a container with a queue of two items, with the first item of the top of the queue representing the item being carried in the robot's hands. Since queues comply FIFO principle, it ensures the item with the highest priority gets delivered first. Similarly, if a robot is using a food tube it is similar to using a queue of three items. We used the pure fabrication pattern and created a general parent class Container to represent that queue. This way we increase cohesion and decrease coupling in Robot. Mail and Food items are stored in MailContainers and FoodContainer classes so Robot can treat *MailContainers* and *FoodContainers* in the same way with polymorphism.

The alternative method was to denote a hand and tube through separate parameters. Since switching to a food container removes robots hands, everything we switched back and forth from food to mail container, we would need to mimic attaching and detaching hands. Since transferring items from tube to hand did not require any time, we concluded using a queue would be the most efficient approach to achieve the desired implementation.

Responsibility for preventing food contamination: Building
When discussing how to make sure that robots do not deliver food at the same time with other robots on that floor, we decided that we needed a class to keep track of when and where robots are delivering items. We decided that it should not be a part of information experts and controllers like Robot, ItemPool and Simulation in order to keep cohesion high. Furthermore, since all the parameters of the Building are static and public, it also means that they are easily accessible from other entities in the code and less complex implementation can be achieved.

We instead kept the information in the Building class. Furthermore, we designed an *enum* FloorLock which contains the different possible states a floor can be for a smoother transition. To minimize delivery, we implemented the floor locking mechanism in such a way that robots can still move towards the destination floor. However, they will only be prevented from making any deliveries only if another food robot is already present on the floor and making a delivery at that specific time step.

The alternative we explored was to delegate the floor locking mechanism to ItemPool. In the template code, items were loaded into the robots using a robot iterator. We initially explored a similar implementation where we would iterate through the robots in order to check if a food robot has reached its delivery destination and lock that particular floor in the next time step. However, this raised scalability issues as we were concerned if the number of robots increased dramatically, this would have an effect on the performance of the system. Hence, we opted out from this approach and agreed upon delegating the responsibility of building class.

Responsibility for loading items to robots: ItemPool

When assigning responsibility for loading *MailItems* and *FoodItems* to the robot we decided that we need a controller-like class. We considered *MailPool* and *Automail*. While we can keep cohesion high by changing *AutoMail* into the controller class because it does not have any existing functionalities, we chose to extend MailPool instead because it is the information expert since it knows what items are waiting to be delivered.
We also assigned the responsibility for choosing the correct container for each Item type to MailPool. While we can keep coupling lower by assigning this responsibility to Robot, we saw that Robot was already quite bloated and decided that it is better to keep cohesion high and keep MailPool as a controller.

Responsibility for keeping track of delivery priority: Item

We extracted the Item class nested in MailPool that was used to wrap MailItems that assigns a default priority if it does not have one. We wanted to keep the MailPool class simple and instead created a separate Item class inherited by MailItem and FoodItem that assigns itself a default priority. Furthermore, we refactored the code to remove PriorityMail class. Instead, we extended the Item class to contain an attribute to indicated the level of priority of an item.

Responsibility for statistics tracking: ReportRobotAction

We discussed whether statistics should be tracked in Auto Mail of Mail Pool. We decided to extract the ReportDelivery class from Simulation and rename to ReportRobotAction. Mail Pool has the list of robots and mail to be delivered and is the information expert. But we decided that Mail Pool should not have that responsibility in order to have high cohesion. Hence we identified ReportRobotAction as the second 'information expert' because it knows the delivery time and item type of every delivery.

We liked that ReportRobotAction followed the IMailDelivery interface because when the system is running in production as opposed to simulation, we may like to pass in a different implementation of IMailDelivery. This is an example of a protected variation when we need to add a messaging class to notify users that their delivery has arrived, we can create another implementation of IMailDelivery.

Because we also need to track the amount of time spent attaching and detaching containers, we replaced the IMailDelivery interface with the RobotActionable interface with an additional *chargeContainer* method that is called when a robot is attaching and preparing its tube. We considered not extending the IMailDelivery interface, but we decided to keep all the information in one place in RobotActionable for simplicity's sake.