SWEN30006 Software Modelling and Design
Project 1: Robomail Revision Report
Semester 1, 2019
Xiuge Chen and Danel Marshall

The aim of this project was to implement a revision of the Robomail system that supports team delivery. To achieve this, the development team made a number of extensions that fall into two broad categories: those allowing robots to work as a team and those changing the way mail is classified.

This report begins by outlining how the development team extended the existing system to accommodate the project's aims, with reference to adopted design patterns. The report then outlines the refactoring that occured to support these extensions before concluding with a discussion of the alternative design options considered by the development team.

1. Extension

1.1 Working as a team

Like robots working alone, robots in teams need to be loaded, to move and to deliver an item upon reaching their destination. However, robots acting in teams are loaded differently, move differently and have different behaviour during delivery. As such, while a robot in a team has similar behavioural objectives as a single robot, the specifics of their behaviour are different.

The development team decided to adopt the Strategy pattern to help accommodate the behaviour of robots working in teams while preserving the behaviour of the original system for light mail items. The strategy pattern is a good choice because it allows robots to vary their behaviour in different situations (i.e in teams) without implementing the specific team behaviours themselves (meaning *Robot* has higher cohesion than otherwise). Also, Strategy provides protected variation by creating a stable interface [1] for robot behaviour that insulates the existing system's functionality from the extensions required for robot team behaviour. This approach also exploits polymorphism. The motivation behind why the strategy pattern was preferred relative to other design alternatives is discussed in section 3.

The development team used one implementation of the Strategy pattern to allow the mail pool to load a mail item onto a single robot or robots working in a team (of two or three) depending on item weight. The development team used a second implementation of the strategy pattern to allow robots to support different movement behaviour (ie only move every 3rd time period while in a team) and different delivery behaviour (i.e. a robot team dissolves after delivery) depending on whether they are in a team.

As outlined in the static design model, implementing the loading strategy used by the mail pool involved creating a loading strategy interface (*ILoadStrategy)* and two concrete strategy implementations (*LoadRobotStrategy* and *LoadRobotTeamStrategy).* Similarly, implementing the robot behaviour strategy used by the robot class involved creating a *IRobotBehaviourStrategy* interface, and

two concrete strategies (*TeamRobotBehaviourStrategy* and *SingleRobotBehaviourStrategy).* Additionally, the development team created a Singleton Factory for each strategy (*LoadStrategyFactory* and *RobotBehaviourStrategyFactory*) responsible for instantiating the correct strategy for a given situation.

To understand how the improved system allows robots to deliver heavy mail items as a team, broadly consider how a robot changes between working singularly and working as a team. First, the mail pool loads robots for dispatch. This is achieved by calling the method *loadItem* (see frame *Robot Loading* of the dynamic design model). The mail pool's *loadItem* instantiates the relevant concrete loading strategy using the *LoadStrategyFactory* and loads the mail item accordingly (see *Load Single Robot Case* and *Load Team Robot Case* frames of the dynamic design model). At this point, any robot involved with the loading of a heavy item transitions from having a team state 'SINGLE' (acting as a solo robot) to having a team state either 'TEAM_LEADER' or 'TEAM_MEMBER' (i.e. is acting in a team).

While the robot is out on a delivery it uses the *deliveringStep* method to control movement and the delivery action (see frame *Robot Delivering* of the design sequence diagram). To achieve this, *deliveryStep* uses the *RobotBehaviourStrategyFactory* to instantiate the correct concrete instance of robot behaviour. Then the robot(s) proceed alone or as a team to either deliver the packet or move towards a destination depending on whether the destination has been reached. Once the destination is reached, the single robot or team leader delivers the mail item. After delivery, any robot that has state of 'TEAM_LEADER' or 'TEAM_MEMBER' (i.e. in a team) reverts to state 'SINGLE' (i.e. acting alone). In this way, robot teams always dissolve after they have completed delivery of a heavy item.

1.2 Mail classification

Classifying mail so that the relevant concrete strategies are implemented is critical to the operation of the updated system. In the original system, the robot class took responsibility for classifying mail as acceptable to carry or too heavy based on weight. The development team felt that a more cohesive design would be to create a Singleton *MailClassifier* class with responsibility for classifying mail. The Singleton pattern was used to provide a global and single point of access (and because only one *MailClassifier* is needed).

2. Refactoring

The development team made a number of refactors so that the original system could support the system extensions. Some of significant refactorings are discussed in this section.

2.1 Move attributes and functions

As mentioned in section 1.2, the team created the *MailClassifier* class to classify mail items in terms of the number of robots needed to deliver them. As a result, *MailClassifier* needs to know the max weight that robots in different configurations can carry. In the original implementation, all weight limitations were stored in the *Robot* class (and used for exception throwing when adding mail items to their tubes and backpacks). In the updated system INDIVIDUAL_MAX_WEIGHT,

PAIR_MAX_WEIGHT and TRIPLE_MAX_WEIGHT are transferred to *MailClassifier*. As it does not make sense for each individual robot to know robot team weight limitations, this implementation improves cohesion of the *Robot* class.

Furthermore, in the updated system, the responsibility for checking mail items are not too heavy for a robot configuration (i.e *ItemTooHeavyException* throwing) is transferred from the *Robot* class to the loading strategies described in section 1.1. In the development team's view, this improves *Robot* class cohesion by assigning responsibility for checking item weight limitations in the robot loading classes rather than within the robot class.

2.2 Resolve long methods

While making extensions, the team found several long methods, such as *step* in the *Robot* class. Such long methods increase the difficulty of understanding and extending. Thus, the team extracted these functions and divided them into smaller private sub-functions, to support our extension and improve maintenance.

2.3 Extract functions into interface

As discussed previously, the strategy pattern was used to support robots moving and delivering differently based on whether they are in the team or not, and to support the mail pool loading items differently depending on their weight. This meant that the team extracted related functions and incorporated them into an interface.

2.4 Add getter/setter

The team used the *IRobotBehaviourStrategy* and *ILoadStrategy* interfaces to implement variation in robot loading and behaviour. The concrete strategies implementing these interfaces required access to some attributes of *Robot* class. So the team added getter and setter methods for the relevant attributes of *Robot* essential to the implementation of each strategy. To achieve improved encapsulation, the development team restricted some of the behaviour of setters (i.e each robot could only move upstairs or downstairs one floor each step).

3. Design alternatives

Several alternative designs were considered for both the 'working as a team' and 'mail classification' extensions discussed in section 1.

3.1 Working as a team

When considering alternatives for the team work behaviour, the development team considered using the a*daptor pattern* or pure *polymorphism.*

The development team envisaged using the adaptor pattern to provide a stable interface to interact with different/new classes of robot delivery services (such as a new classes for team robots and the existing robot class). However, the development team favoured the strategy approach as there are

only subtle behavioural differences between single robots and robot teams, meaning a seperate class for each is not warranted. Furthermore, the development team felt the adaptor pattern would require more refactoring of the original code which is a risk given the project's objective to retain existing behaviour.

Simple *polymorphism* involving subtype substitution could be used to generate the team behaviour. Specifically, it might be possible to have a TeamRobot subclass of *Robot* that overrides movement and delivery behaviour. The development team rejected this approach on the basis that it would introduce complex object life cycle management conditions (ie when to downcast a Robot to RobotTeam or upcast a TeamRobot to Robot) which may making the solution less maintainable and modifiable.

Furthermore, the team discounted this polymorphic approach because we believed it would increase the representational gap. This is because in the problem domain there is only one type of robot which may or may not coordinate with other robots at different points in time (ie a robot team is not a special/different type of robot). The strategy approach reflected this more accurately.

3.2 Mail classification

The development team considered alternatives to solving the mail classification problem. The most reasonable alternative involved using polymorphism to create heavy and light mail items that extend the mail item class. This would allow robots to consider the subtype of the mail item when determining whether to form a team or act individually.

However, the development team opted against this approach for a number of reasons. First, it would potentially introduce many more classes (ie heavy, medium and light mail items each for normal and priority items) which means determining subclass for each mail item would likely involve lots of conditions (reducing modifiability/maintainability). Second, the preferred approach better accommodates modified mail classifications (i.e through changed parameters rather than new classes). Third, using an existing class to classify mail according to class subtype is a less cohesive solution than using the dedicated *MailClassifier* class.

References

[1] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* 3rd Edition, Prentice Hall, 2004 [Ebook] Available: Safari ebook