Robomail Revision Report

Project Team 88
Aman Bhuyan (946264)
Josh Sanon (936965)
Lakshya Mittal (1009364)

Introduction

In our refactoring of the AutoMail simulation system for RMS, we were tasked to implement a “caution mode” for their delivery robots. This caution mode was to enable the delivery of fragile mail items by the robots, using pre-installed “special arms” that can wrap and unwrap the packages as necessary. The original behaviour for delivering normal mail items was meant to be preserved in this simulation.

In addition to the above, we were also required to include statistics tracking (number of packages delivered normally/in caution mode, weight etc.), which was to be shown at the end of the simulation.

The simulation software provided was capable of generating normal mail items for the “robots” to deliver as well as fragile mail items, and thus we were not required to implement that part of the system.

We have decided to show only the changed components from the original program in the Static Design Model, so anything that was not modified as part of this implementation has been omitted. Therefore, in this report, we will discuss only the additions/modifications to the existing program with chronological reference to the Dynamic Sequence Diagram that we have also provided.

Implementation & Justification

An important change was made to the states a robot can be in with the addition of the WRAPPING and the UNWRAPPING state. This allows us to clearly show the process in the sequence diagram while making the delivery process more readable in the console of the simulation. In order to make understanding the diagram easier, we also avoided showing steps for delivering normal items as that is already known, and just need to know how delivering fragile items works.

The mail pool assigns robots with packages and sets them off for delivery. This class required changes to how the mail pool implements the loadRobot() function. We have decided to utilize a loop for this loading to avoid the repetition of code. The items are first checked for fragility, and if the robot has the storage space available to hold the item it is then loaded in the respective slot. The system is implemented in such a way that if there is an attempt to load any type of item into the robot but there is no storage space available, the robot is then dispatched for delivery. The order of loading the non-fragile items into the hand first and then the tube has been preserved. By changing how the mail pool works instead of the Robot, we avoided introducing unnecessary coupling.

We did observe that the original program had introduced an Item class, which is essentially the MailItem class with the destination as an attribute of the class. It was decided that we will not be changing this in order to keep original behaviour intact, as it seemed that it was meant to be applying the Indirection pattern - which is in favour of supporting low coupling and keeping reuse potential high.

The class with the most significant and highest number of changes was the Robot class. Most of its changes were in its step() function, along with the introduction of other functions to help in this process. For instance, with the addition of setInitRoute() we are preferring the fragile item to be delivered before a normal item by the same robot if their destinations are the same floor. This was done in the interest of improving the scores for the overall simulation. moveTowards() was also changed to immediately make a floor unavailable to other robots once the robot making a fragile delivery reached its destination floor.

We considered abstracting the fragile arm/caution mode functionality to a different class called CautionRobot that would inherit from Robot. However, in doing so we realised that a lot of the code for the step() function for CautionRobot would be exactly the same as that of Robot, only with a few extra additions. Thus, it’s not a simple case of being able to override a function as such - the code for normal delivery and fragile delivery work hand-in-hand, so calling the original function from the overridden function would not work either. Therefore, even though the sequence diagram we have provided only shows the steps for delivering a fragile item, it cannot work by itself and requires working in a combined effort - it is only for explaining purposes. This approach also reduces coupling and preserves cohesion.

The responsibility of storing information about the floors like the availability and number of robots per floor was delegated to the Building class. This is logical, and although it increases coupling, we know that it is not a problem since the coupling with a stable element is not a problem - this is a case where the coupling is essential. There was the option to create a different class Floor and have instances of that in an

ArrayList within Building, but we wanted to stay as close to the original structure of the program as possible.

The code for the tracking of statistics of deliveries was kept in Simulation, as it is the most logical place, and also because the tracking refreshes on every step() call. This means that if we were to put it in a different class, there would be higher coupling overall, as Statistics would be linked to Robot - so we avoided that. The statistics were collected in a reasonably simple manner while remaining accurate. For instance, we knew that in our new simulation program, a fragile mail item would always be wrapped (2 units of time) and unwrapped (1 unit of time) regardless of destination. Thus, every time a fragile item was delivered we added 3 to the “time spent wrapping and unwrapping” counter - preserving the level of coupling.

Conclusion

The final version of our implementation as per the requirements preserves all original behaviour while performing all the new tasks. It was achieved by simple modifications of existing classes, and introducing new functions - no new classes were created. The simulation program has the right amount of coupling between classes and cohesion within classes to perform the task well and to allow for easy future refactoring.