Software Modelling and Design

Project 1 Report

Team W9-5

Refactoring

Responsibility Reassignment

A problem in the original design is that Simulation does a lot of things that it's not supposed to do, for example updating MailPool and Robot every iteration (by calling step method of each object). This leads to a design that has high coupling and low cohesion (Simulation has unnecessary interactions with other classes and is not focused on its own tasks).

To make it a better design, we apply GRASP pattern. We move the responsibility of updating MailPool and Robot objects for each iteration from Simulation to AutoMail based on information expert principle. AutoMail knows everything about the system (MailPool and Robot), so it should be responsible for updating the system. Simulation then only interacts with AutoMail when it wants to do something and need not know the detail (which method to call for which action), resulting in a low coupling and high cohesion design. Besides, AutoMail can be considered as a controller, controlling how Simulation, acting as a user, communicates with the internal operation of the system. Also, in the original design, AutoMail is a lazy class, which does nothing but initializing robots. Now, it is responsible for the operation of the system, which is what it should do.

In the design class diagram, there is no association between Simulation and MailPool. AutoMail has associations with both MailPool and IRobot, controlling the whole system.

In the design sequence diagram, AutoMail is responsible for calling step() for MailPool and Robot, instead of Simulation.

Extending

Adding Team Functionality

During our attempt to add team behavior into the system, we found that a single robot and a team of robots are quite similar: the way they load mailitem, move towards destination, update status for each time step. Therefore, we decided to apply composite pattern in the design to help implement the new team functionality.

IRobot is the common interface implemented by Robot and RobotTeam. When there is new heavy mailitem to be delivered, extract enough robots from the list, form a team. When there is a task to perform, AutoMail can just simply ask team to do it, and the task will be forwarded to all the members by team.

One of the advantages of applying composite pattern in this project is extensibility. When there are new types of robots in the future, they can be easily added into the system by creating new classes that implement the common IRobot interface, and a team can consist of different robots without any modifications made to RobotTeam class. In addition, the system achieves polymorphism with the help of composite pattern. When there are different types of robots, a bunch of conditionals can be saved by giving all robots IRobot type. Moreover, as there are team objects, it is also easy to add other team functionalities, such as in-team communication.

However, composite pattern introduces extra coupling into the system. In the original design, AutoMail manages Robot and MailPool. MailPool need not know anything except for robots that are available for delivering mailitem. The interaction between AutoMail and MailPool is one-way: from AutoMail to MailPool. In our current design, robots are divided into two groups: those working alone and those working in the team. As MailPool is the only one that knows when to create a new team, it has to talk to AutoMail, updating its information about the operation of the system. Now, the interaction is two-way. To allow MailPool to be able to communicate with AutoMail, we came up with two solutions: one is giving MailPool an AutoMail object, the other one is making AutoMail a singleton. We chose the former one as our final decision. We did not choose the latter one because of privacy issue. We consider AutoMail to be the central controller of the system, hence should have good access control. Making it a singleton is obviously not helping us achieve that.

In the design class diagram, MailPool has an instance of AutoMail, which is used to inform AutoMail about the changes made to the system (new team created to deliver heavy mailitem) so that AutoMail can update its status. Both Robot and RobotTeam implement IRobot interface, and RobotTeam, being the composite class, has a list of IRobot objects. This shows the use of composite pattern.

In the design sequence diagram, when MailPool detects heavy mailitem to deliver, it creates a new instance of RobotTeam, have the team loaded and dispatched, inform AutoMail about the changes. In the next iteration, AutoMail will step team as a whole, instead of doing it for each individual robot.

Other options

Another way of implementing team functionality is by changing the logic of loadRobot method in MailPool class. When the next mailitem to be delivered is heavy, extract enough robots from robot list, assign this mailitem to them and have them dispatched. No new classes needed. There is no explicit formation of teams. Each robot knows that it is carrying heavy item (by using an attribute as a flag) but knows nothing about its teammates. This is our first design. It is simple but works well. Coupling is low in this design. AutoMail controls the operation of the whole system. MailPool and Robot just do their jobs and need not worry about anything else. The only problem with it is that it is not extendable, hard to maintain, and hard to modify for future changes.