Software Modelling design report

For the existing model. In order to make it a new function that can bring robots together and start carrying a heavy item, and in the existing model, it’s the job of the mailpool that assign the mail item to the robots’ hands and tube, then the robots can start carrying the mail item and deliver them. So, we come up an idea that to do something d­ifferent with the mailpool so it will assign the heavy item to a group of robots.

We have noticed that the mailpool might be doing too many jobs as it not only added mail to itself, and it also do the job of distribute the mail to the robots. So we decide to separate the distribution job from the mail pool, and assign it to a distribution system, to make it has a higher cohesion. The mailpool now has a more focus job and it’s easy to change if one day that people decide that using the same mail pool but distribute the mail in a different way. They only need to change the distributing system, instead of the whole mailpool. So we apply a strategy pattern, create a new interface called IdistributeSystem, with a single function distribute(). If someone decides to change the way of distribution, they just need to implement the IdistributeSystem then override the function.

And knowing that there are already existing 2 different distributeSystems, one is the simple which is provided by the existing model and the other one is the distribution according to weight. In the thought of the future, there might be more way to distribute and people might want to change the distribute way in runtime (i.e. according to the mail items in the mail pool, change the way of distribution so make it more efficient), we create a factory method to create the distribute system, with the thought of the factory is always only needed one instance in the environment, we used a singleton pattern on the distributeSystemFactory class.

In the WeightDistributeSystem.java, when we were writing the function to load the item to the robot, we created a helper class with all the static method to help loading. Therefore, if in the future, if people want to make a more complicated distributeSystem and it including weight, they can use the method straight away from the helper class without creating any extra objects.

After taking care of the distribution of the heavy mail item, we need to notify the robots that they are now on a team as according to the requirement, the robots that moving in team, will be 1/3 times in the movement speed. First we notice that, the robot start moving a function in the Robot class called dispatch(), which is weird that as the robot is the one get dispatched, instead of dispatch someone. And since now we have a distributeSystem, it makes more sense that the system do the dispatch job as it might be a different way to dispatch the robot. In this way, we make a higher cohesion for both the robot and the distributeSystem as they now are more focusing on their own job.

As in the movement speed for the team, the easiest way is just added a variable to the robot class to tell if the robot in a team and then control it’s movement by that. However, this will introduce more variable and a lot of messy function in the class and not extendable. So, according the indirection principle, using another strategy pattern, create a new interface called IBehavior, and the distributeSystem dispatch the robot to control the behavior of the robot, and the moveforward() function of the robot is now moved to the interface as they in different behavior, the movement will be different. There are only 2 different behavior in the scenario: TeamBehaviour and SoloBehaviour. But people can added more in the future.

During the process of using indirection pattern with the Interface Behaviour. We need some information about the robot, in order to control its movement. So, we create an Information expert for the Robot, called Info, to record some information of the robot and then share to the Behaviour to use it.

After taking care of both distribution and the robot behavior. We need to take care of the Delivery, according to the original model, if we have a team of three robots to deliver the mail, it will produce the exception that said the mail is already deliver. So, using the polymorphism pattern, we create a TeamReportDelivery with extends from the basis ReportDelivery and add a counter of mail for the heavy mail, so It won’t be showed as excess delivery again.