Software Modelling and Design: Project 1 Report

Design Class Diagram

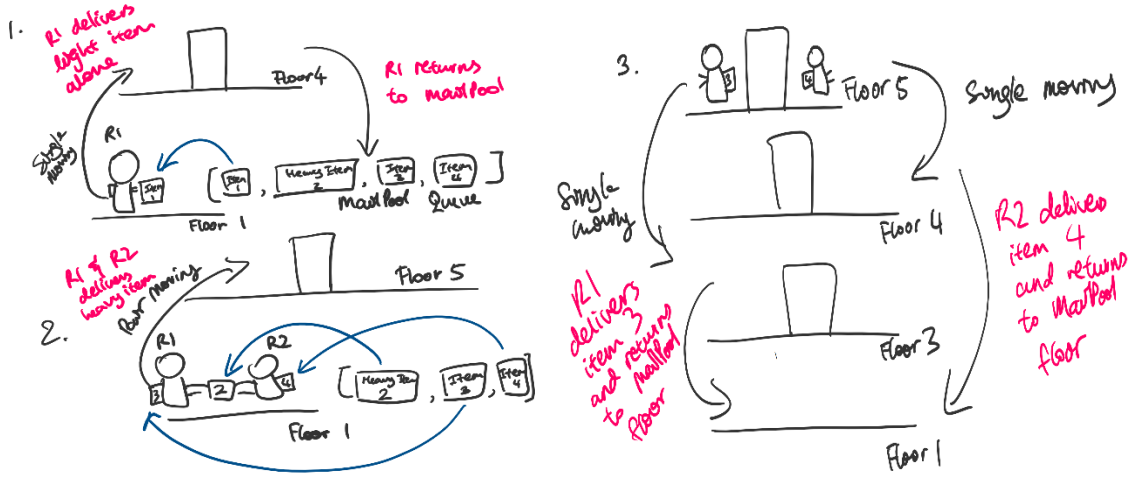
For the Design Class Diagram, we included only the classes that had its attributes changed after implementation, along with all the added classes. We constructed the design class diagram from the domain model and attempted to minimalize the representational gap between them. As evident, there are only minor changes between the two diagrams such as the domain model having a Floor entity which is not present in the design class diagram, but ultimately they represent the same problem scenario in a stakeholder's and software design's perspective respectively.

Design Sequence Diagram

In the design sequence diagram, we modelled the diagram based on multiple use cases by including if-else frames to showcase what happens in the different use cases. For example, we have an alt frame to show how our implementation is able to accept the item weight during run-time and choose the appropriate Team Strategy to handle the carrying of heavy items. (This will be further elaborated in the Code Implementation segment)

During the modelling of the design sequence diagram, we tried to make it complete (i.e showing a variety of use cases), but at the same time, not too detailed. The reason is because we want our diagram to be able to capture how robots in our system transition from operating individually to working as a team, and back to working individuals again, as per the project specifications.

For instance, one can consider the use case which describes a robot that is delivering a light item (<2000 weight) on its own, and upon successful delivery and returning, is assigned to a heavy item (PAIR weight) with another robot on standby. Aside from the heavy item, the robot is also assigned 2 light items on the mail pool queue to its tube. After delivering the heavy item successfully as a pair, the robot then works on its own to deliver the remaining item on their tubes. Our sequence diagram is able to show this use case as one can simply follow the sequence of events for this use case from start to finish in our diagram. Here is an illustration we created to showcase this use case:



Code Implementation

Summary of changes to the code:

For the code implementation, we added a SingleLoadStrategy and a TeamLoadStrategy Class, which both implements the ILoadStrategy interface. The SingleLoadStrategy and a TeamLoadStrategy Class handles how the items are loaded into a single robot and multiple robots respectively, depending on the item's weight. We also added the Loader Class which basically loads the items to the robot using the appropriate strategy, and sets their initial speed. Additionally, we added the SingleMoving and TeamMoving Class which is responsible for moving the robots around the building and ensure they are moving at the appropriate speed during every timestep (floors per step).

Useful Design Patterns:

The reason behind this implementation is because we found that the Strategy Design Pattern is suitable in solving our problem and also offers an extensible design, therefore the classes we added made use of this pattern. To elaborate, the Loader Class has a chooseStrategy() method which chooses the appropriate strategy based on the item's weight. If the weight is less than the individual max weight, than it uses the single load strategy, else, it uses the team load strategy. This allows the system to select the algorithm at run time, which is the main purpose of the Strategy Design Pattern.

We also found that this offers extensible design because if the customer using this system wants to add, for instance, a new way to load fragile items into the robot, they could simply create a new FragileLoadStrategy class to deal with fragile items, and a conditional if statement to tell the system to use this strategy when a fragile item needs to be loaded. Similarly, the SingleMoving and TeamMoving class utilises the Strategy Pattern for this reason as well.

The Strategy classes also makes use of a type of Polymorphism—in the case a TripleLoadStrategy object is created, calling a method on that object will invoke the appropriate method based on its type— which further simplifies the system, as we no longer need to use an if statement to check the object's type explicitly.

Another design pattern we found to be useful is the Indirection Design Pattern, which is implemented by the Loader class. As previously mentioned, the Loader class is solely responsible for loading items into the robots, while the SingleMoving and TeamMoving classes are responsible for moving the robots. In other words, the Loader class acts as a mediator between MailPool and the Moving classes, as the responsibility of loading the items is delegated to the Loader class. This promotes low coupling between MailPool and the Moving classes, as well as high cohesion, as Loader is focused only on loading items and the Moving classes only on the movement of the robots.

Preservation of the System

We also made sure the original behaviours of the system were preserved, such as following the default rule that the first item in the mail pool queue should be added into any available robot and ensuring that the first item is dispatched first. This is true even with the presence of heavier items, where in the case that only a single robot is currently in WAITING state in the mail pool floor, and the next item is a heavy item, that robot will simply wait for another robot to show up to help carry

that heavy item, even if the nest subsequent items are light items and could be delivered during that time. Even though it might be less efficient for that robot to just wait around for its partner to show up, we had to follow the original behaviours which prioritizes dispatching the priority items first (which is already sorted in that manner in the mail pool queue). Therefore, we did not try to change this aspect of the delivery system.

Extendibility and Other Considered Options

With regards to extendibility, we also made sure that the weight class of the items could be easily extended. We kept the original weight class constant variables so that a user can easily define how much weight a robot or a group of robots can carry, and if a user wants to add another weight class, for example quadruple max weight, they just need to add a new constant variable in the Robot class and a new if statement in the TeamLoadStrategy class for the creation of a QuadrupleLoadStrategy.

Furthermore, to allow more extendibility, we designed the single, pair and triple load strategy classes to inherit from the TeamLoadStrategy parent class. Initially, we only had one TeamLoadStrategy class without subclasses. However, we realised that since we allowed the client to add items with new weight classes, it also makes sense for us to allow them to change the robots' speed while carrying that new item type (e.g four robots carrying the item might move at ¼ of the original speed), as that is a possible and likely scenario. Since the load strategy classes are responsible for setting the robots' initial speed, we decided to use inheritance to extend this feature.

Prior to implementing the load strategy and moving classes, we considered using a single class that handles both loading and the delivery of the items, called DeliveryStrategy. Unfortunately, we realised that such a class would introduce high coupling with the MailPool and low cohesion. Not to mention, extending such a class would be more complicated—in the case that a client wants to add a different way of loading an item, but the delivery method is unchanged (or vice versa)—that would cause repeated code and a less understandable function of the class itself, which is poor design. Therefore, we decided to break down this class and delegate the two main tasks to separate classes.

Finally, an aspect of our system that might seem long is the Robot class, and one might wonder why we left the step() method in it, instead of creating another class that controls what the robot does in each step. This is because, as we have decided, that the Robot class already contains sufficient information to decide what to do in each step, and splitting that class would cause high coupling since the new class must have reference back to the robot to determine its state and delivery item. Hence, we just left the class as is otherwise it would make the system unnecessarily complicated.