SWEN30006 Part B

Design Analysis Report

Group 44

Issue among `MailGenerator`, `IMailPool`, `Simulation`

In the original design, `MailGenerator` class is designed for creating all of the `MailItem` objects as well as adding them into `mailPool`. However, it will also return the `PriorityMailItem` (if there is one) and let the `Simulation` class do the notification work. This not only violates the principle of Information Expert but also makes a high coupling relationship between those three classes.

Instead, `MailGenerator` should be considered as a class for generating the objects, which contains necessary information for generating `MailItem` only. To solve this issue, `MailGenerator` is used purely for creating the `MailItem` objects, meanwhile, as a factory which can avoid complexity. For each time step, `Simulation` class will retrieve `MailItem` objects from `MailGenerator` and put the these into the `MailPool` object, and notify the related classes when there is a `PriorityMailItem`.

Without the relationship between `MailGenerator` and `IMailPool`, the coupling is reduced at the same time, by assigning the responsibility appropriately.

Issue among `Automail`, `Robot`, `IMailPool`

Originally, Automail class creates `Robot` and `IMailPool` objects, and passing the `IMailPool` object into `Robot` for handling the case when a robot's state is `RETURNING` or `WAITING`. The relationship among three classes is high coupling which can be considered as a design problem.

In order to reduce coupling, the attribute of `MailPool` object in `Robot` class is removed. The `Robot` object still determines its own state of `RETURNING`, `WAITING` and `DELIVERING` based on its position, the situation of its tube, etc. However, the job of refilling the robot is assigned to the `Automail` class. The `Automail` class will handle the work of empty and refill storage tube when a robot is at the mailroom. As for the robot, all it needs to do is change its state to `WAITING` when it arrives the mail room and keeps waiting until there is something in its storage tube.

Moreover, `Automail` is designed as one façade controller for whole mail delivery system, including the `MailPool` and two `Robot` objects. `Automail` class has the responsibility to create both kinds of objects and interact with the system. Additionally, `step` method in `Automail` class is added for calling from `Simulation`, in order to control the robot to call step method after filling the storage tube.

Issue among `Robot`, `StorageTube`, `MailItem`

From the origin design, `Robot` class contains an attribute called `deliveryItem`, which is the `MailItem` object that robot need to deliver next. Since all of the `MailItem` objects are stored into `StorageTube` object in `Robot` class, it is unnecessary for `Robot` object to store an extra `MailItem` object for only getting the destination floor information, which also causes high coupling.

To decrease the coupling and satisfy the principle of information expert, the `deliveryItem` attribute is removed from `Robot` class. Whenever `Robot` object needs to know the next destination floor, it will gain the information from `StorageTube` object, by an additional method `getNextDestFloor`, which will peek the first MailItem object in `StorageTube` class and return its destination floor.

Issue in `Simulation`

Initially, there is an attribute called `MAIL_DELIVERED` in `Simulation` class to record all the delivered `MailItem` object, which is used for two purposes: one, be able to recognize the problem of a `MailItem` been delivered twicey; and two, counting the total number of delivered `MailItem` to check if the simulation is finished.
Checking if delivered can be considered as information expert problem. Since the information of checking whether the `MailItem` object is delivered can be handled by `MailItem` class itself, a `boolean` type attribute `isDelivered` is created, marked as `true` when the object is delivered.
Also, for counting the delivered `MailItem` object, a private attribute is added into `Simulation` class with initial value of zero, and increased by one for each time a new `MailItem` arrived in deliver method in `ReportDelivery` class. Without the relationship between `Simulation` and `MailItem` class, low coupling is implemented.

Misplaced constants

Constants are placed into inappropriate classes, which is a basic responsibility issue in the original design, related to information expert as well. Two constants are listed below.
`LAST_DELIVERY_TIME` in the `Clock` class can be considered as misplaced, since the static constant `LAST_DELIVERY_TIME` is unrelated to the `Clock` class which is only a representation for clocking. Therefore, it is replaced into `MailGenerator` class, since it is the required information for creating `MailItem` object. After implementing the `PropertyManager`, which is an additional class for reading properties from file only once, `LAST_DELIVERY_TIME` is stored in this class as default property. Therefore, final implementation would be `MailGenerator` class can get the information of last delivery time from `PropertyManager`.
Meanwhile, `MAX_WEIGHT` is misplaced in `WeakStrongMailPool` class, as it is the constraint for weak type of robot only. In this case, `MAX_WEIGHT` should not be existed in any `MailPool` class, instead, `Robot` class would be more reasonable and appropriate. By distinguish the type of `Robot` class by enumeration `RobotType`, the static constant `WEAK_MAX_WEIGHT` will be used.

`PropertyManager` - Pure fabrication

In the extended implementation, `automail.properties` will be read in the `Simulation` class. A class named `PropertyManager` is designed for handling all of the properties uniformly. `PropertyManager` class contains all the default properties, and override them if otherwise defined in the property file. This satisfies the principle of pure fabrication, like for the pure purpose. In term of design, singleton is more suitable than static class in this case, as the `PropertyManager` need to maintain a `java.util.Properties` object to store loaded properties, it follows the OOP patterns better in term of maintaining a single resource with global access. Whereas static class is more suitable for gathering a bunch of methods in a similar topic without a need of maintaining information, like `java.util.Math`.