**Robot:**

Inheritance is used to extend different robot types. Robots will have different Maximum capacities, Maximum carry weight, and different move behaviors. Those different behaviors are achieved by subclass overriding (polymorphism). Furthermore, robots have different flags to determine what kinds of mail item it could carry, these flags do not have one to one relationships with the robot types, for example, Strong (flag) could be both Careful Robot and Standard Robot. They share the Strong property and this property is used further in the give-out process of mail items.

For future additional robot types, it can be extended by constructing a new subclass of the Robot class by overriding new behaviors and classifying new flags (used in fillStorageTube()).

**Property support:**

A list of String (robot types) is read from the "automail. property" document in Simulation class because the list of robots is passed to the Automail class. Inside Automail class, guards are used to identifying different robot types and then create the corresponding robot objects.

For future additional robot types, we will need to add new guards in the Automail class.

**Coupling and Cohesion:**

Between the trade-off on System coupling and cohesion. Our design strategy prefers the functional cohesion of classes. The AutoMail class only handle the creation of robots. The MymailPool class only handles the give-out of the mail items. Because each time loading mails requires the MymailPool to access the robot objects in the AutoMail class, it will result in a high coupling. Therefore, in our design pure fabrication is used to lower their coupling between AutoMail and MyMailPool. The Robot class is the intermediate class that mediates the coupling. Both the AutoMail class and MyMailPool classes have an array of robots. In the Simulation class, a while loop is used until all the mail items are delivered. Inside of the loop, firstly, the robot is iterating through its different behavior states by calling autoMail.getRobot[i].step() in AutoMail class. Then whenever the robot is changing to Waiting state (one of its behavior states), the robot will be registered to the array in MyMailPool class. Therefore, when the MyMailPool start to fill the storage tubes (myMailPool.step()), it does not have to look for the robots with Waiting states inside of the AutoMail class because all the fitted robots are already registered. As a result, the coupling is lowered. Since inheritance is used in Robot types, it is the pure fabrication pattern.

**Limitation**

Regarding the fragile property of the Item class in the use case. In the system design, an extra pool is added to store only the fragile items in MyMailPool class. Furthermore, in the fullStorageTube() method, new guards are added when the items are given out.

The limitation lies in the future additional item property. The complexity of the change would require to rewrite the MyMailPool class. Because with the new coming property, a new algorithm will need to be considered, new mail pools will need to be added (probably here a new class of Pool is required in scaling), new guards in fillStorageTube(). It is basically rewriting the whole class of myMailPool.

The abstract class of MyMailPoolClass is once considered. However, for example, when Item has A, B property, a subclass of MyMailPool1 is created. When Item has B, C property, MyMailPool2 is created. But when Item has three property A, B, C, it is not feasible to reuse or combine MyMailPool1 and MyMailPool2 in order to achieve MyMailPool3 because items could have A, C property that is never considered in both MyMailPool1 and 2. And both the creation of these three subclasses are rewriting almost everything, the abstraction loses its meanings.