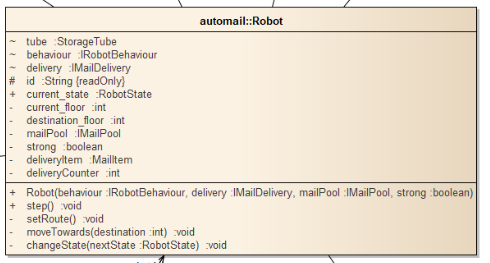
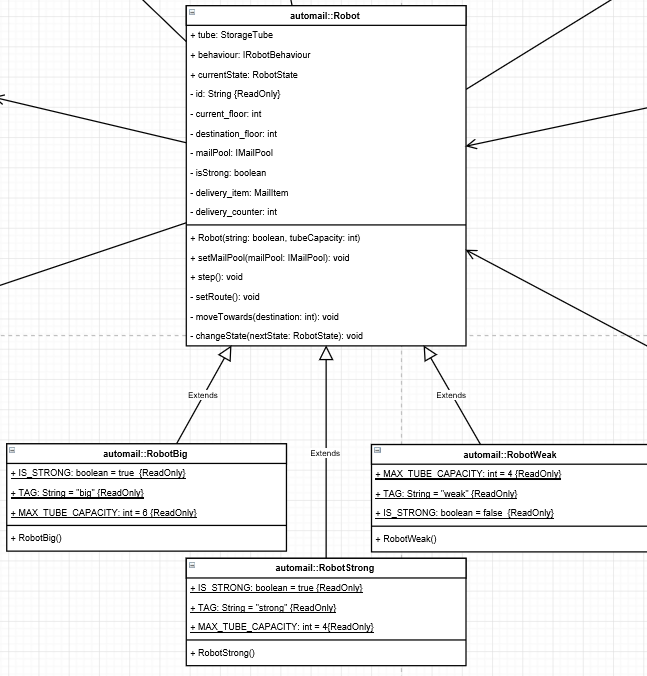
**Design Criticisms:**

**1**. In the existing solution, Robot differentiates between different types of robots with the strong attribute. While it works reasonably well for a system with two robot types, it makes extension and modification of types of robots difficult. We proposed using polymorphism, creating the classes RobotBig, RobotWeak and RobotStrong which inherit robot. This reduces complexity and reuse of code, allowing simpler comparisons through comparing class types, which scales much better with additional types of robots. Also leads to better delegation, where subclasses are in charge of their own creation, containing information such as the strength of the robot and tube size for robot creation. Furthermore, this alternative allows for the addition of another robot type to be simpler and would follow the open-closed principle. For example, if we were to add a new type of robot, a weak robot that can carry 6 items, the suggested system would be closed to modifications, requiring no changes to robot, and open to extension, creating a new subclass of robot to implement the new requirements. We made the Robot super class an abstract class since it would not make sense to have a robot with undefined strength capabilities.

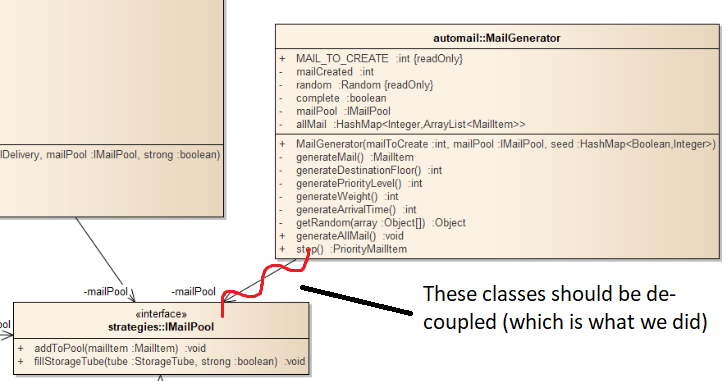
*Old design (single concrete class)*

****

*New design (uses polymorphism, abstract Robot class)*

****

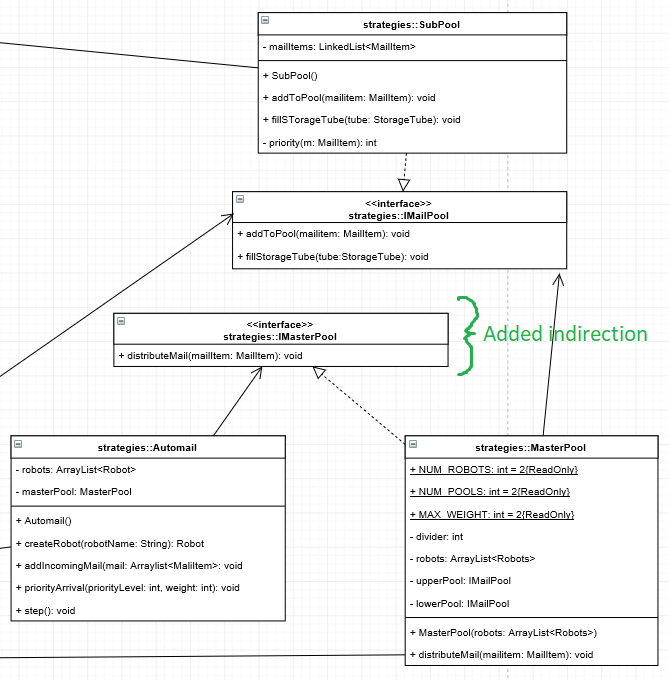
**2**. MailGenerator contains a pointer to MailPool, which reduces cohesion and increases the representational gap for how the system should intuitively work. MailGenerator should intuitively do one thing: generate mail. The existing solution requires simulation to call a method in MailGenerator to directly add the mail to MailPool and call the addToPool method. This reduces cohesion because it gives MailGenerator a responsibility it should not have. Our solution to this is to de-couple MailGenerator and MailPool so that MailGenerator only generates a HashMap of mail. Our system then employs the Simulation class to provide indirection by generating the time ordered mail set from MailGenerator and adding mail to MailPool with each step of time. This is better delegation because Simulation is about directing the time order actions of the various components of the system, and this is one of those actions. If we wanted to have, for instance, multiple MailGenerators and/ or AutoMail systems, the existing solution would not work. Since the specification is simple in this regard it is appropriate to assign Simulation this task of indirection.



**3**. In the existing solution, Simulation directly notifies each individual robot of a new priority arrival. This is poor cohesion and high coupling. Simulation should have delegated priority notification to an appropriate class for the following reasons: Simulation should only be about stepping the main system components through in time order fashion and recording the results, whereas Each robot is a member of an Automail System which has robots and a mail-pool, and it should be the responsibility of this system to notify robots of new arrivals, fill their storage tubes etc. The existing system also creates high coupling because it means if we want to add more robots we need to update the simulation code. Our solution fixes this by using a function within Automail which notifies all robots of priority arrivals, regardless of how many there are. Therefore, after considering the Information Expert principle, we decided that since Automail first receives mail and agregates robots and the Master MailPool, it should take responsibility of notifying robots of new priority arriavals. Therefore, simulation only calls one function, addIncomingMail, and Autommail notifies robots of priority items and adds mail to mailPool.

**4**. In the initial design, IMailPool only works for one configuration of robots, in the given case, weak and strong. This implementation has very poor extendibility, where having a different set of robots requires creating a new <Robot1><Robot2>MailPool implementation of mail pool. This is due to the high coupling between MailPool, Robot and Automail, where MailPool is required to create robot, and robot information is required to create different MailPools. Our solution implements indirection via a new interface (following the strategies pattern) IMasterPool. IMasterPool controls and maintains the mail pools (in our case an IMailPool implementation called SubPool), distributing mail based on MailItem information and coupling robots with their IMailPool. One or more robots are assigned to an IMailPool and return mail directly to this pool if they return with mail. This results in more flexible code, where different IMailPools can be used for different sets of robots. This also leads to better encapsulation, where robots will only interact with a corresponding IMailPool and avoids the previous requirement of giving robot information (such as isStrong) to the pool.

*New design with IMasterPool adding indirection to facilitate multiple mail pools*



**5**. Many variables in the simulation such as building height, simulation run time, robot configurations etc are set as constants in the code. Extending the simulation for different situations requires not only access to the source code but also knowing where the relevant variable is set, the range of changes possible and therefore a general understanding of the whole system, which is considerably expensive. For easier modification of the simulation a common solution is to use a properties file which is easy to understand and use. To facilitate this solution we created a PropertiesLoader class, a singleton pure fabrication which increases flexibility and reusability of the system. Additionally there was reduction in coupling; where these variables were previously passed through objects, now they access the data directly from PropertiesLoader. Although there is new coupling between PropertiesLoader and the classes that use its information, it is still better than the alternative of passing it PropertiesLoader through unrelated objects to the user object.

**Appendix**

**Other changes that were made:**

We added a createRobot method to Automail which takes a robot type tag to create a robot. Reduces repeat code during robot creation. Also, it is clearer, using definitive robot name tags rather than boolean for strong.

We changed robot1 and robot2 to a list of robots. Allows for greater flexibility and extendibility, being able to take input of more than just the required 2 robots and simplifies procedures such as notifying robots of priority mailItems (can run one function which notifies all robots).

ReportDelivery static class in simulation removed, is now a static method instead. Reduces arguments passed into Automail into Robot. Locks us into having one simulation running at a time only but this should be acceptable as reading from the properties file does not support multiple simulations anyway. Removal of interface reduces protection against variation but its minor because the simulation class should be stable compared to strategies where most of the changes are expected to take place (since the whole system is a simulation for testing various strategies).