**Design Analysis Report**

The design analysis report will discuss the changes our group has made to improve the current software design. In addition, it will also suggest a few possible future changes which would make it easier to manage the system.

**1. Design Improvement**

**1.1 Robot subclass - Polymorphism**

In the original design, the robot class is only responsible for representing standard and weak robots. In order to extend the specification, we have made two design changes to it. Firstly, two new attributes are introduced to be used in constructor, namely *careful* (if the robot can take fragile mailItem), and *capacity* (maximum tube capacity). This provides flexibility for extending current design to four robot types. Secondly, we use inheritance to achieve **Polymorphism**. Four subclasses are created (WeakRobot, StandardRobot, BigRobot, CarefulRobot) to represent four types of robots. Robot creation is performed in subclasses with appropriate **super()** method. With subclasses, robots can easily be created with their respective types. We also decide to make Robot *abstract*, since makes no sense to create a robot without knowing what type it is.

**1.2 Reduce coupling – MailGenerator and IMailPool**

**1.3 Reduce coupling – Robot and MailItem**

**1.4 Robot Creation – Adaptors, Factory and Singleton**

One of the biggest changes our group has made to strategies is under robot creation. Originally, robot creation is hard-coded in Automail class. This makes future modification of specifications difficult, and is against the idea of **Creator** in GRASP, since Automail should just be used to add robots to ArrayList rather than actually create them. To solve the problem, we decide to use **RobotAdaptor** together with **RobotFactory**. RobotAdaptor is an interface for robot creation. Four types of adaptors implement it (BigRobotAdaptor etc.) to directly create robot. It adds a level of indirection to varying APIs in other components. Using adaptors provides multiple benefits such as component hiding and **protected variation**. Adaptor is also an example of **polymorphism**, and uses **pure fabrication** to finally achieve **low coupling** and **high cohesion** (From GoF Patterns Lecture).

We create a pure fabrication object, robot factory, to handle the creation of adaptors. It is an abstract factory pattern. It allows responsibility separation and potential strategy improvements in the future such as object caching. We decide to use a singleton as only one instance is allowed to be created. Current robot specification is simple, therefore the factory is relatively small. However, with these design patterns, we delegate creation of objects to Abstract Factory and make it much easier to handle future changes.

**1.5 Misplaced Constant – Information Expert**

Constants should be placed into appropriate classes, to reflect the idea of **information expert**. In the original design, *LAST\_DELIVERY\_TIME* (The threshold for the latest time for mail to arrive) attribute is put in the *Clock* class, which is inappropriate because the class is only a representation for clocking. This is clearly a responsibility issue. To solve that, we place the attribute inside *MailGenerator* class as *LAST\_DELIVERY\_TIME* is only used in mail creation. This assigns responsibility to the information expert.

**2. Further Changes / Considerations**

**2.1 More Robot Types**

In terms of more robot types, we can create more subclasses to inherit from Robot. For instance, after introduction of a *JunkRobot*, which detects junk mails and adds them to junkPool and discard them, we need another subclass of JunkRobot that inherits from Robot. If there are too many robot types, we might add an interface for the Robot superclass to manage Robot types more flexibly.

**2.2 Robot Behaviour Extension**

Our current design for robot creation makes it easier for robot behaviour extension in the future. With the RobotFactory, any further robot actions / requirements can be added to the factory, and adaptors are used to separate responsibility of complex creation logic. For example, robots might need to be created with different responsibilities, such as which floor range to deliver. This can be implemented by expanding RobotFactory with additional components.

**2.3 Property Extension**

In the original design, *automail.properties* is read in the main function in Simulation. In the future, this could be a problem when more properties are added to the specification. For example, we may want to specify the speed of each robot, or the total number of robots needed (e.g. given a limited budget and price of each robot, we wish to maximise efficienty). This can make our main function hard to manage.

Therefore, we could design a new class called **PropertyManager** to handle property inputs uniformly. The class will contain all default properties and they will be overwritten if there are new parameters. The class is designed to solely handle *automail.properties*, which follows the idea of **Pure Fabrication**. It helps us manage file input more easily.