Project 1: Automail – Design Analysis

# SWEN30006 Software Modelling and Design

Nicholas Wong <SID> - Meric Ungor <SID> - Simon Cozens 1071589

# Introduction

This project delivers the latest version of the Automail product to the market. The two major changes in this release are the introduction of a modular Overdrive module for the robots and development of a Statistics Tracker to collect and display data associated with the robot delivery operations. The release also includes some minor fixes and refactoring of code to improve the code base for future development work.

This design covers the solution overview, including a summary of the changes followed by a discussion of the patterns and principles considered during the design phase and concludes with the alternative solutions that we considered.

# Solution Design

## Summary

This design covers the following changes to the Automail simulation:

* Added Statics tracking and statistics classes.
* Added RobotMode abstract class with RobotModeOverdrive and RobotModeNormal concrete classes
* Refactored Robot <-> MailPool interface so that MailPool does not need to know which mode the robot is using nor the carrying capacity of the robot.
* Logic for movement of the robot moved into the mode classes.

Robot to hide addToHands and addToTube (information expert, low coupling) and have a generic addToRobot method. Need to add method torobot hasCapacity() so mailpool can determine if the robot has more space for mail items. MailPool shouldnt be directly placing the mailitems onto the robot hands and tube. Also this would mean the mailpool would need to know which mode to use and that should be the responsibility of the robot. Mailpool just needs to know if it can give the robot another item.

## Patterns and Principles

### Overdrive Mode

#### Discussion

The problem brief mentioned that the overdrive mode may be considered for other uses. Therefore, this led us towards designing modular code which abstracts away the logic of how it modifies the robot’s behaviour and allows for reuse in future designs.

Patterns which were good candidates for the implementation of the overdrive mode included the Entity-Component model, Strategy and Template-method. The entity component model would see us compose the robot of a collection of classes which allows us to implement the delivery mode capability as a discreet component, along with other functions in the robot. In the Strategy pattern we implement the mode as either an abstract class exploiting inheritance to implement overdrive and normal which contain the logic for each type of mode. The Template method takes the abstraction further and implements the generic logic in the base class and then implements the specific logic in the subclass.

The least favourable albeit simple alternative would have been to place all logic for the modes inside the robot itself. While this gives rise to high cohesion and is the responsibility of the Robot, splitting the logic into modes makes the code easier to maintain, allows the modes to be reusable and extensible and also allows for other modes to be plugged in to modify the robot's behaviour without needing to change any code in the Robot.

In terms of GRASP this design is characterised by high cohesion as the mode knows about the cooldown period and how many steps to move the robot complemented by low coupling where the logic on the modes can be different but makes no difference to the implementation of the robot. New modes can be defined which change the behaviour of the robot when delivering.

#### Implementation

The overdrive mode is implemented using the template method involving an abstract class with an abstract method to update (move) the robot. The logic of the mode is then encapsulated within the concrete classes that implement the update method. Logic for movement of the robot has moved into the mode classes. For example, the overdrive mode, when update is called, moves the robot two steps per invocation.

After the MailItem is delivered, the behaviour of the robot is implemented inside the update method of the current mode and either proceeds directly back to the mailroom or if in overdrive, the robot waits 5 steps before the update method moves the robot back to the mailroom.

### Statistic Collection

#### Discussion

The statistics tracker records metrics for the automail system in aggregate and then prints them at the conclusion of the simulation. As the statistics were similar we saw that this could be abstracted away by using a separate class to describe each instance of Statistic and then house these in a StatsTracker class. The StatsTracker is implemented as a singleton because there should only be one, global statistics tracker for the system. We also kept the implementation of the individual Statistics in a separate class to achieve low coupling and repeating code, making the stats tracker simpler to understand than if we implemented the logic all in the one module.

StatisticTracker could also have been implemented as logic in the simulation itself however this would add complexity to the AutoMail class and may be more difficult to maintain as the size of the simulation and complexity increases. We opted to implement Statistics as a separate class to contain the logic required for collecting, calculation and reporting of statistics. We also opted to make each statistic a class of its own so that statistics can be easily added in the future.

#### Implementation

In our chosen implementation each statistic is implemented as an object which has a name, type (count, sum average), message text allong with an overrided toString method to print the statistic at the end of the simulation run. This allows statistics to be extensible and to contain the logic for summing and printing the statistic abstracted from the StatsTracker. More statsitics can easily be added to the system or the logic can be changed per type of statistic in a future release. In this release, the robot makes a call directly to the StatsTracker to notify it of a successful delivery and associated meta-data. This could have been decoupled using the controller pattern to separate the robot from the StatsTracker to anchor both of them to the simulation itself, meaning changes to the robot or StatsTracker are unlikely to directly affect each other and removes the need for each other to maintain knowledge of the other.

# Alternative Solutions

We have described alternative patterns in the sections above.