SWEN30006_2020_SM1 Assignment 1 [RoboMail] Report

Group 68: Ingunn Fløvig, Crystal Brazilek, Xinyi Liu

In the process of modelling the behaviour that a robot must exhibit whilst in caution mode, we have identified the areas of the current design that must be changed or extended, and the responsibilities associated with achieving this behaviour. They are as follows:

Problem 1:

How do we allow robots to act ‘cautious’ if caution mode is turned on, but remain unchanged if it is turned off?

Responsibilities:

Robots must perform all tasks associated with handling fragile mail when caution mode is turned on.

Considerations:

Conversely, this also means that robots must not perform these tasks if caution mode is not on. For example, in non-caution mode, a robot must break a fragile item if it attempts to store it. Robots must not attempt to wrap fragile items if they are not in caution mode.

Our implementations and reasoning:

We have used polymorphism in the form of creating a subclass named ‘CautiousRobot’ that inherits from the ‘Robot’ class. This is because a CautiousRobot should be able to do all the tasks that an ordinary Robot does, with the added methods and attributes required to implement the behaviour needed to handle fragile items. If caution = true, the MailPool class initiates a list of CautiousRobots. If caution = false, the Automail class initiates a list of ordinary Robots without the methods required to handle fragile items. In doing so, we do not increase coupling beyond what already exists, and the design is extendable, allowing for subclasses of robots who can be used to handle different scenarios to be implemented in the future.

***

Problem 2:

How does a robot retrieve fragile mail?

Responsibilities:

A cautious robot must retrieve fragile packages only with its special arm. The robot must retrieve normal packages only with its normal arm/tube.

Considerations:

A cautious robot must be able to differentiate between fragile and normal mail, with the ability to load all 3 of its arms with the appropriate mail type if there is necessity to do so. It must not load more than one package onto each ‘slot’, and must not attempt to load incompatible mail types onto each of its ‘slots’.

Our implementations and reasoning:

A change was made to the MailPool class, wherein a separate linked list was created for fragile items. The ‘pool’ linked list stores all non-fragile MailItems, whilst ‘fragilePool’ stores all fragile MailItems. This change increases cohesion and allows for cautious robots to more easily differentiate between fragile and non-fragile packages.

A new method, ‘addToSpecialArm()’, was created in the CautiousRobot class. This method is accessed by the MailPool class to assign a fragile item to the ‘specialArm’ attribute of a cautious robot. Whilst this method performs similar functions to ‘addToHand()’ and ‘addToTube()’ and therefore its actions are possible to replicate simply with the use of conditional statements within these methods, it has been separated to increase extensibility and ease of modification of the program. The method is assigned to the CautiousRobot class, as it is the Information Expert in this instance, having access to the MailPool and its own attributes for storing the mail packages. As the class contains the aforementioned similar methods, this also increases cohesion whilst not creating any additional coupling.

In this instance, polymorphism in the form of a subclass for fragile mail items was considered, but ultimately deemed unnecessary, as although fragile mail must be treated differently by other classes, the boolean ‘isFragile’ is enough to indicate this to other classes. To do anything further would be to add unneeded complexity.

***

Problem 3:

Wrapping and unwrapping fragile items

Responsibilities:

A cautious robot must wrap a fragile item upon receiving it, taking 2 units of time. It must then unwrap the item before delivery, taking 1 unit of time.

Considerations:

A cautious robot must not exhibit this behaviour if items are not fragile. It must not exhibit this behaviour if it is not carrying anything in its special arms. A normal robot must never exhibit this behaviour.

Our implementations and reasoning:

We overrided the 'step()' method from the Robot class into the CautiousRobot class, implementing code needed to track wrapping. Two attributes were added to track the status of wrapping and unwrapping, and the case statement would wait the appropriate units of time to exhibit these behaviours, if the condition of 'isFragile' is met. Overriding was done in interest to preserve the behaviour of the system in instances where caution mode was turned off, as a normal robot initialised as a Robot object would not have access to this extended 'step()' function.

CautiousRobot also has a method 'fragileProtocol()' that uses methods in the MailItem class 'wrap()' and 'unwrap()'. Within the MailItem itself, there are attributes that track whether an item has been wrapped, and helper attributes that track the unit of time taken. This protocol is only available to the step() function of a cautious robot and thus will never be used by a robot if Caution mode is not true. This once again demonstrates the importance of introducing <u>polymorphism</u> in the form of the CautiousRobot subclass. Although the functionality 'fragileProtocol()' can theoretically be coded straight into the 'step()' function, we have separated this to <u>increase cohesion</u> and ease of extendability. It's also important to note that although 'fragileProtocol()' uses methods from MailItem, the Robot class already has an association to MailItem, so in doing so, we <u>have not increased coupling.</u> It's also possible to 'track' the 'wrapped' status of an item by only using a flag in the Robot class, but adding this as a boolean to the MailItem class itself helps us <u>lower the representational gap.</u>

***

Problem 4:

A cautious robot must only deliver a fragile item if it is the only one on the floor.

Our approach:

We have decided to achieve this through 2 steps:

1. Prevent other robots (currently not on the floor) from accessing a floor once a robot with a fragile item arrives.

2. Make robot carrying a fragile item wait on the destination floor until other robots have departed before delivering the item.

We will now elaborate on this:

When a robot reaches the floor it intends on making a fragile delivery on, it will call a clear floor function. This allows all the robots currently on the floor to finish making their deliveries and leave, but prevent any further robots from entering the floor. Once the floor is empty, the robot making the fragile delivery will be free to unwrap and deliver its package.

Responsibility:

Prevent other robots (currently not on the floor) from accessing a floor once a robot with a fragile item arrives.

Considerations:

Since no new robots are entering, the floor will eventually be cleared for the robot with the fragile item to deliver.

Our implementations and reasoning:

We modified the Building class into a class called ‘BuildingSpecs’ to store information about floors. In this class, there is an array named ‘floorClearCalled[]’, with each element being a boolean corresponding to a floor in the building. A robot carrying a fragile item must use the ‘clearFloor()’ method to set their floor to ‘true’ once they are on the destination floor. If ‘floorClearCalled[]’ returns ‘true’, robots are not allowed to access that building and must wait until ‘floorClearCalled[]’ returns ‘false’ again, which the fragile robot will do once it has left the floor. ‘BuildingSpecs’ is a <u>pure fabrication</u>, as there is no physical object that stores these arrays in the problem domain. We believe that implementing this class <u>reduces coupling</u>, as robots check BuildingSpecs instead of a list of robots (stored in MailPool) to observe the presence of other robots. Each robot only has to interact with BuildingSpecs, so any changes to the floor clearing protocol would also be easier to implement in the future. It is also <u>cohesive</u> to dedicate a class to the functionality of floors.

Responsibility:

Make robot carrying a fragile item wait on the destination floor until other robots have departed before delivering the item.

Considerations:

A fragile robot must know when a floor is cleared before it can begin unwrapping and delivering a fragile item. Therefore, it needs to know if there are any robots apart from itself on the floor. But, it does not need to know any details or what specific robot is on the floor, it only needs *the number* of robots on the floor.

Our implementations and reasoning:

In ‘BuildingSpecs’, there is an array that stores the number of robots present on each floor during each unit of time, called ‘robotsOnFloor[]’. A robot will increment robotsOnFloor[current_floor] by 1 when they enter, and decrement it when they leave a floor. The robot wishing to deliver the fragile package must check that the robotsOnFloor variable is ==1 before they can commence unwrapping.

Again, the purpose of storing this information in the fabricated class BuildingSpecs is to <u>decrease coupling</u> and <u>increase cohesion</u>, as to not have to check every single robot for their location to determine if any robots are present on the floor.

***

Miscellaneous changes/extensions and explanation:

Statistics tracking was added by tracking each individual robot’s statistics, then adding these together in the Simulation. This is because this information needs to be accessible via the Simulation class, but this class can’t track the info, or increment these variables itself without a lot of added coupling. It seems natural then, to have robots track themselves, then have the simulation access this through getters. Since Simulation is already coupled with robot classes, there is no increase to coupling.

CautiousRobot has an additional method called ‘ExecuteDeadlockAvoidance()’ to safeguard against a situation where more than one robot carrying a fragile item arrives on the same destination floor. In this case, the first robot whose ‘unableToWrap’ counter (which increments for every time unit that the robot detects that the floor is not cleared) reaches 5 will move to the floor below, clearing the way for the other robot to deliver.