Mailbot 2: Judgment Day

Brent Cahill,[1, a)] Nicholas Edsall,[1, b)] and Nicholas Spain[1, c)]
School of Computing and Information Systems, University of Melbourne

(Dated: 18 September, 2018)

We present a look at the extended design and implementation of software for a robotic mail delivery system Automail for *Robotic Mailing Solutions Inc.*

I. INTRODUCTION

Continued work with *Robotic Mailing Solutions Inc.* has revealed significant flaws in the current Automail design and implementation. The current system supports robots of two types:

1. A "standard" robot, with a tube capacity of four and an essentially unlimited maximum single item weight.
2. A "weak" robot, also with a tube capacity of four, and a maximum single item weight of 2000 grams.

Robotic Mailing Solutions Inc. has contracted us to extend this implementation to two further robots, a "big" robot, and a "careful" robot, as defined below. In addition, we have been asked to take consideration for possible types of robots that may be added in the future, and therefore generalize the implementation of the robot class.

3. A "big" robot, with a tube capacity of six and an essentially unlimited maximum single item weight.
4. A "careful" robot, also with a tube capacity of three, an essentially unlimited maximum single item weight, but half the speed of the other three robots.

A. Fragility

Currently, *Robotic Mailing Solutions Inc.* has no method of delivering "fragile" mail items. Both the "standard" and the "weak" robots move at a pace too fast for careful handling of fragile items, and therefore break them. Implementation of a "careful" robot to the mail simulation is desired, however the "careful" robot does have two important limitations:

1. The robot moves at half the speed of the other robots.
2. The robot can only carry one fragile mail item at a time.

a)Electronic mail: bcahill@student.unimelb.edu.au
b)Electronic mail: nedsall@student.unimelb.edu.au
c)Electronic mail: nspain@student.unimelb.edu.au

B. Modification of Existing Implementation

Currently, the framework resulting from the engineers at *Robotic Mailing Solutions Inc.* has been immobile, inflexible, and difficult to modify. The proposed solution therefore has taken great lengths to ensure a dynamic and generalized implementation that may still utilize the Automail simulation's existing behaviour.

II. SOFTWARE DESIGN

Since the implementation had to continue to utilize the existing Automail simulation framework, the very first changes that we made were to ensure that the existing simulation framework could accept the new generalized robot type. We did this without changing the true framework of the simulation, simply by migrating the `enum` such that it represented the `RobotType` in the `Simulation` class.

Changes were made to the `StorageTube` class in order to account for the possibility of having a capacity different from 4. This was achieved by creating two different constructors. One constructor had no arguments, and created a storage tube with a capacity of 4 by default, whereas the other took an argument for the capacity of the tube. In addition, we added two separate methods, `getItems()` and `sort(Comparator<MailItem> comparator)`. These methods got all of the items in the tube and sorted the items in the tube in reverse order according to the passed comparator. The former method was used in the latter in order to get all of the items from the tube in order to perform the sorting.

Moving forward, the next logical changes made were to the robot class. In order to generalize, an `enum` was added to the class to determine the `RobotType`, and it was decided that the four different types of robots would be made into subclasses. This meant that the variables would need to be changed to `protected`, rather than `private`, so that we can access the members from child classes both inside and outside of the package. In addition, we added a `pace` variable and a `move_cooldown` variable so that we can account for the move speed of the "careful" robot. We also added a `canTakeMailItem` method that initially ensures that the robot can only take a mail item if it is not fragile, and it does not push the robot over capacity. Finally, we added a `fragileDeliveryCounter`, which we used to ensure that the careful robot was only taking one fragile item at a time.

In the constructor for the class, the only changes we made were to simply initialize these new variables. In most other functions, we simply had to account for the new variables, and add generalization. A lot of the checks were hard-coded, for example, checking if the `deliveryCounter` was greater than 4, rather than checking if it was greater than the tube `MAXIMUM_CAPACITY`. Once these smaller changes were made, the only remaining changes had to be added to the `moveTowards` method. Because the previous framework had not accounted for differences in speeds, changes had to be made in order to generalize the methods and account for any future differences in speeds (for example, a "fast" robot).

In this new `moveTowards` method, we first check to ensure that the robot is not "cooling down," in which case, we do not move it. Otherwise, we simply check to see where the destination floor is, and move the robot either up or down the floors, as before. Finally, we reset the cooldown value so that the "careful" robot's speed is kept at half of the others.

The only remaining changes that we made were a function that actually does the resetting of the cooldown value that we stated above, called `resetCooldown()`, in addition to mutator and accessor methods for the pace value, in order to keep with the principle of encapsulation.

A. Robot subclasses

The four robot subclasses that were added were relatively simple extensions of the `Robot` superclass, with `CarefulRobot` being the most complex.

1. The `CarefulRobot` subclass implemented all of the special features of the "Careful" robot as defined in the project specifications. It inherited all of the constructor arguments from the `Robot` class, but added that the tube capacity in this case was only 3, set the pace to 2 (meaning that it could only move once every 2 steps), and initialized the cooldown value. In addition, the `moveTowards` method was overridden, ensuring that the robot did not break the fragile `Mailitem` it was delivering, should it have only one. Finally, the `canTakeMailItem` method was also overriden to allow the robot to take fragile `MailItems`.
2. The `StandardRobot` subclass inherited everything from the `Robot` superclass, which makes sense, since it is the "Standard" robot.
3. The `StrongRobot` subclass added a tube capacity of 6, and inherited everything else from the `Robot` superclass.
4. Finally, the `WeakRobot` subclass changed the value of the `strong` boolean to `false`. In addition, the `canTakeMailItem` method was overriden, in order to ensure that the robot could not take any `MailItem` that had a weight of over 2000 grams. The `WeakRobot` inherited everything else from the `Robot` superclass.

B. Strategies

The changes made to the existing Automail Strategies were limited to simple adaptations to account for changes in the new robot implementation. In the `Automail` strategy class, the only changes made were in the constructor – an argument that contained a list of the new robot types – and in the initialization of new robots, in order to account for the different types of robots. Both `IMailPool` and `MyMailPool` were left unaffected.

The new strategy, called `MyMailPool2`, however, was entirely different to the original `MyMailPool`, based upon the strategy that Nicholas Edsall created for Part A. The new strategy implemented a greedy approach to sorting the mail, utilizing one queue for fragile items that sorted the mail based on priority, then arrival time, and a queue for non-fragile items that sorted in the same way. As robots arrive, they are loaded one at a time with as many items as possible, then ordered to deliver in priority order. Careful robots are given only one fragile item at a time, should any be available, otherwise, they are loaded similarly to the standard robot, albeit with only a capacity of 3.

C. Stategy Pseudocode

Going into more detail, we can outline the new strategy as follows:

To outline our new strategy, we have 4 essential functions:

1. `addToPool(Mailitem mailitem)`
2. `step()`
3. `fillStorageTube(Robot robot)`
4. `findItem(Robot robot)`

The first of those functions can be outlines as follows:

Algorithm 2 step

```
procedure STEP
    input : mailItems – pool of standard MailItems
    input : fragileItems – pool of fragile MailItems
    input : robots – list of usable robots
    if mailItems !empty or fragileItems !empty then
        for robot ∈ robots do
            fillStorageTube(robot)
            if robot tube !empty then
                sort robot tube by destination floor
                robot.dispatch()
        end
```

Algorithm 1 addToPool

```
input : m – MailItem to be added to pool
fragileIndex : m – current fragile index
standardIndex : m – current standard index
procedure ADDTOPOOL
    Access fragile or standard pool coinciding with m.type
    if m is priority then
        Set it ← (PriorityMailItem)m
        Set it index to 0
        for i ∈ pool do
            if it priorityLevel < i priorityLevel then
                it index++
            else
                break
        end
        Insert it into pool at it index
        if it is fragile then
            fragileIndex++
        else
            standardIndex++
    else
        add m to pool
```

In our next method, we simply are outlining the occurrences at a single time step:

Our third method, `fillStorageTube` does as the name suggests: fills the storage tube of the robot passed as an argument:

Our final method attempts to find a `MailItem` for a given robot – passed as an argument – to deliver.

III. DIAGRAMS

After thorough examination of the software requirements, a *Design Class Diagram* (DCD) was created in order to further understand the interrelation between classes and aid in the speedy and robust final implementation. As can be seen in Figure 4, the classes illustrated in the DCD are only those that have been changed or added, or those classes that are essential for the DCD's ability to be understood. It is also important to note that the Simulation is in a separate package.

After the creation of the design class diagram, a design sequence diagram, Figure 5, was created to aid in the

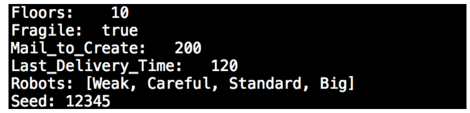

FIG. 1: The initial parameters to the new simulation.

FIG. 2: The results from the new simulation, given the above parameters.

understanding of the basic workings of the software. This was critical in the creation of the final product. This aided the engineers in the understanding at a high level between the mailroom, robot and delivery process, as well as a deeper understanding of the individual methods required for the simulation.

IV. RESULTS

The final implementation successfully accounted for all four of the robot types, as well as allowing for the easy creation of future robots down the road. Most importantly, the new implementation allowed for the carriage and delivery of fragile mail items, whereas the previous implementation simply did not, as can be seen in Figure 1 and Figure 2.

We also display of the failure of the previous imple-

Algorithm 3 fillStorageTube

```
procedure FILLSTORAGETUBE
    input : mailItems – pool of standard MailItems
    input : fragileItems – pool of fragile MailItems
    input : robot – robot to fill
    Set tube ← robot tube
    while tube !full do
        if item !null then
            try
                add item to tube
                if item is fragile then
                    if item is priorityMailItem then
                        fragileIndex- -
                    remove item from fragileItems
                    return
                else
                    if item is priorityMailItem then
                        standardIndex- -
                    remove item from mailItems
            catch TubeFullException
                e.printStackTrace()
            catch FragileItemBrokenException
                e.printStackTrace()
            end try
        else
            return
    end
```

Algorithm 4 findItem

```
procedure FINDITEM
    input : mailItems – pool of standard MailItems
    input : fragileItems – pool of fragile MailItems
    input : robot – robot to fill
    Set tube ← robot tube
    if tube !full then
        if robot is CarefulRobot then
            for item ∈ fragileItems do
                return item
            end
        for item ∈ mailItems do
            if robot.canTakeMailItem(item) then
                return item
        end
    return NULL
```

mentation when given fragile mail items in Figure 3


```
Floors:    10
Fragile:  true
Mail_to_Create:   200
Last_Delivery_Time:   120
Robots: [Weak, Standard, Standard]
Seed: 12345
T:   0 > R0(0/4) changed from RETURNING to WAITING
T:   0 > R1(0/4) changed from RETURNING to WAITING
T:   0 > R2(0/4) changed from RETURNING to WAITING
T:   1 > new addToPool [Mail Item:: ID:    168 | Arrival:    1 | Destination:  9 | Weight: 1577 | Fragile:  no]
T:   1 > R0(0/4) changed from WAITING to DELIVERING
T:   1 > R0(0/4)-> [Mail Item:: ID:    168 | Arrival:    1 | Destination:  9 | Weight: 1577 | Fragile:  no]
T:   2 > new addToPool [Mail Item:: ID:     35 | Arrival:    2 | Destination:  5 | Weight:  272 | Fragile:  no | Priority:  50]
T:   2 > R1(0/4) changed from WAITING to DELIVERING
T:   2 > R1(0/4)-> [Mail Item:: ID:     35 | Arrival:    2 | Destination:  5 | Weight:  272 | Fragile:  no | Priority:  50]
T:   4 > new addToPool [Mail Item:: ID:      9 | Arrival:    4 | Destination:  3 | Weight:  758 | Fragile: yes | Priority:  60]
T:   4 > new addToPool [Mail Item:: ID:     44 | Arrival:    4 | Destination:  3 | Weight:  734 | Fragile:  no]
exceptions.FragileItemBrokenException: Fragile item broken!!
Simulation unable to complete.
        at automail.StorageTube.addItem(StorageTube.java:54)
        at strategies.MyMailPool.fillStorageTube(MyMailPool.java:80)
        at strategies.MyMailPool.step(MyMailPool.java:69)
        at automail.Simulation.main(Simulation.java:106)
```

FIG. 3: The original framework failed to deliver fragile mail items, and did not include an implementation for "Careful" or "Big" robots.

V. DISCUSSION AND FUTURE WORK

We feel confident in our new framework, and think that this is a very suitable implementation for *Robotic Mailing Solutions Inc.* to use going forward. It provides a solid framework that is sufficiently generalized for them to continue to add new robots, and maintains high cohesion and low coupling.

Going forward, it would perhaps be beneficial to modify various aspects of the implementation, inlcuding:

1. Considering arrival time during the sorting of the queue.
2. Distribute items between the robots rather than loading them to capacity one at a time.
3. Monitor when the robots are expected to return to plan deliveries.
4. Give "Careful" robots the lowest priority for jobs

In all, however, we feel very confident in our implementation, and feel as though it satisfies the desires and requirements as stated in the project guidelines.

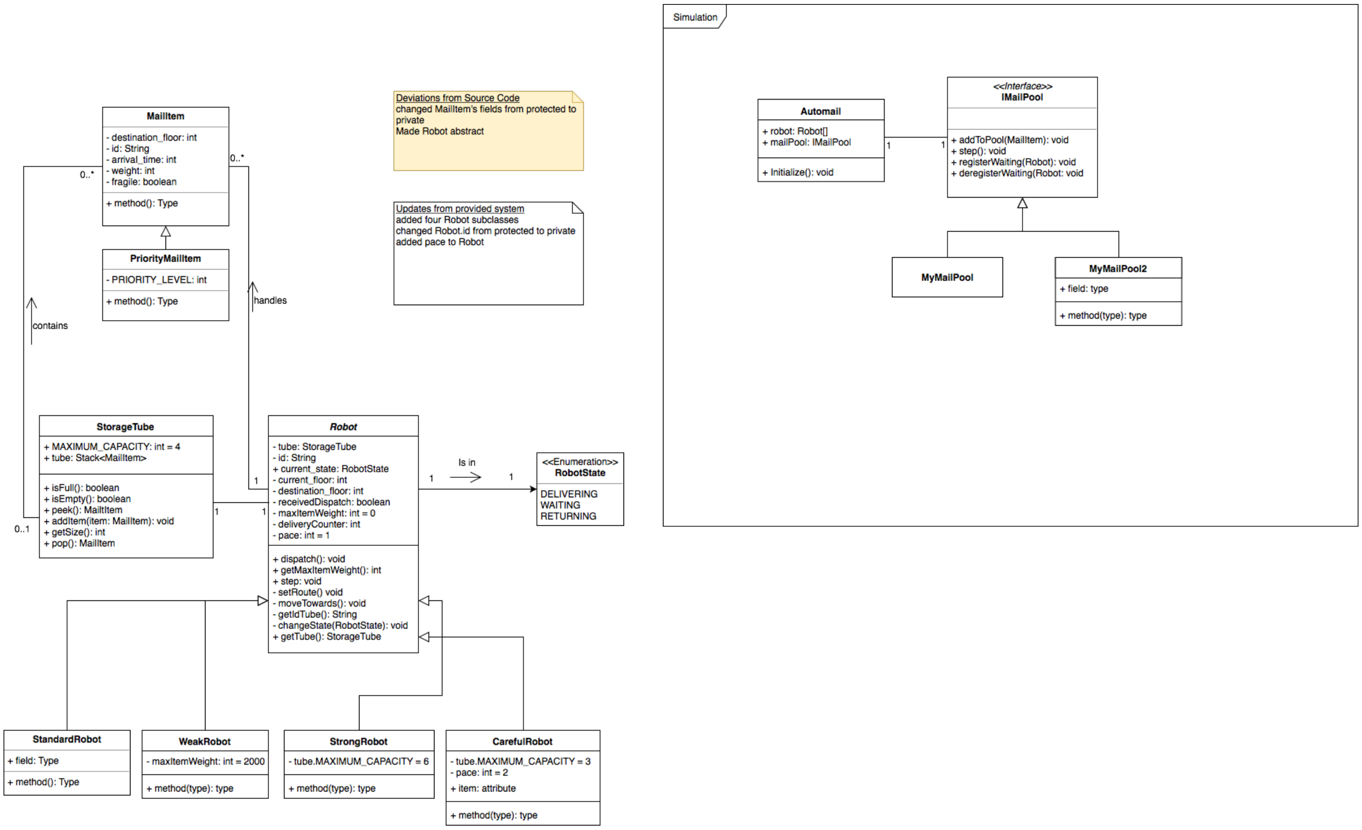

FIG. 4: The Design Class Diagram, illustrating updates and showing class interrelationships.

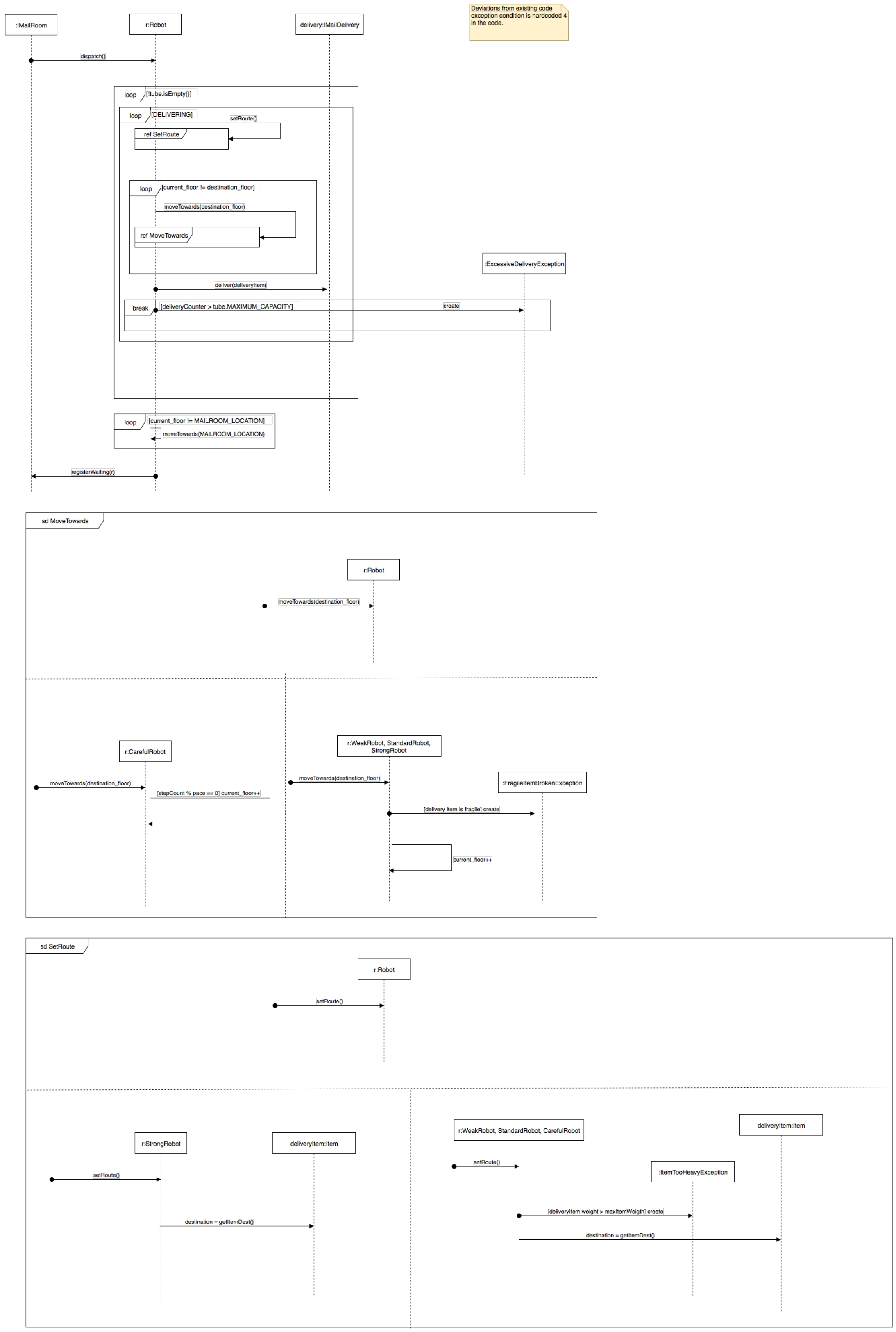

FIG. 5: The Design Sequence Diagram.