SWEN30006 Project 1: Robomail Revision
Design Analysis Report

Team Number: WS12-3
Group member: XuLin Yang(904904), Zhuoqun Huang(908525), Renjie Meng(877396)

In the static design class diagram, all classes have an “a” on its top left corner, this is a reference of UML class used by our UML diagram software, which has no meaning, you could simply ignore this.

Refactoring:

In MailPool, firstly, we change robots and pool attributes with ArrayList data type instead of LinkedList because we need to assign one heavy mail item to multiple robots instead of one robot. We refactor and modify the loadRobot method by applying Strategy pattern twice here, which is shown in the middle right part of figure 2, that is we change the loadRobot method to ISelectMailItemToDeliverPlan, ISelectRobotToDeliverPlan. The reason why we want to have this design is that it provides a high cohesion. MailPool could focus on give mail item to the robot’s hand or tube, while the responsibility of how to load mail item to robot has been delegated to these two strategies. ISelectMailItemToDeliverPlan is responsible for selecting mail items to delivery as well as ISelectRobotToDeliverPlan is responsible for returning robots to take those mail item to be delivered. Secondly, we also have some helper methods in MailPool such as cloneList, unregisterWaitingRobot and unregisterUnLoadedMailItem. Thirdly, we get rid of private class Item as well as the ItemComparator. The reason is that in our opinion the Item class is redundant it only coordinate with the ItemComparaotr to provide a way of sorting Mail Item, however, we could add a class called MailItemComparotor to sort mail item directly, by doing this we reduce the coupling of classes, it is extendable as well, if the way of sorting mail changed we could modify the MailItemComparator to address this, without changing other classes. And we make the functionality of set MailItem with priority with 1 inside the Comparator. Fourthly, we let step() has a return value of ArrayList<IRobot>. The reason is that when the mailPool is stepping, it will use two given strategies to decide an IRobot (single robot or a robot team with 2 or 3 robots) to start delivering and return all generated IRobot to Automail to step these generated IRobots. These are all the changes of MailPool class demonstrated in the static design diagram.

In Automail, firstly, we change mailPool and robots with private to achieve data encapsulation. Secondly, we change robot array with two ArrayList<IRobot> to store the robots to be stepped in the system. The reason why we need these two Lists will be discussed in the talk of step() of Automail later. Thirdly, we move the stepping of two components from Simulation to the AutoMail. The reason is that Automail has information on all components. Thus the Automail has the responsibility to update the system. As a result, we make the Automail more cohesive. Inside the step(), we step the mailPool and add returned IRobot to the List for the IRobot will step in the current time frame. And step each IRobot and add returned the IRobots to be stepped in next time frame When the loop finished, swap these two lists. Alternatively, we let automail be accessed by two components in order to let both of them can register IRobots to be stepped. This is not a good design because it increases the coupling between two of them with the Automail. As a result, we choose the current option instead. These are all the changes of Automail class illustrated in the static design diagram.

In MailItem, we change the privacy of for attributes: destination_floor, id, arrival_time, weight from protected to private. The reason is that this can achieve better encapsulation in the design. These are all the changes of MailItem class shown in the static design diagram.

Extendability and modifiability:

In ISelectMailItemToDeliverPlan interface, there are three method, generateDeliverMailItemPlan, hasEnoughRobot and getPlanRequiredRobot. First, generateDeliverMailItemPlan takes unloaded mail item from mailPool, then return a list of mail item to be delivered back to mailpool. Secondly, hasEnoughRobot takes a list of mail item to deliver any number of robots to decide whether there is enough robot. Thirdly, getPlanRequiredRobot takes a list of mail item to delivery and return number of robot required. The reason why we use this is that it forms a strategy pattern together with SelectMailItemToDeliverPlan which implements this interface (mentioned in the change of MailPool class) which provide a good extendability, if there are new way of delivering, For example, if now a new kind of 2-item-tube robot become available which could carry 2 mail item in its tube, we just need to provide a new implementation of ISelectMailItemToDeliverPlan. As a result, we achieve an extensible and easy modifiable design. Above, all these the changes of ISelectMailItemToDeliverPlan interface and SelectMailItemToDeliverPlan class illustrated in the static design diagram.

In ISelectRobotToDeliverPlan, there is a method selectRobotToDeliver, which is responsible for selecting robots to carry mail item from waiting robots at the mail pool. The reason why we use this design is similar to the reason for ISelectMailItemToDeliverPlan, we could apply a strategy pattern by using this interface (mentioned in the change of MailPool class), which provide extendability as well as increasing cohesion of MailPool as discussed before. What's more, these two objects are loosely coupled with the Mailpool. As a result, a modification in them won't require modification in other classes. This achieves the variation protection in the grasp as well as satisfy the open-close principles. Moreover, using the Strategy pattern can improve readability because, while a class which implements some particular strategy typically should have a descriptive name and the avoidance of using a lot of if/else statements are better for understanding. Alternatively, if we do not use strategy pattern for these two, by using if/else instead, all of the implementations are tied to the implementation of MailPool, making it harder to change in the future. Although strategy increases the complexity of the system (i.e.: more objects in the system), the advantages discussed before is more important. By using the strategy, the only coupling is to the interface of the strategy. Above, all these are all the addition of selection strategies described in the static design diagram.

In RobotFactory, we create this class by applying the singleton and concrete factory pattern, because this design would provide high cohesion as well as high extendability. Although the use of the singleton pattern can create hidden dependencies (As the Singleton is readily available throughout the code base, it can be overused. Moreover, since its reference is not completely transparent while passing to different methods, it becomes difficult to track.), this won't be the problem because we have limited use of the factory in our implementation. Although the use of the singleton pattern provides a bad control of encapsulation, we have no attribute except the factory. So this disadvantage of single pattern won't result in the problem. The use of a single pattern can be beneficial because singleton pattern prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance. In other words, it won't take too much burden to the program without creating too many duplicate objects. In our system, we need to create Robot and RobotTeam in many places and we want to encapsulate the object creation logic instead of having it everywhere since it would be horrible if the creation logic changes (because we need to change everywhere). However, with RobotFacory, we only need to modify this class if the creation logic changed, such as a new type of robot is added into the system.

By creating 3 new classes, IRobot, Robot and RobotTeam, we apply a variant of composite pattern, which is illustrated in the left part of figure 2, IRobot is the component, Robot is the leaf and RobotTeam is the composite, but it owns Robot instead of IRobot. Before discussing each class in detail, let's first discuss why we apply the composite pattern here. Firstly, the **Composite** pattern makes the client simple, because in our design AutoMail acts as the client which has to step each Robot, with the composite pattern it does not care it is a Robot or RobotTeam as long as it implements the interface. Alternatively, if we apply alternative design 3, the automail need to check the type first, then implement step. Secondly, the composite pattern provides low coupling. Because in our design AutoMail only need to know IRobot instead of both Robot and RobotTeam, which reduce coupling between them relatively. Thirdly, the composite pattern is difficult to restrict the components of the composite to only particular types, since it could own itself as well, however, if we make RobotTeam owns Robot, then we restrict TeamRobot to own specific type only which is Robot.

In IRobot, this interface, which implemented by both Robot and RobotTeam implement, declares some methods. Firstly, there some responsibility IRobot needs to do before dispatching. There is an inner class IRobotComparator, this class is responsible for sorting IRobot in the order of robot ID before AutoMail step IRobot, since we notice that in the given system robot is stepping in the order of robot id. Then, listMailItems and listRobots method are responsible for returning all mail item and robot in IRobot, which is used in MailPool for unregistering them from wait robots and unloaded mail item respectively. canAddMailItem method is responsible for checking whether this IRobot could carry a mail item or not. addMailItem method is responsible for adding mail item to IRobot, which would raise an error if the mail item is not able to be added. canDispatch() method is responsible for checking whether the IRobot could dispatch or not, and dispatch() method is responsible for dispatch IRobot. Robot and RobotTeam provide different implementation to this method, in Robot it only dispatch itself, but in RobotTeam. it dispatches all Robot it has one by one. Secondly, after dispatching, canStartDelivery, startDelivery and the deliver method is responsible for delivery of IRobot, canStartDelivery is responsible for checking whether IRobot could deliver or not, startDelivery and deliver are responsible for delivering and report the mail item. Robot and RobotTeam provide different implementation to this method, in Robot, it delivers mail item, but in RobotTeam, it chooses only one robot to deliver, since each mail item needed to be delivered only once. moveTowards method is responsible for the movement of IRobot. changeState method is responsible for changing the state of IRobot. getCurrentMailItem and getFloor methods are responsible for return current mail item and current floor, this is used at RobotState, which discussed below, to provide visibility of destination floor of this mail item and current floor of Robot or RobotTeam. hasNextMailItem and loadNextMailItem methods are responsible for checking if there is next mail to be delivered and load next mail item to be delivered. registerWaiting method is responsible for registering IRobot to the waiting robots in MailPool when it has returned to the MailPool. availableIRobots method returns all available IRobot. Robot and RobotTeam provide different implementation to this method, in Robot it always return itself. However, in RobotTeam, if the RobotTeam is delivering, it would return itself, but if it is returning, it would a list of team member, which are instances of Robot. Several getter to provide visibility, there are getId, getRobotState, getTeamState. And,

changeTeamState is responsible for modifying TeamState, which discussed below. Finally, step method is responsible for taking next step for IRobot. These are all the change we made inside IRobot.

In RobotTeam, we create this class to represent the sense of a team. When there is one heavy item, the robot would form a Robot Team in order to carry this heavy as a team. Compare to the alternative 1 discussed in below alternative design section, now AutoMail has an ArrayList of sorted IRobot, it would step each IRobot one by one, all the member belong to the same team would step in consecutive seconds, which is exactly what we desired.

In Robot, we made several changes to this class by implementing IRobot interface, which has been discussed in the IRobot class. And, factor the RobotState Out, which is been discussed below. We remove the destination floor attribute and setRoute method. Currently, we think the Robot does need to know where it should go, if it needs this information, it could ask the deliveryItem what is the destination floor.

In RobotState, we change the robot state in Robot to a RobotState Enum class which implement IRobotState, each state overrides the step and postDelivery functions declared in the interface. The step method is responsible for taking the next step according to the state, and the postDelivery method is responsible for updating IRobot after the mail item is delivered to the destination.There are several reasons for this design. Firstly, this enum class can hide the complex logic of checking which state the IRobot is at because, inside the step method of robot, it implements this.tobotState.Step, this would force the transition of IRobot's state. Secondly, this design is more robust, since the state transition could happen only in this RobotState class, that is other class like a robot would not need to check state anymore, then there will be no chance of having errors in the transition of states by using state machines.

In TeamState, we move INDIVIDUAL_MAX_WEIGHT, PAIR_MAX_WEIGHT and TRIPLE_MAX_WEIGHT which are originally in the Robot class to the newly created enum class TeamState, the reason of creating this class is based on pure fabrication. This class implement ITeamState interface, which declared two methods validWeight, getNRequiredRobot. The robot needs to know exactly how heavy it could carry, if the item is too heavy it should throw an ItemTooHeavyExcaption. By include TeamState as an attribute of Robot, it could know which kind of team it is in, like single, double or triple, then it could ask TeamState what is his max weight to carry. It is also extendable, for example, if our robot has been upgraded, it could carry more weight now, we only need to change the constant inside TeamState. Or the IRobot has a change in its behaviours, we can change the state class without influencing other parts of the code.

We add three new exceptions as well, there are InvalidAddItemException, InvalidDispatchException and NotEnoughRobotException. Also, we remove the ExcessiveDeliveryException. InvalidAddItemException is used in the method addMailItem to indicates the mail item be added to the IRobot. InvalidDispatchException is used in the method dispatch in IRobot to show the Robot cannot be dispatched yet. NotEnoughRobotException is used in the Simulation to indicate that there is mail item's weight exceed the max carry weight of all available robot in the current system.

Alternative Design:

During the refactoring, team also consider several other alternatives, however, each of them has some flaw.

Alternative 1:
In this alternative, in order to extend the current system to deal with heavier mail items by robots collaboration, we introduce two new classes, Task, TaskGenerator as well as an interface, ITaskGenerator.

We refactor MailPool's way of loading mail items to robots in order to handle team collaboration. In the given system, it loads each item to exactly one hand or one tube. However, if there are heavier items which need more than 1 robot, then we need to load this mail item to several robots' hands. Instead of implementing this inside MailPool class, we decide to create a new class called TaskGenerator which implement ITaskGenerator based on the strategy pattern which increases cohesion and provide good extendibility

MailPool is responsible for creating TaskGenerator based on Creator pattern since it "contains" TaskGenerator.

After MailPool loading items to the robot, the robot would have a Task object which is responsible for guiding the robot to the destination floor.

The reason for rejecting this design is that it is not logically correct. In reality, those multiple robots carry the same heavy item should move at the exactly same time. However, in this system, each robot “step” separately, this means they move sequentially, not move at exactly same time, but we could try to approach this behaviour by making robots which belong to the same team moving in exactly consecutive second, that is at x second team member A move, at x+1 second team member B move, this team stop move until all team members have moved. But in this design, there is no sense of a team when stepping, robot stepping sequentially based on the order of robot id, which means team member with no consecutive robot id would not step one after another exactly, there might be a robot moving between within this team, but it does not belong to this team.

Alternative 2:
Let each robot knows who are his teammates (i.e.: has a List of Robot), this is not a good design, since then robot need to deliver item as well as keep track of team state which leads to not only the low cohesion and it is not extendable, but also Robot has too many responsibilities and make it less cohesive.

Alternative 3:
Create a class RobotTeam which contains Robot as its team member, it is not good since it is not extendable. When a new kind of robot called RobotA now is available which do not support tube, then we need to create a new class for this robot A and modify the Robot team, since previously it contains Robot, but now it needs to contain RobotA as well. However, if we apply the composite pattern here, let Robot and RobotA both implement IRobot, then we do not need to change RobotTeam.

Robot’s state transition from operating individually to working as a team and back to working as an individual:

Based on our sequence diagrams, firstly, from figure 3, when the mailPool is stepping, it returns the individual robot with 2 or fewer light mailItems and a robot team with one heavy mailItem and several light mailItems which just start delivering. When the mailPool is stepping, in figure 4, it makes individual robots to team robot (provide team behaviour) by using the IRobotFactory. The IRobotFactory is responsible for deciding to return individual robot or distribute robots to a team (figure 5). Then both the IRobot starts delivering and IRobot is delivering are going to be stepped in automail. Secondly, when each of IRobot is stepped, no matter it is an individual Robot or a RobotTeam, both two types of IRobot will use its state to step. When a RobotTeam is stepping, if the deliveryItem is delivered, in postDelivery() method, the RobotTeam will return Individual Robots who are continuing the delivering of lightItem in tube previously, as described in figure 6 and figure 7. Above all, this is how we realize the Robot component in the automail system to achieve the robot’s state transition from operating individually to working as a team and back to working as an individual.

Appendix:

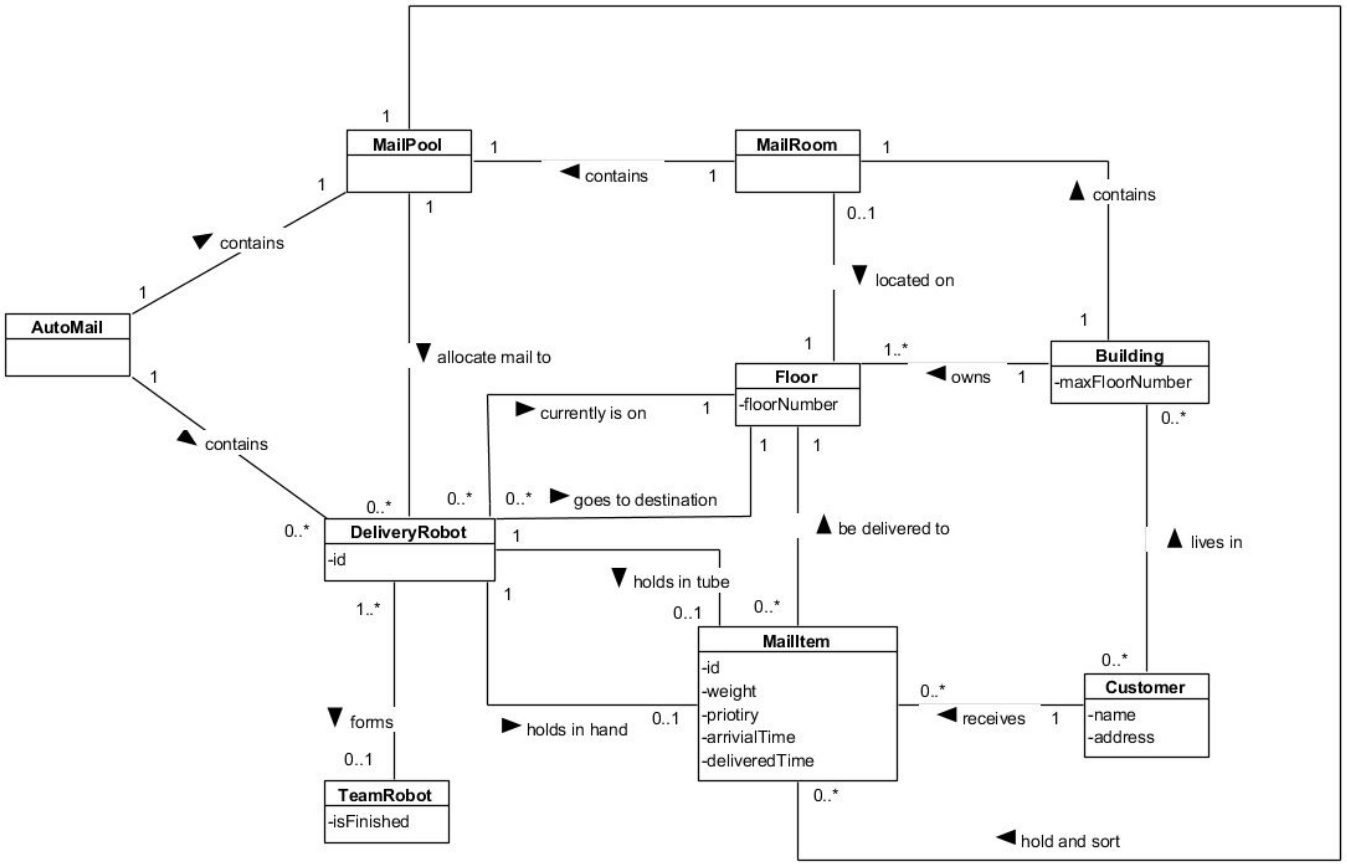

figure 1: Domain Model

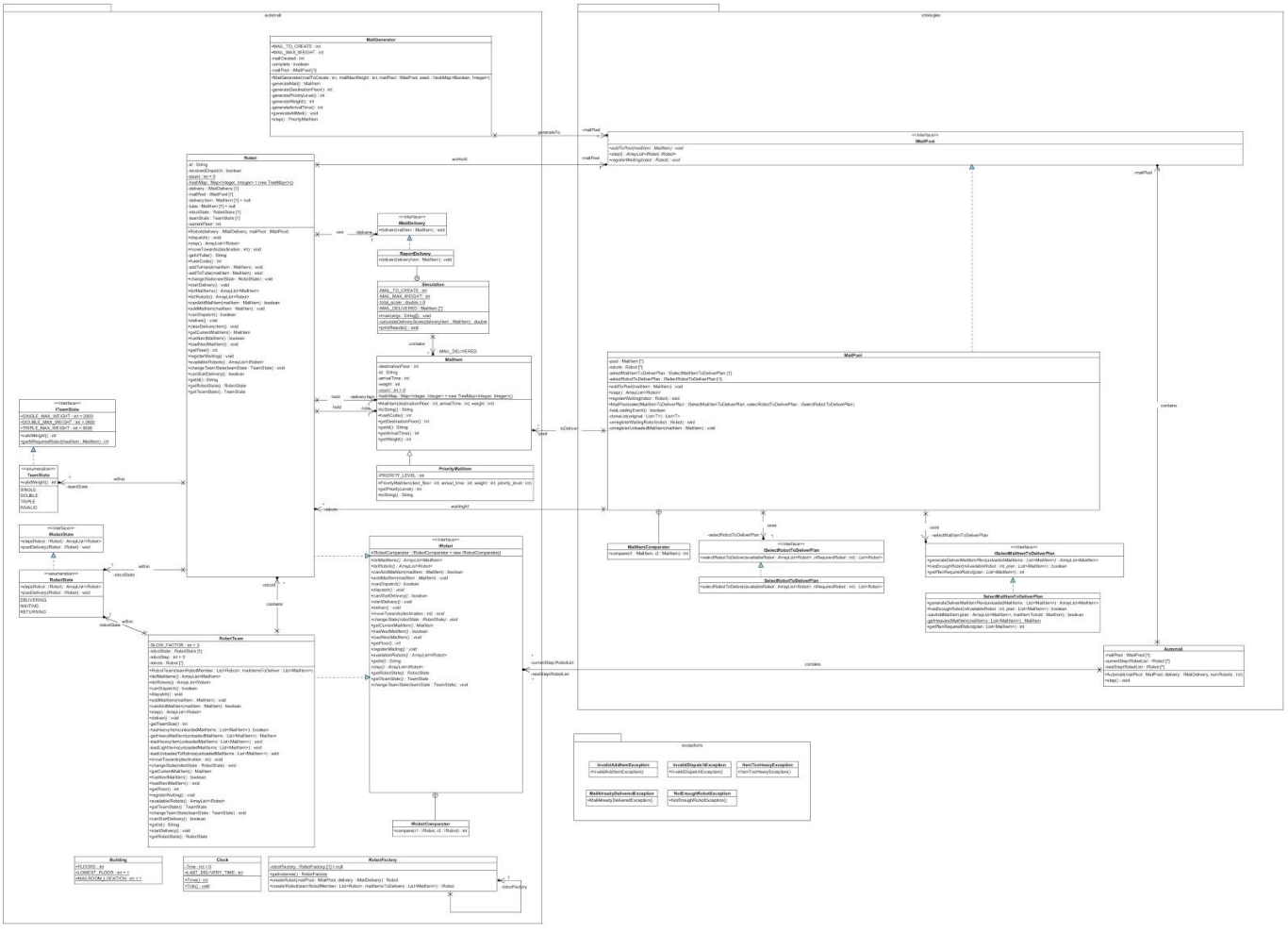

figure 2: design class diagram

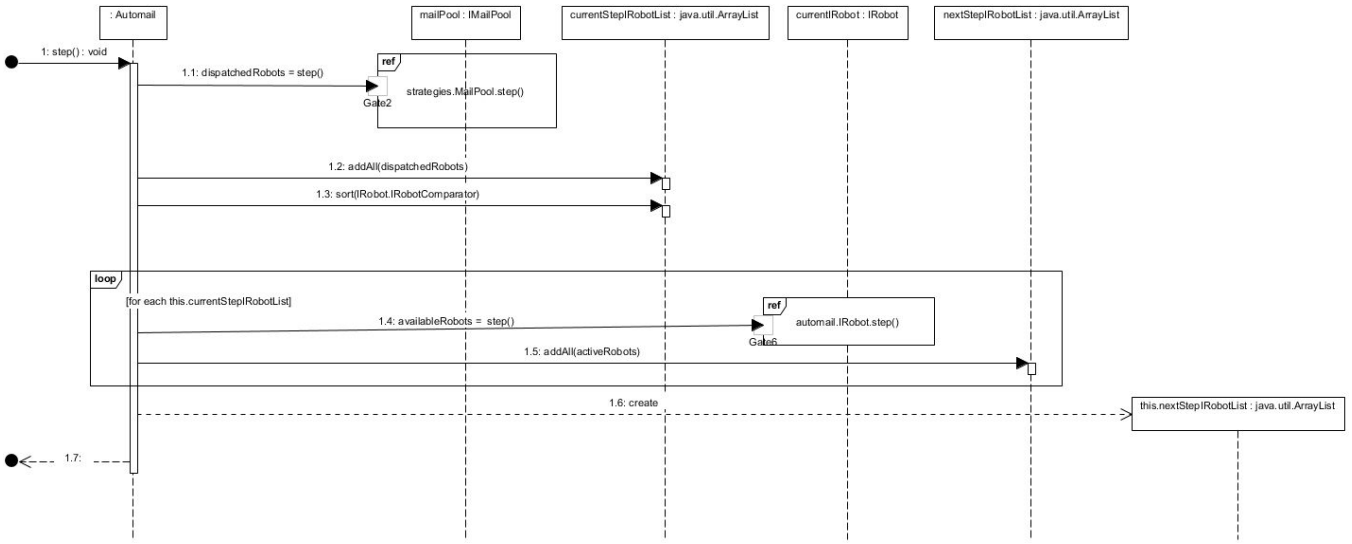

figure 3: automail step

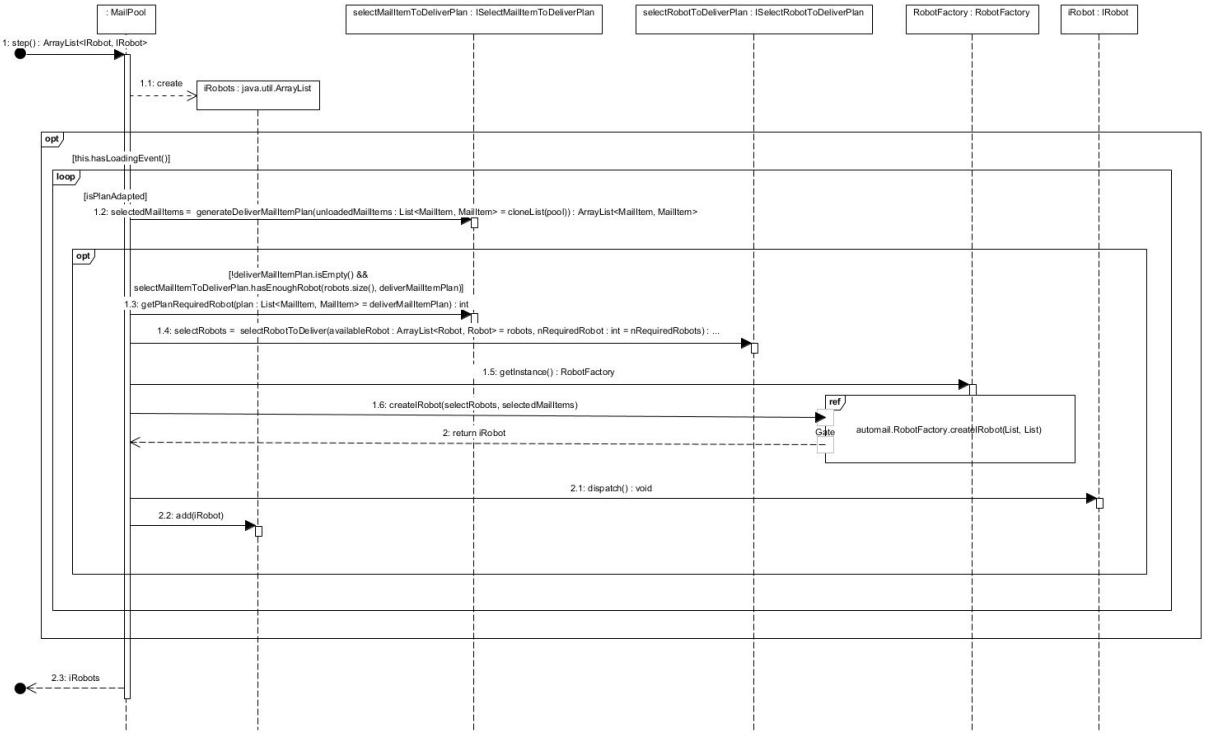

figure 4: mail pool step

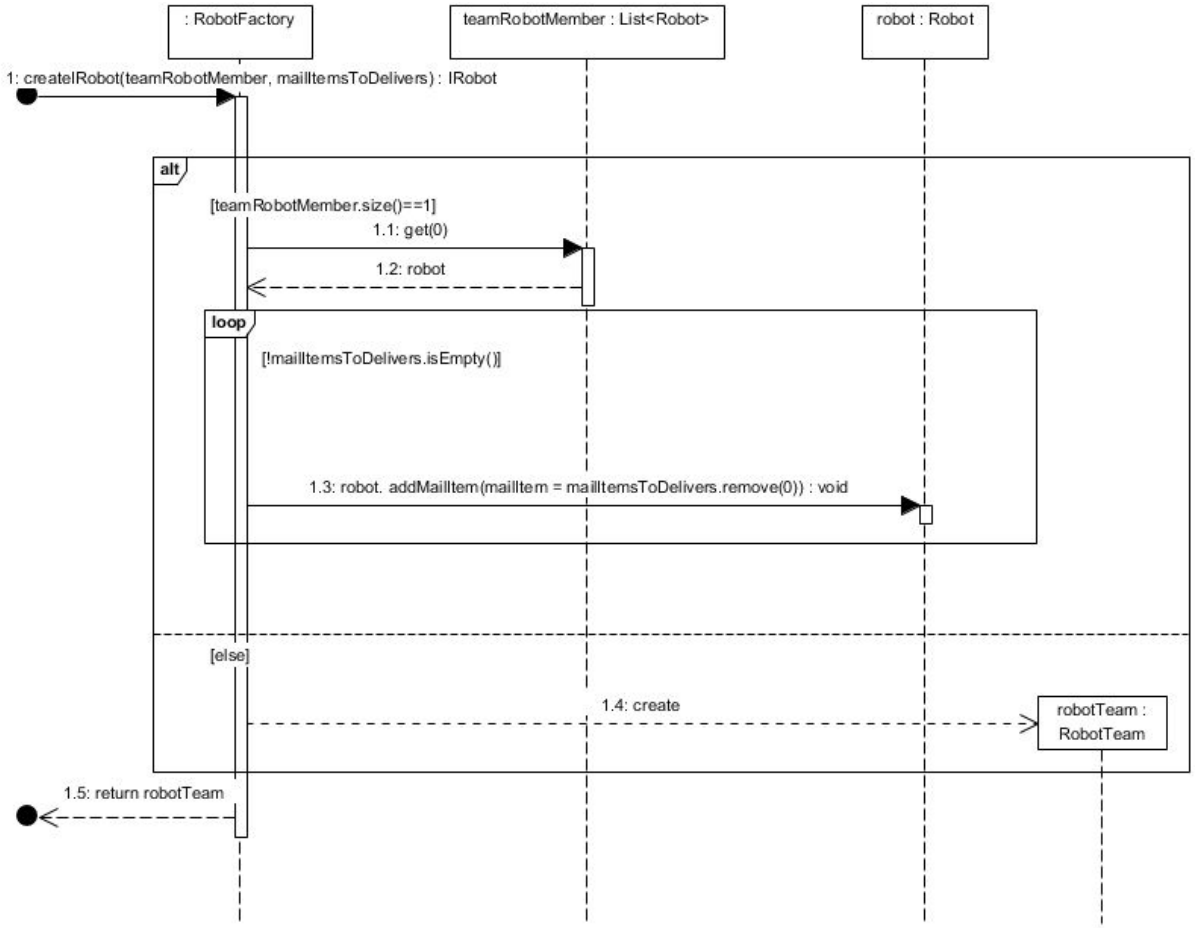

figure 5:robot factory create IRobot

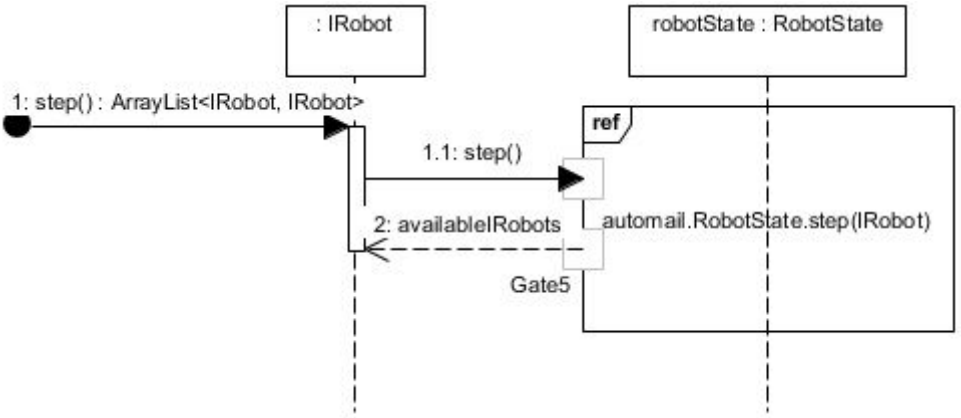

figure 6: IRobot step

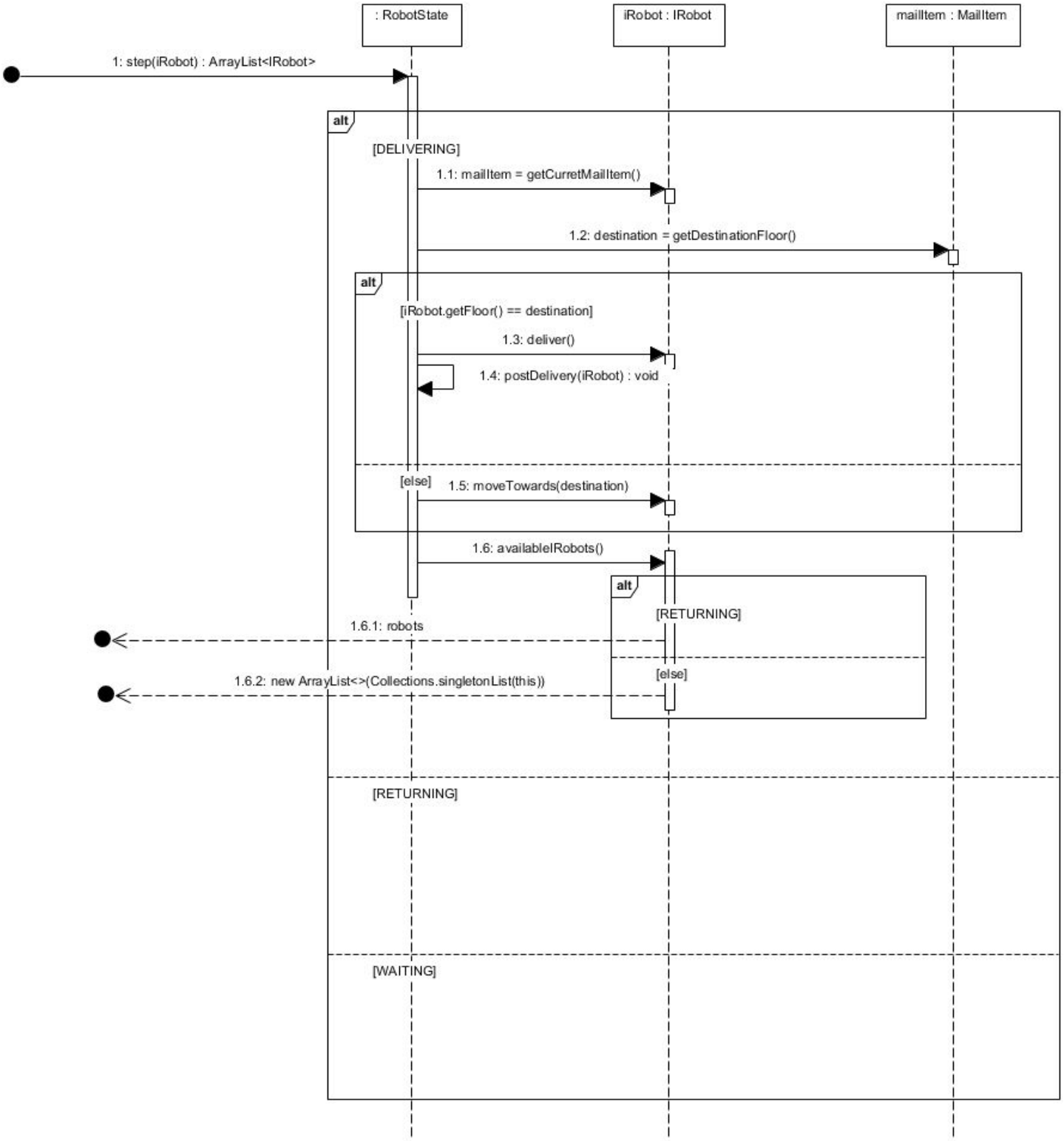

figure 7: IRobot step