SWEN30006 Project 1: Robomail Revision Design Analysis Report[1]

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

Refactoring:

MailPool:

- For *MailPool*, we first change *robots* and *pool* attributes from *LinkedList* to *ArrayList* because we don't need to add sth at the head of the *List*. The *loadRobot* method is also refactored by applying the **Strategy pattern**, as shown in the middle right of figure 2. We decompose *loadRobot* to *ISelectMailItemToDeliverPlan*, *ISelectRobotToDeliverPlan*, which provides higher **cohesion** and more **extensibility**. The responsibility of *MailPool* is limited to only giving *MailItem Robots*, while the planning/loading of MailItem is delegated to these two interfaces. Here, *ISelectMailItemToDeliverPlan* is given the responsibility of selecting *MailItem* for delivery. In the meantime, *ISelectRobotToDeliverPlan* is responsible for allocating robots to deliver a selection of MailItems.
- In addition to the above change, some helper methods in MailPool such as *cloneList*, *unregisterWaitingRobot* and *unregisterUnloadedMailItem* are factored out for better information flow within *MailPool*.
- Private class Item and *ItemComparator* are replaced by an *IMailItemComparator*. We justify this change by considering *Item* class redundant: it only coordinates with the *ItemComparator* to sort *MailItem*. Through providing *IMailItemComparator* to sort MailItem, we enforced the **protected variation** by **lowering coupling** and support **extensibility**. This is also part of the **open/closed principle**. We preserve the original *ItemComparator* behaviour in our *MailItemComparator* which implements the interface.
- Moreover, *step()* now returns *ArrayList* of active *IRobots*, such *IRobots* can then be added to *AutoMail* for further stepping. In exchange for a little coupling, we have now gained the ability to control *active IRobot* in the system and abstracted away from the concept of concrete *Robot* and *RobotTeam* from MailPool.

All the above changes are shown in the static design diagram listed in appendix **figure 2**.

AutoMail:

- *mailPool* and *robots* are made *private* to achieve better data encapsulation.
- The old *robots Array* are modified to two different *Array*s. We will support this change in our further discussion for *step()* in *AutoMail*
- Stepping of *Robot* and *MailPool* are moved from *Simulation* to *AutoMail*. We applied the **Information Expert** pattern here, as *Automail* has all the information necessary to carry out such action. As a result, *Automail* now is delegated the responsibility of running the System while Simulation only focuses on external objects and progressing time. Both the *AutoMail* and *Simulation* are more **cohesive** now!
- Within AutoMail.step(), *MailPool* and *active IRobots* are stepped separately. We make all *step()* return an ArrayList of *IRobots* indicating the active *IRobots* in the next time step. We put these robots in a message queue for processing in next timestep. This design mocks the process of a **Dispatcher pattern** and dramatically improves the robustness of the system.
  An Alternative Design will be giving adding attributes to both *IRobots* and *MailPool*, so they can register *IRobots* to the queue. This Design is utterly horrible for increasing the coupling considerably between those three classes.

All mentioned modifications for *Automail* are illustrated in our static design diagram as well.

MailItem:

- Four attributes: *destination_floor*, *id*, *arrival_time*, *weight* are made from *protected* to *private*. Better **data encapsulation** is achieved through such a design.

Extensibility and modifiability:

ISelectMailItemToDeliverPlan:

- Three *abstract methods*, *generateDeliverMailItemPlan, hasEnoughRobot and getPlanRequiredRobot* require definition to use this interface.
- The first one, *generateDeliverMailItemPlan* takes unloaded *MailItems* and return *List* of *MailItem* to be delivered.

[1] 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.

- Our second interface, *hasEnoughRobot* takes a *List* of *MailItem* and number of *robot* available and outputting true or false based on whether these robots are enough to send MailItems.
- *getPlanRequiredRobot* takes a *List* of MailItem for delivery and return number of *robot* required. We've chosen such operations because we need to tell the *ISelectRobotToDeliverPlan* how much robots needed to be selected for this plan.
- The reason why we use this is that it forms a **strategy pattern** together with *SelectMailItemToDeliverPlan* which is the implementation of this interface can provide a **good extendability** (as mentioned in the change of MailPool class). If there are new ways of delivering required, e.g.: a new kind of 2-item-tube robot become available which could carry 2 mail item in its tube, what can be done is just to provide a new implementation of *ISelectMailItemToDeliverPlan*. As a result, we achieve an **extensible** and **easy modifiable design.**
- Above all, these are the changes of *ISelectMailItemToDeliverPlan* interface and *SelectMailItemToDeliverPlan* class are illustrated in the static design diagram.

ISelectRobotToDeliverPlan:

- The operation, *selectRobotToDeliver*, is responsible for selecting a set of *Robots* for carrying *MailItems* from available *Robots* provided. As it's listed above, such design, together with *ISelectMailItemToDeliverPlan*, form a ***strategy pattern***, providing **more extensibility** as well as **higher cohesion**.
- Such interfaces are also **loosely coupled** with the MailPool. Further modifications, extensions will not require the change of other classes. Both Protected Variation principle in GRASP and Open-close principle is addressed by such design.
- The improvement of **design** can bring is not only limited to the design principle. **Readability** is also a bonus. By hiding the logic behind the scene, the reader can focus on the goal of the program, then being bogged down by implementation details like control statements (if/else). The Self-explanatory method names is also beneficial.
- Without a strategy pattern, all of the implementations will be **coupled tightly** to *MailPool*, increasing the difficulty for future improvement/modification.

Refer to Static Design Diagram relating to MailPool for more information.

RobotFactory:

- This class is created based on **singleton pattern** and **concrete factory pattern,** now this class is responsible for all the creation logic of IRobot, **high cohesion** is achieved by such design.
- 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 the use of the factory has been limited.
- Although the use of the singleton pattern provides a bad control of encapsulation, there is no attribute except the factory. Thus, this disadvantage of the singleton pattern is solved by this design.
- 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.

Composite Pattern:

- Based on the **composite pattern**, *IRobot* and *RobotTeam* are created, as demonstrated 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. Implementation and behavioural details are hidden away as a consequence of the design.
- This also provides low coupling. Because AutoMail only needs to know IRobot instead of both *Robot* and *RobotTeam*, which reduce **coupling** between them relatively.
- Alternatively, applying alternative design 3, the Automail need to check the type of instance first, then implement step, reducing its cohesion.
- Cooperation between *RobotTeams* can be supported in the future making RobotTeam contain *IRobot*. However, we choose to restrict it only to contain *Robot* because if not so it will introduce an unnecessary method to *IRobot* compared with the current design, hence reduce **the coherence** of the design.

IRobot:

We view the robot as a completely unintelligent agent. It has actions necessary to achieve a certain goal, but need external "brain" to order what actions it should take. Based on this objective, we choose to support the following operations:

- The inner class *IRobotComparator* is responsible for sorting *IRobot* by increasing robot ID. This makes our pseudo-async design to sync with the original initial *stepping* order. It's not delegated away to a separate class for it's not likely to be extended in the future.

- *listMailItems* and *listRobots* will return all *MailItems* loaded to current *IRobot* as well as *Robot* member in this *IRobot*.
- The *canAddMailItem* method checks if this *IRobot* could carry a specific MailItem.
- The *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.
- The *canDispatch* method is responsible for checking whether the IRobot could be dispatched or not, and the *dispatch* method is used for dispatching *IRobot*. (*Robot and RobotTeam* provide different implementation to this method, in Robot it only dispatches itself, but in *RobotTeam* dispatches all Robots one by one.)
- The *canStartDelivery* checks if an *IRobot* could delivery. *startDelivery* and *deliver* methods are responsible for delivering and report the MailItem. (*Robot* and *RobotTeam* provide different implementations to this method, in *Robot*, it reports the successful delivery of mail item, but in *RobotTeam*, it chooses only one robot to report the successful delivery, since each mail item is needed to be delivered only once.)
- The *moveTowards* method is responsible for the movement of IRobot. The *changeState* method is responsible for changing the state of IRobot.
- The *getCurrentMailItem* and *getFloor* methods are responsible for returning current mail item and current floor respectively. They are used by *RobotState*, which is discussed below, to provide visibility of the destination floor of this mail item and current floor of *IRobot*. Also removes the use association between IRobot and MailItem.
- The *hasNextMailItem* and *loadNextMailItem* methods are responsible for checking if there is next mail to be delivered and load the next MailItem to be delivered. Currently, only the *Robot* needs this functionality.
- The *registerWaiting* method registers IRobot back to MailPool.
- The *availableIRobots* method returns all available IRobot for next time step. (Robot and RobotTeam provide different implementation to this method, in Robot, it always returns 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 getters are added. They are *getId, getRobotState, getTeamState*.
- The *changeTeamState* method is responsible for modifying *TeamState*.
- The *step* method is responsible for taking the next step for *IRobot*.

RobotTeam:

- The creation of this class is to represent the idea of robots work as a team. When there is one heavy item to be delivered, the robot would form a Robot Team in order to carry this heavy as a team. Compare to the alternative 1 discussed in the alternative design section, below 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 time unit, which is exactly what desired and satisfies the **real world scenario**.

Robot:

- Several changes are made to this class including implementing IRobot interface, which has been discussed in the IRobot class. And, factor out the logics in *step* to the *RobotState*, which has been discussed below. The *destinationFloor* attribute and *setRoute* method are removed because the robot is just an object to do actions required by the system to deliver *MailItem*. Thus the *Robot* just queries the *RobotState* as the guidance to tell *Robot* how to behave in various conditions. 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 evoked by the call from *RobotState*.

RobotState:

- Replace robotState enum in Robot with a RobotState Enum class.
- This class implements *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.
- This enum class **hide the complex logic** of checking which state the IRobot is at. Because inside the step method of the robot, it implements `this.tobotState.Step()`, this would implement the step method according to the current state of IRobot.
- It reduces the chance of the wrong transition, since the transition of state could happen in this RobotState class only. This means other class like Robot, would not need to check state anymore, then there will be no chance of having errors in the transition of states.

TeamState:

- Remove constants, INDIVIDUAL_MAX_WEIGHT, PAIR_MAX_WEIGHT and TRIPLE_MAX_WEIGHT which are originally in the Robot class. Then, create an enum class TeamState, the reason of creating this class is to make it easier to check a robot max carrying weight easier because there is no need to check which team state the robot at.
- This class override two methods *validWeight, getNRequiredRobot* declared in the *ITeamState interface.validWeight* is responsible for returning max carrying weight according to the robot team state.
- *getNRequiredRobot* is responsible for returning the number of robots required to carry a mail item.
- This is also **extensible** for example, if our robot has been upgraded, it could carry more weight now, only the constant inside TeamState needed to change.

ExcessiveDeliveryException: Removed and not used any more.
InvalidAddItemException: Indicates the mail item cannot be added to the IRobot.
InvalidDispatchException: Thrown when dispatch method id called when IRobot cannot be dispatched.
NotEnoughRobotException: Thrown when system configuration **cannot handle** given max MailItem weight.

Alternative Design:

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

Alternative 1:
In this alternative, three new classes are created to address team collaboration of robot, which is TaskGenerator, ITaskGenerator and Task, which is illustrated at the bottom left corner of **figure 8**. TaskGenerator implements ITaskGenerator, which is used in MailPool to generate a Task. Then, it is passed to the robot which is responsible for guiding the robot to the destination floor. Robots have the same Task instance are in the same team.

This design is not logically correct. In reality, robots in the same team carry the same heavy item should move at the exact same time. However, in this system, each robot "step" separately, this means they move sequentially, not move at the exact same time, but it is possible to approach this behaviour by making robots which belong to the same team moving in the exactly consecutive second unit. For example, at x second, team member A move, then 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 *robotId* would not be stepped one after another exactly, there might be a robot which does not belong to this performs an action when all members in team is stepped.

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 bad extensibility, but also the *Robot* has too many responsibilities thus 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:

- Based on our sequence diagrams **figure 3**, when the *MailPool* is *stepping*, it returns the *individual robot* with 2 or fewer light mailItems or a *RobotTeam* with one heavy mailItem and several light mailItems who just starts delivering. When the *MailPool* is stepping requires a team task for multiple robots, it makes *individual robots* to be distributed to *RobotTeam* (to provide team behaviour) by using the RobotFactory. (shown in **figure 4**)
- The *RobotFactory* is responsible for deciding whether to return an individual *Robot* or distribute to *RobotTeam* (**figure 5**).
- Then both the *IRobot* starts delivering and *IRobot* is delivering are going to be stepped in automail's *step*. 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 to destination, in postDelivery() method, the RobotTeam's *step()* will return Individual Robots who are continuing the delivering of lightItem in the tube, as described in **figure 6 and figure 7**. This is how we let robots in a team back to working individually.
- 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: (Disclaimer: http://tiny.cc/y3h45y says Diagram Appendix are not included in report length)

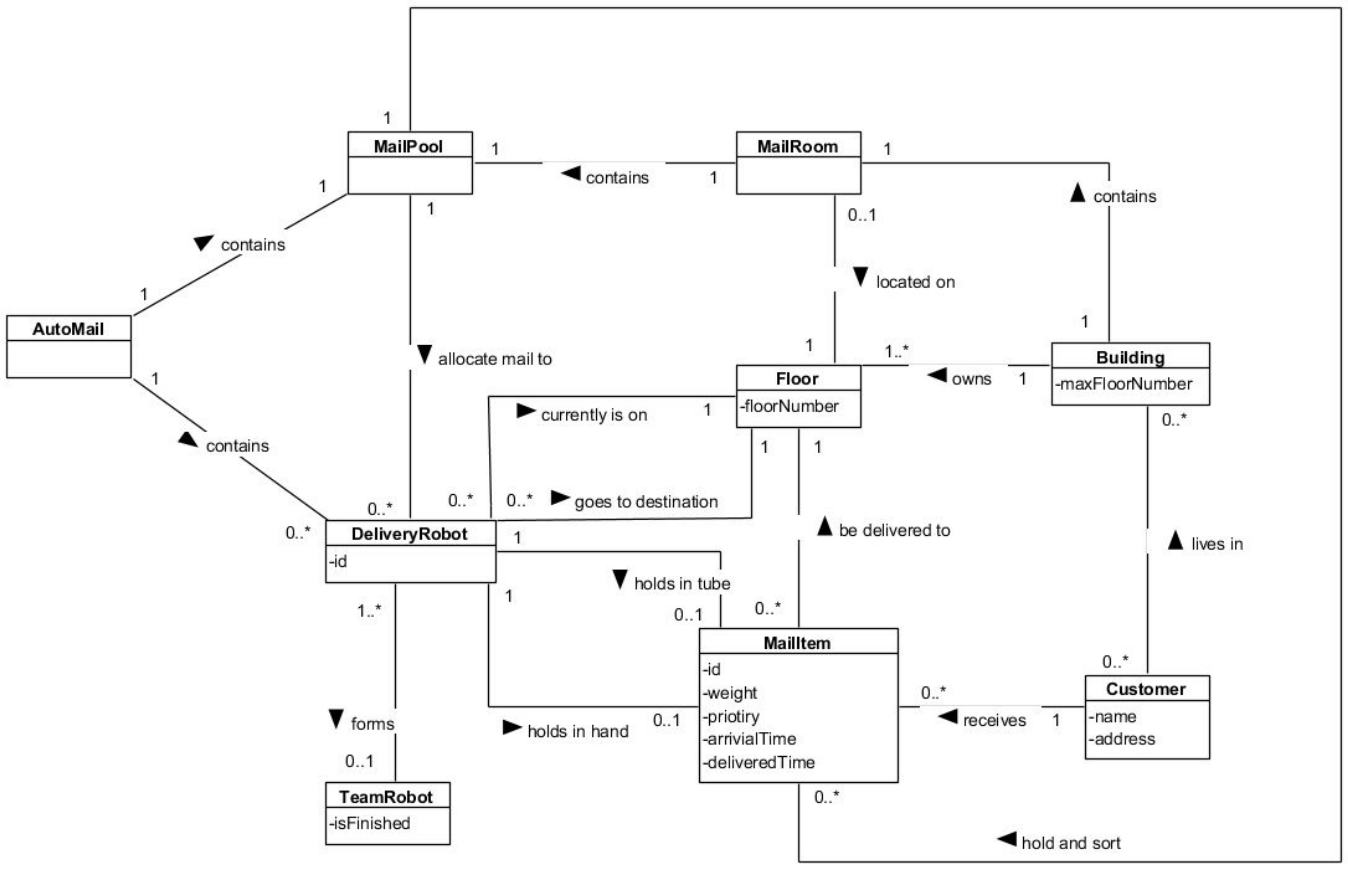

figure 1: Domain Model

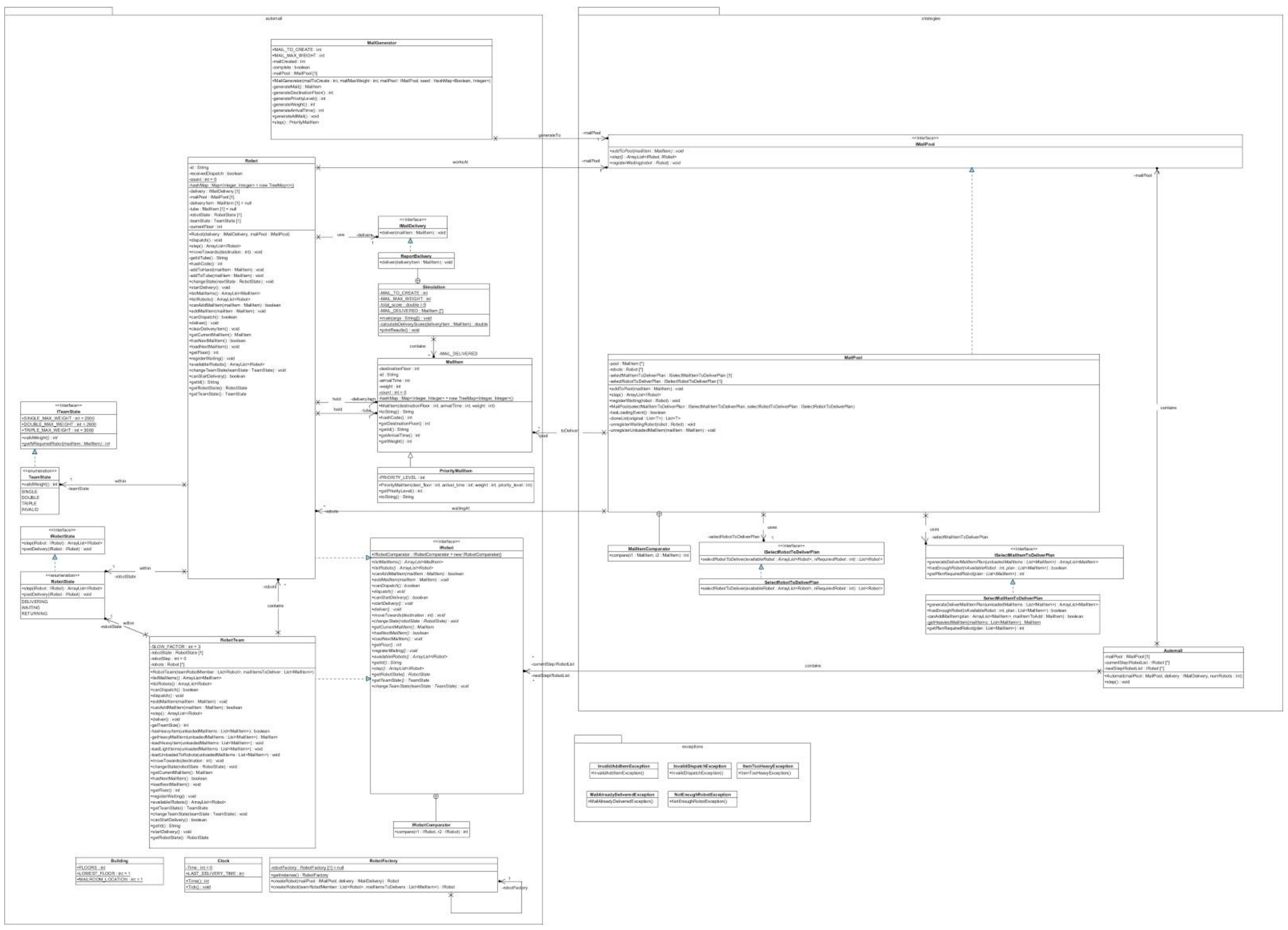

figure 2: design class diagram

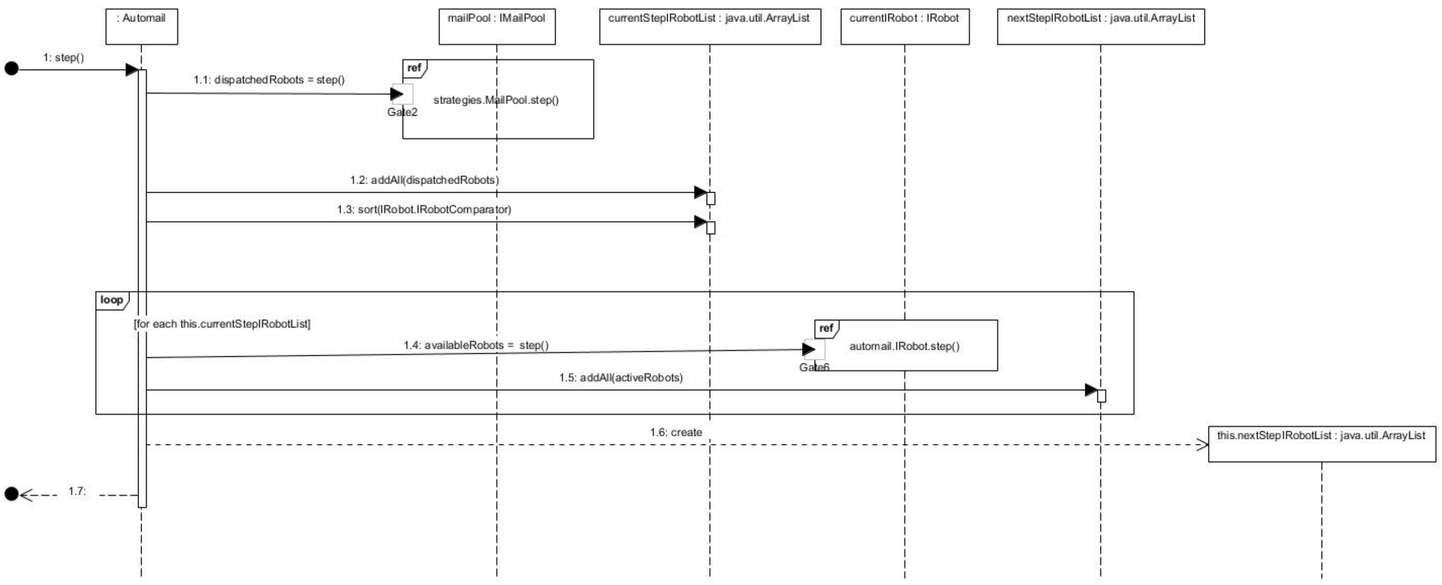
: Automail
mailPool : IMailPool
currentStepIRobotList : java.util.ArrayList
currentIRobot : IRobot
nextStepIRobotList : java.util.ArrayList
1: step()
1.1: dispatchedRobots = step()
ref
Gate2
strategies.MailPool.step()
1.2: addAll(dispatchedRobots)
1.3: sort(IRobot.IRobotComparator)
loop
[for each this.currentStepIRobotList]
1.4: availableRobots = step()
ref
Gate6
automail.IRobot.step()
1.5: addAll(activeRobots)
1.6: create
this.nextStepIRobotList : java.util.ArrayList
1.7:

figure 3: automail step

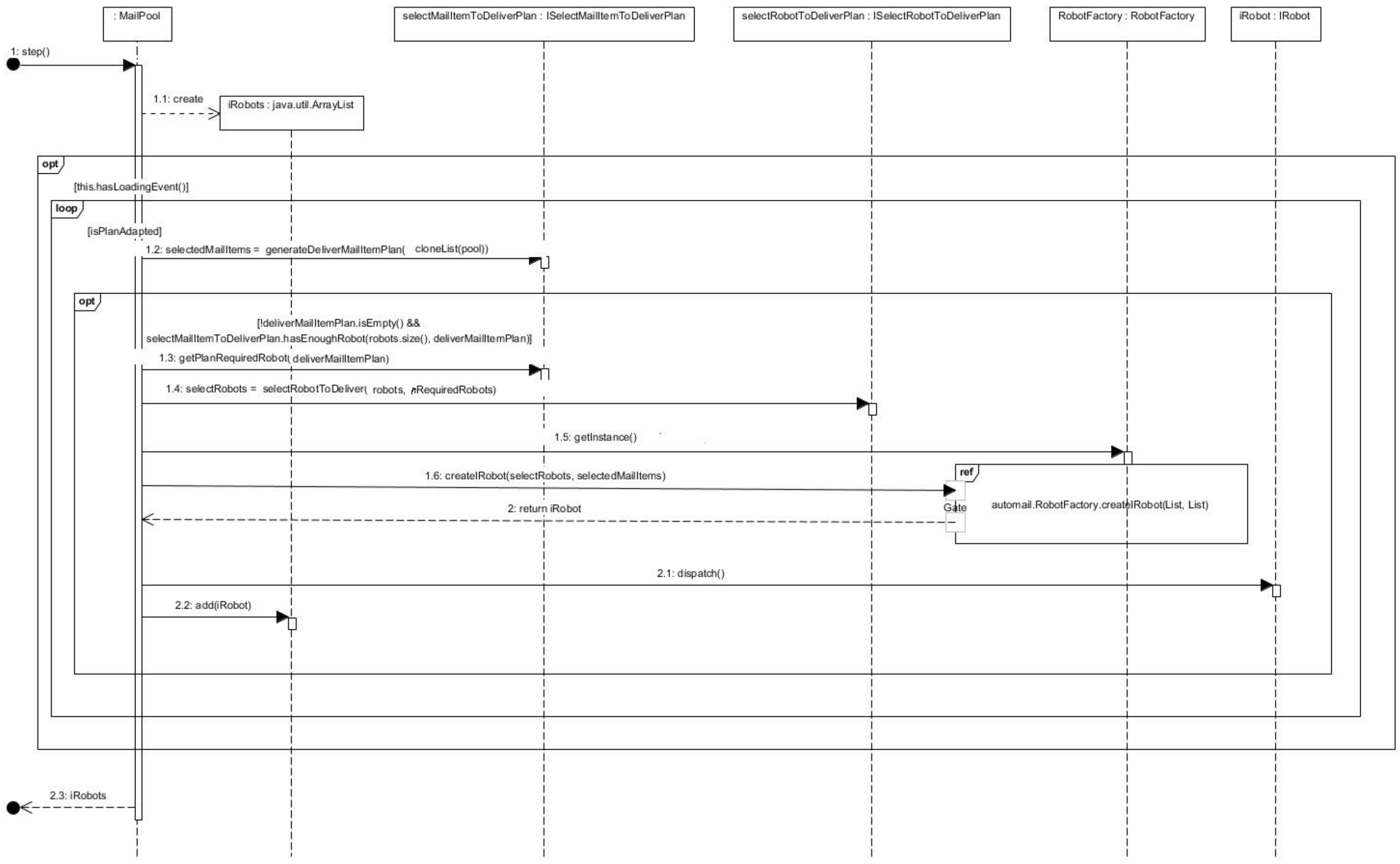
: MailPool
selectMailItemToDeliverPlan : ISelectMailItemToDeliverPlan
selectRobotToDeliverPlan : ISelectRobotToDeliverPlan
RobotFactory : RobotFactory
iRobot : IRobot
1: step()
1.1: create
iRobots : java.util.ArrayList
opt
[this.hasLoadingEvent()]
loop
[isPlanAdapted]
1.2: selectedMailItems = generateDeliverMailItemPlan(cloneList(pool))
opt
[!deliverMailItemPlan.isEmpty() && selectMailItemToDeliverPlan.hasEnoughRobot(robots.size(), deliverMailItemPlan)]
1.3: getPlanRequiredRobot(deliverMailItemPlan)
1.4: selectRobots = selectRobotToDeliver(robots, nRequiredRobots)
1.5: getInstance()
1.6: createIRobot(selectRobots, selectedMailItems)
ref
Gate
automail.RobotFactory.createIRobot(List, List)
2: return iRobot
2.1: dispatch()
2.2: add(iRobot)
2.3: iRobots

figure 4: mail pool step

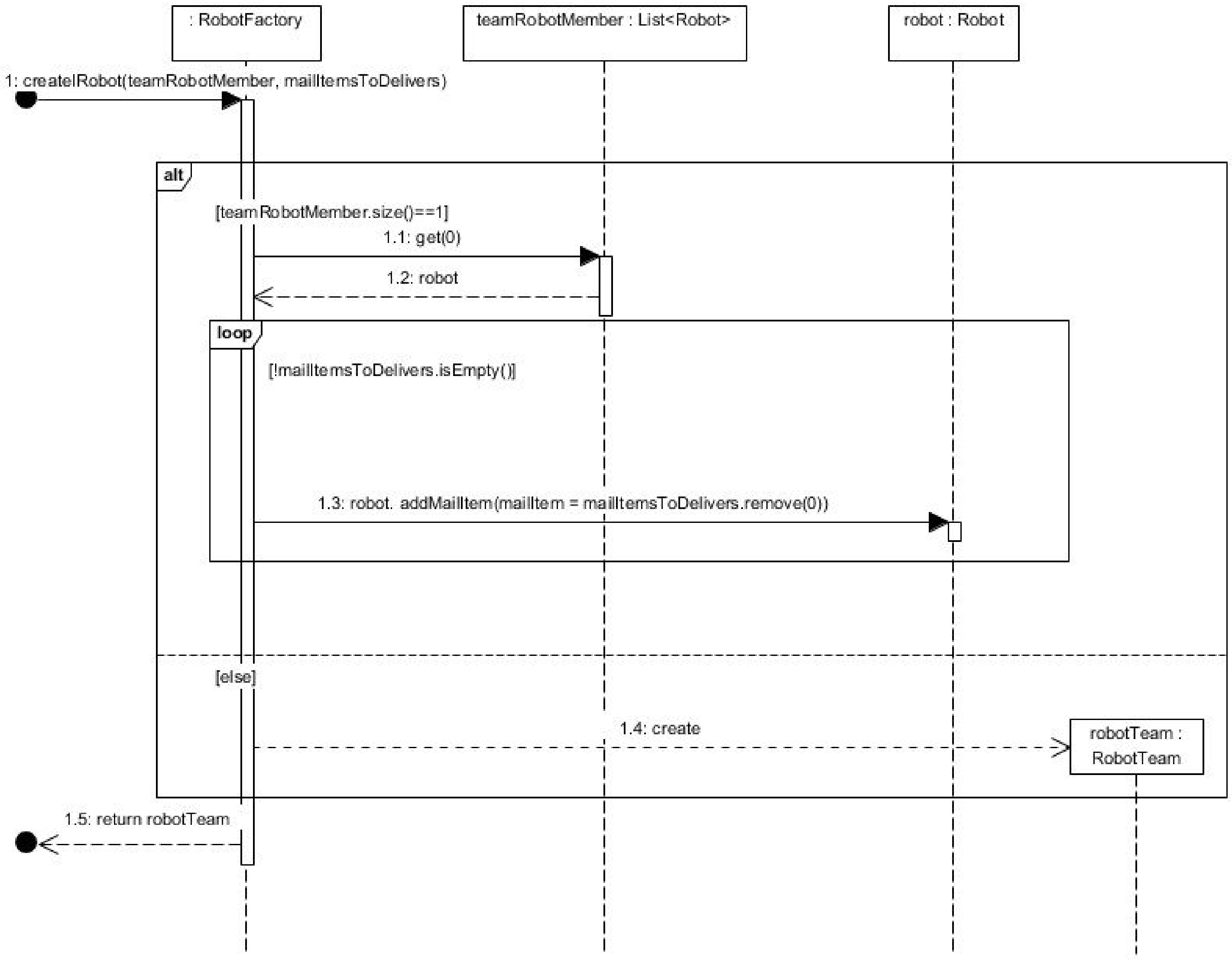
: RobotFactory
teamRobotMember : List<Robot>
robot : Robot
1: createIRobot(teamRobotMember, mailItemsToDelivers)
alt
[teamRobotMember.size()==1]
1.1: get(0)
1.2: robot
loop
[!mailItemsToDelivers.isEmpty()]
1.3: robot. addMailItem(mailItem = mailItemsToDelivers.remove(0))
[else]
1.4: create
robotTeam : RobotTeam
1.5: return robotTeam

figure 5: robot factory create IRobot

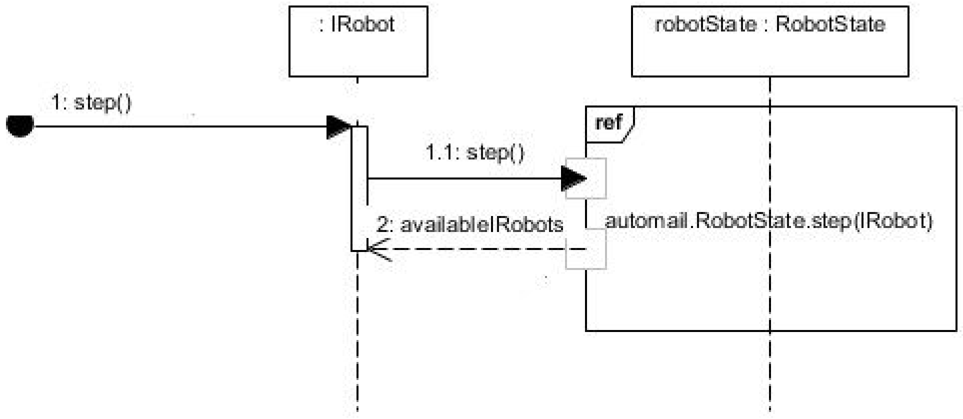

figure 6: IRobot step

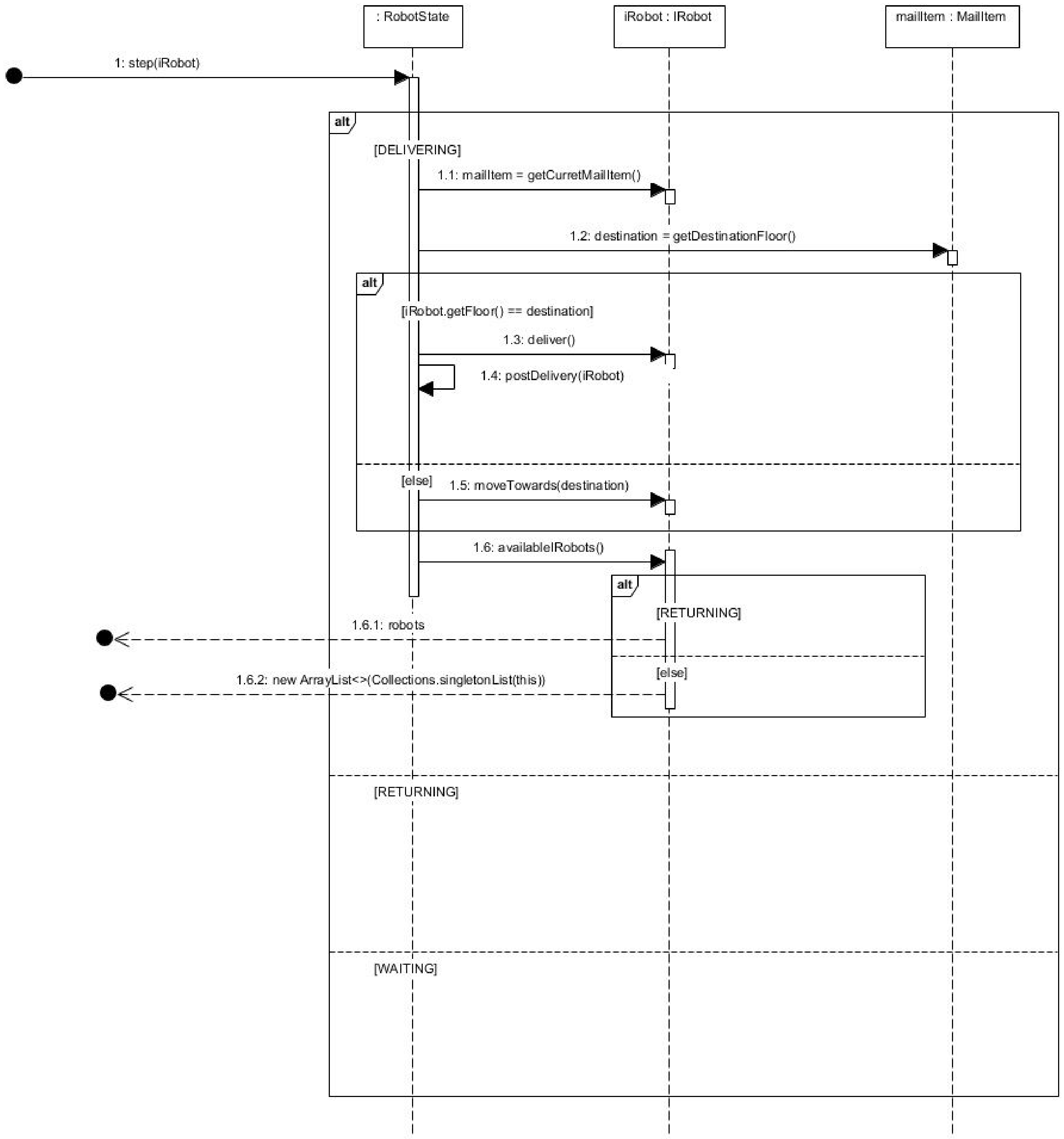
: RobotState
iRobot : IRobot
mailItem : MailItem
1: step(iRobot)
alt
[DELIVERING]
1.1: mailItem = getCurretMailItem()
1.2: destination = getDestinationFloor()
alt
[iRobot.getFloor() == destination]
1.3: deliver()
1.4: postDelivery(iRobot)
[else]
1.5: moveTowards(destination)
1.6: availableIRobots()
alt
[RETURNING]
1.6.1: robots
[else]
1.6.2: new ArrayList<>(Collections.singletonList(this))
[RETURNING]
[WAITING]

figure 7: RobotState step

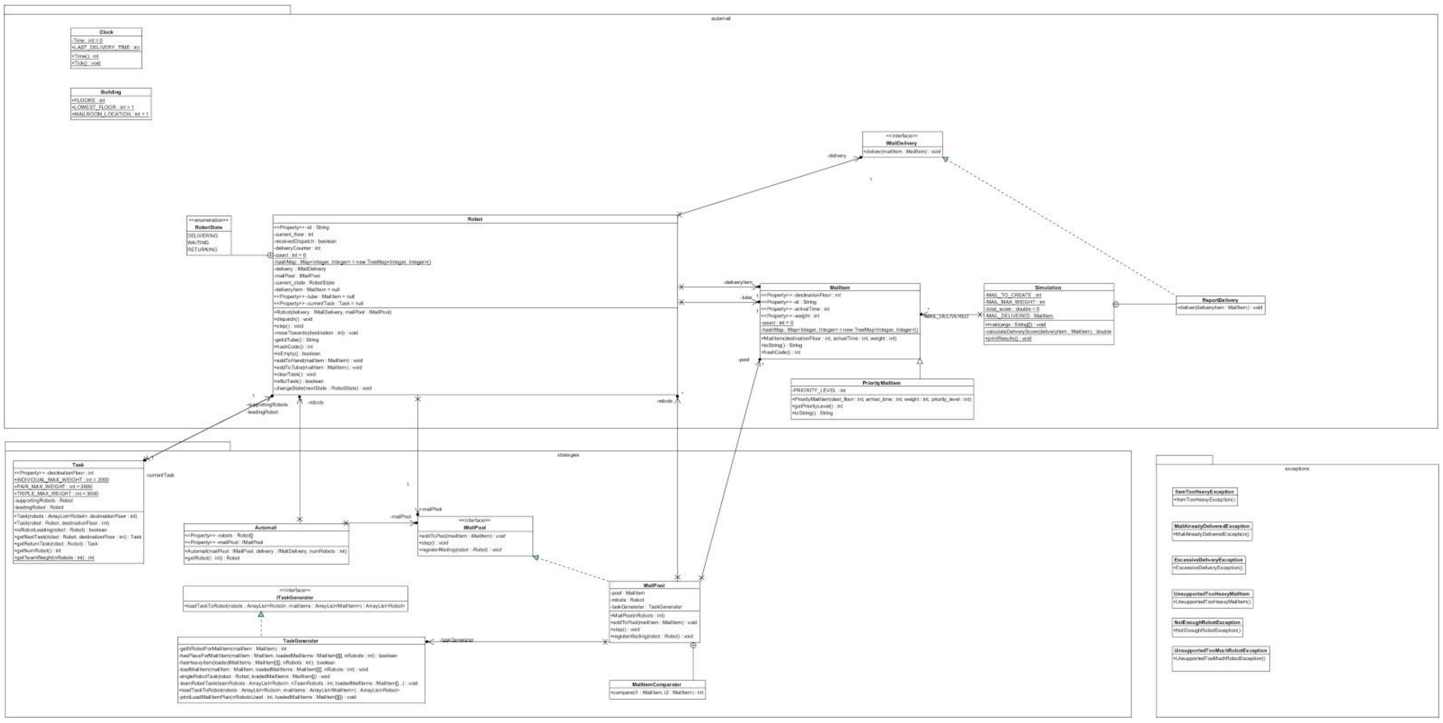

figure 8: unadapted design