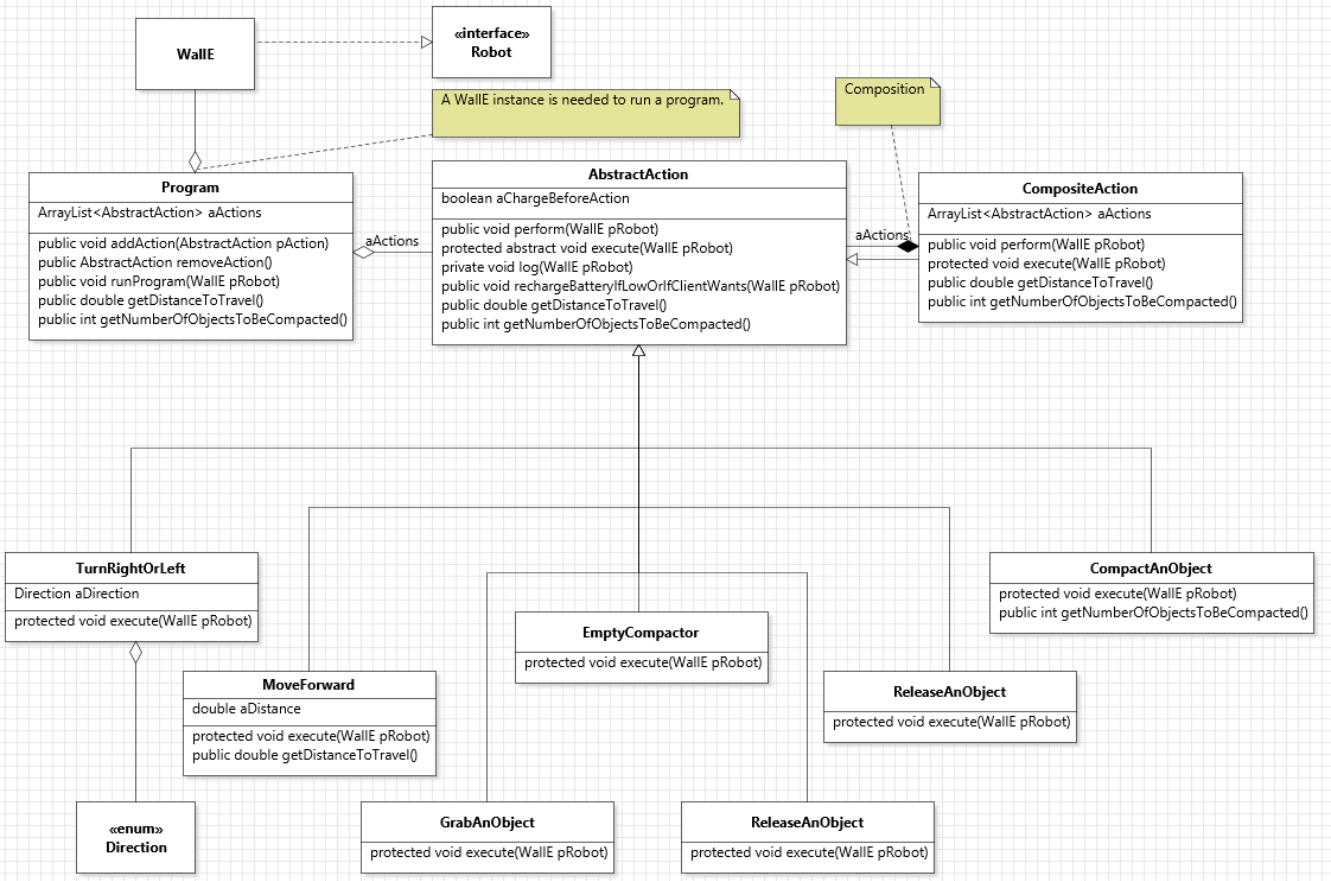
Firstly, I used the template design pattern and inheritance in order to organize all the robot actions. In fact, each basic action has a separate class which extends an abstract class that I called “AbstractAction”. I chose to make this upper class abstract because certain methods such as execute() are specific for the type of action performed and cannot be implementated in the upper class, so I just give the method signature. At the same time, an abstract class helps in reducing code duplication for methods that are the same for all subclasses, such as recharging the battery before performing an action.

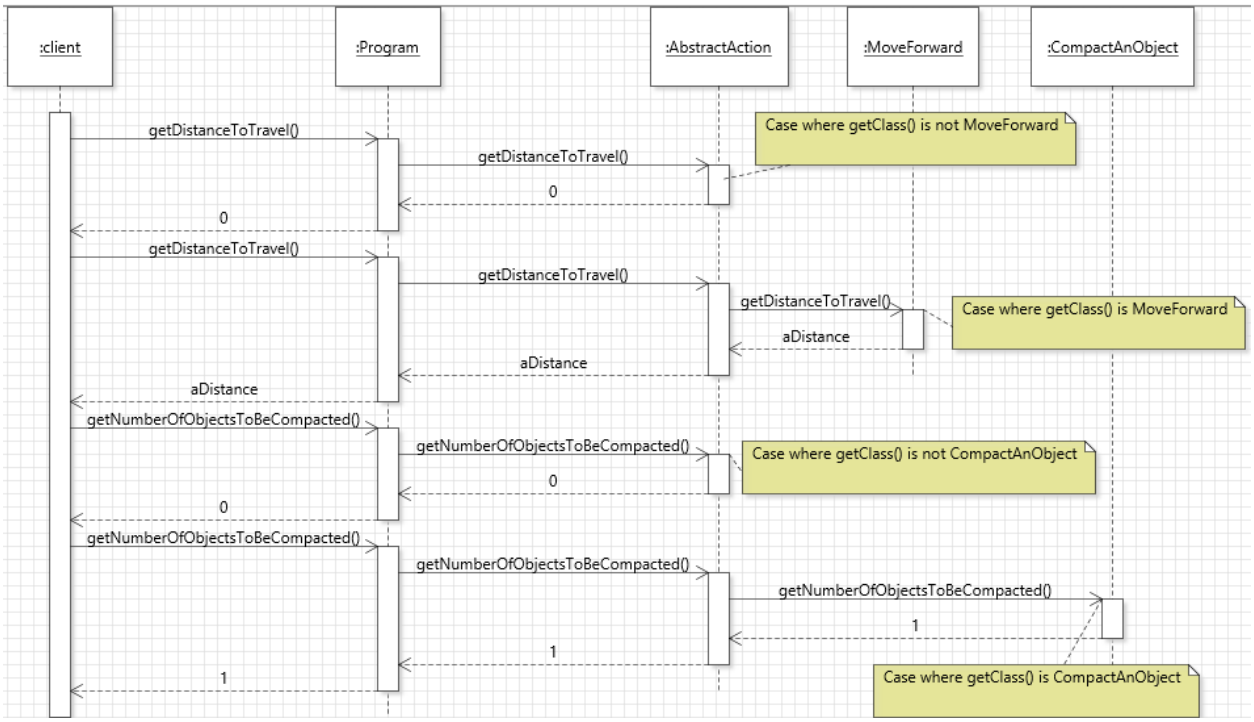
Moreover, I used the composite design pattern in order to allow clients to create complex actions that are composed of basic actions. I created a “CompositeAction” class which extends the “AbstractAction”, but at the same time, it aggregates a list of AbstractAction which are performed consecutively when execute() is called on the CompositeAction instance.

Next, we have the “Program” class which has the field ArrayList<AbstractAction> aActions. After initializing a program, clients can add or remove from that list any “AbstractAction”, which refers to both basic and complex actions. For sure, they have to initialize the action first using the corresponding constructor. Once all the actions desired are in the list, clients can use the method runProgram(pRobot) on a robot to perform all the actions in order. This will also trigger the private log() method in the “AbstractAction” class which prints to the console which action was performed by using the built-in getClass.getName() and shows the battery level as well. The way of logging can easily be changed by only editing this method, if clients prefer to print to a file for example.

On the next page, I give an overview of the whole program that I just described, using a class diagram.

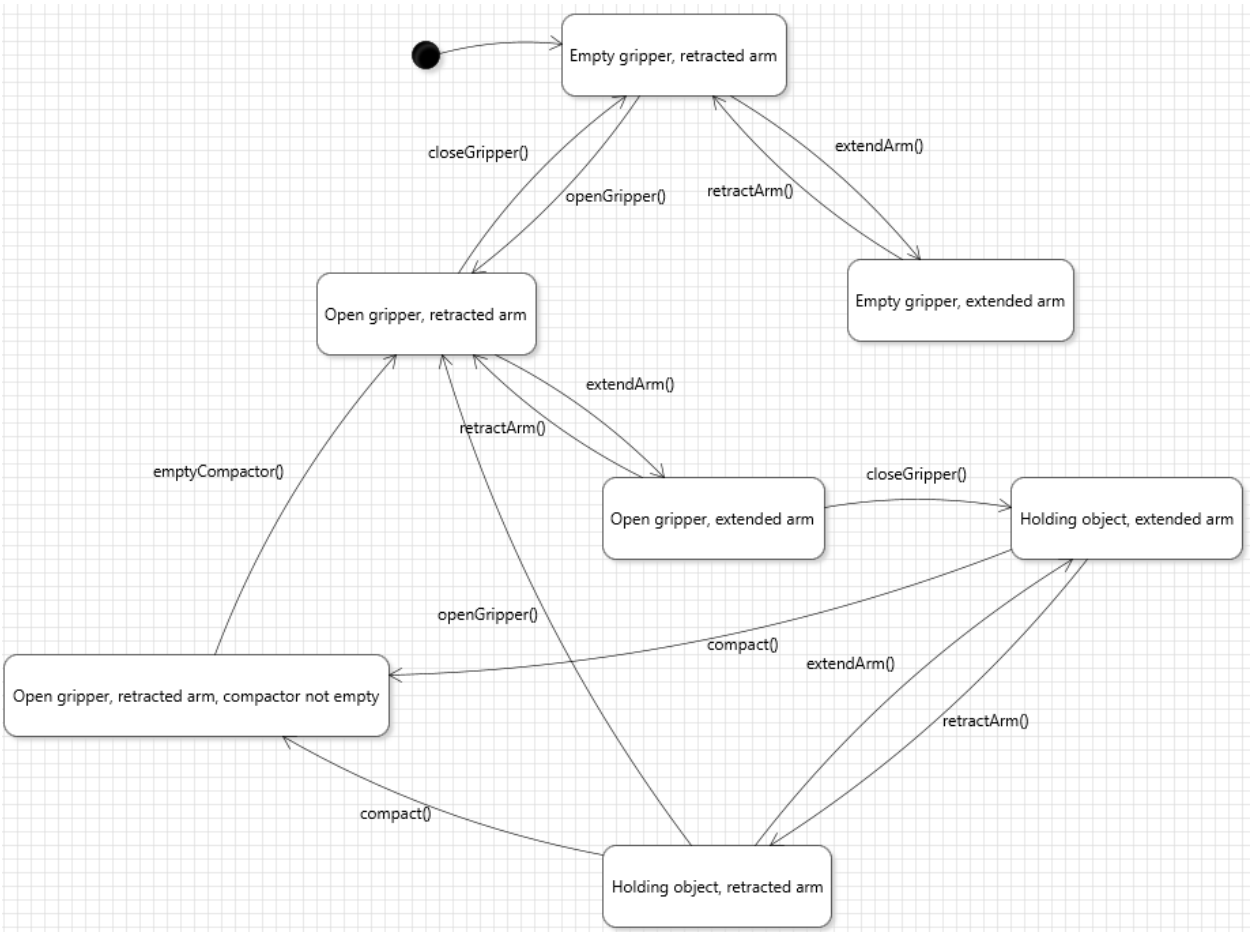


Now, as required by the instructions, a program must offer the possibility for clients to do some useful computations without executing the program. So, in the “Program” class, I added methods that compute the distance to travel and the number of objects to compact based on the actions in the list. In the “AbstractAction” class, I also added two methods that return 0 for both the distance traveled and the number of objects to be compacted. These will be overriden only by the concerned subclass. For example, MoveForward overrides getDistanceToTravel() and returns the field aDistance instead of 0 and the “CompositeAction” class loops through all the actions and sums all the distances instead of returning 0. The same idea applies to the number of objects to be compacted. On the next page, I show a sequence diagram representing how a computation works.



Furthermore, the “WallE” class gives some preconditions that must be taken into account before calling certain methods on a robot instance. Now, since they raise an exception if they are violated and at the same time, it is hard for clients to keep track of (e.g. we must always retract the arm before moving the robot), I designed the basic actions in a way that they automatically change the state of the robot to not violate the preconditions when methods from the “Robot” interface are called. For example, if a robot’s arm is extended and the client wants the robot to move, then before performing this action, I call the retractArm() method. This makes the software more user-friendly instead of raising an exception every time.

On the next page, I show a state diagram containing all the states of a robot and what action can be performed based on that state. That diagram helped me in designing the software in a way that it does not raise an exception.



On a final note, I created unit tests to thoroughly test every aspect of my software. In the tests, I tried to cover all the branches that are present in the code. As a simple example, I tried moving the robot when the arm is extended and when the arm is retracted to see if the program behaves accordingly.