Final Project - Pixar

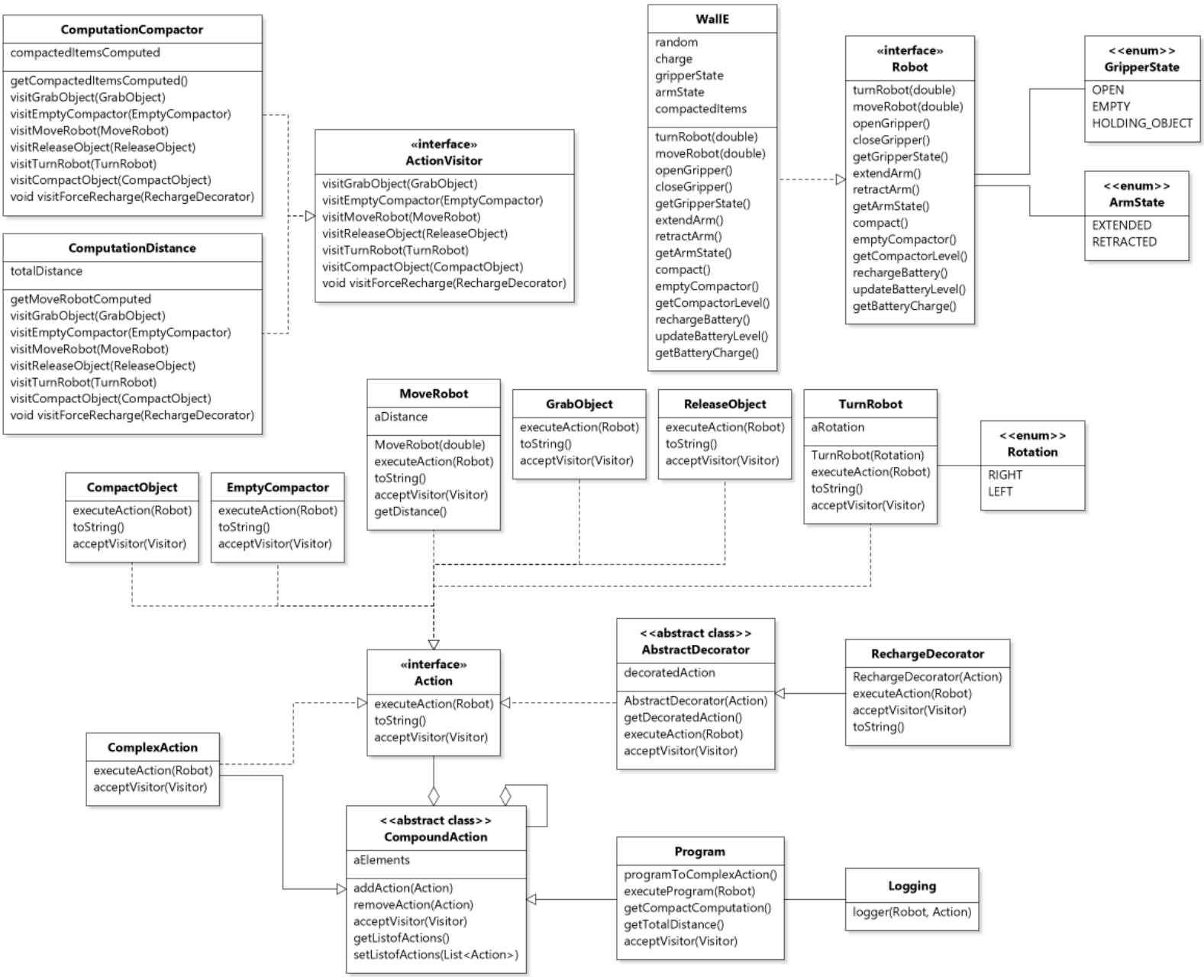
1. Action interface and Program class which (indirectly) has a list of Actions. This list is what is used to load actions onto the program for later execution. Action interface has boolean executeAction() for questions 1-7, and acceptVisitor() for question 8. The Program class has executeProgram(), and other methods.
2. The following concrete classes are created for basic actions. Each of these classes implements the Action interface:
    1. MoveRobot
    2. TurnRobot
    3. GrabObject
    4. ReleaseObject
    5. CompactObject
    6. EmptyCompactor
3. Since there are 3 things to do every time a basic action is going to be executed, within the Program class, while running a for loop through the list of Actions it has, it checks for those conditions.
4. Now a ComplexAction class is introduced but it is fairly similar in structure to the Program class, with the main difference being that the ComplexAction is an Action. Hence, both Program and ComplexAction extend an abstract CompoundAction class that holds the similarities. The ComplexAction class additionally implements Action. This is the Composite Design Pattern.
5. Using the Decorator Design Pattern, abstract AbstractDecorator class is created, which implements Action and holds an Action attribute. Then I implemented RechargeDecorator as a concrete decorator class that extends AbstractDecorator and also implements Action. In this concrete class, the executeAction() is done by first recharging the battery and then calling the original execute method on the Action attribute.
6. The abstract CompoundAction class has methods to add and remove Actions from its list of Actions. Hence, Program that extends that abstract class also can add and remove Actions, serving an editing purpose. To start the program, the executeProgram() method is called in the client code. As an additional functionality, the ComplexAction class is also editable in exactly the same way as Program (you can not only sequence actions, but also remove actions).
7. All basic actions are written in a way such that if a precondition is not met,
    1. The unmet condition is fixed simply if possible. For example, for the MoveRobot basic action, if the arm is extended, MoveRobot cannot be executed. However, we can simply retract the arm and then execute MoveRobot.
    2. If the unmet condition is not fixable, then the action is not done and the boolean executeAction() for that basic action returns false. This is later used in the ComplexAction or Program class while iterating through the list of Actions. While iterating, if any basic action returns false, then that action is skipped over for updating the battery level or logging.
8. Using the Visitor Design Pattern, I created the interface Visitor that declares methods to visit every existing concrete Action class. Thus the Action interface has acceptVisitor(), that will let the Visitor visit them. The acceptVisitor() method in ComplexAction or Program classes call acceptVisitor() on each Action that they consist of.

There are also concrete classes implemented like ComputationCompactor and ComputationDistance that each implement the required visit methods to update their attributes (int compactedItemsComputed and double totalDistance, respectively). These values are returned to client via simple methods in the Program class.

9. Override the toString() for all the basic actions and decorated actions. Then created a Logging class with a single method logger() that takes in an Action and a Robot. Using the Robots' objects address for uniqueness it creates a text file (or appends to it if it already exists) and logs the action executed and remaining battery charge of the robot. I chose this over including a log method in the interface itself to avoid unnecessary coupling. Additionally, this way allows other log methods in the existing Logger class to be added for different methods of logging.
   In the Program and ComplexAction class, the logger() method is called whenever an action is executed successfully.
10. Tests were extensively used for the entire project to ensure robustness and confidence in the functionalities. Every basic action class was first tested individually, with any other methods executed via stubs. Reflection was used often for white box assertations. Black box testing was also used as an initial failsafe. After testing the basic actions, the other classes were tested with fewer stubs, because the basic actions were already tested. The final coverage was 95% of the classes and 71% of the lines and 81% of the methods. This includes statistics for the Driver class, for which no tests were done.

Siddharth Raghavan
260740094

11. Diagrams
    a. Class diagram



Siddharth Raghavan
260740094

b. State diagram for operations

Siddharth Raghavan
260740094

c. Sequence diagram for computation