1. Action interface and Program class which implements the Action interface

2. Implementing the Action interface, the described concrete classes are created.
   - Move, Turn, Grab, Release, Compact, Empty

3.
   1. The conditionCheck method for the Program and the ComplexAction is used to check battery level, they ensure that this check will take in place in any case, since the client will not be executing basic actions themselves.
   2. The execute method calls conditionCheck and if it is not satisfied, it recharges the battery
   3. Execute then attempt to execute the individual actions and updates the battery level for each successful action

4. CompositeAction abstract class consists of a list of actions and has methods to add and remove actions from the said list. ComplexAction class implementing Action, overrides Action methods such that the method for each action is called. Battery check and update happens here, too.

5. Using the Decorator Design Pattern, abstract ActionDecorator class is created, which implements Action and holds an Action attribute. RechargeDecorator, concrete decorator class extends ActionDecorator and implements execute by first recharging the battery and then calling the original execute method on the Action attribute.

6. As started in parts 3 and 4, Program class now has methods to add and remove Actions from a Program. Remove attempts to remove all given Actions and returns false if >=1 was not in the Program to begin with, if found it removes the first matching Action, in case of duplicates.

7. All basic actions are written in a way that if a precondition is not met there are two options, if the unmet condition can be fixed simply (e.g. gripper is empty instead of open), then it is fixed and then the action is executed, if the unmet condition requires another basic action to fix (e.g. trying to execute grab when gripper is holding item) then the action is simply not executed. In the case of a complex action (Program) the second case breaks the sequence, and no following actions are performed.

8. Using the Visitor Design Pattern, I created the interface ActionVisitor that declares methods to visit every existing concrete Action class. To accommodate this, I also added a method (accept) in the Action interface that will let the Visitor visit them. Accept in CompositeAction classes (ComplexAction and Program) call visit on each Action that they consist of.
   The concrete classes implementing ActionVisitor, DistanceVisitor and CompactorVisitor, implement the visit methods to update their attributes (double distance and int

compacted, respectively) these values are returned to client via simple methods in in the Program class, which accept the visitors and return their attribute values.

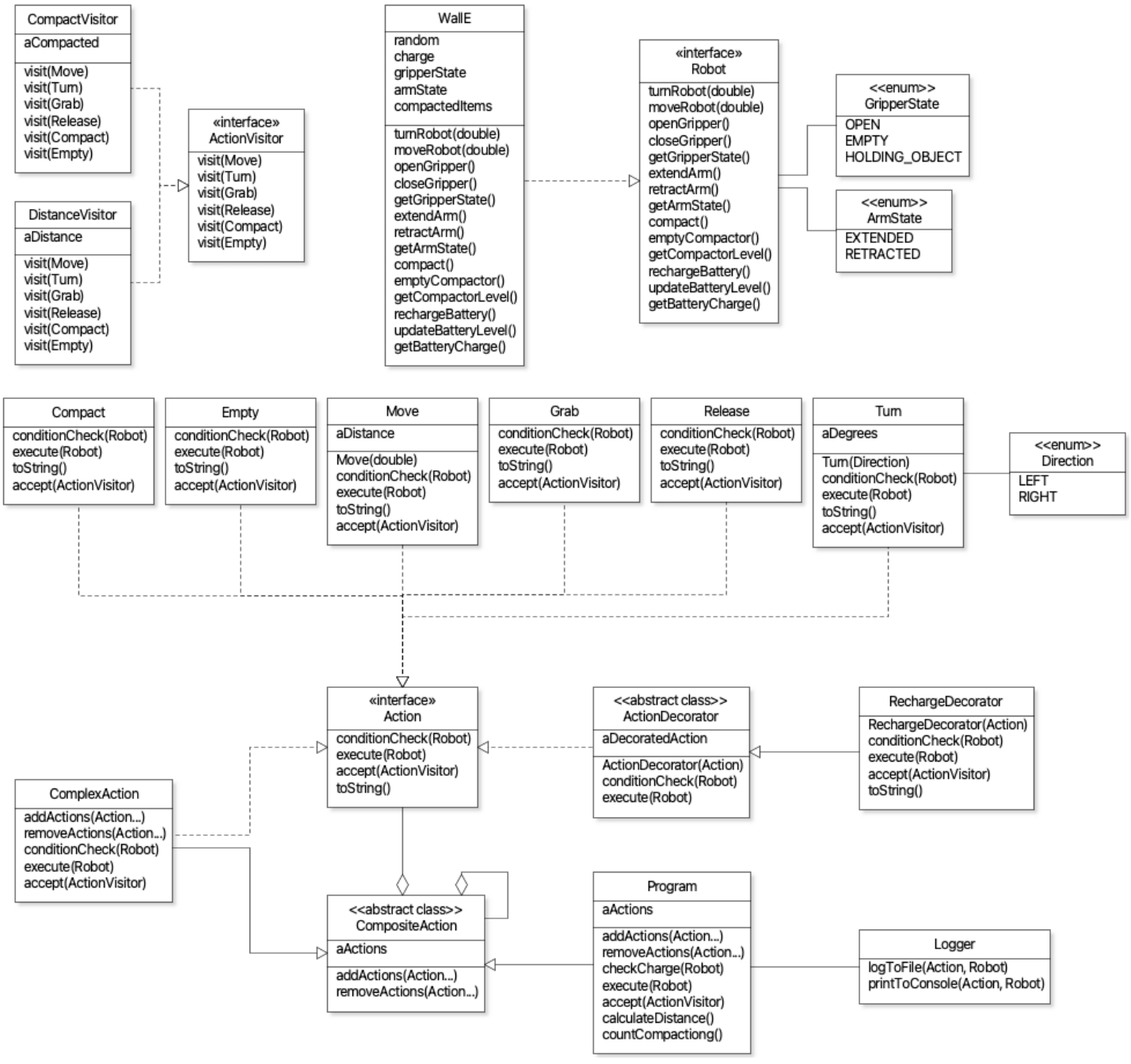
9. Added override to Object toString to the Action interface to enforce all concrete actions implement this method, then created a Logger class with logging methods logToFile and printToConsole that both receive a Robot and an Action. Th eprimarily used method logToFile, creates a text file (or appends to it if it already exists) using the robots' object address for uniqueness and logs the action executed and remaining battery charge of the robot. I choose this over including a log method in the interface itself to avoid unnecessary coupling and this way other methods can be added existing Logger class can be added for different methods of logging, like the printToConsole method I have included. In the Program Class, Logger.logToFile is called whenever an action is executed successfully.

10. Created individual test classes for each written class. Used black- and white-box testing, using stubs and reflection.
    - 43 successful tests:
    - 95% class coverage: all classes except driver is covered
    - 81% method coverage: concrete visitor classes have low method coverage since they have methods implemented from the visitor interface that are not useful for their specific goal. Also, I have an alternative log method for which I could not write tests for, since it simply prints on the console.
    - 78% line coverage: adding on top of unused methods there are clauses in WallE that are not reachable using the basic actions we have (e.g. Arm will never be retracted when attempting to close gripper)

11. Diagrams

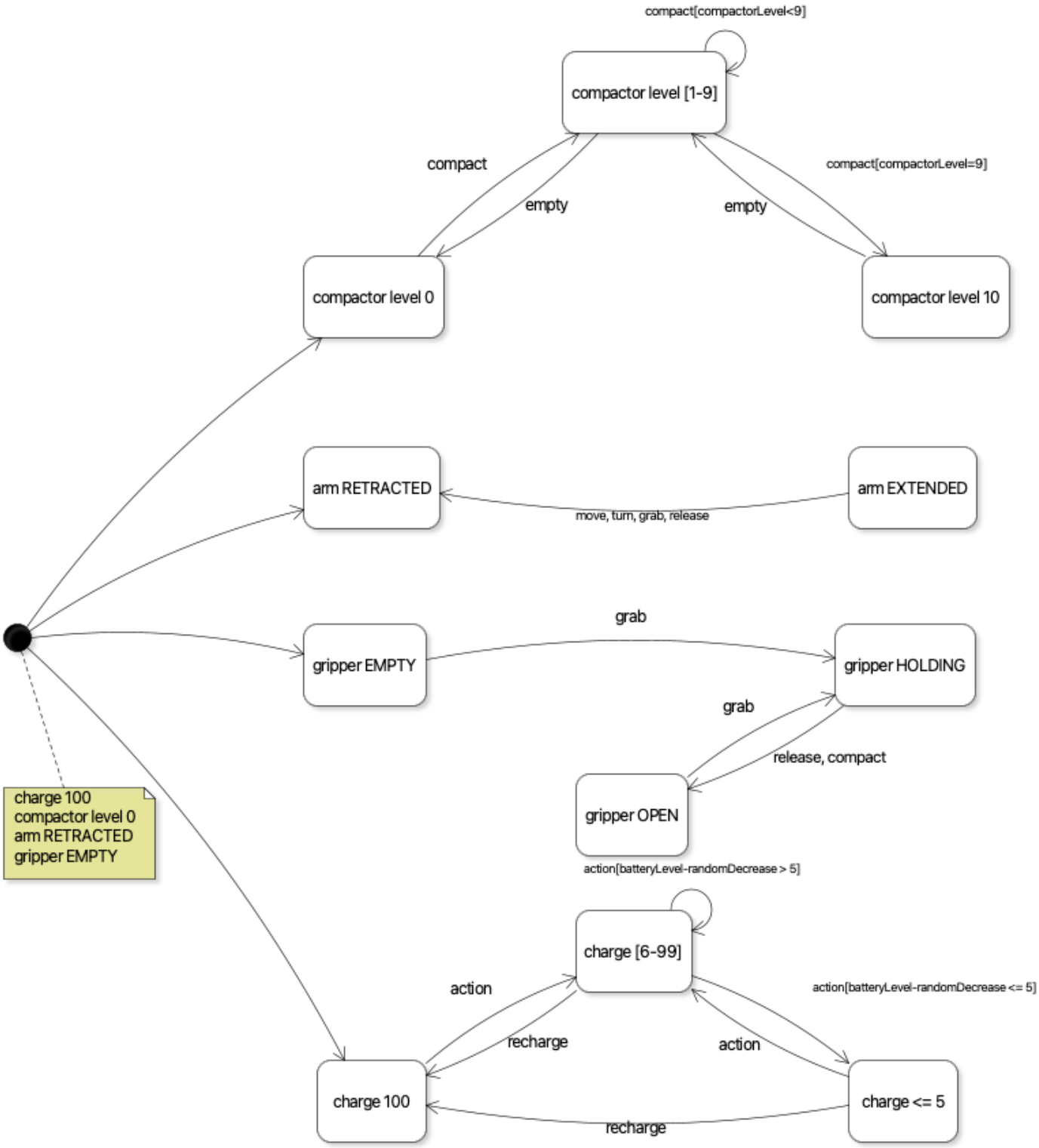
a. Class diagram

I did not show all associating lines between separate groups for clarity of the graphic.



b. State diagram for operations

I kept compactor level, charge, arm- and gripper states separate from each other for clarity. This especially made sense since, when operations effect multiple of these states, the changes made are not dependant on one another.



c. Sequence diagram for computation
Assuming that in client class we have a Program that consists of two basic actions, one Move and one Turn action (in that order). The diagram shows the events for when the client calls aProgram.calculateDistance().