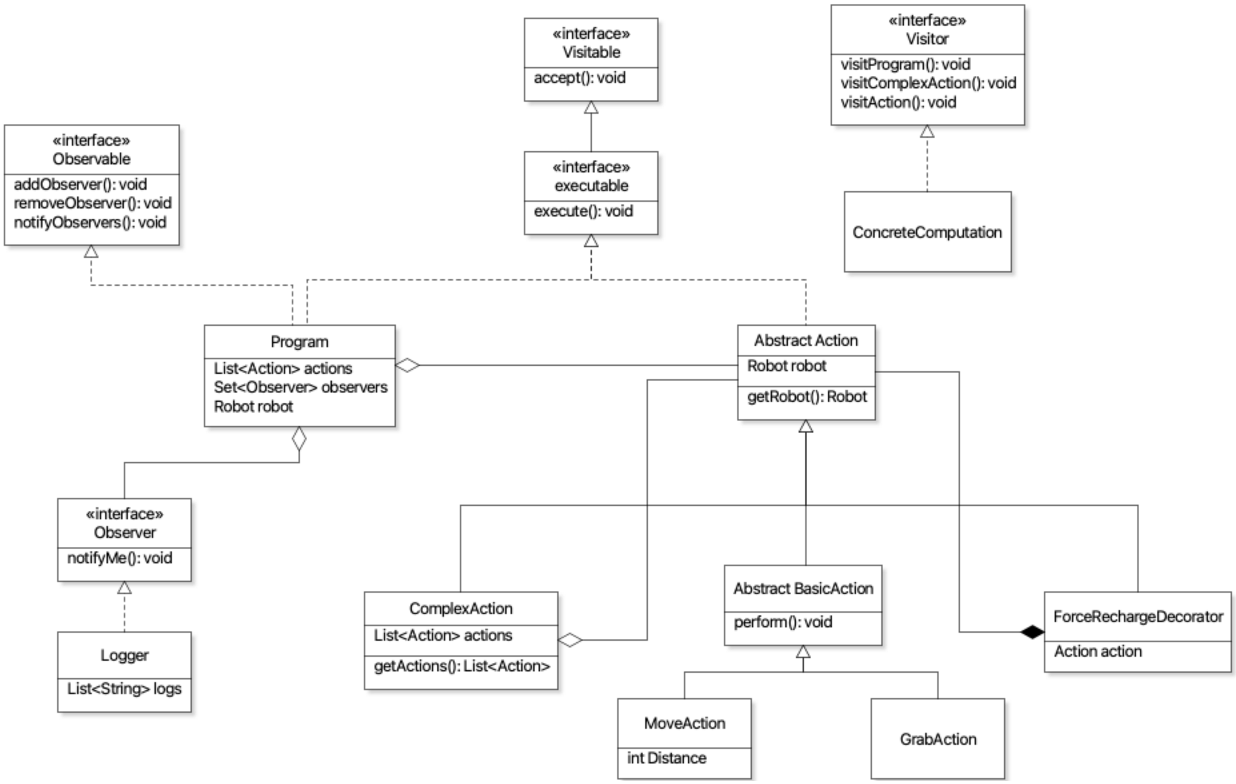
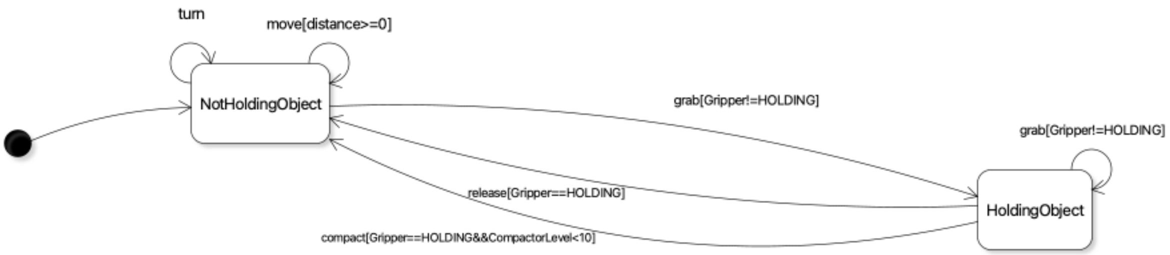
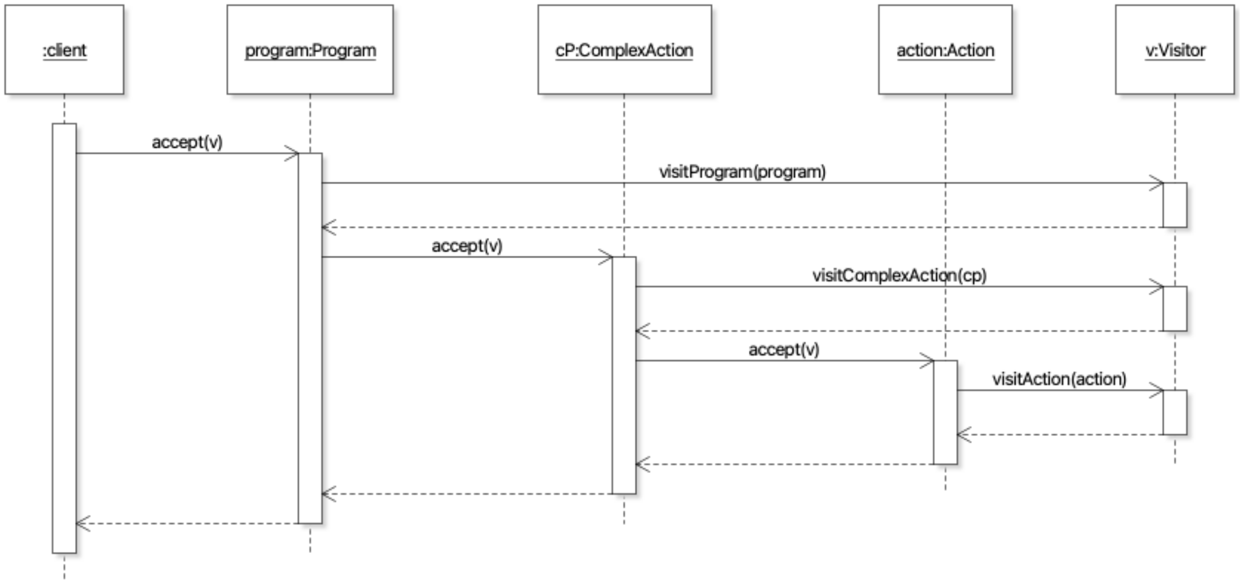
1. Since both action and program are executable, I decided to create an executable interface, which both action and program implement. Since both actions and programs are tied to a specific robot, I decided to ask the client to specify the robot at construction.
2. An abstraction for basic actions are created in the form of an abstract class that extends action, and each specific implementation is declared final to demonstrate the concept of a basic action. I considered using singleton design pattern, but if it was used, then managing multiple robots would not be possible as each basic action instance is tied to the class.
3. To ensure all basic actions follow the same protocol and to avoid unnecessary duplicated code, I used the template design pattern. Each specific basic action is implemented by overriding the perform() method.
4. A complex action is also an action, so I used the composite design pattern to address this problem. I avoided an addBasicAction() method to simplify class hierarchy, and because run time modification is not necessary. Instead, to create a complex action, the client is required to add all at run time with varargs.
5. The decorator design pattern was used to achieve this specific behaviour.
6. Naturally, since programs implement executable, the execute() method in program class is overridden to execute every action involved with the program in order.
7. To separate the behaviour of each basic action, for preconditions that require the action of another basic action, an error is thrown to prevent the execution of the action. If for instance, the robot arm is extended before moving, the software automatically retracts the arm before moving the robot. The overall implementation is within the execution() method of each action class. Another design is adding a check precondition() method in the action superclass to better generalize behaviour. However, I didn't use this design to enable precondition enforcing in cases where it is possible.
8. The visitor design pattern is used to address this problem. For traversing the program, I decided to implement it within the accept() of nodes of the program to allow better encapsulation and to decrease the amount of code the client needs to write. While letting client implement traversal allows for more inversion of control, since the traversal order usually does not matter in program, I decided to use the former design. If the overall aggregate structure changes however, there would need to be some significant code change.
9. The observer design pattern is used to address this problem. There were 2 available designs; to make actions observable or to make the program observable. I decided to not make actions observable since in cases where there are multiple programs using the same action on a robot, there could be unexpected behaviour. For instance, if 2 programs use action1, but the logger is only interested in one of the programs, executing 1 program would notify the logger nevertheless. The logger is notified whenever the action can execute successfully.



The class diagram demonstrating ISP and use of inheritance



State diagram showing the preconditions. Note that empty is not included due to JetUML. It would have an arrow of self-loop on NotHoldingObject and have conditions of Compactor!=EMPTY. Also note that since some preconditions are automatically enforced, such as retracting arm if the arm is extended in move and turn, the automatically enforced preconditions are not included.



The sequence diagram showing structure traversal and visit order