COMP 303 Winter 2021

Assignment 6

Belle Pan 260839939

22nd April, 2021

`Action` abstract class

- Abstraction of basic and complex `Action`s to avoid code duplication.
  - As each `Action` has different execution methods declared within individual robots, the method `doSpecificAction(Robot)` is specified for each `Action` and accesses the methods within the robot.
  - Each `Action` is executed by the `execute(Robot)` method, which performs the protocols before an action is performed: check the state of the battery, if the charge of the battery is less than 5 units, then recharge the battery, then call upon `doSpecificAction(Robot)` to perform the action, and, finally, update the battery level.
- `Action`s throw `AssertionErrors` when preconditions are not met. These errors are handled during the execution of `Program`s, as clients should not be executing actions directly and should rather be using `Program`s to do so.
  - `Program`s are also executed using the `execute(Robot)` method; this method calls each `Action` in the `Program` one by one using `Action.execute(Robot)`, and terminates if any preconditions are violated and an `AssertionError` is thrown.
- Some `Action`s, namely `Charge`, `Move`, `Turn`, and `ComplexAction`, have getter functions to retrieve values that are necessary for their execution and computation: the distance to move, the angle to turn, the action to execute after recharging the battery, and the list of actions that make up a complex action, respectively.
- Utilizing an abstract class for all actions ensures that we are executing each action using the same protocols. This also ensures that we can easily add new actions and that they all execute in the same manner!

`Sequenceable` interface

- Abstraction of operations performed on `ComplexAction`s and `Program`s, such as adding an `Action`, removing an `Action`, getting the number of `Action`s, and getting a copy of the `Action`s stored.
- Using an interface for sequenceable objects allows us to add new types of sequenceable objects very easily; for example, if we wanted to declare a specific type of `Program` that takes in only `Move` actions, then we can do so very easily by creating a new class that implements the `Sequenceable` interface and writing our own methods that override those within the interface.

`ComputationTypes` abstract class

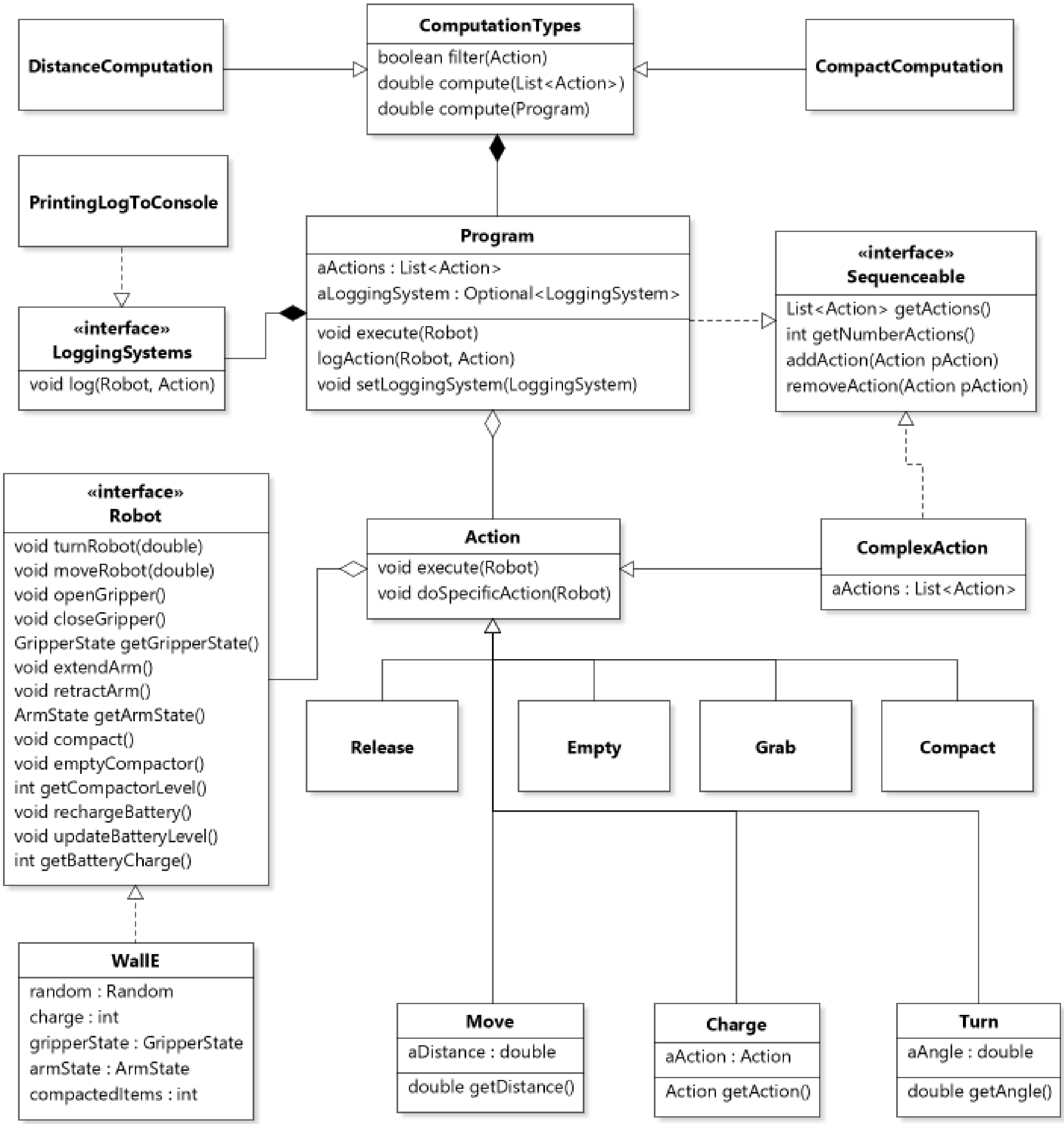
- Abstraction of different types of computations that can be performed on `Program`s.

- Ensures that new ways of computations may be added very easily by extending this class, and clients may be able to manipulate different types of data.
- Currently, two concrete classes of `ComputationTypes` have been created: `DistanceComputation` and `CompactComputation`.
- When `compute(Program)` is called, all actions within the program are put into an `ArrayList` and passed to `compute(List<Action>)`. Then, actions are filtered using the `filter(Action)` method and actions that should be included in the computation are tallied and computed accordingly.

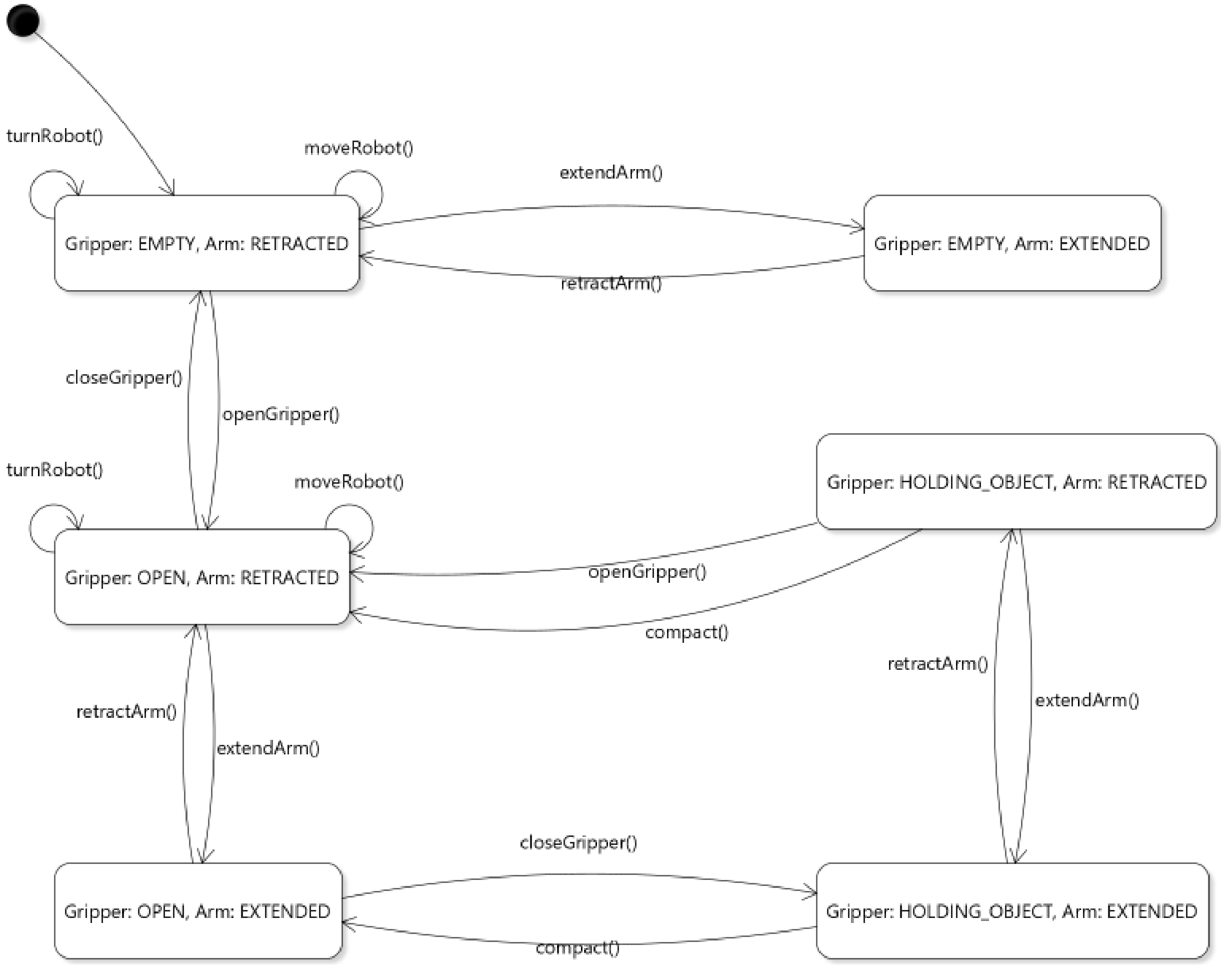
`LoggingSystem` interface

- Abstraction of methods that are used to log actions that are executed within `Program`s.
- Ensures that new ways of logging may be added very easily by implementing this interface.
- Currently, one concrete class has been created: `PrintingLogToConsole`
- The logging action utilizes the visitor interface, such that there is no need to alter different types of `Program`s to accommodate different forms of logging.
- Each `Program` may optionally hold a reference to a `LoggingSystem` and apply it to their execution. If the `Program` does not hold such a reference, the `Program` is simply executed without logging. For every `Action` executed, the `Program` calls `log(Action)`.
    - The `Program` terminates if any preconditions for the `Actions` are not met, so `log(Action)` only occurs after `Action.execute(Robot)` is called and does indeed run.

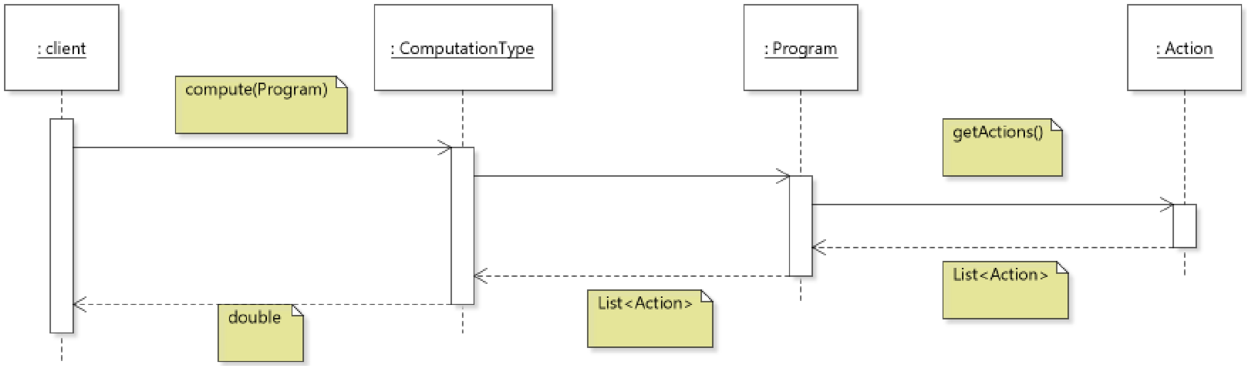
The class diagram below shows the entire application:



The state diagram below shows the valid operations that WALL-E can perform without violating preconditions:


turnRobot()
moveRobot()
extendArm()
retractArm()
Gripper: EMPTY, Arm: RETRACTED
Gripper: EMPTY, Arm: EXTENDED
closeGripper()
openGripper()
turnRobot()
moveRobot()
Gripper: HOLDING_OBJECT, Arm: RETRACTED
Gripper: OPEN, Arm: RETRACTED
openGripper()
compact()
retractArm()
extendArm()
retractArm()
extendArm()
closeGripper()
Gripper: OPEN, Arm: EXTENDED
Gripper: HOLDING_OBJECT, Arm: EXTENDED
compact()

The following is a sequence diagram that demonstrates a computation performed on a program:


: client
: ComputationType
: Program
: Action
compute(Program)
getActions()
List<Action>
List<Action>
double