1. I create a base for all actions called Action. This class is inherited by all actions and allows for the sharing of data members and select methods. This class also implements the interface ActionInterface. ActionInterface was created separate from Action so that decorators could provide the necessary APIs for use in the context of actions.

The class program uses composition where in it has as a data member a ComplexAction class. The ComplexAction class allows the programmer to create a series of actions that can be executed sequentially. Note that should one of the actions fail to meet the pre-conditions that ensure the safety of the Robot, the action in question will abort execution, and also terminate the execution of the complex action, and as a ripple effect the execution of the program.

2. The basic actions inherit from the Action class and implement the ActionInterface. This allows for less code replication and the sharing of data members.

3. In order to reduce the amount of code duplication, and ease future maintenance, a centralized approach was used to action execution is always preceded by checking the state of the battery, recharging when necessary, followed by execution, logging of the action's execution followed by the updating of the battery level. This was implemented through the function method in the base class Action which implements the ActionInterface. This allows the super class to always perform the aforementioned sequence, while still being able to execute the specifics of the action in question.

4. The ComplexAction class inherits from Action, while also using the composite pattern to store a list of actions to be executed in sequence. Note that a ComplexAction can also include other ComplexActions within it.

5. To enforce the charging of the battery before the execution of any action regardless of current charge, the decorator pattern was used to implement the class ForeceBatteryRechargeDecorator. This last class inherits from ActionDecorator, which allows the programmer to easily create future decorators while requiring a minimum amount of new code.

6. Programmers can use the Program class to add and remove actions from the program. A program can be run by using the run method which expects a Robot interface.

7. To ensure that preconditions of the WallE robot interface are not violated, several approaches were used. Firstly, we attempt to change the state, where possible, to ensure that the necessary preconditions are satisfied. Should it not be possible to do so, the execution of the action is aborted, and it returns false. In the case of a program or a complex action which executes an action that fails, further execution is stopped.

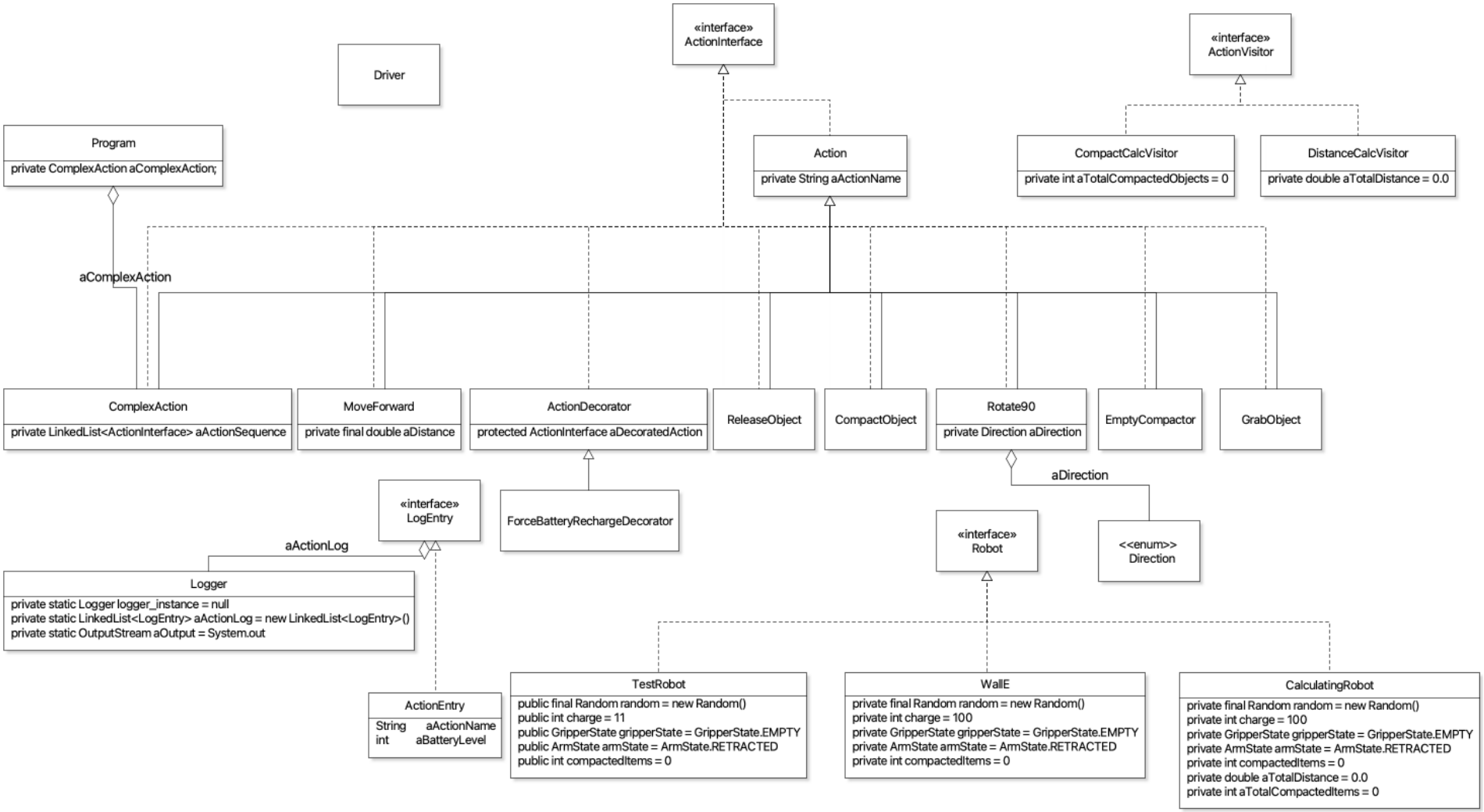
8. Two solutions are provided to the programmer. The first solution consists in using the visitor pattern whereby each action node is visited to accumulate either the distance traveled or the number of items compacted. However, note that because the visitor pattern does not execute and change the state of a robot, this approach does not take into consideration that the distance and number of compacted objects could be smaller because of an action failing to execute due to pre-condition violation. As a result of this, we used polymorphism to implement a custom robot in

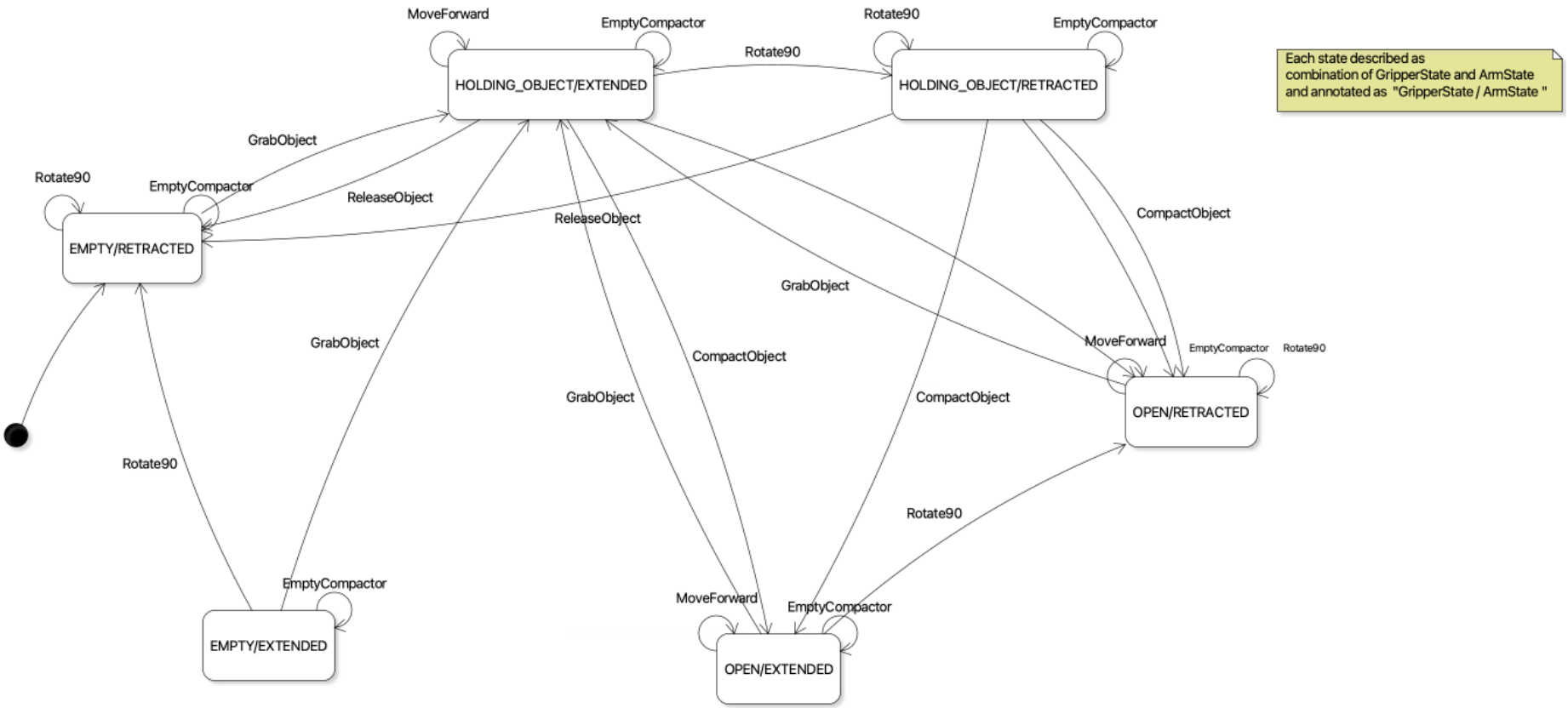
the class called CalculatingRobot. This class simulates a WallE and its state transitions throughout the computation of distance and number of compacted objects. This approach provides a more accurate calculation should an action fail to execute due precondition violation.

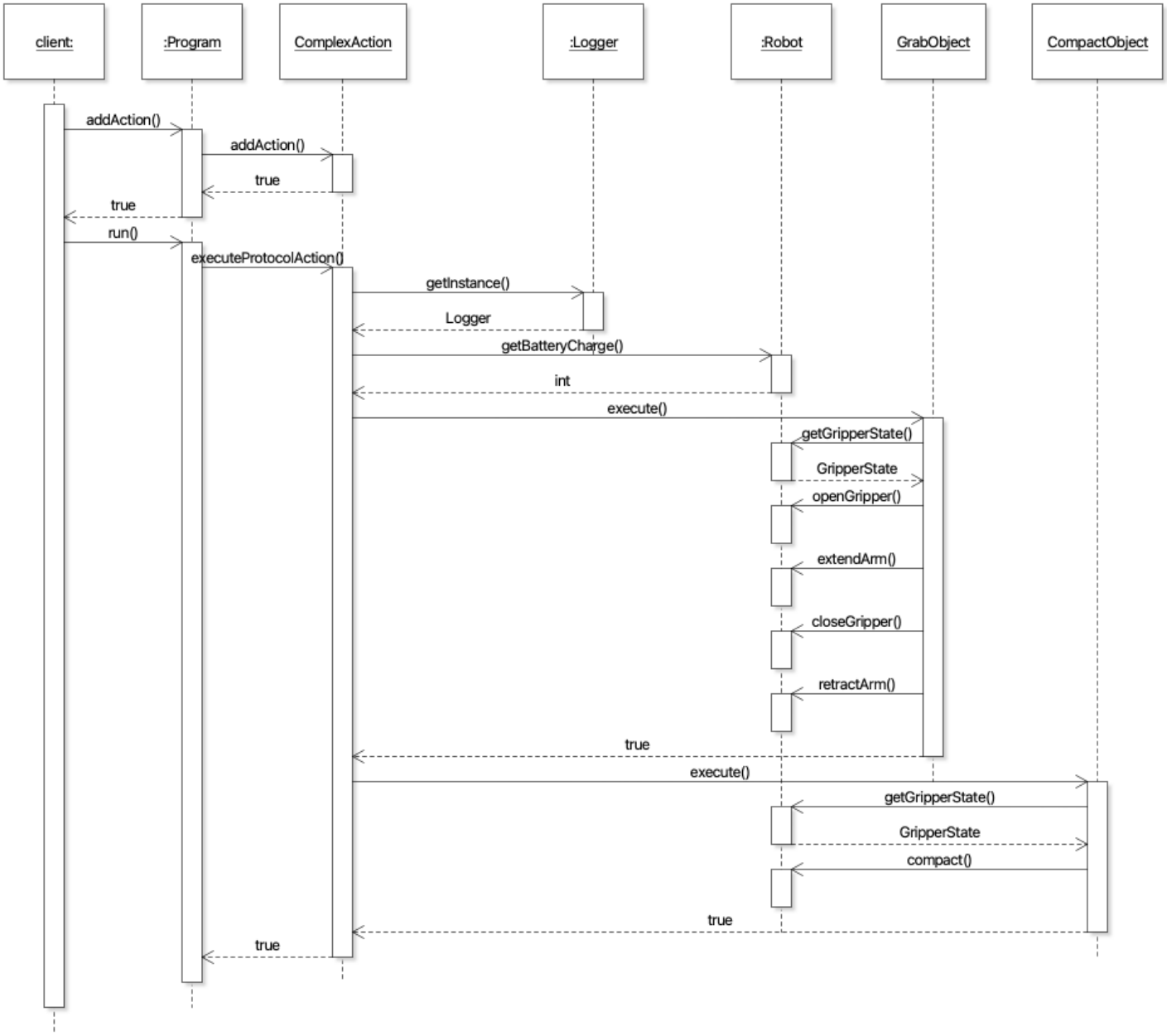
9. Logging was implemented using the Command pattern, whereby the command manager is the Logger class, and objects are added to the logger class upon the successful execution of an action. It should be noted that the Logger is flexible in that it has no expectations as to the objects being logged, but rather relies on an interface defined by LogEntry. ActionEntry implements the required interface that is specific for the execution of an action, however one can see how easily a new log object type could be created. This interface requires the implementation of a method which returns a string representing the state information. As well, the logger relies on the use of an OutputStream object to output its information. By default, this is set to standard out, however, as demonstrated in the test code, the user is free to use any class which implements the OutputStream interface, including a class such as FileOutputStream. The Logger also uses the Singleton pattern, so that a centralized logging system is place.

Testing

Several techniques were used to ensure that a high-quality application is delivered to Pixar. Stub objects were used by developing a stub Robot in the class TestRobot. This allows easy access to reading and modifying data members to ensure that 100% path code coverage was obtained for the entirety of the code base, with 1 exception. That exception is with the class GrabObject which uses a switch statement with the gripper state. In this code I added a case statement for each state along with a default case. Although I could have removed one of the states and have it covered by default, I judged that for future software maintenance reasons, it would be desirable to catch if an additional state was added and cause the execution to fail to ensure that the robot was not damaged.
As well I used reflection in order to access private data member of WallE so that we could easily exercise all code paths.


«interface»
ActionInterface
«interface»
ActionVisitor
Driver
Program
private ComplexAction aComplexAction;
Action
private String aActionName
CompactCalcVisitor
private int aTotalCompactedObjects = 0
DistanceCalcVisitor
private double aTotalDistance = 0.0
aComplexAction
ComplexAction
private LinkedList<ActionInterface> aActionSequence
MoveForward
private final double aDistance
ActionDecorator
protected ActionInterface aDecoratedAction
ReleaseObject
CompactObject
Rotate90
private Direction aDirection
EmptyCompactor
GrabObject
aDirection
«interface»
LogEntry
ForceBatteryRechargeDecorator
«interface»
Robot
<<enum>>
Direction
aActionLog
Logger
private static Logger logger_instance = null
private static LinkedList<LogEntry> aActionLog = new LinkedList<LogEntry>()
private static OutputStream aOutput = System.out
ActionEntry
String aActionName
int aBatteryLevel
TestRobot
public final Random random = new Random()
public int charge = 11
public GripperState gripperState = GripperState.EMPTY
public ArmState armState = ArmState.RETRACTED
public int compactedItems = 0
WallE
private final Random random = new Random()
private int charge = 100
private GripperState gripperState = GripperState.EMPTY
private ArmState armState = ArmState.RETRACTED
private int compactedItems = 0
CalculatingRobot
private final Random random = new Random()
private int charge = 100
private GripperState gripperState = GripperState.EMPTY
private ArmState armState = ArmState.RETRACTED
private int compactedItems = 0
private double aTotalDistance = 0.0
private int aTotalCompactedItems = 0


MoveForward
EmptyCompactor
Rotate90
Rotate90
EmptyCompactor
HOLDING_OBJECT/EXTENDED
HOLDING_OBJECT/RETRACTED
Each state described as
combination of GripperState and ArmState
and annotated as "GripperState / ArmState "
GrabObject
Rotate90
EmptyCompactor
ReleaseObject
ReleaseObject
CompactObject
EMPTY/RETRACTED
GrabObject
GrabObject
CompactObject
MoveForward
EmptyCompactor
Rotate90
GrabObject
CompactObject
OPEN/RETRACTED
Rotate90
Rotate90
EmptyCompactor
MoveForward
EmptyCompactor
EMPTY/EXTENDED
OPEN/EXTENDED


client:
:Program
ComplexAction
:Logger
:Robot
GrabObject
CompactObject
addAction()
addAction()
true
true
run()
executeProtocolAction()
getInstance()
Logger
getBatteryCharge()
int
execute()
getGripperState()
GripperState
openGripper()
extendArm()
closeGripper()
retractArm()
true
execute()
getGripperState()
GripperState
compact()
true
true