Vacuum Cleaner Simulator
– Advanced Topics in Programming –
Exercise I

1. Introduction

In this document we will give an overview of our implementation of the Vacuum Cleaner Simulator (exercise I). This is only a complementary explanation to the attached class and sequence diagrams, while the only complete documentation of the simulator is the code itself.
It is important to note that our program contains several unnecessary complexities – they result from our aspiration to lay good foundations for the next exercise. They include creating a list of houses and a list of algorithms, on which we iterate in a "Round Robin" fashion, loading a house from a file, and more. We note that these complexities make the execution of our program somewhat less efficient, but it is unnoticeable and gives the program robustness and a sort of scalability.
In the following paragraphs we will supply a basic overview of the important or unique classes, but we will by no means elaborate on all of the classes in our code. We believe that most of the classes in this project are intuitive and self-explanatory – a House is a representation of a house and SensorImpl is an implementation of an AbstractSensor – we left these explanations out of this text to keep it succinct and to the point. The reader may find it helpful to observe the class diagram if any of the classes is not clear from this text or from a brief glance at the code.

2. Main

The flow of our program begins with the Main function, where we, using helper methods, parse the command line arguments, load files based on them, populate the configuration map with parameters read from the configuration file and deserialize the house file into a House object. These are preparations that are done prior to the creation and execution of the Simulator, which does not handle files – we thought this separation of responsibilities between the Main method and the actual simulation execution to be good practice for the design.
The Main class is also the only part of the program which is aware of algorithm implementations. Lower in the chain of execution, all other classes, beginning with the Simulator, treat the algorithms as AbstractAlgorithms – also, for robustness' sake.

3. Simulator

A simulator instance is created in the Main method and then executed a few lines later. The Simulator is constructed with a configuration map, a list of House instances and a list of AbstractAlgorithms. The later are passed on to the constructor of another important class, which is the Robot class – the Simulator keeps a list of those as an instance member. Another member is the scoreMatrix, which is only a 1x1 matrix in this exercise but will hopefully be easily adjustable for the next exercise.
Our Simulator is run using a call to execute(). In this method, we iterate over the houses and for each house, iterate over all robots (each robot corresponds to a different algorithm) in a "Round Robin" fashion, and execute them step-by-step.

The Simulator enforces the simulation constraints, orchestrates the robots and manages the simulation framework (mainly the score matrix). The “robotic” logic is separated from the simulation hassle by delegating those responsibilities to the helper class Robot, which will be explained next.

4. Robot

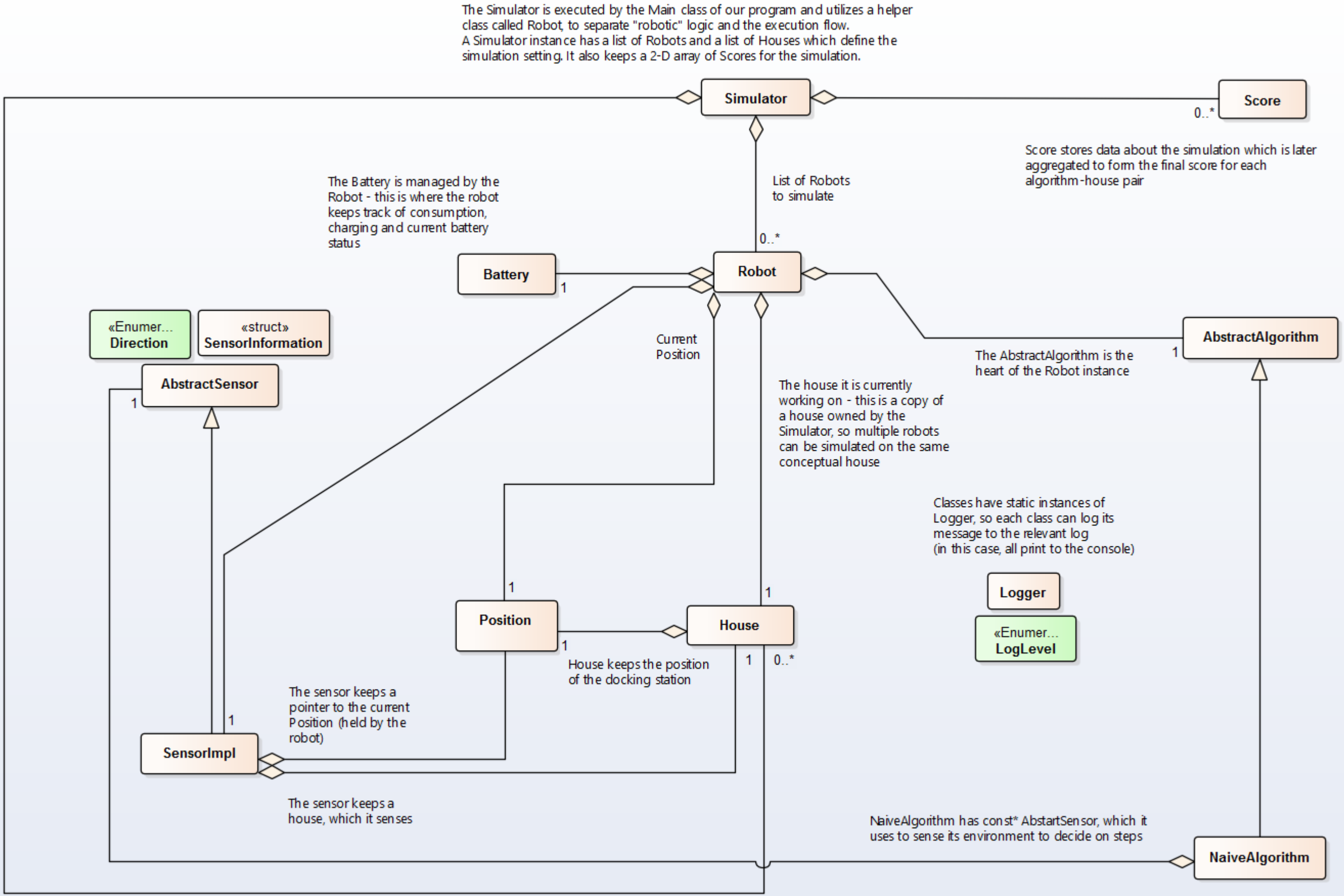
The Robot class keeps an AbstractAlgorithm, a House, a sensor implementation, a Battery and a Position. The algorithm is a reference, because it defines the robot and once initialized should never be changed. The House, on the other hand, is a pointer, to allow for the robot to be reused to clean several houses. This structure seems to capture the natural notion of a real-life robot – it has one algorithm which acts as its virtual mind, but the same robot may be placed at different houses at different times. Other integral parts of the robot are the sensor and the battery, as mentioned.
It is worth to note that the algorithm receives the configuration and therefore may keep track of an assumed battery status, but it is unrelated to the battery managed by the robot. As said above, the Robot class is simply a helper class to delegate responsibilities from the Simulator – so the battery managed may be viewed as the Simulator enforcing battery regime and rules.
The final member of the Robot instance, not including Boolean flags which are out of the scope of this document, is the position, which simply holds the current position of the robot, and is also shared with the sensor (as reference).
All communication between the Simulator and the battery, house (e.g. total dust in house) or sensor are coordinated by the Robot.

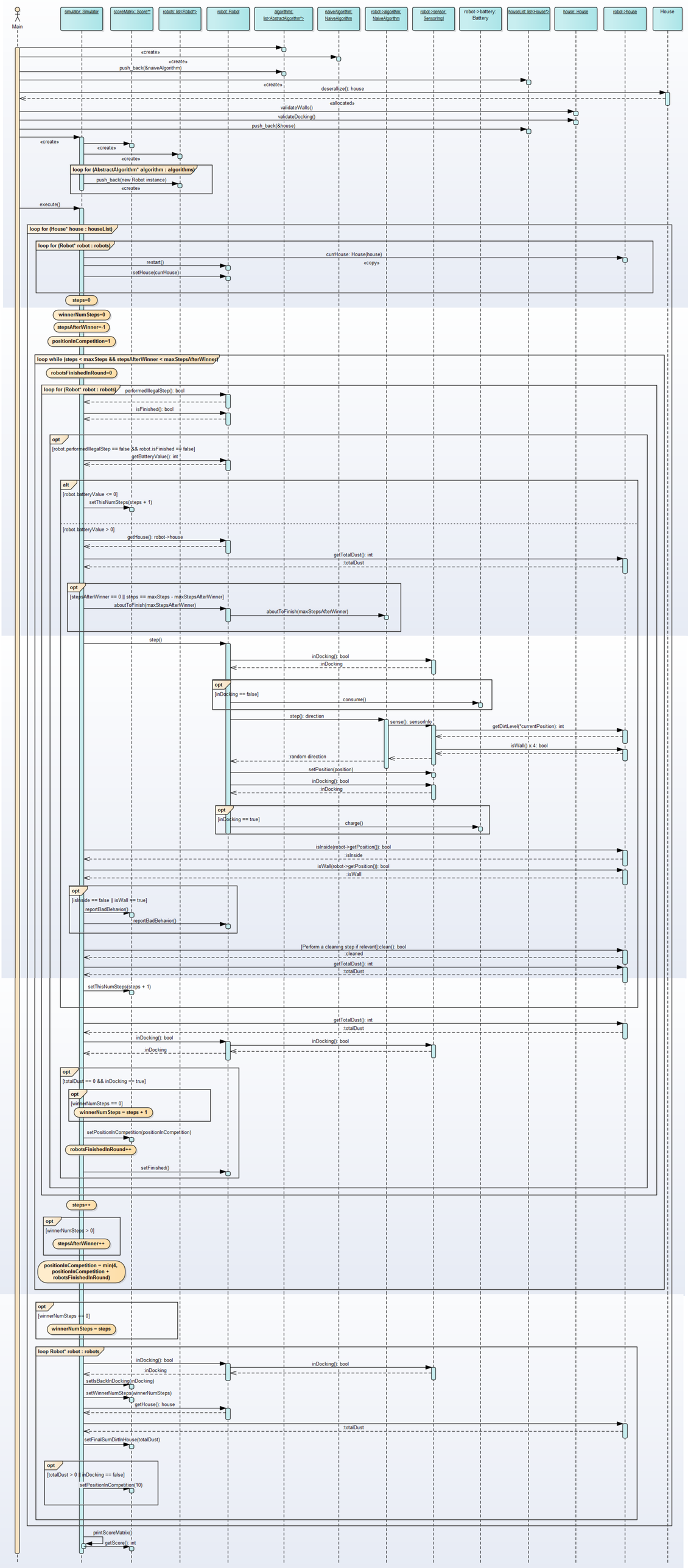
5. Logger

In order to allow efficient and productive debugging, we have written a very simple Logger class. This class prints messages to the console with their timestamp, logger name (i.e. executing class) and log level. This last piece of information allows us to control the verbosity and severity of the messages printed to the console. The exercise will be submitted with log level set to Fatal, so only fatal errors will be output to the screen, but you may change this to your convenience.

6. Conclusion

We hope that this overview gives the reader a solid starting point to examine the diagrams and code. We recommend starting with the class diagram, in order to complete the picture portrayed in the paragraphs above.


The Simulator is executed by the Main class of our program and utilizes a helper class called Robot, to separate "robotic" logic and the execution flow.
A Simulator instance has a list of Robots and a list of Houses which define the simulation setting. It also keeps a 2-D array of Scores for the simulation.
Simulator
Score
0..*
Score stores data about the simulation which is later aggregated to form the final score for each algorithm-house pair
The Battery is managed by the Robot - this is where the robot keeps track of consumption, charging and current battery status
List of Robots to simulate
0..*
Battery
1
Robot
«Enumer... Direction
«struct» SensorInformation
AbstractSensor
1
Current Position
The AbstractAlgorithm is the heart of the Robot instance
1
AbstractAlgorithm
The house it is currently working on - this is a copy of a house owned by the Simulator, so multiple robots can be simulated on the same conceptual house
Classes have static instances of Logger, so each class can log its message to the relevant log (in this case, all print to the console)
Logger
«Enumer... LogLevel
1
Position
1
House
1
House keeps the position of the docking station
1
0..*
The sensor keeps a pointer to the current Position (held by the robot)
1
SensorImpl
The sensor keeps a house, which it senses
NaiveAlgorithm has const* AbstartSensor, which it uses to sense its environment to decide on steps
NaiveAlgorithm


Main
simulator: Simulator
scoreMatrix: Score**
robots: list<Robot*>
robot: Robot
algorithms: list<AbstractAlgorithm*>
naiveAlgorithm: NaiveAlgorithm
robot->algorithm: NaiveAlgorithm
robot->sensor: SensorImpl
robot->battery: Battery
houseList: list<House*>
house: House
robot->house
House
«create»
«create»
push_back(&naiveAlgorithm)
«create»
deserallize(): house
«allocated»
validateWalls()
validateDocking()
push_back(&house)
«create»
«create»
«create»
loop for (AbstractAlgorithm* algorithm : algorithms)
push_back(new Robot instance)
«create»
execute()
loop for (House* house : houseList)
loop for (Robot* robot : robots)
currHouse: House(house)
«copy»
restart()
setHouse(currHouse)
steps=0
winnerNumSteps=0
stepsAfterWinner=-1
positionInCompetition=1
loop while (steps < maxSteps && stepsAfterWinner < maxStepsAfterWinner)
robotsFinishedInRound=0
loop for (Robot* robot : robots)
performedIllegalStep(): bool
isFinished(): bool
opt
[robot.performedIllegalStep == false && robot.isFinished == false]
getBatteryValue(): int
alt
[robot.batteryValue <= 0]
setThisNumSteps(steps + 1)
[robot.batteryValue > 0]
getHouse(): robot->house
getTotalDust(): int
:totalDust
opt
[stepsAfterWinner == 0 || steps == maxSteps - maxStepsAfterWinner]
aboutToFinish(maxStepsAfterWinner)
aboutToFinish(maxStepsAfterWinner)
step()
inDocking(): bool
:inDocking
opt
[inDocking == false]
consume()
step(): direction
sense(): sensorInfo
getDirtLevel(*currentPosition): int
isWall() x 4: bool
random direction
setPosition(position)
inDocking(): bool
:inDocking
opt
[inDocking == true]
charge()
isInside(robot->getPosition()): bool
:isInside
isWall(robot->getPosition()): bool
:isWall
opt
[isInside == false || isWall == true]
reportBadBehavior()
reportBadBehavior()
[Perform a cleaning step if relevant] clean(): bool
:cleaned
getTotalDust(): int
:totalDust
setThisNumSteps(steps + 1)
getTotalDust(): int
:totalDust
inDocking(): bool
inDocking(): bool
:inDocking
opt
[totalDust == 0 && inDocking == true]
opt
[winnerNumSteps == 0]
winnerNumSteps = steps + 1
setPositionInCompetition(positionInCompetition)
robotsFinishedInRound++
setFinished()
steps++
opt
[winnerNumSteps > 0]
stepsAfterWinner++
positionInCompetition = min(4, positionInCompetition + robotsFinishedInRound)
opt
[winnerNumSteps == 0]
winnerNumSteps = steps
loop Robot* robot : robots
inDocking(): bool
inDocking(): bool
:inDocking
setIsBackInDocking(inDocking)
setWinnerNumSteps(winnerNumSteps)
getHouse(): house
:totalDust
setFinalSumDirtInHouse(totalDust)
opt
[totalDust > 0 || inDocking == false]
setPositionInCompetition(10)
printScoreMatrix()
getScore(): int