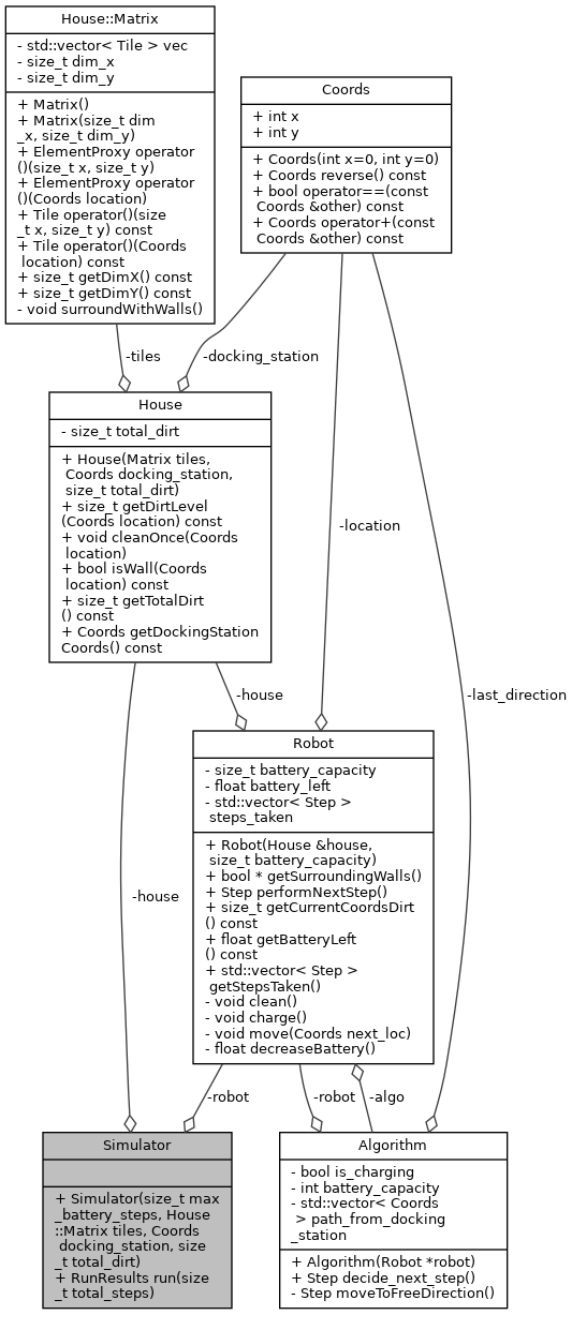
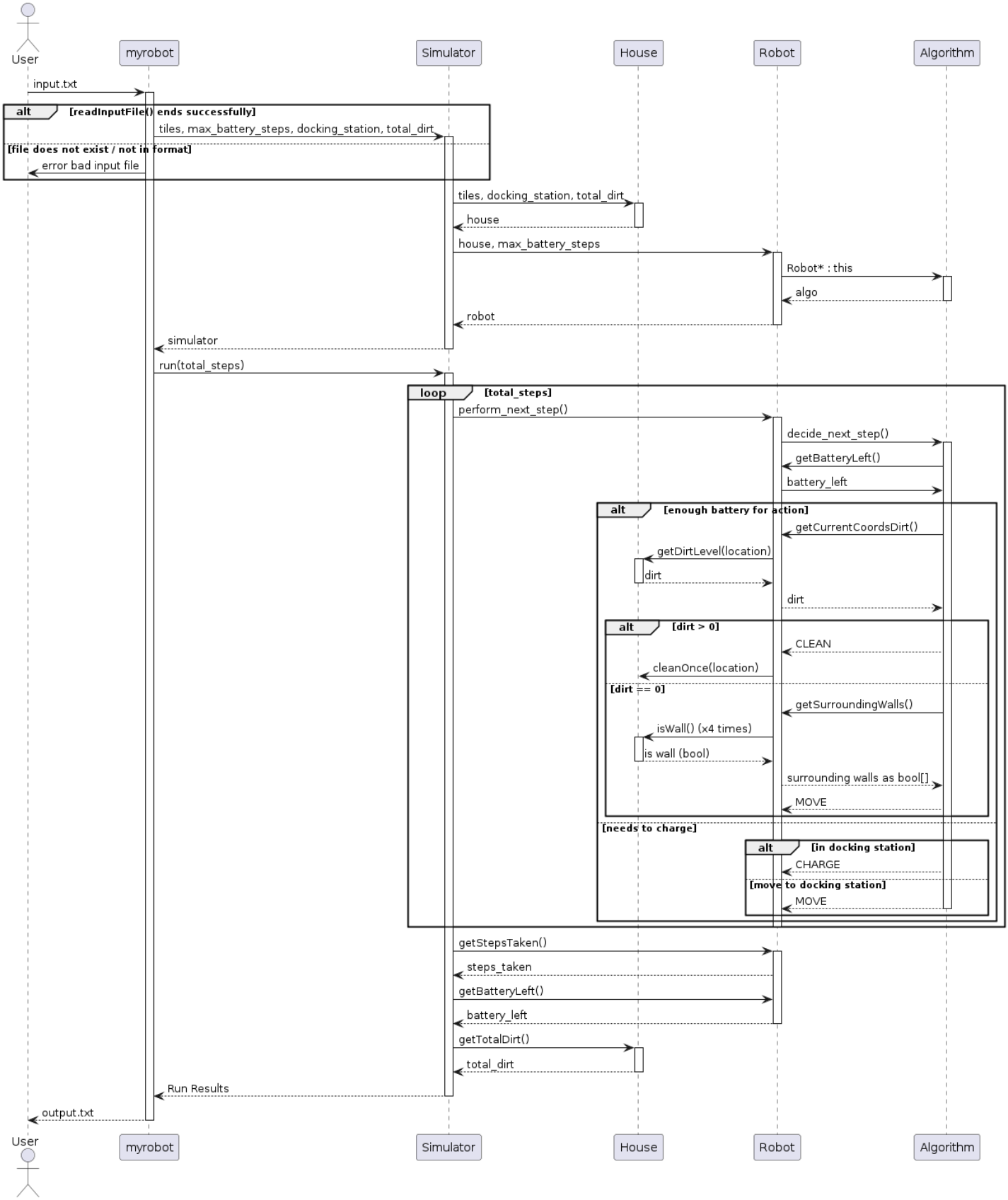
# High Level Design

**In order to automate the process of documenting the project, we used a documentation tool called Doxygen. In order to view full documentation HTML, go to the doc/html folder and run the index.html file in your browser.**

UML class Diagram (automatically generated by Doxygen)



UML sequence diagram (manually generated by us)



## Design considerations and alternatives:

**Input** – we chose to take input as a .txt file that its rows represent the data we need.
The first row represents the max_battery_steps parameter and thus should contain a single integer in ASCII digits. The same applies to the second row, which represents the total number of steps of the robot to be run for.
All the next rows represent the house map as a matrix of tiles sized $n \times m$. The number of rows set the $n$ value of the matrix and the longest row sets the $m$ value. Each row should contain characters from $\{0, \dots, 9, W, D, SPACE\}$ only, where the digits represent the amount of dirt, space represents no-dirt, W represents a wall and D a docking station (which is checked to be single). We chose this representation because it is minimalistic and easy to work with. The conversion from the input text to actual objects was quite intuitive, and so was error checking.
Before this version, we considered the format of the input to be just a long list of coordinates and dirt-value tuples, but it would be complex to generate and non-intuitive.

**Class structure** – we chose the input/output to be done in myrobot.cpp and the other functionalities to be held in classes. After input validation, input parameters are sent to a Simulator class object. Its purpose is to separate file operations from actual robot logic.
The simulator then creates objects of classes House and Robot and runs the robot around the house for total_steps times. The House class contains a matrix of tiles as its 'floor', and the Robot class contains an Algorithm class object, which it enquires every time it is asked by the simulator to take a step. The algorithm takes decisions according to information gained from its robot and passes those action decisions back to the robot.

**Sensors** - the robot is designed to pass information that its sensors collect to the algorithm. We considered adding a class for each sensor, but decided that it would be unnecessary and would create redundancy. We didn't want to have an ad-hoc class that's used only once and does only one simple thing. So, we just simulated the notion of sensors as public functions in Robot that query the House object's public functions.

**Tiles** - another consideration we took was how to represent the tiles of the house. At first, we thought of having the tiles held inside a hash map that takes coordinates as key and returns the tile. But we gave up this idea because we wanted to minimize the search time of the tiles. The problem with hash maps is that they do not take advantage of spatial locality in the queries the robot makes to the house's tiles (in a hash map the tiles are scattered around a buffer in a random order). So, we chose a Matrix: a random access 2D vector. We defined the Matrix class, and it contained a standard C++ vector, which contained all the tiles. To make it 2D we made sure that accessing the matrix is done by the '()' operator (specifying coordinates) and to return the (x,y) entry of the matrix as the (x*dimx + y) entry of the vector.
In addition, we used ElementProxy to enable accessing the matrix nicely similar to an array (for example: 'mat(3,10)' ). It was a little complex to implement, but it might aid readability in future exercises.

**Steps and Coordinates** - We chose to implement the steps as a struct, which includes a field for StepType and a field for Coords. StepType is an enum containing the following elements: CLEAN(0), MOVE(1), CHARGE(2), and Coords is a class that represents coordinates and implements comparison, addition, and reversion of coordinates. The implementation of steps and coordinates greatly facilitated the coding process (for example, comparing and adding coordinates) and its clarity.

**The algorithm** – the algorithm also went through considerations. At first, we chose to let the robot go in a completely random direction on every clean tile, but then we thought it wouldn't get very far, so we went to something in the middle. We made the robot choose a random direction, and then go in a straight line until it encounters a wall or a dirty tile. We also assume that it will handle narrow corridors better.
Also, when our robot reaches a dirty tile, it cleans it completely, we thought it is efficient because a random robot rarely reaches non-clean tiles, and it should spend a given opportunity. Another modification we made to the algorithm is that it will choose a new direction each time it is recharged at the docking station. Before the modification, there could be situations where, for example, if there was a docking station in the middle of a long corridor that could not be reached to the end (due to battery considerations), the robot would never explore the direction opposite to the first direction it chose (because the algorithm would continue in the original direction selected as long as it didn't encounter a wall).

The algorithm maintains the path it has taken since the last recharge using a vector that stores the direction of travel each time it moves away from the docking station. The algorithm uses this vector to know how far it is from the station (the size of the vector) and how to return. When the algorithm detects that the battery size is less than the distance from the docking station plus 2 (meaning if it takes another step, the battery may not suffice to return), it starts guiding the robot back to the docking station poping the last direction from the vector, reversing the direction, and steering the robot there. When the robot reaches the docking station, the algorithm instructs it to recharge fully.

**Robot -** The robot holds the house and the algorithm as parameters. The simulator instructs the robot to take one step, and in response, the robot asks the algorithm to know its next move. The robot exposes "the sensors" as public functions so that the algorithm can make a decision. The algorithm responds to the robot by returning it the next step (type of step and the offset (direction) to move towards it,  if the step is not a move, then the offset is [0,0]). The robot then performs the step and the necessary updates to the fields it holds.

**Output** – the output file first contains a list of the steps taken, in a format of a step's coordinates (the coordinates that the robot will be in after the step is executed) the followed by the step type (clean, move or charge). It also states the total number of steps performed, the battery remaining, the amount of dirt left in the house and whether or not the robot is in the docking station.

## Testing approach

In conducting the tests, we aimed to verify three main issues:

1. **File Input Validity** - We checked both handling files with invalid parameters (appropriate error throwing and clean exit without crashing) and the correct parsing of a valid file.
2. **Robot Operation Validity** - We verified that the robot performs operations as required by the task.
3. **Output File Validity** - We checked that the output file contains accurate data corresponding to the runs we conducted.
4. **Return value correctness.**

The tests were performed manually by running files, most of which were designed to test specific cases or edge cases (e.g., a file with a non-numeric battery size, a house that has only a docking station surrounded by walls, etc.), and some are more “standard” ones that we used to ensure the robot's operation was correct on a bit bigger testing sample.

With the invalid files, we confirmed that an error message was indeed printed and that the software ended correctly. With the valid files, we followed the software's printouts, compared them to the output file, and verified that all robot movements were legal and desired according to the task specifications and that the data in the output file was valid relative to the run.

**Some of the tests performed:**

**Invalid files:**

1. Invalid parameters (battery capacity and maximum number of steps) - non-numeric, float, or negative.
2. Illegal characters in the house description (illegal signs, floats between 1-9, negative numbers).
3. A house without a docking station.
4. A house with more than one docking station.
5. Files with fewer than 3 lines (missing house description or parameters).

**Valid files with edge cases:**

1. A house with a single tile that is a docking station.
2. A house with only one tile beside the docking station.
3. Battery size of 1.
4. Battery size of 2.
5. A house without any dirt.

6. A house received without built-in walls (testing that the wall completion in the file is properly done and the robot does not exceed the house boundaries).
7. A corridor house with various amounts of dirt, number of steps, and battery sufficient for the robot to finish charged and the house clean (testing to ensure that the robot cleans effectively, and the output states that the mission was completed successfully).

**Things we checked during the robot's run to ensure the operation's validity:**

1. It does not stay in a clean spot for more than one move.
2. It does not enter walls.
3. The battery is charged properly and depleted appropriately.
4. Cleaning is performed by reducing the dirt level by one each time.
5. It moves 1 tile at a time,  choosing a direction that matches the way we wanted it to operate.