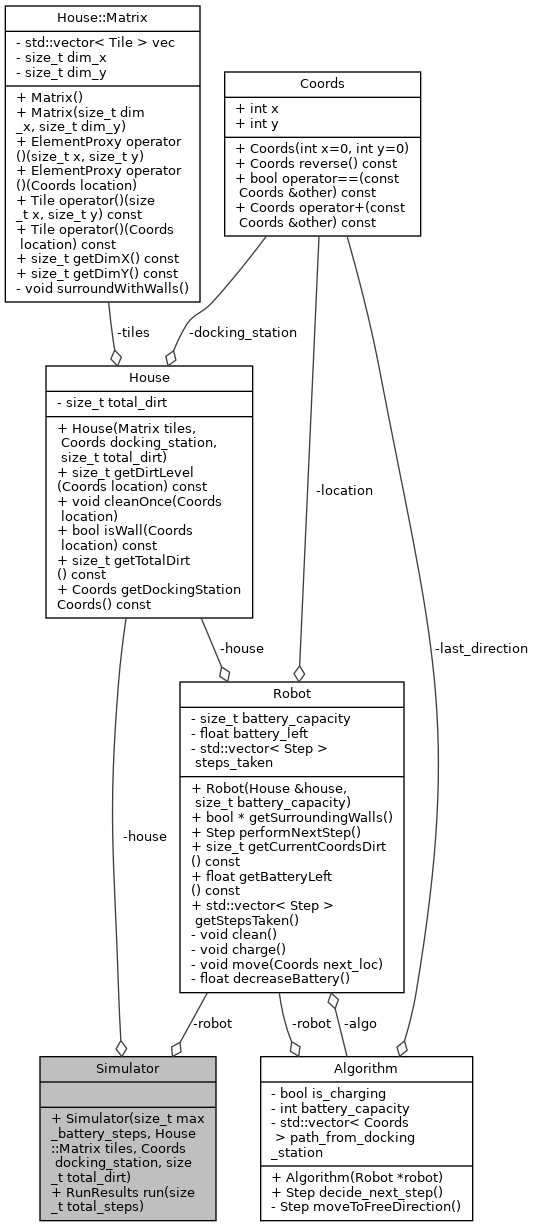
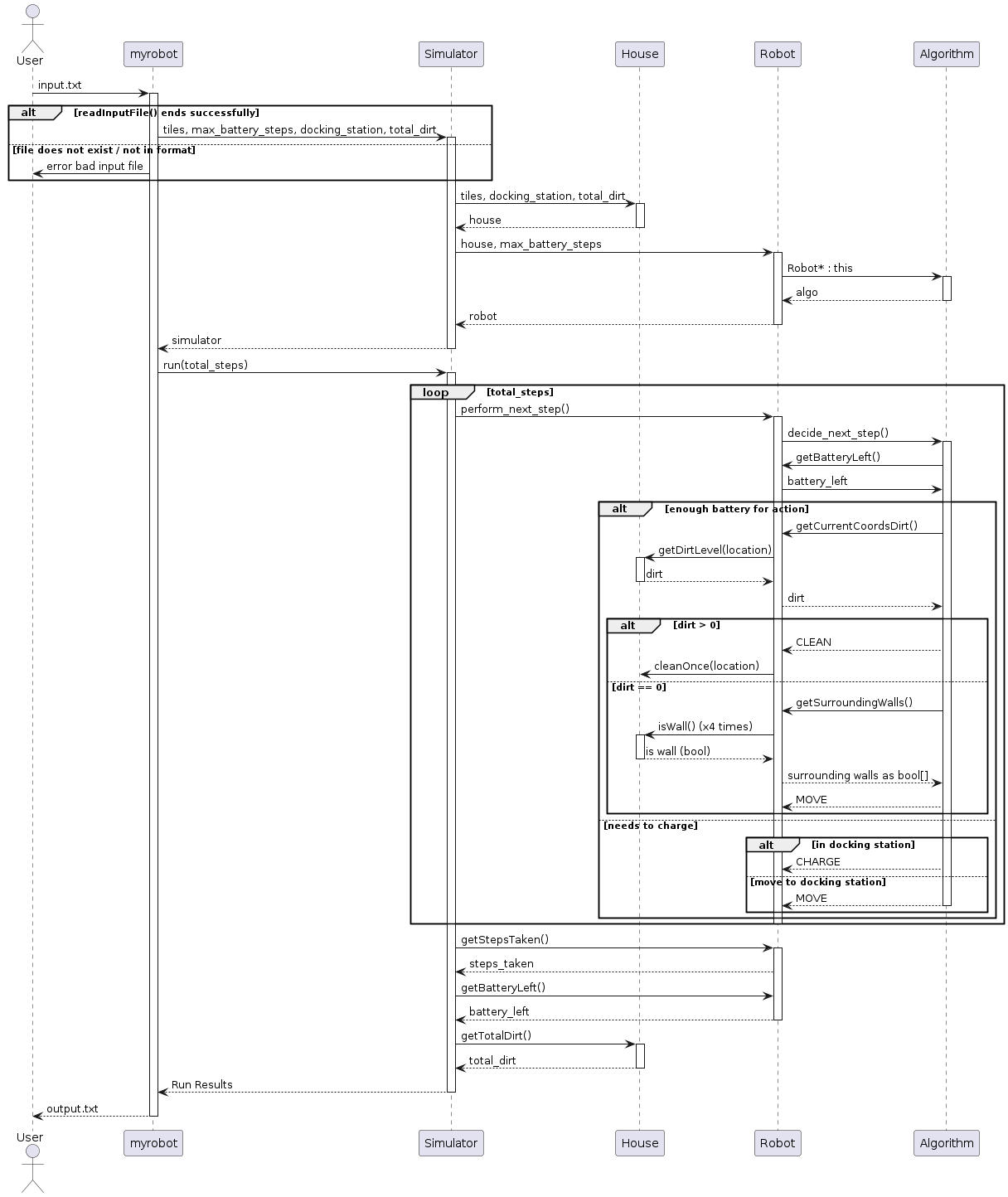
**High Level Design**

**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 . The number of rows set the value of the matrix and the longest row sets the value. Each row should contain characters from 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.

**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.

**Testing approach**