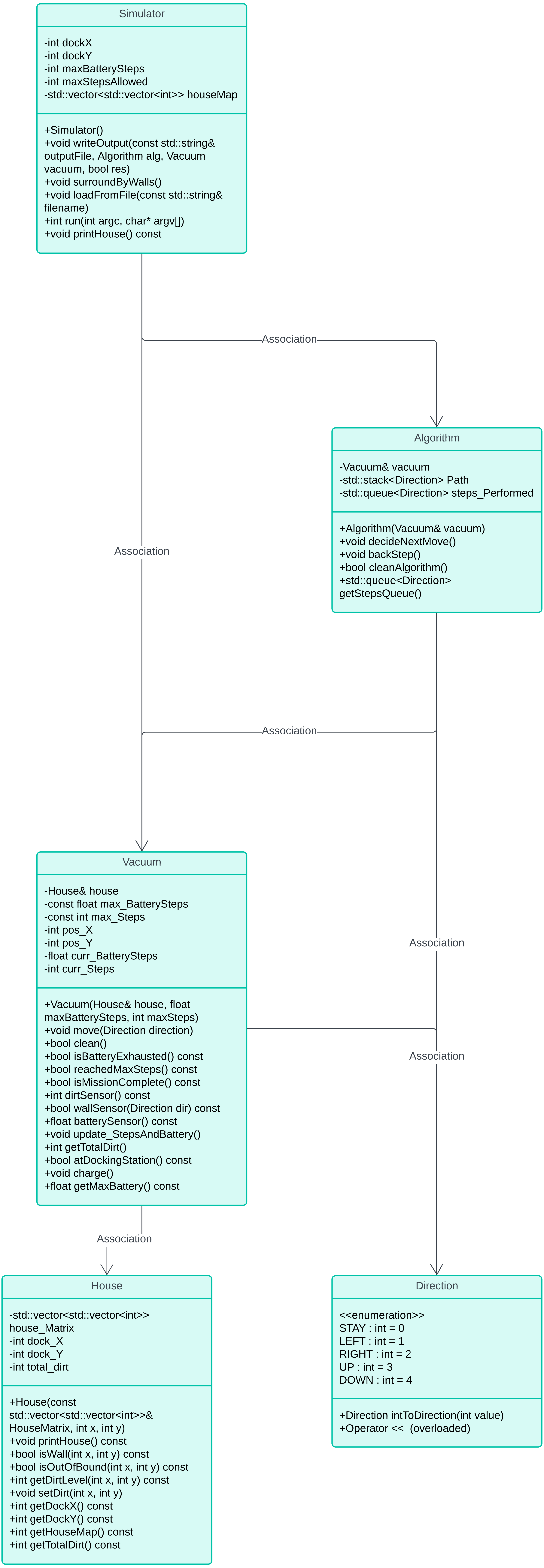


Simulator

-int dockX
-int dockY
-int maxBatterySteps
-int maxStepsAllowed
-std::vector<std::vector<int>> houseMap

+Simulator()
+void writeOutput(const std::string& outputFile, Algorithm alg, Vacuum vacuum, bool res)
+void surroundByWalls()
+void loadFromFile(const std::string& filename)
+int run(int argc, char* argv[])
+void printHouse() const

Algorithm

-Vacuum& vacuum
-std::stack<Direction> Path
-std::queue<Direction> steps_Performed

+Algorithm(Vacuum& vacuum)
+void decideNextMove()
+void backStep()
+bool cleanAlgorithm()
+std::queue<Direction> getStepsQueue()

Vacuum

-House& house
-const float max_BatterySteps
-const int max_Steps
-int pos_X
-int pos_Y
-float curr_BatterySteps
-int curr_Steps

+Vacuum(House& house, float maxBatterySteps, int maxSteps)
+void move(Direction direction)
+bool clean()
+bool isBatteryExhausted() const
+bool reachedMaxSteps() const
+bool isMissionComplete() const
+int dirtSensor() const
+bool wallSensor(Direction dir) const
+float batterySensor() const
+void update_StepsAndBattery()
+int getTotalDirt()
+bool atDockingStation() const
+void charge()
+float getMaxBattery() const

House

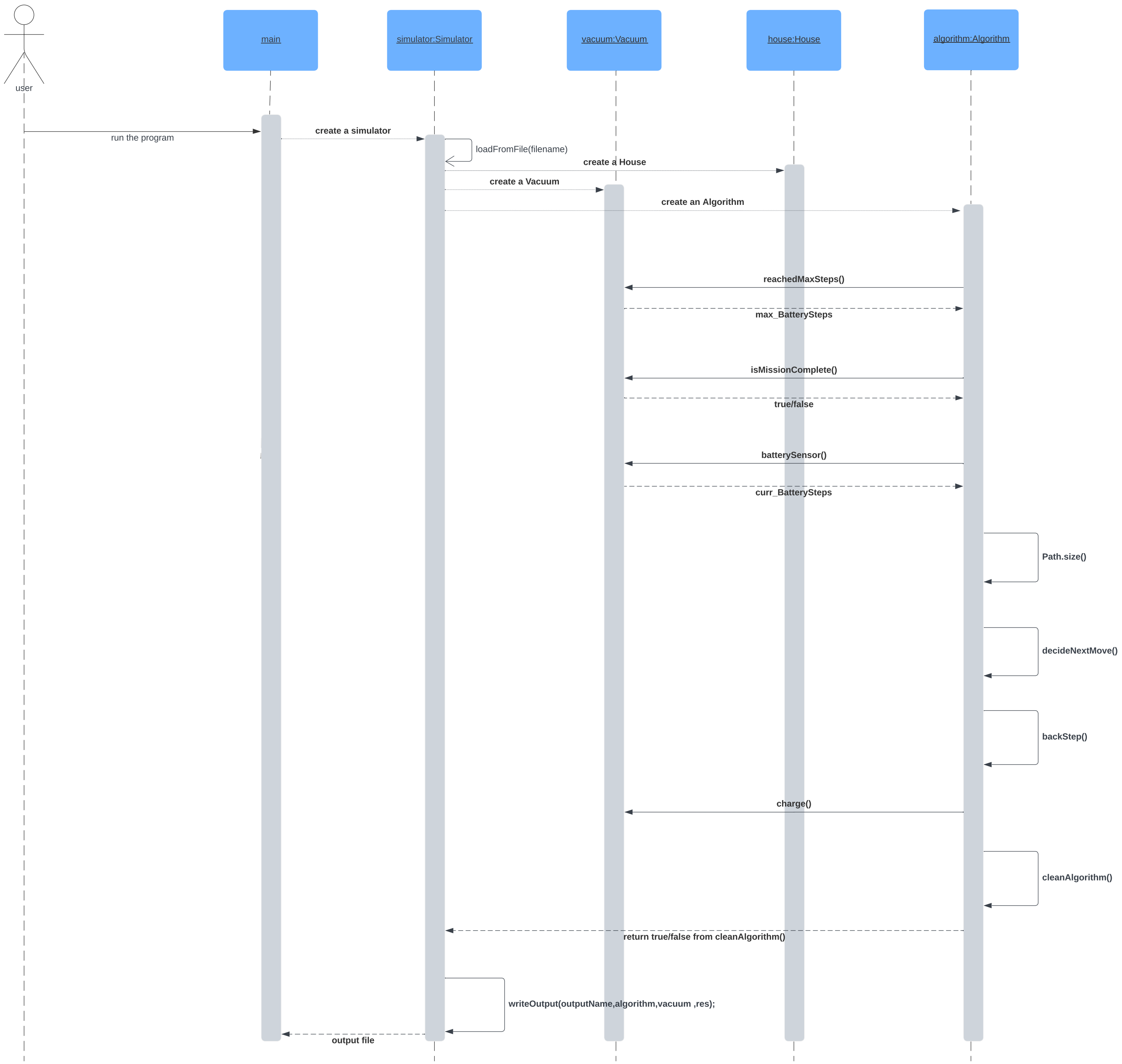
-std::vector<std::vector<int>> house_Matrix
-int dock_X
-int dock_Y
-int total_dirt

+House(const std::vector<std::vector<int>>& HouseMatrix, int x, int y)
+void printHouse() const
+bool isWall(int x, int y) const
+bool isOutOfBound(int x, int y) const
+int getDirtLevel(int x, int y) const
+void setDirt(int x, int y)
+int getDockX() const
+int getDockY() const
+int getHouseMap() const
+int getTotalDirt() const

Direction

<<enumeration>>
STAY : int = 0
LEFT : int = 1
RIGHT : int = 2
UP : int = 3
DOWN : int = 4

+Direction intToDirection(int value)
+Operator << (overloaded)


main
simulator:Simulator
vacuum:Vacuum
house:House
algorithm:Algorithm
user
run the program
create a simulator
loadFromFile(filename)
create a House
create a Vacuum
create an Algorithm
reachedMaxSteps()
max_BatterySteps
isMissionComplete()
true/false
batterySensor()
curr_BatterySteps
Path.size()
decideNextMove()
backStep()
charge()
cleanAlgorithm()
return true/false from cleanAlgorithm()
writeOutput(outputName,algorithm,vacuum ,res);
output file

Design Considerations and Alternatives

Explanation of the Input File

The input file for your vacuum cleaner simulation contains critical information about the house layout, the vacuum cleaner's docking station, battery capacity, and maximum allowed steps. Below is a detailed explanation of each part of the input file:

1. **Docking Station Coordinates**:
   - **Format**: `x, y`
   - **Description:** This line specifies the coordinates of the docking station in the house matrix. The docking station is the starting point for the vacuum cleaner and is always clean (i.e., it does not contain any dirt). The coordinates are given in the format x,y where x is the row index and y is the column index.
   - **Example**: `2,3` indicates that the docking station is located at row 2, column 3 in the house matrix.
2. **Maximum Battery Steps**:
   - **Format**: A single integer or float value.
   - **Description**: This line specifies the maximum number of steps the vacuum cleaner can take before its battery is exhausted. This value determines how long the vacuum can operate before it died.
   - **Example**: `100` indicates that the vacuum cleaner has 100 steps worth of battery life.
3. **Maximum Steps Allowed**:
   - **Format**: A single integer value.
   - **Description**: This line specifies the maximum number of steps the vacuum cleaner is allowed to take during the simulation.
   - **Example**: `1000` indicates that the vacuum cleaner can take up to 1000 steps in the simulation.
4. **House Matrix**:
   - **Format**: A matrix of integers.
   - **Description**: The remaining lines of the input file describe the layout of the house as a matrix. Each element in the matrix can have one of the following values:
     - `1-9`: Indicates the amount of dirt present at that location. Higher values represent more dirt.
     - `-1`: Indicates a wall or out-of-bound area where the vacuum cleaner cannot pass through.
     - `0`: Indicates a clean, navigable area with no dirt.
   - The house matrix may or may not be surrounded by walls initially. If it is not surrounded by walls, the program will check and add walls around the matrix to ensure it is enclosed.

- **Example**:

  With walls:

  -1 -1 -1 -1 -1 -1 -1 -1

  -1 5 -1 -1 -1 -1 -1 -1

  -1 0 -1 0 -1 5 5 -1

  -1 0 -1 -1 -1 0 1 -1

  -1 2 5 3 4 0 6 -1

  -1 -1 -1 - 1 -1 -1 -1 -1

  Without walls (the same example):

  ```
  5  -1  -1  -1 -1  -1

  0  -1   0  -1  5   5

  0  -1  -1  -1  0   1

  2   5   3   4  0   6
  ```

  In this example:

  - The matrix represents a house layout with 4 rows and 6 columns.
  - `1`, `2`, and `3` indicate dirt levels at their respective positions.
  - `-1` indicates walls or out-of-bound areas.
  - `0` indicates clean areas with no dirt.

Example Input File:

Here is an example of how the input file might look:

```
2,3
50
200
5  -1  -1  -1  -1  -1
0  -1   0  -1   5   5
0  -1  -1  -1   0   1
2   5   3   4   0   6
```

*Ensuring the House is Surrounded by Walls

The program will check the house matrix to ensure it is surrounded by walls. If any side of the matrix is not -1 (wall), it will add walls to the top, bottom, left, and right sides. This ensures that the vacuum cleaner cannot move out of bounds during the cleaning process.

- The docking station is located at row 2, column 3.
- The vacuum cleaner has a maximum battery life of 50 steps.
- The vacuum cleaner is allowed a maximum of 200 steps in the simulation.
- The house matrix describes the layout with dirt, walls, and clean areas as detailed above.

Design Considerations

Classes and Their Responsibilities:

1. Algorithm Class:

- **Description:** This class is responsible for controlling the behavior of the vacuum cleaner. It determines the next move of the vacuum cleaner using the sensors that the vacuum has, manages the path to be followed, and keeps track of the steps performed. (decide the next move uses just the 3 sensors and it doesn't know anything about the house structure).
- **Members:**
  - `Vacuum& vacuum`: A reference to the vacuum cleaner object.
  - `std::stack<Direction> Path`: A stack that contain the coordinates that the vacuum has visit (in order) to keep track of the path followed for potential backtracking, and getting back to the station to recharge the battery. (it contains just up, down, left, right- without stay).
  - `std::queue<Direction> steps_Performed`: A queue to record all steps performed for logging and analysis.
- **Methods:**
  - `Algorithm(Vacuum& vacuum)`: Constructor to initialize the algorithm with the vacuum cleaner reference.
  - `void decideNextMove()`: Determines the next move for the vacuum cleaner using sensors.
  - `void backStep()`: Perform one back step to the docking station.
  - `bool cleanAlgorithm()`: Executes the cleaning algorithm by deciding the next step randomly, ensuring the chosen direction is free of walls and the vacuum cleaner has **enough battery to proceed****.

    enough battery to proceed:** The vacuum cleaner continues cleaning as long as the battery level is greater than the number of blocks visited. For example, if the vacuum has visited 10 blocks, it will only continue cleaning if the battery level is more than 10; otherwise, it returns to the docking station to recharge, while the get back step is chosen from the stack that we saved. (the stack has all the directions, without stay- which means the vacuum moved from one block to another).

At the docking station, we check if the mission is complete. If the mission is complete, we return the appropriate outputs; otherwise, we continue cleaning the house after fully recharging.

Note that often you will encounter that the battery exhausted but the mission is succeeded, that is because we made sure that the vacuum should use the maximum that he could until he can't proceed without returning back to the docking station and fully recharge.

  - `std::queue<Direction> getStepsQueue()`: Returns the queue of all steps performed in the algorithm.

2. Direction Class:

- **Description:** This class provides an enumeration for the possible directions the vacuum cleaner can move. It also includes utility functions for direction conversion and output.
- **Members:**
  - `enum class Direction`: Enumeration for the directions (STAY, LEFT, RIGHT, UP, DOWN).
- **Methods:**
  - `Direction intToDirection(int value)`: Converts an integer to a Direction enumeration.
  - `std::ostream& operator<<(std::ostream& os, Direction dir)`: Overloads the stream insertion operator for Direction.

3. House Class:

- **Description:** Represents the environment in which the vacuum cleaner operates. It contains the matrix of the house, the docking station position, and methods to access and modify these attributes.
- **Members:**
  - `std::vector<std::vector<int>> house_Matrix`: Matrix representing the house layout.
  - `int dock_X, dock_Y`: Coordinates of the docking station.
  - `int total_dirt`: Total amount of dirt in the house.
- **Methods:**
  - `House(const std::vector<std::vector<int>>& HouseMatrix, int x, int y)`: Constructor to initialize the house with the matrix and docking station coordinates.
  - `void printHouse() const`: Prints the house matrix. (only for debugging)
  - `bool isWall(int x, int y) const`: Checks if a position is a wall.
  - `int getDirtLevel(int x, int y) const`: Gets the dirt level at a position.
  - `void setDirt(int x, int y)`: Sets the dirt level at a position.
  - `int getDockX() const`: Gets the X-coordinate of the docking station.

    - `int getDockY() const`: Gets the Y-coordinate of the docking station.
    - `int getHouseMap() const`: Gets the house map.
    - `int getTotalDirt() const`: Gets the total dirt in the house.

4. Simulator Class:

- **Description:** Manages the overall simulation. It loads the house configuration, initializes objects, and runs the algorithm to clean the house and returning the appropriate output.
- **Members:**
    - `int dockX, dockY, maxBatterySteps, maxStepsAllowed`: Configuration parameters for the simulation.
    - `std::vector<std::vector<int>> houseMap`: Matrix representing the house layout.
- **Methods:**
    - `Simulator()`: Constructor to initialize the simulator.
    - `void writeOutput(const std::string& outputFile, Algorithm alg, Vacuum vacuum, bool res)`: Writes the simulation results to an output file.
    - `void surroundByWalls()`: Ensures the house is surrounded by walls, and surround it if not.
    - `void loadFromFile(const std::string& filename)`: Loads the house configuration from a file, and the rest parameters.
    - `int run(int argc, char* argv[])`: Runs the simulation.

5. Vacuum Class:

- **Description:** Represents the vacuum cleaner. It keeps track of its position, battery level, and the steps it has taken. It also provides methods to move and clean.
- **Members:**
    - `House& house`: Reference to the house object.
    - `const float max_BatterySteps`: Maximum battery steps allowed.
    - `const int max_Steps`: Maximum steps allowed.
    - `int pos_X, pos_Y`: Current position of the vacuum cleaner.
    - `float curr_BatterySteps`: Current battery steps remaining.
    - `int curr_Steps`: Current steps taken.
- **Methods:**
    - `Vacuum(House& house, float maxBatterySteps, int maxSteps)`: Constructor to initialize the vacuum cleaner.
    - `void move(Direction direction)`: Moves the vacuum cleaner in the specified direction and updates battery and steps.
    - `bool clean()`: Performs the cleaning action.
    - `bool isBatteryExhausted() const`: Checks if the battery is exhausted.
    - `bool reachedMaxSteps() const`: Checks if the maximum steps are reached.

  - `bool isMissionComplete() const`: Checks if the cleaning mission is complete (the total dirt is 0 and we are in the docking station).
  - `int dirtSensor() const`: Senses the amount of dirt at the current position.
  - `bool wallSensor(Direction dir) const`: Checks for a wall in the specified direction.
  - `float batterySensor() const`: Gets the current battery level.
  - `void update_StepsAndBattery()`: Updates the steps and battery level.
  - `int getTotalDirt()`: Gets the total dirt cleaned.
  - `bool atDockingStation() const`: Checks if the vacuum cleaner is at the docking station.
  - `void charge()`: Charges the vacuum cleaner. (single step)
  - `float getMaxBattery() const`: Gets the maximum battery capacity.

Use of STL Containers

The design extensively uses Standard Template Library (STL) containers to manage collections of data:

- **std::vector:** Used in the `House` class to represent the house matrix. Vectors are chosen for their dynamic size and ease of access.
- **std::stack:** Used in the `Algorithm` class to keep track of the path followed by the vacuum cleaner, enabling backtracking.
- **std::queue:** Used in the `Algorithm` class to record the steps performed by the vacuum cleaner, facilitating logging and output generation.

Error Handling

Error handling is implemented to manage various potential issues:

- **File I/O Errors:** The `Simulator::loadFromFile` method checks if the input file can be opened and handles the error by displaying an error message and exiting.
- **Invalid docking station coordinates:** we checked if the docking station coordinates that were given is valid, if not we threw an error message to the user to try again.
- **Dirty docking station :** if the docking station that the user gave is dirty we recover that error by changing the value of dirt in the docking station to be 0.
- **Invalid house blocks values :** we checked that the dirt values are valid (0-9) or -1 if it is a wall, otherwise we throw an error to the user to change the input matrix values.
- **Invalid matrix dimensions:** we checked that the matrix dimensions is valid (m*n), otherwise we throw an error to the user to change the matrix.
- **Invalid maximum battery steps :** we checked that the maximum battery steps is valid (>0), otherwise we throw an error to the user to change the value of it.

- **Invalid maximum steps allowed :** we checked that the maximum steps allowed is valid (>0), otherwise we throw an error to the user to change the value of it.
- **In the Direction enum class in the functions :** intToDirection and operator<< , we checked if a faulty case was given due to a bug in the program.

Alternatives Considered

Several alternatives were considered during the design process:

- **Using Arrays Instead of Vectors:** Initially, arrays could have been used for the house matrix, but vectors were chosen for their dynamic resizing capabilities and ease of use.
- **Single Class Approach:** A single class to manage all functionalities was considered but rejected in favor of a more modular approach to improve maintainability and readability.
- **Different Algorithms for Pathfinding:** Various algorithms for pathfinding and cleaning were considered – like BFS, DFS…. The random-based approach was chosen for its simplicity.
- **Different Algorithms for back Step:** Various algorithms for backstepping like BFS, DFS. The stack-based approach was chosen for its simplicity and effectiveness in this context.

Testing Approach

Testing is crucial to ensure the reliability of the system:

- Added print statements to track the vacuum's movement and house state.
- Verified that the vacuum interacts correctly with the house and performs cleaning and charging as expected.
- Simulation Runs: The complete simulation is run with various input files to test different scenarios, including edge cases like fully enclosed walls and different amounts of dirt.
- Output Verification: The output file is checked to ensure it accurately reflects the steps performed (different output to the same input but with more than run the program – randomly algorithm), the battery status, and the cleaning results.