Testing Approach Overview

For our project, we adopted a test-driven development (TDD) framework using Google Test. This approach emphasizes writing test cases before the actual implementation of the code, ensuring that our tests remain independent of the implementation and are focused on the expected behavior.

Test Design and Implementation

1. **Independent Tests**: Each test is designed to be independent, focusing solely on a specific unit of functionality. This independence allows for more accurate identification of errors in code functionality.
2. **Test Cases**: We developed a suite of 17 test cases that assess both basic functionalities and edge cases. These tests help ensure comprehensive coverage of the code.
3. **Return Values**: Tests verify the correctness of return values:
    - A return value of `0` indicates that the operation completed successfully.
    - A return value of `1` signifies an exception or error during execution.
4. **Output Summary**: Tests evaluate the output summary to confirm whether the mission or operation was successful or resulted in failure.
5. **Exception Handling**: Given that our code throws exceptions in erroneous conditions, our tests are designed to detect these exceptions effectively. By checking for specific exceptions, we can confirm that the code handles error states as expected.

Execution and Code Refinement

After implementing the initial code, we run our comprehensive test suite. This process not only validates the functionality but also helps in refining the code. If a test fails, we modify the corresponding code segment to fix the issue and rerun the tests to ensure that the problem is resolved.

Design and Development Strategy

The success of our testing strategy stems from the initial phase of meticulously designing the structure of the code and outlining expected behaviors. This foundation allows for a more systematic implementation and testing phase, ensuring robustness and reliability in the final product.

Conclusion

This organized approach to testing not only enhances the reliability of the software but also streamlines the development process, making it more efficient and error-resistant.

Design Considerations:

While reading the exercise we concluded that there are three main logical entities in the setting of the assignment, each corresponds to a class in our code:

- The **House** – the space where the vacuum cleaner should work in. The most natural data structure to represent the house structure is a matrix because the optional movements of the cleaner are to immediate neighbors.
- The **VacuumCleaner** – the robot itself which has an updating battery level, a maximal battery capacity etc. We think of the robot as an entity that is bound to a specific House instance and that's why this class is part of the House class.
- The **Algorithm** – to create an independent decision environment, we think of the algorithm as a separate entity which has no prior knowledge of the house structure. Furthermore, the algorithm class will have a simple logical rule – making an instantaneous decision of what operation to do in the next step.

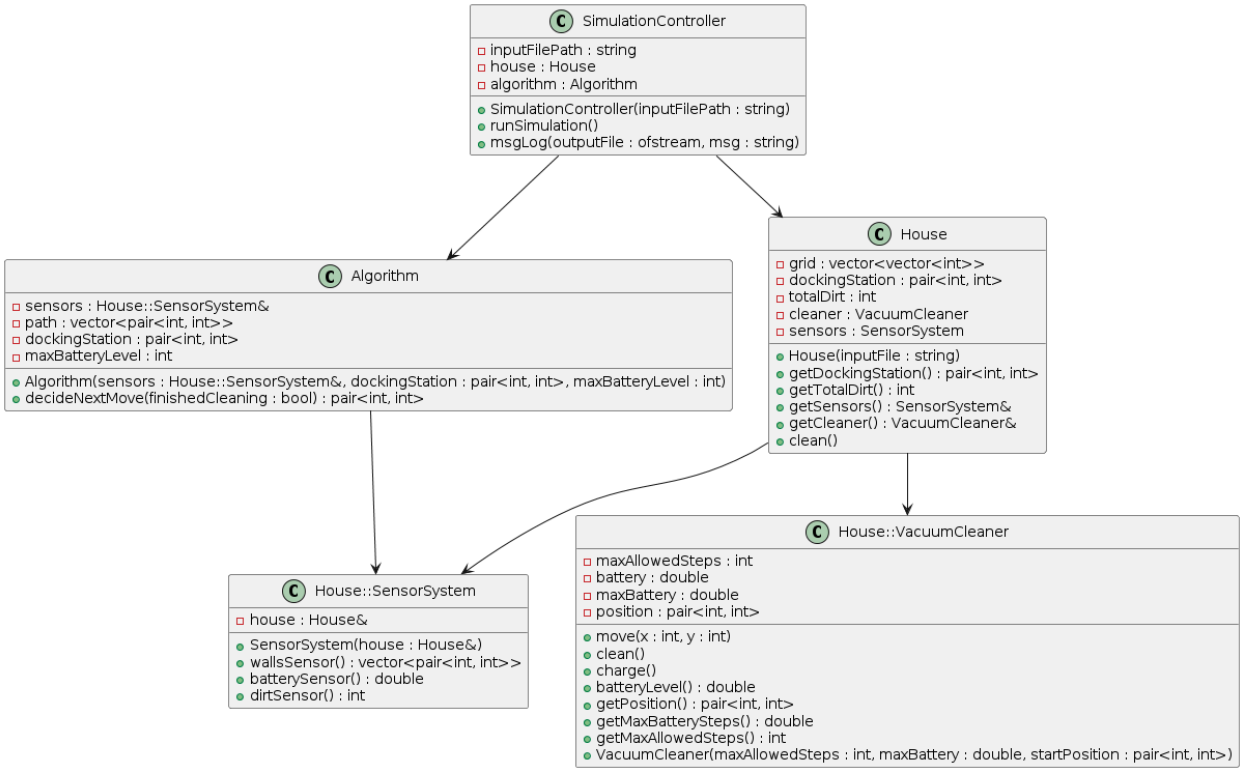
In addition to those, there's a need for an API which will represent the sensors that the algorithm may use when choosing the next step. For that, we have the **SensorSystem** class which is, like the VacuumCleaner class, bound to a specific House instance. This class has only three public function members - which represents the three different sensors the Algorithm can use and indeed, the algorithm code has access to a SensorSystem object only. Of course, those functions could have been implemented directly in the House class, but separating them into another class emphasizes the limited access the algorithm has to the House and lets us have better control over the visibility of the different services the House class supplies.

The only missing part is to wrap the whole step-by-step flow together and update the House and VacuumCleaner properties properly according to the algorithm decision at each step. This is done via the runSimulation function of the **SimulationController** class. The reason for having it is to avoid complex coding in the main function and create a link between a specific simulation and its corresponding Algorithm instance.

The relations between the classes are derived by their semantic relations – a SimulationController is building a House which owns a VacuumCleaner and a SensorSystem, and communicates with an Algorithm which uses the same SensorSystem.

We believe this design is compatible with the description and demands of the assignment while providing relevant entities and correct relations between them.

UML Class Diagram:



UML Sequence Diagram: