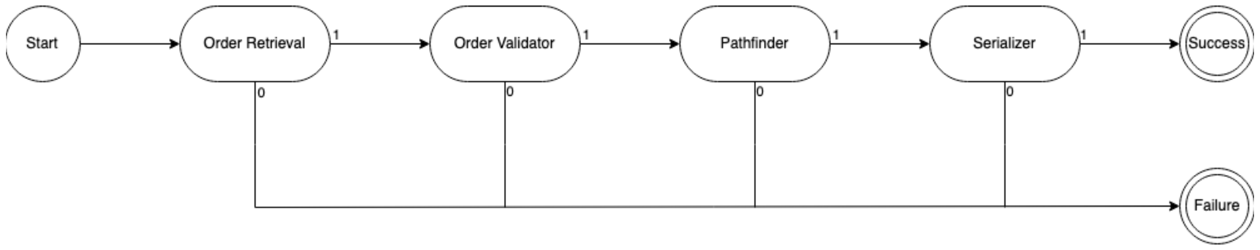
The following document outlines a comprehensive test plan. It is split into two sections: validation and verification to distinguish between tests and inspections.

Testing objectives

Validation

My testing strategies involve a variety of techniques derived from modeling the program as a finite state machine.



The sequences of states are mapped to the various subsystems, where each subsystem has a unique test specification. One goal of testing is to ensure that this FSM conforms to our program, and has the three correctness relations. Therefore, I approach the testing of the system by dividing it into smaller subsystems. I apply the divide and conquer principle[1], in the sense that I divide testing of the entire system in terms of rigorously testing each subsystem's functionality. By ensuring the correctness of the subsystem, as well as the subsystems' interactions with one another, I ensure the correctness of the entire system. This approach allows for a good trade-off between testing efficacy and time, making it relevant to the solo developer-tester context.

Order Retrieval Subsystem:

Testing for **correctness:**
The correctness of JSON parsing. Ensure that JSON to POJO is correct. (Integration Level)

Testing for **security:**
Test that the system only permits SSL connections. This should adhere to the sensitivity principle, a false negative is strictly preferred to a false positive. (Unit Level)

Order Validator Subsystem:

Testing for **correctness:**
Ensure MC/DC coverage. Fully explore the decision tree. (Unit Level)

Testing for **robustness/security:**
Should be robust against malicious/unexpected inputs. Any opportunities for null checking will throw errors in our system. Check for null input exhaustively to combat this. In the future, the

[1] reference divide and conquer section 3

project may include a front-end component, which may or may not implement null checking, therefore we are adhering to the policy of redundancy. (Unit Level)

Pathfinder Subsystem:

Testing for **correctness:**
Since all paths are required to be "valid" and close to "optimal" we require two tests.

Path validity can be tested with the principle of restriction.[2] By restricting the potentially complex problem of solving for geometric intersection, we instead ask the question; is the point we are currently in a no- fly zone? (Unit Level)

Path optimality can be tested with a benchmark method combined with statistical techniques. I create a large number of random paths. I calculate the length of the path and compare it with the length of the true shortest distance. I then try out different benchmarks to see what percentage of the paths will be rejected and which will be accepted. From this I ascertain a reasonable benchmark that I can test with future path generations, as well as research an interesting geometric problem. (Unit Level)

Testing for **performance/scalability:**
The pathfinder subsystem is the most computationally expensive part of the system, due to the inherently large search space, and n^3 time complexity of the visibility graph pre-processing step,complex geometry may bog the system down. The speed of the system as a whole depends on the speeds of its components, especially the pathfinder. (System Level)

Serializer Subsystem:

Testing for **correctness:**
The serializer subsystem must be correct, one area would be checking that the correct angle format correlates to the serialized files. We can learn from NASA's Mars Climate Orbiter crash,[3] to check for unit consistency so it is important to make sure that our angle standard is consistent.

Testing for **robustness:**
Currently, exceptional circumstances such as being out of disk space, or not having sufficient permissions has not been handled yet, which would violate the requirement of graceful failure.

Subsystem-Agnostic

Testing for **memory efficiency:**

[2] reference restriction principle in the textbook
[3] How NASA Lost Its Mars Climate Orbiter From a Metric Error (simscale.com)

Via the utilization of profiling tools, we can inspect where memory usage is highest, and what we can do to reduce it. If memory usage is too high, then that entails either expensive hardware upgrades (buying more server ram) or an unusably slow system (incurring paging), significantly harming the user experience.

Process Checklist:

- [x] ~~Implement all unit tests for the order validator, such that the order validator has full branch coverage. Create a new test runner for each guard clause. Store in a file named OrderValidatorTest or similar. (Time expected: 3 hrs)~~
- [x] ~~Implement null checking for the order validator. Store in a file named OrderValidatorTest or similar. (Time expected: 1 hr)~~
- [ ] Implement at least 10 synthetic JSON files and test for correct POJO translation. Store in a file named JSONSerializationTest or similar. (Time expected: 3 hrs)
- [x] ~~Implement a unit test that verifies that a given connection to a web server is indeed an SSL connection. Store in a file named SSHTest or similar. (Time expected: 1 hr)~~
- [x] ~~Implement a unit test that verifies if a large batch of flight-paths and store them in a file named PathWalkerTest or similar. (Time expected 2 hrs)~~
- [x] ~~Implement a benchmark test that determines the divergence between computed flight-paths and optimal flight-paths, and store the results in a graph named "benchmarks-for-path-optimality.png" or similar. (Time expected: 3 hrs)~~
- [x] ~~Generate a set of synthetic data which increases the complexity of the no fly zones and test the performance of the system. Store the results in a graph named "benchmarks-for-path-performance.png" or similar. (Time expected: 2 hrs)~~
- [x] ~~Profile the memory usage of the path-cacher function and store it in a file named "flamegraph.html" or similar. (Time expected: 3 hrs)~~
- [x] ~~Implement an integration test that checks for consistency between the angles used in the source code and the angles used when serialized. Store in a file named "AngleConsistencyTest" or similar. (Time expected: 1hr)~~

Schedule Review

Test/Result Name	Expected Time (hrs)	Actual Time (hrs)
OrderValidatorTest	3	2.5
OrderValidatorTest (Null Checking)	1	1
PathWalkerTest	2	2
benchmarks-for-path-optimality	3	2

benchmarks-for-path-performance	2	2
flamegraph	3	3
AngleConsistencyTest	1	1
JSONSerializatioNTest	3	NOT COMPLETED

Verification:

Due to a lack of a team, verification activities are to be done concurrently with validation activities. If I am writing a unit test to assess the logic of a function, I will also inspect and review the code to make sure that it is well documented, syntactically correct, and stylistically consistent.
As I go through every function, I fill out the following table:

Function name: fooBar()		
Technique	**?**	**Further comments if applicable**
Consistent formatting (brace on the same line, sensible white spacing, indentation)	✓	Fixed indentation.
Descriptive and camelCase variable names	✓	
Single purpose functions	✓	
Concise comments	✓	

Having a compliant and consistent codebase is done to fulfill the requirement of scalability and maintainability. Rigorous consistency would imply an easier onboarding process for new developers, which would assist with the application's scalability. Furthermore, its predictable structure makes the codebase easier to maintain and improves the speed of debugging.