Gaps and Omissions:

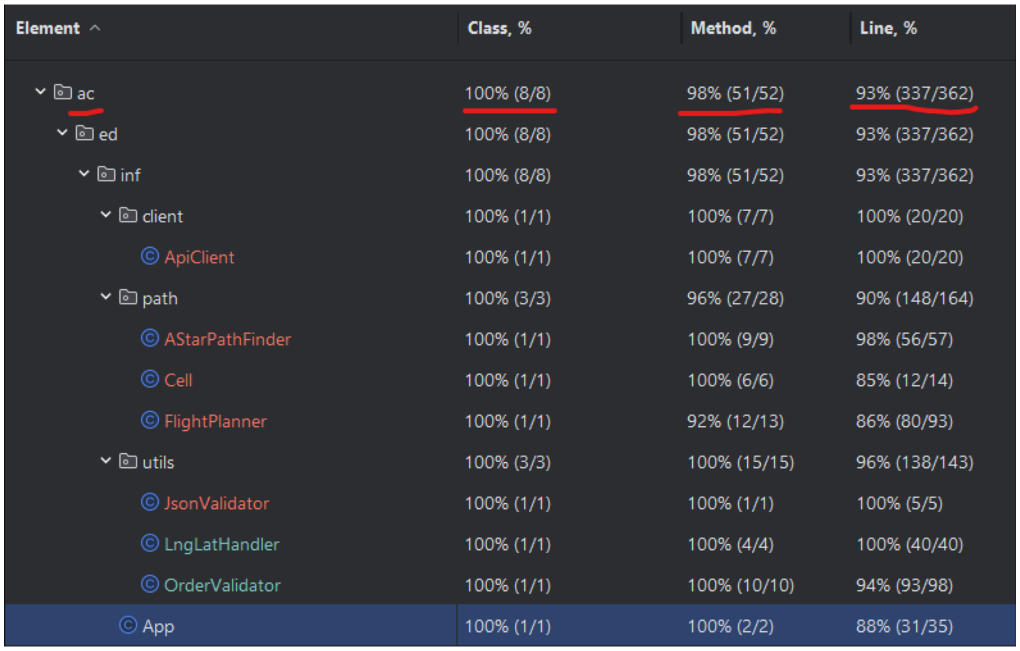
- Testing process generally addressed most functional requirements, however these requirements are not refined to the highest degree. That is, in the case of pathfinder and flight planner modules, there's still a high degree of coupling, blurring the separation in detailed description of the relevant functionality, resulting in worse-defined interface & features for both, which reflects in the precision of testing and could potentially hold bugs, that can only be uncovered by further decoupling the two modules.
- In combinatorial testing in certain integration tests, where a sample order from a sample restaurant are used, and since those were manually designed and created to trigger certain edge cases, they might not be representative of the actual user input or they might be erroneous in certain conditions, such as unforeseen state of the program. This can be addressed by more advanced techniques, such as mutation testing and collected data about user input that caused errors before (these can be collected and programmatically put through in the testing stage of the CI pipeline).
- The testing also implicitly covered the path finding algorithm by assessing the output of the flight planner, rather than covering the actual module. This should be addressed in future iterations, since the design is supposed to be modular and the flight planner shall function with any valid pathfinding algorithm.
- The most important and probably the most sever gap of this project is the lack of proper documentation of the design and the implementation of the project. Must be addressed in the following iteration to simplify the testing process and allow new developers to easily join the project.

Limitations of the given testing process:

Designer bias - since I designed, implemented and tested the code, I could have only identified and covered with tests only what I deemed prone to be erroneous and not identified all the edge cases. The testing also does not account for logical errors, such as boundary condition checking and could potentially be addressed by other techniques, such as mutation testing, further more detailed modelling and combinatorial testing. However, these approaches are to be employed in the future iterations as they would have created much overhead right now.
The design and implementation is not well-documented, which should also be addressed in the future versions of the project.

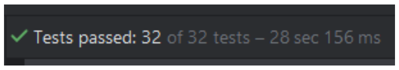
Target coverage

The ideal target was to cover 100% of components, 100% of methods and 100% of lines to ensure robustness and correctness of the functionality. The target was also to ensure that any arbitrary execution performance is under 60 seconds to provide user experience in a timely fashion.
According to the coverage report and the timing of the execution for each of 5 days worth of data from the API, we can see the following figures:



Element	Class, %	Method, %	Line, %
ac	100% (8/8)	98% (51/52)	93% (337/362)
ed	100% (8/8)	98% (51/52)	93% (337/362)
inf	100% (8/8)	98% (51/52)	93% (337/362)
client	100% (1/1)	100% (7/7)	100% (20/20)
ApiClient	100% (1/1)	100% (7/7)	100% (20/20)
path	100% (3/3)	96% (27/28)	90% (148/164)
AStarPathFinder	100% (1/1)	100% (9/9)	98% (56/57)
Cell	100% (1/1)	100% (6/6)	85% (12/14)
FlightPlanner	100% (1/1)	92% (12/13)	86% (80/93)
utils	100% (3/3)	100% (15/15)	96% (138/143)
JsonValidator	100% (1/1)	100% (1/1)	100% (5/5)
LngLatHandler	100% (1/1)	100% (4/4)	100% (40/40)
OrderValidator	100% (1/1)	100% (10/10)	94% (93/98)
App	100% (1/1)	100% (2/2)	88% (31/35)

100% of all classes were covered, meaning that every component has been invoked at least once; 98% method coverage as described in `LO3-Testing.pdf` does not account for certain auxiliary debugging methods; and 93% line coverage that missed some exception handling all signify that the targets have been achieved covering Structural testing. All tests successfully complete as can be seen below, covering Functional testing:



```
✓ Tests passed: 32 of 32 tests – 28 sec 156 ms
```

High relative coverage is good as it identifies dead code and ensures all components and methods are used in some capacity, however it does not guarantee the correctness, which is assessed by other testing techniques.