This section describes the techniques and methods used in the testing of the PizzaDronz application described in `LO1-Requirements.pdf` and further examined in preparation for testing in `LO2-Testing Plan.pdf`. This section also discusses the outcomes and results of the testing and introduces quality and performance metrics and discusses further steps as well as limitation of the testing done so far.

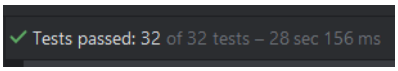
Range of techniques:

- ## White-box Testing
  Test cases, where individual components could selectively be invoked to test a piece of functionality. Allowed to conduct in-depth simulations of various scenarios. Example: `testFileCreated`.
- ## Black-box Testing
  Certain methods and classes were analysed as standalone interfaces and only the exposed certain methods without uncovering the internal logic. Helps to understand the user experience and focus on the delivery of a service. Example: `testCentralArea`, `testInvalidJsonFromServer`.
- ## Functional Testing
  Test cases were designed and implemented based on the functional requirements described `LO2-Testing Plan.pdf`. Bottom-up approach with Unit, Integration and System level tests ensured that the components performed the desired actions and produced correct output.
- ## Structural Testing
  Ensured code quality and correctness of the logic.
  - ### Combinatorial Testing
    Test cases of combinatorial testing work alongside Functional testing. To improve developer experience and reduce overhead.
    Different combinations of valid and erroneous inputs have been used to reach edge cases and ensure correct behaviour.
    Examples: testNextPosition, testInvalidDateArguments, testInvalidNumberOfArguments.
- ## Scaffolding
  - Additional instrumentation has been developed that enabled to reduce the amount of code required for constructing a test case. Notably, the most useful tools were:
  - Generating an order dynamically and then validating/invalidating based on specific attributes desired, for example: instead of manually creating an order with the same details to check if invalidating one attribute triggers an error at the client side, the method returns a blank method that has both UNDEFINED status and validation code. Then, modularly, an order can be assigned In/ValidPaymentDetails, pre-coded Pizzas and Restaurants to match.
  - Mock-up REST server that imitates the actual API. Makes use of the aforementioned methods and can dynamically serve JSON data on specified APIs and gracefully shut down at the end of a test case.
  - Instrumentation has also been developed to dynamically verify that computed flight paths conform to the criteria (closeness and orderliness), so that time can be saved at Integration and System level tests.

Results & Evaluation

During testing, the main loop was rewritten to accommodate for a more convenient and more reliable Exception handling.

Branch testing is not available in Java, and therefore the application code has been adapted to allow for modular testing, ensuring different conditions and execution branches are covered.
Combinatorial testing, where valid and invalid arguments are dynamically put together to ensure adequate behaviour and minimize unexpected outcomes.



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)

A total of 32 tests had been written and compiled successfully producing a 100% class coverage (all components were used at least once), achieving component-based testing; 98% method coverage, which accounts for excluded `noFlyZoneExporter` in FlightPlanner.java, which was used as a debugging tool in the development process, as well as `equals` in the **Cell.java** class of the path finding algorithm, as Java evidently resorts to hashing comparison, rather than the actual content.
93% of lines in all classes had been covered, which is discovered to be certain exception handling logic in the **OrderValidator.java**.

Performance

As required and can be seen in the AppTest.java, lists of orders fetched from the live API were timed and recorded execution times under 2.5seconds for each day, which is way below the required 60 seconds limit, achieving the goal.