# 1 Testing

## 1.1 Methodology

Our testing strategy has evolved from simple "on the run manual testing" to full systematic unit testing of our entire codebase. We followed our project's modular and class-based architecture while designing our testing structure. Each class has a corresponding test class, whose test methods aim to mirror the methods of the class it is testing. However, it is possible that several test methods cover different parts of the same method, or that in turn one test method covers several methods.

Following the white-box testing strategy, each test class is designed by the team members who wrote the original code and who therefore have the best insight regarding the expected behaviour of the original class. Currently we have focused our testing on covering as many branches as possible, each of which is analysed for the different possible behaviours it may have. We are measuring our code coverage by the percentage of statements covered.

As we progress with our development we aim to augment our testing strategy to include component, integration and system tests. Component tests will cover end-to-end cases for each of our modules, while integration tests will cover the connections between each of these components. Finally, system tests will cover the complete project, when all modules have been connected between themselves and the web client.

## 1.2 Implementation

Given that all of our code is written in Python, we use Python's unittest framework. To measure code coverage we are using Python's coverage library, which allows us to run tests on each file separately and combine results to create comprehensive reports.

## 1.3 Results - we need to redo tests

Our current test coverage is 81%. Detailed results from our latest test run covering all of our modules can be found in Appendix C.

## 1.4 Challenges

At this stage, some of the functionalities we are developing have not been factored out optimally, which causes some methods to be very tightly interlinked. This in turn requires more complex tests, some of which will need extensive modification as we restructure the code. We have thus decided to omit some of these methods from our current tests. As we gain more experience with unit testing and refine our design, these methods will be covered.

We also found it difficult to test the graphical output that our modules produce. While testing it manually is very straightforward, we encountered problems while writing tests to do so systematically. One solution we employed was to store images of the graphics and check the values of specific pixels against the expected ones, which we defined beforehand. This approach becomes more complicated when random elements are involved and when the result can vary. We currently handle this by setting specific seeds for the pseudorandom number generators and allowing some error in the test calculations. However, we are still working to make these tests more resilient.