# Testing

## Methods & Approaches

We decided to take a requirements focused view of testing, so that our tests are built around making sure the requirements were being met for the project, but also making sure that we tested key parts of our code that may not directly correspond to any requirement. We decided to use two types of testing, these can be seen below.

As we chose a Java project for Assessment 4, no changes to the testing approaches or methodology were made. The project we chose uses JUnit unit tests, like our original project, the only difference is that we enabled CircleCI automated testing. We decided to keep our original testing report, and adapt it where necessary. We have also kept most of our acceptance tests, as we felt they were still applicable and are appropriately linked to our requirements. The unit tests were taken from the project we inherited, and when needed we continuously added to, modified or removed these tests as we developed the project.

**Acceptance Testing**: this allows us to ensure the end product meets the requirements, and that a user can perform common scenarios successfully.

- Each test takes the form of a list of steps to carry out in the game.
- If a user can perform the list of steps successfully, the test passes, else it fails.
- These tests have to be run manually, because it would take too long to automate them and we have time constraints on the project.
- They are good for testing both functional and nonfunctional requirements.
- Can test functional requirements as the tests can confirm whether the game has the appropriate functionality to meet the requirement.
- Can test non-functional requirements as tests can confirm the project behaves as expected.
- Allows us to detect regressions during refactoring or other code changes.
- We use a **test-driven development** approach here, where the acceptance tests were written before the code, and more tests should pass as more of the requirements are implemented [1].

**Unit Testing**: this allows us to check that our code does what it is meant to.

- Takes into account both normal and edge cases to ensure normal behaviour and boundaries are handled correctly.
- The unit tests are written as the code is being implemented, due to their reliance on the code structure and design decisions.
- We have used both black box testing (doesn't take into account how code works) and white box testing (uses inner workings of code) when designing our unit tests.
- Allows us to detect regressions during refactoring or other code changes
- Written as JUnit unit tests in Java, because it is compatible with our project, easy to use, and has high quality documentation [2].
- These tests are automated, so whenever a commit is pushed to GitHub, the test script is run on CircleCI (our continuous integration server).
- We chose CircleCI for it's reporting abilities, as it provides a html website, and breakdowns of failing tests with debug logs where applicable, making it easy to diagnose problems.
- After the CI runs the unit tests, the result is pushed back to GitHub and updates our team Slack. GitHub is setup to block merging pull requests if a unit test fails.
- Continuous integration helps remove much of the effort required to run tests, and ensures that unit tests are taken into account when developing the project [3].
- These tests also alert us if the project fails to build with each commit, as the project is built by the CI server in order to run unit tests. This can let us know about possible code problems, like syntax errors.

-

Our software development process involves creating new branches for new features, and when a branch is ready, a pull request is created to merge the branch into the master branch. We've setup our pull request flow to block merging if the unit tests have failed, which helps us catch code issues sooner rather than later. Even within a couple of days of starting programming, the testing checks had saved us from merging code that looked good, but was in fact broken in a non-obvious way.

For Assessment 3, we added automated test coverage, using JaCoCo, which determines how well unit tested our project is. JaCoCo is an open source toolkit for measuring Java code coverage, which indicates how much of the code is tested [4]. This has been helpful for identifying which parts of the game are lacking in tests, and we've used the associated data to improve our test coverage. We run test coverage alongside our JUnit tests on CircleCI so that we always have a current statistic that we can use. We continued to use both of these in Assessment 4.

# Bibliography

[1] AgileData.org - Test Driven Development [Online] http://agiledata.org/essays/tdd.html [Accessed 16/02/2017]

[2] JUnit [Online] http://junit.org [Accessed 16/02/2017]

[3] Thoughtworks.com - Continuous integration [Online] https://www.thoughtworks.com/continuous-integration [Accessed 16/02/2017]

[4] JaCoCo [Online] http://www.jacoco.org/jacoco/ [Accessed 16/02/2017]

# Test Report

## Overall Statistics

**52 tests** **0 failures** **100% successful**

## Unit Tests

Below are the statistics generated by our unit test runner on CircleCI for our Assessment 4 code. These tests are functions within the code (although they don't get compiled into the final executable) that tests a small unit of code at a time. They are fully automated, and were automatically run when the appropriate commit was made to GitHub.

**23 tests** **0 failures** **100% successful**

All of the unit tests in the project have passed, which indicates the absence of bugs in the tested code sections. More comments on what this result means are on the next page. The unit tests have test IDs indicated by 1, followed by their index in the list.

See Appendix E for a full listing of the unit tests.

## Acceptance Tests

We have manually run through our acceptance tests to check the game works as intended, and to verify that the game fulfils the requirements. The results are shown below.

**29 tests** **0 failures** **100% successful**

More comments on what this result means are on the next page. The acceptance tests have test IDs indicated by 2, followed by their index in the list.

See Appendix D for a full listing of the acceptance tests.

## Test Information

The tests are associated with an appropriate requirement to allow for traceability, and to check that the code meets any associated requirements. Not all requirements have associated tests, and vice versa - this is because some requirements cannot be explicitly tested, and some tests do not link directly to a requirement but are still needed to ensure the code functions as intended.

There is a criticality measure against each test, for both acceptance and unit tests - this is to represent how important the test is to the overall function of the code. Criticality is on a scale - high criticality means that if that test fails, the project will not function at all; low criticality means that if the test fails, the project will still mostly work as intended.

## Results & Evaluation

The acceptance and unit tests for this project all passed.

The test completeness is not perfect, nor is it possible to be. The unit tests only check their section of the code works as intended, and doesn't cover the integration of that code, or it's use within the entire project. What the unit tests do indicate, is that the specific code that they test works as intended, and they indicate the absence of bugs in that specific code. When combined with the acceptance tests, the test completeness is improved (but still not perfect) as not only are key functions of the code tested, but the overall product is tested to ensure it meets the requirements.

This project doesn't have perfect test correctness either. The unit tests could have bugs in them meaning bugs could be missed in the code, or the unit tests may not cover every edge case or normal use case that exists, which means bugs and issues could slip by undetected. The same sort of thing happens with acceptance testing, as typically a limited range of scenarios are tested, which may not account for the all of the very many possible ways of using the game. Also, the acceptance tests included in this project need to be run manually, which means that human error could occur and affect the overall correctness of the tests.

To improve our testing **completeness** and **correctness**, additional types of testing would be useful additions, such as fully automated end-to-end testing, or integration testing, along with more tests of any type. Introducing different types of tests would alert developers about the presence of a different kind of bug, which would allow identification of bugs that have previously been uncaught. Adding more tests would also decrease the chances of code regressions being missed, or other issues or bugs being missed during the testing process.

# Testing material

Additional testing material can be found on the website (http://www.lihq.me).

This comprises of:

- Executable test plan: http://docs4.lihq.me/en/latest/Assessment4/executableTestPlan.html
- Test design & results:
    - Acceptance tests: http://lihq.me/Downloads/Assessment4/AppendixD.pdf
    - Unit tests: http://lihq.me/Downloads/Assessment4/AppendixE.pdf