Test Methods

Our tests are implemented via JUnit and associated libraries and are clearly indicated within the code, as any class ending with “test” contains tests only for its related main class. RoboticonTest contains all the tests for the Roboticon class. This structure allows for easy tracing of errors both in your Java IDE and continuous integration platform of choice.. Our tests were implemented such that:

- Every method (That is not a Setter or a Getter or part of the UI structure) has an associated test, where appropriate methods may have Valid, Invalid and Boundary value tests to ensure they operate and respond correctly.
- All the requirements expected at this point in the project are tested, be this with separate tests or the tests mentioned above.

Our Tests will be considered at an acceptable state when all unit tests are passing and all other tests of implemented functions are passing.

Functional Testing

Unit Testing, All methods and classes were unit tested as mentioned above. Unit tests are located in the classes ending with ‘Test’. These unit tests will always be applicable to the project as it develops and can be modified as necessary if the architecture develops.
Integration Testing, Some of the unit tests required multiple classes to be performed, this makes them essentially unit tests **and** integration tests due to the fact they’re testing multiple classes at once whilst checking the correctness of a single method. Dedicated integration tests separate from the unit test directory were deemed unnecessary as the system testing is the next logical step up for our use case.
System Testing, Whilst not fully implemented as true black box system testing, the system is regularly deployed to check the UI is displaying correctly. And whilst not formal or complete, the system is deployed on a regular basis.
Acceptance Testing, Not included as of yet due to the stage of development we currently sit at.

Non-Functional Testing

Performance, As with system testing, basic performance testing has informally occurred throughout development under the expected use case.
Compatibility, throughout development our team members have been informally deploying on both macOS, Windows and Linux environments.

The full test suite is provided alongside the the code and is clearly distinct thanks to our code structure.

Documentation for the Unit Tests is provided alongside each test. Essentially each test maps to a corresponding function it is testing the correctness of so test<Function name> in the class test<Class name> is testing that function in that class.

Requirements testing has been carried out throughout the development. Since our requirements were listed as testable statements this is very simple. A developer merely needs to go through the list of Requirements and test whether or not each requirement has been implemented fully. This is done by multiple developers across multiple systems for compatibility testing.

Sources:
All Functional and Non-Functional Testing definitions were sourced from:
https://www.inflectra.com/Ideas/Topic/Testing-Methodologies.aspx

Test Report

The Unit tests and Integration test results can be seen at:

https://teamfractal.github.io/assessment3/reports/tests/index.html

We ran 31 Unit tests, the majority being true unit tests, some being modified unit tests which tested a single method but required other objects to be correct to function properly. These tests were required to have 100% pass rate, otherwise internal logic would be broken potentially jeopardising the game.

Qualitative test link:

https://teamfractal.github.io/assessment3/RequirementsTesting.pdf

All qualitative testing passed and can therefore the game can can be considered to have fully implemented the requirements.

Alongside this test suite it should be noted that some requirements were implicitly met, we will list them here so as to avoid any confusion and state why we believe them to be met:
2.a) The map displayed in the game meets the specification set