Software Testing Report

The unit testing was done using JUnit since the game is implemented in Java and is a built in feature in Eclipse. NFR2 requires that the game can run on both Linux and Windows which it does as validated by playtesting.

Some testing was done throughout the implementation period. This was mostly white-box unit testing and integration testing as additional features and code were added. Test classes were created - e.g. CollisionTest, ScoreComponentTest - and used as test materials to ensure current code was in working order before further features were implemented. This form of testing was applicable during development because it was needed to establish that the game was working well enough to add new features, and to ensure that logical classes were working as intended, as many are linked in how they operate. It is also fairly quick and easy to test the system throughout implementation, as any recently written code is run to check that the result is as expected.

Some acceptance testing was actually carried out by members of the group as well. This was done in a black-box format and consisted of playing the game whilst trying out different scenarios implemented by another group member such as killing an enemy, picking up health/ammo, colliding into walls and other entities, and dying (ending the game with 0 health). This form of testing was suitable as it allowed group members who hadn't implemented certain features of the game judge, in a setting similar to a user playing the game, whether the feature had sufficiently fulfilled the requirement.

The majority of testing was done after implementation, through unit-testing. Because of the way our code is structured - all the logic contained in system classes, and data contained in component classes - the isolated nature of each class lends itself to effective unit testing after implementation. These tests were fully planned before being carried out; a spreadsheet was created with a full list of implemented classes, which were marked, regarding tests, as either 'unneeded', 'needed', 'in progress' or 'done'. This method of planning was used because it ensured a thorough coverage of the code, as well as a log of which tests were being carried out, finished, or still needed. The testing could then be carried out by whoever was available; if a test was 'needed', they could pick it up and set it to 'in progress'. Other group members would then know to not pick up that particular test. This plan allowed a fair amount of flexibility especially when group members in charge of testing were not available to work in the same room.

The test environment was creating through forking the GitHub project; testing could then be carried out without danger of accidentally adversely affecting the code - i.e. by forgetting to reset a constant which had been temporarily changed for testing purposes. This effect was made further reliable by GitHub's tracing of changes. By thoroughly and comprehensively testing the implementation, which was based around our requirements, we ensured that testing covered the requirements implemented in assessment 2.

Unit tests

Test Class	**Pass/Fail**
AABBTest	Pass
CircleTest	Pass
CollisionTest	Fail
IntersectionTest	Pass
PatrolRouteComponentTest	Pass
PositionComponentTest	Pass
ScoreComponentTest	Pass
WeaponComponentTest	Pass
EntityEngineTest	Pass
EntityManagerTest	Pass
EntityTagManagerTest	Pass
EventManagerTest	Pass
SystemManagerTest	Pass
CollisionEventsTest	Pass
EnemyCollisionTest	Pass
PlayerCollisionTest	Pass
SearchWorldCollisionTest	Pass
AmmoPickupTest	Pass
HealthPickupTest	Pass

CollisionTest fails test on two specific methods: circleToAabb() and getCloserAndFurtherPoints(). These failures are both due to a bug in closerAndFurtherPoints(), which was discovered late in development and we were unable to fix. This method should find the two points on a box which are closest and furthest respectively from the circle. However, in some circumstances, it will return two identical points. circleToAabb() relies upon the results from getCloserAndFurtherPoints() to detect intersection between these shapes, and this bug will sometimes lead the logic in this method to believe that there is no intersection even when there is. However, this bug only occurs under very specific circumstances, and is in practice never visible during normal gameplay. All provided tests should pass upon fixing of this single bug.

Acceptance testing

We also ran regular play sessions of our game with the whole team. The implemented functionality of the game would be described to team members focussing on other portions of the project, and then they would play the game as if from the eyes of a new player. This helped ensure that the game played in a way that a new player could follow. We formulated a table of tests that could be executed with regularity, capturing a list of currently implemented functionalities, which is shown on the testing report on our website. These reflect the currently implemented requirements from our requirements document.
Our current testing schedule passes all tests.

Website testing material viewable at http://teal-duck.github.io/teal-duck/tests.html