Testing Methods:

The extent to which we preserved the actual testing suite we inherited was fairly limited due to the relatively large-scale scope of our changes. As well as many additive changes, much of the structure of the code was altered by further corrective and perfective changes. Where the logic used on the inputs from the tests already performed by Team Muscovy was not affected, these tests were not considered necessary to repeat - for example, we did not alter the collision code which worked well upon inheriting the game. We were able to preserve the testing suite where the code logic was not changed. Where our code changes had affected the logic of the systems observed in these tests, or introduced new code which interacted with the already-tested functionality, we repeated the test. This ensured the previously written logic would continue to work correctly even after changes to other areas of the code.

With regards to our testing methods, we maintained the dynamic format of testing – with some changes. We noticed that some of the input testing was done with irrelevant numbers: for example, they included a test for 0.0000000001 touch damage in their Enemy_AttackType test. This test was based on Java's inherent rounding rules, was expected to fail and had no bearing on the overall result of their testing as they would never have used a number this small as their touch damage value. Therefore the test was not necessary. We wanted to ensure maximum effectiveness of our testing, while limiting its cost to the lowest possible [1], so we did not include similar tests in our updated testing suite. Our tests were designed not to be exhaustive – an unworkable strategy for the time frame and cost – but complete in terms of our universe of system inputs [1].

Due to the unfinished nature of and lack of large-scale changes to the software, testing had not previously utilised any regression testing. As our code was extensively changed, regression testing was required throughout. We used it as a safety measure against our changed code possibly adversely affecting functionality which had already been tested. Our regression testing was performed alongside development; in the cases where regression testing required tests which we did not formally redo, if the result was the same as in the test previously carried out it was considered to pass the regression test. This was often left to the discretion of the team members in the general playtesting throughout development. In all other cases, our regression testing overlapped with the tests we repeated, and so was thoroughly done.

As previously, we used a mixture of grey box and white box testing to evaluate our software's correctness. Grey box testing was carried out by the members of the group whose duties had mostly been documentation; their knowledge of the code was more basic than others'. Inputs were not changed as extensively as they were in the previous test suite, for the reasons detailed above; only the necessary changes were made. These included changes to inputs which would demonstrate that changing a variable resulted in the correct, expected effect. They were kept within the set of reasonable inputs so as to reduce our testing costs whilst showing that our system would work correctly when played standardly.

We used the game's requirements as our first port of call when defining the tests to be done. After ensuring we would cover all requirements with our range of tests, we refined them using our implementation. This allowed us to systematically review the completeness of our tests with regards to both requirements and actual implementation; often, implementation would create additional need for tests which we could not discern from the requirements alone. For example, requirements was unspecific in detailing acquirable abilities, while after implementation we knew exactly what to expect from each ability and so could devise comprehensive tests.

Testing Report

Code/Methods and classes being tested	Test Number	Description/Procedure	Result	Action Taken
Enemy_AttackType	1.0/1.1	This is testing how an enemy damages a player depending on what attackType they have.	Passed	N/A
Enemy_MovementType	1.0/1.1/ 1.2	This is to test the movement for the	Passed	N/A

		different types of enemies. Some don't move at all, some move randomly and some follow the player.		
Enemy_ShotType	1.0/1.1/ 1.2	This is to test the enemies different shotType	Passed	N/A
Player _PlayerShotType	1.0/1.1	This is to test that the player projectiles are working as they should.	Pass	N/A
PlayerCharacter _AttackInterval	1.0	This tests that the Rate of Fire pickup works and that changing AttackInterval behaves as expected.	Pass	N/A
Obstacle_Damage	1.0/1.1	Tests that when the player walks into a dangerous object, the correct amount of health is taken from the player, and that enemies are undamaged	Passed	N/A
Quit_Game	0.1	By pressing “P” the game is paused and once pressed again, the game becomes unpaused.	Pass	N/A

Link to Testing Material:
http://teal-duck.github.io/teal-duck/Assessment%203%20files/TestingMaterial.pdf

Bibliography

[1] Vegas, S., "Identifying the relevant information for software testing technique selection," in *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on* , vol., no., pp.39-48, 19-20 Aug. 2004