Testing Methods

In order to fully test our system we will need to test it at various different levels (e.g. unit tests and manual testing). We plan to use JUnit to unit test the methods and classes within our code and will follow-up with further manual tests, since uni testing alone is not detailed enough to encompass our entire system and its requirements; some requirements may be too broad to adequately unit test but will fit easily into a manual test (e.g. visually seeing textures change on moving objects). It'll also be necessary for us to use a mocking framework, like Mockito, in order to implement the functionality of certain dependencies without needing to fully instantiate it along with all of *its* dependencies and so on. An example of this would be our GameScreen class; when instantiating fire trucks to test their behaviour in the unit tests, we'll need to create a "mock" instance of GameScreen in order to successfully add an instance of a truck to the test, but we don't *actually* require any of the functionality of the GameScreen class itself. This will save memory and time when running the tests. Returning to manual tests, we envision needing to test most graphics and screen changes via this method by running the software, attempting to perform various actions and recording how the system responds. We'll then compare these outputs with a list of "expected" outputs to verify whether the tests passed.

When designing our tests, we plan to use functional requirements which have been derived from our user requirements to ensure that we are testing the most relevant and important parts of our system. Some tests very obviously need to be included, like the movement of objects in the game, and rendering of textures/sprites. We plan to use a Traceability Matrix to track all of our tests and measure them against the functional requirements of our game, ensuring that we have every requirement accounted for. The matrix will need to include, at a bare minimum, the inputs for each test, the expected outcome, and the actual outcome.

We have chosen to use Statement Coverage as a measure of our testing. Statement Coverage allows us to check for code that doesn't function as intended, as well as find unused statements to make our code more efficient [3]. It is a White Box method [1] as it is focused on the internal workings of the software and how different statements interact with each other. Branch coverage is a testing method that would allow us to test all possible branches within our code and ensure that all methods and conditions perform as expected [4]. However, we have chosen not to use this testing method explicitly as, providing we have a high percentage of Statement Coverage (over 70% for both methods and lines), the majority of branches should be covered by this. We can use IntelliJ to calculate the Statement Coverage for us upon completion of all our unit tests, giving us a good idea as to how many of each class's methods we've tested, the percentage of lines of code in each class that have been tested etc. If any classes stand out with a low Statement Coverage, we can more easily identify them and provide extra tests to cover them, ensuring our system performs as we expect it to, as much as possible.

Test Report

From our testing design, we created tests that met the requirements we had come up with. The unit tests are grouped between three test classes; FireTruckTest, FireStationTest and FortressTest, each testing the main functionality of FireTruck, FireStation and Fortress classes respectively. We also have several manual tests that we conducted that we thought would be more suitable than unit tests due to the nature of the features being tested.

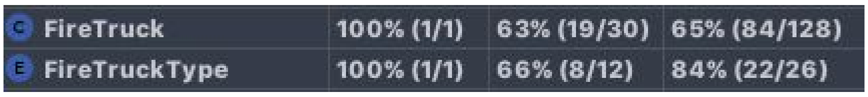
These manual tests and results can be accessed here:
https://bensilverman.co.uk/kroy-assessment3/assessment3/ManualTests.pdf
To view a more in-depth view of our tests, you can see our traceability matrix here:
https://bensilverman.co.uk/kroy-assessment3/assessment3/TraceabilityMatrix.pdf
You can see our test results and download our overall test coverage at these two links:
https://bensilverman.co.uk/kroy-assessment3/assessment3/TestResults/
https://bensilverman.co.uk/kroy-assessment3/assessment3/TestCoverage.zip

Unit Tests

Test 1 - FireTruckTest
Outline: This test class covers; making sure all of the specified stats of a truck (e.g. speed, reserve, attack points, etc) are unique compared to the other type of truck, whether our truck can attack fortresses and whether it can move around our map.
Test categories:
As there are many tests within each test class, we grouped tests together and gave them an ID which can be used to identify them in our traceability matrix:

TRUC_SPEED show that trucks can move at different speeds
TRUC_VOLUME show that trucks can hold different levels of water
TRUC_HEALTH show that trucks have a unique maximum health
TRUC_RANGE show that trucks have different attack ranges
TRUC_ATTACK show that trucks can attack a fortress
TRUC_MOVE show that trucks can move between tiles

Pass/Fail: 24/24 (100%)
Coverage:



FireTruck	100% (1/1)	63% (19/30)	65% (84/128)
FireTruckType	100% (1/1)	66% (8/12)	84% (22/26)

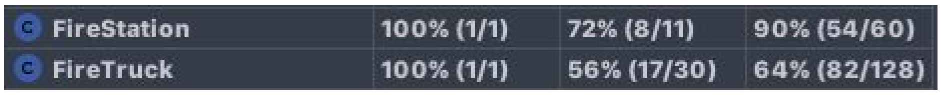
Comments: There are certain methods of the FireTruck class that are used and accessed only by the FireStation and Fortress which are tested in their own Test classes, so we are happy with the line and method coverage of FireTruckTest. It is apparent that 100% of our tests succeeded, however when looking deeper into the coverage reports, there are certain if statements that are not fully explored during testing, therefore more tests could have helped simulate even more eventualities. On the other hand, any ‘draw’ methods such as ‘drawPath()’, which render items to the screen, cannot be tested in unit tests but are key for any manual tests we perform as the common ‘checks’ for such tests are visual.

Test 2 - FireStationTest
Outline: This test class covers; repairing, refilling and making sure the trucks don't crash into each other.

Test categories:
As there are many tests within each test class, we grouped tests together and gave them an ID which can be used to identify them in our traceability matrix:

STAT_REPAIR shows that the trucks can be repaired when at the fire station
STAT_REFILL show that the trucks can be refilled when at the fire station
STAT_COLLIDE check that trucks cannot overlap or occupy the same tile.

Pass/Fail: 8/8 (100%)
Coverage:



FireStation	100% (1/1)	72% (8/11)	90% (54/60)
FireTruck	100% (1/1)	56% (17/30)	64% (82/128)

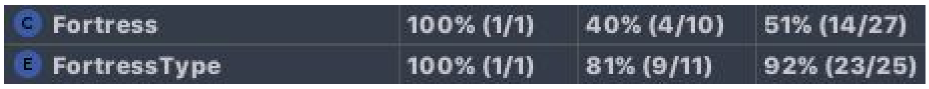
Comments: Again, 100% of tests passed, and looking through the coverage report suggests that the methods not tested are: two getters, the method to draw the station and another method to remove a truck from the trucks list. These four methods are all called in GameScreen which, we do not unit test, and have basic functionality so we are not too bothered by them not being run during our tests.

Test 3 - FortressTest
Outline: This test class covers; checking that each fortress has a unique stats (e.g. range, attack points, health, etc) and that each type of fortress can deal a specific amount of damage to a fire truck if it is within range.

Test categories:
As there are many tests within each Test Class, we grouped tests together and gave them an ID which can be used to identify them in our traceability matrix:

FORT_HEALTH show that fortresses have a unique maximum health
FORT_RANGE show that fortresses have different attack ranges
FORT_RATE show that fortresses attack trucks at different rates
FORT_ATTACK show that fortresses deal different damage to trucks
FORT_ATTACK_WALMGATE,FORT_ATTACK_CLIFFORD,FORT_ATTACK_REVOLUTION,FORT_ATTACK_TRAINSTATION,FORT_ATTACK_MINSTER,FORT_ATTACK_SHAMBLES show that each type of fortress can deal a certain amount of damage to a truck and only do so only within a certain range

Pass/Fail: 32/32 (100%)
Coverage:



Fortress	100% (1/1)	40% (4/10)	51% (14/27)
FortressType	100% (1/1)	81% (9/11)	92% (23/25)

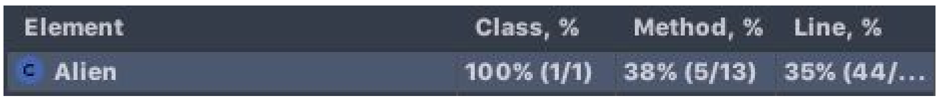
Comments: 100% pass rate, very high method and line coverage. There is only one out of four branches of if statements that are not covered when looking at the coverage report, however to make tests complete we should aim to cover all branches next time.

Test 4 - AlienTest
Outline: This test covers: check left/right/up/down movement and collisions with fire trucks along patrol routes.

Test categories:
As there are multiple tests testing similar things within each Test Class, we grouped the tests together and gave each of the groups a unique ID so we can easily identify them in our traceability matrix:

ALIEN_MOVE tests that aliens can move in all four directions (left, right, up and down).
ALIEN_COLLIDE tests whether aliens should collide with fire trucks when on their patrol path.

Pass/Fail: 6/6 (100%)
Coverage:



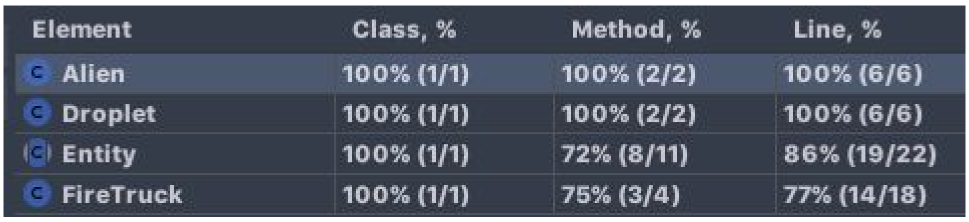
Element	Class, %	Method, %	Line, %
Alien	100% (1/1)	38% (5/13)	35% (44/...

Comments: 100% pass rate, but our method and line coverage could be improved. 35% and 38% are quite low for line- and method-coverage respectively but, upon looking over the code that was not covered in the tests, it's all comprised of getters and graphics-based methods, which would be hard to test via a unit test anyway. In order to ensure these methods work, we will need to add them to the manual test plan.

Test 5 - MinigameTest
Outline: This test covers: comparing an entity's position to its Rectangle position, left and right fire truck movement, alien movement and water droplet movement.

Test categories:
As there are multiple tests testing similar things within each Test Class, we grouped the tests together and gave each of the groups a unique ID so we can easily identify them in our traceability matrix:
MINI_RECT tests that an entity's Rectangle has the same position as the entity position.
MINI_TRUC_MOVE tests the fire truck's ability to move left and right on the screen.
MINI_ALIEN tests that aliens move down the screen upon instantiation.
MINI_DROP tests that water droplets move up the screen upon instantiation.

Pass/Fail: 5/5 (100%)
Coverage:



Element	Class, %	Method, %	Line, %
Alien	100% (1/1)	100% (2/2)	100% (6/6)
Droplet	100% (1/1)	100% (2/2)	100% (6/6)
Entity	100% (1/1)	72% (8/11)	86% (19/22)
FireTruck	100% (1/1)	75% (3/4)	77% (14/18)

Comments: 100% pass rate, with really good method and line coverage for all minigame entities! The only code not covered were getters, and a method changing the fireTruck's texture. We will, however, need to manually test the code within the MinigameScreen class itself, as it will be easier to visually confirm whether the tests pass via that method.

References

[1] https://www.tutorialspoint.com/software_engineering/software_testing_overview.htm

[2] https://smartbear.com/learn/automated-testing/software-testing-methodologies/

[3] https://www.guru99.com/code-coverage.html#4

[4] https://www.tutorialspoint.com/software_testing_dictionary/branch_testing.htm