Testing

Mozzarella Bytes | Team 18

Assessment N°2

Daniel Benison

Elizabeth Hodges

Kathryn Dale

Ravinder Dosanjh

Callum Marsden

Emilien Bevierre

Testing Methods

In order to fully test our system, we will need to test it at various levels, including at unit and system level. We plan to use JUnit to test the methods and classes within our code, however this is not enough to test the whole system because unit tests are not as realistic as a system test needs to be. This is because there is no test as to how well the different methods and classes work together. It may be necessary for us to use some form of mocking to allow us to test each class independently without needing to create instances of dependencies such as Screens and Kroy (Game), which we are not testing. We will also need to perform manual tests by running the software, attempting to perform various actions and recording how well the system performs. Most graphics and screen changes will have to be tested this way.

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. A Traceability Matrix is a good method that we plan to use to measure our tests against the functional requirements and ensure that we have every requirement covered.

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.

Test Report

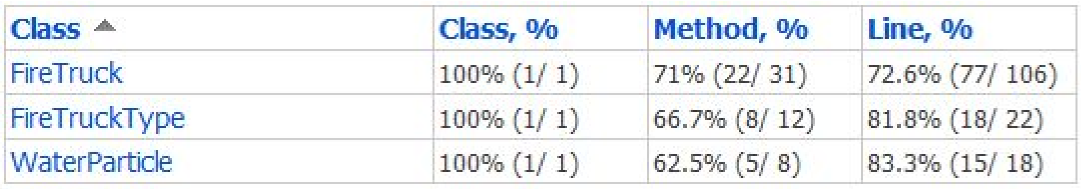
From our testing design, we created tests that met the requirements we had come up with. The unit tests are grouped between three test classes; FireTruckTest, FireStationTest and FortressTest, each testing the main functionality of FireTruck, FireStation and Fortress classes respectively. We also have 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://emhodges.github.io/SEPR-game/assessment2/ManualTests.pdf
To view a more in-depth view of our tests, you can see our traceability matrix here:
https://emhodges.github.io/SEPR-game/assessment2/TraceabilityMatrix.pdf
You can see our test results and download our overall test coverage at these two links:
https://emhodges.github.io/SEPR-game/assessment2/TestResults/
https://emhodges.github.io/SEPR-game/assessment2/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.
Requirements Satisfied: FR_FIRE_TRUCKS, FR_TRUCK_ATTACK, FR_MOBILITY
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: 18/18 (100%)
Coverage:



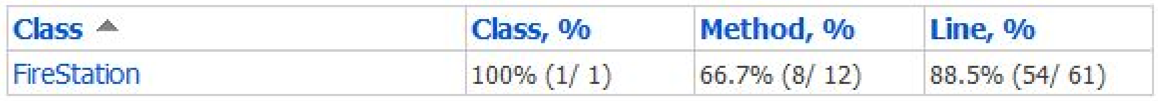
Class ▲	Class, %	Method, %	Line, %
FireTruck	100% (1/ 1)	71% (22/ 31)	72.6% (77/ 106)
FireTruckType	100% (1/ 1)	66.7% (8/ 12)	81.8% (18/ 22)
WaterParticle	100% (1/ 1)	62.5% (5/ 8)	83.3% (15/ 18)

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.
Requirements Satisfied: FR_REPAIR_REFILL, FR_MOBILITY
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 show 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:



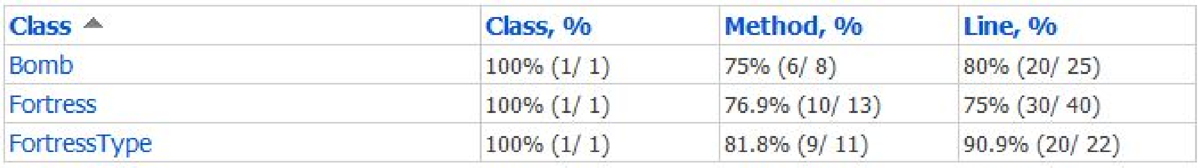
Class ▲	Class, %	Method, %	Line, %
FireStation	100% (1/ 1)	66.7% (8/ 12)	88.5% (54/ 61)

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 ran turing 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
Requirements Satisfied: FR_FORTRESS, FR_FORTRESS_ATTACK, FR_AI
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 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: 16/16 (100%)
Coverage:



Class ▲	Class, %	Method, %	Line, %
Bomb	100% (1/ 1)	75% (6/ 8)	80% (20/ 25)
Fortress	100% (1/ 1)	76.9% (10/ 13)	75% (30/ 40)
FortressType	100% (1/ 1)	81.8% (9/ 11)	90.9% (20/ 22)

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.

System Test:
In order to test the overall functionality of the project, we had an independent third party play the game [5] and asked them to tick off the different functions that they believed the game met on a copy of our requirements. This helped to ensure that we had covered all features, and also was a measure of whether the game was genuinely enjoyable for people to play. It was highlighted to us that the way in which the trucks are moved, while being generally effective, was slightly difficult around corners and so if possible we should try to allow for a margin of error when the user draws their path.

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

[5] http://softwaretestingfundamentals.com/system-testing/