

Change Report

Team 12

February 2020

1 Formal Methods

Change Management involves a transition of people, groups and projects from their current position to a different one. Adaption of the new project may require adjusting of the roles played by each team member to improve the chances of meeting the overall objectives. Our team decided to go with Lewin's Change Management model which consists of three stages. 'Unfreeze' is the first stage which addresses preparation for the change in order to ensure team members remain motivated in completing the required task. The second stage is the change itself, where effective communication and plans are made to understand the new project and figure out what needs to be developed. The final stage is 'Refreeze' taking place after a period of time in which the workings of the team become stable and a normal routine is established [1]. We felt that this model was appropriate to adapt as it gave a directional indication into how to approach the transition.

1.1 Deliverables

The strategy our team developed in order to approach the change in deliverables was to first analyse what requirements are present and what yet are to be addressed. Once identified, the deliverables are split and each team member is assigned the responsibility of their implementation. The next part is to maintain a communications plan which as our chosen method states, is to have regular meetings. The purpose of these is to discuss ongoing changes, updates and to highlight any problems a team member may be facing in order to effectively deal with them.

1.2 Documentation and Code

Our team has appropriately handled the change in documentation by first reading and improving it where necessary, after which adding more in areas of code not documented. We have then maintained the same style of documenting code in our implementation to ensure consistency. To address the handover in code, we first applied the refactoring mechanism in order to understand the existing code and identifying any bugs before adding new implementation to complete the game. This process involved making sure that the program is split appropriately to separate and relevant classes that contain methods or removing unnecessary ones while simultaneously checking that the variable names are suitable, as well as disposing parameters if there are too many within the methods. [2]

The final process vital in the transition of code is testing. As the tests were already created for the implementation that was handed over, the main role involved inspecting that they address the functionality present in the whole program and adding more if that's not the case. We will then produce the complete implementation and complementary test cases.

2 Changes Made

2.1 Testing Report

2.1.1 Methods and Approaches

Our team's initial software testing approach consisted of white and black box testing which include a combination of J Unit, System as well as manual tests. In addition, mocking was used in order to test different classes in the code that have dependencies present. Line coverage was also utilised to evaluate the effectiveness of our tests. The team whose project we have taken on applied the same methods extensively, hence we have decided to make no changes to the testing technique mentioned in our previous assessment [found here](#), except for improving testing design by adding input and expected result for automated test cases of new class additions

2.1.2 Testing Material

[Testing Materials](#)

2.1.3 JUnit and Automated Tests

For the unit tests that were already present, namely FireTruckTest, FireStationTest and FortressTest had to all be modified. These changes were directed towards the variable names as we did not find them to be instructive enough. For example, a 'speed' parameter was used previously to define a firetruck type and this has been replaced by one of the firetruck types predefined in the FireTruckType class to use for testing purposes. Some methods in the Blaster particle class such as createTargetPosition() and updatePosition() we felt, aren't suitable for unit testing as these methods are not deterministic as they use some source of randomness, hence specific values cannot be asserted. Moreover, tests could not be made for the Patrol class as they require the render function, which would be complicated to implement as mocking would need to be used. Methods within the Bomb class use variables declared as final so cannot comprehensively be tested.

2.1.4 Manual Tests

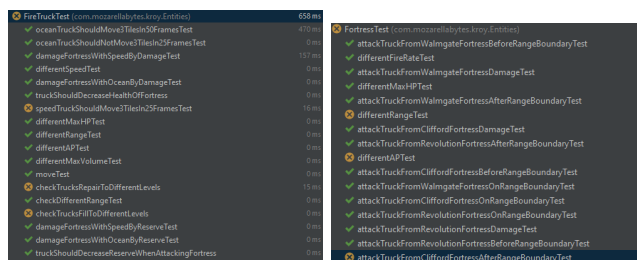
The manual tests were used to test screens and some classes for the mini-game that have complex code and graphical elements as well as sound features.

2.1.5 Traceability Matrix

The traceability matrix made by the previous team has been restructured in a way that the specific methods in a test case were separated from the matrix so that it clearly shows the link between requirements and our test cases without adding unnecessary detail. New automated and manual test cases were combined with the tests made previously.

2.1.6 Test Statistics

3 tests within the Firetruck and Fortress test classes failed as they use the GdxTestRunner which consists of rendering the graphical aspects. As a team, we felt it was best to dedicate the time to ensure the main elements of the game's functionality are tested.



FireTruckTest (com.monolithgames.king.kingdoms) 558 ms	FortressTest (com.monolithgames.king.kingdoms)
✓ oceanTruckShouldMove3TilesIn50FramesTest 470 ms	✓ attackTruckFromWaingateFortressBeforeRangeBoundaryTest
✓ oceanTruckShouldMove3TilesIn25FramesTest 0 ms	✓ differentFireRateTest
✓ damageFortressWithSpeedilyDamageTest 151 ms	✓ attackTruckFromWaingateFortressDamageTest
✓ differentSpeedTest 0 ms	✓ differentMaxHPTest
✓ damageFortressWithOccasionallyDamageTest 0 ms	✓ attackTruckFromWaingateFortressAfterRangeBoundaryTest
✓ truckShouldDecreaseHealthOffFortress 0 ms	○ differentRangeTest
○ speedTruckShouldMove3TilesIn25FramesTest 16 ms	✓ attackTruckFromCliffordFortressDamageTest
✓ differentMaxHPTest 0 ms	✓ attackTruckFromRevolutionFortressAfterRangeBoundaryTest
✓ differentRangeTest 0 ms	○ differentAPTest
✓ differentAPTest 0 ms	✓ attackTruckFromCliffordFortressBeforeRangeBoundaryTest
✓ moveTest 0 ms	✓ attackTruckFromWaingateFortressOnRangeBoundaryTest
○ checkTruckRepairToDifferentLevels 19 ms	✓ attackTruckFromCliffordFortressOnRangeBoundaryTest
✓ checkDifferentRangeTest 0 ms	✓ attackTruckFromRevolutionFortressOnRangeBoundaryTest
○ checkTruckIfAtToDifferentLevel 0 ms	✓ attackTruckFromRevolutionFortressDamageTest
✓ damageFortressWithSpeedilyReserveTest 0 ms	✓ attackTruckFromRevolutionFortressBeforeRangeBoundaryTest
✓ damageFortressWithOccasionallyReserveTest 0 ms	○ attackTruckFromCliffordFortressAfterRangeBoundaryTest
✓ truckShouldDecreaseReserveWhenAttackingFortress 0 ms	

2.2 Methods and Plans

Our method, planning, and use have tools has changed this assessment based on an evaluation of the successes and failures of our approach in the previous assessment, and Mozzarella bytes' approach. Our updated Gantt chart referenced in the document can be found [here](#).

2.2.1 Development Method

Previously we had tried to adhere closely to the Lean process, but this had proven to be too free-form. For example, developers had assigned themselves to tasks using a shared taskboard. However this meant developers were often unsure what work they should do. Mozzarella Bytes had followed a fairly relaxed Scrum method, but with the Scrum Master assigning work directly. We took on their approach in this assessment, and found it made the distribution of work more even. Daniella was assigned the role of Scum Master. We continued our weekly meeting structure as a place to report progress and assign jobs.

2.2.2 Planning

To create our Gantt chart for Assessments 3 and 4, we used Mozzarella Bytes' Gantt as a base, but with alterations. In the previous assessment we had tried to work as concurrently as possible, producing tests, code, and the write-up alongside each other ([the link to our old gantt chart can be found here](#)). This had the issue of splitting the team focus, and we decided it would be more productive focusing on one task at a time. Mozzarella Bytes' planning was structured very linearly, and split the work into several chunks with a deliverable produced at the end of each. As a result, our Gantt chart is structured more linearly, with the majority of documentation being produced in the last week. This way the deliverables couldn't become out-of-date and require updating if the project changed. Another element we added to the Gantt was customer meetings, which had been extremely useful in the previous assessment. Mozzarella Bytes hadn't planned any of these, so we altered the timeline to include a customer meeting midway through development. This meeting lead to us reevaluating requirements. The log of our consultation with the customer following our meeting can be found on the website [here](#).

2.2.3 Tools

Previously, we had used the Azure continuous development platform as a hub for our project. This had proven time consuming to maintain though, and we had only ended up using a small subset of the features available. Mozzarella Bytes had used GitHub for version control and email for organisation and communication. We opted to move to plain GitHub for our version control, as it was simpler to organise. We continued to use the task boards from Azure however, as we were familiar with that system and couldn't see any additional benefit from moving to email. We also benefited from using the polling feature in Facebook Messenger, which Mozzarella Bytes had advocated in their write-up.

2.3 References

[1] "Major Approaches Models of Change Management", Cleverism, 2020 [Online]. Available: <https://www.cleverism.com/major-approaches-models-of-change-management/>

[2] D.Kolovos, Lecture 9-Software Engineering Project. Topic: "Technical Debt and Refactoring", University of York, January 2020