Module	SEPR
Year	2019/20
Assessment	4
Team	Dalai Java
Members	Jack Kershaw, Max Lloyd, James Hau, Yuqing Gong, William Marr, Peter Clark.
Deliverable	Evaluation and Testing

Evaluation

To deliver a product that met the exact specifications of the stakeholders, we worked to take the necessary features provided to us in the brief and abstract them down into smaller, manageable and implementable tasks. We were left with a set of atomic features of the product that would be necessary to match the exact specifications provided to us in the product brief. This list then went on to be our deliverable for the requirements. We then methodically worked our way through the requirements, implementing them into the game from the ground upwards. This meant we first designed and implemented features like the map (**UR_select_level**) [1]. Once the level system was created, we were then able to build the product onto the existing system and add features like fire trucks (**UR_four_trucks**) and alien fortresses (**UR_six_ETs**). These features are part of the core experience of the game and as a team we came to the conclusion that a significant proportion of the game would be developed around these features, therefore for maximum efficiency while developing, they needed to be created first.

With reference to the product brief, it was specified that was necessary to be a single player game where the user controls fire engines to avoid/battle alien patrols and destroy the alien fortresses. As outlined in the brief and above, the intuitive first step to meeting the product brief was to implement **UR_select_level, UR_four_trucks** and **UR_six_ETs** first as they would allow us to begin to implement the detailed section of the brief. We needed to have the fire engines have the ability to attack ET's/aliens only when they are in range of their water cannon. This is reflected in the functional requirement **FR_engine_fire** which defines that "The user should be able to fire water at enemy patrols and fortresses when in range."

The fire engines were required to be unique. This meant that each fire engine had a different but pre-defined speed, water capacity and health. This is reflected in the requirement **FR_unique_engines**.

The fire engines were required to be able to refill their water capacity at a fire station. We decided that this was able to be implemented comprehensively over multiple features, those being **UR_station_refill, UR_refill_warning, FR_no_refill, NFR_ingame_warning**. These simply defined that the fire trucks could refill water at the station, the player would be warned when their water capacity is low however the warning should be non-intrusive and that once the fire station is destroyed, the player can no longer refill water.

To satisfy the specification that each alien fortress must be unique we reflected this in the requirement **FR_unique_enemy.** These alien fortresses would also need to repair themselves and become harder to defeat over time and this was implemented through **FR_fortress_recover.**

With one alien fortress per map, we implemented 6 maps which would satisfy the requirement that there needed to be a total of 6 alien fortresses (**FR_6_levels**). Furthermore, to satisfy the requirement for 4 fire trucks there is a user requirement **UR_four_trucks.**

The game is either won or lost when either all of the fire trucks are destroyed or the alien fortress is destroyed. This is seen in **UR_eng_game_screen, FR_enemies_die and FR_engine_destroyed.**

To implement the alien patrols converging on the fire station after a set amount of time has passed, first we needed to implement ET patrols (**UR_patrols**). We could then define that after a set amount of time elapses, the ET patrols converge onto the fire station (**FR_move_towards_station**).

The minigame was required to be a different style of play to the main game, while being no harder and not taking away from the main game's core experience. These requirements were reflected in **UR_minigame, FR_open_minigame, FR_minigame_opponents, FR_minigame_bomb**. The concept of this is that the player drops a bomb to destroy alien opponents.

In the most recent iteration of the assessment, our team received a notification that there was a change in the requirements for the game. This included the ability to save the game, close the application and then reload this save, select a difficulty from a series of pre-defined difficulties and for powerups to spawn around the map. To implement this we used the same strategy as we did at first and abstracted these down into atomic tasks which could then be added into the game in a manageable fashion.

Starting with the powerups, they were contingent on **UR_collisions** and **FR_deny_collision** as we did not want powerups spawning on inaccessible areas. The powerups themselves were added in with **FR_powerup**.

To implement the save features, we needed two new functional requirements being **FR_save_game** and **FR_load_game**. These defined that the user should be able to save and load 3 separate save files which would preserve the state of the game exactly. We also needed to define the user requirement **UR_save_load_quit** which meant the system would allow the user to perform the action of save and load and would preserve the game state.

Lastly to implement the difficulty setting, we needed to provide the available difficulties being easy (half health of enemies), normal (no deviation from original game) and hard (double enemy health, increased damage). This requirement is defined under **UR_difficulty**. We finally needed the last requirement which enabled the user to select these difficulties and remember the preference, this was defined as **FR_select_difficulty.**

Testing Report

We tested our code to ensure that it was of the appropriate quality for our customers. We use the term 'appropriate quality' as defined by the ISO 8204 and as discussed by Petrasch [2]; that the quality of our software is determined by the existence of characteristics that relate to our requirements. Hence the purpose of our testing is to ensure that the characteristics (or features) included in our product successfully fulfill the requirements they have been assigned to in our Table of Requirements.

For the most part, we retained DicyCat's approach to testing. We continued to use Test Driven Development, as outlined by Janzen and Saiedian [3]; this required a large amount of discipline, as we were required to explicitly outline what each section of code should do before implementing it so that Unit tests could be written beforehand. In some cases this was not possible and hence the Unit tests were written after the code had been implemented; we were aware that this could lead to incorrect values in testing and hence each Unit test was evaluated by other members of the team for accuracy. We also continued to evaluate each other's code through Peer Testing [4], and continued to use JUnit as our choice for automated Unit testing.

The new project we chose had a fairly large number of pre-existing Unit tests, however they did not fully cover every testable method; this is likely a result of previous development teams struggling with certain JUnit/Mockito functions. In order to amend this, we added a number of Unit tests to ensure that maximum possible line coverage was achieved. We also added new Unit tests to accommodate the changes we made to the code, as documented in our Implementation Report.

The testing documentation and traceability matrix provided to us by DicyCat only covered the new tests that had been implemented in Assessment 3; we therefore merged their documentation with the documentation table created by NP Studios in Assessment 2 in order to create a comprehensive table of tests, adding the new Unit tests we had defined in the code as well as a large number of manual tests in order to ensure that every requirement was sufficiently met. We have retained the 'related requirements' section of the table in order to show which requirements the given characteristics are attempting to fulfill. In addition to this, we have created a new, final, traceability matrix linking the updated tests with the updated requirements.

Unit Tests

The table below presents the statistics from our Unit tests, including success rate and line coverage.

Class	Tests Ran	Tests Passed	Tests Failed	Method Coverage	Line Coverage
Alien	11	11	0	81%	79%
Character	2	2	0	100%	100%
Entity	4	4	0	57%	65%
FireTruck	18	18	0	100%	81%

Fortress	9	9	0	100%	100%
PowerUps	1	1	0	100%	100%
Projectile	5	5	0	100%	100%
Unit	10	10	0	100%	100%

In addition to this, the FireStation class has 100% method and line coverage despite not having any tests explicitly allocated to it.

The majority of classes tested had full method coverage from the tests, showing completeness in the classes where this is the case. In the case of those which did not, such as the Entity class, this contained code specific to Gdx features which we were unable to mock. However, we ensured this was covered comprehensively via our manual tests. The FireTruck class also does not have complete line coverage; this is largely due to this class incorporating timer scheduling features. Whilst code was written to mock this feature, we were unable to explicitly test whether the functions themselves worked, hence they were not included in the line coverage.

Manual Tests

We included manual tests in order to provide comprehensive testing for cases in which JUnit was not a realistic option. As manual tests have the chance of producing human error, we repeated each test to ensure that our results were consistent. In total we ran 28 manual tests, all of which passed our specified criteria.

Comments on Testing

Our testing documentation allows us to clearly see that each test we have defined has been successful. This means that each characteristic included in the game fulfils the requirement it is set out to fulfil. As depicted in our Traceability Matrix, our tests cover every requirement in our Table of Requirements, which means that every requirement has related characteristics in our game that fulfil said requirement. Hence, using the definition outlined previously, our code is of an appropriate quality.

Links

Traceability Matrix:
https://baffledwhiskey.github.io/files/Assessment%204/Traceability%20Matrix.pdf

Test Tables: https://baffledwhiskey.github.io/files/Assessment%204/test_documentation.pdf

Unit Test Traceability:
https://baffledwhiskey.github.io/files/Assessment%204/Unit%20Test%20Traceability.pdf

Bibliography

[1] Dalai Java, Baffledwhiskey.github.io, 2020. [Online]. Available: https://baffledwhiskey.github.io/files/Assessment%204/Updated-Requirements-tables.pdf. [Accessed: 20- Apr- 2020].

[2] R. Petrasch, “The Definition of ‘Software Quality-: A Practical Approach.” *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE)*, Fast Abstracts & Industrial Practices. IEEE Computer Society, Nov. 1999.

[3] D. Janzen and H. Saiedian, “Test-driven development concepts, taxonomy, and future direction,” *Computer*, vol. 38, no. 9, pp. 43–50, Sep. 2005.

[4] N. Clark, “Peer testing in Software Engineering Projects”, *ACE ‘04 Proceedings of the Sixth Australasian Conference on Computing Education*, vol. 30, pp. 41-48, 2004.