Module Code
COM00008I

BSc, BEng and MEng Degree Examinations 2019–20

DEPARTMENT OF COMPUTER SCIENCE

Software Engineering Project (SEPR)

Open Group Assessment

Module	Software Engineering Project (SEPR)
Year	2019/20
Assessment	4
Team	The Dicy Cat
Members	Michele Imbriani Daniel Yates Luke Taylor Isaac Albiston Martha Cartwright Riju De Sean Corrigan
Deliverable	Evaluation and testing report

1. EVALUATION AND TESTING REPORT

1.1 Code Evaluation

To ensure that our code successfully adhered to and ultimately satisfied the product brief, we took a two stage approach, firstly confirming that we had implemented all the features we could from our requirements (including some non-essential quality of life changes), and secondly, using a robust testing method to verify that all these functions were working as intended. The key to achieving both these goals was maintaining a consistent code style alongside rigorous documentation.

Our team believes that our final product meets the product brief in its entirety. From the earliest stages of the project's development, we made a priority to not only always keep the Requirement Tables up to date with the team's decisions and with the new client requests that arrived at the beginning of each Assessment, but also to make sure they reflected an accurate and precise representation of what our team was expected to produce. This was done at various point during Assessment 1, 2, 3 and 4 by double checking whether our decisions resonated with the client's will, who constantly provided us immediate clarifications and feedback. We can confidently affirm that there has not been a single point in time throughout the development process in which our product sidetracked from the client's vision, nor that the Requirement Tables were -without considering minor grammar mistakes- unprecise about what the product brief requested.

One thing that proved to be crucial in helping us with the testing -as defined in the Assessment paper- of the final product was the use a set of robust guidelines from Brigham Young University [1], as will be covered and highlighted in bold below, which allowed us to ensure our code was of a high quality and standard. As a matter of fact, the robust and reliable sets of tests that were developed throughout the course of Assessments 2 and 3 (since we selected our own initial project) which provided us with a high level of confidence, was coupled with this set of guidelines which we used to check our code's quality against, therefore making sure that our judgment was as close to academic standards as possible.

'Effective class, method, and variable names'

This aspect of code quality was straightforward for us to establish a standard and stick to it. We agreed on camel case as a Java standard for our project which produced consistency in our variable, method and class names creating a codebase-wide readability standard, as well as allowing us to easily access methods and variables from other modules in our code. We also wanted to provide informative, but concise variable names. Luckily due to the nature of the project it was very easy for us to name methods and variables in a way that was very clear as to how they worked within the program, for example it is entirely unambiguous as to what 'refillWater' achieves in the context of our 'FireTruck' class. Finally we not only maintained consistency in naming convention and structure, but we were also able to utilise methods consistently over multiple similar classes due to our heavily object-oriented approach, for example the 'update' method is consistent over a large number of the interactive objects in our program allowing us to iterate over them every frame - this compartmentalisation makes it very clear where it is appropriate to implement new features and functions.

'Effective top-down decomposition of algorithms'

The decision of adopting a top-down approach to our code from the very beginning of our project clearly paid off towards the latest stages of the development: all our methods are concise, and perform a specific, unique task. We were also aided by our code style, we found that developers working in separate branches could write methods performing an identical function, but due to our naming conventions these would be named identically, and we could eliminate redundant code appropriately during git merges. Finally, due to our top-down class structure, we were able to further optimise our code by keeping essential, universal functions in parent classes leaving the fine unique detail isolated to the child classes that needed them.

'Code layout should be readable and consistent'

As touched upon in the previous two rules of evaluation, we were stringent in ensuring our code was consistent with the conventions we established as a group at the beginning of development. Of course, consistency does not equate to readability -code can be consistently bad- however, by rigidly adhering to said conventions and

ensuring they were an industry standard from the beginning, we are confident that the codebase we produced is consistent, readable and easy to comprehend.

By ensuring our code had a standard structure and style, we were able to effectively and quickly comprehend modules/classes/methods that we personally did not write to confirm they met the prescribed requirement as well as being able to utilise said code blocks elsewhere in the program. Our continued approach to documentation also streamlined our development process for this development cycle, we continued to not only document code we wrote for ease of reading, and effective searching of our codebase but we also maintained a separate style for modifications to previously existing sections of code. This helped us twofold as it aided git merges by allowing us to quickly identify and incorporate new changes from other branches as we merge, but also to track what changes we should play-test and write JUnit modules for.

'Effective source tree directory structure'

In choosing LibGDX as the basis for our project, which uses Gradle to organise the file structure into a general standard, a large amount of our project structure was auto-generated. This wasoptimal, as it allowed us to focus our main functionality into the *'src'* directory with periphery directories used for cross platform functionality, a welcome addition if the project was to ever be ported to further platforms as suggested by the client himself in the very first client meeting. Due to the large focus of our work taking place within the *'src'* directory, we further compartmentalised our files into appropriate locations for ease of searching and accessing, keeping game entities, debugging tools, game scenes and other code blocks separate, but grouped with their appropriate components.

'Effective file organisation'

Another paradigm that was straightforward for us to follow. Our segmented approach to code organisation and implementation from the beginning was to separate class files and ensure no class or file was overburdened with unnecessary, or redundant information - i.e. repeated code from another class. We also tried to keep core, large classes outside the categorised directories for quick and easy access when a new feature was to be added to the main gameplay loop.

'Correct exception handling'

We took an approach to avoid exceptions unless absolutely necessary, in doing so we created a closed system, where - as far as our testing has shown us - can only throw exceptions if the source files or assets are directly altered by an end user **[? - Is this true]**. The user input is strictly limited to the controls we have introduced into the code and in doing so take away the influence the user has to initiate any kind of error. By restricting the user in this way, we can focus on our JUnit tests as if these are robust in testing our code we can be confident no errors will be created during runtime.

'Good unit test cases'

As is comprehensively covered in the next section of this report, when it came to test design and generation, we took a more holistic approach than in previous assessments. We found previously that by delaying testing we ended up with insufficient time to write and run effective and meaningful test cases. Taking this on board we immediately structured our team into two groups: designing, and implementation. In doing so we allowed our test designers to focus solely on the scope and overall comprehensiveness of the tests with those implementing said designs to ensure the tests were written well and correctly. This black-box styled approach meant that tests were neither repeated, nor were any that should have been added absent (justification for black-box method in following paragraph).

Code Testing

When our team decided to take on the project from NPStudios, we were aware of the peculiarity in the testing approach that the previous team adopted. As explained on their website, they decided to *"create an alternate code base where the testing team would slightly alter the code to allow for the classes to be tested"* [2]. Despite being unfamiliar with this way of testing, and seeing the positive results obtained by NPStudios with their

tests, we therefore decided to embrace this method and employ it for our own practices as well. This turned out to be a sensible decision, as it sped up the process of test generation and execution, and allowed to flawlessly transition from one test to the other in a short period of time. This, for instance, revealed to be especially true when generating the tests for Powerups: by having an alternate code base, we were able to apply the same theoretical concept for testing a powerup for every single one, therefore optimizing the time that needed to be spent designing the tests.

During the course of the project's development, various forms and approaches of testing have been adopted by our team. As mentioned earlier, for this last Assessment we decided to change our previous approach of using a white-box style of test development to give way to a black-box style. This decision was made in order to reflect the team organization changes which will be discussed in the Project Review Report: the team felt that the most efficient way of assigning tasks was based on a combination of competence and affinity. Thus, it naturally followed to split our team into Designing team and Implementation team, so that the members assigned to the Designing team -considering the objectively less requiring nature of the task- could get reassigned to other section of the development which needed to be worked on.

Alongside checking our code's quality against the afore-discussed guidelines, another criterion our team took advantage of for assessing the appropriateness was gameplay's features coverage and safety: in other words, functional testing. This was done by initially creating use cases and developing counter tests for each use case, and then by playing the game itself and triggering the feature at hand that is being tested. Moreover, after every iteration of design and development of each test, we made it a priority to always make sure every single feature of our game was covered, in order to ensure that the end user (the player) would have an error-free and enjoyable experience.

In conclusion, our team believes that our final product not only meets the product brief in its entirety, but it also does it with the highest possible quality code-wise that our team could have produced. Looking back, every single team member is satisfied with what was accomplished with each iteration of the Assessment, and thinks that, despite a minor setback in Assessment 3 concerning the tests, the utmost effort was invested into the development of our Kroy game. Finally, our team believes that the product is ready for deployment to the public, and can affirm with the most confidence that it will have a great impact on the player no matter of their previous gaming experience.

1.2 REQUIREMENTS

For this assessment, dealing with the changes in requirements, and therefore making our product meet them in their entirety, was by no means a difficult task. The reason for this is because we decided to pick back our own project from a team (NPStudios) which picked our project in Assessment 3: we effectively returned to our initial project. This choice was indisputable for two main reasons: firstly, it allowed us to have a complete, low-level and inside-out understanding of the game's architecture along with its strengths and weaknesses, which simplified the task of meeting all the requirements; secondly, the success obtained by our project among our cohort in Assessment 3 and 4 (2 teams chose our project in Assessment 3, while 6 teams chose one of the two projects based off ours in Assessment 4) was another validation, alongside our own satisfaction and the positive marking and feedback that followed, that the product our unit delivered in Assessment 2 was a solid, respectable and worthwhile pick.

With that being said, it is important to note that NPStudios did make two major changes to requirements which led to the game mechanics and architecture to stray from our team's initial vision in subtle but noticeable ways. In our initial plan [2] we discussed with the product owner and agreed that the implementation of the 4 different playable fire trucks requested in the product brief would be done by prompting the player with a selection screen int which one out of 4 different fire trucks -each with unique characteristics- could be chosen to start the game with 4 stocks (lives) (SFR_FIRETRUCKS_STATS, SFR_FIRETRUCKS_SELECTION) [2].

The first major change NPStudios made is that instead of pushing forward our idea, they decided to change it instead [3] by making the four different fire trucks playable simultaneously (SFR_FIRETRUCKS_STATS, UR_FIRETRUCK_MIN_ START) [3] thus eliminating the need of a truck selection screen and a lives-based system (the game ends once all the four fire trucks in the game are destroyed).

The second relevant change the other team made to our requirements concerns the minigame. Initially [], our squad planned the minigame to be a SuperMario-styled game, triggered whenever an ET fortress' life points are brought to 0. The aim of the minigame would be to make the firefighter fight his way to the main alien in order to destroy the fortress (SFR_MINIGAME [2]). NPStudios, on the other hand, decided to radically diverge from this idea: the minigame they created and implemented gets triggered when a single fire truck with non-full water tank moves in the proximity of the fire station, and the aim is to connect pipes so that water can flow within and refill the tank(SFR_MINIGAME [3]). For Assessment 4, despite having decided to carry on their idea, we noticed that the minigame obstructed the flow and momentum of the main game: we therefore decided to implement the feature that requires the user to click -using the cursor- on the fire station to start the minigame and refill all trucks that are within its range. By doing so, a player can refill all fire trucks at once, triggering the minigame only once instead of four times.

Finally, as mentioned above, the newly added requirements were fairly straight-forward to accommodate and meet since everyone in the team had a thorough understanding of the underlying architecture and structure of the game.

Please find the updated requirements tables on our team's website. [4]

In conclusion, for the aforementioned reasons, and for the ones explained in the previous section, we believe that our final product meets all the elicited requirements.

References

[1] Faculty.cs.byu.edu. 2020. *Code Evaluation Criteria*. [online] Available at: <https://faculty.cs.byu.edu/~rodham/cs240/code-eval/CodeEvaluation.html> [Accessed 14 April 2020].

[2] Assessment 2, Updated Requirements: https://drive.google.com/file/d/1JXH3xYIedqd_BajWwBFWTkifSvhY8hZ6/view

[3] Assessment 3, NPStudios' New and Changed Requirements: https://npstudios.github.io/files/NewandChangedRequirements.pdf

[4] Assessment 4, Updated Requirements Tables: https://dicycat.github.io/files/UpdatedRequirementsTables.pdf