

Student Name: Leon Buckney

Student Number: 2404085

Student Email: 2404085@student.uwtsd.ac.uk

Module: Software Engineering Principles and Testing

1. Introduction

This report will discuss the design and testing stages of a battleship's web application, which was assigned to our team for us to develop. The system which we had to design was based on an original Battleships game, which followed the same game rules as the original, being two player must place their naval ships on a grid and must take turns trying to locate and destroy each other's ships. To make the game more interesting, we were tasked with adding new features, such as a bomb that can destroy an entire ship, as well as a maneuver ability that allows a player to reposition a selected ship during the game, but each ability could only be used once. For our team to implement these features, careful planning was required to ensure they both work correctly and keep the game balanced.

During the design phase, our team used several techniques that had both benefits and drawbacks, but helped us organize and structure our systems. User stories were the first technique we used, and they helped us describe how the system would function from a player's point of view, which greatly helped us keep the design majority focused on usability. Another great technique we were recommended to use was CRC cards. By being able to use them, we were clearly able to identify the primary elements of the system and how they interact. Before implementation, we used wireframes, which allowed the team to visualize the UI layout and enhance the overall user experience.

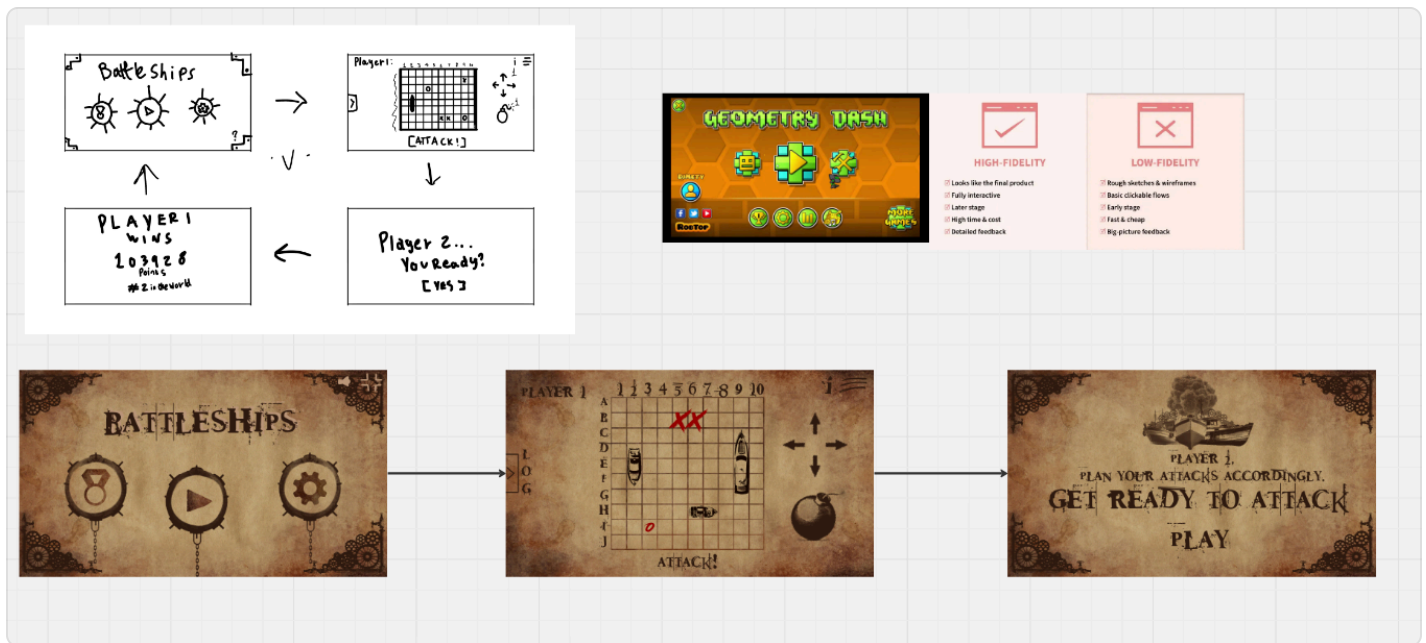
Throughout the report, you will see evaluations of how effective these design techniques were and how well they worked within a team environment. Additionally, it discusses the significance of software testing, including white & black box testing, and how these methods might be applied to guarantee that the system would function as intended if it were to be created.

2. Design Overview

The design of the Battleships web application was developed through a collaborative, team-based approach, enabling the exchange and refinement of ideas throughout the development process. Working in a group allowed consideration of multiple perspectives, resulting in a more detailed and well-structured system design. The team ensured that all members had a comprehensive understanding of the system requirements and the interplay among the application's components through continuous feedback and discussions.

A variety of software engineering techniques were implemented to facilitate the design process. To ensure the system remained centered on the user's needs and overall experience, user stories were developed to define the functional requirements from the player's perspective. These user stories provided a structured, unambiguous way to identify the game's most significant features and interactions. Furthermore, the system's structure was modeled using CRC (Class, Responsibility, Collaboration) cards to identify the primary classes, their responsibilities, and their interactions. This facilitated the establishment of a logical foundation for the application and guaranteed the organization and coherence of the system design. Additionally, wireframe diagrams were created to facilitate the team's evaluation of usability and design decisions prior to implementation by providing a visual representation of the user interface's layout and navigation.

Figure 1



The system was created to integrate both conventional battleship mechanics and supplementary features to improve the overall gameplay. Players use a grid-based interface to position ships of varying sizes and take turns attempting to strike their opponents' positions. Additional complexity was introduced by incorporating features such as the drop bomb and counter maneuver, necessitating meticulous consideration to ensure they were effectively integrated and balanced within the overall design.

3. Evaluation of Design

One of the team's main advantages during the design phase had to be our collective attitude towards it. By every team member being involved and speaking up when they felt they needed to, we were able to design a great depiction of what the Battleships game should look like. Because the team had the courage to speak up individually and give each other feedback, we were able to identify issues earlier, so we wouldn't need to fix them later in the project, slowing our progress. The quality and coherence of the design were greatly improved compared to the start of the project, thanks to being able to communicate as a team and express our opinions on different design ideas.

Another great advantage we had as a team was the use of user stories. These were used in order to define the functional need of that game from a user perspective. User stories allowed the team to brainstorm ideas that best suited the specification requirements and the desired user gaming experience. Functions such as ship placement, turn-based interactions, and other mechanics, such as counter-maneuver, were made very easy to identify and prioritize. As a result, our team was able to create a system that was both useful and entertaining for users.

By using CRC and wireframe diagrams, we were able to further strengthen the design process even more. CRC cards allowed us to create classes and their duties as well as their relationships, which provided a clear representation of the system structure. This made it much easier for the team to create a well-organized, logical design, which will be really useful when we start implementation. Referring to Figure 1, the Wireframes we created helped the team gain a clearer understanding of the user interface, enabling us to identify potential usability issues and improve the layout before deployment. All together, these methodologies allowed our team to evaluate the user experience and technical framework with great caution and success.

3.2 Weaknesses of the Design

However, even though there were many advantages, the design process was affected by several negative aspects. One of the most significant problems we faced was improper communication as a team. Throughout the first stage of the design process, there was no clear process for reaching a final decision since numerous people were submitting ideas, which led to occasional confusion and inconsistencies regarding the design. This became particularly noticeable when we focused on more complex features, such as the counter maneuver, because not all possible scenarios were fully considered. For instance, edge cases, like how ship movement interacts with grid boundaries or pre-existing attacks, were not always well described.

Although CRC cards were helpful for sketching the structure of the system, they frequently lacked enough detail in order to fully support implementation and testing. This was one of the major downsides of using CRC cards. In most cases, the descriptions of responsibilities and interactions were basic, without delving into more comprehensive logic or validation criteria. Translating the design into code could have become more challenging and harder to execute. Similarly, Wireframes were very useful for visualizing how the user interface would look, but they did not depict how the system would function in the background. As a result, logical issues could have been missed and not identified during the design phase. These flaws show that even though the design provided a solid basis, it could have greatly benefited from having detailed documentation and earlier testing.

3.3 Reflection on Teamwork

Even though our team faced many challenges throughout the design process, there were plenty of positive aspects of working within a larger team. Having a larger team with each individual having a unique set of skills allowed us as a collective to distribute tasks more effectively and for members to contribute to multiple areas of the design rather than being limited to a single responsibility. This allowed more flexibility within the team and enabled work to continue even when tasks took longer than expected. As mentioned previously, each team member has a unique set of skills, which enabled the team as a whole to provide a clear perspective and help create a more comprehensive and intricate system design. For instance, two members of the team had skills that were more focused on UI design, and they really wanted to start making concept design for the interface as soon as the project began, whilst other members of the group concentrated on design CRC cards which would allow us to structure the system or user stories to define requirements. Because of everyone's diverse skills and passions, the team approached the design from both technical and creative standpoints, which, over time, improved the application's overall quality. Nevertheless, despite having these benefits, there were drawbacks that affected the team's work efficiency. At times, there was little communication amongst the team during conversations, which made it challenging to exchange ideas honestly and make decisions quickly. Another struggle we had as a group was a lack of consistent work, which then led to an imbalance of contributions and increased accountability for others. As a result, many components of the design were not examined as expected, which occasionally affected the depth of conversation. To avoid these issues, there should be regular team progress meetings to track each member's progress on their segment, and enhanced communication with more explicit expectations could really help the team understand what needs to be completed. By using these strategies in future projects, all team members should stay involved and make valuable connections.

4. User Stories and Story Map

4.1 User Stories

One key method that is commonly used in Agile software development is User Stories. By depicting the smallest unit of work from the end user's point of view, user stories ensure teams focus on delivering value by concentrating on the overarching aim or outcome the user intends to achieve, rather than going into depth on technical aspects. In most cases, user stories are written in a simple, informal format that allows both developers and customers to understand the system's intended functionality. Because they provide a clear, succinct description of what the system should accomplish without adding unnecessary complexity, user stories are a great way to express requirements within a team [5]. Also, user stories are commonly used in Agile frameworks such as Scrum and Kanban, where they support planning, prioritization, and organizing work into manageable tasks. By breaking the system into small components, user stories allow teams to approach development in a more organized and adaptable way.

To represent key features and interactions in the Battleships application, several user stories were created that cover both core gameplay and additional features. These included basic tasks such as selecting the game mode, positioning ships on the grid, and alternating between attacking the opponent's board by selecting particular grid locations. In addition to providing awareness of ship status and grid positions during gameplay, user stories also highlighted the need for feedback, enabling players to understand the results of their actions through a hit-or-miss notification. More sophisticated features, such as the ability to use special abilities, such as bombs, to destroy an entire ship in a single move, and perform a counter maneuver to reposition ships and avoid attacks. Further stories focused on enhancing the overall user experience, including showing logs of both users' actions, allowing players to end their turn early, modifying game settings, and displaying the final outcome via a game-over screen and leaderboard. All of these user stories used Acceptance criteria to support them, ensuring precise parameters were established for how each feature should operate within the system.

Throughout the design phase, user stories helped define features from the player's perspective and served as a key tool for the team. By being able to focus on what the user should be able to do in the game. We were able to ensure that the system was designed with usability and functionality in mind. Instead of just being focused on the technical implementation, each user story showed key game interactions such as placing ships, rotating ships, and special abilities, which made it a lot easier to comprehend how the system should act. This also helped the team be able to break down the design into smaller, manageable segments, making this whole phase easier for us to follow and structure. The team was also able to specify exactly how features should function and what conditions must be met. This made it simpler to determine if features were working as expected during testing and helped maintain consistency throughout the system. Overall, user stories made this phase more centralized around the users and more organized.

4.2 Benefits of User Stories

User stories offer a large range of advantages that can improve the design phase in software development. Because they are written in a clear style, they can be understood by technical and non-technical members alike, making every team member involved, which is what allowed our team's communication to improve. Also, by dividing the system into smaller tasks, user stories can help organize each task, making it become more easier to plan and manage work. This allows teams as a whole or individually to work on tasks while still contributing to the whole system.

Another great benefit of using User stories is that they can determine which features should be prioritized, which makes it effortless to identify key features, and makes it possible to add less crucial features later on while ensuring that crucial functionality is built initially. They also provide flexibility, as modifications can be made more readily without impacting the system as a whole. In general, user stories contribute to the development of a design process that is more organized, productive, and collaborative.

4.3 User Story Map

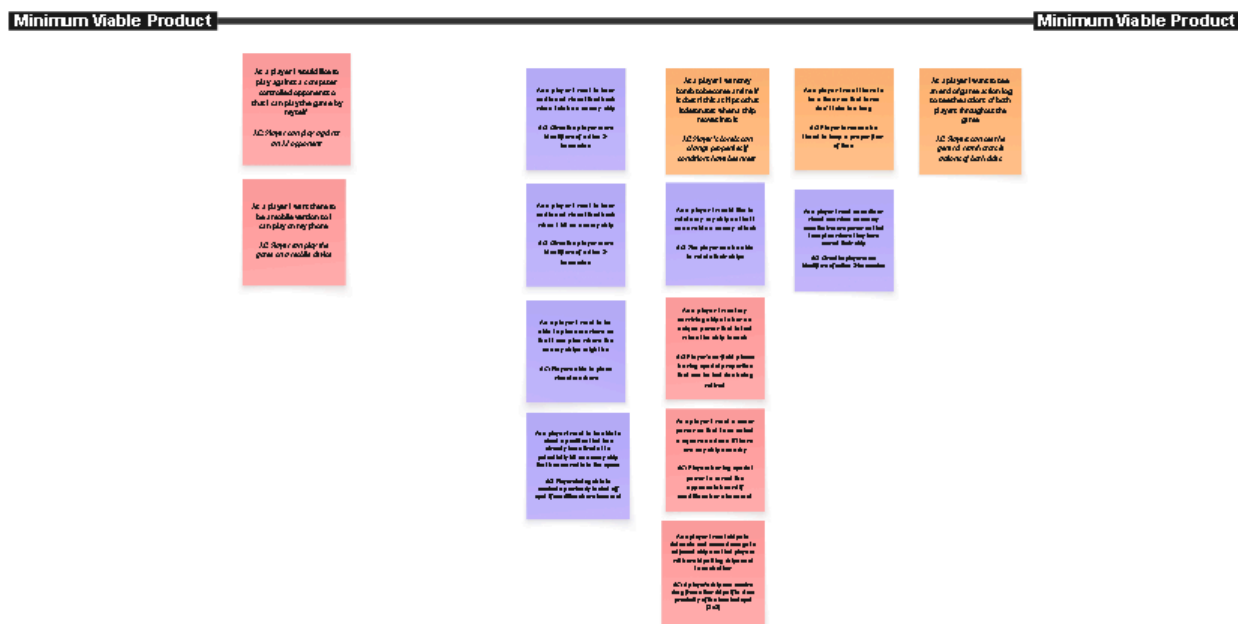
The user stories were organized on a user story map, which also shows how the different sections are connected. Instead of looking at each feature individually, user maps allow you to see how the entire flow of the game would flow, such as actions like beginning the game, placing ships, rotating turns, and game ending. Using user maps makes it simpler to comprehend how the player engages with the system at each stage.

Figure 2



As shown in Figure 2, the user maps we created were organized using a prioritization strategy, with categories such as “Must have”, “Should have”, “Could have”, and “Would have”. We were able to group features into those that made it quicker to figure out which were crucial to the main gameplay and which weren't. For sophisticated features, they were given a lower priority compared to basic features, which fundamental to the game, like placing ships and attacking.

Figure 3



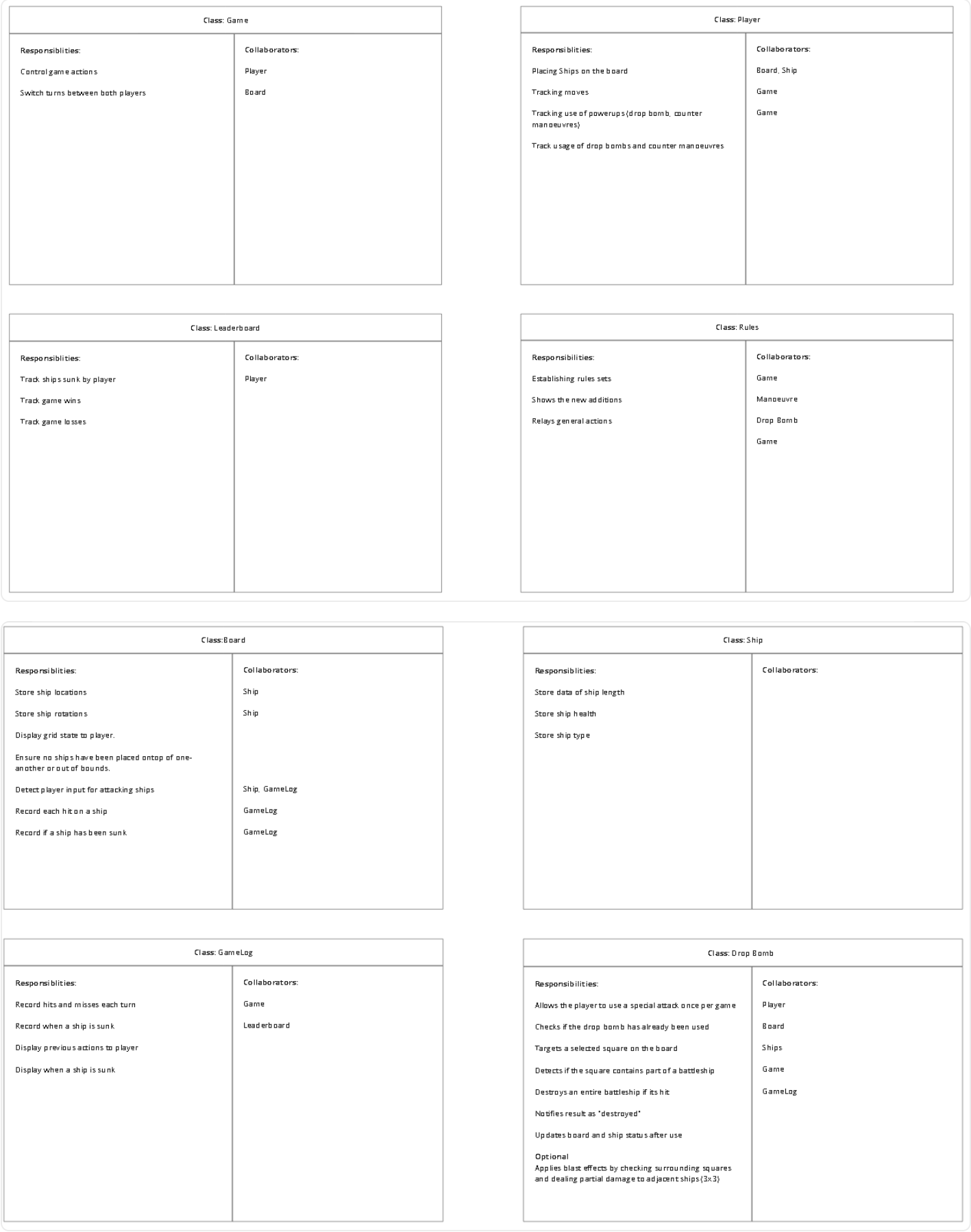
One core concept we discussed as a group, which was later added, was an MVP (Minimum Viable Product) as shown in Figure 3. This required us to focus on creating a workable version of the game with only the most important features first, before incorporating additional features. This strategy we used made it so we all focused on the key components and made them all well defined, whilst trying to keep the design straightforward and structured.

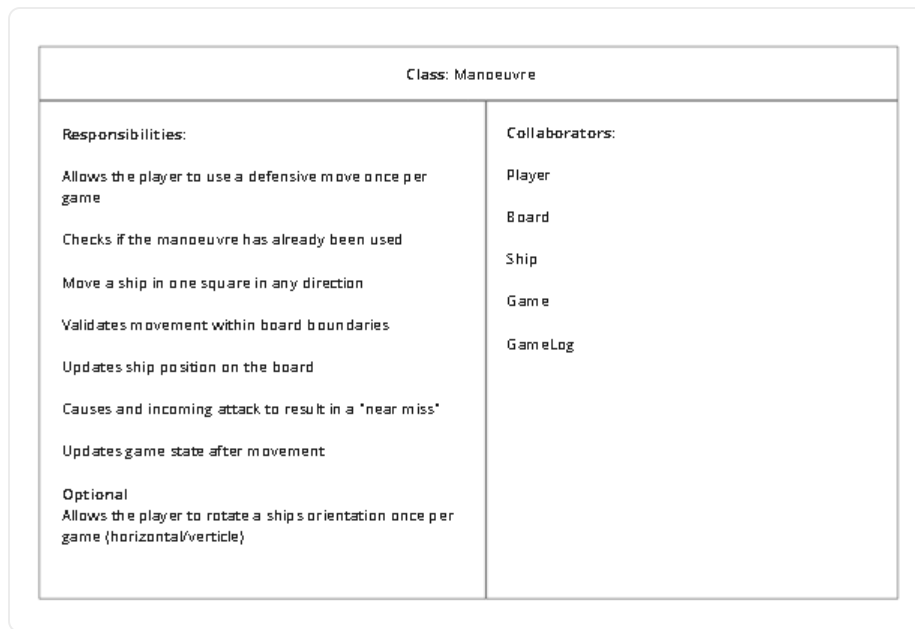
5. CRC Cards

5.1 Explanation of CRC Approach - 5.2 Evaluation of Your CRC Cards

To represent the key classes for the Battleships game, our team used CRC cards (Class, Responsibility, and Collaboration). We were able to create many key classes, such as player, game, board, and rules, as well as assign them their own responsibilities and how they communicated with other cards. This approach allowed the team to view how individual components interact, rather than treating the application as a single massive structure. Breaking our system into smaller sections allows us to gain a clear picture of how the game will work and greatly improve organization, not to mention it also illustrates how the system is divided into key components and how tasks are distributed across software, as shown in Figure 4.

Figure 4

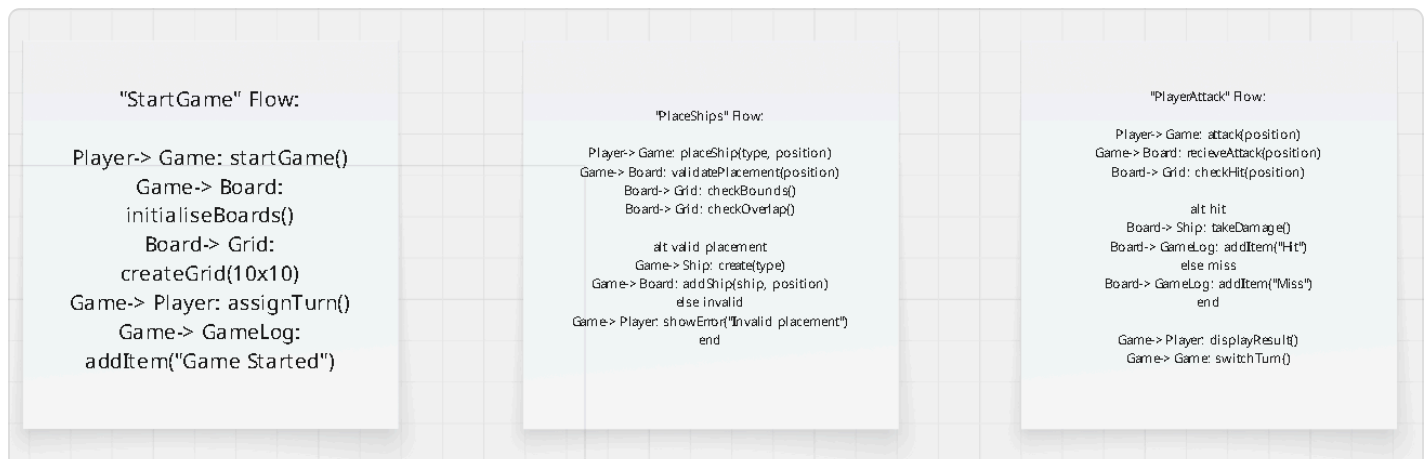


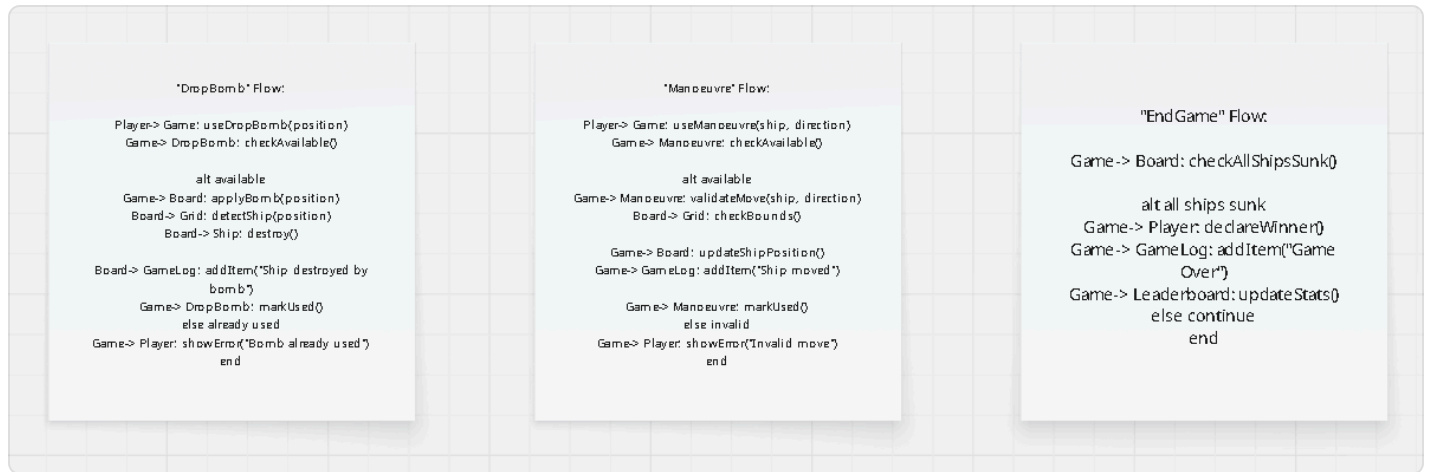


As mentioned previously, CRC cards were used to represent key classes [1] such as player and game, and each was assigned different tasks, e.g., the player class is responsible for performing moves and interacting with the game, while the grid class is responsible for maintaining the ship's location on the grid and placement. But the overall movement and game termination for the battleships game are handled by the game class. Additionally, the collaborative component of CRC cards allows you to see how other cards interact with one another, such as the players interacting with the board to select where they would like to attack. This makes it much easier to see how data and actions flow through the system and to ensure that tasks are assigned correctly.

Another great benefit of using CRC cards is that they structure and methodize the design process. Being able to clearly assign different tasks to classes makes implementing code much simpler and easier to understand. It can also reduce the chance of potential tasks overlapping, which could lead to miscommunication and errors during the development stage. Even though CRC cards are great at defining the overall structure of a system, they do have their drawbacks, one in particular being that, for instance, in a scenario where responsibilities were defined at a higher level and lacked sufficient logic or validation rules, it can be quite hard to translate the design into a functional implementation. Also forgot to mention that, with the help of System flow diagrams, we were able to understand how key gameplay actions, such as attacks and ability use, are processed within the Battleships application, as shown in Figure 5.

Figure 5

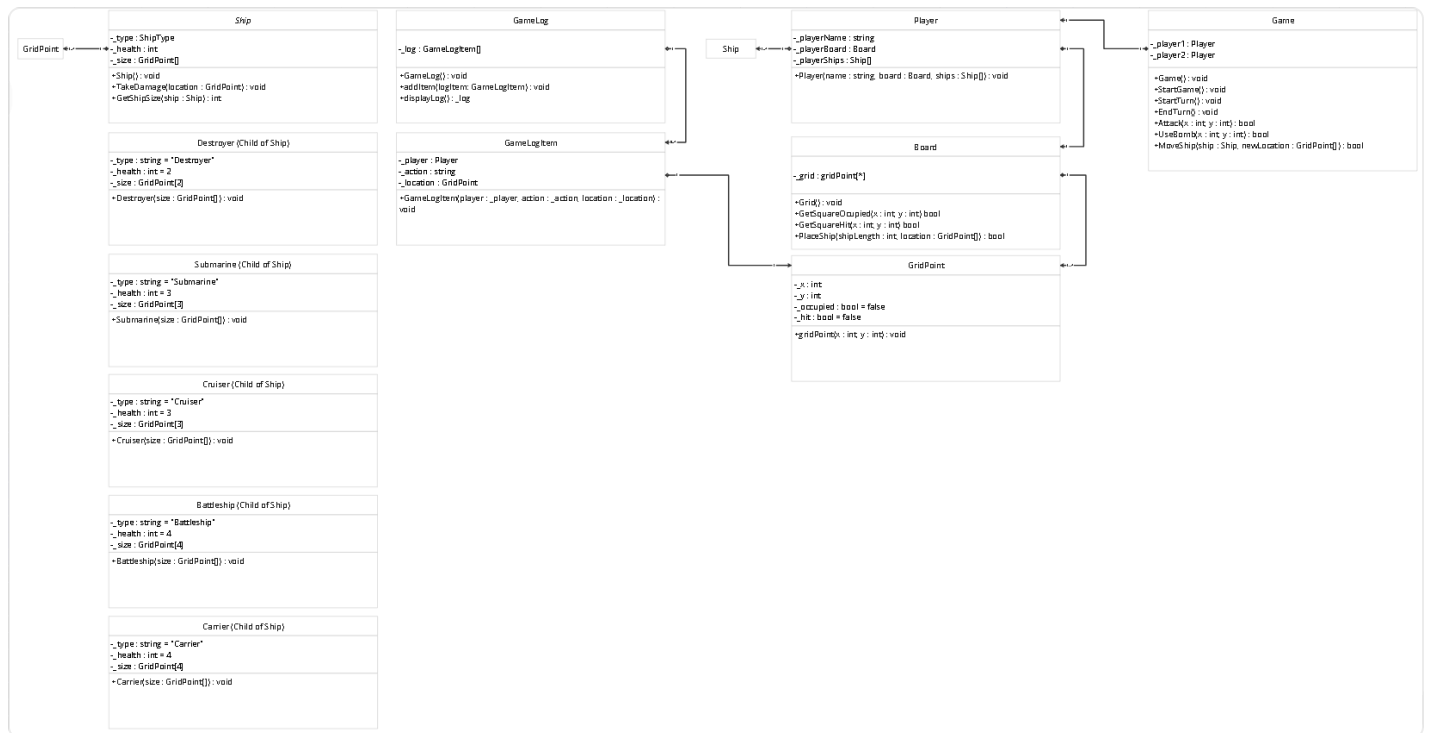




Given everything that's been mentioned, CRC cards are a great tool that can provide a transparent way to define component relationships and organize the design. They are essential for ensuring the system's logical structure and support throughout the design phase, regardless of their lack of specificity.

A UML diagram was also created [2] to provide a more detailed view of the system structure and component relationships, as shown in **Figure 6**.

Figure 6



6. Software Testing Evaluation

6.1 Importance of Software Testing

Within the software development lifecycle, there is one crucial part named software testing, which is used to evaluate and improve the quality of a system. There are many reasons why testing is important, but one that stands out most is that it helps identify defects and errors earlier in the development process, when they are easier and less expensive to fix. By identifying errors earlier, the likelihood of serious issues later, such as incorrect functionality or system failures, is reduced. Another reason why software testing is important, especially during the design phase, is that it can provide confidence that the system is working as intended and satisfies the requirements given. Considering the task our team has been given, being an interactive system, this is quite important as key features such as ship placement, turn-based gameplay, and impact detection must operate properly to guarantee fair and consistent gameplay. Failure of these features could have a detrimental impact on the user experience and lower the system's overall quality.

Additionally, testing ensures the program is dependable, functional, and user-friendly, thereby increasing user satisfaction. When testing, it can highlight potential weaknesses within a system and help reduce risks related to performance and security. And it's important to mention that whilst during the testing stage feedback will be given, and therefore it can be used to refine features as well as overall system performance. This is really important, as the Battleship game has several complex features. As mentioned previously, the drop bomb and counter maneuver require careful observation to ensure they behave correctly across different playstyle scenarios. Altogether, software testing is needed in large-scale projects [3], and it can both reduce risk as well as improve system quality if done correctly to meet both technical and user expectations and requirements.

6.2 Black Box Testing

Black box testing is a type of software testing in which the tester doesn't focus on the software's internal knowledge or specifications, but instead validates functionality against the given specifications or requirements. By focusing on the functionality, this type of testing becomes particularly useful for validating that the application meets its intended purpose. If black box testing were used on the Battleships game, it would involve testing key game features like placing ships on the grid as well as taking turns to attack or even receiving feedback on whether a player hit or missed a ship. The tester doesn't need to know how the system computes these results; rather, they need to know whether the correct outcomes occur when an action is performed.

Another great use of black-box testing is confirming that more sophisticated functions, such as drop bombs or countermeasures, work as intended. For example, a tester might drop a bomb at a ship's position to see if the enemy ship is destroyed. If so, then a notification should appear explaining whether it hit or missed. Another great example is that a tester could use the maneuver ability to check whether the system actually changes the ship's position. Using this approach, all user-facing features function properly and deliver the expected results. Altogether, black-box testing is a great way to verify the system's usability and functionality from the end user's perspective.

6.3 White Box Testing

White box testing does the polar opposite of black box testing: it focuses on testing the software's internal knowledge and specifications rather than its functionality requirements and specifications. Meaning, instead of focusing on inputs and outputs, it focuses on system implementation, e.g., code, logic, and execution paths that make sure all parts of the system function as they should. In most cases, white-box testing is used to verify the correctness of algorithms and the conditions under which the application's internal processes operate.

Regarding the Battleship game, white-box testing would involve analyzing and testing the internal logic that controls its key gameplay features. For example, to ensure it accurately checks the grid and modifies the game state, e.g., the logic that determines whether an attack results in a hit or a miss, would need to be checked. The drop bomb function would also need to be tested to ensure it accurately detects and destroys an entire ship when dropped. Not to mention the maneuver function, which ensures ships are moved appropriately within the grid space and doesn't produce errors or invalid positions; its logic would need to be tested as well as the turn management logic, which controls the rotation between players after each turn. Overall, white-box testing ensures that the system's internal operations are reliable and perform as planned.

6.4 Comparison

Black- and white-box testing follow completely different approaches to ensuring a system's quality and dependability. As mentioned previously, Black box testing focuses on the system's external behavior, with the tester interacting with the application by supplying inputs and observing outputs without being aware of the actual code. Because of this, it works exceptionally well for confirming that the system meets the users' needs and operates as intended from the user's viewpoint. White box testing, on the other hand, requires knowledge of the code's implementation and focuses on the system's logic and structure. This form of testing is intended to ensure that applications' algorithms, conditions, and execution paths function as intended.

Regarding the Battleships game, these testing methods are crucial for this application. As mentioned previously, in order to confirm that gameplay features like ship positioning and attacks work, receiving feedback allows users to make sure that functions work correctly from the user's perspective. White box testing, on the other hand, would concentrate on the internal logic underlying features, including turn management, impact detection, and unique capabilities such as drop bomb and counter maneuver. White box testing verifies the accuracy and reliability of the underlying logic, whereas black box testing ensures the system operates appropriately for the user. Overall, using both methods together can provide a more thorough and efficient testing plan.

6.5 Testable Code

Software that can be easily tested, maintained, and verified is often called testable code. This involves writing code that can be easily written and understood, not to mention organized, such that different sections can be tested separately. Testable code is important because it allows developers to easily find and correct mistakes within a small amount of time while testing. If the code is well organized, it's less likely to contain hidden flaws, thereby increasing the system's overall dependability. It becomes more challenging to isolate and find problems in code that is challenging to test, since the majority of it involves intricate logic or components.

In order to write testable code for the Battleships game, it would involve separating various system components or functions. For example, the logic to decide whether an attack is a hit or could be executed as a distinct function. Also, the same goes for the special abilities, it would be simpler to separate these two functions and test them both individually in multiple scenarios. This type of system architecture design ensures that every software component operates as intended, making it easier to deploy both Black-Box and White-Box testing techniques. With everything that has just been said, testing code is important for improving the system's reliability, maintainability, and quality.

7. Test Cases

Within this section of the report, I have created several test cases based on the final design. As of now, the game hasn't been fully developed, so the tests you will see are theoretical and show how the system should be tested if it were created. Within each test case, it includes the following Pre-condition, Input, Expected outcome, and a Description in order to make sure that the system would hypothetically work.

Test Case 1: Invalid Ship Placement

Pre-Condition: A player is placing a ship on the grid

Input: The player tries to place a ship on the outside of the grid or overlapping another ship

Expected Outcome: The system should reject the placement and show an error or custom message

Purpose: The purpose of this test is to see whether or not the ships can be placed in valid positions. This is very important as the game would become unfair if players could place ships outside of the grid or even within each other.

Test Case 2: Performing An Attack On A Grid

Pre-Condition: the game is in progress and it's one of the players' turns

Input: The player selects one of the many grid squares to attack

Expected Outcome: The system should return a message to the player saying "hit" or "miss" and update the leaderboard

Purpose: This test would check whether the system correctly determines whether an opponent's ship is on the selected square the other player chose, and whether the board has been updated after the player's move.

Test Case 3: Player Repeated Attack Blockage

Pre-Condition: A player has attacked a location on the grid already

Input: Player tries to attack the previous attack location

Expected Outcome: The system should recognize that the player has already attacked this position and prompt a message that says "location has already been attacked."

Purpose: This test checks whether the player cannot attack the same location twice. This helps keep the game fair and avoids confusion or accidental inputs from happening.

Test Case 4: Use Bomb Ability

Pre-Condition: the player has not used their bomb ability yet and has chosen what location they would like to attack, which has an enemy ship on it

Input: player selects the bomb ability and selects the location they picked

Expected Outcome: the enemy's whole ship should be destroyed, and the player should have a message appear regarding destroying their ship

Purpose: This test shows the bomb ability has worked as expected and isn't allowed to be used twice

Test Case 5: Use Manoeuvre Ability

Pre-Condition: A player has a ship on the grid and hasn't used the maneuver ability

Input: The player activates the counter maneuver ability

Expected Outcome: The player should be able to rotate the ship's position

Purpose: This test shows if the movement feature has worked properly and does not allow invalid positions

Test Case 6: End Game Results

Pre-Condition: One player has one ship remaining

Input: the other player destroys the enemy's last ship

Expected Outcome: the game should end and display the winner

Purpose: This test shows whether the game ends correctly and shows the correct information.

8. Unit Testing

In order to make sure that each component of a system is running correctly, a software called Unit Testing is used. The majority of the components that are being tested are likely brief code segments that can perform a particular activity, like functions or procedures. Its main goal is to simply ensure that a component works as it should before the whole system is put together. Doing these tests can help find problems earlier on and can make them easier to fix before the system is fully built.

Because it can increase a system's dependability and reduce the risk of errors during development, Unit testing is crucial [4]. Instead of panicking and trying to debug an entire system in one go, you can simply test each component, making it easier to determine where the issues lie. By being able to divide the system into smaller segments and testing them individually, it allows developers to organize their system perfectly.

Regarding the Battleships game, unit testing can be used to test specific game mechanics individually. For instance, the function that checks to see whether a player has hit an enemy ship could be tested on its own to see if it returns the correct result based on the grid data. Or you could also test to make sure that the ships do not overlap each other during the placement phase. And if you really wanted to, you could test each ability separately to make sure that they both follow the rules of the game. Testing an entire system in smaller segments makes it much easier to see whether each component works correctly.

9. Conclusion

To conclude this report, this report has examined both the design and testing phases of the Battleships application, demonstrating how different techniques and strategies were used to organize and design the system. Techniques such as CRC cards, Wireframes, and user stories helped divide the system into manageable chunks and ensured that the design stayed user experience-focused. Using these methods made it easier to understand how components work together and how the application would function.

This report also covers the importance of testing a system to ensure it functions correctly if it was created. Throughout the second half of the report, multiple testing methods, such as black-box, white-box, and unit testing, were discussed, along with how they can identify potential risks and mitigate them. By creating planned test cases, they were able to show how fundamental features and mechanics can be tested effectively.

Overall, combining scheduled testing with structured design approaches provides a solid foundation for a reliable, functional application. Even with limitations, such as the lack of implementation, the design process still enabled the creation of a well-defined, structured system.

10. References

- [1] R. Wazlawick, *Object-Oriented Analysis and Design for Information Systems: Modeling with UML, OCL and IFML*. Burlington: Morgan Kaufmann, 2014.
- [2] D. Rosenberg and M. Stephens, *Use Case Driven Object Modeling with UML: Theory and Practice*. New York: Apress, 2013.
- [3] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*, 4th ed. Boca Raton: Auerbach, 2013.
- [4] M. Stephens and D. Rosenberg, *Design Driven Testing: Test Smarter, Not Harder*. New York: Apress, 2010.
- [5] Agile Alliance, "What is Agile?" [Online]. Available: <http://www.agilealliance.org/>