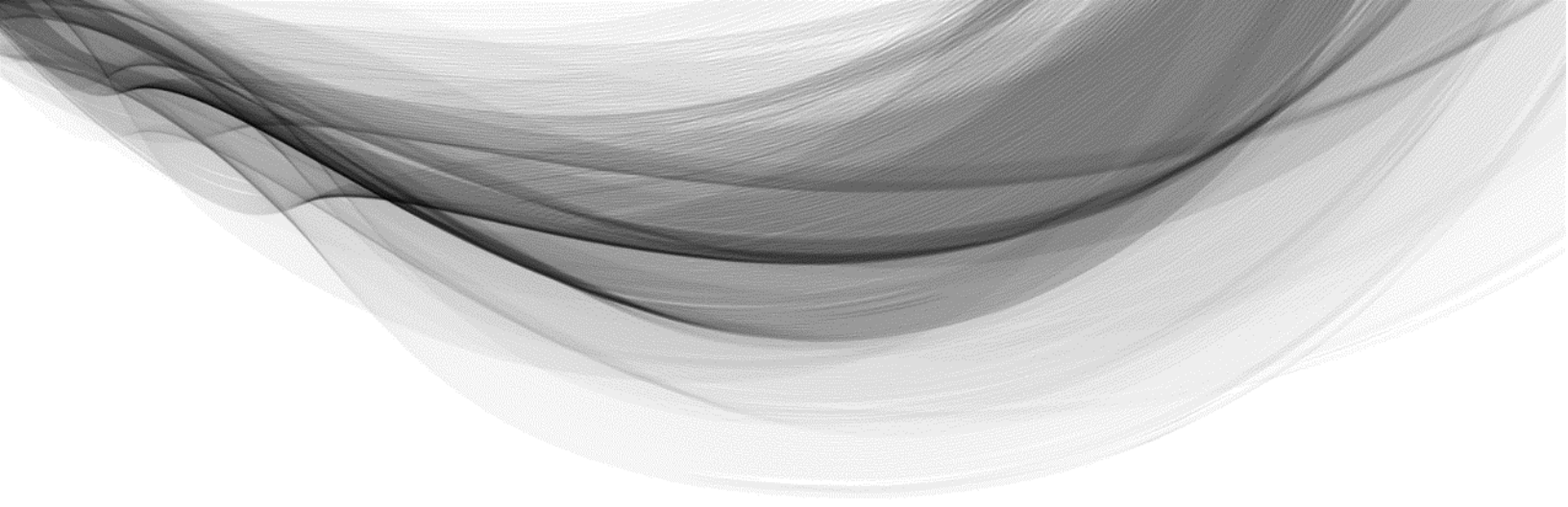




Software Engineering Design & Testing

L5 – Software Engineering Principles and Testing

Module Code: ACCB5030

BSc Computing (Software Engineering)

Contents

01. Introduction	04
02. Design Evaluation	04
2.1 Design Process Overview	04
2.2 Strengths of Design Approach	04
2.3 Weaknesses of Design Approach	05
2.4 Wireframe Design Considerations	05
03. User Stories and Story Map Evaluation	07
3.1 User Stories	07
3.2 Story Map	08
3.3 Story Prioritisation	08
3.4 Evaluation	08
04. CRC Cards Evaluation	09
4.1 CRC Cards Overview	09
4.2 CRC Card Tables	09
4.3 Evaluation	10
05. Software Testing Evaluation:	11
5.1 Software Testing Importance	11
5.2 Black-Box Testing	11
5.3 White-Box Testing	12
5.4 Comparing White-Box and Black-Box Testing	12
5.5 Testable Code	12
06. Test Cases	13
6.1 Testing Strategy	13
6.2 Black-Box Test Cases	13
6.3 White-Box Test Cases	14
07. Unit Testing	14
7.1 Unit Testing Principles	14
7.2 Unit Testing Applications	15
08. Conclusion	15

09. References	16
10. Appendix	17
Appendix A: Original Miro design board screenshot	17
Appendix B: Link to high-definition image of original Miro design board	17

1. Introduction

Within modern software development, the quality of a system's design and subsequent testing plays an essential role in the production of a reliable and maintainable system. This report critically evaluates these considerations through the development of a modified web-based implementation of the classic game Battleships. The modified design introduces additional game mechanics, including a single-use drop bomb capable of destroying an entire ship upon impact, and a one-time use counter manoeuvre allowing ships to be repositioned during play. These modifications to the base game introduce additional complexity and testing requirements, providing an appropriate context for the evaluation of user stories, CRC cards, wireframes, and testing approaches, including black-box and white-box techniques.

2. Design Evaluation

This section aims to evaluate the proposed design, reflecting on both the strengths and weaknesses of the approach. User stories are discussed and evaluated in order to assess how effectively they meet system requirements of the modified Battleships application. The role of collaboration within the design process is also considered, examining whether the chosen approach supports a structured and testable design.

2.1 Design Process Overview

The design process for the modified Battleships web application was structured around established software design techniques, particularly user stories, story mapping and Class-Responsibility-Collaborator (CRC) cards. These techniques provided a structured approach for moving from general application requirements to a more organised software design. The application's requirements were first recorded via user stories, including starting a game, placing ships and using special abilities and the end-game.

These user stories were organised into a story map, following the gameplay cycle from start to finish and are later presented in Section 3. The story map used MoSCoW prioritisation, separating requirements into Must Have, Should Have, Could Have and Would Have categories. This helped to identify essential gameplay requirements such as ship placement and special abilities such as the drop bomb, with non-essential features found to be outside the Minimum Viable Product (MVP) scope not being developed further. Once the story map was created, CRC cards were developed in order to identify potential classes, responsibilities and collaborators and are later evaluated in Section 4. This approach linked player-informed requirements to object-oriented design decisions, providing a foundation for later refinement, evaluation and testing.

2.2 Strengths of Design Approach

A key strength of the design approach was the use of user stories in keeping the application's design focused on the player's experience. Recording requirements from the perspective of the player helped to identify what the application needed to provide at each stage of gameplay, while connecting the product to the user's goals [1]. Within the Battleships application, this meant features such as placing ships and attacking grid positions directly related to the player's actions rather than being isolated functions of the application.

The story map further strengthened the design by integrating these stories into the game loop, providing a basic view of a user's experience within the application. This helped to view the application from the user's perspective, while exposing gaps where additional requirements may be needed, such as a button to end a turn.

The design was strengthened further through the use of MoSCoW prioritisation. Separating requirements into Must Have, Should Have, Could Have and Would Have categories helped to identify which features were essential for the MVP. This meant that core features such as ship placement and attacking could be prioritised over less essential features, including AI opponents and player turn timers.

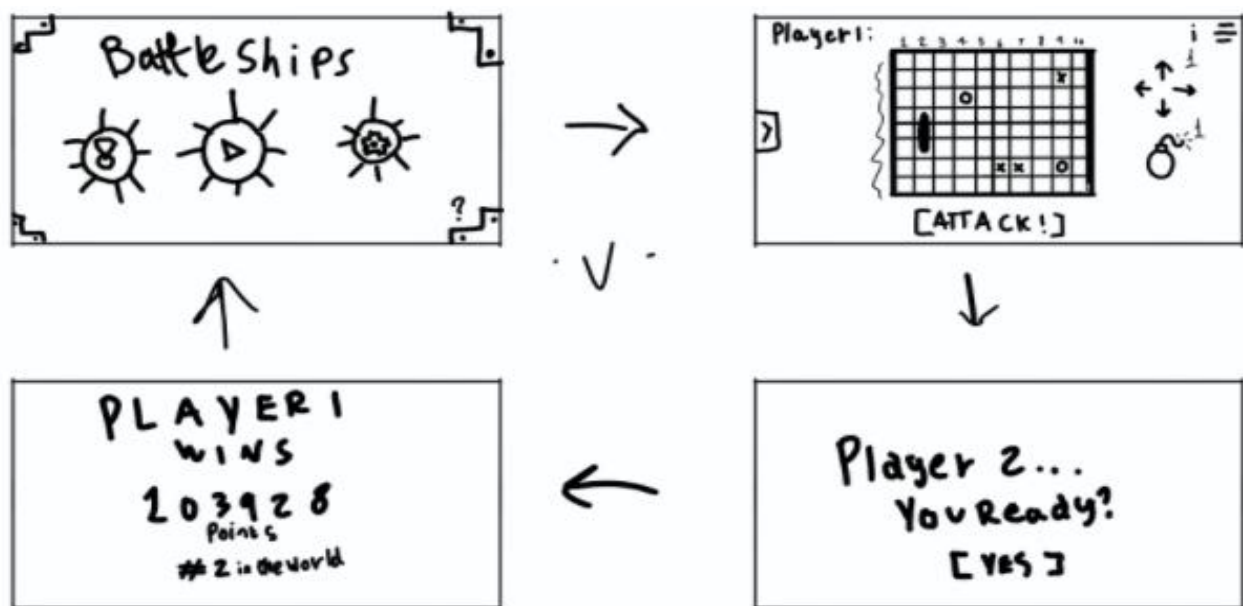
Finally, the use of CRC cards helped move the design from requirements towards implementation. Despite not providing full code-level detail, they help to identify classes that could later be represented within the application, such as Game, Player and Rules, and responsibilities that could be developed into functions. An example of this is the Ship card, responsible for storing a ship's type, health and length. This made the CRC cards useful for early object-oriented design, although further detail would still be needed before implementation.

2.3 Weaknesses of Design Approach

Despite providing a clear structure for the design of the modified Battleships application, the design approach also faced some limitations. Both the user stories and story map gave clear insight into the general game loop, but they did not completely describe the more how individual requirements would function in detail. For example, both the counter manoeuvre and drop bomb were identified as requirement, but the story map alone does not showcase how the application would respond to every possible situation, such as an attempted counter manoeuvre outside the provided grid or multiple uses of the drop bomb.

Furthermore, while the CRC cards provided an early object-oriented design, identifying classes, responsibilities and collaborators, they did not define attributes such data types, and required further development within class diagrams and test cases in order to make the design more suitable for implementation and testing.

2.4 Wireframe Design Considerations



[Figure. 1: Modified Battleships Application UI Wireframes]

The wireframe within Figure. 1 supported the design process by providing a low-fidelity visualisation of the user interface (UI). A focus was placed on structure and navigation, with four main slides containing the home, board, turn transition and end-game screens. This early design provided a foundation for the implementation of the applications UI. Despite these benefits, the wireframe was limited in its capacity to showcase dynamic behaviour, such as how a hit, miss, near miss or destroyed feedback would be presented to the user. Therefore, it was useful within early UI planning but would require further refinement before implementation.

3. User Stories and Story Map Evaluation

This section presents the selected user stories and story map for the modified Battleships application, aiming to examine how these stories represent the main gameplay requirements, support MVP prioritisation and help to define the player-focused design.

3.1 User Stories

Table. 1 presents selected user stories from the wider story map, representing the main gameplay cycle of the modified Battleships application. This includes stories for both starting and ending a game, ship status and concealing the locations of a player's ships. Lower priority features outside of the MVP such as auditory feedback for sunk ships were excluded as they are outside the MVP scope.

ID	User Story	Acceptance Criteria	Priority
US1	As a player, I want to start a new Battleships game so that I can begin playing.	The player can start a new game from the start screen.	Must Have
US2	As a player, I want to place my ships on the grid so that I can plan my layout before the game starts.	The player can place ships on valid grid squares before gameplay begins.	Must Have
US3	As a player, I want to conceal the location of my ships so that the enemy cannot view them.	The opponent cannot see the player's ship positions during gameplay.	Must Have
US4	As a player, I want to select a grid square to attack so that I can damage the enemy's ships.	On the player's turn, one square on the opponent's grid can be selected for attack.	Must Have
US5	As a player, I want to see the position and status of my ships so that I can track whether they have been hit or sunk.	The player can view their own grid, including ship positions and hit/sunk status.	Must Have
US6	As a player, I want a button to end my turn so that the enemy player can take their turn.	The player can end their turn, allowing the opponent's turn to begin.	Must Have
US7	As a player, I want to use a drop bomb once per game so that I can destroy an enemy ship with one attack.	The player can use the drop bomb once instead of a normal attack.	Must Have
US8	As a player, I want to move my ship in response to a possible hit so that I can avoid the attack.	The player is given the option to move a ship in response to a potential hit.	Must Have
US9	As a player, I want to see the winner of the game so that the match has a clear ending.	The game-over screen displays the winning player.	Must Have

[Table. 1: Selected User Stories for the Modified Battleships Application]

3.2 Story Map

Table. 2 presents the selected user stories within a story map in order to demonstrate the gameplay cycle of the modified Battleships web application. This helped structure the software's design around the player's journey, beginning with user stories such as starting a game and placing ships, before moving into turn-based attacks, special abilities and the end-game. By arranging the user stories in this manner, the map demonstrates how each requirement fits into the game cycle as opposed to being treated as an isolated feature.

Start Game	Place Ships	Player Turn	Abilities	End-game
US1: Start a new Battleships game	US2: Place ships on the grid	US4: Select a grid square to attack	US7: Use a drop bomb once per game	US9: See the winner of the game
	US3: Conceal ship locations from the enemy	US5: View ship position and status	US8: Move a ship in response to a possible hit	
		US6: End turn so the enemy player can take their turn		

[Table. 2: Story Map of Selected User Stories for the Modified Battleships Application]

3.3 Story Prioritisation

The selected user stories were prioritised using the MoSCoW method in order to define the MVP. Within this design as shown in Table. 1, US1, through to US9 were labelled as Must Have requirements as they support the base game loop. For example, the game must allow players to start a match, place their ships and end a game in order to display a winner. Furthermore, enhancements such as the drop bomb and counter manoeuvre must be included. Despite being amendments to the base game they can be considered core gameplay features.

Lower priority features from the wider story map located in the appendix, such as sonar detection and audio cues, were treated as Should Have, Could Have and Would Have requirements, and found to be outside the MVP scope. This approach to prioritisation ensured the scope of the design more refined, separating essential gameplay requirements from optional additions.

3.4 Evaluation

The selected user stories were beneficial to the application's design as they linked the application's requirements to specific player goals. In place of describing features as functions, the user stories showcased the needs of the player while describing why each requirement mattered. An example of this is US2 within Table. 1, which placed a focus on placing ships within selected cells, identifying both the benefit to the player when controlling the initial layout, and the system requirement to ensure ship placement before play commences.

These benefits were enhanced further with the creation of a story map, organising user stories within the wider game loop. This enabled a visualisation of how the player would move throughout a game, from starting a match, placing ships, taking turns and reaching the end of a match. By showing the sequence of actions in which the game would take place,

the map assisted in identifying any gaps within the game loop. In conjunction, the two artefacts improved the design by making the player's journey clear, outlining the MVP and creating requirements that could later be tested.

4. CRC Cards Evaluation

This section presents the selected CRC cards for the modified Battleships web application, evaluating how effectively they translate user stories into object-oriented classes, responsibilities and collaborators.

4.1 CRC Card Overview

Class-Responsibility-Collaborator (CRC) cards are a low-tech and collaborative way to design class structure. These cards are used to identify potential classes for later implementation, describing their responsibilities and potential collaborators needed to meet said responsibilities, providing a direct link between early user stories and later object-oriented design [2]. For example, user stories related to ship placement or attacking grid squares help to identify possible classes such as Board and Ship. This makes CRC cards helpful when transitioning player requirements into structured object-oriented design.

4.2 CRC Card Tables

Tables. 3 - 7 present refined selected CRC cards from the wider project board found in the appendix. These cards were chosen as they represent the main classes required to support the core game loop of the Battleships web application. This includes Game, Player, Ship, Board and Rule cards.

Game	
Responsibilities:	Collaborators:
Control game actions.	Player
Switch turns between players.	Board

[Table. 3: Game CRC Card]

Player	
Responsibilities:	Collaborators:
Placing ships on board.	Board, Ship
Tracking moves.	Game
Tracking use of drop bombs and counter manoeuvres.	Game

[Table. 4: Player CRC Card]

Ship	
Responsibilities:	Collaborators:
Store ship length.	
Store ship health.	
Store ship type.	

[Table. 5: Ship CRC Card]

Board	
Responsibilities:	Collaborators:
Store ship locations.	Ship
Store ship rotations.	Ship
Display grid state to player.	
Ensure ships do not overlap or exceed borders.	
Detect player input on attack.	Ship, GameLog
Record hit ships.	Game
Record sunk ships.	Game

[Table. 6: Board CRC Card]

Rules	
Responsibilities:	Collaborators:
Validate ship placement rules.	Board, Ship
Enforce one time use drop bomb.	Player
Enforce one time use counter manoeuvre.	Player
Determine win condition has been met.	Game, Player, Ship, Board

[Table. 7: Rules CRC Card]

4.3 Evaluation

The use of CRC cards enabled the translation of earlier user stories into a clear object-oriented structure, with each selected class representing a distinct subsection of the wider application. For example, the Board card places a focus on the grid's state, ship locations and detected attacks, while the Ship card stores data pertaining to a ship's attributes, such

as length, health and type. This approach ensured that no class is given an overwhelming amount of responsibility with the addition of providing a foundation for specific test cases.

The CRC cards also help to identify potential methods and attributes that could later be used within class diagrams, a visual representation for Unified Modelling Language (UML) that shows a class structure [2]. For example, a class diagram for Ship may contain attributes such as `shipType` and `health` alongside methods such as `takeDamage` and `getShipSize` as demonstrated within Figure. 2.

Ship
<code>-_type : shipType</code> <code>-_health : int</code> <code>-_size : gridPoint[]</code>
<code>+Ship(shipType : ShipType, position : GridPoint[])</code> <code>+takeDamage(location : gridPoint) : void</code> <code>+getShipSize() : int</code> <code>+isSunk() : bool</code>

[Figure. 2: Ship Class Diagram]

While CRC cards help to identify classes, responsibilities and collaborators, they are limited in their ability to demonstrate how objects interact during gameplay. For example, the rules card identifies the placement and special abilities responsibilities but it cannot alone describe how they interact with the Board, Ship, Player and Game classes. This demonstrates the use of CRC cards as an early design tool, but they require further support from artefacts such as class diagrams in order to provide a more complete description for implementation.

5. Software Testing Evaluation

This section evaluates the importance of software testing within the proposed Battleships web application. Both black-box and white-box testing practices are taken into consideration in order to check user derived behaviour and internal logic respectively. The importance of testable code is also examined in relation the production of a reliable and maintainable system.

5.1 Software Testing Importance

Software testing refers to the process of checking correctness, completeness and accuracy of an implemented software design. This enables developers to evaluate a system and verify whether an application's requirements have been met and identify the differences between actual and expected results [3]. Within the modified Battleships application, this is important as features including ship placement, drop bombs and counter manoeuvres all depend on a set of clear rules being implemented correctly. Without rigorous testing, unexpected behaviours could arise such as overlapping ships and multiple uses of special abilities, diminishing the user's experience. This testing directly links to the acceptance criteria within the earlier user stories displayed in Table 1.

5.2 Black-Box Testing

Black box testing is a functional testing technique that designs test cases based on information from the specification. Software testers do not have access to the internal source code and instead interact with the application as a user would

[4, 5]. Within the modified Battleships game, this may involve selecting a grid square to attack, with the expected result being clear feedback in relation to a hit, miss or destroyed ship.

This approach to software testing is useful as it accurately reflects how the end user will interact with the application. However, it is limited in its ability to reveal why errors may have occurred internally. For example, the application may allow users to place ships outside of the allowed grid, but black-box testing alone will not show whether the fault is caused by the grid logic, ship data or rule validation.

5.3 White-Box Testing

White-Box testing is a structural testing technique that designs test cases using information derived from the source code of a given application. Unlike black-box testing, it requires knowledge of how the system works internally. A software tester may execute methods with certain parameters, exposing the inner workings of an application while examining data flow. [4]

Within the modified Battleships application, white-box testing could be used to check the logic behind ship placements, hit detection and drop bomb behaviour. An example of this could be a test that examines whether the attack function correctly compares a ship's position with the selected grid coordinate. It may also be used to check whether the counter manoeuvre correctly moves a selected ship by one grid square without exceeding the outer grid boundaries. This makes white-box testing useful for identifying errors that may not be obvious using the user interface alone.

5.4 Comparing White-Box and Black-Box Testing

Both black-box and white-box testing play an essential role within software testing, allowing developers to examine an implemented design from multiple perspectives. Black-box testing excels in checking whether an application behaves as expected from the user's perspective, while white-box testing can be used to expose logic errors within the source code. were to become too focused on white-box testing, usability issues may be missed. Alternatively, if a tester were to become too focused on black-box testing, internal logic errors may be missed. An example of this could be a displayed "hit" message while a ship's health remains the same. Therefore, a combination of both approaches is required in order to test a given application effectively.

5.5 Testable Code

Testable code can be broken down into four key points [5]:

- **Simple:** Simple components and applications are inherently easier to test.
- **Observable:** Internal structures and data should be visible in order to determine if a test passes or fails.
- **Control:** In the case of an application possessing a threshold, the ability to set and adjust said thresholds makes testing easier.
- **Knowledge:** Testers should be able to refer to documentation in order to ensure test results are correct.

Within the modified Battleships application these points are relevant as the system contains multiple rule-based behaviours that must be checked accurately. Simplicity is supported by separating responsibilities across classes, such as Ship, Board, Rules, Game and Player as demonstrated within the CRC cards in Section 4. Observability is also important

as testers need to be able to confirm whether a ship's health, grid position and special ability usage have been correctly updated after a player's action.

Control is required to enable testers to set up specific scenarios within the application in order to test additional features, including the drop bomb and counter manoeuvre. Knowledge is provided through a combination of user stories, acceptance criteria and CRC cards. Together these principles make the application more testable as expected outcome can be compared against actual results.

6. Test Cases

This section presents selected black-box and white-box test cases for the proposed design of the modified Battleships web application. These test cases place a focus on the core gameplay requirements and amended mechanics, including ship placement, drop bomb usage and counter manoeuvre usage and win condition checks.

6.1 Testing Strategy

The proposed testing strategy combines both black-box and white-box testing in order to check behaviour related to user input and internal system logic. Black-box tests are used to determine whether the application responds correctly to player actions such as placing a ship, while white-box tests are used to examine the internal logic behind said behaviours, including grid validation and hit detection. This combined approach ensures that the modified Battleships application from both the player and source code's perspective.

6.2 Black-Box Test Cases

ID	Feature	Pre-condition	Input	Expected Output	Purpose
BB1	Valid ship placement	Game has started and player in setup phase.	Player places a ship without overlapping another.	Ship is placed successfully within the grid.	Check valid ship placement.
BB2	Invalid ship placement	Game has started and player in setup phase.	Player attempts to place ship outside grid / overlapping ship	Placement rejected and player receives error message.	Check invalid ship placement is prevented.
BB3	Attack feedback	Both players place ship and it is player's turn.	Player selects grid square on opponent's board.	Application displays, hit, miss, near miss or destroyed feedback.	Check attack results are communicated to player.
BB4	Drop bomb usage	Player has not used drop bomb.	Player attacked grid square containing enemy ship with drop bomb	The enemy ship is destroyed and drop bomb becomes unavailable.	Check one use drop bomb rule works from players perspective.

BB5	End-game Display	One player has sunk all ships.	Final successful attack completed.	Game displays winning player within end-game screen.	To check match ends clearly when win condition met.
-----	------------------	--------------------------------	------------------------------------	--	---

[Table. 8: Black-box Test Case Table]

6.3 White-Box Test Cases

ID	Feature	Pre-condition	Input	Expected Output	Purpose
WB1	Grid boundary validation	Board validation function available.	Call validation function with coordinates outside grid.	Function returns false and rejects placement.	Check invalid coordinates are rejected.
WB2	Ship overlap detection	Ship occupies selected cell.	Call validation function with overlapping coordinates.	Function returns false and rejects placement.	Check ship placement logic for overlapping ships.
WB3	Hit detection logic	Ship stored at know grid coordinate.	Call attack function using occupied coordinate.	Function returns hit and updates ships health / status.	Check attack logic compares selected coordinate with ship position.
WB4	Drop bomb rule	Player has not used drop bomb and target contains ship.	Call drop bomb function on target coordinate.	The ship is marked destroyed and drop bomb availability equal to false.	Check drop bomb destroys ship and cannot be reused.
WB5	Win-condition logic	One player's ships are all marked as sunk.	Call win condition function.	Function returns winning player.	Check system identifies when match has ended.

[Table. 9: White-box Test Case Table]

7. Unit Testing

This section describes the main principles of unit testing, examining how they could be used to perform some of the white-box test cases within Table. 9 of Section. 6.3.

7.1 Unit Testing Principles

Unit testing can be described as a small piece of code designed to test a specific aspect of an application's source code in isolation. This approach typically involves testing each individual class and method separately. The goal of unit testing is to verify that each unit of an application's source code executes as expected, independent of the rest of the system [5].

7.2 Unit Testing Applications

Within the modified Battleships application, unit testing could be used to test methods linked to core game rules. An example of this is the Ship class, which could include an `isSunk()` method that returns true when all grid squares containing a single ship have been hit by a player. A Board class could contain a `validatePlacement()` method, ensuring ships are only placed within the 10x10 grid and do not overlap.

This form of testing aligns closely with white-box testing, requiring knowledge of the internal source code. Instead of checking what the user sees, unit tests examine individual methods and help validate the returned results. For example, if a ship is undamaged after a direct hit, unit testing can help determine if the problem stems from the Ship class or somewhere else, such as the Rules class. This supports the maintainability of the modified Battleships application, allowing unexpected behaviours to be located and patched.

8. Conclusion

In conclusion, the modified Battleships web application provides an appropriate context for evaluating the application of both software design and testing principles. The use of user stories and story mapping aided in connecting the applications requirements with grounded player goals. MoSCoW prioritisation helped further by outlining the MVP scope. CRC cards were useful in translating these requirements into object-oriented classes, responsibilities and collaborators, which were further strengthened using class diagrams.

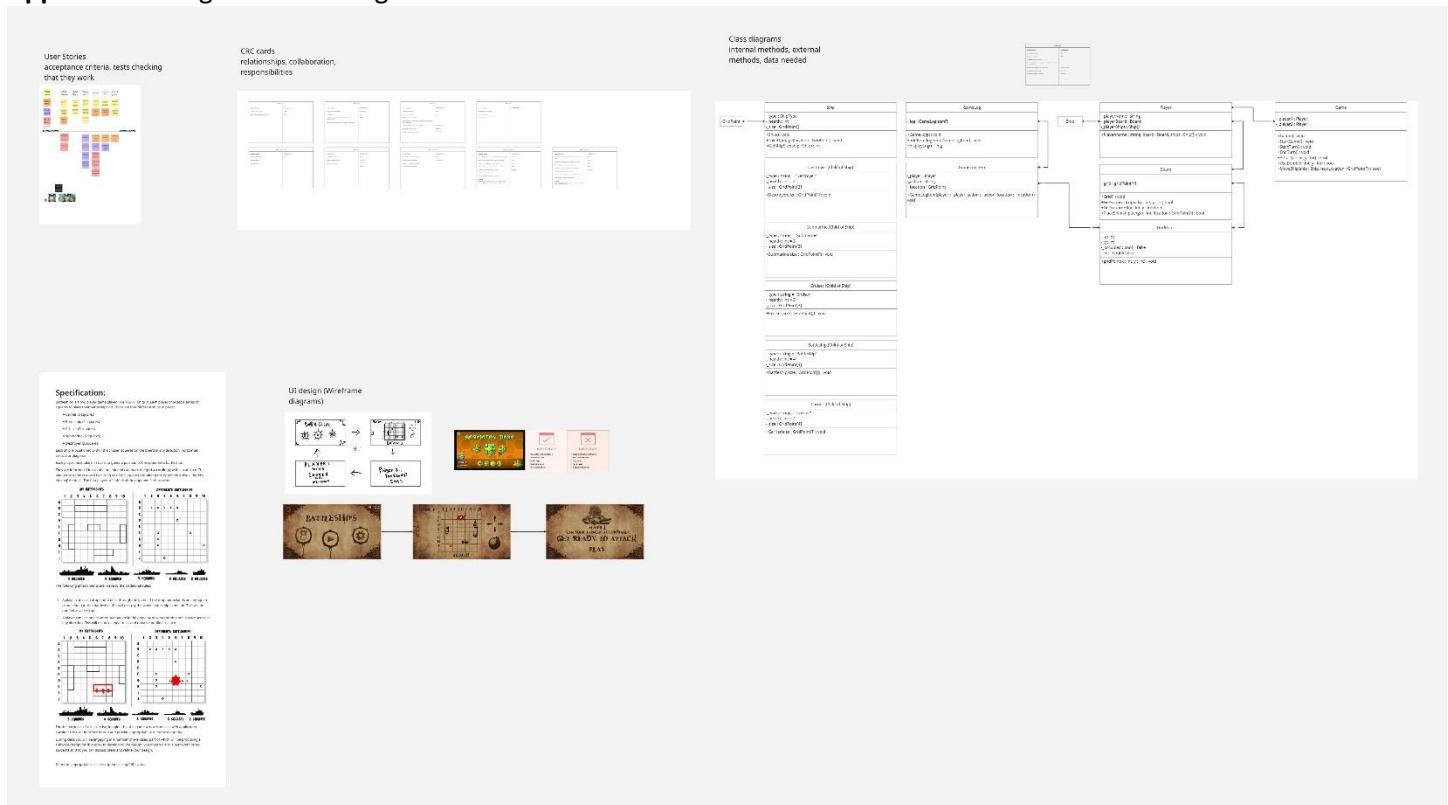
Testing was shown to be essential in producing a maintainable product, ensuring a smooth user experience alongside accurate rule enforcement and consistent behaviour within classes. Black-box testing supported validation from the perspective of the user, while white-box and unit testing aided in examining the inner logic of the system, including a ships placement, hit detection and win conditions. Overall, the outlined design and testing considerations support a structured, testable and maintainable application.

9. References

- [1] D. Gruen, T. Rauch, S. Redpath, and S. Ruettinger, "The Use of Stories in User Experience Design," *International Journal of Human-Computer Interaction*, vol. 14, no. 3–4, pp. 503–534, Sep. 2002, doi: <https://doi.org/10.1080/10447318.2002.9669132> (Accessed: April. 18)
- [2] M. Huntly, "Software Design CRC and Class Diagrams," *Software Engineering Principles and Practice*, University of Wales Trinity Saint David, lecture slides, 2026 (Accessed: April. 24)
- [3] A. Uddin and A. Anand, "Importance of Software Testing in the Process of Software Development," *IJSRD - International Journal for Scientific Research & Development*, vol. 6, no. 12, pp. 141–145, 2019
https://www.researchgate.net/profile/Abhineet-Anand/publication/331223692_Importance_of_Software_Testing_in_the_Process_of_Software_Development/links/5c6d13464585156b570ae061/Importance-of-Software-Testing-in-the-Process-of-Software-Development.pdf (Accessed: April. 25)
- [4] S. Nidhra and J. Dondeti, "Black Box and White Box Testing Techniques — A Literature Review," *International Journal of Embedded Systems and Applications*, vol. 2, no. 2, pp. 29–50, Jun. 2012, doi: 10.5121/ijesa.2012.2204
https://www.researchgate.net/profile/S-Nidhra/publication/276198111_Black_Box_and_White_Box_Testing_Techniques_-_A_Literature_Review/links/570e313f08ae2b772e46aa40/Black-Box-and-White-Box-Testing-Techniques-A-Literature-Review.pdf?sg%5B0%5D=started_experiment_milestone&origin=journalDetail&rtd=e30%3D (Accessed: April. 25)
- [5] M. Huntly, "Software Testing," *Software Engineering Principles and Practice*, University of Wales Trinity Saint David, lecture slides, 2026 (Accessed: April. 25)

10. Appendix

Appendix A: Original Miro design board screenshot:



Appendix B: Link to high-definition image of original Miro design board for download: https://uwotds-my.sharepoint.com/:i:/g/personal/2317034_student_uwotds_ac_uk/IQCyi4OQTWpXRBoTw2iefJRNAekaf35X-mErvkMoEVHEmrU?e=TsZhfhf