



# ACCB5030: SOFTWARE ENGINEERING PRINCIPLES & TESTING

Assignment: Software Engineering Design &  
Testing

Joe Frohwein

2315545

2315545@student.uwtsd.ac.uk  
Computing (Software Engineering)

## Contents

|                                                     |    |
|-----------------------------------------------------|----|
| 1. Introduction.....                                | 2  |
| 2. Evaluation and reflection of design.....         | 2  |
| 2.1. Design Process .....                           | 2  |
| 2.2. User Stories .....                             | 4  |
| 2.3. CRC Cards .....                                | 5  |
| 3. Software Testing .....                           | 7  |
| 3.1. The Importance of Software Testing.....        | 7  |
| 3.2. Black and White Box Testing .....              | 7  |
| 3.3. “Testable” Code .....                          | 8  |
| 4. Test Cases for the Battleships application ..... | 9  |
| 5. Unit Testing .....                               | 11 |
| 6. Conclusion.....                                  | 12 |
| 7. References .....                                 | 13 |

## 1. Introduction

Battleships is a grid-based, 2-player board game in which each player places a variety of 'battleships' on their grid, before taking turns guessing the positions of their opponents' battleships, hoping to 'hit' and eventually 'sink' all their ships. Once a player has sunk all their opponents' ships; they are the winner. This assignment introduces several additions to the traditional version of this game. The biggest change is its transition from physical board game to a web application, requiring software to be developed and tested to allow the turn-based game to be played. Additionally, some extra game mechanics have been introduced. The first new mechanic is a 'Drop bomb', which can be used once per game and, if dropped on an enemy ship, will destroy it in one hit, returning a "destroy" message to the user. The second new mechanic is a 'counter manoeuvre', which can also be used once per game, allowing a ship to be moved one square across in any direction to avoid an incoming enemy attack, which then must be notified as a "near miss" to the opponent.

The design process for an application like the specified Battleships involves creating User Stories, a user story map, and CRC (Class, Responsibility, Collaborators) cards. These design elements allow the design team to define and structure the functional requirements of the system, allowing them to ensure that the application has user-friendly interfaces, robust logic and ultimately fulfils the intended functionality of the application.

This report will evaluate the design process, focussing on the strengths and weaknesses of the approach taken, as well as reflecting on the experience. It will discuss the User Stories, particularly evaluating the "benefit" of the story. It will also evaluate the CRC card design approach and how effective it is at translating from requirements into testable code. Additionally, this report will explore software testing, including both black and white box tests, as well as the importance of writing testable code and the different types of tests that can be performed on an implementation such as this one. Finally, unit testing and its application will be discussed.

## 2. Evaluation and reflection of design

### 2.1. Design Process

The design process for this Battleships web application involved collaborative work within a small team to develop a structured design. This design involved creating a series of design tools, including User Stories, CRC cards and sequential diagrams. This design process highlighted both strengths and weaknesses within the approach.

One main strength of the approach was the use of communication and teamwork, allowing us to collectively brainstorm ideas together, providing multiple perspectives to help identify potential issues early and avoid having to re-design later on in the process, improving the overall quality of the design. For example, there was discussion about

how the 'counter manoeuvre' function would work, discussing whether the ship would only be able to move lengthways as a ship typically moves, or if it would be able to move sideways, as well as discussion around how a ship that has already been partly hit would behave and whether it would maintain its partly-hit parts or if it would move away from the hits. Over time, this communication would allow us to refine this functionality, as well as many others, and improve them, ensuring that they function properly.

Another strength was the use of the structured design tools. Using these tools, particularly the User Stories and the CRC cards, allowed the system requirements, from both the perspectives of the user and the developer to be clearly defined, as well as illustrating a clear breakdown of all entities, their responsibilities and what other entities they will collaborate with.

However, there were also limitations to this approach. One challenge faced, which is directly linked to the first advantage that was mentioned, was the need to ensure consistency throughout the team, as all team members had slightly different interpretations of the requirements of the application. This would be achieved through significant amounts of discussion so that all members' interpretations of the requirements were aligned and were all designing towards the same goal. Failure to maintain consistency within the team throughout the design phase could lead to inconsistency in the design. This inconsistency could occur when designing the User Stories, failing to properly establish the user requirements in advance, or when translating from the User Stories into the CRC cards, failing to establish what entities would interact with others.

Another drawback would be the time constraints. As previously mentioned, to remove inconsistency, there would have to be a large amount of discussion to clarify the application's requirements. Although this phase of planning is an absolute necessity of a starting point, it can be very time consuming. Unfortunately, the produced work reflects this, as there was no sequential diagram produced within the timeframe. If a similar project with the same design process was to be done again, it would be ensured that the discussion and planning phase was streamlined to assign enough time to the design phase as possible, ensuring that the useful design tools were all utilised and created effectively and efficiently.

Overall, the design approach that was used was effective in encouraging collaboration and teamwork, leading to the creation of a structured, functional design that could be used when developing the application. However, the design process could still be improved upon by streamlining communication and prioritising time management to efficiently design the application and its requirements.

## 2.2. User Stories

User Stories “describe a user and the reason why they need to use the service you’re building” [1], typically following the format of “As a [ROLE], I want [GOAL], so that [BENEFIT]”, ensuring that each requirement provides some form of value to the user. For this application, the end user would be the player, so the User Stories were designed accordingly. Some examples of the User Stories created are:

- “As a player, I want to place all of my ships on the board during a timed setup phase so that I am ready to begin gameplay without delaying the match.”
- “As a player, I want to aim my shots directly to where I believe the opponent to be so that I can win the game.”
- “As a player, I want to drop a bomb, once per game instead of a regular turn, which will sink any ship it hits instantly.”

Using these User Stories, a User Story Map was created which allowed each set of User Stories to be organised into a structured sequence. The map was very useful to the overall design process as it helped to lay out the flow of the application and how each section will appear in the timeline of each game. The map created consisted of the main goal, which was “Allow users to complete a game of Battleships that includes new game mechanics”. This was followed by the main flow of activities, which were the main actions such as “Start game”, “Place ships” etc. Then followed the User Stories, consisting of more detailed activities such as the examples given previously. This structure allowed the application to be broken down and separated into sections, so that each section could be focussed upon and ensured that it meets the requirements. This made it far easier to work on as it meant that each section could be looked at as its own separate part and made it easier to evaluate separately, rather than as one big overall game, ensuring that each section was designed effectively and had no missing features.

The ‘Benefit’ aspect of the User Stories is particularly important, as it highlights why that feature is necessary and how it provides value to the user. For example, in the User Story “As a user I want to receive a response whether my last turn was a “hit”, “miss”, “sink” or “near miss””, the benefit is that the player can track their progress, keeping them engaged in the game, as without any feedback of hits and misses, the player is unable to monitor their progress and make decisions based on their previous guesses. This demonstrates the importance of implementing the benefit, as it will improve the general engagement and playability of the game.

Focussing on the User Stories and the map created, they were able to be used to identify all of the main goals and activities, as well as identify several of the more specific user requirements that would be needed within this application. User Stories were written and arranged in a way that the overall flow of the application could be

identified and demonstrated, and each section's requirements could be shown to show exactly what actions must happen at all points of the game's timeline.

Although the User Stories provide such value, there were some drawbacks too. As can be seen from the User Story map, some User Stories were vague, lacking technical detail, meaning the amount of value that they provide can be limited if not written effectively. Additionally, some User Stories can also be very descriptive and lack any form of instruction on how to implement each User Story into the application. If this section of the design phase was to be repeated, it would be ensured that all User Stories are written with efficient detail, ensuring to keep them short whilst highlighting the benefits of each story.

Despite these minor drawbacks, the User Stories were able to be used to create a user-focussed map, outlining the overall structure of the application and each individual game, and ultimately led to the creation of the CRC cards.

### 2.3. CRC Cards

CRC Cards are "a design tool used in object-oriented programming to capture and organize information about classes, their responsibilities, and their collaborations within a software system" [2]. These cards focus on three main parts:

- Class – The name of the object (e.g. player)
- Responsibilities – What the class does (e.g. Use the 'Drop bomb' feature)
- Collaborators – Other classes it interacts with (e.g. ship)

During this design process, these CRC cards were used to break down the system into smaller, more manageable components, so that each class could be easily defined.

This aided the visualisation of the application as a whole, by demonstrating how different entities interacted with one another to form one cohesive application.

Additionally, the CRC cards were able to provide the next step in the design process by translating from User Stories into a more structured representation of the application, with added technical detail, outlining the systems components and interactions. This meant that all entities that were identified within the User Stories were being considered when developing the overall application.

Through the creation of these CRC cards, the classes identified were Player, BoardGrid, Ship, Game, Coordinates, TurnManager, SpecialAction and ScoreManager. Each class was assigned specific responsibilities which reflected its role within the game, as well as collaborators that showed how the class interacted with others. Key classes to mention include the Player class, which is responsible for all major gameplay mechanics including placing ships, guessing opponent's ship coordinates and activating the special actions such as the 'Drop bomb' or the 'Counter manoeuvre', and collaborates with the BoardGrid and Ship classes to complete these mechanics. Another key class is the BoardGrid which manages the game and holds information

regarding the player's ship placements as well as any guesses that are made, whether they be successful or unsuccessful hits. The ship, another key class, stores attributes such as ship size and its current status in terms of what parts of the ship have been hit, which helps the system to determine whether the ship is 'sunk' yet or not.

Other classes, such as the TurnManager and the ScoreManager, despite not being physical entities, are still vital as they contribute to the management and flow of the game, allowing turns to be alternated between players correctly and to ensure that game progress is tracked accurately. Additionally, the SpecialAction class is also important to implement the new game mechanics, enforcing that each is only available to each player once per game and ensuring that each mechanic works as designed.

The use of CRC cards was very beneficial to this design process as they allowed the system to be further organised from the User Stories setup while clearly identifying responsibilities and collaborations of each class. This ensured that each class of this application, such as the Player, the BoardGrid and the Ship, were all clearly defined within the system. They were also very helpful as they provided enhanced visualisation, clarity and understanding of the system and how it operates, particularly through the demonstration of how classes interact. The creation of the CRC cards also encouraged teamwork as different ideas and perspectives were collectively brainstormed, leading to a more complete design. Additionally, the CRC cards were able to be used as an iterative design tool, being updated and evolved as more classes and their collaborations were discussed and developed, allowing the User Stories to be fully transitioned into a more structured, technically detailed representation of the system [2].

Despite these benefits, there are some drawbacks to CRC cards too, which particularly focus on whether the cards contain sufficient detail to enable the code to be implemented and tested. One main limitation of the approach is the ambiguity when advancing into the development of the application. Although these cards provide responsibilities and collaborators, they don't include code-specific knowledge, such as methods, data structures, or inputs and outputs, meaning that a team of developers may have different interpretations of the classes and what they require within the system. This will also prevent the possibility of using these cards for testing, meaning that further work is required before developing the code so that it can be implemented and tested.

Overall, the CRC cards provide a structured, object-oriented design, where responsibilities and collaborations of each entity within the system are clearly defined. This structure ensures that the system is manageable and fulfils the system requirements outlined by the User Stories.

### 3. Software Testing

#### 3.1. The Importance of Software Testing

Software Testing is “the process of evaluating and verifying that a software product or application functions correctly, securely and efficiently according to its specific requirements” [3].

Software Testing is critical to the software development process. Failure to perform software testing when designing an application such as Battleships can have severe consequences to the final product. The testing process ensures that software is reliable and secure, as well as providing high performance, ultimately saving money and building customer trust. This process allows bugs to be detected early and user requirements to be validated, therefore improving the overall performance, reliability and user-satisfaction of the final product.

Cost effectiveness is one of the main reasons that software testing is so critical. Detecting bugs early in the design process will help to prevent costly fixes later in the design process, or even costlier post-release fixes. If a client returns to the software developer with glitchy software, the developer will bear the cost of fixing and re-releasing the software, which can be a huge loss for a business.

Security is another of the main reasons that highlights the importance of software testing, allowing developers to identify vulnerabilities in the system during the early stages of development, as well as assisting in solving security benchmarks.

Another, and arguably the most important reason, is the quality of the software. As previously mentioned, software testing will detect bugs and defects in the code and allow them to be fixed before releasing to the end-users. Failure to rigorously test the code and remove these bugs will lead to glitchy, unreliable software. Tested code will result in high-performance, reliable software, reflecting a high overall quality, which will also provide high customer satisfaction which is another prime example of why software testing is so crucial [4], [5], [6], [7].

In a system such as the specified Battleships application, involving several rules and interactions between entities, rigorous testing ensures that the gameplay will be fair, accurate and consistent for all players. In this example, an untested release of the Battleships application could have bugs such as incorrect hit detection, inconsistent sequencing in player turns, or broken game logic in the ‘Drop bomb’ or ‘counter manoeuvre’ functionalities, which will all reduce the overall quality of the software.

#### 3.2. Black and White Box Testing

Two of the main methods of identifying defects in code are Black Box testing and White Box testing.

Black Box testing is when testers assess the functionality of an application without knowing its internal code or structure, focussing primarily on the inputs and outputs. The primary goal of black box testing is to validate the output of the software based on various inputs, which are entered through its user interface or through an API, ensuring that they meet specification and behave as expected. The term “Black Box” is a reference to the fact that the tester cannot see the internal code of the software, much like how a black box cannot be opened or examined.

Black box examples are generally simpler, providing various inputs and expected outputs to be returned from them. For the Battleships application, one example would be to try to place a battleship at a certain position, then upon starting the game to check and ensure that the battleship is exactly where it was originally placed. Another simple example would be to guess a coordinate that the tester knows is either a miss or a hit and ensuring that the actual output correlates to the expected output of “miss” or “hit”. Lastly, the ‘Drop bomb’ could be tested by using it on a position that the tester knows will return a hit and ensuring that the output returns a “destroy” message.

White Box testing is when testers examine the internal code, logic and structure of an application, focussing on the software’s infrastructure and integrations. This method is performed by reading through the code, identifying the different execution paths, and designing ‘test cases’ which will force different inputs into the system, allowing the tester to test that all code structure elements, such as loops, conditionals and variables, behave as expected. In contrast to Black Box testing, the term “White Box” refers to the fact that the tester has full access to the code, similarly to how a white box can be opened and checked.

White Box examples are generally slightly more complex, involving testing the overall behaviour of a certain functionality. For the Battleships application, one example would be to hit the last position of a certain battleship, and debug the loops and conditionals of the code as it checks each of the ship’s coordinates and whether they have been hit or not, and ensuring that the battleship is sunk and notifies both players correctly. Additionally, the ‘counter manoeuvre’ could be tested by activating it and debugging to ensure that each of the ship’s coordinates are offset by 1 in the direction that the user entered, ensuring that the loop behaves as intended [8], [9], [10].

### 3.3. “Testable” Code

“Testable” code refers to code that is designed in a way that makes it easy to test and verify. This means that the code is structured so that individual components can be isolated and evaluated to be tested individually without interference from other components.

One key characteristic of testable code is modularity, which means the code is organised into separate, discrete units with minimal dependencies, which will allow the

testers to write focussed test cases in order to test specific behaviours and outputs without interference from other components. Another key characteristic is clarity, particularly in terms of inputs and outputs, while ensuring that code is understandable and its purpose is clear, allowing testing to be simple and straightforward.

The importance of testable code in software engineering is significant. The main reason for writing testable code is that it'll improve the quality of the code through its assistance in detecting bugs early in the development phase. Testable code can easily be regularly tested during the design phase, allowing the developer to identify bugs and rectify them at the source, reducing the need for subsequent code to be altered because of any fixes. Another reason that highlights the importance for writing testable code is that it'll reduce development costs. Although writing testable code will require more effort initially, similarly to the previous example, it'll simplify and streamline the debugging process, as well as improving its maintainability, making it easier to modify and extend, which will all contribute to the overall cost effectiveness of the design process. Additionally, testable code will be easier to understand due to its modularity, clarity, reusability and independence [11], [12].

#### 4. Test Cases for the Battleships application

In order to ensure that the application functions properly, a set of test cases must be designed in order to validate the core gameplay mechanics, including ship placement, turn management, hit, miss and sink detection, as well as the added gameplay mechanics. Each test case must include pre-conditions, inputs, expected outputs and a description.

- Test case 1 – Ship Placement
  - Pre-conditions: Pre-game with an empty grid
  - Inputs: Place a ship on a specific set of coordinates on the grid
  - Expected Outputs: A ship of the same size is successfully placed on the same coordinates
  - Description: Verifies ships can be placed correctly within the grid and tests whether ships can be placed in invalid positions (out of bounds, on top of existing ships)
- Test case 2 – Turn management
  - Pre-conditions: In-game
  - Inputs: End of one player's turn
  - Expected Outputs: Automatically switches control to the second player and disables any actions of the first player
  - Description: Ensures turn-based gameplay
- Test case 3 – Hit Detection
  - Pre-conditions: In-game with all ships placed

- Inputs: Guess aimed at a pre-defined position where a ship has intentionally been placed by the opponent
- Expected Outputs: A “hit” message is displayed to the player and marked as a hit on both player’s grids
- Description: Confirms that the system correctly identifies a successful hit on a player’s ship
- Test case 4 – Miss Detection
  - Pre-conditions: In-game with all ships placed
  - Inputs: Guess aimed at a pre-defined position where a ship has intentionally NOT been placed by the opponent
  - Expected Outputs: A “miss” message is displayed to the player and marked as a miss on both player’s grids
  - Description: Confirms that the system correctly identifies an unsuccessful miss
- Test case 5 – Sink Detection
  - Pre-conditions: In-game with all ships placed and a ship with one remaining un-hit position
  - Inputs: Guess aimed at the last remaining position of the opponents ship
  - Expected Outputs: A “sink” message is displayed to the player and marked as a sink and displaying the sunken ship on both player’s grids
  - Description: Confirms that the system correctly identifies a successful sink when the entire ship has been hit
- Test case 6 – ‘Drop bomb’
  - Pre-conditions: In-game and the player has not yet used the ‘Drop bomb’
  - Inputs: Use the ‘Drop bomb’ at a pre-defined position where a ship has intentionally been placed by the opponent
  - Expected Outputs: A “destroy” message is displayed to the player and marked as a sink and displaying the sunken ship on both player’s grids, and the ‘Drop bomb’ feature is disabled for the player for the rest of the game
  - Description: Ensure that the ‘Drop bomb’ instantly sinks a ship when it hits it and ensures it can only be used once per game
- Test case 7 – ‘Counter Manoeuvre’
  - Pre-conditions: In-game, opponent has a guess aimed at a pre-defined position where a ship has intentionally been placed by the player, and the player has not yet used the ‘Counter manoeuvre’
  - Inputs: Use the ‘Counter manoeuvre’ and move the ship in any direction
  - Expected Outputs: A “near miss” message is displayed to the opponent and marked as a near miss on both player’s grids, and the player’s ship is relocated to the new position successfully

- Description: Ensure that the 'Counter manoeuvre' can be used to avoid an opponent's guess and ensures it can only be used once per game

## 5. Unit Testing

Unit Testing is the process of testing each individual component of a system in isolation to ensure it behaves correctly and as intended. These components are generally code structures such as functions, methods or classes. Unit Testing supports White Box testing because the developer knows the internal logic of the method allowing them to design tests for each specific execution path.

The main principles of Unit Testing are defined to make the testing as simple and streamlined as possible. A Unit Test should be independent and isolated, focussing on one unit at a time, to ensure that each unit works properly. If testing a unit that relies on another unit, the tested unit may return a bug which originated from the relied upon unit, despite the tested unit working as intended, making it difficult to pinpoint the exact cause and location of the bug. Isolation of a unit can be achieved by setting mock data to achieve a known state before the function or class is called, so that the exact setup of the input is controlled and replaces any dependencies, allowing the unit to be tested independently. For example, if testing the 'Drop bomb' feature, the game would be forced to a situation where all ships are placed in the same position every time so that the tester knows exactly what grid positions should return a "miss" and which should return a "destroy" so that it can be tested easily.

Another principle of Unit Testing is that it should include all kinds of values. The tester should implement input values that include normal, expected values, as well as boundary values, out of range values and invalid values. For example, when testing the process of guessing an opponent's ships positions, the tester could give normal values such as 5,5, boundary values such as 10,10, out of range values such as 11,11 and even invalid values such as a string (if possible) to test all possibilities are accepted or rejected as expected.

Another principle is that the tests should be repeatable. This means that if an input is given, it should return the same output every single time. Failure to do this means that the code is unreliable and cannot be trusted, meaning that the issue must be investigated [13], [14], [15], [16].

In the Battleships application, Unit Testing could be applied to many of the functions within the system, ensuring that each gameplay mechanic functions correctly in isolation, before being fully integrated amongst other components. This is important as, due to the number of classes, failure to identify the faults early can lead to bigger, more complex problems later on.

For example, one of the simplest tests on the BoardGrid class would be to check the unit that will validate whether a guess is a “hit”, a “miss”, or a “sink” This could be a function called ValidateGuess(x, y) and the tester could input a series of values with pre-defined ship locations, giving expected outputs of either “hit”, “miss”, “sink” depending on if a ship has been intentionally placed there or not. The guess should return the expected output in terms of the game state too, displaying each guess on both players grids.

A Unit test that could be performed on the SpecialAction class is to test the functionality of the ‘Drop Bomb’ which is very similar to the previous test, but this one specifically will have to check that the entire ship is destroyed and the “destroy” message is outputted to the user, and also ensure that the ‘Drop bomb’ option is no longer available to the user after its single usage.

Another Unit test that could be performed on the SpecialAction class would be to test the functionality of the ‘Counter Manoeuvre’ by pre-setting a ship to a certain location and upon forcing an opponent’s guess at the ship, ensuring that the functionality to activate the move is available to the user and to ensure that the ship moves one square in the intended direction. This should be further tested to ensure that no ships can be moved in the direction which would result in any part of the ship being out of bounds or overlapping with an existing ship.

Additionally, a test that could be performed on the TurnManager class to ensure correct turn management is to end the opponent’s turn and verify that the opponent’s control is disabled and the player’s control is enabled, ensuring that no inputs can be activated outside of each player’s turn.

Therefore, this clearly shows how unit testing would allow the pre-defined rules and logic of Battleships to be tested in a controlled and repeatable way, making it a useful method of testing.

## 6. Conclusion

To summarise, the design and testing process for this Battleships web application displays a structured approach to developing a functional gameplay system. The usage of User stories and a map, as well as CRC cards, helped to clearly define the system requirements and translate them into the object-oriented approach that is required for developing a software application. This approach allowed it to be ensured that core gameplay mechanics, as well as additional features such as the ‘Drop bomb’ and the ‘Counter manoeuvre’ were all properly considered, discussed and designed during this process.

Furthermore, the consideration of software testing, particularly black and white box testing, as well as unit testing, highlights the importance of writing “testable” code that

can be validated against the requirements. Unit testing has been shown to be an especially useful process that ensures individual components work correctly in isolation, reducing the likelihood of errors and the need for debugging once the system has been integrated as a whole.

In conclusion, the use of the mentioned design tools and testing strategies has contributed to the user-focused creation of a reliable, maintainable web application for the traditional Battleships board game.

## 7. References

- [1] "Writing user stories," GOV UK, 23 05 2016. [Online]. Available: <https://www.gov.uk/service-manual/agile-delivery/writing-user-stories>. [Accessed 24 04 2026].
- [2] "Class-Responsibility-Collaboration Card," Geeks For Geeks, 15 04 2026. [Online]. Available: <https://www.geeksforgeeks.org/system-design/class-responsibility-collaboration-card/#what-are-crc-cards>. [Accessed 24 04 2026].
- [3] S. Susnjara and I. Smalley, "What is software testing?," IBM, [Online]. Available: <https://www.ibm.com/think/topics/software-testing#:~:text=Ian%20Smalley-,Software%20testing%2C%20defined,according%20to%20its%20specific%20requirements..> [Accessed 07 04 2026].
- [4] "Importance of Software Testing," Geeks For Geeks, 02 04 2026. [Online]. Available: <https://www.geeksforgeeks.org/software-testing/why-software-testing-is-important/>. [Accessed 07 04 2026].
- [5] "Top 7 Reasons: Why is Software Testing Important?," The One Technologies, 09 05 2022. [Online]. Available: <https://theonetechnologies.com/blog/post/top-reasons-why-software-testing-important#cost-saving>. [Accessed 07 04 2026].
- [6] "What is Software Testing and Why is It Important?," Edure, 10 03 2025. [Online]. Available: <https://edure.in/what-is-software-testing/https://edure.in/what-is-software-testing/>. [Accessed 07 04 2026].
- [7] "The Importance of Software Testing," IEEE Computer Society, [Online]. Available: <https://www.computer.org/resources/importance-of-software-testing#why-is-software-testing-important>. [Accessed 07 04 2026].

- [8] "Differences between Black Box Testing and White Box Testing," BrowserStack, 14 05 2025. [Online]. Available: <https://www.browserstack.com/guide/black-box-testing-and-white-box-testing>. [Accessed 07 04 2026].
- [9] "Differences between Black Box Testing and White Box Testing," Geeks For Geeks, 23 07 2025. [Online]. Available: <https://www.geeksforgeeks.org/software-testing/differences-between-black-box-testing-vs-white-box-testing/#what-is-black-box-testing>. [Accessed 07 04 2026].
- [10] "Black Box Vs White Box Testing," PractiTest, 04 05 2024. [Online]. Available: <https://www.practitest.com/resource-center/article/black-box-vs-white-box-testing/>. [Accessed 07 04 2026].
- [11] E. Elliott, "TDD: Writing Testable Code," Medium, 21 01 2026. [Online]. Available: <https://medium.com/javascript-scene/tdd-writing-testable-code-30ac7a3bf49c>. [Accessed 07 04 2026].
- [12] "Good Code, Testable Code," Epic Web, 10 06 2024. [Online]. Available: <https://www.epicweb.dev/good-code-testable-code>. [Accessed 07 04 2026].
- [13] P. Powell and I. Smalley, "What is unit testing?," IBM, [Online]. Available: <https://www.ibm.com/think/topics/unit-testing#:~:text=Unit%20testing%20involves%20isolating%20units,by%20allowing%20for%20automated%20testing..> [Accessed 07 04 2026].
- [14] "Unit Testing - Software Testing," Geeks For Geeks, 22 07 2025. [Online]. Available: <https://www.geeksforgeeks.org/software-testing/unit-testing-software-testing/>. [Accessed 07 04 2026].
- [15] "What is Unit Testing?," aws, [Online]. Available: <https://aws.amazon.com/what-is/unit-testing/>. [Accessed 07 04 2026].
- [16] D. Rueda, "The 5 principles of Unit Testing," DEV, 14 09 2023. [Online]. Available: <https://dev.to/one-beyond/the-5-principles-of-unit-testing-1p5f>. [Accessed 07 04 2026].