

# NicholasAldous

*by* Nick Aldous

---

**Submission date:** 30-Apr-2025 10:26PM (UTC+0100)

**Submission ID:** 257343917

**File name:** 45151\_Nick\_Aldous\_NicholasAldous\_648183\_1734538210.docx (1.26M)

**Word count:** 5510

**Character count:** 30391

---

Digital Degree Apprenticeship  
Software Engineering

---

# Software Engineering Principles & Testing

ACCB5030

---

Nicholas Aldous  
2237649

---

## TABLE OF CONTENTS

|                                                 |    |
|-------------------------------------------------|----|
| INTRODUCTION .....                              | 3  |
| USER STORY MAPPING AND USER STORY MAPPING ..... | 4  |
| CRC CARDS AND OBJECT ORIENTED DESIGN .....      | 8  |
| GAME WIREFFRAME DESIGN .....                    | 12 |
| SOFTWARE TESTING PRINCIPLES .....               | 15 |
| BLACK BOX TESTING AND WHITE BOX TESTING .....   | 17 |
| UNIT TESTING .....                              | 20 |
| EVALUATION & REFLECTION .....                   | 22 |
| CONCLUSION .....                                | 23 |
| APPENDICIES .....                               | 24 |
| BIBLIOGRAPHY .....                              | 25 |

## INTRODUCTION

Software engineering principles and testing play a crucial role in ensuring the development of high quality, reliable and maintainable software applications. This project explores the principles of designing and testing a web based version of the classic game Connect 4, which has been developed with additional gameplay features. The game follows the traditional 6 x 7 grid structure, where two players take turns dropping counters into columns, aiming to create a line of four counters vertically, horizontally or diagonally. However, players can also capture their opponent's counters based on additional rules, adding a new layer of complexity to the game. A player can capture an opponent's counter by trapping it between two of their own counters horizontally, vertically or diagonally. If a player captures three of their opponent's counters, those counters are eliminated from the grid. Any resulting gaps are filled by counters dropping down from above within the same column. The counters that drop to fill these gaps cannot capture or destroy additional counters.

The development process of this project involved several agile software engineering techniques, including user story construction, CRC card development and explored various testing methods. These tools helped identify user requirements, source core components and outline class responsibilities, which all contributed to shaping the design and development.

The project follows a collaborative design approach, developed through teamwork and regular discussions. We produced a clear user storyboard and a detailed CRC card board by sharing ideas and refining our design together. These tools formed the foundation for evaluating how well the design supported the development of the game and the testing of its functions.

Software testing is a critical aspect of this project to ensure that the game functions as intended. Both black box and white box testing approaches were explored, with a focus on how they relate to the game's performance. Additionally, this project examines the principles of unit testing, explaining how testing methods can be applied to verify the accuracy of individual sections of the application.

An evaluation and reflection on the design and testing process provides insights into the project's strengths and weaknesses. This helps illustrate how software engineering principles contribute to creating a robust and maintainable application.

By examining the design, testing methods and overall evaluation of the game, this project highlights the importance of structured software development practices. The following sections will explore each aspect in detail, offering a comprehensive analysis of the design and testing processes.

## USER STORY MAPPING AND USER STORY MAPPING

User stories are a key component of agile software development to capture requirements from the perspective of the end user. They provide a concise, goal specific description of the user needs, ensuring that development remains focused on implementing features that meet the user requirements. In the development of the Connect 4 game, user stories were helpful in identifying features, understanding user behavior and guiding implementation priorities.

A user story is usually structured using the following simple format –

“As a [role of user] I want to [user goal] so that [user benefit/expected outcome]”

This structure ensures that the user's role, the user's goal and the benefit or expected outcome behind the action are clearly defined. This makes the requirements easier to understand and prioritise while helping the development team stay focused on delivering functional features [1].

One of the initial user stories developed for the game was the following –

“As a player I want to read the rules so that I know how to play the game.”

This user story outlines a fundamental player action and highlights its role within the gameplay. It reflects the minimum required interaction necessary for participating in the game.

While this user story is important, it is considered weaker as it doesn't directly impact the core gameplay mechanics. It focuses on providing background information rather than interacting with the game. While essential for user experience, it doesn't add significant complexity to the development or testing compared to core features like gameplay mechanics.

Another user story that was created for the game was as follows –

“As a player I want to drop a counter down the selected slot to the lowest free space so that I can try to connect four counters.”

This is a strong user story as it is essential to the core gameplay mechanics. It describes the main action that drives the game's progression. It directly affects how the player interacts with the game and requires logic to place the counter correctly in the grid. This action is crucial for the game's function and development, involving complex interactions that need careful development and thorough testing to ensure accuracy and reliability

By reviewing these two examples, it becomes clear how user stories can vary in terms of complexity and impact within the project. The first story focuses on player onboarding, which is useful but doesn't drive the core gameplay. The second story is significant to the gameplay, making it more critical to the system's functionality and testing. In developing software, it's crucial to prioritise the user stories that have the greatest influence on central functionality and system performance.

To organise and prioritise the user stories, a user story map was developed. User story mapping allows the user needs to be visualised across a path and highlights how different activities relate to each other and grouping them into releases [2]. In this project, the user story map divided the game into the following functional areas –

- User Accesses The Application
- User Starts The Game
- User Moves
- User Wins or Loses
- User Captures a Counter
- User Interface
- End of Game
- User Exits The Game

This technique proved valuable in identifying essential game features and additional functionality introduced for the new rules. For instance, the user story for capturing an opponent's counter added new user requirements –

“As a player I want to capture my opponent's counter when I trap it between two of my own so that it becomes mine.”

By positioning more complex stories later in the map, the team could ensure that basic functionality was implemented first, building a strong foundation before introducing advanced mechanics.

User story mapping provided several benefits throughout the design phase, including –

- Prompted a user first perspective, helping to ensure that features were developed based on actual player needs and priorities.
- Allowed collaborative discussion, as the visual layout allowed the team to identify gaps in functionality.
- Supported prioritisation, helping the team to plan the development in stages and deliver core features first.
- Served as a blueprint for testing, as each story could be directly tied to test cases and acceptance criteria.

Additionally, grouping user stories under high level goals provided clarity about the game's overall flow, from accessing the game to the reaching the end of the game.

Each user story included acceptance criteria, which defined the conditions that must be met for the story to be considered complete [3]. The acceptance criteria played a dual role by guiding development and serving as a foundation for testing.

The following is one of the user stories along with its acceptance criteria –

User Story

“As a player I want to be able to switch turns so that I can play against another player.”

Acceptance Criteria

“After a turn is played, the other player should be allowed to make a move.”

The criteria provides a specific, testable outcome that directly informs how the feature should function. By defining expected behaviour, the development process becomes more focused and lays a strong foundation for creating both black box and white box tests. This alignment between user expectations and testable functionality strengthens the connection between design and testing, helping ensure the process remains organised, consistent and easy to track.

During the development process, stories were regularly reviewed and refined. As the complexity of the capture mechanics became clearer, several stories were rewritten to better reflect their technical implications. For instance, the single story “Capturing The Counters” was broken down into the stories, “Identifying a Captured Counter” and “Destroying The Captured Counter”.

This refinement ensured that each story remained small enough to be implemented and tested independently. It also supported future maintainability as clearly defined stories allow features to be traced back to specific requirements during debugging.

While working within a collaborative team, the user stories helped bridge the gap between technical and non-technical team members. Their straightforward language enabled all contributors to engage in design discussions and propose changes. Furthermore, linking the user stories to CRC cards ensured consistency between what the user expects and how responsibilities are divided between different components.

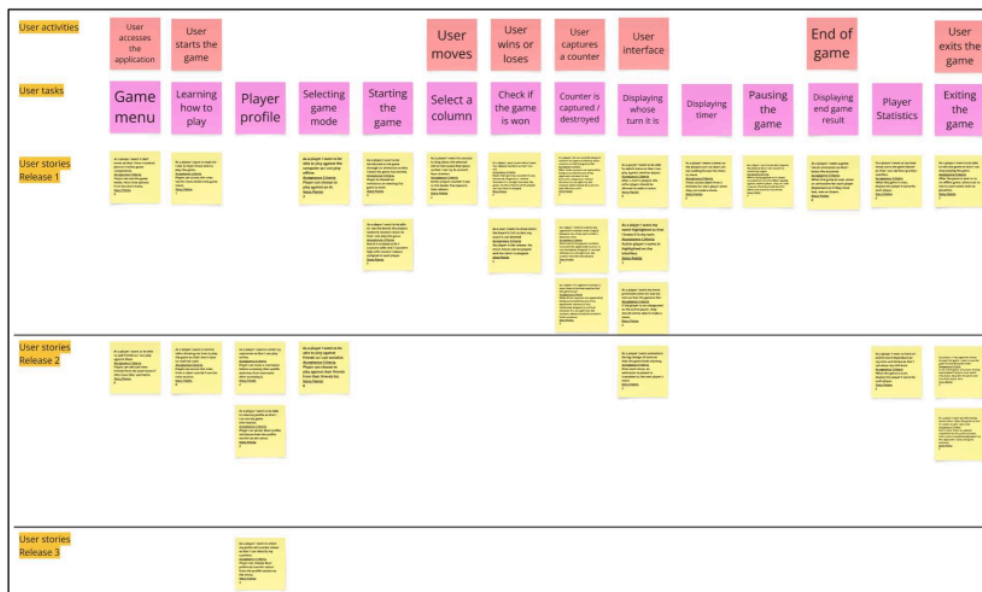


Fig. 01 – User Story Map (detailed view can be accessed via appendices)

## CRC CARDS AND OBJECT ORIENTED DESIGN

Class Responsibility Collaborator (CRC) cards are a tool used in object-oriented design to identify and define the key classes within a software project. Each card represents a class and outlines its responsibilities and what other classes it interacts with. This technique helps clarify the structure of the software and is valuable during the initial phases when designing the application [4].

In this project, CRC cards were used to break down the components of the game into manageable sections. These cards helped structure the system's logic and define how various elements of the game would interact with another. They also served as a connection between user stories and code, ensuring that the action described by the user would be directly reflected in the software design.

The CRC process began after the user stories were finalised and the user story map was developed. Using these stories as a foundation, the team identified which objects were required to support key gameplay actions such as placing counters, changing turns and implementing the new capture mechanics. Each user story was used to guide the identification of responsibilities.

To highlight how user stories influence system design, the following user story illustrates a key gameplay requirement –

"As a player I want the counter to drop down the selected slot to the lowest free space so that I can try to connect four counters."

The user story directly led to the creation of the Grid class, which is responsible for managing the layout and state of the game grid [5]. In response to this functionality, the following class responsibilities were defined –

- 'Display Counters' - Ensures the counters are shown in their correct positions on the grid.
- 'Display Scores' - Keeps track of the player scores and updates them during the game.
- 'Display Timer' - Manages and displays the time passed during gameplay.

By linking this user story to the design of the Grid class, the development process ensures that the system aligns with the intended gameplay, supporting a seamless player experience.

One of the central CRC cards developed during the design phase was the 'Menu' class. This class plays a crucial role in the user interface and overall flow of the game as it interacts with the player to initiate, manage and end the game.

The responsibilities of the 'Menu' class are as follows -

Start/Exit Game

Handles the starting and exiting of the game, allowing players to begin a new game or exit the application.

Overlay Over Grid

Provides an interface overlay over the game grid for displaying options without interfering with the core visuals of the game.

Show Tutorial

Displays a tutorial for players, guiding them on how to play the game and introducing the additional rules

Play vs AI

Manages the functionality that allows players to play against the AI, including setting the difficulty level.

Play vs Friend

Facilitates multiplayer gameplay, allowing two players to play against each other.

Display Rules

Displays the rules of the game, ensuring players understand how the game works, especially the unique capturing and counter destroying mechanics.

These responsibilities directly relate to the functionality needed for the user interface and overall game play. Without the 'Menu' class, players would not be able to interact with the game.

The 'Menu' class does not operate on its own, it relies on several other classes to carry out its responsibilities effectively.

The collaborators of the 'Menu' class are as follows –

Player

The 'Player' class interacts with the 'Menu' class to determine whose turn it is and what counter the current player will place on the grid. The 'Menu' class uses the 'Player' class to switch between players, ensuring smooth transitions between turns during gameplay.

Rules

The 'Rules' class is responsible for coordinating and enforcing the game rules. The 'Menu' class relies on the 'Rules' class to ensure that the tutorial and game rules are accessible, and that the game functions correctly according to the stated rules.

### Grid

The 'Grid' class manages the placement of the counters and tracks the state of the game. The 'Menu' class collaborates with the 'Grid' class to control the flow of the game, such as when the game is paused, resumed or when a new game is started.

By working together, the 'Menu', 'Player', 'Rules' and 'Grid' classes ensure that the game functions smoothly, with each class taking responsibility for specific tasks that interact seamlessly to create an engaging and functional user experience.

The CRC card process developed alongside the refinement of the user stories. Initially, a single 'Game Mechanics' class was proposed to manage all rule based decisions. However, as additional gameplay elements were discussed, such as the capturing mechanic, it became clear that the logic needed to be more modular which led to splitting the responsibilities across separate classes. The breakdown followed object oriented design principles, particularly the Single Responsibility Principle and Separation of Concerns [6], making the system easier to test, extend and debug. This approach allowed each class to be developed independently and integrated later, reducing overlap and supporting a more adaptable and agile development process.

Each CRC card was developed within the group using the collaborative digital workspace Miro which helped to shape the design of the project by clearly outlining the responsibilities of each class. This clarity made it easier to evaluate white box testing methods, as responsibilities could be directly translated into unit tests. For instance, the 'Counter' class responsibility, 'Detect trapped counter between two others' became a specific, testable condition that could be validated during testing.

Additionally, using CRC cards enhanced the maintainability of the code. When bugs emerge or updates required, developers could refer to a class's defined responsibilities to pinpoint the issue. This makes it easier to isolate problems within a specific class, reducing the chance of causing unintended impacts elsewhere in the system.

CRC cards played a critical role in the object oriented design of the Connect 4 project. They helped translate user stories into technical responsibilities, identify logical class groupings and facilitate clear communication across the team. As the project grew in complexity, CRC cards ensured the system remained modular, scalable and easy to understand. By combining CRC cards with the user story map, the team maintained a strong connection between user expectations and system behaviour. This approach supported a structured and organised development process and set a solid foundation for thorough testing and future enhancements.

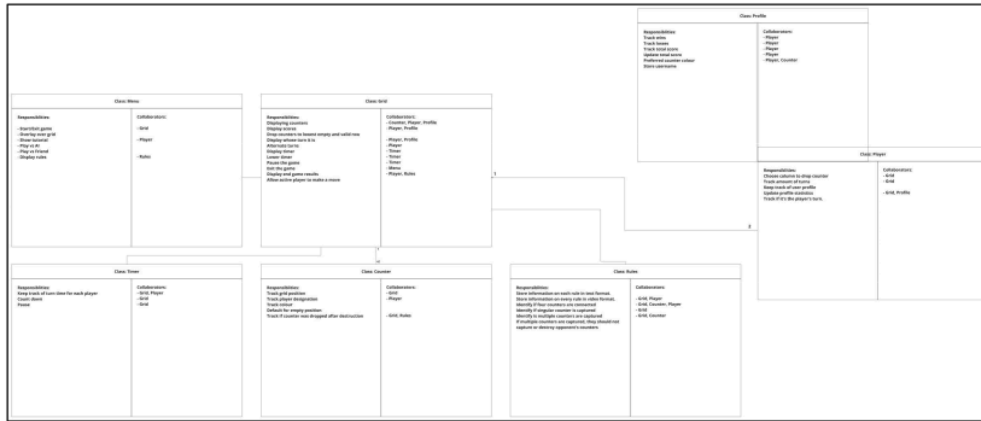


Fig. 02 – CRC Cards (detailed view can be accessed via appendices)

## GAME DESIGNS & DIAGRAMS

To support the development of user stories, CRC cards and clarify how users would interact with the game, a basic wireframe of the Connect 4 interface was designed and created. This visual layout helped outline the structure and flow of the user interface, allowing the team to consider how users would interact with key features. It also supported communication during the design process, helping to align ideas across the team and maintain consistency between the intended user experience and system functionality.

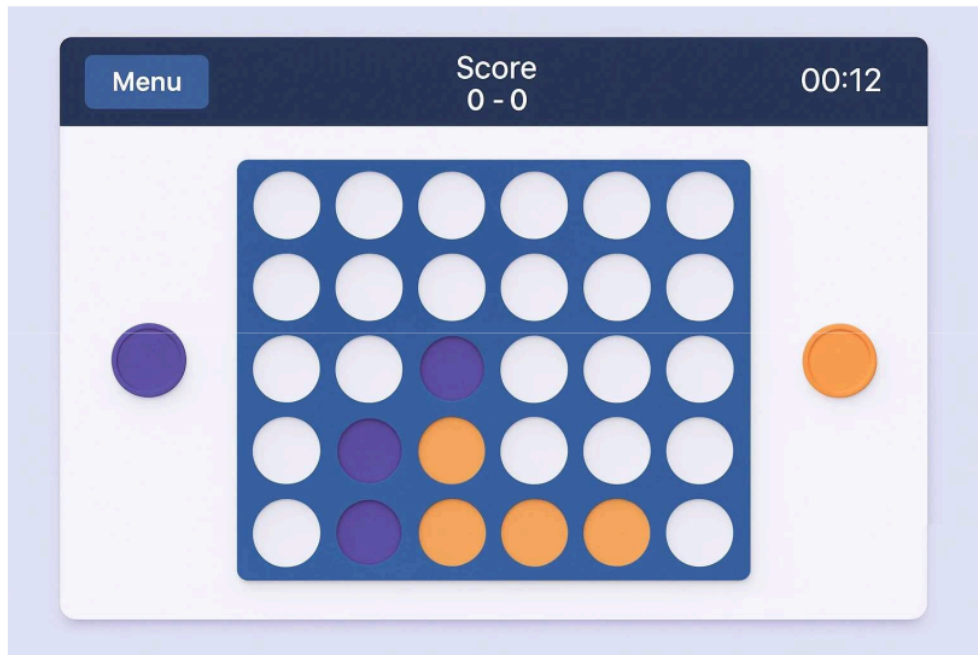


*Fig. 03 – Initial Basic Game Design Illustration*

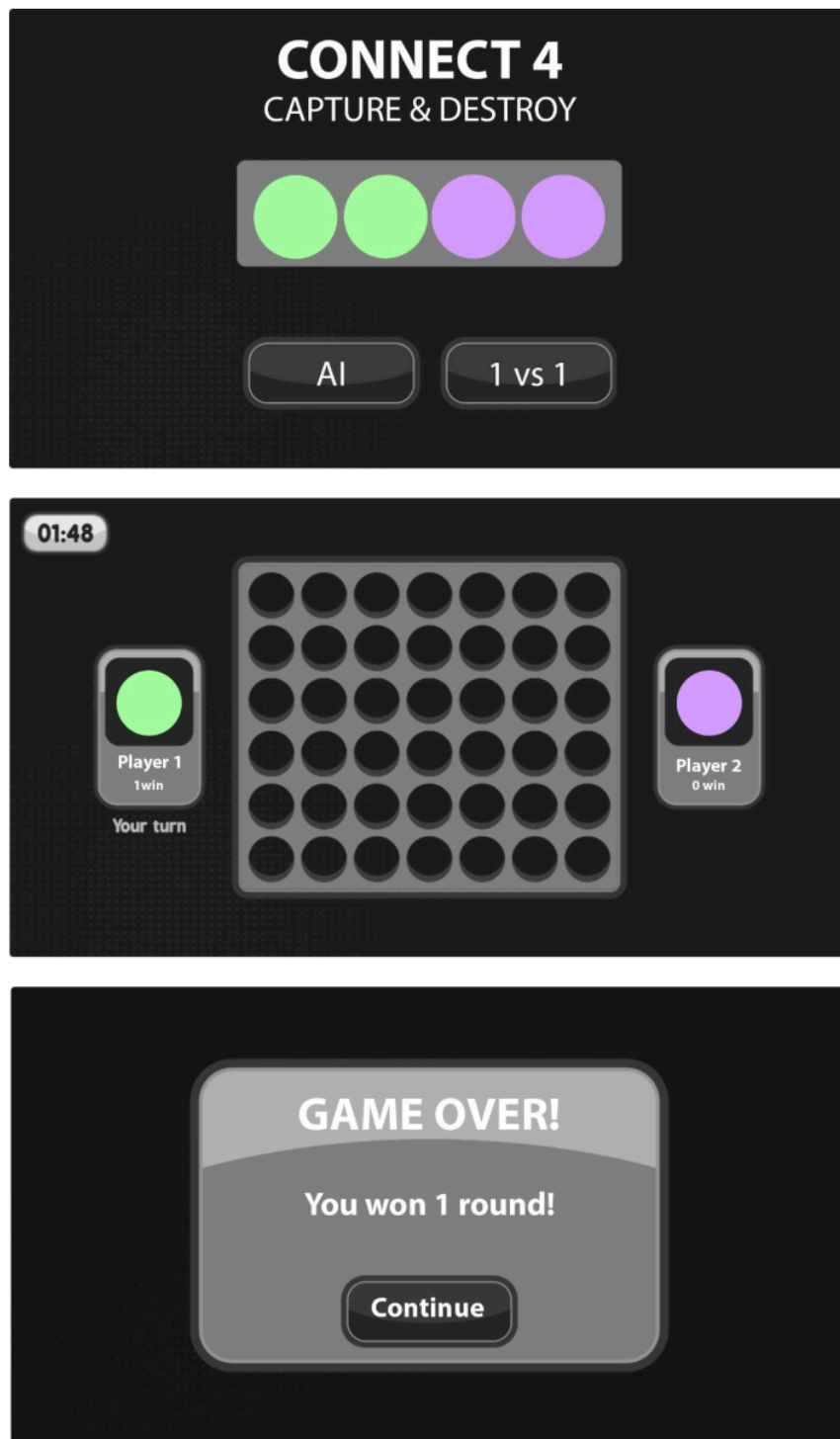


*Fig. 04, 05, 06 – Unique Final Proposed Game Design Illustrations*

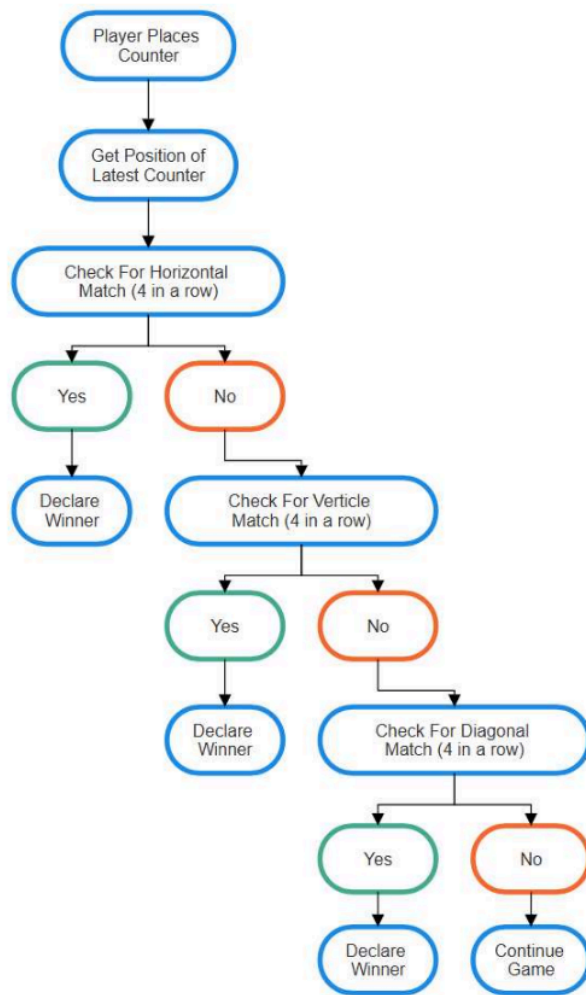


*Fig. 07 – Winning Condition Evaluation Logic Flow Diagram*

## SOFTWARE TESTING PRINCIPLES

Testing is a vital activity in software engineering, ensuring that software is functional, reliable and maintainable. According to Sommerville, "testing is the process of executing a program with the intent of finding errors" [7]. Planning testing from the beginning allows developers to anticipate issues early in the development process, reducing the risk of costly fixes later on and helping to ensure that the final product meets user expectations.

In the development of the Connect 4 game, testing was considered from the start to support the verification of core game mechanics, such as turn switching, win conditions and counter capturing. Incorporating a testing mindset early helped structure the design around detectable errors, preventing defects in logic from growing into larger problems, especially as there are additional rules and features.

Focusing early on testing improves reliability by encouraging developers to think through how the system should perform under various conditions. Reliable software leads to higher user satisfaction and reduces the likelihood of gameplay failures. Additionally, designing with testing in mind supports maintainability by making it easier to modify and extend the system without introducing unintended errors [8].

Both black box and white box testing approaches were explored to assist in verifying the system. Black box strategies focus on external behaviors such as placing a counter, while white box strategies focus on internal processes such as detecting four connected counters. These test versions closely align with the responsibilities defined in the CRC cards, ensuring that each class's behavior can be systematically verified against specific tests.

Testable code refers to code that is easy to verify through logical testing. It's modular, with clearly defined boundaries between components and well organised, which facilitate writing and executing effective tests. As Freeman and Pryce explain, "code that is hard to test is often hard to understand, maintain and evolve" [9]. Prioritising testability during development leads to a cleaner, more maintainable code base.

In the Connect 4 project, testability was promoted through the application of object oriented design principles, particularly the Single Responsibility Principle, which dictates that each class should have one clearly defined purpose [10]. For example, the 'Counter' class was designated to detect and process captured counters, while the 'Profile' class was given the responsibility of tracking user wins and losses. By giving each class a focused role, the design makes it easier to plan and perform unit tests independently.

In object oriented design, loose coupling between components means that the classes or modules in the system do not depend heavily on one another. The less one class relies on another, the more independent it is, making the system more flexible and easier to test. By minimising dependencies between classes, it becomes easier to mock or substitute collaborators during testing. For instance, when planning tests for the 'Grid' class, the 'Player' and 'Rules' classes could be mocked, allowing focus exclusively on 'Grid' specific functionality like counter placement [11].

Additionally, the team's use of user stories and acceptance criteria strengthened the foundation for testability. Each user story outlined a clear user expectation and was tied to measurable outcomes, which could be directly translated into test cases. For instance, the story "As a player I want to switch turns so that I can play against another player" was paired with the acceptance criteria "After a turn is played, the other player should be allowed to make a move." This provided a clear, actionable test condition that could guide both implementation and verification.

By designing the application with testing in mind, the project established better error anticipation, improved system reliability and stronger support for future modifications. Testing was not treated as an afterthought but was integrated into the design thinking process, ensuring that class responsibilities, system behavior and user requirements were consistently aligned.

## BLACK BOX TESTING AND WHITE BOX TESTING

In software engineering, testing methods can be categorised into black box testing and white box testing, each having a distinct purpose in verifying the accuracy and reliability of a system.

Black box testing focuses on the system's inputs and outputs without requiring any knowledge of the internal code structure or logic. The tester considers the software as a 'black box', entering inputs and observing the output to determine if the behavior matches expectations [12]. This approach focuses on functionality from an end user's perspective, making it ideal for validating requirements and checking usability.

In contrast, white box testing examines the internal workings of the system. The tester has access to the source code and uses that knowledge to create tests that ensure specific code paths, branches and conditions are executed [13]. This method focuses more on the implementation details rather than the system's external behavior.

Key differences between black box and white box testing can be seen in areas such as Perspective, Visibility and the Tools used for each approach –

- Perspective: Black box testing is conducted from a user or external viewpoint, while white box testing takes a developer view.
- Visibility: Black box testers have no access to internal structures, whereas white box testers examine source code, logic and architecture.
- Tools: Black box testing tools include automated GUI testers and functional testing frameworks. White box testing tools include code coverage analysts, unit testing frameworks and static analysts.

Together, these two testing types complement each other, black box testing ensures that user requirements are correctly met, while white box testing verifies that internal logic and code structures function properly [14].

Throughout the Connect 4 project, both black box and white box testing approaches were considered to help ensure quality at multiple layers of the game.

### Black Box Test

#### "Drop Counter into a Full Column"

An important black box test scenario focused on attempting to drop a counter into a column that was already full. In this case, the internal code implementation of the 'Grid' was not considered. Instead, the test simply involved –

- Simulating a player dropping a counter into a column that already contains the maximum number of counters.
- Verifying that the system rejected the move and displayed an appropriate error message.

As the tester did not need to know how the grid managed the action internally, this was purely an external functionality validation based on user behavior.

|                    |                                                                                                                                     |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| Test Name          | Drop Counter Into Full Column                                                                                                       |
| Precondition       | The column selected already contains a full column of 6 counters                                                                    |
| Input              | The player attempts to drop a counter into the full column                                                                          |
| Expected Output    | The system rejects the move and displays error                                                                                      |
| Purpose of Test    | Validates that the game correctly handles invalid inputs without examining internal logic.                                          |
| Test Type          | Black Box                                                                                                                           |
| Related User Story | "As a player I want the counter to drop down the selected slot to the lowest free space so that I can try to connect four counters" |

*Fig. 08 – Black Box Test Case*

### White Box Test

#### "Capture Logic Evaluation"

A key white box test scenario targeted the capture mechanic, where a player traps an opponent's counter between two of their own counters. In this test –

- The 'CaptureManager' class was tested directly.
- Test code set up different grid layouts to simulate capture scenarios.
- The code function was run to see if it correctly identified captured counters.
- Various directions (horizontal, vertical and diagonal) were tested to check different capture cases.

Since the tester knew exactly how captures were detected in the source code and constructed tests to exercise specific conditions, this qualified as a white box test.

Both black box and white box testing were considered essential in the Connect 4 project. Black box tests validate that the system responds correctly from the player's perspective, ensuring features such as placing counters, winning or capturing performed as expected. While white box testing ensures that the internal logic is robust, correctly implemented and functions under different conditions.

Using both methods provides a comprehensive quality assurance approach, helping identify external functional errors as well as deeper structural flaws, which ultimately contribute to a more reliable and maintainable game.

|                         |                                                                                                              |
|-------------------------|--------------------------------------------------------------------------------------------------------------|
| Test Name               | Capture Destruction Logic                                                                                    |
| Precondition            | The game grid is produced so that one counter is between two matching counters                               |
| Input                   | Execute the capture destruction function within 'CaptureManager' class                                       |
| Expected Output         | The system identifies the trapped counter and signals it for destruction                                     |
| Purpose of Test         | To validate the internal logic for detecting captured counters across horizontal, vertical or diagonal axes. |
| Test Type               | White Box                                                                                                    |
| CRC Card Responsibility | 'Counter' – Detect trapped counter between two others.                                                       |

*Fig. 09 – White Box Test Case*

## UNIT TESTING

Unit testing is a foundational practice in software engineering that involves testing individual units or components of code in isolation to verify that each performs as intended. According to Meszaros, “A unit is typically the smallest testable part of an application, such as a function or method” [15].

Unit tests are written and executed by developers to detect and correct bugs early to ensure changes do not break existing features and improve code reliability. This type of testing aligns closely with white box testing methods because it requires access to the internal structure of the code.

To be effective, unit tests should follow several guiding principles [3] –

### Isolation

Each test should focus on a single unit of work without relying on the behavior of other parts of the system. For example, testing a function should not depend on other parts such as input handling.

### Repeatability

A good unit test should always produce the same result when run under the same conditions. This helps ensure reliability and helps to immediately detect failures.

### Fast Feedback

The tests should run quickly to promote frequent execution, especially during development. Fast feedback enables developers to find and fix issues instantly after code changes.

### Independent Tests

Unit tests should not depend on each other or on a specific order of execution. This increases maintainability and simplifies debugging when failures occur.

In the Connect 4 game, unit testing was considered for several core functions. The important function ‘dropPlayerCounter()’, controls the placement of a player’s counter into the lowest available row in one of the available columns.

To ensure the function performs correctly, the following unit tests could be produced to –

- Check that the counters fall to the correct position based on the state of the grid.
- Verify that full columns reject additional counters.
- Confirm turn based logic only allows valid moves.

Another key function, ‘checkWinner()’, could be tested to verify it correctly validates horizontal, vertical and diagonal lines of four matching counters. Each of these directional checks represents a separate logical path, with white box testing being useful for ensuring complete coverage.

Unit testing for this project would benefit from using a modern testing framework. One of the most suitable options for a web based application like the Connect 4 game is Jest. Developed by Meta, Jest is a popular testing framework known for its simplicity, speed and all in one capabilities. As described by Osherove, frameworks like Jest help developers “build better code by enabling fast, automated checks of small, testable units” [16].

Jest includes essential tools for unit testing such as a test runner, assertion library, mocking support and built in code coverage reports. This comprehensive setup simplifies the process of writing and managing tests and ensures that developers can receive quick, actionable feedback. Tests can be organised using functions like `describe()` and `test()`, making the test suite easier to navigate and understand.

For this project, Jest could be applied to critical functions like ‘dropCounter’, to ensure counters land in the correct position and ‘checkWinner’, to verify the detection of winning conditions. These functions can be tested in isolation by mocking dependencies such as the ‘Player’ or ‘Grid’ objects.

As Osherove highlights, isolating units during tests helps prevent side effects and ensures that each test evaluates only the logic it is meant to validate [16]. Additionally, Jest’s compatibility with continuous integration tools makes it ideal for maintaining quality throughout ongoing development. Automated testing ensures that new changes do not introduce regressions, promoting code stability and maintainability over time.

By designing the system with unit testing in mind through modular and testable components, the project aligns with good object oriented and testing practices. Planning the tests early in the design stage lays a strong foundation for future development and debugging.

## EVALUATION & REFLECTION

One of the most successful aspects of the design process was the structured progression from user stories to CRC cards and finally to testing. This allowed for a logical flow from understanding user requirements to identifying class responsibilities and validating implementation through testing. The early focus on user story mapping helped ensure that functionality aligned with player expectations and game mechanics, while the CRC cards served as a connection to transform user goals into software components.

Despite the overall success of the project, there were some challenges. One of the difficult parts was implementing the new capture and destruction rules within the game logic and added extra complexity to the grid. Making these features fit into modular classes while following object oriented principles tested our understanding of class structure and responsibility. Working within a group also brought challenges, such as different opinions and coordinating work using the collaborative tool Miro. Ensuring that everyone clearly understood the aims and objectives took time and needed regular discussion and review.

Team collaboration had a clear impact on the final outcome. Group discussions brought in different opinions, which helped improve user stories and identify gaps in the design. For instance, one team member pointed out that the original 'Game Mechanics' class was too general. This led to useful feedback and a redesign into smaller, more manageable classes.

Creating user stories and mapping them was a very collaborative task. Brainstorming as a group helped us decide which features were essential and which were less important. This also improved our acceptance criteria by making the items more clear. Group work also supported ongoing improvements. CRC card details were updated regularly based on feedback and changes in user stories. This process helped keep the design consistent and closely aligned to the user goals.

One of the main strengths of the project was how closely it followed agile methods. The design changed and improved over time based on user needs. By breaking down the project into user stories with clear goals, it was easier to focus on one part at a time. CRC cards also helped by showing what each class should do, which made it easier to plan how the code would be tested.

A key weakness was the limited opportunity to review the design through multiple revisions. Once the initial CRC cards were created, there was little time to revisit or refine them based on evolving insights. This made it harder to spot potential overlaps or gaps in responsibilities, which could affect clarity and maintainability when developing the system.

## CONCLUSION

This project provided a hands on look at software engineering by designing and planning a unique version of the Connect 4 game. It covered everything from creating user stories to building CRC cards and researching test strategies, displaying how helpful a clear and structured approach can be in software design.

One of the main lessons was the importance of starting with user needs and then turning them into testable design elements. User stories helped define what the game should do, while CRC cards gave a clear view of how to build it. Thinking about testing early on, learning about both black box and white box approaches would go on to help to make the application more reliable and easier to maintain.

Teamwork was also important. Working with others led to better ideas and clearer designs. Using shared tools like Miro made it easier to collaborate and stay organised. Feedback from team members helped spot problems early and made the project more consistent.

Using software engineering principles was especially useful when dealing with the game's capturing and destroying counters new rules. Keeping the design modular and focusing on one responsibility at a time made it easier to plan.

Overall, this project showed how useful a structured process can be. In the future, involving users early, designing in small steps and focusing on testability will help make better software that works well and meets user needs.

## APPENDICIES

Below is a link to the collaborative Miro workspace, where the User Story Map and CRC Cards can be viewed in greater detail.

[https://miro.com/app/board/uXjVLnFvZ9M=?share\\_link\\_id=224694028217](https://miro.com/app/board/uXjVLnFvZ9M=?share_link_id=224694028217)

## BIBLIOGRAPHY

- [1] M. Cohn and K. Beck, *User stories applied : for agile software development*. Boston Etc.: Addison-Wesley, , Cop, 2004.
- [2] J. Patton, P. Economy, M. Fowler, A. Cooper, and M. Cagan, *User story mapping : discover the whole story, build the right product*. Beijing ; Sebastopol, Ca: O'reilly, 2014.
- [3] K. Beck, *Test-driven development by example*. Boston Addison-Wesley, 2014.
- [4] W. C. Wake, *Extreme programming explored*. Boston: Addison Wesley, 2002.
- [5] G. Booch, *Object-oriented Analysis and Design With Applications*. Addison-Wesley Professional, 2011.
- [6] R. C. Martin, *Clean Architecture : a craftsman's guide to software structure and design*. Boston Columbus Indianapolis New York San Francisco Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo: Prentice Hall, 2017.
- [7] I. Sommerville, *Software Engineering*. Pearson Higher Ed, 2011.
- [8] R. S. Pressman, *Software engineering : a practitioner's approach*. Boston, Mass.: Mcgraw Hill, 2010.
- [9] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*. Pearson Education, 2009.
- [10] R. C. Martin, *Clean Code a Handbook of Agile Software craftsmanship*. Prentice Hall, 2008.
- [11] E. Gamma, R. Helm, R. E, and J. Vlissides, *Design patterns : elements of reusable object-oriented software*. New Dehli: Pearson Education, 2015.
- [12] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge, United Kingdom New York, Ny, Usa: Cambridge University Press, 2017.
- [13] B. Beizer, *Software testing techniques*. New Delhi: Dreamtech, 2003.
- [14] C. Sandler, T. Badgett, and G. J. Myers, *The art of software testing*, 3rd ed. Hoboken, N.J.: Wiley, 2013.
- [15] G. Meszaros, *XUnit test patterns : refactoring test code*. Boston, Mass.: Addison-Wesley, 2007.
- [16] R. Oshero, *The Art of Unit Testing*. Simon and Schuster, 2013.

FINAL GRADE

82 / 100

GENERAL COMMENTS

Eval Des (20): 16

User Stories (20): 17

CRC (20): 17

SW Testing (20):15

Unit Testing (20):17

FEEDBACK:

=====

User stories and mapping discussed in good detail, but I wanted a little more on the process.

It would have been nice to have expanded upon some of the testing strategies. A few more tests could have been identified for the application. Also there was not a lot of discussion on the term "testable" code.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

---

PAGE 10

---

PAGE 11

---

PAGE 12

---

PAGE 13

---

PAGE 14

---

PAGE 15

---

PAGE 16

---

PAGE 17

---

PAGE 18

---

PAGE 19

---

PAGE 20

---

PAGE 21

---

PAGE 22

---

PAGE 23

---

PAGE 24

---

PAGE 25

---