

Software Engineering Assignment 1

by Sophie Meyrick-Vernon

Submission date: 09-May-2025 01:23PM (UTC+0100)

Submission ID: 258595591

File name: 41934_Sophie_Meyrick-Vernon_Software_Engineering_Assignment_1_648183_215855980.docx
(551.33K)

Word count: 2664

Character count: 15027

Software Engineering Principles and Testing

ACCB5030

Software Engineering Design & Testing

Sophie Meyrick-Vernon

Contents

Introduction 3

User Stories and User Story Map.....3

 Strengths and Weaknesses of User Stories.....4

CRC Card Design Approach 5

 Strengths and Weaknesses of CRC Cards 7

 Acceptance criteria..... **Error! Bookmark not defined.**

Software Testing.....8

 Blackbox vs White Box Testing 8

 Black Box Testing 9

 White Box Testing.....10

Unit testing.....10

Conclusion.....11

Introduction

This document presents a detailed analysis and design plan for a web application based on a variant of Connect 4. In this version, players can capture their opponent's counters by surrounding them with two of their own, either horizontally, vertically, or diagonally. This version's unique feature is removing three opponent counters upon capture, creating gaps that are filled as other counters fall.

The document features user interface designs alongside user stories and their associated benefits, CRC card designs for the application classes, and a comprehensive testing strategy incorporating both black box and white box methods. It aims to demonstrate a strong understanding of software design principles, user-centred design methodologies, and the testing practices essential to successful software engineering.

User Stories and User Story Map

A user story is a tool used in Agile software development to describe a software feature from the user's point of view [1]. They outline the type of user, the action that they would like to perform, and the reason for it. User stories provide a simplified, user-focused description of requirements for the development team. From this, the team can determine how to create the code to fulfil the requirements outlined in the user stories.

The user story map in the provided image shows a hierarchical approach to organising user activities, tasks, and stories with priority levels. Based on this map, we can extract and analyse key user stories.

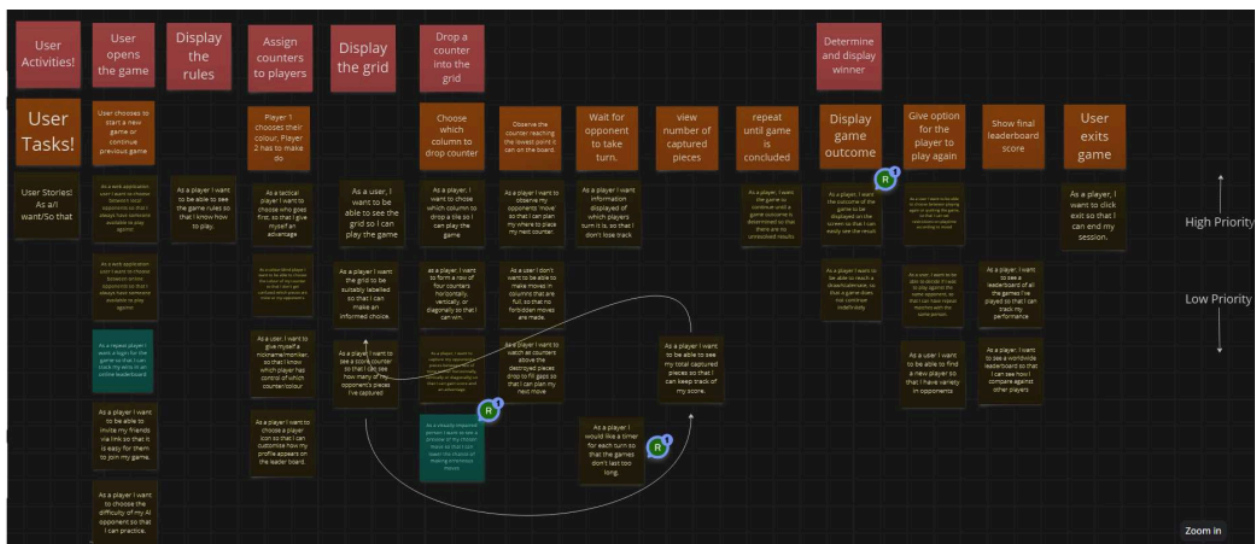


Figure 1: In-class user story

As shown in Figure 1, each user activity is displayed at the top, with the tasks listed underneath. Beneath each task, the related user stories are organized, starting with the higher-priority stories and followed by lower-priority ones. This layout outlines how users would interact with the application, providing development teams with clear insight into user expectations and helping to identify potential issues in the early stages of development. This helps to create a smooth and efficient design and development process.

User stories bring a lot of value to the development process. For instance, the user story, "As a player, I want to be able to see the game rules so that I know how to play." Without this user story, the development team may not have considered implementing the game rules into the application. The team might have only considered the game mechanics and functionality, overlooking small details such as the game rules that a user would find essential. By including this user story, the development team can add this feature before user testing, saving the team time during the development process.

Another user story from the board is, "As a player, I want to observe my opponent's moves so that I can plan where to place my next counter." This user story ensures the game has a strategic feel, giving the players a chance to make informed decisions on their next move, creating a more engaging game experience, and an element of competitiveness.

"As a player, I want to see my number of captured pieces so that I can keep track of my score." As the game introduces a new capturing mechanic, a captured pieces tracker is an essential quality-of-life feature for users to see their scores. It can also add to the competitive depth of the game, as players can change their play style based on their scores, making the game more engaging.

The two user stories "As a visually impaired person, I want to see a preview of my chosen move so that I can lower the chance of making erroneous moves." and "As a colour-blind player I want to be able to choose the colour of my counter so that I don't get confused which pieces are mine or my opponent's." are beneficial for the development team to consider the wider audience, being able to cater to specific disabilities is essential when creating a game, it leads to a more inclusive experience ensuring that the game is accessible to a diverse range of users.

Strengths and Weaknesses of User Stories

User stories have multiple strengths in the application planning process. The user story map provides thorough coverage of the entire user journey, from the start of the game to its conclusion. It organises stories in a hierarchy, separating user activities and tasks so that it is easy to understand the scope of user interaction with the application. The stories can be sorted into prioritisation order with clearly outlined high- and low-priority tasks, enabling the team to focus on the most crucial features.

User stories help maintain a strong focus on the end user throughout the development process, ensuring that the team remains aligned with user needs during implementation. The consistent use of the "As a [role], I want [goal], so that [benefit]" structure solidifies this user-centred approach by clearly illustrating the value each feature provides to the user.

Despite their strengths, the user stories have several potential limitations. Some stories appear to overlap in scope, which may lead to redundant implementation efforts if not carefully managed. While the stories are good at capturing user experience, they often lack the technical detail necessary for implementation, such as performance requirements or backend data structures.

The current focus is mainly on the "happy path," without considering potential error scenarios. Another concern is that there are no clearly defined acceptance criteria in the story map, which may complicate the testing process. Finally, there is a noticeable gap between the high-level stories and the specific implementation steps, which could result in inconsistent interpretations among developers.

CRC Card Design Approach

CRC cards are a technique for capturing each class's responsibilities and how it collaborates with other classes. CRC cards make it easier for teams to discuss ideas and improve their design [2]. They are beneficial during early design stages, helping teams refine their software design through collaboration.

The purpose of CRC cards in software design is to outline the responsibilities of each class and what the class is supposed to do [3]. This helps developers understand the class's role within a system or application. CRC cards also identify how classes collaborate; this is especially important for teams to know how the classes will interact and that necessary interactions are defined [2]. Often used in group settings, CRC cards encourage teams to work together and communicate by including all team members in the design discussion. Finally, there's a clear gap between the high-level user stories and the detailed implementation steps, which could result in varying interpretations among developers.

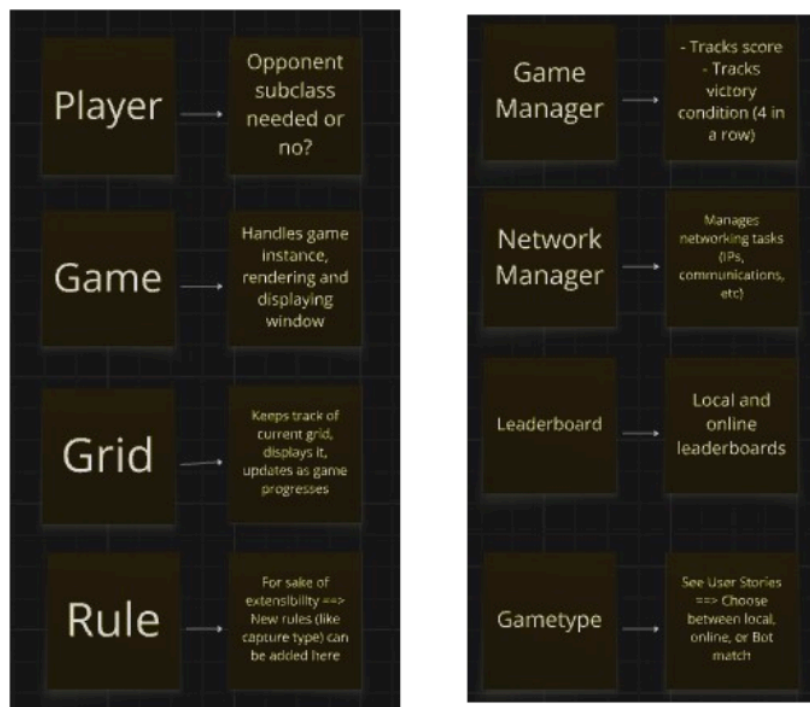


Figure 2: Classes

Here is a breakdown of some of the classes from Figure 2 made into CRC cards for the Connect 4 game design:

Board Class

Responsibilities	Collaborators
Input grid (6w x 7h)	Player
Track if the cell is occupied or not	Counter
Handle the dropping and filling of counters.	Game rules
Notify when a capture occurs.	

Player Class

Responsibilities	Collaborators
Track players 1 and 2	Board
Determine which counter the player is using	Game rules
Manage player turns	
Captured counters	

Game rules class

Responsibilities	Collaborators
Define winning conditions	Game
Define capture rules	Board
Validate moves	
Multiple capture logic	
Player score	

Counter class

Responsibilities	Collaborators
Represent counters	Board
Destruction of captured pieces.	Player

Strengths and Weaknesses of CRC Cards

CRC cards offer several advantages during the early stages of application design. One of the most significant benefits is how clearly, they define each class's responsibilities [3]. By laying this out from the start, teams can avoid overlap and stick to the single-responsibility principle, which helps keep the design clean and manageable. The collaborator section also gives a quick view of how classes connect, making it easier to understand the application and how different parts work together.

Another major plus is how simple CRC cards are; they don't rely on teams to know the technicalities of the program, meaning non-technical team members can join the conversation and contribute. This is useful before any code is written. CRC cards are also flexible and easy to edit and change when new ideas are brought in, which makes them perfect for early brainstorming sessions. Teams can visualise the design through the CRC cards, which helps keep everyone on the same page and visualise the product they are creating.

That said, CRC cards do have their limitations. They usually don't include detailed information about class methods or attributes, which makes it harder to go straight from design to implementation. CRC cards also don't capture how the system behaves over time, like how objects interact or respond to user actions, so they are not ideal for designing dynamic features.

Software Testing

Software testing ensures the quality, functionality, and stability of any application. Without proper testing, a program can have bugs that result in incorrect behaviour, poor user experience, or even failure to execute [4].

Software testing is a foundational element of delivering reliable and user-focused applications. It goes far beyond finding bugs—testing ensures that systems behave exactly as intended and align with real-world user needs. Through rigorous validation and verification, development teams can confirm that the software meets technical specifications and delivers meaningful value to its users.

One of the key advantages of thorough testing is its role in risk mitigation. By identifying potential issues early in the development lifecycle, teams can prevent costly failures in production [4]. This proactive approach helps minimise downtime, reduces the likelihood of security vulnerabilities, and avoids negative user experiences that could harm trust and reputation.

Performance testing is another critical component in evaluating how a system handles varying loads and stress [5]. It allows teams to uncover inefficiencies and pinpoint bottlenecks. This ensures that the software remains fast and reliable, even under heavy usage, providing a seamless experience for end users.

Test cases also serve a dual purpose: not only do they validate system functionality, but they also act as living documentation. They record the intended behaviour of features and workflows, making it easier for current and future team members to understand the system's design and purpose [5].

Blackbox vs White Box Testing

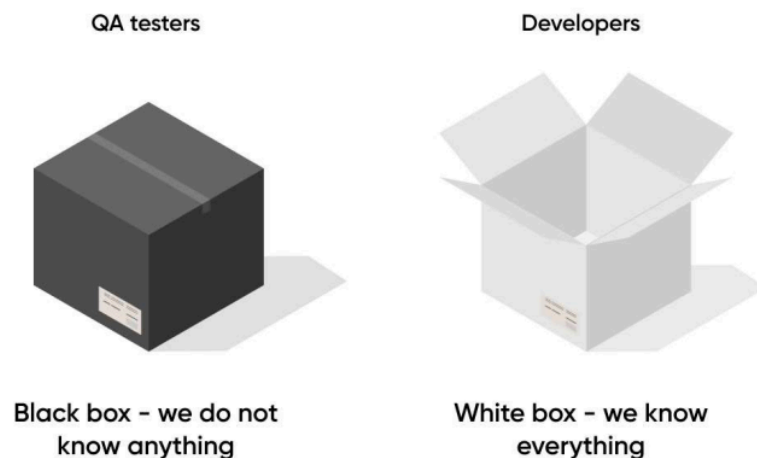


Figure 3: Black box and white box [6]

Black box testing and white box testing are two fundamental approaches to software testing. Black box testing evaluates a system's functionality without knowing its internal workings. Testers treat the software as a 'black box', providing inputs and observing outputs to verify that the system behaves as expected. This method commonly validates user-facing features and is ideal for functional, acceptance, and system-level testing [4].

White box testing examines the application's internal logic, structure, and code. Testers must know the underlying implementation to create tests that cover specific code paths, conditions, and loops. This approach is often used for unit and integration testing, helping to ensure that all parts of the code are working correctly and efficiently [6].

Black box and white box testing complement each other. Black box testing ensures the software meets user expectations, while white box testing ensures the code is robust, secure, and logically sound.

Black Box Testing

Black box testing focuses on testing the software's functionality without concern for the internal structure of the code. The tester concentrates only on the inputs and expected outputs, validating that the system performs as required. In this case, this type of testing helps verify the game's behaviour from a user perspective [7].

Some examples of black box tests:

Test 1: Basic Counter Drop. This test would be repeated for each column.

Test Purpose: Verify that a counter drops to the lowest available row in a selected column.

Pre-condition: An empty grid (6x7).

Input: Player selects column 3.

Expected Output: Counter appears in the bottom cell of column 1.

Test Goal: Ensure correct placement logic for counter drop.

Test 2: Full Column. This test would be repeated for each column.

Test Purpose: Verify that selecting a full column is not allowed.

Pre-condition: All six cells in column 1 are filled.

Input: Player tries to place a counter in column 4.

Expected Output: Invalid move message or ignored input.

Test Goal: Enforce legal move rules.

Test 3: Player Score

Test Purpose: Confirm that a player's score increases correctly after wins.

Pre-condition: Player has 0 points.

Input: Player wins a game.

Expected Output: Player's score increases by 1.

Test Goal: Validate scoring system.

Test 5: Counter Destruction After 3 Captures

Test Purpose: Verify that the opponent's counters are destroyed after three captures, and the remaining counters drop to fill gaps.

Pre-condition: Three captured counters marked. Game engine allows destruction.

Input: Trigger destruction event (after 3 captures).

Expected Output: Destroyed counters removed. Counters above them drop. No unintended captures happen from this drop.

Test Goal: Validate destruction and gravity logic.

White Box Testing

White box testing focuses on the internal structure of the code. The tester examines the logic, structure, and execution paths of the program to ensure that all code paths are covered and working correctly [7]. This means that the tester must be proficient in the technologies used, meaning only individuals with knowledge of the system and programming languages used can perform the tests [7].

Unit testing

Unit testing is a type of software testing that focuses on verifying the smallest testable parts of an application, typically individual functions, methods, or classes, to ensure they work as expected [8]. Each unit is tested in isolation from the rest of the system, often using mock objects or stubs to simulate dependencies. The goal is to catch bugs early in development before they can affect larger parts of the code [8].

Developers usually write unit tests, which are often automated. This allows them to be run frequently, especially during code changes or before deployments. This helps maintain code quality and quickly identify regressions. A good suite of unit tests acts like a safety net, giving developers confidence that their changes won't break existing functionality [8].

Effective unit testing leads to cleaner, more modular code, as writing testable units often encourages better design and separation of concerns. While unit testing doesn't replace other forms of testing, like integration or end-to-end tests, it forms the foundation of a reliable testing strategy [8].

Conclusion

The design of this Connect 4 game with modified rules has been approached with a strong emphasis on functionality, modularity, and user experience. Using CRC cards early on helped us clearly define what each part of the system is responsible for and how different components interact, which made the structure more organised and easier to work with. Additionally, testing both black-box and white-box ensures that the game functions correctly in different scenarios. By implementing unit tests and focusing on testability, we can ensure that the code remains stable and reliable. Overall, the goal has been to create a smooth, bug-free experience that players will enjoy.

References

- [1 M. Rehkopf, "User stories with examples and a template," Atlassian, [Online]. Available:
] <https://www.atlassian.com/agile/project-management/user-stories#:~:text=Summary%3A%20A%20user%20story%20is,simply%20put%2C%20software%20system%20requirements..> [Accessed 5 April 2025].
- [2 "Geeks for Geeks," Class-Responsibility-Collaboration Card, 15 April 2024. [Online]. Available:
] <https://www.geeksforgeeks.org/class-responsibility-collaboration-card/>. [Accessed 6 April 2025].
- [3 "Class Responsibility Collaborator (CRC) Cards: An Agile Introduction," Agile Modeling, [Online].
] Available: <https://agilemodeling.com/artifacts/crcmodel.htm>. [Accessed 6 April 2025].
- [4 R. Mishra, "Black Box Testing And White Box Testing: A Complete Guide," Keploy, 8 Jan 2025.
] [Online]. Available: <https://keploy.io/blog/community/black-box-testing-and-white-box-testing-a-complete-guide>. [Accessed 23 April 2025].
- [5 "What is software testing?," IBM, 9 August 2021. [Online]. Available:
] <https://www.ibm.com/think/topics/software-testing>. [Accessed 26 April 2025].
- [6 Y. Savchuk, "A Guide To: White Box, Black Box, and Gray Box testing.," 24 Jan 2023. [Online].
] Available: <https://www.careerist.com/insights/a-guide-to-white-box-black-box-and-gray-box-testing>. [Accessed 27 April 2025].
- [7 "Differences between Black Box Testing and White Box Testing," Geeks for Geeks, 12 April 2025.
] [Online]. Available: <https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/>. [Accessed 2 May 2025].
- [8 "Unit Testing – Software Testing," Geeks for Geeks, 21 April 2025. [Online]. Available:
] <https://www.geeksforgeeks.org/unit-testing-software-testing/>. [Accessed 7 May 2025].

Software Engineering Assignment 1

GRADEMARK REPORT

FINAL GRADE

61 / 100

GENERAL COMMENTS

Eval Des (20): 13

User Stories (20): 13

CRC (20): 14

SW Testing (20): 13

Unit Testing (20): 8

FEEDBACK:

=====

I would have liked more discussion of the user story and mapping process. It is interesting that you mentioned a gap between high-level stories and implementain, and of course it is the agile process through stories that prevents this gap opening.

I wanted some discussion of the CRC examples you provided.

I was hoping for a little more discussion of testing techniques and testable code.

What is a unit test, how do they work and more on why we use them.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12
