

SEP Assignment

by Sohaan Sureshkumar

Submission date: 17-Apr-2025 04:28PM (UTC+0100)

Submission ID: 255971069

File name: 51186_Sohaen_Sureshkumar_SEP_Assignment_648183_461147025.docx (2.18M)

Word count: 5307

Character count: 28466



SOFTWARE ENGINEERING PRINCIPLES & TESTING

Submitted By:

2221048

Sohaam Sureshkumar

**Module: SOFTWARE ENGINEERING PRINCIPLES
& TESTING**



Table of Contents

INTRODUCTION.....	3
EVALUATION OF DESIGN & TEAM EXPERIENCE	4
1.1 Initial Understanding and Discussions	4
1.2 User Stories and Their Benefits	5
1.2.1 My Personal Contribution	5
1.2.2 How these User Stories helped our design:	7
1.3 User Story Mapping and Evaluation of Design Approach	7
1.3.1 Collaborative Use of The User Story Map	7
1.3.2 Benefits Of The User Stories	8
1.3.3 Strengths Of Our Approach	9
1.3.4 Weakness Of Our Approach	9
1.3.5 Reflection On The Mapping Process	9
CRC-CARDS DESIGN & APPROACH	9
2.1 Introduction to CRC Cards	9
2.2 Our CRC Card Set	10
2.3 Class by Class Breakdown.....	11
2.3.1 GameBoard	11
2.3.2 Player	12
2.3.3 GameLogic	13
2.3.4 UIManager	14
2.3.5 GameController	15
2.4 Evaluating The CRC Approach	16
2.5 Alignment With The User Stories	17
SOFTWARE TESTING EVALUATION.....	17
3.1 Importance Of Software Testing	18
3.2 Black Box VS White Box Testing	18
3.2.1 Black Box Testing.....	18
3.2.2 Whitebox Testing.....	19
3.2.3 Why Both Tests Matter	19
3.3 Testable Code	19
3.3.1 What It Is and Its Importance	19
3.3.2 Its Relevance In Software Engineering	20
DESIGNING TEST CASES.....	20
4.1 BLACK BOX TESTS	20
4.1.1 Vertical Win Test.....	20
4.1.2 Full Column Test.....	21
4.1.3 Turn Switching.....	21

4.1.4 Win Detection After Capture	22
4.2 WHITE BOX TESTS	22
4.2.1 Capture Rule Test	22
4.2.2 Diagonal Win Detection Test	23
4.2.3 Multiple Capture In One Move Test	24
UNIT TESTING	24
5.1 What Is It And Why Its Important	24
5.2 Units Tests Which Could be Applied	25
5.2.1 Capture Win Test	25
5.2.2 Diagonal Win Detection Test	25
5.2.3 Multiple Captures Test	25
5.3 Why Unit Testing?	26
CONCLUSION	26
REFERENCES	27
APPENDICES	30
5.4 APPENDIX A – Team User Story Map	30
5.5 APPENDIX B – CRC CARDS	31

TABLE OF FIGURES

Figure 1 - User Story Map	8
Figure 2 - CRC Card Set	10
Figure 3 GameBoard CRC Card	12
Figure 4 - Player CRC Card	13
Figure 5 - GameLogic CRC Card	14
Figure 6 - UIManager CRC Card	15
Figure 7 - GameController CRC Card	16

INTRODUCTION

Software design requires more than programming because it demands knowledge of system usage patterns and logical system functionality [1]. The assignment requires

us to work on Connect 4 through a modified version which added counter capturing and destruction features. The task appeared straightforward at first yet the actual complexity of implementing rules became apparent when we began the design.

Our task extended beyond game development because we needed to design the system according to software engineering principles while determining user interactions and maintaining system consistency. The game development required user stories, story maps and CRC cards as tools to analyse the game structure and organize our thoughts correctly.

The design received equal importance to testing during this process. We mastered test planning through user perspective analysis and direct examination of the logic structure. Through this process we gained knowledge about the essential role of black-box testing and white-box testing and unit testing in achieving system reliability.

The assignment required us to develop engineering techniques for a fun system which resulted in a fully thought-out system.

EVALUATION OF DESIGN & TEAM EXPERIENCE

1.1 Initial Understanding and Discussions

When this project started our team thought we would simply tweak the Connect 4 game. The game seemed easy to understand at first because of its familiar design so we believed we could solve it without much effort. Reading the new rules and

assignment brief properly showed us that the game was more complex than we had imagined. The rules for counter capture and destruction together with counter drops into empty spaces proved that the game required more complexity than first appearances suggested.

During our initial team gatherings, we dedicated our time to ensure everyone grasped the game rules correctly. We discussed different hypothetical situations while attempting to predict the board's responses. We discussed two specific scenarios during our team meetings including diagonal counter captures and complete column destruction when a counter stands in the middle. Our team discussions revealed the numerous small details we needed to consider before starting to design the system.

The group first wanted to make sure everyone understood the game logic before moving into technical tools or diagram creation. The team discussed how turns should progress and what should occur during exceptional situations. The discussion provided us with better clarity before starting the design phase. The team members had unorganized discussions at first which resulted in a unified understanding that made it easier to develop user stories and CRC cards.

1.2 User Stories and Their Benefits

The first thing we learned about was user stories. Writing basic sentences like 'As a player...' or 'As a developer...' became essential tools for streamlining our design approach [2].

1.2.1 My Personal Contribution

I worked on six user stories that captured the essence of the project. The first one was:

“As a player, I want clear instructions so that I know how to play.”

A basic yet essential advantage is that players will become frustrated and quit the game if they do not understand the rules or how the game works, especially with added features like capturing and gravity [3]. This story told us to make something that people can enjoy and engage with confidence.

In terms of the technical side, I added two stories from the developer's point of view:

“As a developer, I want to implement a grid system so that counters land in correct positions,”

and

“As a developer, I want to refine game logic so that capture mechanics function correctly.”

The benefit of these stories is that they kept our system reliable. If the grid did not function consistently then the entire game would break [2]. And the capture logic, being one of the trickiest parts, needed to be precise to avoid unfair moves or broken gameplay.

I also designed from the design perspective:

“As a game designer, I want to implement basic turn-based mechanics so that each player gets a fair chance.”

This story ensures that the game stays balanced. It would be confusing or even unplayable if there are no clear turn-based rules.

“As a game designer, I want to ensure counters follow gravity rules so that they drop correctly.”

This one helped us focus on making the game feel natural. Pieces should drop down not hover or float, so this story kept the gameplay intuitive.

As for the tester story I contributed, I had:

“As a tester, I want to verify that win conditions are detected correctly so that the game can determine a winner.”

It guaranteed closure for the game, and this is the benefit of this story. The players need to have clear feedback when they've won, and this also helps the game to transition to an end state without any glitches.

1.2.2 How these User Stories helped our design:

The user stories gave us a better structure for planning our project. We could avoid missing important features or creating gaps in our logic by thinking from different perspectives [2][4]. They were useful when it came time to do the design diagrams and testing plans later on because we already had a clear breakdown of what each part of the system should do. The user stories were one of the most useful tools in keeping our project focused and organised, even though they were short.

1.3 User Story Mapping and Evaluation of Design Approach

1.3.1 Collaborative Use of The User Story Map

Our team started the design process by creating a user story map to organize the project tasks based on user needs. We began by establishing four key roles: the player, developer, game designer, and tester. We produced stories using the following structure: 'As a [role] I want [goal] so that [benefit].' The stories served to direct our attention toward actual user requirements and they improved internal group communication clarity and consistency [5].

1.3.3 Strengths Of Our Approach

Our design approach reached a state of excellent organization among its strongest features. Through the map the team distributed their workload effectively and members took responsibility for the stories they understood best. Each task enabled us to recall the purpose which improved our ability to prioritize work while maintaining focus.

1.3.4 Weakness Of Our Approach

Release 3 presented us with multiple challenges throughout our work. Multiple user stories contained vague information and exceeded the available timeframe. The paper-based ideas for move history features and customization failed to deliver sufficient detail needed for proper development. Some of the stories were added to the project after its start which led to modifications in earlier work that required additional effort for completion.

1.3.5 Reflection On The Mapping Process

The user story map proved essential in maintaining our design's direction throughout its entire duration [5]. Through the story map we achieved strong teamwork while creating a structured plan that maintained user-centred priorities.

CRC-CARDS DESIGN & APPROACH

2.1 Introduction to CRC Cards

CRC stands for Class, Responsibilities, and Collaborators, it is a design tool that is used to help teams decompose and organize object-oriented systems before the actual coding takes place [8]. Each card is a class in the system and the focus is on what the class does (responsibilities) and which other classes the class interacts with (collaborators). CRC cards are very useful in a team environment like ours as they help us to visualize the system at an early stage to understand how our Connect 4 game will be structured [8].

We applied the CRC cards approach in a group discussion to design the architecture of our custom Connect 4 game. These discussions assisted us in moving beyond the basic level and thinking about such questions like “What should this class actually be in charge of?” and “How do these classes interact with each other?”[8][9].

2.2 Our CRC Card Set

Class: GameBoard Responsibilities Store the grid state (rows & columns) in a 2D array. Allow players to drop counters into a column. Check if a column is full before allowing a move. Validate and update the grid after each move. Communicate with GameLogic to determine if a player has won. Collaborators Player: Receives input from players about where to drop counters. GameLogic: Checks if the current move results in a win or draw. UIManager: Updates the display of the board based on the grid state.	Class: Player Responsibilities Take turns in making moves. Drop counters into selected columns. Store player identity (e.g., Red or Yellow player). Request game board updates after each move. Collaborators GameBoard: To drop counters in the grid. GameLogic: To verify where turn is and enforce turn order. UIManager: To update the interface and highlight active players.	Class: GameLogic Responsibilities Determine whose turn it is. Validate moves before allowing them. Check if a move results in a win (horizontal, vertical, diagonal). Check if the board is full, leading to a draw. Apply special rules (e.g., restoring mechanics if implemented). Collaborators GameBoard: To analyze the grid state and detect winning patterns. Player: To confirm valid moves and switch turn accordingly. UIManager: To display win/lose messages and game status updates.
Class: UIManager Responsibilities Display the game board and counters visually. Update the screen after every move. Indicate whose turn it is. Show messages for win/loss or invalid moves. Provide buttons for restarting the game. Collaborators GameBoard: To retrieve the latest board state for rendering. Player: To highlight the active player's turn. GameLogic: To display game outcomes and rule enforcement messages.	Class: GameController Responsibilities Manage the overall game flow. Initialize game components (GameBoard, Player, GameLogic, UIManager). Handle player input and ensure correct sequence of events. Restart the game when needed. Collaborators GameBoard: To reset or update the grid when the game restarts. GameLogic: To control turn-based logic and check for game overstates. UIManager: To refresh the display and notify users of events. Player: To track active players and keep turns.	

Figure 2 - CRC Card Set

Our final CRC design consisted of five main classes:

1. GameBoard
2. Player
3. GameLogic
4. UIManager
5. GameController

All of the cards had the following structure i.e. name of the class at the top, then responsibilities of the class, and classes with which the class collaborates. This helped us concentrate on single-purpose class responsibilities which is important for the cleanliness, testability and extensibility of the code.

2.3 Class by Class Breakdown

2.3.1 GameBoard

Class: GameBoard	
Responsibilities: Store the grid state (rows & columns) in a 2D array. Allow players to drop counters into a column. Check if a column is full before allowing a move. Validate and update the grid after each move. Communicate with GameLogic to determine if a player has won.	Collaborators: Player: Receives input from players about where to drop counters. GameLogic: Checks if the current move results in a win or draw. UIManager: Updates the display of the board based on the grid state.

Figure 3 - GameBoard CRC Card

The GameBoard class is the heart of our grid system. Its main responsibilities include storing the current board state and updating it whenever a move is made [9]. It's responsible for drawing the board for the player (in cooperation with UI Manager) and responding to player actions by allowing counter placements.

It works with GameLogic, Player and UIManager classes because it is the core of gameplay. It is the interface that the player sees and the truth of what is going on the board.

What worked well in this card was that we kept it focused. It wasn't overloaded with logic for win detection or player control which was delegated to other classes, which keeps things modular [8]. That said, we could have added more detail about how board updates work when captured counters are removed especially since our version of Connect 4 involves special "drop down" mechanics after counter destruction.

2.3.2 Player

Class: Player	
Responsibilities: Take turns in making moves. Drop counters into selected columns. Store player identity (e.g., Red or Yellow player). Request game board updates after each move.	Collaborators: GameBoard: To drop counters in the grid. GameLogic: To verify whose turn it is and enforce turn order. UIManager: To update the interface and highlight active players.

Figure 4 - Player CRC Card

The “Player” class was designed to manage everything related to a player’s actions and identity. Its responsibilities include taking turns, deciding where to place a counter, and interacting with the game board [10]. It also stores basic information like the Player Identity and possibly their score or capture count in future versions.

It collaborates with the GameBoard, GameController, and UIManager. This shows it isn’t working in isolation, its actions feed into game control and affect how the board updates visually.

This CRC card was especially helpful when thinking through our user story: “As a player, I want clear instructions so I know how to play”.

2.3.3 GameLogic

Class: GameLogic	
Responsibilities: Determine whose turn it is. Validate moves before allowing them. Check if a move results in a win (horizontal, vertical, diagonal). Check if the board is full, leading to a draw. Apply special rules (e.g., capturing mechanics if implemented).	Collaborators: GameBoard: To analyze the grid state and detect winning patterns. Player: To confirm valid moves and switch turns accordingly. UIManager: To display win/loss messages and game status updates.

Figure 5 - GameLogic CRC Card

GameLogic deals with the basic rules of the game like the win conditions and the capture rules [10]. This was probably the most complex card because of how our version of Connect 4 introduced non-traditional mechanics, like counter destruction and gravity-based refill of the board.

Some of its responsibilities include determining whether a move is valid, implementing the capture rules, determining if a player has won or drawn the game, and managing the internal logic that powers the gameplay. It collaborates with the GameBoard, UIManager, and Player.

This card was one of the best examples of how our design process was effective. This is where all the thinking about rules was consolidated [10]. A separation of concerns within our group revealed that visual side (UIManager) and control flow (GameController) did not need to be concerned with the specific rules of capturing counters. It also made our planned unit tests more straightforward to create later since GameLogic was properly isolated.

2.3.4 UIManager

Class: UIManager	
Responsibilities: Display the game board and counters visually. Update the screen after every move. Indicate whose turn it is. Show messages for win/loss or invalid moves. Provide buttons for restarting the game.	Collaborators: GameBoard: To retrieve the latest board state for rendering. Player: To highlight the active player's turn. GameLogic: To display game outcomes and rule enforcement messages.

Figure 6 - UIManager CRC Card

The UIManager was in charge of displaying the current state of the game to the players. It doesn't control or implement the game logic, but it displays whose turn it is, displays the board after each turn, and gives feedback such as a win or an invalid move.

The Game Logic, the Player and the GameController are some of the collaborators of the UIManager. Another good practice that was identified in the discussion was that the UIManager does not "decide" anything, it just shows what is happening [10]. This division made the UI layer testable.

2.3.5 GameController

Class: GameController	
Responsibilities: Manage the overall game flow. Initialize game components (GameBoard, Player, GameLogic, UIManager). Handle player input and ensure correct sequence of events. Restart the game when needed.	Collaborators: GameBoard: To reset or update the grid when the game restarts. GameLogic: To control turn-based logic and check for game outcomes. UIManager: To refresh the display and notify users of events. Player: To track active players and swap turns.

Figure 7 - GameController CRC Card

This class controls the whole game. It is responsible for initiating the game, turning, accepting the player's input, and determining the end of the game. It does not check for wins (that's GameLogic), and it does not directly change the board (that's GameBoard) ,but it knows which classes to call at which time.

Almost everyone is its collaborators: Player, UIManager, GameBoard, and GameLogic. This card really helped us determine the sequence of the game: who is to go first, when logic checks should be made, and how the turns are to be taken.

2.4 Evaluating The CRC Approach

Using CRC cards as a team really paid off. The physicality of writing and discussing the cards made us reflect on class responsibilities. It was a rather comfortable approach to begin designing as none of us had to worry about syntax or implementation at this point.

One of the big strengths was that it made the responsibilities more obvious. There was no confusion about what each class was supposed to handle. It also prevented us from assigning too many responsibilities to one class.

I noticed some weaknesses too. A few cards (like Player) were a bit too simple and we could have included more methods or specified interactions.

However, in general, CRC cards worked quite well together with our user stories.

2.5 Alignment With The User Stories

A last strength of our CRC cards is that they are well adapted to the user stories we created. For example:

- The story “As a tester, I want to ensure that win conditions are identified properly....” was directly related to GameLogic.
- The story “As a player, I want to have clear directions...” was closely related to both Player and UIManager.
- The “As a game designer, I want to implement turn based mechanics...” was handled by Game Controller in terms of turn logic.

This mapping gave us confidence that our design supported the real needs of the system and players and would be easy to test later.

SOFTWARE TESTING EVALUATION

3.1 Importance Of Software Testing

Testing for software is an essential component of the development cycle rather than an added feature at the end of the process. It makes sure a program works as intended and helps find bugs before users encounter them [11]. Testing allowed us to test logic, user interaction and special rules (like counter capture) in a structured manner in our Connect 4 project.

The value of testing extends from basic functionality assessment to determining release confidence. The game detection of wins and invalid input handling became essential to test or else we would end up with a final product that appears sound but fails when tested in extreme situations [11][12]. Bug problems in games represent more than system flaws because they impact user experience and trust.

Software testing plays a crucial role because it enables developers to build upon their projects in the future [12]. A project which receives adequate testing becomes more manageable to update or expand without disrupting current features. Future implementation of an AI opponent in our Connect 4 game will be possible because automated tests will prevent us from damaging fundamental gameplay elements during development.

3.2 Black Box VS White Box Testing

The software development process incorporates two fundamental testing approaches: black box testing and white box testing. Both methods offer benefits yet analyse software structures in different directions.

3.2.1 Black Box Testing

Black box testing involves analysing software functionality by ignoring programming details [13]. Testers do not require any knowledge of the internal code structure. Testers interact with the system externally by submitting input values while examining resulting outputs against anticipated results [13].

Scenario 1: Counter Drop Test

A player places a counter into column 4. The test verifies if the counter ends up in the lowest available position of that column. The tester does not review the logic implementation but only checks if the outcome matches the expected result [13].

Scenario 2: Diagonal Win Detection

The player establishes a diagonal line consisting of four counters. The system needs to announce victory after this condition. The test examines only the result that appears on screen without verifying the win detection code.

3.2.2 Whitebox Testing

White box testing stands as the opposite method because it examines code internals [14]. The process tests the internal functions and control flow together with loops and logic conditions to verify the correct operation of all code paths [14]. A test author must understand the program architecture to execute this type of examination which developers typically create.

Scenario 1: Capture Logic Function

The developer checks the function which determines counter capture status. The developer checks each condition individually to validate that counters get destroyed only when two matching counters surround them from opposite sides.

Scenario 2: Method Execution Order

The developer checks the win detection function call order after the board update and counter destruction. The developer checks the code call sequence to validate the proper sequence of execution [14].

3.2.3 Why Both Tests Matter

Using both black box and white box testing together gives a complete picture of whether the program works and whether the code inside it is reliable. Relying only on black box testing might mean we miss issues hidden deep in the logic (like a rare bug in a win detection function) [11]. On the other hand, doing only white box testing might lead us to assume something works just because the logic looks correct. but the user experience might be broken.

3.3 Testable Code

3.3.1 What It Is and Its Importance

Testable code describes programming that allows for simple testing through both manual and automated methods [15]. The development of the game would become significantly easier if we implemented testable code because it enables bug detection and program checks [16].

Testable code demonstrates specific characteristics through its clarity and division into small parts and its single-purpose design for each class or function [16]. Our CRC design implementation aimed to adhere to these principles. GameLogic operates independently from GameBoard and UIManager so gameplay rules can be tested independently from visual elements and user interactions.

The ability to test code safely enables developers to make future modifications [16].

3.3.2 Its Relevance In Software Engineering

Today in software engineering, testability is a best practice, not just something optional. With the rise of agile development and continuous integration, the ability to test and validate code regularly is vital. Projects move fast and without automated tests or easily testable components, changes can lead to bugs that are hard to track [15].

In professional settings, companies usually use automated test suites, and developers are required to write unit tests for new features. A poorly tested project may work once, but it will not be able to handle the growth or even the next update [15]. That's why concepts like TDD (Test-Driven Development) [17], unit testing, and integration testing are now incorporated into the development cycle from the very start.

DESIGNING TEST CASES

4.1 BLACK BOX TESTS

4.1.1 Vertical Win Test

Preconditions:

There are already three counters from the same player in Column 3. It's their turn.

Input:

They Drop a third counter in the same column.

Expected output:

The system announces that the player has won, and the game ends.

What this test checks and how it works:

This test verifies that the game detects a vertical win and behaves accordingly. We don't look at how the system checks for wins. We just see if the right message shows up after the move [19].

[18]

4.1.2 Full Column Test

Preconditions:

Column 5 is full and now It's Player 2's turn.

Input:

Player 2 attempts to drop another counter in column 5.

Expected output:

The system says, "Column is full. Try another one," and nothing changes on the board.

What this test checks and How it Works:

This test checks whether the game properly prevents illegal moves and whether it gives proper feedback. We play it like a user and see if the message and behaviour match what should happen.

[18]

4.1.3 Turn Switching

Preconditions:

It's currently Player 1's turn.

Input:

Player 1 makes a valid move.

Expected output:

The system updates the board, then switches to Player 2 and shows that clearly with a message or a visual.

What this test checks and How it Works:

After each move, turns should be properly alternating. This is about user experience. We observe the game's actions and verify that the following turn belongs to the correct player.

4.1.4 Win Detection After Capture

Preconditions (Setup):

The current player controls three adjacent counters. Player 1 gains the opportunity to capture one of the three counters before Player 2 can play their fourth piece.

Input:

Player 2 then drops the fourth counter as planned.

Expected Output:

The game maintains that Player 2 did not win because their sequence got interrupted by the capture.

What this checks & how it works:

This checks that the game always checks the board state after each move, including after captures, and doesn't falsely detect a win. We analyse the modified board to confirm the game avoids triggering a win announcement when the pattern disappears.

4.2 WHITE BOX TESTS

4.2.1 Capture Rule Test

Preconditions:

A row has the pattern: Player 1 Player 2, Player 1. As per the design, this should allow Player 1 to capture Player 2's counter. The logic for checking captures would be part of the checkCapture() function.

Input:

Player 1 plays a move that causes this capture situation.

Expected output:

The function removes the counter of Player 2 from the board and drops any counters above it if there are any.

[18]

What this test checks and How it Works:

Checks if the internal logic for identifying and removing captured counters work as intended.

4.2.2 Diagonal Win Detection Test

Preconditions:

A possible win in the diagonal direction is set up on the board. A checkWin() function would be used to check for win conditions in all directions.

Input:

A player puts a counter that would form a diagonal line.

Expected output:

The function returns true, indicating a win.

What this test checks and How it Works:

This Test Checks Whether the code within the win checking function correctly detects the diagonal pattern. The test would feed in a version of the board with a diagonal win and verify that the function returns the correct answer.

[18]

4.2.3 Multiple Capture In One Move Test

Preconditions (Setup):

Player 1 drops a counter in such a way that it results in three captures i.e. horizontally, vertically, and diagonally.

Input:

Player 1 drops a counter that completes all three patterns.

Expected Output:

All three captured counters are removed, and the board is correct.

What this checks & how it works:

This test checks that the if logic can process multiple captures at once. The method that checks captures internally needs to check all directions before it applies the proper changes to the board data [19]. We would test by establishing a particular board configuration then checking that all captured pieces get correctly removed during verification.

[18]

UNIT TESTING

5.1 What Is It And Why Its Important

Unit testing is the process of testing small, individual parts of a system such as functions or methods to ensure that they behave as expected [20]. These are called "units," and each unit test is focused on just one specific task, for example, checking for a win or validating a capture.

The main goal of unit testing is to ensure that each component is correct on its own before the whole system is put together [21]. This helps to prevent bugs from emerging throughout the application and makes it easier to determine where a problem occurred if something goes wrong [20] [21]. It also helps developers modify parts of the system

in the future without damaging another component, if they do, the unit tests will most likely catch it.

In the development cycle, unit testing is done early, either during or right after coding, before moving to larger tests like integration testing or user testing [20] [21]. In many professional settings, developers write unit tests as they write their code, and test automation tools are used to run all tests on a regular basis [21]. This makes the system reliable even when it is growing [21].

5.2 Units Tests Which Could be Applied

Below are the white box tests we would perform as unit tests based on our design:

5.2.1 Capture Win Test

This unit test is on the capture logic. For instance, if a row has the pattern: Player 1 – Player 2 – Player 1, the system should be able to recognize that Player 2's counter is pinned and remove it. In the test, we would manually create this scenario on the board data and invoke the method that checks for captures [19]. Then we would check that the right square was cleared and any counters above it dropped down if necessary. This would help ensure that the logic inside the capture method is correct.

5.2.2 Diagonal Win Detection Test

This test ensures that the method that is used to detect wins is able to correctly detect diagonal patterns. We would mimic a board with three diagonal counters of the same player, then add the fourth one to form the win. The test would then invoke a method like `checkWin()` and expect it to return `true`. This confirms that the code is correctly scanning in all directions and handling cases like diagonal wins.

5.2.3 Multiple Captures Test

This test is to ensure that the system can handle a more complicated move that results in three captures at a time, one horizontally, one vertically, and one diagonally. The test would set up a particular board configuration, execute the triggering move, and then verify that all the three captured counters were properly removed. This is crucial because it tests whether the system can check in multiple directions and handle all of them in one update cycle.

5.3 Why Unit Testing?

The implementation of unit tests such as those described would provide assurance that core game logic functions correctly [20] before adding features like animations or user input to the Connect 4 game. The system would provide development support through functional testing of new features without compromising existing functionalities [20]. Running these tests would enable us to verify future changes to the capture rule by confirming that all functions remain functional as expected.

CONCLUSION

The assignment revealed the extensive structure required for software design through the analysis of a modified version of the game Connect 4. Through our work with user stories and CRC cards and different testing types we developed skills to solve problems methodically while creating systems that function well and are easy to test and maintain. The assignment allowed us to visualize system operation in real conditions while determining our testing approach. The project provided us with an effective base to implement software engineering principles when designing actual systems.

REFERENCES

[1]

"Why Software Design Is Important," *IEEE Computer Society*.

<https://www.computer.org/resources/importance-of-software-design-is-important>

[2]

M. Cohn, "User Stories and User Story Examples by Mike Cohn," *Mountain Goat Software*, 2019. <https://www.mountaingoatsoftware.com/agile/user-stories>

[3]

Gillian, "Writing Good Instructions for Workshop Games – Bright Green Learning Blog," *Bright Green Learning*, Jun. 25, 2013.

<https://brightgreenlearning.com/2013/06/writing-good-instructions-for-workshop-games.html> (accessed Apr. 17, 2025).

[4]

M. Rehkopf, "User Stories," *Atlassian*, 2019. <https://www.atlassian.com/agile/project-management/user-stories>

[5]

"A guide to user story mapping (with examples)," *www.aha.io*.
<https://www.aha.io/roadmapping/guide/release-management/what-is-user-story-mapping>

[6]
Kuba Czubajewski, "Walking Skeletons: Not-so-scary Framework for Product Managers," *As We May Think — products & tools for thought*, May 28, 2024.
<https://fibery.io/blog/product-management/walking-skeleton/> (accessed Apr. 17, 2025).

[7]
M. Cohn, "Project Advantages of User Stories as Requirements," *Mountain Goat Software*, Oct. 05, 2022.
<https://www.mountaingoatsoftware.com/articles/advantages-of-user-stories-for-requirements>

[8]
"What are CRC cards? Definition, Benefits and How to Create them," *airfocus.com*.
<https://airfocus.com/glossary/what-are-crc-cards/>

[9]
"Class Responsibility Collaborator (CRC) Models: An Agile Introduction – The Agile Modeling (AM) Method," *Agile Modeling*.
<https://agilemodeling.com/artifacts/crcmodel.htm>

[10]
Paul Gestwicki, "CRC Card Analysis Example," *YouTube*, Feb. 05, 2016.
<https://www.youtube.com/watch?v=otKUer13HnA> (accessed Apr. 17, 2025).

[11]
P. Jorgensen, *Software testing : a craftsman's approach*. Boca Raton Florida: Crc Press, Taylor & Francis Group, 2014.

[12]
"The Importance of Software Testing | Informative Guide," *TSG Training*, Sep. 16, 2024. <https://www.tsg-training.co.uk/blog/what-is-software-testing-and-what-is-certification/> (accessed Apr. 17, 2025).

[13]
Imperva, "What is Black Box Testing | Techniques & Examples | Imperva," *Learning Center*, 2024. <https://www.imperva.com/learn/application-security/black-box-testing/>

[14]
Wikipedia Contributors, "White-box testing," *Wikipedia*, Oct. 16, 2019.
https://en.wikipedia.org/wiki/White-box_testing

[15]
E. Elliott, "TDD: Writing Testable Code - JavaScript Scene - Medium," *Medium*, Jan. 21, 2024. <https://medium.com/javascript-scene/tdd-writing-testable-code-30ac7a3bf49c>

[16]

Studying With Alex, "How to make code more testable, by factoring out and abstracting side effects," *YouTube*, Oct. 19, 2021.
https://www.youtube.com/watch?v=XVZpi7VJ_ws (accessed Apr. 17, 2025).

[17]

Wikipedia Contributors, "Test-driven development," *Wikipedia*, Oct. 24, 2018.
https://en.wikipedia.org/wiki/Test-driven_development

[18]

"How To Write Test Cases? Detailed Guide With Examples," *katalon.com*, 2024.
<https://katalon.com/resources-center/blog/how-to-write-test-cases>

[19]

Francesca, "Lessons Learned in Making Connect 4 - Francesca - Medium," *Medium*, Feb. 26, 2019. <https://medium.com/@fsadikin/lessons-learned-in-making-connect-4-41c0b86df360> (accessed Apr. 17, 2025).

[20]

Kitakabee, "Unit Testing: A Detailed Guide," *BrowserStack*, Dec. 23, 2022.
<https://www.browserstack.com/guide/unit-testing-a-detailed-guide>

[21]

A. Sterkowitz, "What is Unit Testing? Why YOU Should Learn It + Easy to Understand Examples," *YouTube*. Nov. 05, 2018. Accessed: Oct. 19, 2023. [Online]. Available: <https://www.youtube.com/watch?v=3kzHmaeozDI>

APPENDICES

5.4 APPENDIX A – Team User Story Map



5.5 APPENDIX B – CRC CARDS



Class: GameLogic	
Responsibilities: Determine whose turn it is. Validate moves before allowing them. Check if a move results in a win (horizontal, vertical, diagonal). Check if the board is full, leading to a draw. Apply special rules (e.g., capturing mechanics if implemented).	Collaborators: GameBoard: To analyze the grid state and detect winning patterns. Player: To confirm valid moves and switch turns accordingly. UIManager: To display win/loss messages and game status updates.
Class: UIManager	
Responsibilities: Display the game board and counters visually. Update the screen after every move. Indicate whose turn it is. Show messages for win/loss or invalid moves. Provide buttons for restarting the game.	Collaborators: GameBoard: To retrieve the latest board state for rendering. Player: To highlight the active player's turn. GameLogic: To display game outcomes and rule enforcement messages.

Class: GameController	
Responsibilities: Manage the overall game flow. Initialize game components (GameBoard, Player, GameLogic, UIManager). Handle player input and ensure correct sequence of events. Restart the game when needed.	Collaborators: GameBoard: To reset or update the grid when the game restarts. GameLogic: To control turn-based logic and check for game outcomes. UIManager: To refresh the display and notify users of events. Player: To track active players and swap turns.

SEP Assignment

GRADEMARK REPORT

FINAL GRADE

58 / 100

GENERAL COMMENTS

Eval Des (20): 10

User Stories (20): 11

CRC (20): 13

SW Testing (20): 12

Unit Testing (20): 12

FEEDBACK:

=====

The design process was a little brief and high level. The user stories should be written from the perspective of the "user" not the developer/designer of the software. Generally you have demonstrated an understanding of the story mapping process but it lacked a bit of detail.

Interestingly you seem to have a better grasp of CRCs.

The discussion of SW testing lacks detail. Explanation of testable code also required more detail, how do you write such code/what does it look like?

A few tests outlined, needed more consideration of unit testing.

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

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

