

ACCB030 - Software Engineer - Jacob Evans

by Jacob Evans

Submission date: 02-May-2025 01:52PM (UTC+0100)

Submission ID: 257621011

File name: 50439_Jacob_Evans_ACCB030_-_Software_Engineer_-_Jacob_Evans_648183_1277157574.docx
(986.83K)

Word count: 4750

Character count: 24850



ACCB030 - SOFTWARE ENGINEERING PRINCIPLES AND TESTING

Assignment

Evaluating software development approaches,
methodologies, techniques and collaborative working, while
recognising the importance of software testing.

Jacob Evans
2311160@student.uwtsd.ac.uk

Contents

Introduction.....	2
Design Methodologies.....	2
User Story Map.....	2
User Tasks	3
User Stories.....	3
User Story Map.....	6
CRC Cards	6
Software Testing	9
Testable Code.....	9
Black and White Testing.....	10
Black box Testing	10
References	15

Introduction

Software engineering is incredibly important for the development of technology and innovation. The development of technology is reliant on offering the highest quality of product to the consumer, as well as providing high quality services that the consumer is asking for. Software engineering relies on multiple methodologies and approaches, which includes the Agile Team method. This report will evaluate these methods and approaches. The report will discuss the benefits and flaws of working as an Agile team, discuss the user stories created. It will also evaluate the importance of software testing, such as black and white box tests, and when should software testing be implemented.

Design Methodologies

There are multiple methods and approaches to developing software, with examples as “Waterfall”, “Scrum”, “Spiral Model” and “Agile”. Agile was decided upon when approaching how to develop code the game idea stated in the Assignment. This is extremely important when developing software/code, as we want to guarantee that the team all know what, when and how to implement and develop the application/game.

Unified Modelling Language (UML) is a blueprint for software engineering that allows for high communication and collaboration within a project [1]. UML has a varied number of approaches, with different diagrams achieving the same goal but all look different and are implemented differently.

User Story Map

For this assignment, the team discussed a User Story Map diagram seen in figure 1. This is a form of Agile Method, where collaboration is incentivized, where people in a group can bring ideas forward on what is needed for the application, who would use this application, why the application is created and how to develop the application.

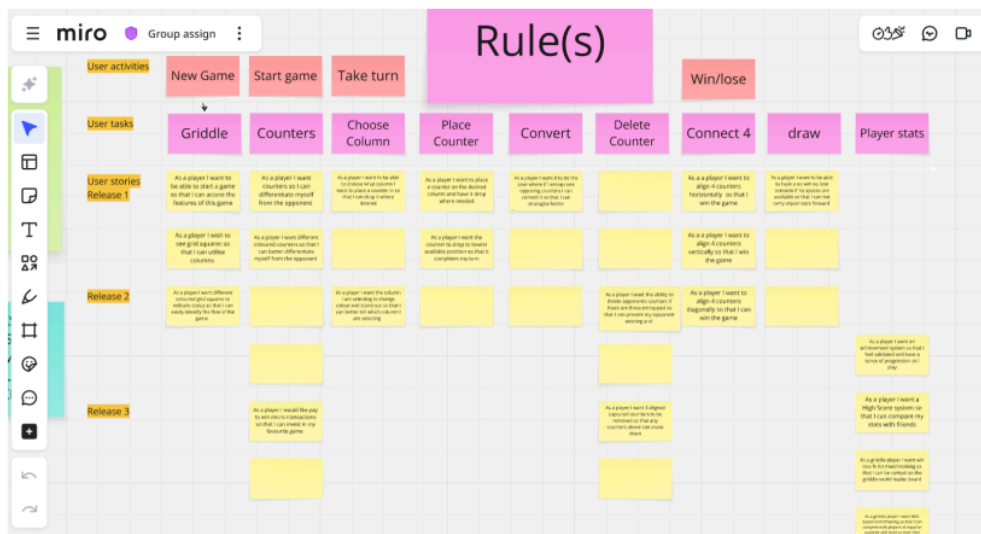


FIGURE 1 - USER STORY DIAGRAM

We can see that the User Story diagram has “layers” down the side of the diagram, mapped as “User Activities”, “User Tasks”, and “User Stories Release 1-3”. This gives us as software engineers a basis/skeleton as to what the “customer” has asked to be created (the game).

The main goal of this collaboration is to make a working game following the rules provided by the assignment. By following a main “flow”, the game has been discussed about what is needed, which are the “User Activities”. A new game is started, and a player takes a turn. The game then continues, implementing “rules”, until there is a winner or loser. This is the main flow of the User Story. As a team, this was flow was agreed upon quickly, as we all knew how the game should be implemented. The collaboration worked well at this point, as a decision was made decisively quick.

User Tasks



FIGURE 2 - USER TASKS

The next row would be the “User Tasks”, which would be “tasks” performed by the user when performing one of the mentioned activities. For example, when “start game” is pressed, the user would have to choose their counter colour (red or yellow). The next task then would be for the game to officially start, and the user choose a column on where to place the counter. The counter would then be placed. User 2 would then get the chance to choose a column, until one of the three rules occur. The rules being the counter being converted, deleted or 4 connected counters (the win statement), wherein the game will do something different. The user stories below these user tasks provide more detail as to what these tasks need to implement. This collaboration allowed us to think about the user needs/wants, as well as the needs/wants of the application itself. The story mapping allowed us to identify how complicated the rules would be to implement, and to prioritise what would be needed to create the desired application.

User Stories

User stories provide a description of who the user tasks affect, a how of the desired effect and why the user task is implemented. This gives the software development ideas on how to implement a

user task/story, and why it should be implemented. If there is no “who”, or “why”, then the developer is unsure who is affected and why task is needed to be implemented in the first place. It allows others to understand what is needed, for whom, and why.

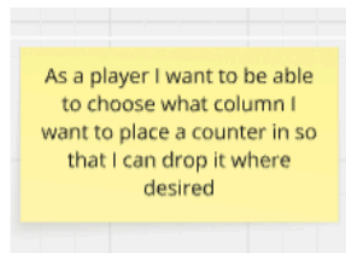


FIGURE 3 - USER STORY COLUMN

As seen above in figure 3, a user story has been created for the user to decide where the counter should be placed. This is necessary for the user, as the game would not be playable if the user watches the application place the counter, or the counter gets placed in a different column. Without this story, the game could not be implemented or even played correctly and takes out the strategy of the game. This story can also be extended easily. An easy extension to this specific story would be that when the counter is placed, the counter falls down the column to the first empty position. This is necessary for the game to work properly and gives a specific description to the software development. A specific descriptive user story allows the software developers to develop what the consumer wants. Despite this user story being descriptive, and implies that the counter should “drop”, the user story would be better as “As a player, I want to choose a column to place the counter, where the counter will fall to the next available position, so I can place my counter at the desired position”. This would be easier for a software developer to understand, as they now know to implement “gravity” into the game.

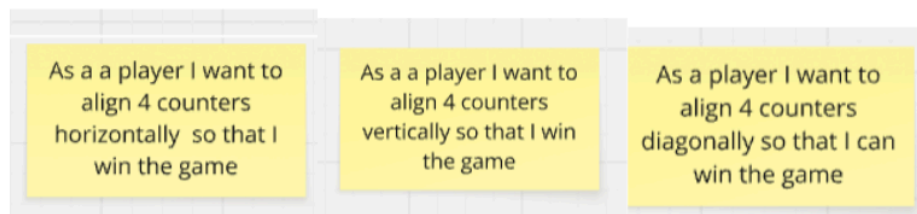


FIGURE 4 - USER STORY WIN CONDITION

Figure 4 shows the “win condition” user stories. These stories provide the rules on how the user would win the game. These are core gameplay mechanics required of the application, and without these, the users cannot win. The win/lose condition is necessary, as the application would have no way of ending without closing the application, and the users would have no use or fun using the application. Despite the user stories being descriptive, again, the user stories can be changed so the developers understand correctly. Adding “same counters aligned”, would mean that the developer knows to implement 4 same-coloured counters as a win condition, rather than any 4 coloured counters. A precise win condition would also be easier to test during test phasing of the development.



It's always better to be descriptive but concise, like speaking to a toddler.

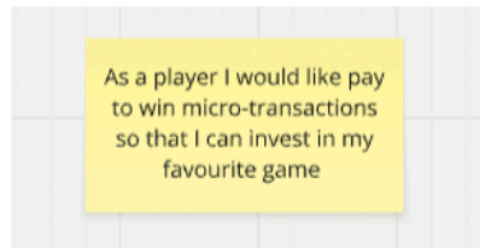


FIGURE 5 - UNNECESSARY STORY

The final story that will be discussed would be the collaborative “unnecessary story” as seen in figure 5 above. This was added as a discussion point for the group, as the group was split 50/50 on whether this would need to be implemented into the final product. This would be considered a weakness of using a User Story Map diagram and the Agile method, as not all decisions are made unanimously. Disagreements are common in collaborative work, and this was no different. Members of the group were determined to have this as a user story, as a way of making profit when creating this application, while others disagreed as micro-transactions are wrong, especially if the app is needed to be purchased in the first place. The understanding was that micro-transactions can be used in a store for purchasing different counter colours/designs and background designs, which would add some creativity and colour to the game. This was a showcase that not all user stories are necessary, and that collaboration will always have disagreements.

User Story Map



FIGURE 6 - USER STORY MAP TIME-EFFORT

This was another UML diagram created by the group. This was created to discuss to effort and time needed to implement elements of the application. On the left would be the "easiest" and the "fastest" elements to implement, and the far right is the "hardest" with the most effort. The elements on the left were agreed upon quickly, that these would be the simplest to implement effectively and efficiently. Several agreements and disagreements were made with the rest of the elements, with a "high score system" being a talking point as to how simple it would be to implement. The group had decided on keeping the "high score system" in the second column despite disagreements on the simplicity of creating a high score system.

The User Story Map has several strengths and weaknesses, where collaboration has allowed new ideas to be expressed and discussed, with the potential to assign tasks. Those who believe that certain tasks are easier than others can assign those tasks to themselves, which may increase productivity and efficiency during a timescale. Allowing a group to collaborate effectively and create descriptive yet precise stories allow this Story map to be referenced with other software developers, who can read the user story map diagram and understand what the application wants/needs. The story map also allows for new ideas to be developed later in different releases, so developers can prioritise a functioning application that fills the criteria. Relying on finishing the first release allows the consumer to give feedback on what the other things the application can introduce.

CRC Cards

CRC cards are used as another design way to collaboratively create "Classes" for the development of the application. We use CRC cards to decide the responsibility of the class, and the collaborators. Classes are given objects that the class contains, and then how the classes are connected [2]. In the example of the CRC for "Menu", its responsibilities would be to contain the menu options such as "Start Game" and "Exit Application". The Menu class would collaborate with

the “Game” class, so we know that when “Start” is selected at the menu, then the code is then directed to the “Game” class. This is incredibly useful when used with the User Story map, as it allows software developers to understand how the application should work. Software developers can start to imagine how they would implement the user stories into the classes. CRC cards are the foundation of implementation but aren’t usually descriptive. CRC cards work better when they are referred to the more descriptive and expanded User Story Map. Each class needs its own responsibilities but does not need to collaborate with any other class. The design is incredibly simple and should always be easy to understand.

We can see that CRC cards were created for the Connect 4 application, with the classes “Player”, “Grid”, “Game”, “Counter”, “Menu” and “Control”. During the discussion of these CRC cards, we believed that the CRC cards were detailed enough to create this application. Despite this at the time, these classes look to be missing some details.



FIGURE 7 - CRC CARDS

“Menu” class would need to have a collaboration with the “Game” class, otherwise the Menu would not be able to start the game. This CRC card would not be detailed enough to be implemented correctly, despite it having the correct responsibilities. This CRC is rather a skeleton of CRC card, which contains enough responsibilities, and does reflect the criteria identified in the user stories, but most likely does not contain enough details to enable code to be developed.



FIGURE 8 - MENU CLASS

While the “Game” class is the most detailed CRC card that contains all necessary responsibilities and collaborators.

Software Testing

Software testing is incredibly important during the development of software. Software bugs appear in lines of code written by software developers. Bugs are simply described as parts of code that does something that isn't expected of the program, parts of code that does not do what is expected of the program. Bugs can also cause system crashes, freezes and can cause serious consequences, ranging from security flaws to monetary loss. Software testing is the act of monitoring and testing code as the code is developed to mitigate these issues. It is said that for every 1000 lines of code, there is estimated to be 150 errors [3]. Even 5% of code errors of a 25000-line code estimates to be 125 errors, therefore testing these during development process is incredibly important. Businesses tend to implement "bug tracking" which is a tool that integrates testing into the development of code. This allows for bugs to be located, monitored and reported, allowing for mitigation or removal of bugs.

An example of this was CrowdStrike in 2024, where the company had released a faulty/errored software update that caused 8.5million Windows devices to be corrupted [4]. This caused major issues and monetary losses across the world, and even social media was affected for 6 hours (no TikTok). Bugs/errors can usually be detected if the tests are completed during development, and writing "testable code" is the best way to mitigate bugs. Testing code sooner rather than later makes editing the code far less strenuous, and mitigates the damage done by bugs.

Testable Code

Testable code can be measured in several ways, such as "Operability", "Simplicity", "Observability", "Stability" and "Understandability" [5].

- **Operability** - When an application operates correctly, is it easier to test, as the tester can see the desired (or undesired) outcomes of the application. This would make the testing more efficient and effective.
- **Simplicity** - Keeping the code simple, precise and without repeating code allows for the software tests to be implemented effectively, as the tester will know what areas have the bug. If code is repeated 20 times, then the tester would have to check/change those 20 lines of code, which is a waste of time, and laziness can cause issues further down the development line.
- **Stability** - When code is not changed much when sending updates, then testing will remain efficient, as the code has been tested previously. By making many changes, the changes can cause disruptions to the testing.
- **Understandability** - Testing is more efficient and effective when there is more information available. Documentation is incredibly important when testing software, as bugs that have been reported in the past may get discovered again. Documentation would allow testers to implement the same solution as before. Changes to the code/design should be documented, well organised and accurate.

Tests can be done dynamically and statically; wherein dynamic tests are done while the software is running and using the software to test if the application is working correctly. Static testing is

visually inspecting the requirements of the application, as well as inspecting and reviewing the code. Visual Studio has a debug tool that allows users to debug and run tests without the software running.

Black and White Box Testing

Black Box testing is testing by focusing on validating the functionality of the provided requirements, from the CRC cards and User Map Story, while White Box testing is testing based on knowledge of the software created by looking at the code/application.

Black and White Box testing have different objectives, methods and scope for testing software/applications, which is why both are used when testing code but usually at different instances of the software development.

Black Box Testing



FIGURE 11 - BLACK BOX TESTING

Black Box testing is testing the functionality of an application without the code and software. The premise is does 15 times 10 make 150. The desired outcome is that 150 is the answer, with the desired numbers multiplied by each other being 15 and 10 respectively. Black box testing prioritises the end product; what the user sees/interacts with. Black Box testers have no need for coding knowledge, as they'll test the interactions of using the application (i.e. counters moving as intended, application swaps player turn after counter is played, menus work correctly).

There are 3 types of Black Box testing [6]:

- **Functional Testing** - This is testing if the application works as intended/described. The tester performs test cases, which covers different functions of the application. The tester would put an input into the application and then check the output. This is vital testing for creating correct and efficient applications.
- **Non-functional Testing** - This is testing the performance of the application, rather than what the application does. This is testing is on the scalability, reliability, and performance of the application. Does the application run slower than expected? Is the application using more memory than expected? Does the application have a large CPU usage than expected? These are incredibly important during testing, as it can determine the limitations of the application, gauge how the application looks/runs to the user and can save time and money before the application is released on different devices.

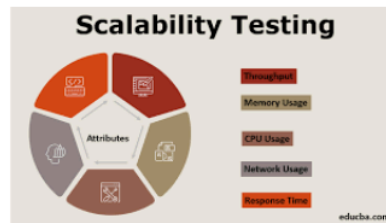


FIGURE 12 - SCALABILITY TESTING [7]

- **Regression Testing** - As “regression” implies, this is running the same tests after changes to code are implemented. This ensures that the code is functional and reliable and maintains its integrity throughout the development of the application.

There are many advantages to using Black box testing, as testers do not need former knowledge of programming skills to test an application. Many “game testers” use black box testing through releasing “alphas” and “betas” of the game to the public, as this broadens the black box testing and if bugs are detected, then the feedback can be left and read through reviews (Steam).

These tests are easily reproducible (required that the tests are documented) and is used to find contradictions in the application.

Example scenarios of Black box testing Connect 4 application:

- **Test 1: Creating empty grid/gameboard.**
 Pre-conditions: Main Menu, mouse hovering over “Start” button.
 Input: Clicking start game.
 Expected outcome: New screen with created empty grid/gameboard ready to play.
 Goal: Validate that the main menu text has disappeared, and the board has been created.

This is to validate that the Main Menu option removes the main menu text and creates a grid ready to play. This should be a fast process, with no lag and low CPU usage, as the grid should be a 6 x 7 grid, and CPU usage should be low. This is to test the scalability and performance of the application.

- **Test 2: Counter drop in column 5.**
 Pre-conditions: Empty grid/board.
 Input: Select column 5 with mouse.
 Expected outcome: Counter falls to the bottom of column 5, grid position 5 x 7.
 Goal: Game mechanics work correctly, and counter stops at the bottom of column.

This is to validate that the counters interact correctly with the grid, where the counter should appear at the bottom of column 5. This is a required mechanic of the game, and we don’t want the counter not appearing or disappearing off the screen.

- **Test 3: Opponent chooses column 5.**
 Pre-conditions: Column 5 with one red counter at bottom.
 Input: Select column 5 with mouse.
 Expected outcome: Yellow counter falls to grid position 5 x 6, lands on top of red counter.
 Goal: Counters interact with each other correctly, core game mechanic works, counter stops at next available space.

This is to validate that the counters interact correctly with each other, where the yellow counter should appear on top of the first red counter placed. This allows for the game to be played correctly and allows for the counters to be stacked on each other.

- **Test 4: Game Over/Win/Loss Screen.**
Pre-conditions: 3 yellow counters placed vertically in column 3.
Input: Select column 3 as yellow player.

Expected outcome: Yellow counter falls, font appears with "Winner: Yellow. Play again? Yes, No", and the 4 yellow counters illuminated with transparent colour.

Goal: Game implements Win/Loss screen and allows user to either play again or close application.

This is our Win/Loss condition test, to confirm that the game works correctly. The goal of this game is to Connect 4, and the players should be awarded with "Winner" should they win. We do not want this game to be looped, therefore a smaller menu should pop up which allows the user to play again or exit the application. This test would be implemented differently if the application was played online, where a "Ready" option would be implemented so each player can decide if they want to play again.

White Box Testing

White Box testing is different from Black box testing, where the tester has knowledge of the internal workings of the application (the code and architecture). It is also known as "Structural" or "Code-based" testing, where the tester creates test cases to see the path the application follows, and to see if they meet specified requirements. The White Box tester performs tests early in the development phase to make sure the code is functioning as intended, and the tests are done as the developer, rather than an end user [8].

Commonly used white box testing are:

- **Input and Output Validation** - Applying different inputs and recording the outputs to see if it is the desired output. For example, a simple calculator application would need the application to multiply correctly, therefore different inputs would be tested to see if the output is correct.
- **Loop Testing** - Any loops developed in the application is tested to confirm that the program is functioning correctly and is not causing any infinite loops.
- **Path Testing** - This will check all possible execution paths in the program and checks to see if the code behaves correctly. It verifies that all logical loops and conditions are functioning correctly.

There are 3 types of White Box testing:

- **Unit Testing** - checks if function of the application is working correctly and will check to see if it meets requirements from the CRC cards and User Map Story.
- **Integration Testing** - This testing examines how different parts of the application interact with each other, and make sure that these different parts work with and without one another.
- **Regression Testing** - This is the same as the Black box testing method, wherein that the testing focuses on making sure the application/code is still functioning after updates/changes to the code.

Example scenarios of white box testing Connect 4 application:

- Test 1: Counter falls to empty cell in column.
Pre-conditions: Empty grid/gameboard.
Input: void counterDrop(char "yellow", int column5).
 bool CheckForEmpty()
Expected output: Counter drops until it reaches bottom of the grid.

This should traverse the code path from the top of the grid to the bottom of the grid, which would verify that the counter is interacting correctly with the grid "hit box", as well as interacting correctly with empty spaces within the grid. The bool can be looped until the Boolean returns false.

- Test 2: Destroying captured counters.
Pre-conditions: Red counter 7x1, yellow counter 7x2, 7x3, 7x4.
Input: void counterDrop(char "red", int column7).
 bool CheckForEmpty()
 bool CheckForColourDown(if char yellow, int row2,3)
 deleteCounter()
 CheckForEmpty(char "red", int column7)
Expected output: As red counter drops, yellow counters between two red counters are deleted. The red counter then drops down to 7x2.

This is to test the counter removal logic provided by the rules for the CRC cards and User Story map. Its testing the removal of code path and then opening the code path again.

- Test 3: Full column.
Pre-conditions: Varied red and yellow counters filling column 1.
Input: void counterDrop (char "yellow", int column1).
 bool CheckForEmpty()
 If false, return "Column filled, you cannot go there".
Expected outcome: Text shows above stating column is full and it is an illegal move, and to try again.

This is testing to see whether this move is rejected or not. A player should not be able to place another counter in an already filled column, and the application should advise the player that this is illegal. This also tests a failure exit path, and that the counter should not drop or effect any other counter on that grid.

These are also examples of unit testing in the scenarios above. Unit testing is making checks on individual parts of code, checking to see if those codes/functions run one-by-one in the application, and it improves code quality by testing each unit in isolation. These tests are run automatically when code is changed.

Advantages of unit testing [9]-

- Bugs are detected earlier in the development process; this saves time, effort and money further down the "waterfall" process, as bug fixing becomes more tedious and expensive the further into the development process.
- Code quality is improved as developers become more confident with their code if the tests come back correct. This mitigates the chances of bugs being created/found as the process continues.

- Documentation of the tests and bugs allow for developers to understand the application/tests, as well as maintain the software correctly. Regression testing would be improved with better documentation.

Disadvantages of unit testing [9] -

- More effort and time put into testing during the development process, and even more time spent when dealing with larger or more complex applications/software.
- Testers would need previous knowledge of coding languages to begin testing, compared to Black box testing. Test code would need to be clear, precise and documented effectively, which would be complicated when the tester does not have experience in coding languages.
- Automation can cause bugs to be overlooked, or not even discovered as humans are more likely to think of new ways to break the software. Bugs could arise that aren't visible during automated testing but appear during public release of the software.

Conclusion

When developing software or an application, the software engineering principles and methodologies are incredibly important. Creating clear and precise code and mitigating bugs and errors allow for the creation and innovation of new applications, which would have a more positive effect rather than a negative one. This report recognises the importance of software testing and understands that an Agile method is incredibly useful when creating an application as a group. Everyone has their own opinion and ideas, and these can be incredibly constructive towards a project. While group collaboration relies on good communication, individuals in the group may have different knowledge and experiences which can cause some “push and pull” when communicating. This was seen when creating the User Map stories and User Story Map (effort and time diagram), but new ideas were forged from the tension. There are several design methodologies and techniques used in different businesses and industries, and receiving first-hand experience in a safe environment has brought a newfound respect for these methodologies and techniques.

References

References

- [1] "Welcome to UML Web Site!" Accessed: April 25, 2025, [Online]. Available: <https://www.uml.org/>
- [2] "Class Responsibility Collaborator (CRC) Cards: An Agile Intro", *Agile Modeling*, <https://agilemodeling.com/artifacts/crcmodel.htm> (accessed April 25, 2025).
- [3] "What Is Bug Tracking? | IBM." IBM. <https://www.ibm.com/think/topics/bug-tracking> (accessed April 26).
- [4] J.Tidy. "CrowStrike IT outage affected 8.5 million Windows devices", *BBC*, <https://www.bbc.co.uk/news/articles/cpe3zgznwjno>
- [5] K.Walia. "Writing Testable Code: Principles, Patterns, and Pitfalls", *Linkedin.com*, <https://www.linkedin.com/pulse/writing-testable-code-principles-patterns-pitfalls-kamal-walia-vmmrnc> (accessed April 30, 2025).
- [6] "Black Box Testing - Software Engineering." <https://www.geeksforgeeks.org/software-engineering-black-box-testing/> (accessed April 30, 2025).
- [7] Y. Goyal, "Scalability Testing", *EDUCBA*, <https://www.educba.com/scalability-testing/> (accessed April 30, 2025).
- [8] "White box Testing - Software Engineering." <https://www.geeksforgeeks.org/software-engineering-white-box-testing/> (accessed April 30, 2025).
- [9] "Unit Testing - Software Testing." <https://www.geeksforgeeks.org/unit-testing-software-testing/> (accessed May 1, 2025).

FINAL GRADE

90/100

GENERAL COMMENTS

Eval Des (20): 17

User Stories (20): 18

CRC (20): 20

SW Testing (20): 18

Unit Testing (20):17

FEEDBACK:

=====

I would have liked a little more discussion of user story examples.

I would have liked a more evaluation e.g. I would have expected more stories in the map perhaps you could have discussed this.

You could have expanded more on the types of 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
