

Assignment – Software Engineering Design & Testing

Ben Roberts (2205382)

Software Engineering Principles and Testing (FT - ACCB5030)

University of Wales Trinity Saint David – School of Applied Computing

Table of Contents

Table of Contents.....	2
Introduction.....	4
Task 1.....	5
Scenario and user story map.....	5
Analysing user stories.....	6
Task 2.....	11
CRC Cards.....	11
Analysing CRC cards.....	12
Task 3.....	15
Importance of software testing, Black & White Box Testing.....	15
Testable Code.....	16
Task 4.....	17
Black Box Tests.....	17
White Box Tests.....	20
Unit Testing.....	21
Conclusion.....	22
References.....	23

Table of Figures

Figure 1: User story map.....	6
Figure 2: Brainstorming section of the board labeled idea dump.....	7
Figure 3: Screenshot of an expanded Miro user story.....	8
Figure 4: Another unfinished user story.....	9
Figure 5: Medium quality user story.....	10
Figure 6: Example of a CRC card.....	11
Figure 7: Full screen view of all the CRC cards used during the planning process.....	12
Figure 8: Co-ordinates CRC card.....	13
Figure 9: Rust code based on what was written in the CRC card.....	14
Figure 10: Example of a unit test in Rust.....	21

Introduction

The purpose of this document is to explore the process of software engineering and testing by designing a theoretical web-based Battleships game in accordance with the Agile methodology, while discussing topics related to Agile and software testing. It also devises a number of tests that could be performed on the aforementioned Battleships game.

Task 1

Scenario and user story map

A design was produced for a theoretical Battleships game that would run in a web browser. It follows a derivative of the standard rules for the board game (being based on a 10x10 grid where players place five different ships on their side and attempt to guess the location of each ship the opponent is using) but with the addition of two features as per the specification:

- Players are able to perform a defensive manoeuvre once per game, which moves one of their ships by a one tile offset.
- Players can also use a “drop bomb” ability once per game, which destroys an entire ship at once if it hits.

The design work was to be done under the assumption that the theoretical product is a web-based browser game, and this design was produced as a group task with a group of several other students using a Miro board.

The design attempts to conform to Agile specifications, which are a set of practices and development methodologies focused on teamwork, communication and efficiency [1]. Agile is focused on initially producing a minimum viable product and then expanding on it over time. Agile projects tend to be split into multiple smaller teams focused on a singular aspect of the software and involve frequent short standup meetings on a daily basis in order to reassess priorities as necessary. Agile also involves frequent negotiation with the customer and/or product owner to alter specifications as necessary.

The primary benefit of the Agile model is its purported flexibility when compared to older models such as Waterfall which involve rigid specifications. The specifications of Agile software are made with the full intent of changing over time, which allows it to adapt to shifting requirements during the development process. Welcoming changing requirements rather than rejecting them is considered an essential component of the Agile methodology [1].

User stories were written in an “idea dump” section of the Miro board and were then translated into a user story map, which is a method of visually laying out a potential workflow in order of priorities and releases. User story mapping is helpful to build a shared understanding among all participating members of a project [2].

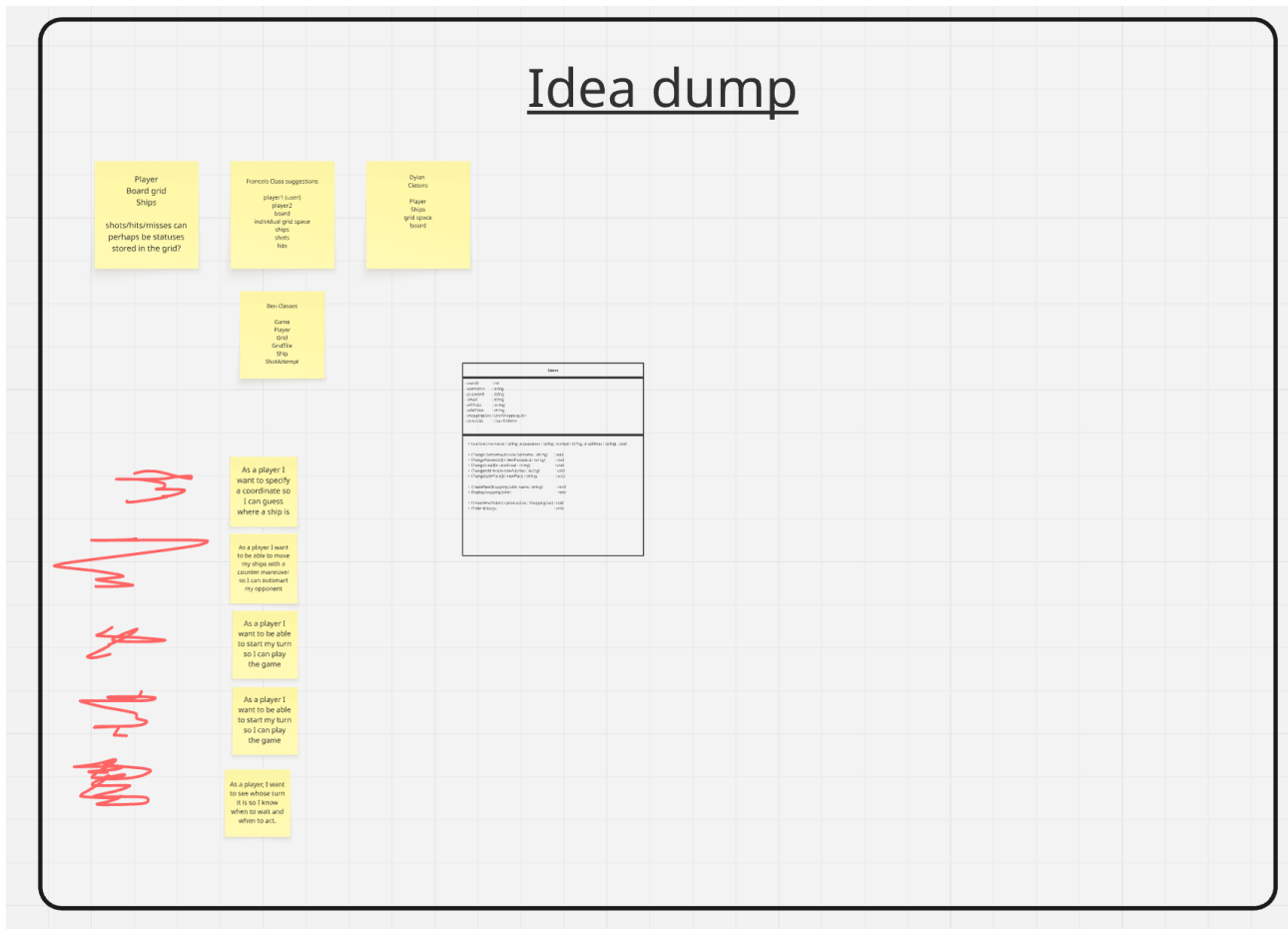


Figure 2: Brainstorming section of the board labeled idea dump

Following one of the topmost user stories remaining in the idea dump, it reads “as a player I want to specify a coordinate so I can guess where a ship is”. This user story is relatively concise but is ultimately lacking in detail. Nonetheless, this hook could later be expanded upon in the user story map when it is worked on later.

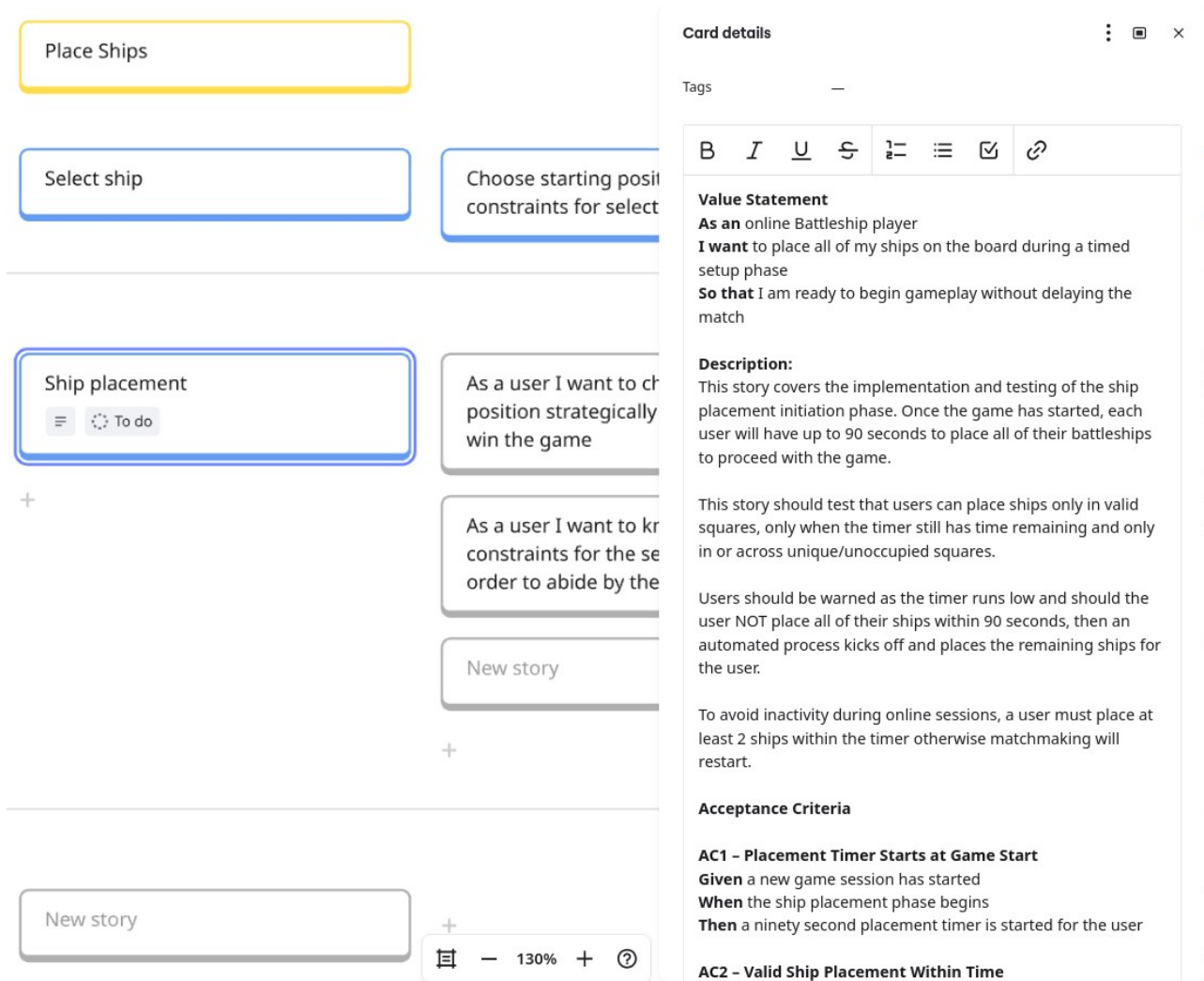


Figure 3: Screenshot of an expanded Miro user story

The user story could be expanded to view a more detailed description which went into further discussion including descriptions, requirements and acceptance criteria. The user story hook itself was more detailed in general without being excessively over-detailed. The specifications overall include more detailed requirements including time constraints and potential fallbacks should the user not provide adequate input. The acceptance criteria is similarly concise, featuring a given state when conditions are fulfilled as well as what to do in that scenario. This acceptance criteria could, as an example, be used to implement the software and/or be used as a basis to describe a CRC card (explored below).

However, due to time constraints and a number of team members not being present, a number of user stories (especially those near the latter half of the user story map) remained unfinished. As an example, one of those user stories was labelled "As a user I want to be able to choose where to put my ships so I can play."

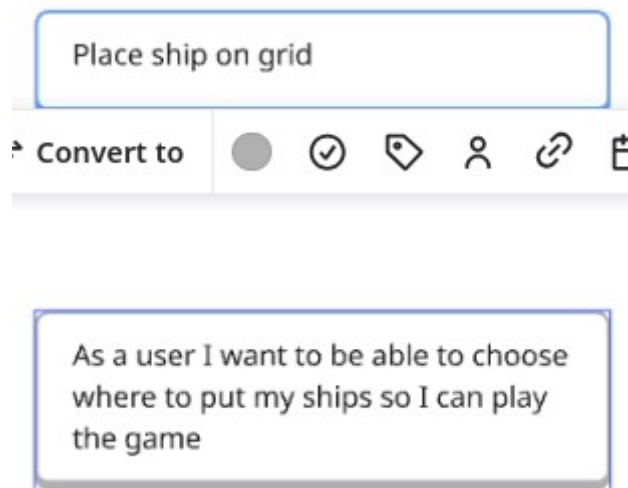


Figure 4: Another unfinished user story

An immediately apparent issue with this user story is the ultimately limited description, not unlike the prototype user stories from the idea dump. This user story also does not have an expanded description and, in combination with the limited latter third of the description (“so I can play the game” is ultimately lacking in explanation) leaves too much up to interpretation to properly create a CRC card or write software based upon it. Furthermore, acceptance criteria is not provided which adds a further obstacle to the implementation process due to a lack of explanation as to what is required. Given more time and experience on the writer’s part this user story could have possibly reached a similar level of quality to the other one.

Finally, this is a later user story covering the win-lose state which is less detailed than the first user story but is much more developed than the second one.

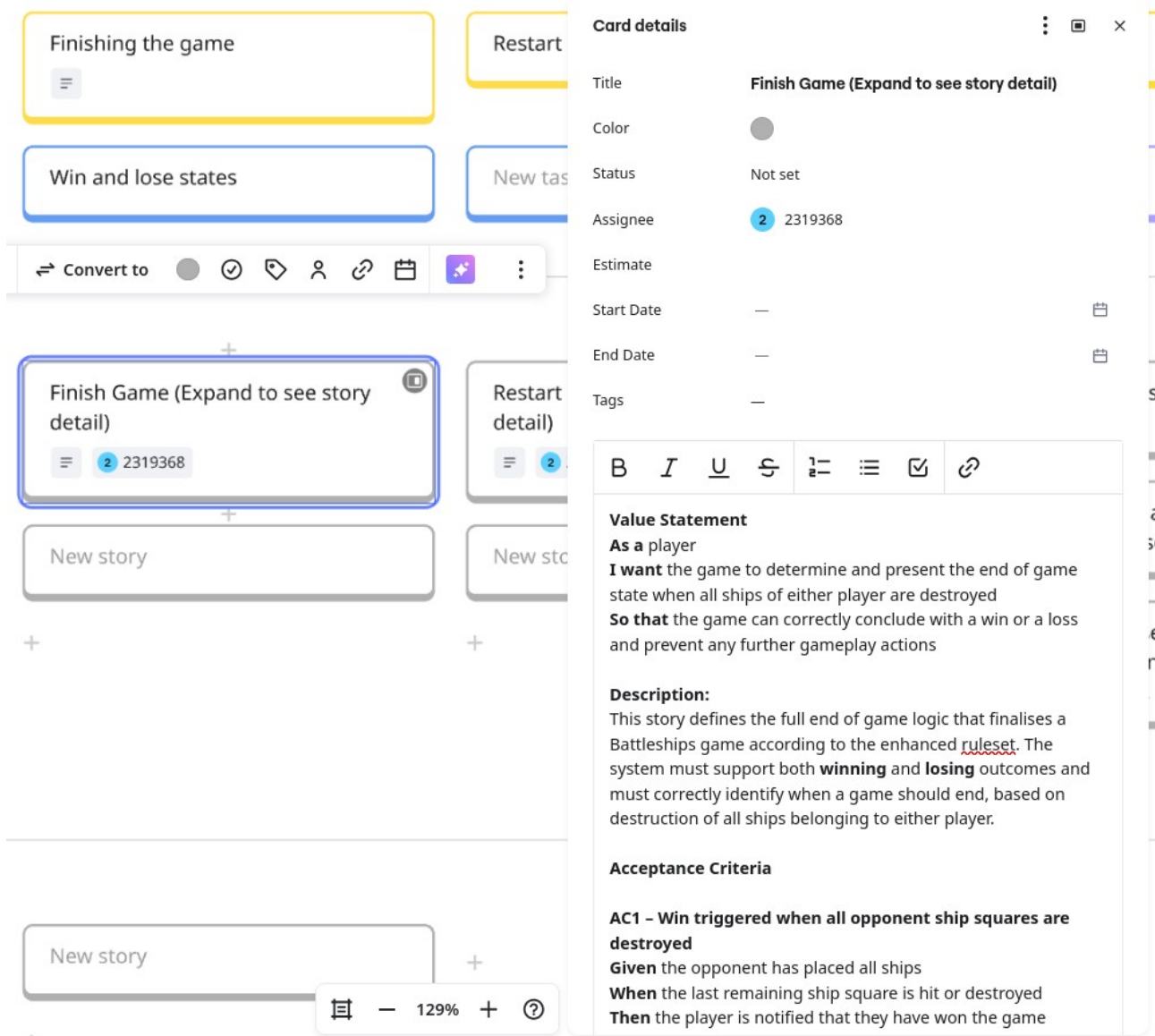


Figure 5: Medium quality user story

This is a user story covering the win-lose state. It reads “as a player I want the game to determine and present the end of game state when all the ships of either player are destroyed so that the game can correctly conclude a win or a loss and prevent further gameplay actions”. This user story, while descriptive, is too long and would likely work better as two or more separate stories. The description presented is also shorter than the one in the first user story, being only a single paragraph, and less acceptance criteria is available.

Task 2

CRC Cards

Class-Responsibility-Collaboration (CRC) cards are a method of modelling potential data structures that may appear in a software project [4]. They are commonly used in object-orientated programming that is frequently related to the Agile process. CRC cards list the name of the class on the top its responsibilities (i.e, what role it would actually serve in the program) and its collaborators which are other classes it is likely to interact with.

A primary benefit of CRC cards is that they can be used to quickly describe a potential class in a without needing to implement it directly.

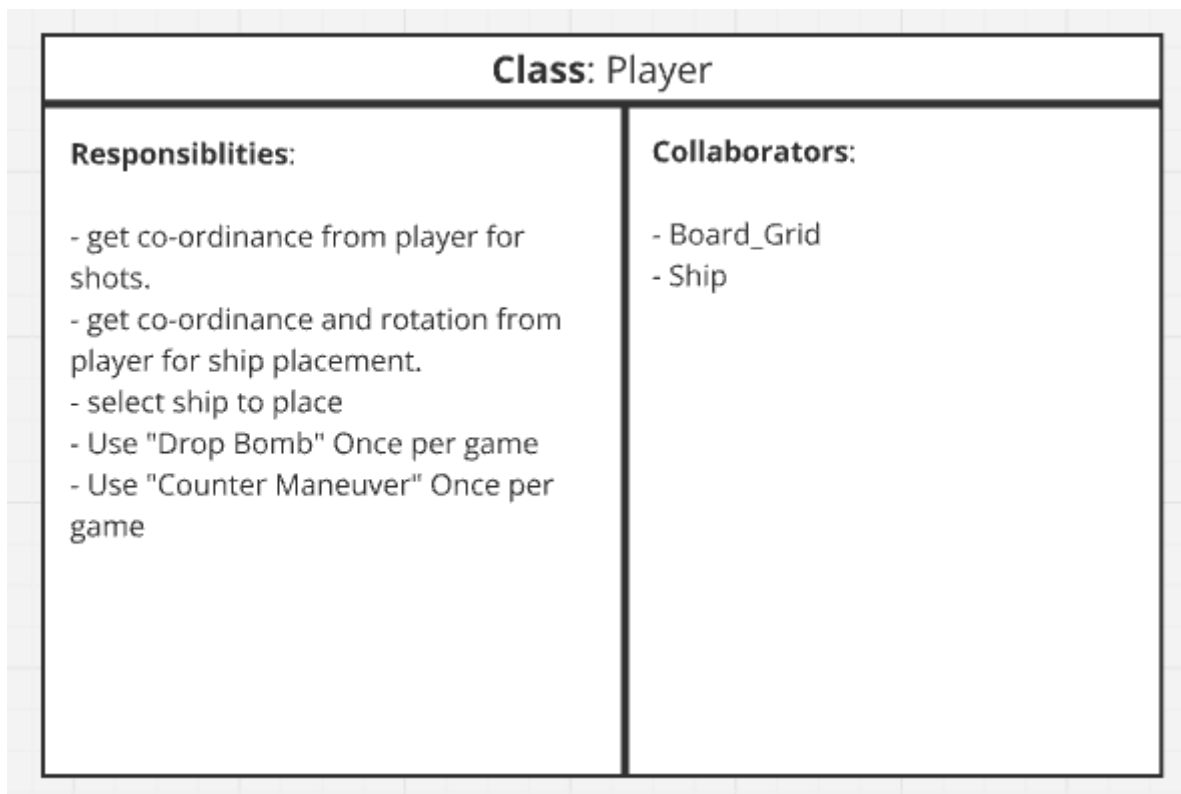


Figure 6: Example of a CRC card

Analysing CRC cards

With the above user stories in mind, a set of CRC cards were made to model how the program could potentially work.

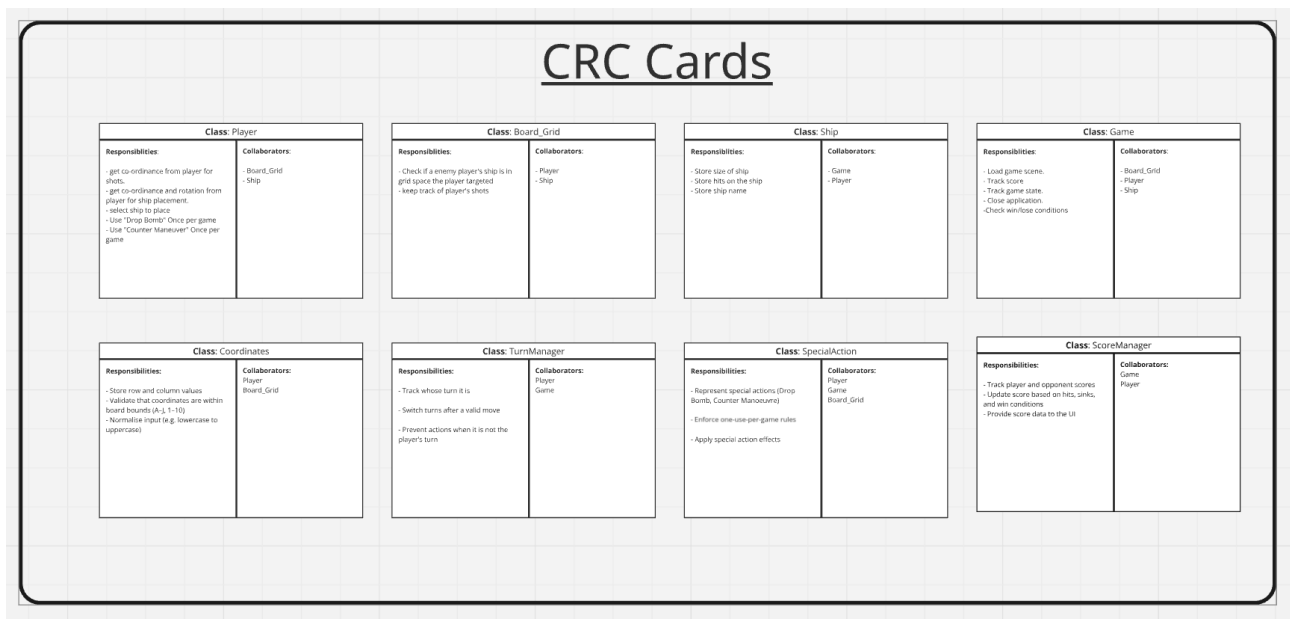


Figure 7: Full screen view of all the CRC cards used during the planning process

Compared to the user story map which ultimately got too complicated and uneven, the CRC cards are comparatively much simpler to understand. However, the term collaborators could be difficult to understand as it could be interpreted as either a dependent or a dependency (it's best described as encompassing both).

Some classes may not make it through to implementation as they appear redundant and could be fulfilled in less code and/or memory space by another class.

Using the coordinates CRC card as an example since it is the simplest, some Rust code was written for demonstration purposes in order to see whether or not it is possible to write a theoretical program using these CRC cards as a basis.

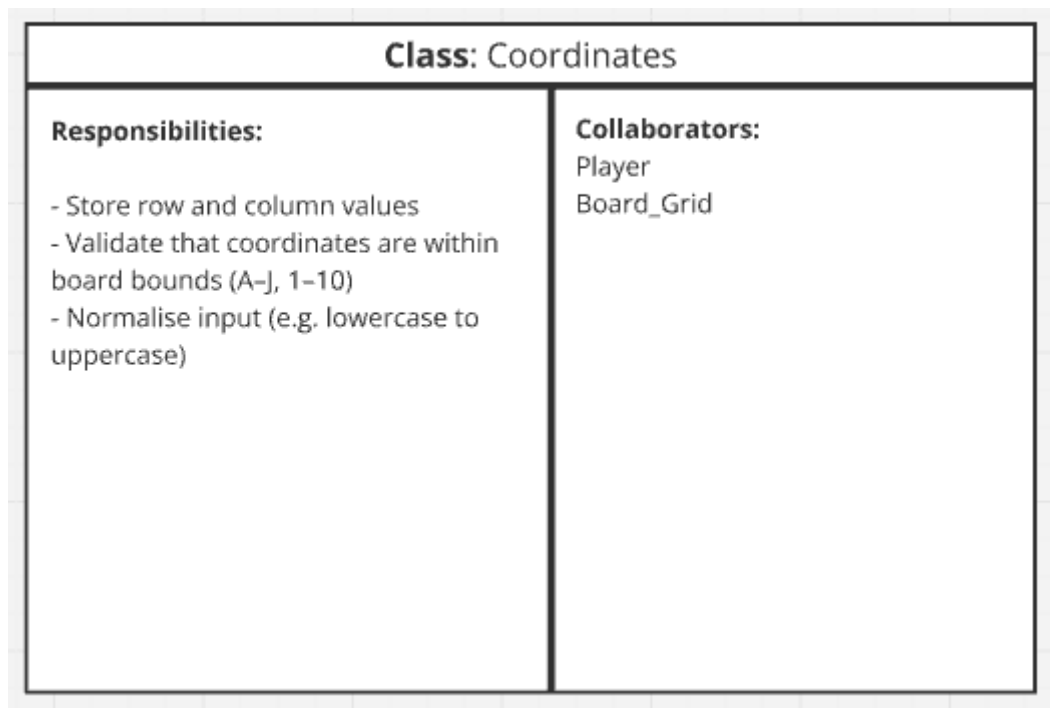


Figure 8: Co-ordinates CRC card

Following the information that was written on the CRC card, a simple file containing only a class (referred to as “structs” in Rust) along with accompanying methods and a helper function for extracting integers from a passed character value for the purposes of reading them as coordinates. This particular CRC card contained sufficient detail to implement everything that the coordinate class needed, although its collaborators may require a more complex implementation.

```

use std::io::{Error, ErrorKind};

struct Coordinates {
    > row: u8,
    > column: u8
}

impl Coordinates {
    > pub fn get_row(&self) -> u8 {
    >     > self.row
    > }
    >
    > pub fn get_column(&self) -> u8 {
    >     > self.column
    > }
    >
    > pub fn new(x: u8, y: u8) -> Result<Self, Error> {
    >     > if x >= 10 || y >= 10 || x < 10 || y < 10 {
    >     >     > return Err(Error::new(ErrorKind::Other, "Co-ordinates out of range"))
    >     > }
    >     >
    >     > Ok(
    >     >     > Self {
    >     >     >     > row: x,
    >     >     >     > column: y
    >     >     > }
    >     > )
    > }
}

pub fn letter_to_column(letter: char) -> Result<u8, Error> {
    > let lower = letter.to_lowercase().to_string();
    >
    > match lower.as_str() {
    >     > "a" => return Ok(1),
    >     > "b" => return Ok(2),
    >     > "c" => return Ok(3),
    >     > "d" => return Ok(4),
    >     > "e" => return Ok(5),
    >     > "f" => return Ok(6),
    >     > "g" => return Ok(7),
    >     > "h" => return Ok(8),
    >     > "i" => return Ok(9),
    >     > "j" => return Ok(10),
    >     > _ => return Err(Error::new(ErrorKind::Other, "Invalid character"))
    > }
}

```

Figure 9: Rust code based on what was written in the CRC card

Task 3

Importance of software testing, Black & White Box Testing

The purpose of software testing is to ensure that the program does not experience any unintended behaviour that could impede the standard functionality of the program and introduce reputational, commercial and potentially even societal damage [5]. An organisation that does not properly test and debug their software will find that it will be negatively received by customers and as such will likely fail to make a return on investment. In some cases inadequate testing in widely used software may cause untold amounts of issues, outages and disruption worldwide should it emerge – an example of such as the incident in 2024 with the CrowdStrike Falcon software which caused a large number of enterprise systems running Microsoft Windows to enter a boot loop, resulting in a large number of essential services such as hospitals and airports to become unable to operate [6], [7].

The process of testing often involves debugging, which is focused on uncovering and fixing potential issues that come up whilst trying to avoid introducing more errors in the process. The end goal of debugging in this case is to achieve the intended outcome of the program. Many tools exist for debugging, such as code analysis tools (such as linters), compiler breakpoints and log tracing can be used for debugging. Software debugging often involves a test harness, which is a specific software setup for the purpose of testing and debugging.

Software tests can be split into two major groups, those being white box and black box tests. The difference between the two is that white box testing is performed with source code access while black box testing is not. It is preferable to design black box tests before designing and implementing white box tests because knowledge of the code can cause tests to become biased around the functionality of the code itself rather than the actual intent for the software. White box testing is often used for verification purposes while black box testing is used for validation purposes [8].

An example of black box testing would be manually inputting data into the program while it is running to see if it still works after that. White box testing would instead automate the testing process by directly inserting entry data into the program and performing assertions to see if the intended outcome has occurred.

One particular example of black box testing would be User Acceptance Testing (UAT). This specific approach involves getting an end user (such as the stakeholders for the program or potential customers) to use it in order to determine whether or not it meets the required specifications. As an example of a black box test which involved UAT, a program designed to assist pharmacists in screening diabetic patients for low blood sugar was sent for testing by a focus group of 32 pharmacists of different demographics who each needed to perform a number of actions (such as accessing menus and viewing patient data), after which they would fill in a questionnaire wherein they would rate a statement from 1 to 5 points

depending on how much they agreed with it, such as whether or not they agreed that the application was easy to use and whether they found it consistent in practice. [9]

White box testing often involves code specifically written to perform automatic tests on software. An example of a kind of automated white box test would be unit testing which involves small tests written to test only a single particular facet of the software [5]. For example, in a test case for different forms of testing, white box testing was performed on a demo program for a registration database [8]. When white box testing was used, every file was individually checked for issues relating to logic and security. Issues with code quality were found, such as unused functions and that it did not comply with standard programming procedures. As an example, it was found that database connections were embedded onto the page which caused issues with performance and resulted in overall worse code quality with frequent use of duplicated code. A code review was performed afterwards, greatly improving the quality of the program for production.

Testable Code

Testable code is software code that is designed to be easy and practical to test, often with a focus on simplicity, portability, understandability and maintainability [10]. Testable code is often heavily commented and uses clear function and variable names to assist with readability. Writing code for testability is essential for ensuring proper quality control when in the testing stage and also makes white-box testing processes significantly less cumbersome [10].

Test-driven development is a form of software development which aims to achieve high testability by writing the white box tests for the program before the actual implementation [11]. This allows for faster feedback, easier maintenance and overall higher reliability when used, but it requires a significant amount of experience with white box testing in order to work efficiently [12].

Task 4

A number of potential tests have been designed for this program in accordance with the acceptance criteria, and both black and white box tests are available.

Black Box Tests

Name	Description	Pre-conditions	Inputs	Expected Results
Open registration form	Pressing the register button should prompt the user to register	Server must be running	Press register button	Registration form should be displayed
Mandatory field validation	User must fill all required fields of the registration form	Server must be running, registration form must be accessible	Leave one or more mandatory fields blank	Registration should be prevented and user should be informed of empty fields.
Password minimum rules	Passwords must meet minimum requirements (e.g, at least a certain minimum length, uses uppercase and lowercase letters, numbers and special characters, not on most common passwords list)	Server must be running, registration form must be accessible	Enter password that does not conform to the requirements	Registration should be prevented and user should be informed of password requirements.
Prevent duplicate usernames	If a user tries to use a username that has already been registered by another user, registration must be prevented	Server must be running, registration form must be accessible, account must already be created	Enter username that has already been reserved	Registration should be prevented and user should be told to choose a different username
Authentication	Entering the correct username and password for an account should provide access to the game	Game server must be running, an account must have been created	Valid username and password	That particular user should be logged in and be shown a dashboard
Incorrect username	Entering an unrecognised username should not log the player in and should prompt them to register for an account.	Game server must be running	Invalid username	User authentication should fail due to an invalid username. Prompt player to create an account.

Correct username, incorrect password	Entering an incorrect password should block logins. Consider telling users to reset password	Game server must be running, an account must have been created	Valid username, invalid password for that user	User authentication should fail due to an incorrect password. Prompt user to reset password if they forgot it.
Web app functions correctly on modern browsers	Using a modern browser with HTML5 support should let you play the game	Game server must be running, use a HTML5-compatible browser	N/A, just go to the webpage	Page should load fine regardless of browser as long as it's up to date
Block incompatible browser	Using an older browser without HTML5 support should block access, tell users to install a newer browser	Game server must be running, use a browser incompatible with HTML5	N/A, just go to the webpage	Page should redirect users to an error page that tells them to install a newer browser (e.g, Firefox)
Start matchmaking	If a logged in player decides to start random matchmaking, they should be placed in a matchmaking queue and be matched up against a different user	Two accounts must be logged in on different devices and should start matchmaking	Both accounts start matchmaking	Server should detect that two clients are in the matchmaking queue and should pair them up for a game
Cancel matchmaking	While in the queue, users can cancel matchmaking if they change their mind about playing the game before it starts.	User must be in the matchmaking queue	Cancel button is pressed	Client should signal to server that the user wants to leave the matchmaking queue, user should return to the front dashboard
Prevent matchmaking	If the user is already in the queue or playing a game, prevent starting matchmaking	User must be in an active game or the matchmaking queue	Press the button to start matchmaking	Matchmaking attempt should be prevented and user should be informed
Match start	Upon starting a round, the player must be able to choose where to place their ships	Match has started	N/A, just start round and see the prompt to place ships	Round starts and ship placement screen is visible
Ship placement	User can choose anywhere on the 10x10 grid to place a ship. Ships may only be placed	Match has started	Click on the grid	Ships are placed

	where they can fit.			
Valid placement enforcement	Attempting to place a ship outside the grid should fail.	Match started	Click on the very edge of the grid	Ship placement should be restricted to inside the grid
Placement rotation	Pressing the rotate button should attempt to place the ship horizontally instead of vertically	Match started	Press the button to rotate	Rotating works
Turn order decision	Turn order should be randomly selected by a coin flip	Both players placed ships	N/A, Just have both players place their ships	Turn order should be random; repeat multiple times to ensure randomness
Coordinate selection	Player should be able to choose a coordinate on the grid and attempt to guess if a ship is there	Turn has started	Click anywhere on the grid	Guess should be parsed and feedback should be given
Shoot down	If all locations of a ship are guessed, the ship must be eliminated	Turn has started, enemy ship has been hit enough times	Click on all the grid locations of an enemy ship	Ship should be marked as eliminated
Drop bomb	The drop bomb is a special action that can eliminate an enemy ship with only a single successful guess of its location	Turn has started, drop bomb has not been used	Click the button for drop bombs, then click the location of an enemy ship	Drop bomb should be used up and enemy ship should be marked as eliminated
Counter manoeuvre	The counter can be used to move a ship by a single tile offset once per game	Turn has been started, counter manoeuvre has not been used	Choose a ship to move, then choose a direction	Ship should move by one tile in that direction
Special action single use	Special actions (drop bomb, counter manoeuvre) should be single use, clicking the button to use a special action after using it once should prevent from working.	Turn has started, at least one of the special actions has already been used	Click the button to use the special action after it has already been used	Special action should fail and turn should not end
Turn end	Guessing a location or using a special action should end the turn	Turn has started	Guess a location or use a special action	Turn should end and control should move to the opponent; opponent should then be able to

				move
Game end	If all ships are eliminated on either side, the game should end and declare the player whose ships were not entirely eliminated the winner	All ships on either side have been destroyed	Guess the final location	Game should end

White Box Tests

Name	Description	Pre-conditions	Inputs	Expected Results
Panic handling	The program should detect invalid errors and try to recover from them instead of crashing	Error handling code must be made	Intentionally pass an invalid result that would cause an error	The program continues to function and does not panic
Variable types	The program should use the smallest possible variable types for global variables to save memory	Variables must be implemented	Pass a value higher than what is reasonably needed (e.g, 256 for 10x10 coordinates)	Program should refuse to parse those values and not panic, underflow or accept them
Asset loading	Assets should load from the server into the client	Assets are added	Load an asset	Asset loads fine and matches the checksum
Converting co-ordinate letters to numbers	The program should turn letters A through J into numbers 1 through 10	Function for letter-to-number conversion is implemented	Pass A, B, C, D, E, F, G, H, I, J (both upper and lowercase). Then pass any other letter not mentioned above.	A, B, C, D, E, F, G, H, I, J return numbers, and should be case-insensitive for future-proofing. Others return an error.
Converting co-ordinate numbers to letters	The program should turn numbers 1 through 10 into letters A through J	Function for number-to-letter conversion is implemented	Pass 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Then pass any other number not mentioned above.	1, 2, 3, 4, 5, 6, 7, 8, 9, 10 return lowercase letters. Others return an error.
Password encryption	Any user passwords must be properly hashed, and should not be readable in	Password reading and hashing is written	Enter a password and try to access its direct value from the output	Output is garbled and impossible to interpret. Output should also be salted differently for the same

	cleartext			password value each time.
Memory management	Classes should be freed from memory when unneeded to prevent memory leaks	Classes implemented	Create a variable and check its location in memory	Memory should have been overwritten (or at least set as free)

Unit Testing

Unit testing is when code is written to test specific individual aspects of a program. It is done with the intent of concisely testing a singular feature of the program rather than the entire program at once [13].

```

52 | }
53 |
54 | #[cfg(test)]
55 | mod tests {
56 |     » use super::*;
57 |     » #[test]
58 |     » fn test_letter_to_column() {
59 |     »     » assert!(1 == letter_to_column('a').unwrap())
60 |     »     }
61 | }
62 |

```

Figure 10: Example of a unit test in Rust

A number of the white box tests mentioned above could likely be performed as a unit test. One of those tests would be “Converting co-ordinate letters to numbers” (which is demonstrated in the above figure using Rust’s implementation of unit tests) as well as its counterpart “Converting co-ordinate numbers to letters”. These would likely require an assertion to check whether or not the specified characters return the intended value. Another check which could be performed using unit testing would be the loading test, which could load a specific file from the server and check its SHA-256 checksum to ensure that it has been loaded correctly (thus confirming that file loading from the server works on both the client and server side).

Conclusion

In conclusion, the Agile methodology has been found to generally be appropriate to developing software as a team. The project planning processes used within Agile have been found to enable quick planning of a software project in order to enable rapid turnaround for a program. However, it is heavily reliant on a team being present and depends on said team being experienced with the Agile process. Non-attendance in the assigned group for the task also served to add further difficulty to the Agile process. This was most likely exemplified fact the writer (and most likely the rest of the group save for one person) was unfamiliar with the Agile methodology prior to working on the task.

The benefits of software testing have also been discussed and analysed. Compared to Agile which is strictly reliant on a team, many of these testing practices and methodologies could also be applied to a project which is only being produced by a solo developer, although few case study resources were available for white box testing in the databases that the writer had access to.

Given more time and an actual requirement for the Battleships game to be produced, it may have been possible to gain a better understanding of Agile methodologies and testing processes in practice

References

- [1] A. Stellman and J. Greene, *Learning Agile: Understanding Scrum, XP, Lean, and Kanban*. Sebastopol, UNITED STATES: O'Reilly Media, Incorporated, 2014. Accessed: Jun. 04, 2026. [Online]. Available: <http://ebookcentral.proquest.com/lib/trinity/detail.action?docID=1840882>
- [2] J. Patton and P. Economy, *User Story Mapping: Discover the Whole Story, Build the Right Product*. Sebastopol, UNITED STATES: O'Reilly Media, Incorporated, 2014. Accessed: May 29, 2025. [Online]. Available: <http://ebookcentral.proquest.com/lib/trinity/detail.action?docID=1781254>
- [3] I. K. Raharjana, D. Siahaan, and C. Fatichah, 'User Stories and Natural Language Processing: A Systematic Literature Review', *IEEE access*, vol. 9, pp. 53811–53826, 2021, doi: 10.1109/ACCESS.2021.3070606.
- [4] L. Hvam, J. Riis, and B. L. Hansen, 'CRC cards for product modelling', *Computers in industry*, vol. 50, no. 1, pp. 57–70, 2003, doi: 10.1016/S0166-3615(02)00143-4.
- [5] Aniche Maurizio, *Effective software testing: a developer's guide*, [First edition]. Shelter Island, NY: Manning Publications Co., 2022.
- [6] E. Belitzky and N. Sullivan, 'CrowdStrike Outage: Digital Catastrophe Lesson in Social Responsibility.', *Journal of Critical Incidents*, vol. 18, pp. 57–59, Jan. 2025, Accessed: Jun. 18, 2026. [Online]. Available: <https://research.ebsco.com/plink/54373c99-3a05-30fc-8086-e6e6ebe2e53b>
- [7] J. Yimga, 'Cybersecurity failures and air travel disruptions: estimating the impact of the CrowdStrike IT outage on Atlanta International Airport', *Journal of transportation security*, vol. 18, no. 1, pp. 6–, 2025, doi: 10.1007/s12198-025-00296-w.
- [8] Vedran Marić, 'Difference and Application of White-Box, Black-Box, and Grey-Box Testing', *International Journal of Innovative Solutions in Engineering*, vol. 2, no. 1, pp. 39–53, 2026, doi: 10.47960/3029-3200.2026.2.1.39.
- [9] M. K. A. Jaya, Z. Ikawati, F. Rahmawati, N. M. Yasin, and I. G. N. A. C. Putra, 'Digitalization of HYPOGLYRISK and Evaluation of Its Utility for Pharmacist-Led Hypoglycemia Risk Assessment in Type 2 Diabetes Mellitus', *Jurnal Ilmiah Medicamento*, vol. 12, no. 1, pp. 146–157, 2026, doi: 10.36733/medicamento.v12i1.13619.
- [10] S. Parsa, M. Zakeri-Nasrabadi, and B. Turhan, 'Testability-driven development: An improvement to the TDD efficiency', *Computer standards and interfaces*, vol. 91, pp. 103877–, 2025, doi: 10.1016/j.csi.2024.103877.
- [11] T. Hasanli, S. Siddeeq, B. Khanal, P. Kotilainen, T. Mikkonen, and P. Abrahamsson, 'TDD Governance for Multi-Agent Code Generation via Prompt Engineering', 2026, doi: 10.48550/arxiv.2604.26615.
- [12] T.-B. Trinh and N.-T. Truong, 'TCG: an approach to improving test-driven development by exploiting use case specifications', *International journal of system assurance engineering and management*, vol. 17, no. 4, pp. 1159–1174, 2026, doi: 10.1007/s13198-025-03025-7.
- [13] Aniche Maurizio, *Effective software testing: a developer's guide*, [First edition]. Shelter Island, NY: Manning Publications Co., 2022.