Advanced Programming Module A2

Safiya Behanzin

Battleship Contents page:

AdaShip Assignment Backlog:

1.Challenge Outline

1.1 Problem Outline

Proposed with a challenge to render a refined high-level game that exercises a range of intricate logic and total functionality that embodies gamification within a strict time frame.

1.2 Solution

Build a Battleship game that has a prominent theme of sea warfare, consisting of a centralised problem of ships being under ‘attack’. The battleship game requires the implementation of several functionalities that meet the complexity of the game's requirements in addition to other creative exclusive additions to appeal to the user and differentiate this battleship game from others.

1.3 User stories

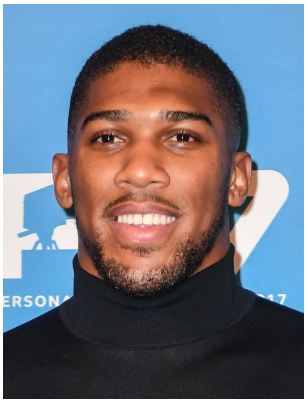


Figure 1, user story

Antony Joshua

Antony is a professional boxer, who competes globally for heavyweight lifting championships.

He enjoys playing PlayStation/Xbox games and general quick games to keep him occupied while resting or commuting to gym sessions.

He likes to bond with his younger nieces and nephews regularly.

During family gatherings, Antony must be updated with all the cool exclusive games his more youthful family members are interested in so that he can remain the winner so multiplayer games are ideal.

 *Figure 2, user story* **Tammy Hem**	Tammy is a fitness coach with two sons. She likes to get fun thought-provoking games that are not excessively violent for her kids to play with. Tammy also looks out for readable simple formatted games that her children can follow easily without her shadowing.

1.4 User requirements

Based on sea warfare game research and user stories I established clear SMART objectives to remain on track throughout the project.

Figure 3, screenshot from miro board of user requirements based on user stories

1.5 UML Diagram

The UML diagram below was designed after a few iterations of my design planning.

My UML diagram is a graphical representation of the structure and behaviour of the Battleship games system, using a set of standardised symbols and notation. Specifically this is a class UML diagram expressing all classes within the game, their attributes, methods and clear relationship between them.

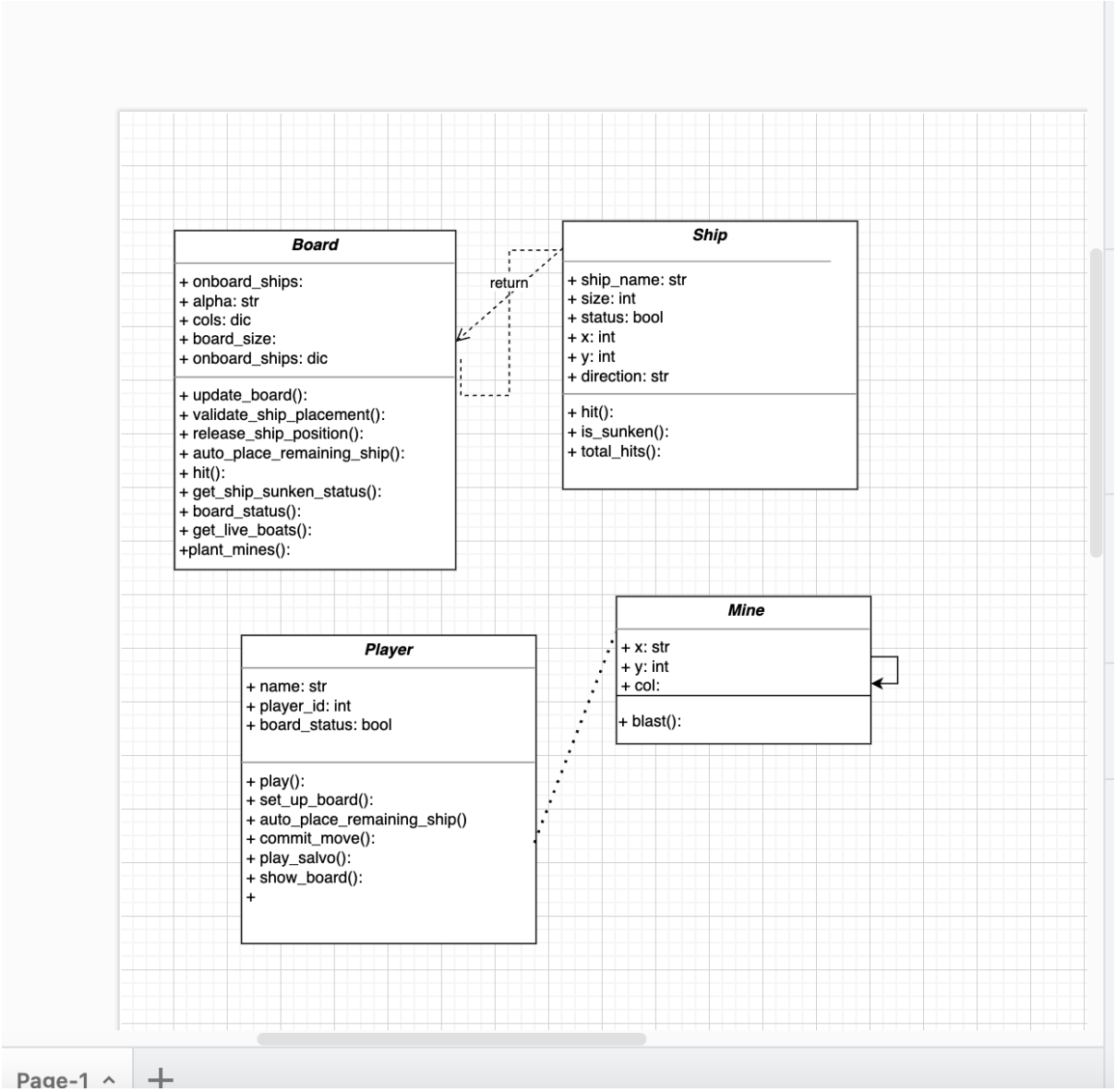


Figure 4, UML Diagram of Battleship

1.6 Initial working plan

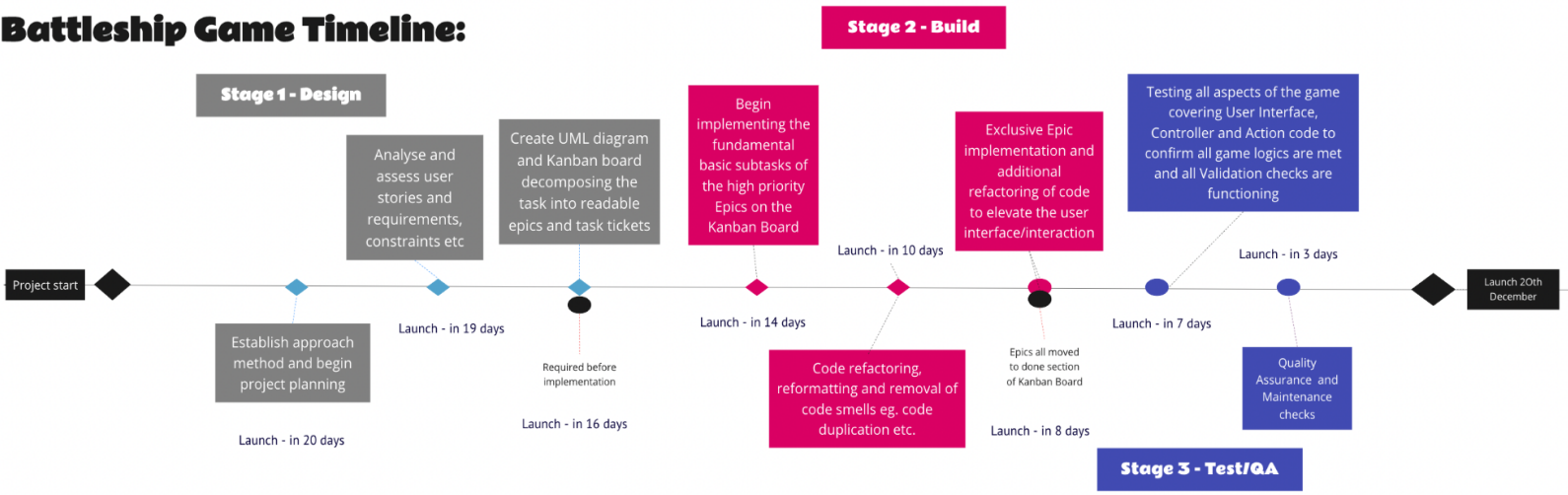


Figure 5, Battleship Game Timeline

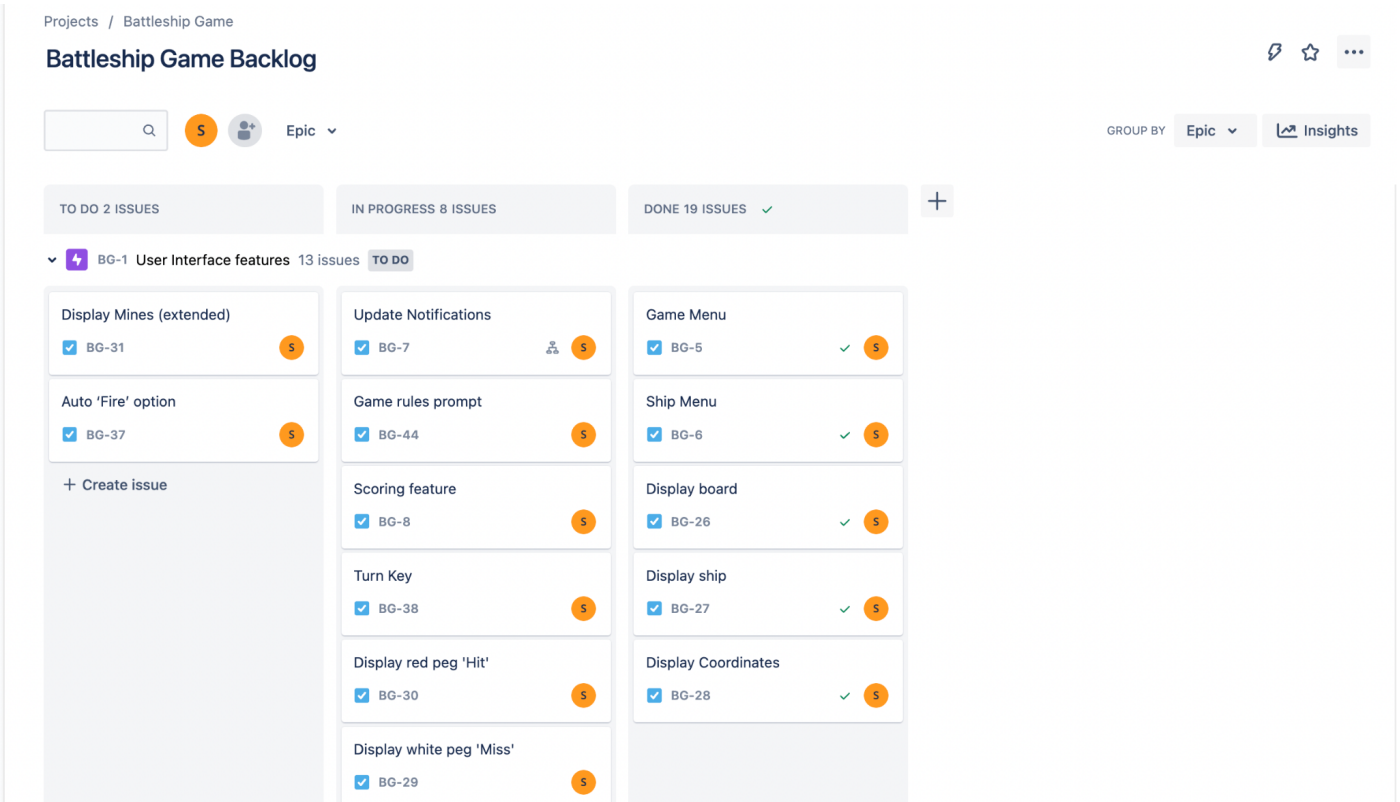


Figure 6, Battleship Game Kanban Board using Jira

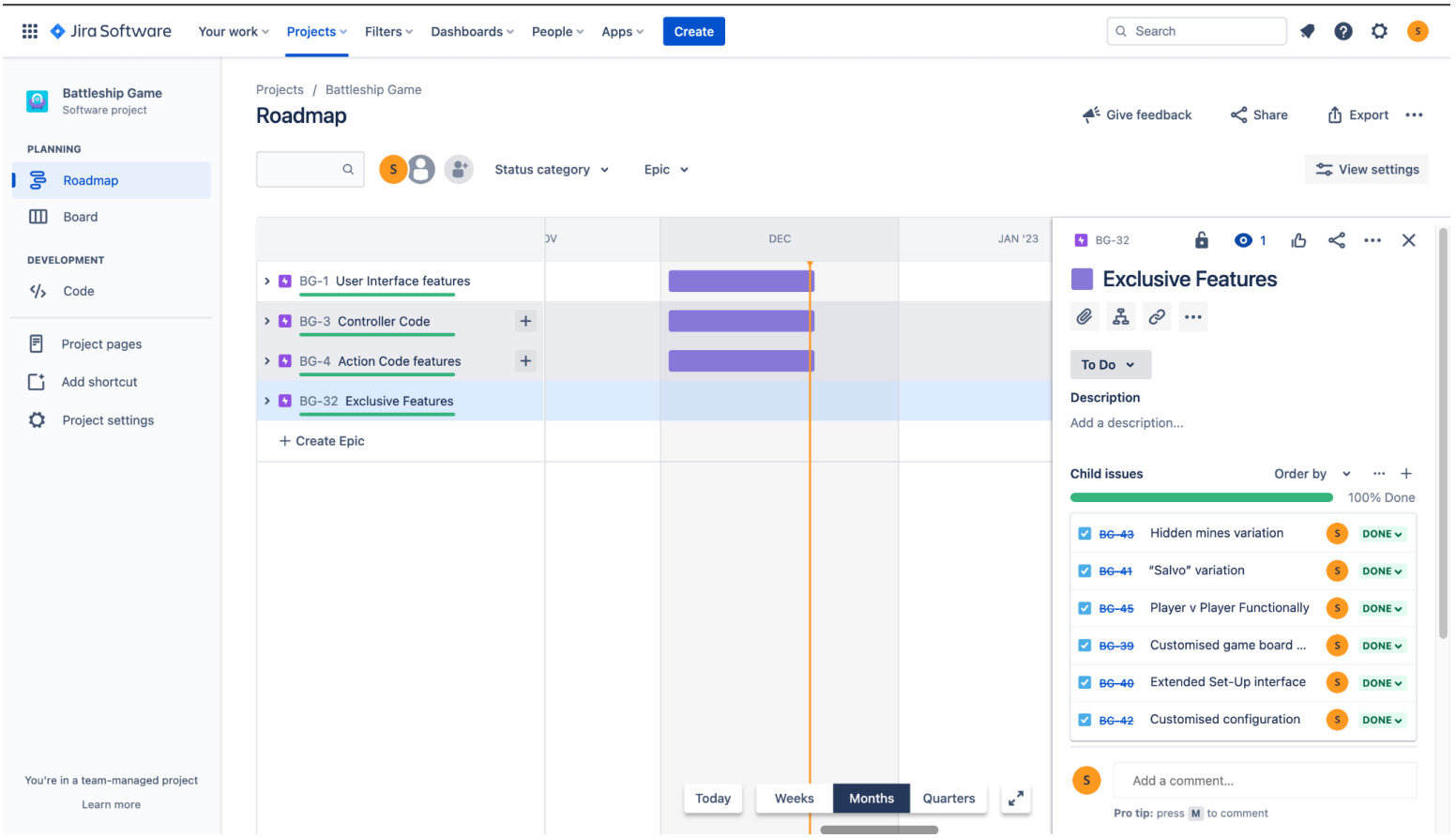
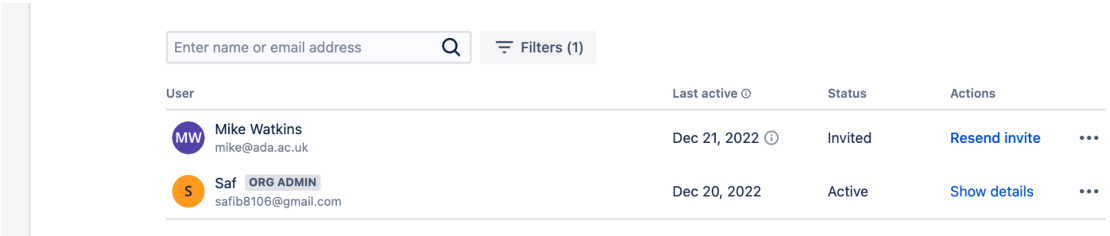


Figure 7, Battleship Game Roadmap using Jira

Link to Board: Invitation to Kanban board sent on the 20/12/22 please see to this invitation if the Kanban Board does not successfully connect to my git repository.

https://saf1ya.atlassian.net/jira/software/projects/BG/boards/1



The timeline schedule and kanban board display my initial working plan for the Battleship MVP. I adopted the Agile methodology to manage this project. Initially, I deliberated following the top-down approach, however, after some evaluation I selected the bottom-down approach.

Using a Kanban board enabled me to monitor my progress while constructing the Battleship game MVP and check all content is updated throughout. I maintained high productivity by initially decomposing my end goal into clear epics which I further broken down into measurable related tasks and subtask tickets.

The four Epic types:

1. User interface
2. Controller Code
3. Action Code Features
4. Exclusive Features

Overall the Kanban board supported flexible working when completing outstanding tasks and gave me a clear visual as to how I progressed through the project to successfully meet my deadline.

1.7 Overall Approach

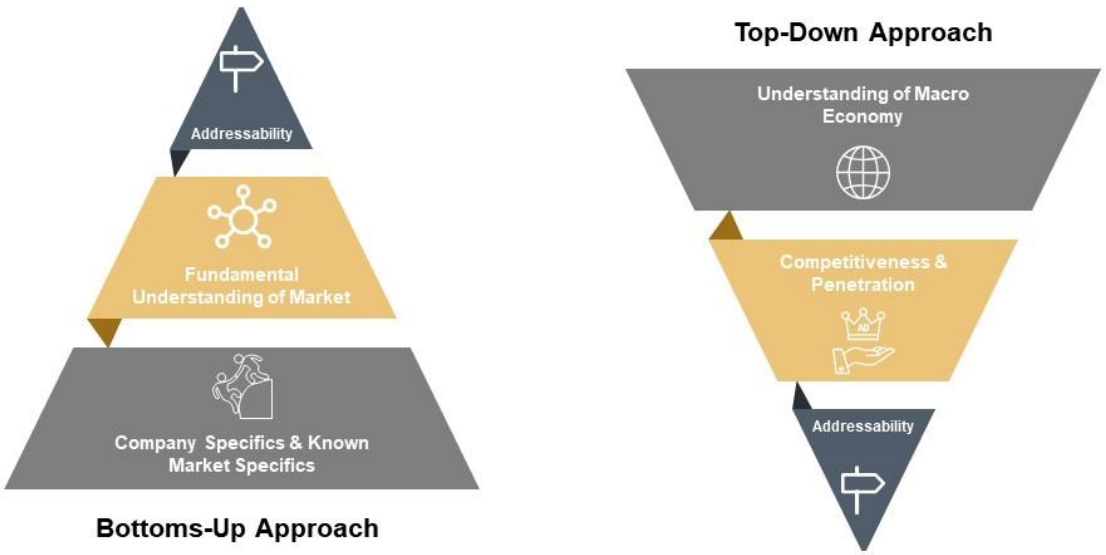


Figure 7, Bottom-Up/Top-Down Approach

	<u>Top down</u>	<u>Bottom up</u>
Advantage	• **Strong management** as clear authority lines are established • **Decreased risk** in decision making	• **Company-Wide communication**:everyone actively participate in the decision making

	process: starts with high priority/level requirements ● **Good organisation**: as business goals are already set up by upper management ● Facilitates **quality control**	● **Increased collaboration:** more collaborative and iterative process by enabling employees to have more of a say(contribution) ● Forming a unique perception of the company, goals and employees ● Ability to measure **operational risk** ● **Similarity referencing:** enables time to identify similar instances to the project
Disadvantage	● **Limited creativity** ● **Dictatorial:** approach may seem oppressive ● **Not time efficient**, slow response to challenges as upper management are the only members permitted to contributing solutions	● **Slowed time creating plans and reaching goals:** due to the number of opinions and contributions to take into account ● **Inaccurate Reflections of Data:** may cause skewed/inaccurate decisions in the long term

Analysis of the common approach methods:

The globally practised approaches Top down and Bottom up propose recognised advantages that are well suited depending on the business or project aim although they both equally unveil disadvantages. After careful analysis of these approaches, I concluded that the bottom-up approach is more beneficial in my case. Due to the project's constraints: a strict time frame of 20 days and a limited team size of one the bottom-up facilitates more planning time to refine my epics and task choices for the rendering of a kanban board, a more iterative development process, in addition to the ability to identify similarities in past projects that I have deployed and existing projects out there constructed by competitors to improve the features of my game.

Arguably the bottom-up approach introduces errors regarding inaccurate reflections of data, whereas the team size being one alleviates this capability as there are not several team contributions and opinions to manage and account for.

1.8 Development strategy and Approach to quality

There are several practised methods that I adopted to successfully maintain quality throughout the approach to programming tasks:

1. **Establish clear guidelines and standards** for what constitutes good code. Covering code organisation, naming conventions, relevant commenting, testing etc.
2. **Marked code reviews**, which are typically carried out by other developers who give feedback on code before it is deployed. I planned to use this method as it identifies problems and generally improves the overall quality of the codebase. However, in my case, I remained consistent with self-code reviews before implementing new functionality and closing out epics/tasks on my Kanban board.
3. **Testing:** testing of code is critical for ensuring that it is reliable and bug-free. Including unit testing, integration, and validation testing.
4. **Measuring** and **tracking performance**

Development: Continuous integration and delivery (though pushed commits to git assists in regulating a schedule)

2. Development

Link to my git repository https://github.com/saf1ya/AdaShip.git

2.1 Adoption of 'good' standards

Main 'good' coding standards I aimed to maintain:

- Clear documentation
- Formatting (Indentation, line length and spacing)
- Consistency with naming conventions
- Same structure code reviews
- Testing changes throughout (integration)

2.2 Phase 1: Designing and collecting requirements for the game based on different versions and documenting data

All these sections have previously been covered and explained in sections (1, 1.1,1.2,1.3,1.4,1.5,1.6,1.7 and 1.9).

2.3 Phase2: Implementing the design and algorithm into a code

These are the main sections of phase 2, I have provided code reviews and further explanation where necessary.

- **Creating code based on initial user requirements, following the epics set out in Kanban board**
- **Validating and testing the code for version 1**
- **Modifications based on errors**
- **Final run for Version 1: Player vs Computer**

2.3.1 Creating code based on initial user requirements, following the epics set out in Kanban board

First Git Commit Code Review: This is my initial beginning phase targeting the user interface epic task display board tasks/tickets.

```
class Board:
  def __init__(self, row, column) -> None:
       self.row = row

       self.column = column
```

```
    def emptyBoard(self):


        board = []
        for i in range(10):
            row = []
            for j in range(10):
                row.append(' ')
            board.append(row)
        return board


    def displayBoard(self):
        board = self.emptyBoard()
        for i in range(len(board)):
            print('|'.join(board[i]))
            print('-'*20)



obj = Board(10, 10)
obj.displayBoard()


shipChoice = input("Enter ship choice:")
print("Ship choice is: " + shipChoice)
```

This meets the task requirement by displaying a simple board to the user successfully, there is no additional functionality to meet.

First code review:
Good:

- Overall the use of a nested for-loop method confirms the boards' reliability as it limits over complication
- Begins to address the ship menu task

Possible improvement points:

- Lacks a security level to maintain the code and practise safe code usability for possible future developments of this Battleship game.

Final Git Commit for this Version (Player vs Computer): Too large to display

Last commit for version code review:
Good:

- Overall meets the all user requirements and set epics covering all User Interface, controller code and Action code tasks
- Implements recognised designed patterns such as singleton and several good coding practices, encapsulation through class/singleton, reduced code smells such as code duplication (example: nested for loops),tuple/lists to support efficient functionality.
- All action code acknowledged and understood after several iterations and code refactorings.
- Left with a scalable model Battleship Game.
- Security and validation of all elements both visible to the user such as the board and ship placement as well as the game type.

Possible improvement points:

- Improve overall consistency in code specifically with naming conventions, commenting and timely committing my gits (avoid experimenting with code, duplicating the file as a copy and forgetting to push to git, rather than just instantly pushing confirmed changes to git)

2.3.2 Validating and testing the code for version 1

I optimised my Battleship games functionality and logic through extensive testing, specifically altering between unit, regression and integration testing.

Unfortunately, the deployment process of this game resulted in several iterations, as a result consistent testing became time consuming. To resolve this issue and reduce inefficient time use I began testing before commits or when needed to successfully carry out or complete additional functionality of the task/ticket i was working through.

2.3.3 Modifications based on errors

Modifications made based on failed tests to fix error messages within the terminal.

2.3.3.1 Main challenges when completing version 1: Player vs Computer:

Based on my Kanban board backlog (Figure 6 section 1.6) you can see the main challenges laid in the controller and action code epics.

Link to my git repository https://github.com/saf1ya/AdaShip.git

1. **Ship validation**

Here I targeted the user interface epic task game menu, ship menu, display board, display ships and controller code epics including the validate board/ship placement.

```
def updateBoard(self, row, column, ship, direction)
```

```
    def shipIndex(self, coordinateShip):
```

```
        column, row = list(coordinateShip)
        row_index = int(row) - 1
        col_index = coordinates[column]
        return row_index, col_index

    def updateBoard(self):
        ships_list = list(Board.ships.keys())
        i = 0
        while i<len(ships_list):
            ship = ships_list[i]
            directionShip = input("Vertical or Horizontal: [V/H]: ")
            coordinateShip = input("Enter {} coordinates:
".format(ship))
            if not self.shipValidation(ship, coordinateShip,
directionShip):
                print('Coordinates in use, please re enter valid
coordinates')
                continue
            row, column = self.shipIndex(coordinateShip)
            if directionShip.upper() == 'V':
                for i in range(row, row+Board.ships[ship]):
                    Board.board[i][column] = ship[0]
            elif directionShip.upper() == 'H':
                for i in range(column, column+Board.ships[ship]):
                    Board.board[row][i] = ship[0]
            i+=1
        return Board.board

    def shipValidation(self, ship, coordinate, direction):
        row, column = self.shipIndex(coordinate)
        if direction.upper() == 'V':
            for i in range(row, row+ships[ship]):
                Board.board[i][column] = ship[0]
                for i in range(row, row + Board.ships[ship]):
                    if Board.board[i][column] != ' ':
                        return False
```

```
        elif direction.upper() == 'H':
            for i in range(column, column+ships[ship]):
                Board.board[row][i] = ship[0]
        return Board.board
            for i in range(column, column+Board.ships[ship]):
                if Board.board[row][i] != ' ':
                    return False
        return True
```

```
obj = Board(10, 10)
board = obj.emptyBoard()
obj.displayBoard(board)
```

This meets the task requirement by carrying out the ship placement validation functionality by the addition of the ship index and ship validation functions and additions made to the update board function addressing vertical or horizontal alignment successfully.

Code review:
Good:

- Overall the added functions carry out the necessary functionality
- Extended if statements validate if the coordinate is not equal to an empty string and in that case i'll return false, meaning if the program acknowledges a ship then it invalidates the entire placement of that ship as it simile detects an overlap.
- Clear prompt for vertical or horizontal alignment that the user selects which is stored in a single variable (directionShip) following the task/ticket designs I initially set. Limits error and validates alignment by accepting both lower case and upper case letters either ‘v’ for vertical or ‘h’ for horizontal.
- Inclusion of shipIndex() function

Possible improvement points:

- Human readability could be improved here for the naming of functions.
  For example shipValidation() within the board class may appear as out of place if reviewed by external developers that are foreign to the project therefore in the case of paired programming or for group purposes I would name this functionality more logically such as boardValidation().

The main change was the addition of the valid ship placement function below which holds the vertical and horizontal alignment of the ship; this was separated from the update board function.

```
def validate_ship_placement(self, x: int, y: int, direction: str, size: int):
      sum_of_free_place = 0
      try:
```

```
            if direction.lower() == "v":  # for vertical alignment of boat
                for index in range(size):
                    if self.board[index+y][x] != ".":
                        sum_of_free_place += 1
            elif direction.lower() == "h":  # for horizontal alignment of boat
                for index in range(size):
                    if self.board[y][index+x] != ".":
                        sum_of_free_place += 1
            return sum_of_free_place == 0
        except IndexError:
            return False
```

In addition to this a release ship position function was added to later assist with hit and miss functionality.

```
def release_ship_position(self, ship: str):
        ship_obj = self.onboard_ships.get(ship)
        x, y, direction = ship_obj.get_xyd
        try:
            if direction.lower() == "v":  # for vertical alignment of boat
                for index in range(ship_obj.size):
                    self.board[index+y][x] = 0
            elif direction.lower() == "h":  # for horizontal alignment of boat
                for index in range(ship_obj.size):
                    self.board[y][index + x] = ship_obj
        except IndexError:
            return
```

This function stores the ship position. If a ship within the coordinates already placed the function removes those coordinates from the empty list of board positions, this updates what positions are now vacant and available for placement of any new ships.

Code review:
Good:

- Overall refactoring of code and addition of functions successfully enables **efficient future scalability**, the method is reusable for applying the later autoplace ship and hit functionalities within my design user requirements.

2. Auto place remaining ships functionality

```
def auto_place_remaining_ship(self):
        for ship, ship_obj in self.onboard_ships.items():
            if ship_obj.x is None:
                y = randint(0, board_size[0])
                x = choice(tuple(self.cols.keys()))
```

```
                direction = choice(("H", "V"))
                while not self.update_board(x, y, direction, ship, True):
                    y = randint(0, board_size[0])
                    x = choice(tuple(self.cols.keys()))
                    direction = choice(("H", "V"))
        return True
```

The main challenge was identifying which ships had already been placed to avoid duplicate ships on the board and using some form or random functionality on only some of the ships.

To fulfil my design plan for the autoplace actions I decided to use the function randint(). I began with some basic experimentation and practice using the function. In my case the randint() randomly selects an integer between the higher limit being the board size and lower limit being zero. This supports the auto place function when allocating valid coordinates for the ship to be placed in based on the remaining coordinates left.

To differentiate between the existing ships on the board and missing ones I included an if statement within the for loop to acknowledge that if that ship object is not present then follow through and randomly generate a position whereas if it is not then return an updated board with the same items.

```
def get_placed_and_not_placed_ships(self) -> tuple:
        """
        Use this method to get the separated list of completed ships placement and
not placed ships list
        :return: tuple of placed and not placed ships
        """
        placed = []
        not_placed = []
        for ship_name, ship_obj in self.onboard_ships.items():
            if ship_obj.x is not None:
                placed.append(ship_name)
            else:
                not_placed.append(ship_name)
        return placed, not_placed
```

3. Remove of ship coordinate once ‘hit’

```
def hit(self, x:str, y:int, auto_hit=False):
        # get the col of the board
        col = self.cols.get(x.upper())
        if self.board[y][col] == "M" and auto_hit:
            return False
        print("Coordinate", f"{x}{y}")
        coordinate_state = self.board[y][col]
        if isinstance(coordinate_state, Ship) and (not
coordinate_state.is_sunken()):
```

```
            coordinate_state.hit()
            self.board[y][col] = "H"
            print("%s%s is a 'Hit' on %s's boat!" % (x, y, self.player))
        else:
            self.board[y][col] = "M"
            print("%s%d is a 'Miss' on %s's boat!" % (x, y, self.player))
        return True

    def get_ship_sunken_status(self):
        return {key: value.is_sunken() for key, value in self.onboard_ships.items()}

    @property
    def board_status(self):
        return not all(boat_obj.is_sunken()==True for boat_obj in
self.onboard_ships.values())
```

In order to clearly remove the ship from the board I included a get sunken ship status function so that once a ship is completely hit and sunken then it will update its states to sunken which enabled me to link through an if statement later on to physically remorse the ship from the users view(documenting as a Hit).

- **Final run for Version 1: player vs Computer**

2.4 Phase 3: Implementing code for version 2 based on the design

These are the main sections of phase 3, I have provided code reviews and further explanation where necessary.

- **Creating initial code for version 2 and modifications in version 1**
- **Validating and testing the code for version 2**
- **Modifications based on errors**
- **Final run for Version - 2**

2.4.1 Creating initial code for version 2 and modifications in version 1

Represented in git commits.

2.4.2 Validating and testing the code for version 2

Followed the same step listed in section 2.3.2 above.

2.4.3 Modifications based on errors

Modifications made based on failed tests to fix error messages within the terminal.

2.4.4 Final run for Version - 2

Main challenges when completing version 2

1. **Implementing the mines exclusive feature functionality (referred to in section 3.2.3)**
2. **Maintaining of Board class**

2.5 Phase 4: Implementing code for version 3 based on the design

These are the main sections of phase 4, I have provided code reviews and further explanation where necessary.

- **Creating initial code for version 3 and modifications in version 2**
- **Validating and testing the code for version 3**
- **Modifications based on errors**
- **Final run for Version - 3**

2.5.1 Creating initial code for version 3 and modifications in version 2

Represented in git commits.

2.5.2 Validating and testing the code for version 3

Followed the same step listed in section 2.3.2 above.

2.5.3 Modifications based on errors

Modifications made based on failed tests to fix error messages within the terminal.

2.5.4 Final run for Version - 3

2.2. Ensuring quality

I referred back to all points stated in my Battleship game plan throughout the implementation process, specifically from section 1.8 in this document.

2.3.1 Testing and resolving bugs

- Code reviews
- Unit testing
- Refactoring
- Regression testing + integration testing once combining the read config file knowledge

Detailed explanation referred to in sections 2.1 through to here.

Main challenges when completing version 3

1. **Implementing the salvo play exclusive feature functionality (referred to in section 3.2.3)**

3. Evaluation

In this evaluation I will assess the quality and effectiveness of my battleship game: I will examine various aspects of the project, including its design, implementation, and performance to ensure that the project meets the desired standards of quality and effectiveness, and to identify any area for improvement.

Typically programming evaluations are conducted by a team of experts of professionals who are familiar with programming best practices and have experience in evaluating similar projects. The goal of the evaluation is to identify any weaknesses or areas of improvement in the project, and provide possible recommendations for theories to address the problem. In my caseI will carry out this review myself.

3.1 Analysis

3.1.1 Refactoring

1. **Before completing Version 1:**

Code review of when I implemented the configfile and refactored all code to correctly implement the singleton design pattern

This meets the add plain text file called ‘adaship_config.ini’ that defines the game’s configuration; this structure requirement is set on my kanban board. The initial git commit phases from 1-3 included a lot of experimentation and practice of controller and action gameplay logic where, hereafter after re-examining the adaship documentation I began to implement better structured commits and implement the practices design patterns within the lectures.

```
Board: 8 x 14
Boat: Battleship, 5
boat: Destroyer, 3
Boat: Submarine, 3
Boat: Patrol Boat, 2
Boat: Submarine, 3
Boat: Exp1, 4
```

Good:

- Overall the use of a config file establishes the parameters(properties) and settings for the games board applications and operating systems. This is crucial to enable me to customise interactions with applications and how the applications interact with the rest of the system.
- Emphasises good coding practice by allowing future management of config files in other environments which may be a future possibility.

2. **Before completing Version 2:** building on version 1s functionality and keeping preparation for version 3 in mind
   Version 2 enabling Player vs Player game menu option

```
ship_obj = self.onboard_ships.get(ship)
        ship_obj.x = col
        ship_obj.y = y
        ship_obj.direction = direction
```

I created an object of a ship which inherits all attributes/properties from the ship class. This enables multiple player logic.

I also extended the set up menu selection.

3. **Before completing Version 3:**
    - **Referred to in section 3.2.3 below.**

3.1.2 Reuse of code

1. **Majority of methods within the Board class** used to execute all additional functionalities, for example calling of the same hit() and ship_validation() functions for the mines feature, salvo feature etc. **See section 3.2.3 for a code example** of the hit function used in the mines class. Although code reuse is expressed as a type of 'code smell' in this case the negatives of code reuse are excluded as the recalled methods carry out the main functionality of the battleship game, therefore making the game more maintainable and scalable between versions 1,2 and 3 following object oriented programming principles.

3.1.3.Code smells

I aimed to limit code smells through my code reviews however arguably in some cases they were unavoidable.

Large board class

Large classes generally translate to bad coding practice as it indicates unorganised coding, by holding several fields and methods it makes the class difficult to understand and maintain. In this case refactoring by splitting up the class is a common solution and aids a better structured code file.

Nevertheless due to the games capacity I decided to keep all methods/fields within the board class that were related directly to the boards functionality and validation, by splitting up the class to further divide the validation functions such as ship validation and functionality functions such as update board I believe would have compromised my codes human readability and reusability which was a high priority to support future scalability of my Battleship game.

Unused code

In between commits and specifically when finishing my final commit to git I ensured to remove any neglected code. Unused code (code no longer called by the codebase) can clutter the codebase and make it more difficult to understand and maintain.

Code Reuse

Explained in section 3.1.2 above with a code example.

3.2 Implementation and effective use of ‘advanced’ programming principles

3.2.1 Algorithms

Programming **algorithms** are responsible for determining the behaviour of the game and actions of the players, therefore making them crucial to my battleship game. There are several important algorithms I implemented in my game which each serve specific purposes.

For example, I used an algorithm to determine the placement of the ships on the game board (code explained in section 2.3.3.1 above). This algorithm was achieved by using a built-in python random integer function to populate the ships within valid coordinates on the board in a balanced placement, while establishing errors to eliminate ship overlaps.

In addition to this other examples of algorithms within my battleship game were, to determine outcomes of each player's attacks (Hit or miss logic), keeping track of the players display game board and indicating a game winner.

Based on the strongly programmed epics and targets that meet these requirements the use of algorithms overall were beneficial to the project and enabled me to learn and experiment different approaches to best implement the functionality needed to simply meet the user requirements and support future game scaling.

3.2.2 Design patterns

The **singleton design pattern**'s purpose is to ensure that a class has only one instance, while simultaneously providing a global access point to this instance. My Battleship game follows an object oriented programming design and the addition of the singleton gave several benefits.

1. Allowed me to ensure that all objects eg. the ship had access to the same instance of the particular board class. Through a global point of access, this aided in easy management of the communications between objects and prevention in the creation of unnecessary copies.
2. Reduced **overhead** associated with the creation of multiple instances of a class. In my case resources were limited therefore building multiple instances could have resulted in significant memory consumption. This would overall reduce game performance especially if scaled in the future.

I achieved the task of embedding this design pattern through my code readability and clarity from my code reviews. I am happy with the singletons outcome and contribution to the games programme.

3.2.3 Exclusive features

1. **Improved game play “Salvo” variation.**

The salvo implementation updates the basic game play by allowing the current ‘player’ to ‘fire’ one torpedo per their remaining ships. Once entered, details on any ‘hits’ and/or ‘misses’ are clearly provided, and all appropriate boards are updated to reflect this salvo.

This addition was made to meet the salvo exclusive feature. The following code shows my approach to this including the function play_salvo() in the players class.

```
   def play_salvo(self):
       print("%s's chance - " % f"Player_{self.player_id}")
       self.show_board()
       live_boats = self.board.get_live_boats()
       coordinates = input("Enter the coordinates as salvo to hit E.g.: F3 G1:
").strip().upper().split()
       if not coordinates:
           print("Do you want to quit game?")
           selection = input("(y/n): ").strip()
           if selection[:1] == "y":
               print("Quiting the game.")
               return
```

From the player's point of view the user is prompted to enter the coordinate values of where they want the torpedo to ‘hit’, this data is stored in a tuple data structure.

The function still supports the quit game functionality previously required for the other game versions.

```
   def play_salvo(self):
       print("\nComputer_%s's Chance" % str(self.player_id))
       self.show_board()
       live_boats = self.board.get_live_boats()
       for coordinate in range(len(live_boats)):
           x = choice(tuple(Board.cols.keys()))
           y = randint(0, board_size[0] - 1)
           hit_status = self.opponent.board.hit(x, y, True)
           if hit_status == "exists":
               continue
           elif hit_status == "Blast":
               print("%s%s is a 'Blast' of Mine" % (x, y))

           while not hit_status:
               x = choice(tuple(Board.cols.keys()))
               y = randint(0, board_size[0] - 1)
               hit_status = self.opponent.board.hit(x, y, True)
           else:
               print("%s%s is a '%s' on %s's boat!" % (x, y, hit_status,
self.opponent))
       input('Press any key to continue: \n')
```

```
        print("--" * board_size[0])
        return True
```

The program then checks for all the mentioned coordinates and returns ‘M’miss if nothing is hit and ‘H’ if a ship is identified and hit. A while loop is then run, referring back to the hit method previously created within the board class. As the board class mimics the singleton design pattern, where only one instance of the class is created (object), therefore the line `hit_status = self.opponent.board.hit(x, y, True)`
Is called one by one opponent then board then to the boards method hit() as it follows a singleton design method so must be called as a chain.

Although the singleton design pattern provides several advantages that adhere to good coding standards stated in section 3.2.2 above, it arguably makes maintaining the pattern, in the event that I need to scale the game to support two objects/instances of the board or of any class I am restricted.

Hidden Mines

The hidden mines implementation updates the basic game play with five randomly dropped mines. The hidden mines are added to each ‘players’ board and remain ‘hidden’ during the set-up phase. If an opponent’s torpedo ‘hits’ a hidden mine at that location and the eight immediately surrounding it ‘explode’.

I made this edition of the plant_mines function for the board class, it contains a while loop calling the same random integer generator function that I successfully used for the auto placing functionality and uses the same logic to randomly place 5 mines. This is a great example of code reuse to implement the exclusive functionality rather than using a completely different method.

```
def plant_mines(self):
        from mine import Mine
        count = 0
        while count < 5:
            y = randint(0, board_size[0]-1)
            x = choice(tuple(self.cols.keys()))
            col = self.cols.get(x)
            self.board[y][col] = Mine(x, y, self)
            count+=1
```

Following that I created a Mines class (below) taking advantage of the object oriented programming format, as the mine is a different entity carrying different properties and behaviours of hitting 8 surrounding coordinates. This class follows the same design pattern and principles covered in section 3.2.3 above.

The major difference with mine's implementation is I wanted to refrain from confusing the user and improve the user interface slightly. Instead of the mines being symbolised as an ‘M’ similarly to the symbol for missed hit signifier, I used ‘%’ as it clearly stands out from all over symbols in use already.

```
class Mine:
```

```
def __init__(self, x: str, y: int, board: Board):
    self.col = x
    self.x = Board.cols.get(x.upper())
    self.y = y
    if isinstance(board, Board):
        self.board = board
    else:
        raise ValueError("Cannot connect to board")
    print(y, x)
    if isinstance(self.board.board[self.y][self.x], Ship):
        # name the mine to the name of the ship on which the mine is set
        self.name = self.board.board[self.y][self.x].name[:1].lower()

        # hold the ship object in ship_obj to reference to it later
        self.ship_obj = self.board.board[self.y][self.x]
        self.board.board[self.y][self.x] = self
        self.is_ship_and_mine = True
    else:
        self.name = "%"  # this is how mines are represented on the board
        self.is_ship_and_mine = False

def blast(self):
    if self.is_ship_and_mine:
        self.ship_obj.hit()
        self.board.board[self.y][self.x] = "H"

    # hit the neighbors 8 coordinates of the board
    # first Column
    try:
        self.board.hit(tuple(Board.cols.keys())[self.x - 1], self.y - 1)
    except IndexError:
        pass
    try:
        self.board.hit(tuple(Board.cols.keys())[self.x], self.y - 1)
    except IndexError:
        pass
    try:
        self.board.hit(tuple(Board.cols.keys())[self.x + 1], self.y - 1)
    except IndexError:
        pass

    # second Column
    try:
        self.board.hit(tuple(Board.cols.keys())[self.x - 1], self.y)
    except IndexError:
```

```
            pass
        try:
            self.board.hit(tuple(Board.cols.keys())[self.x + 1], self.y)
        except IndexError:
            pass

        # third Column
        try:
            self.board.hit(tuple(Board.cols.keys())[self.x - 1], self.y + 1)
        except IndexError:
            pass
        try:
            self.board.hit(tuple(Board.cols.keys())[self.x], self.y + 1)
        except IndexError:
            pass
        try:
            self.board.hit(tuple(Board.cols.keys())[self.x + 1], self.y + 1)
        except IndexError:
            pass

    def __str__(self):
        return self.name

    def __repr__(self):
        return self.name
```

3.2.3 Reflective review

In conclusion I am very proud of my Battleship game and have successfully programmed a refined high-level game that exercises a range of intricate logic and all complex functionalities and even exclusive game features. This assignment provided clear direction which allowed me to carefully research and render a systematic easy to understand design plan and approach to render my own version of the game.

Generally I am most proud of the algorithms I developed for the ship auto placement, the determining outcomes of each player's attacks (Hit or miss logic), keeping track of the players display game board and indicating a game winner. Although the solutions I built took a few iterations to develop and implement, it enabled me to learn a lot about the basis of action and controller code which I can now add to my skills and utilise in future projects where appropriate.

Although I found the entire experimentation aspect of this assignment hugely beneficial after reviewing my final Battleship product I would have liked to have focused a sizable amount on improving the user interface and overall user experience. I would have achieved this by importing a few libraries and possibly incorporating basic html and css to add come buttons(for ‘reset’ and ‘quit’’)

and message boxes such as ‘player turn: ‘ or ‘WINNER!’etc, these additions would have increased the games visual appeal and aligned with certain user requirements.

More specifically reviewing my codebase, I kept well organised files throughout my development process, adhered to initially set coding standards to mitigate code smells, code readability, maintainability with clear comments etc and usability. Moreover as for improvement opportunities within my codebase: I should have remained mindful of the scalability and reusability aspects of this project therefore should have stuck to underscore naming conventions when it came to functions. Typically when deploying smaller projects I prefer to use capitalisation however I find it confusing to follow when working with larger programs, I anticipate longer function names to keep them logical and human readable/easy to follow. In addition to this, be more weary of the time constraint of the project and follow my timeline more closely. In this event I was also sick for a time period of this project development therefore hindered my initial progress but planning for these possibilities may have reduced stress.

Finally, remaining consistent with making commits to my git repository, as this project included a lot of research, experimentation and trials I became very accustomed to cloning the file while making changes then forgetting to go back and commit the previous working version to my git repository. This is simply due to me being a little too fixated on closing out tickets and moving on over keeping a steady backlog and updated documentation. So to regulate this I went backwards to commit my code to my git while completing the documentation for that iteration of the project then repeating the process to commit the next commit etc.

Taking these self acknowledged improvement points into consideration, I am overall extremely satisfied with my final game product and feel that I have improved my programming skills as a result.