Battleship NEA

Bilal Nawazani

Table of Contents

Overview of the Problem

Battleship is a world renown board game which consists of 2 players placing a fleet of ships of different shapes and sizes in secret from each other, the ships are located at different points on a 10x10 grid, and the 2 players have to outsmart each other through some psychological mental warfare. Once all the ships are placed the 2 players play in a turn-by-turn basis in which each turn usually consists of 1 shot, if the shot is successful the player gets to shoot again at the opponent's grid until they miss, once the player has missed the game is switched over to the opponent, and the previous players grid where they repeat the same process until they miss, once all the ships are eliminated the game is over and the player who sunk their foes ship is deemed the winner.

AI has changed the fascination with the game completely with the introduction of the first digital battleship game in the 1970s which featured a simple AI algorithm to play against the user. Modern battleship AI models feature machine learning and highly complex understanding of strategies of the game, these have taken the immersion of playing with AI to new heights as the incorporation of efficient search techniques and ship finding algorithms have changed the way the game is played completely. The presence of highly challenging AI within the game have forced players to adapt and come up with new strategies frequently, which has made the popularity of the game reach a new level of interest.

Historical Demographic

The early years of battleship 1930-1950: had a larger presence of children and early-teens, more men than women would be found in the war-themed games.

The initial take off in popularity started from the 1960-1980: when the commercialization of the game began, the demographic grew from just children and early-teens to families and the game had become a household name.

When taking a look at the modern state of the game from the 1990s-Present: we can see that the demographic had garnered the attention of adults, likely due to the aging of children who played to game who are now adults playing the game for the nostalgic factor, the gender bias of the game had become more balanced with an equal number of female interest in the game as male, and the game had made its way out of just Europe and North America to certain parts of Asia

How Could the Computational Method Be Used?

These are the properties of battleship that allows it to be produced using the computational method:

- **Finite ship configuration**: Due to the relatively small grid and the fixed sizes/shapes of the ships there is not a large number of complex ways the ships could be placed.
- **Fixed grid**: A game of battleship is always played on a 10x10 grid which makes the design of the game relatively simple to create.
- **Predictable Outcome**: During a game of battleship there are two input of a shot or power-up, and the outputs that can be provided from these inputs is either a hit or a miss, this predictability allows for precise blueprinting on the game's functionalities.
- **Implementation of AI**: The approach to the addition of AI could initially be randomized, but switched to a more targeted approach when it hits a ship this allows for an effective way to keep the game interesting and fair to the player.

These are the ways the computational method could be used to improve the digital version of battleship over its physical predecessors:

- **Abstraction:** Abstraction can be used in the digital version of battleship to remove unnecessary clutter and excessive detail from the physical version which makes it easier to design the functionalities of the project. The core components of battleship could be explored by separating them into their own classes and giving them their necessary attributes, this clear representation of the game mechanic makes it easier to create components of the final product.
- **Decomposition:** Decomposition could be used to divide the development of the game into a multi-stage step-by-step process such as the initialization of the game, gameplay loop and the AI functionality. This clear separation makes the debugging and testing of the game mechanics easier as well as allow the maintainability of the game to be clearer and more concise. Decomposition could also be extended objects and functionalities of the game example the ship layout, the event of a hit/miss and power-up usage.
- **Divide and Conquer:** With divide and conquer one main problem such as where to shoot next can be divided into 2 sub-programs, which are whether the target has missed in which case another random shot is executed or in the case of a ship has been hit, then a searching algorithm is run to find the remaining parts of the ship and destroy it. Examples of where divide or conquer could be used:
    - **Ship Placement:** generation of the ship, the placement and orientation of the ship followed by the action of placing the ship on the grid.
    - **Hit detection:** deduce whether a ship has been hit, the following actions could result in a ship sinking or updating the ships damage status.
    - **AI Guessing strategy:** the actions following after a ship has been hit or a miss, in the case of a hit the entire ship is destroyed, if not the AI can continue randomly shooting.

Appeal to Stakeholders

This timeless simplistic game has a large number of notable stakeholders such as the popular conglomerate hashbrown which continuously looks for new ways to keep the game fresh and interesting for the modern audience, stakeholders can be found on a more personal level, such as teachers and fellow students which have shown interest in the project and the final product, the AI components of this project have largely caught the attention of my stakeholders and the competitive gaming community are always interested in experiencing extensions of a product they're used to.

Identification of Stakeholder Needs

The main method used to collect information on the requirements from stakeholders for the game were surveys using survey monkey.

The questions asked were:

Q1: Do you enjoy turn-based board games?

Q2: What feature are you most interested in seeing in battleship?

Q3: Is there any way you want the game to deviate from the classic version of the board game?

Q4: What power-ups would you like to see and why?

Q5: What level of graphics quality do you expect?

Q6: How would you like to see the AI to be implemented in the game?

Responses Received:

Name	Q1	Q2	Q3	Q4	Q5	Q6
Jack	Yes	The AI	Yes, I would like to see multiple unique takes on the classic battleship game	Radar, to add a new element to the game	Low – simple and accessible graphics	It should be complex/smart, and must not have a predictable shooting pattern
Lily	Yes	Ship placement mechanic.	I would like a mode in which the ships positions are continuously randomize whenever the players turn is over	A large bomb which shoots 4 squares of the players choice, it speeds up the game a bit more	Low – simple and accessible graphics	I want a certain amount of complexity to it, but don't make it too difficult
Omar	No	The AI	Keep it simple	Radars would keep the game interesting	Medium – good quality but not too demanding	Have an AI player that I can face against

Oscar	Yes	The AI	Yes, I would enjoy another mode that allows you to keep shooting until you miss	Radar, it adds to the tension and makes the game seem less repetitive	Low – simple and accessible graphics	I want it to be fair and resemble mannerisms of a real human

Qualitative Breakdown of the Responses

Based on the ratio of people that enjoy the battleship game I have discerned that a battleship game is still to this day a very engaging experience and this idea has fortified my desire to go through with the project. Below is my understanding of the responses received by the stakeholders and the ideas I've gotten on developing this game from them:

- **Jack:** Jack wants a new take of the classic battleship genre which requires me to think out of the box and make a mode which is still familiar and fair yet different from the classic battleship genre. His interest in AI demands a requirement of attention to detail when developing the AI, the mention of the AI not having a predictable shooting pattern could be overcome by having an element of randomness to the AI's shooting pattern.

- **Lily:** The interest of ship placement mechanics from lily could spark a lot of functions to the ships such rotational movement and inability of overlap, this portion of development currently seems to be the most difficult concept to add as the ships have multiple rules that need to be followed to make the game play smoothly, the mode suggestion of lily is not possible as a change in the ship positions of the ship would interfere with the positions already shot on the grid and this could lead to the players ships being hit even though they were initially missed, this addition seems unfair and will not be implemented in my development phase of the game. However, a bomb seems like an interesting suggestion but, this mechanic should be implemented carefully as it could ruin the game if it can be used whenever the players want. The contrast in stakeholder AI responses on the difficulty of the game will be solved with different difficulties to the game.

- **Omar:** With the popular power-up suggestion of the radar scanner I will place priority on developing this when making the first iteration of my game. Omar's requirement of medium graphics will be met with the inclusion of several image files to make the game even more immersive.

- **Oscar:** Oscar's suggestion of a mode where you can shoot until you miss is how most modern battleship games are played whilst a turn-by-turn shooting mechanic has a resemblance to a more vintage version of the game. When Oscar mentions 'resemble mannerisms of a real human' this furthermore hint at my stakeholders wanting to AI to shoot in a random pattern as most new battleship players have that approach until they get a hit.

The stakeholders seem to have a larger interest in the functionality of the game rather than the form so the graphics will not be a priority to me.

Roles of the Stakeholders

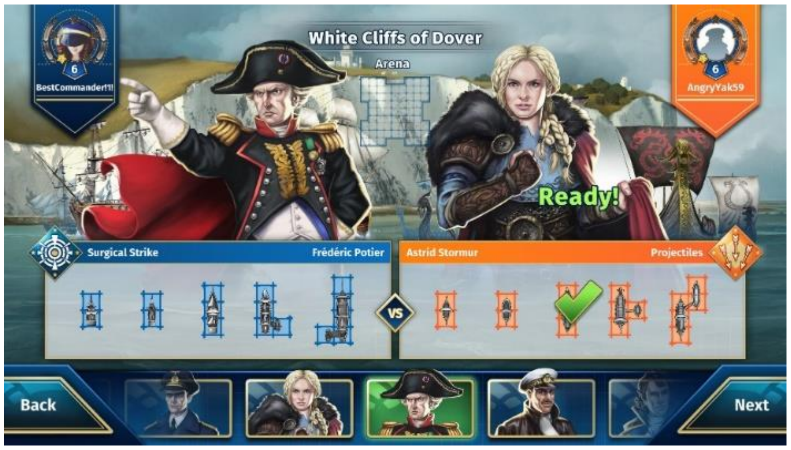
- **Jack** - Subject Teacher/Project Manager
    - Jack's guidance as project manager has allowed me fill in the academic criteria's set out for the project, such as structuring the algorithms to have elements of complexity within them. One of Jack's roles is to overview the progress of my project and offer support to make the best possible final product.
- **Lily –** Intermediate Back-end Developer
    - Lily's contribution to my project found is the debugging and correcting errors in the development phase of my game, with her unique approach and attention to detail on this project I was able to add distinctive features which enhance the functionality and overall experience of playing the game.
- **Omar** – Front-end Developer
    - With Omar's technical background on developing the user-interface and intractability of many games and simulations that span across multiple visually appealing projects, I have been able to create a layout for the UI of my battleship game following the directions and expertise of Omar who has provided useful feedback through the development of the main menu, ship design and grid layout.
- **Oscar** – Experienced Game Tester
    - Oscar has played a multitude of games during his youth, this experience in the game industry has made him develop a critical eye when it comes to the functionalities of games, the main role of Oscar during the progressive development of my project is to identify bugs and improvements that may elevate the experience of playing the game as well as provide ideas for additional features that may engage the user into playing the game for longer periods of time without getting bored.

Research and Existing Solutions

Hasbro's Battleship:

Hasbro's Battleship is the most modern and complex iteration of the battleship game, the game features a variety of characters with different power-ups and abilities and the UI is incredibly visually appealing with each ship having high quality textures, the ships used are standard, the normal game mode of this game does not follow the classic rules as when a ship is hit the player is continually allowed to shoot until the miss or sink the ship. Hasbro's battleship features the largest amount of game modes out of every battleship game, which keeps the game engaging and immersive.

Commander Selection

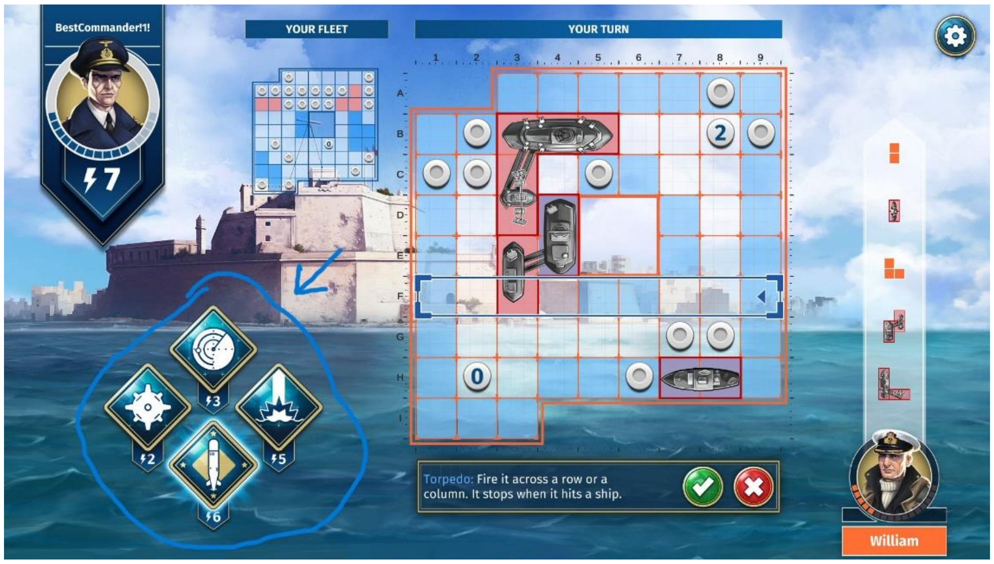


The game starts off with a commander selection menu, each commander grants you different abilities and ships based on the commander you choose to play. The abilities have a cooldown which lasts a certain number of rounds to keep the game fair and add a strategic component to the game. The abilities can either be offensive or defensive depending on the player you pick which adds to the depth of the commanders and introduces multiple playstyles depending on the chosen commander.

Ship Structure

From the image above it can see that different commanders have different shapes and sizes to their ships which can create unique placement opportunities and requires the player to spend time learning and gaining familiarity with each character's ships, this feature could spark a competitive area of the game has it requires a certain amount of skill to deduce the ways a player could place their ships depending on their chosen commander.

Ability Incorporation



Torpedo: Fire it across a row or a column. It stops when it hits a ship.

The most intriguing element of Hasbro's battleship is the seem-less integration of abilities based on which commander you choose to play as; these abilities can range from the common sonar pulse to the unusual but fantastic mines which are placed on the players grid and if the opponent hits it a random area on the opponent's grid is fired at, this ability adds a level of consequence to the plays you make on your opponent's grid and makes you think carefully before you take a shot. Some other abilities found in the game which I may take inspiration from in the development phase of my project are:

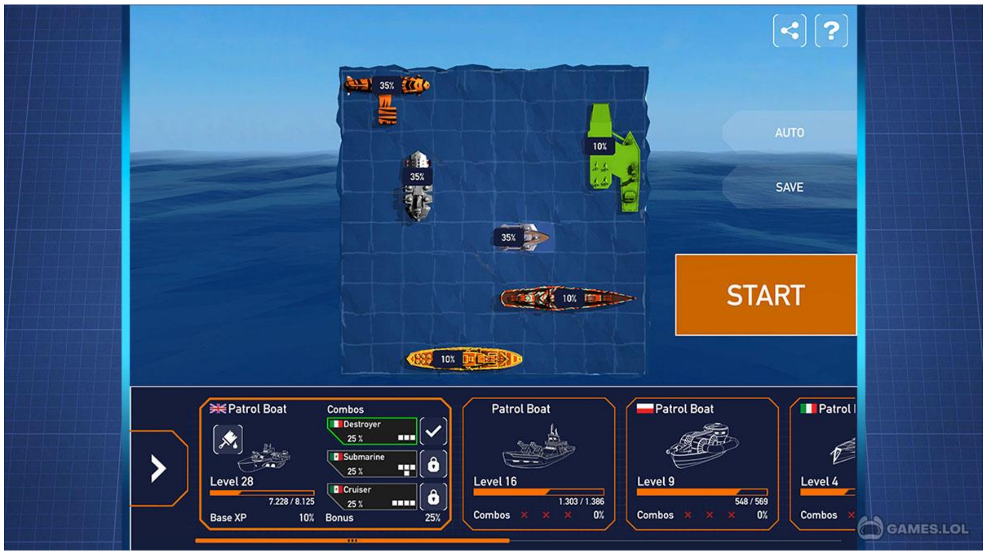
- **Airstrikes**: Targets multiple squares on the opponent's grid which can eliminate entire ships if the shot is successful, if not it can eliminate a multitude of potential squares in which the ships could be placed.
- **Decoy ships**: These ships can allure the opponent into wasting rounds and precious shots on a fake ship which takes the shape of a real ship but ultimately is revealed to be a fake.
- **Torpedo**: A plane is deployed which fires across a single column/row, this ability has a large cooldown as if the attack is successful, it can deal a significant amount of damage onto the opponent.
- **Repair crews**: Whenever a ship is damaged the repair crew powerup can be used to restore the ship to its original state, this power up can allow the player to buy time and sniff out the opponent's ship before their ship is destroyed and the opponent moves on to discover another ship.
- **Commander abilities:** These are unique abilities associated with the commander you choose to play as; these abilities keep the game less repetitive and allow players to adapt to certain playstyles to gain a competitive edge.

Hasbro's battleship was developed using C# as it was paired with unity, this was likely done for the visual link with functionality that unity provides, however for my solution I will be using python as my take on the game will not feature a large amount of significance to the user interface and focus strongly on the functionality.

Hasbro's Battleship AI

Hasbro's AI initially attacks randomly, as the game progresses the AI picks up on the players shooting patterns and adjusts the way it places its ships accordingly, once the AI finds a ship it runs a probability algorithm based on the previous games it's played to find out the remaining parts of the ship and eliminate it. This is a common integration of AI in the realm of battleship and will be used in my project, the actual algorithms are kept secret since the game is closed-source so I'll make a few adjustments to the algorithms to better suit my version of the game.

Fleet Battle – Sea Battle:



Smuttlewerk Interactive’s fleet battle is primarily a mobile game which runs on the IOS/Android platform, the game features a different layout of ships based on the country you choose to play as, each country has a different layout of ships which can be transferred to different countries once you get the blueprint, this mechanic allows you to be able to customize your ship layout.

Combo Mechanic

A combo is can be acquired through 2 possible means, one is leveling up which unlocks new combos or allows you to improve existing ones, the second way of getting combos is by reaching milestones or doing achievements, milestones such as winning a certain number of matches unlocks certain combos, combos that are unlocked by milestones cannot be acquired through leveling up and vice-versa.

Combos don’t differ too much from the feature in Hasbro’s battleship however there is 1 key difference and that is upgrading mechanic which can increase the area the combo attacks, reduces the cooldown before it can be used again and increase accuracy and precision which can completely change the way certain combos are used. This upgrading mechanic is unique to fleet battle, it increases the complexity of the game mechanics and allows for a more compelling experience.

Influence of the Research on My Project

With the research I’ve gathered from these 2 different versions that digitalize battleship, I’ve realized the main mechanics required to implement into my game, these functionalities are the ones that are present in both of them such as a 10x10 grid, different game modes, power-ups, an interactive UI and a different layout of ships based on who or what you choose to play as. These features will be in the base template of my game and other unique features will be implemented later on.

As for how I would integrate these features into my version of the game, I can use certain techniques that are commonly used by these 2 games, such as:

- **Modularity**: The code is broken down and organized into smaller manageable blocks of code that perform a certain function. These sub-routines could be helpful as they could allow the use of recursion when trying to display a certain function on the canvas. This reduces the amount of repetition in code and this makes the code much tidier for future maintenance.
- **OOP** (Object-oriented programming): OOP can be used to initialize game object which can be worked with and changed. Each class deals with a various game object such as the grid, AI, player or ship these classes reduce the overlap in code that would be present if the components of the code were not separated using OOP.
- **Methodology**: The development methodology used in my project is RAD (rapid application development), I have chosen this because I will create multiple different prototypes for my project, each prototype will be assessed by the stakeholders and further amendments/additions would be present in the next version of the game.

Limitations

The biggest limitation to my battleship is integrating all the features I want into the game without making it seem cluttered, due to this limitation I had to sacrifice some of the features I wanted to include to have a more polished final product. The game cannot be perfected to Hasbro's standard of the game due to time and budget restrictions. The UI would not be up to my standard as I don't specialize in front-end development, this hindrance could be overcome to a certain extent by me learning the mechanism behind sprites using pygame. All these limitations could be improved upon in future modifications of the game, but for the time being I will have to do without them.

Game Requirements

Required Features in the first prototype:

- 10x10 Game grid
- Ship intractability
- Snapping ships onto the grid
- Default ship position when not on the designated grid
- Randomizable ships on both the player and opponent grid
- Ship rotational property
- Ships overlapping handler
- Shooting mechanic
- Visual indicator of hit, missed and currently not hit or missed spaces
- Queue to start and end the game through buttons
- Inability to the ship position when the game starts
- Buttons before the start of the game:
    - **Deployment** – to signal the start of the game
    - **Randomize** – to randomize all ships
    - **Reset** – Reset ship positions
- Buttons when the game is in progress:
    - **Redeployment** – restart the game and break out of the playing phase
    - **Quit** – close the application
    - **Radar Scan** – Scan the opponent grid for a ship

- Hidden opponent grid
- Radar power-up that provides a visual on found ships
- Simple AI which randomly shoots on the grid
- Indicator of who won once all the ships are destroyed
- Hidden enemy grid

Additional features in the second prototype:

- Fix bugs from the first prototype
- 2x2 Explosion allowed once throughout the game
- Restriction on radar usage throughout the game
- Player versus player mode addition
- Hard mode addition
- Shoot until miss mode
- Title screen to choose the desired game-mode
- Sound effects
- Polished user UI
- Limiting ability usage to once every 4 rounds

Hardware Requirements

As the game is not too demanding the hardware requirements to run it will not be too costly. Below this is the minimum hardware requirements to run this game without any issue:

- **Processor:** A mid-level Ryzen 3 or I3 processor chip
- **Graphics card:** Nvidia's GTX 580 and up or Radeon HD 6000 series and up
- **RAM:** 4GB DDR4 and up
- **Storage:** 50 Megabytes on a hard drive
- **OS:** Windows XP, Linux 2.0, MAC OS 10 and up

Software Requirements

Below are the software requirements to run the battleship game successfully on a system:

- **Device Management:** allows the mouse to communicate with the game through a series of inputs and provides an output onto the monitor and speaker.
- **IDE:** Have a working IDE that can debug and run the code on the system
- **Python interpreter:** Used to process the code line-by-line looking for errors and produce a working output onto the system. Handles the functions and intractability between the code and the user.

These software attributes must be present to turn the code into a working game.

Functional requirements

Functional requirements are interactable components of the battleship game, these functional requirements are as follows:

Functionality	**What it does**
Ship placement	Used to move and snap the ship onto the grid, checks for criteria's that need to be fulfilled to place the ships such as making sure they don't overlap.
Shooting mechanic	Shoots at a position chosen by the player and the AI, used to destroy the ships and progress the game.
Randomize ship positions	Random positions are chosen on the grid where the ships are placed and do not overlap.
Reset ships	Moves the player's ships from the grid onto their default position.
Game start initializer	Prevents the player from changing the ships positions mid-game and enables the shooting mechanic of the game.
Quit game	Closes the game software.
End game/ restart	Returns the ships to their default position and clears the grid of the explosions.
Radar Scan button	Triggers the radar scanner which is used to locate a single ship.
Ship rotation	Rotates the ships by 90 degrees when being placed
Status of ships	Used to update the status of a ship in case it is destroyed or shot at.

Non-functional requirements

Non-functional requirements are visual/audio mechanics used to allow the user to understand the current status of the game better. These do not serve any purpose except for informing the user on the status of the game:

Non-functionality	**What it does**
Title screen	Displays the game-modes for the player to choose.
Sound after shooting	Acts as a back-up to the ship explosion in case it fails or provides a wrong output, it also acts as an immersive feature.
Ship explosion	Provides a visual indicator to the user on the status of the ships.
Winner text	Informs the user on who won the game.

Success Criteria

Classification	**Success criteria**	**Criteria provider**
Robustness	The functions of the game must complement and work with each other.	Stakeholders
Robustness	The game must run without giving any errors.	Stakeholders
Usability	The interactable UI must be user-friendly.	Stakeholders
Usability	The desired output must be received when a user provides an input.	Research
Usability	Visual indicators must be clear and understandable.	Research
Functionality	When the game ends a clear winner and loser must be provided.	Research
Functionality	The AI must not be predictable.	Stakeholders
Functionality	The ship placement must only be limited to the grid.	Research
Functionality	The AI must have some level of difficulty.	Stakeholders

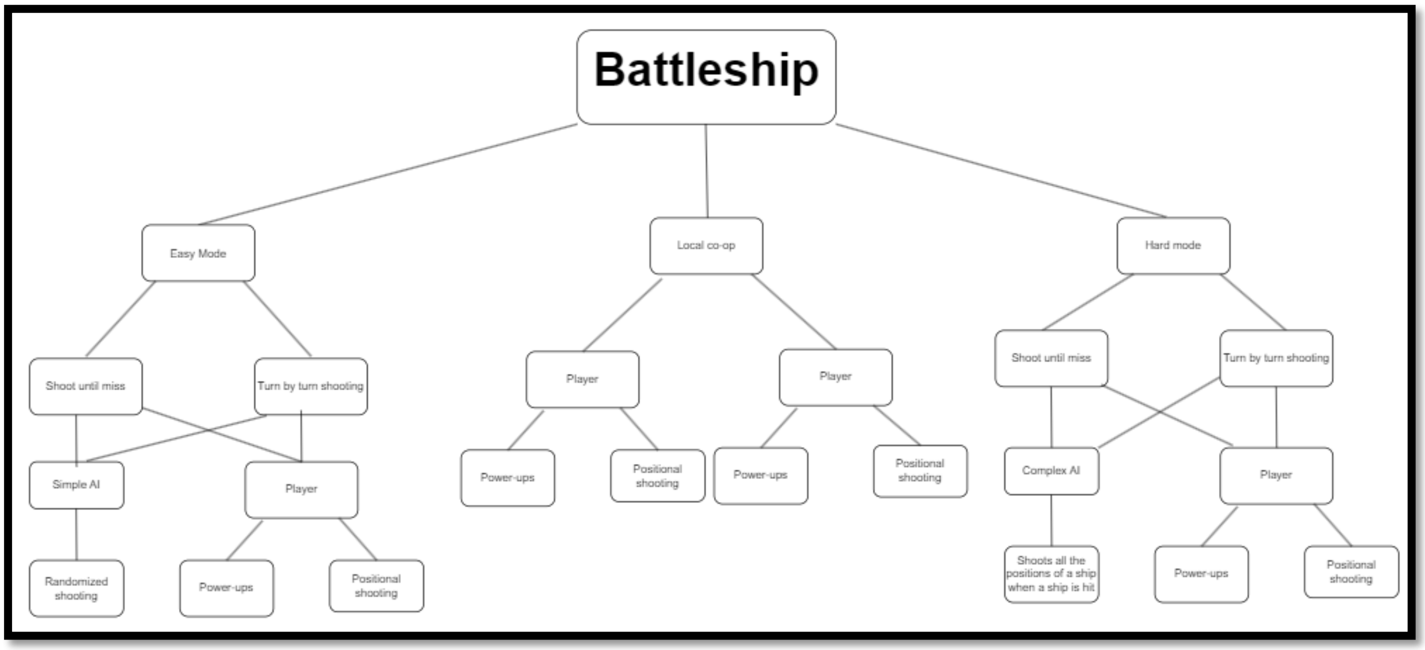
Design

Applying Decomposition to the Problem

I'll show how I broke down each object into developing simpler functions for it delete this bit.

To the solve the computational problem of making a battleship game, I will initially breakdown the entities that need to be present in the game into a series of classes which will be further broken down into attributes and functions through a sequence of flowcharts and other diagrams, these flowcharts will be used to simplify the problem and create a larger understanding of what needs to be developed. The main file will be broken down as a class to keep a consistent architecture to the problem which will be easier to understand.

The way the game will be broken down will be in a progressive manner, which means initially the larger entity will be broken down into a series of larger entities and these entities will be consecutively broken until they are much smaller and understandable.



- ❖ The diagram shown above present the 3 modes my game will have and how they'll differ from each other and what they have in common.

Structure of the Methodology

According the RAD methodology used in my project my prototype version once developed will be given to the stakeholders to gather feedback and this feedback will be used to create the next version until the users are satisfied with the development of the game. This methodology was chosen over the other methodology due to its brisk development phase as I have a limited amount of time to develop this project. The first prototype will create a strong base to the project which will make it much easier to build onto the project as most of the functions and objects will be created in the first iteration, the second iteration will require me to slightly amend the first iteration of the game and change pre-existing functions to create the essential features requested by the stakeholders (that are not present in the first iteration).

Approach to the Algorithms

A 10x10 grid will be made that will be output to the console initially for reference to what's happening to the ships and will then be output to the game screen, this grid will be searched using a linear searching algorithm and will have empty spacing for the cells which don't contain a ship and are not hit, O for ships that are in a cell but not hit, T for hit ship and X for missed cells, these missed cells will originally be empty spaces. The reason I have chosen these letters is because they are very different from each other and will not lead to confusion on the status of a cell block.

Due to the presence of ship objects present at different portions of the grid and an AI I have foreseen the presence of multiple searching algorithms, the most probable searching algorithm to be used would a linear searching algorithm as all the rows and columns on a grid must be searched to find ships. Below are the algorithms to be expected in the project in each of the modes:

Local co-op: With the co-op mode of the game, when a cell block on a grid is shot at the status updates, a cell block that has been previously been shot at will not be able to be shot at again, this is to prevent the players from wasting their turn, all the ships will be hidden before the game starts and must be randomized to prevent the other player from knowing the locations of the ships on the opponent's grid. Once a player shoots their turn will end and they will not be able to shoot again (if it's the turn-based mode), a button will need to be pressed to allow the opponent to shoot the players grid, since both the grids will be hidden, the players do not need to look away.

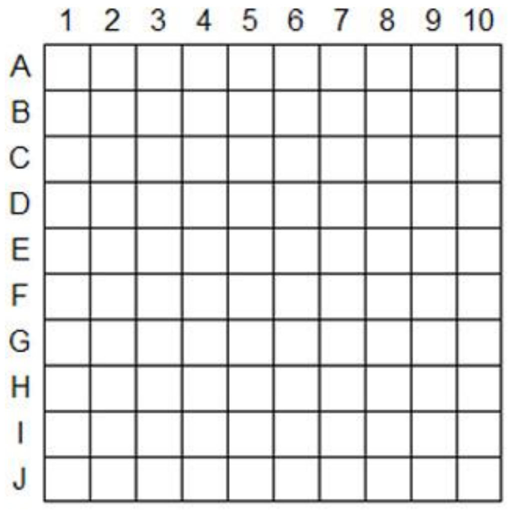
Easy mode: The easy computer will look for a random position on the 10x10 grid, created initially and check the status of the cell so that it doesn't shoot at the cells which are already hit, once a random position on the grid is chosen the grid is shot at and the turn ends (if it's the turn-based mode) and the grid updates the status of the grid that has been hit, this update will then be seen on the game grid.

Hard mode: The hard computer picks a random position on the grid same as the 10x10, but the difference is once hard computer hits a ship it will use a searching algorithm to search the 4 cell blocks surrounding it, the cell block that contains a ship will be shot at, this process will continue until the entire ship that has been found is destroyed.

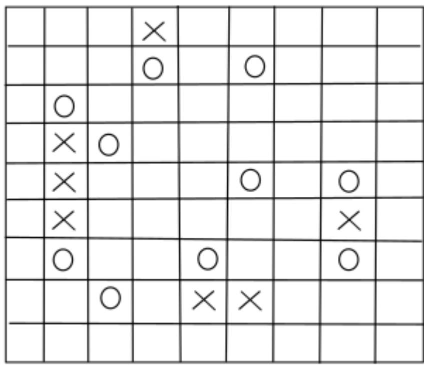
Describing the solution

The solution of the battleship game will require the algorithms below:

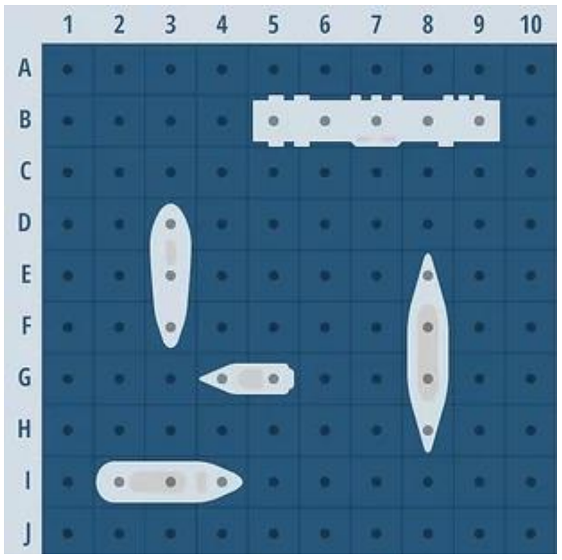
- To create the game grid, you will need to scale it to the screen size so that 2 grids, the buttons and ships can fit on the screen with sufficient space, to create the grid the rows, cols and the cell size must be defined, so that individual cells can be created according to the number of rows and columns and to allow the manipulation (placing ships and shooting) of the cells when running the game.



- The game logic would be required to get the status of the ships, the grid above will be drawn from the game logic grid, this grid will determine which cell has been hit, missed, contains a ship and is empty, based on this information the grid will be updated and the winner, loser and destroyed will be determined.
- The shooting algorithm will use this game logic alongside the mouse position that matches the cell block location to update the game grid with the new shot.



- Randomizing the ship positions, this algorithm will run multiple checks to find a valid position to place the ships, these checks will include determining whether the ships are on the grid, overlapping or rotated, based on these factors a ship will be placed on the grid, the random positions will only be valid until all these criteria are obeyed.
- Another randomized algorithm would be the computer shooting algorithm, this algorithm will determine a valid position to shoot at by comparing the game logic with certain criteria's that are required to be met for a shooting position to be valid.



- Determining whose turn, it is, this algorithm will be based on the status of a series of Boolean values, these Boolean values will control who will be able to shoot at the grid, if a player is able to shoot the opponents Boolean value will signal false to prevent both players from shooting at the same time.

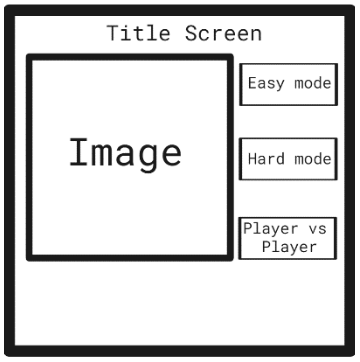
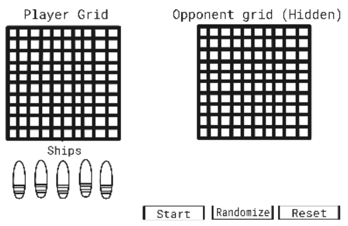
- Building the ship objects and functions, once the ships attributes are initialized, they utilized with a series of functions:
- An algorithm to check whether a ship is on the grid will carry of a series of calculations and set conditions which will compare the position of the ships with the grid, the ship must complement these calculations in order to be placed.
- To snap the ship onto the grid the position of the ship must be compared to the cell block positions and where they are determined to collide is where the ship must be placed.
- To rotate the ships the images must be changed and the calculations to snap onto the cell block will differ from its default rotation counterpart.

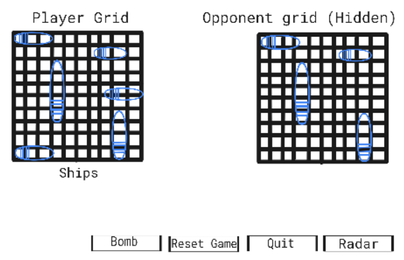
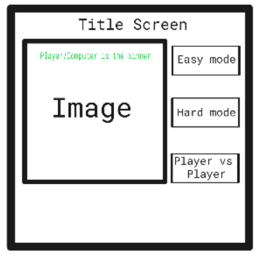
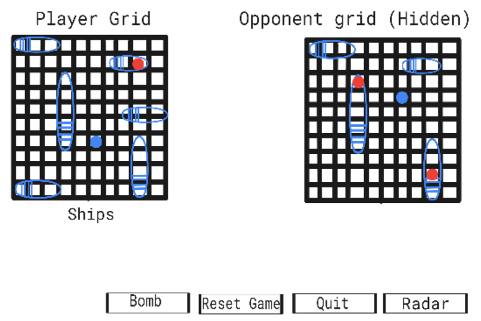
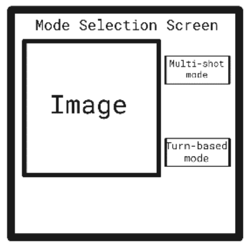
- There must be different buttons present before the game has started and after the game has ended, these buttons will be used for the following functionality
    - Starting the game: Initializing the game.
    - Ending the game: Resetting the grids.
    - Power-ups: Radar & Explosives
    - Quitting: Closing the application.

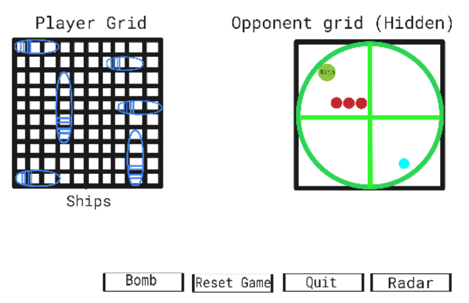
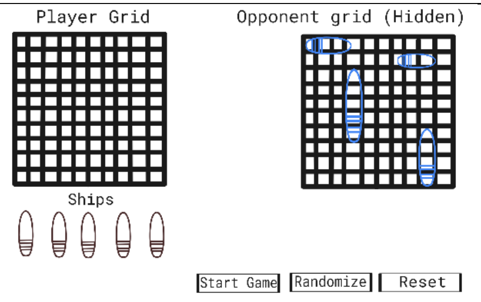
- The explosion of the ships is required to provide a visual indicator of the progression of the game, these explosions will be updated via a large number of images that are looped over in order to provide the animation. These explosions will have a timer between each displayed image to not make the animation seem instant, once a ship have exploded an explosion animation will play followed by a looping fire animation, this animation will continue to play on the destroyed ship until the game ends.

Usability Features

The usability features for the solution will be in the form of buttons with a clear and suitable name that are appropriate for the function they are meant to perform. Below is a table of different usability buttons, and a description of they do in detail:

Usability Feature	**What it's for**	**UI design (sketch)**
Title Screen	The title screen will be used to choose the mode the player wants to play in, these modes will be represented by buttons.	 Title Screen Image Easy mode Hard mode Player vs Player
Before the game starts	When the game-mode is chosen a pre-game setup screen will show up where the player or player's must place and finalize their ship positions before pressing a button to start the game.	 Player Grid Opponent grid (Hidden) Ships Start Randomize Reset

After the game starts	The buttons that were previously used to manipulate the ships and start the game will change to abilities, resetting and quit buttons, and the ships will be locked, the functions required to shoot on the grid will be enabled.	 Player Grid Opponent grid (Hidden) Ships Bomb Reset Game Quit Radar
Game ends with a winner	Once the game ends the user is directed back onto the initial title screen with a new text prompt on top the screen that will declare the winner of the game in case it was not clear who won based on the ships destroyed.	 Title Screen Player/Computer is the winner Image Easy mode Hard mode Player vs Player
Game in progress	While the game is in progress, there will be a visual indicator of ships hit, missed and empty spaces to indicate the present status of the game.	 Player Grid Opponent grid (Hidden) Ships Bomb Reset Game Quit Radar
Mode selection	The same screen that at the title screen will show at the mode selection screen the key difference will the buttons available, the multi-shot button will run a shoot until miss version of the game whilst the turn-based mode will feature a classic turn-by-turn shooting mechanic.	 Mode Selection Screen Image Multi-shot mode Turn-based mode

Radar display	When the radar button is pressed, an animated circular radar will show up onto the screen with the shot at positions on the grid overlapping it and a blip position that will appear moments after the radar is initialized and disappears once the radars animation ends.	 Player Grid Opponent grid (Hidden) Ships Bomb Reset Game Quit Radar
Reset	Resetting the game will clear the grid of all the blown-up cells, the opponent's ships will be randomized again whilst hidden and the players ships will return to their original position, the buttons will also revert to their pre-game state.	 Player Grid Opponent grid (Hidden) Ships Start Game Randomize Reset

Variables, Data structures & Classes

Variables

Make another column for the data type of each variable, data structure

Below are the key variables required in my project to make it work:

Variable	Why it's required
Number of rows, cols & cell size	The cell size will be iterated over the number of rows and cols to create a 10x10 grid onto the screen.
Game status	A Boolean value to track the play state of the game, this will be used to enable and disable, as well as change game mechanics based on whether the game is running or not.
Running	Used as a Boolean flag to keep the game running and end once set to false.

Screen width & screen height	These variables will be used to scale all the objects and create the game screen.
Scanner	A Boolean value that will be used to trigger the radar animation and will be used to trigger the algorithm to find and show a blip onto the screen.
Blip Position	Determines where to display the radar blip during a scan.

These variables were chosen as global variables because they're presence is fundamental for the battleship game to work properly.

Data Structures

The data structures used in the project are a mix of dictionaries and lists. Below are the key data structures required in my project to make it work:

Data Structure	**Why it's required**
Fleets	This will be a dictionary used to store the attributes of a fleet using their names as their key, the keys will be iterated over to create individual ship objects.
Buttons	A list will be used to create button objects, each list item will send attributes to a class to create an object.
Fire explosion list	This data structure will store all the images for the explosion, if there are a large number of images a function will be called to load the images.
Player & Computer fleets	All the values in ship fleet will be used to create ship objects, this list will be used to store all the computer and player fleet objects.
Player & Computer game logic	Used to store the information of the positions and the status of the ships and cell blocks on the grid.

Classes

These classes will be used to create game objects. Below are the key classes required in my project to make it work:

Class	Why it's required
Ships	Used to define attributes and functions given to individual ship objects.
Buttons	Individual interactive button objects, that will each have their own functions based on their name.
Player	Represents the player, handles the player action like shooting.
Easy computer	Represents the computer, handles the AI's actions such as randomized shooting.
Hard computer	Represents the computer, handles the AI's actions such as precision shooting.
Explosions	Used to animate the explosions once a ship has been shot at, to provide user feedback.

Plans for Testing

Testing The Prototype

What needs to be tested, how it will be tested

My approach to testing the code will follow the following steps:

- Identifying what can cause an error.
- How can it cause an error?
- Programming preventions and validations for the error.
- Testing the program to find out if the error occurs.
- If an identified or unidentified error occurs, I add for error handlers.
- Else if the error does not occur, I conclude the functionality a success and move on to the next problem.

Below is a table of a possible error causes and preventive measures for them:

Error cause	How it can cause an error	Preventive and validation measures	Testing procedure	Error handlers
Invalid coordinate grids for the ships	Ship will be placed outside the grid or on invalid coordinates.	The ship coordinates are validated before placing.	I'll try to place the ships outside the grid which is an invalid position.	If the ship is placed at an invalid position, it will be reset back to its default position.
Overlapping ships	Ships overlapping each other in the grid.	Check for overlapping ships before placing them onto the grid.	This can be tested by placing ships on overlapping positions on the grid.	The player will not be able to place the ship onto the grid until the position of one ship does not match another.
Invalid rotation	An invalid rotation can cause the ship to be outside the grid when rotated.	Validate ship alignment and rotation at certain positions.	The ships are placed at the ends of the grid.	Reset the ships orientation if it's leaving the grid when placed at the ends or reset the ship position when any part of the ship is leaving the grid.
Incorrect turn handler	Turns not switching correctly between the player and the opponent.	Ensure turn logic is properly implemented.	With the turn-based mode try shooting at the opponent's grid to check if it switches turns properly, with the multi-shot mode try missing the ships to check whether the multi-shot works with the computer ships. Log turns onto the console.	A Boolean value will be used to switch turns between the computer and player once any of them misses/shoots.
Animation error	Invalid path used to import an image.	Ensure the state of the animation is updated consistently after each action.	If an error occurs somewhere along the animation cycle break the animation images up and test them in batches to	Use a default static image if the animation does not load in properly and log the error onto the console.

			narrow down on the error image.	
Inconsistent game state	Game state not updated correctly between actions.	Ensure game state is updated consistently after each action.	Log the game state as a Boolean value to pinpoint the error.	If a certain game state results in an error, the game state will not change to that value until it doesn't cause an error.
Missing resources	Missing images, sounds assets.	Log the asset that is not correctly imported onto the console.	An image that loads up properly can be seen on the game screen, if it is not seen it has been identified as a missing resource.	If an asset does not exist or causes an error it can be ignored.
Invalid game rules	Rules such as ship placement and shooting not enforced correctly.	All game rules will be validated when a relevant action is to occur.	Game is tested with different scenarios to ensure the rules are enforced correctly.	If a rule is not implemented properly, the game action is reverted.

Post Development Phase

After I have finished developing my prototypes, I plan on distributing it to my stakeholders to conduct alpha testing, if my solution has satisfied my stakeholders the game is finalized, however if my prototype is not to their liking and they wish for me to amend it, I will conduct further research to implement their requested change or feature addition into my project and conduct another test, this process will continue until my stakeholders are satisfied with the solution. To be cautious for any missed errors white box testing will be conducted by me post development to make sure my solution is fully polished and error-free.

Development

Prototype Content Breakdown

Prototype 1: The first prototype will be used to develop the fundamental components of the game, this will include the necessities to create a working base for the battleship game that will be built upon in prototype 2, prototype 1 will have a larger focus on functionality rather than intractability.

Prototype 2: Once the main functionalities for the game have been made the focus will shift more onto what the stakeholders require from me such as power-ups and the user interface, before the development of prototype 2 my stakeholders will be asked to test prototype 1 and their feedback on bugs and complaints about the game will be addressed in prototype 2.

Breakdown and Explanation of Prototype 1

Prototype 1 consists of 3 files; main.py which is used to create the main program, the second file would be GameClasses.py which will be used to create all the classes these class will later be used to create objects that will be used in main.py, the final file will be called cirularImport.py this file possesses the shared utility functions, variables and lists, the reason I have chosen this name is because my python interpreter cannot simultaneously import from 2 different files at the same time, so to avoid repeating code I have put them all in a separate file to prevent the circular import error, this way both the game classes and main file can include data from each other.

Main.py

Importing and Initialization

Code*:*

```
"""Modules and Initialization"""
import pygame
from GameClasses import Ships, Buttons, Player,
EasyComputer, HardComputer
from circularImport import loadImage
import random

pygame.init()
pygame.mixer.init()
```

Explanation*:*

Two libraries have been imported (pygame and random). The pygame library is used to handle graphics, sound and sound functionalities which will be output onto the game canvas. The random library will be used to randomize ship positions, shots and other functions. The GameClasses file contains all the classes which will be used to create interactable objects on the grid. The file circular import contains shared resources which will be used in both the GameClasses file and the main.py file, the load image function will be used to load image files and provide them with a size. Some modules in pygame require initialization such as the display and sound modules, without the initialization of pygame these modules could not be used. The mixer function that has been initialized provides me with sound functionalities that will be used during the later stages of the game.

Game setting and variables

Code*:*

```
# GAME SETTING AND VARIABLES
screenWidth = 1260
screenHeight = 960
ROWS = 10
COLS = 10
cellSize = 50
buttonImg = 'assets/images/buttons/button.png'
deploymentStatus = True
SCANNER = False
INDNUM = 0
# 1 ship position
BLIPPOSITION = None
```

Explanation*:*

Variables and what they're for:

- **ScreenWidth, ScreenHeight:** The width on the game screen window will be set to 1260 pixels and the height to 960 pixels, the reason the width is considerably larger than the height is because the two grids will not be placed on top of each other rather, they will be placed at each other's side.
- **ROWS, COLS:** The number of rows and columns will be iterated over to create a 10x10 grid.
- **cellSize:** Each grid will have 100 cells; the size of each cell will be 50 pixels by 50 pixels.
- **buttonImg:** The assets folder contains all the assets used in the game; this variable specifies the path to it which will later be used as the image of the button objects.
- **deploymentStatus:** This Boolean value will indicate whether the game has started or not, once the game has started this Boolean value will restrict all the pre-game functions and allow different functions to occur instead.
- **SCANNER:** The scanner Boolean value will be used to trigger the scanner power-up which will be used to search the grid for ships.
- **INDNUM:** Since the scanner will have several images to provide animation this variable will act as an index counter that will keep track of the scanner images being displayed.

- **BLIPPOSITION:** The blip position be used to store the position of a random cell block on the grid which contains a ship, this position will be displayed on the grid as a blip when the scanner is used.

Pygame Display/ Initialization

Code*:*

```
gameScreen = pygame.display.set_mode((screenWidth,
screenHeight))
pygame.display.set_caption('Battle Ship')
```

Explanation*:*

The gameScreen variable contains the main game window which is the size of the width and height I declared earlier, this window will be where all the graphics, objects and animation will be displayed. The display.set_caption function from the pygame library provides the heading 'Battle Ship' for the game window that will be displayed on the top-left border of the screen, this is to be aware of what this window represents.

Game Lists/ Dictionary

Code*:*

```
"""Game Lists/Dictionary"""
# This is a dictionary of all the ships and the guns and
will be accessed later to make ship objects which will be
drawn
# to the grid
# Each vertical ship is initially 75 pixels apart
horizontally
# Sizes are adjusted for the scale
# I wanted ships that went into different rows/columns
but I thought the amount of effort required for something
so simple
# would just be a pain since I can't find ship assets
that're similar to my current assets
FLEETS = {
    'battleship': ['battleship',
'assets/images/ships/battleship/battleship.png',
                   (125, 600), (40, 195), 4,
'assets/images/ships/battleship/battleshipgun.png',
                   (0.4, 0.125), [-0.525, -0.34, 0.67,
0.49]],
    'cruiser': ['cruiser',
'assets/images/ships/cruiser/cruiser.png',
                (200, 600), (40, 195), 2,
'assets/images/ships/cruiser/cruisergun.png',
                (0.4, 0.125), [-0.36, 0.64]],

    'destroyer': ['destroyer',
'assets/images/ships/destroyer/destroyer.png',
                  (275, 600), (30, 145), 2,
'assets/images/ships/destroyer/destroyergun.png', (0.5,
0.15),
                  [-0.52, 0.71]],

    'submarine': ['submarine',
'assets/images/ships/submarine/submarine.png',
                  (425, 600), (30, 145), 1,
'assets/images/ships/submarine/submarinegun.png', (0.25,
0.125), [-0.45]],

    'carrier': ['carrier',
'assets/images/ships/carrier/carrier.png', (350, 600),
(45, 245), 0, None, None, None],

    'patrol boat': ['patrol boat',
'assets/images/ships/patrol boat/patrol boat.png', (50,
600), (20, 95), 0, None,
                    None, None],

    'rescue ship': ['rescue ship',
'assets/images/ships/rescue ship/rescue ship.png', (500,
600), (20, 95), 0, None,
                    None, None],
}
```

Explanation*:*

The FLEETS dictionary contains attributes about the 7 different ships I was granted from the assets folder, I chose the dictionary structure because it makes it easier to keep track of the ships as a key attribute relationship makes it clear and easier to iterate over. Each entry in the FLEETS dictionary follows the following structure ship name(as the key) : Ship Name, Ship image path, the initial position of the ships, the size of the ships, the number of guns (this was included because I was provided with an asset folder with the guns for each ship), the asset for the gun image, the scale of the gun in contrast with the ship and the gun offset (this value is used to place the ship onto the center of the ship at different positions with respect to the ship size, each offset value is the position for different guns on the ship). The ships with their gun path, gun scale size and offset as None were not given a gun asset in their folder. These values will be used as the attributes to create the 7 ship objects. The comments I made before the dictionary was the provide understanding of the different values in the dictionary for future maintenance, I have also the changes I would like to see in a future prototype such as ships go into different columns and rows since, this prototype had ships that go in straight lines down a row or column, this is due to the asset scarcity on the web. The offset and size for the guns were determined from trial and error until the gun was at a satisfactory position on the ship.

Code:

```
BUTTONS = [
    Buttons(buttonImg, (150, 50), (1100, 800),
'Randomize'),
    Buttons(buttonImg, (150, 50), (900, 800), 'Reset'),
    Buttons(buttonImg, (150, 50), (700, 800),
'Deployment')
]
```

Explanation:

The list buttons contains the initial buttons that will be displayed in the pre-game state, the Buttons class that has been imported from the game classes file is used to create objects from the 4 attributes:

- **Button image:** This is a dark rectangular image without any text.
- **Button size:** This is the size of the button, the first argument of the tuple is the width and the second argument is the height, the width is larger than the height because it is meant to be a rectangle, the reason I chose a tuple is because the size values are not to be changed in the development process.
- **Button position:** These are the top-left x, y coordinates of the buttons, each button is positioned 200 pixels apart from each other to prevent mistaken presses onto the wrong button.
- **Button name:** This is the string value for the name of the button which will later be processed into text and placed on the center of the button image, these names will also be used to provide functionality for each button.

Game Utility Functions

Code:

```
# GAME UTILITY FUNCTIONS
def createGameGrid(rows, cols, cellsize, pos):
    startX = pos[0]
    startY = pos[1]
    coordGrid = []
    for row in range(rows):
        rowX = []
        for col in range(cols):
            rowX.append([startX, startY])
            startX += cellsize
        coordGrid.append(rowX)
        startX = pos[0]
        startY += cellsize
    return coordGrid
```

Explanation:

The createGameGrid function takes in 4 parameter; rows which represents the number of rows on the grid (10), cols represents the number of columns (10), cellsize represents the cell size which in this case would just be a single value since I want the cell size to be a square rather than a rectangle and the final parameter within the function represents the top-left starting position of the grid, the position would be provided as a tuple as the x, y values would be different for the player and opponent grid and the positions should remain unchanged throughout the game. The starting position of the grid would be given in a (x, y) format, the first positional value (0) would represent the position in the x-axis for the grid and the second value (1) would represent the position in the y-axis. The coordGrid list will contain all the top left positions for every cell, I used a nested loop to perform this because the inner loop in a nested loop works horizontally (row) in each column therefore, the top-left y value should remain unchanged and since each cell is 50 pixels wide each iteration that is stored in the coordGrid list must increment by 50 pixels in the x-axis. After one row is finished the outer loop increased the y-value by 50 pixels to go to the next row, and the startX value is reset to its original position since I want it to start from the start of the next row and work its way through the row. Once this is done for each row in a column, the function returns a 2D list, each list represents 1 row, so the value coordGrid[0] would represents the first row, coordGrid[0][0] would provide the top-left position of the first cell, coordGrid[1][0] would provide the top-left position in the next row first column and coordGrid[0][1] would be the top-left position of the next column in the same row, this 2D list can therefore be used to manipulate and draw the grid as it provides the top-left position of every cell within the grid.

Code:

```
def createGameLogic(rows, cols):
    # creates gamelogic spaces ' ' for empty spaces on
the grid, this will be used to represent and make changes
to the grid
    gamelogic = []
    for row in range(rows):
        rowX = []
        for col in range(cols):
            rowX.append(' ')
        gamelogic.append(rowX)
    return gamelogic
```

Explanation:

The createGameLogic function will be used to represent the structure of the grid, the grid will be used to keep track of the state of the game, as you can see the structure game logic seems similar to the game grid, the empty spaces (‘ ‘) will be used as empty grid spaces, these grid spaces will later be updated to indicate a presence of a ship (‘O’), a hit ship(‘T’) or a missed shot (‘X’) which was previously an empty cell.

Code:

```
# Updates the gameGrid with the positions of the ships
def updateGameLogic(coordGrid, shipList, gameLogic):
    for i, rowX in enumerate(coordGrid):
        for j, colX in enumerate(rowX):
            # Checks for a hit or if a ship is there in a
specific cell
            if gameLogic[i][j] == 'T' or gameLogic[i][j]
== 'X':
                continue
            else:
                gameLogic[i][j] = ' '
                for ship in shipList:
                    if pygame.rect.Rect(colX[0], colX[1],
cellSize, cellSize).colliderect(ship.rect):
                        gameLogic[i][j] = 'O'
```

Explanation:

This function will be used to update the game logic spaces with the positions of the ships on the board with an ‘O’, the 3 parameters will be used as followed; since the coordGrid has the same structure as the gameLogic it can be iterated over to search for the ships, I chose to use enumerate instead of range in the loop because the current value of ‘i’ is automatically assigned to rowX which reduces the number of lines I need to write and makes it easier to understand. The value ‘T’ and ‘X’ represent a hit and miss, if a cell with these values is encountered, I’ve chose to skip processing because I do not overwrite these cells unnecessarily. However, if an empty cell block shares the same coordinates as a ships position, the cell is updated to an ‘O’ to represent a ship, the shipList is a list that contains all the ship objects that have been made from the ship class and rect is an attribute that has contains the coordinates of the 4 corners of the ship.

Code:

```
#Draws the player and computer grid onto the gameScreen
def showGridOnScreen(window, cellsize, playerGrid,
computerGrid):
    gamegrids = [playerGrid, computerGrid]
    for grid in gamegrids:
        for row in grid:
            #print(row)
            for col in row:
                #print(col)
                pygame.draw.rect(window, (255, 255, 255),
(col[0], col[1], cellsize, cellsize), 1)
```

Explanation:

The showGridOnScreen function displays/draws the player and computer grids onto the main game screen, the window parameter embodies the main game screen and the playerGrid and computerGrid are variables that were created from the createGameGrid function these variables contain the player grid, and the opponent's grid respectively. The function that is within the loop pygame.draw.rect requires the surface where the grid needs to be drawn (the game screen), the colour of the grid (white), the top-left and the top-right column position since the top-right position is 50 pixels apart from the top-left position it should be the position that comes right after the first position, the 5th and 6th argument are the height and width of each cell, and the argument is the thickness of the rectangle being drawn which in this case is 1 pixel.

Code:

```
def printGameLogic():
    print('Player Grid'.center(50, '#'))
    for i in pGameLogic:
        print(i)
    print('Computer Grid'.center(50, '#'))
    for i in cGameLogic:
        print(i)
```

Explanation:

This function displays the two logic grids onto the console, the two logic grids are separated by their respective grid names, this will be used to keep track of the current game scenario, which would be useful for debugging.

Code:

```
def createFleet():
    # Empty list that will hold fleet information
    fleet = []
    # Number of keys is 7 so it cycles through the 7 keys
    for name in FLEETS.keys():
        fleet.append(
            Ships(name,
                  FLEETS[name][1],
                  FLEETS[name][2],
                  FLEETS[name][3],
                  FLEETS[name][4],
                  FLEETS[name][5],
                  FLEETS[name][6],
                  FLEETS[name][7])
        )
    return fleet
```

Explanation:

This function will be used to create the ship list, this function creates a list called fleet which is initially empty, the function iterates over the 7 keys and passes their information to the ships class in order to create the ship objects, once all the 7 keys in the FLEETS dictionary are iterated over the fleet list is returned with all the ship objects.

Code:

```
def randomizeShipPositions(shiplist, gamegrid):
    """Selects random locations on the game grid for the
battleships"""
    placedShips = []
    for i, ship in enumerate(shiplist):
        validPosition = False
        while validPosition == False:
            ship.returnToDefaultPosition()
            rotateShip = random.choice([True, False])
            if rotateShip == True:
                yAxis = random.randint(0, 9)
                xAxis = random.randint(0, 9 -
(ship.hImage.get_width() // 50))
                ship.rotateShip(True)
                ship.rect.topleft =
gamegrid[yAxis][xAxis]
```

```
            else:
                yAxis = random.randint(0, 9 -
(ship.vImage.get_height() // 50))
                xAxis = random.randint(0, 9)
                ship.rect.topleft =
gamegrid[yAxis][xAxis]
            if len(placedShips) > 0:
                for item in placedShips:
                    if ship.rect.colliderect(item.rect):
                        validPosition = False
                        break
                    else:
                        validPosition = True
```

Explanation:

The randomizeShipPositions functions will be used to place the ships in random locations across the grid through a series of validation. An empty list called placedShips is declared initially, this list will be used to store the randomized ship positions one by one until all the ships are appended. Valid position is initialized to false within the shipList loop (which iterates 7 times for 7 ships), the valid position flag will be used to determine whether a ship position follows all the validation rules and only then a ship will be allowed to be placed on the grid. The validation required to place the ship on a randomized position on the grid, one of these rules is that the ships must not overlap, this has been implemented by first validating that there is a ship is present on the grid, this would mean that there is a possibly of collision, determining this has been done by checking whether there are any randomized positions in the placedShips list. If there are positions within the placedShips array, these positions are checked against the current randomized ship position, these the coordinates do not overlap then the new randomized ship position is valid and is appended to the placedShips array, on the contrary if it is not then another position is checked, this process continues until the coordinates do not intersect. A randomized ship position is determined using the randint method from the random library which selects a random index position from the player/computer grid list, the height/width of the images are divided by the cell size and subtracted from the randomized positions to prevent the ship from being placed outside the bounds of the grid. If a ship is chosen to be rotated a method from the class ships called rotateShip is set to true which uses an image for the ship that has been rotated by 90 degrees.

Code:

```
def sortFleet(ship, shipList):
    # The ships overlap over each other this function
moves the current selected ship to the bottom of the list
so there's no overlap
    shipList.remove(ship)
    shipList.append(ship)
```

Explanation*:*

As explained within the annotations of the code, the sortFleet function moves a selected ship (with the mouse) to the bottom of the shipList, which will make it appear on top of all the other ships when selected. Ships that are drawn later appear above their predecessors, so by removing the selected ship and moving it below it will essentially appear on top of the other ships since it's drawn later.

Code*:*

```
def pickRandomShipPosition(gameLogic):
    validChoice = False
    posX = 0
    posY = 0
    while not validChoice:
        posX = random.randint(0, 9)
        posY = random.randint(0, 9)
        if gameLogic[posX][posY] == 'O':
            validChoice = True

    return (posX, posY)
```

Explanation*:*

This function will be used later on to gather a random ship position on the grid which will later be displayed as a blip when the radar is used. This function gets a random position on the grid which consists of a ship and returns the value.

Code*:*

```
def loadAnimationImages(path, size, numImages):
    imageList = []
    for imageIndex in range(0, numImages):
        #the structure of the radar images are different
from the others
        if path == 'assets/images/radar_base/radar_anim'
or path == 'assets/images/radar_blip/Blip_':
            if imageIndex < 10:

imageList.append(loadImage(f'{path}00{imageIndex}.png',
size))

            elif imageIndex < 100:

imageList.append(loadImage(f'{path}0{imageIndex}.png',
size))
```

```
            elif imageIndex >= 100:
imageList.append(loadImage(f'{path}{imageIndex}.png',
size))
        else:
            if imageIndex < 10:
                imageList.append(loadImage(f'{path}
00{imageIndex}.png', size))

            elif imageIndex < 100:
                imageList.append(loadImage(f'{path}
0{imageIndex}.png', size))
            elif imageIndex >= 100:
                imageList.append(loadImage(f'{path}
{imageIndex}.png', size))
    return imageList

            elif imageIndex < 100:
                imageList.append(loadImage(f'{path}
0{imageIndex}.png', size))
            elif imageIndex >= 100:
                imageList.append(loadImage(f'{path}
{imageIndex}.png', size))
    return imageList
```

Explanation*:*

This function will be used to create a list of all the animation images for a certain feature, the logic for the index of an image changes for every power of 10 is because the assets I found have a ‘000’, ‘010’ and ‘100’ structure, so to successfully load the image I have had to add this validation. The images for the radar’s properties differ slightly from the other animation images, so I’ve added a small precaution for that at the beginning.

There are some functions I will not explain now as they require further context to understand. These functions will be explained later once I’ve provided the context to understand them.

Loading game Variables

Code*:*

```
"""LOADING GAME VARIABLES"""
# p stands for player and the grid in which the player has is initialised here
pGameGrid = createGameGrid(ROWS, COLS, cellSize, (50, 50))
pGameLogic = createGameLogic(ROWS, COLS)
pFleet = createFleet()

# c stands for computer and the grid in which the computer has is initialised here
cGameGrid = createGameGrid(ROWS, COLS, cellSize, (screenWidth - (ROWS * cellSize), 50))
cGameLogic = createGameLogic(ROWS, COLS)
cFleet = createFleet()
randomizeShipPositions(cFleet, cGameGrid)

player1 = Player()
computer = EasyComputer()

# Outputs all top-left coordinates for the game grid, use for debugging
# for i in pGameGrid:
#    print(i)

# for i in cGameGrid:
#    print(i)

# printGameLogic()
```

Explanation*:*

In this section, I create variables which contain a returned value from a function, the cGameGrid, cGameLogic and cFleet variables are used to computer grid, computer logic and a list of ship objects that will be used on the computer/opponent side of the game board, these variables are then repeated for the player with different arguments sent to the functions. Once these variables are created the ship positions are randomized, the reason these ship positions are randomized for the computer on start-up and not the player is because the computer fleets are meant to be unknown, while the player fleets must be placed according to the players decision. Towards the end of this section the player and computer objects are initialized from their respective classes which will be explained later, the reason I haven't created the second players object is because my priority for the first prototype is developing an engaging and playable battleship game. The bottom print statements are meant to understand how the values in the game grid work.

GameClasses.py

Initialization Of The Ship Class

Code*:*

```
class Ships:
# The offset is how many pixels the gun is from the
center according to the ships scale
    def __init__(self, name, img, pos, size, numGuns=0,
gunPath = None, gunsize = None, gunCoordsOffset = None):
        self.name = name
        self.pos = pos
        # v means vertical image size it's originally a
vertical image
        self.vImage = loadImage(img, size)
        self.vImageWidth = self.vImage.get_width()
        self.vImageHeight = self.vImage.get_height()
        self.vImageRect= self.vImage.get_rect()
        #This moves the image to the place I need it to
be
        self.vImageRect.topleft = self.pos
        #Loads horizontal image
        #this is for when the player chooses to rotate
the image you know to place it
        #h means horizontal image since I'm going to
transform it
        self.hImage =
pygame.transform.rotate(self.vImage, -90)
        self.hImageRect = self.hImage.get_rect()
        self.hImageRect.topleft = pos
        #Images and Rectanges
        self.Image = self.vImage
        self.rect = self.vImageRect
        self.rotation = False
        self.active = False
        self.numGuns = numGuns

        self.gunList = []
        if numGuns > 0:
            for guns in range(numGuns):
                self.gunList.append(Guns(gunPath,
                    self.rect.center,
                    (size[0] * gunsize[0], size[1] *
gunsize[1]),
                    gunCoordsOffset[guns]))
```

Explanation*:*

This is where the ship class is created and its attributes are declared. All the parameters are turned into attributes for the ship at first except for the size, this is because I want them to act as global variables that will be used throughout the class. The positional parameter will be the starting position of the ships, so that is where the ships will be initialized. The number of guns, gun path and gunsize defaults to None because some ships do not have a gun path, however, if they do have a gun path, the guns will initially be placed on the center of the ship and will then be scaled according to the size of the ship, then the position of the gun is determined relative to the ships center, where it is then placed. To represent the rotated (horizontal) image the vertical image is transformed by 90 degrees anti-clockwise, the rectangle position of the images will be later used to snap the ships onto the grid when placed. I have initialized 2 variables for validation; the self.active flag will be used to determine whether a ship has been clicked, if a ship is clicked the flag is set to true and the movement functionality is performed, the other validation flag declared alongside self.active is self.rotation, this flag is set to true if the user wishes to rotate a ship and false when a user does not want to rotate a ship.

The Functions of the Ship Class

Code*:*

```
def shipMove(self):
    if self.active:
        self.rect.center = pygame.mouse.get_pos()
        self.hImageRect.center = self.vImageRect.center =
self.rect.center
    for event in pygame.event.get():
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 3:
                self.rotateShip()
```

Explanation*:*

This function will be used be to move/rotate the ship by clicking on them, when a ship is clicked the self.active flag is set to true and this function executes, within this function the selected ships rectangular position is set to the mouses position, then all the ships positions are aligned with the current position of the ship, this has been done to synchronize the ships on the screen. For the rotation of the ship a pygame function is used determine whether a button has been pressed, if the button pressed is a right-click (which is what button 3 refers to) then the ship is rotated by calling the rotate function.

Code:

```
def rotateShip(self, doRotation=False):
    """switch ship between vertical and horizontal"""
    if self.active or doRotation == True:
        if self.rotation == False:
            self.rotation = True
        else:
            self.rotation = False
        self.switchImageAndRect()
```

Explanation:

The doRotation flag within the parameters of the rotation function is received from the randomizeShipPositions functions which was explained earlier, this flag is initialized to false to prevent accidental rotations, to rotate a ship with a right mouse click it must first be selected, that is why one of the criteria to rotate is the ship is that self.active must be true. If this function is called and either of these conditions are present the ship rotation flag is set to true if it has not yet been rotated, however, if it has been rotated the flag is set to false, and this flag used within another function that is called (switchImageandRect()) to switch the image of the ship depending on the flag's status.

Code:

```
def switchImageAndRect(self):
    """Switches from Horizontal to Vertical and vice
versa"""
    if self.rotation == True:
        self.Image = self.hImage
        self.rect = self.hImageRect
    else:
        self.Image = self.vImage
        self.rect = self.vImageRect
    self.hImageRect.center = self.vImageRect.center =
self.rect.center
```

Explanation:

This function is used to switch the image the rotational image depending on the status of the self.rotation value, if this flag is set to true the rotated image and position will be set to the main position and image of the ship, rather, if it's false the vertical image and position will be shown. Finally, all the images are aligned to the center of the image being shown on the grid.

Code*:*

```
def snapShiptoGrid(self, gridCoords):
    snapped = False
    cellSize = 50  # Define the cell size here
    for row in gridCoords:
        for cell in row:
            cell_rect = pygame.Rect(cell[0], cell[1],
cellSize, cellSize)
            if self.rect.colliderect(cell_rect):
                self.rect.topleft = (cell[0], cell[1])
                self.adjustShipPosition()

                self.vImageRect.center =
self.hImageRect.center = self.rect.center
                snapped = True
                break
        if snapped:
            break
    if not snapped:
        self.returnToDefaultPosition()
```

Explanation*:*

When the position of any ship is within a cell this function equates the top-left position of that ship onto that grid cell which creates the illusion of it being snapped on when it's placed. All the positions in the grid are cross-checked with the ship that has been selected and if any of these positions intersect with the rectangular position of the ship, the ship will be snapped onto the grid. The flag snapped has been declared to figure out the positional status of the ship, once a ship is snapped to a grid this flag is set to true to signal a ship has been successfully placed on a grid and the position of the ship is slightly adjusted from the top-left of the grid to make it appear as if it's on the center with another function, on the other hand, if the ship has not been collided with any positions on the grid the snapped flag will remain false and the a function will be called to return the ship to its original position.

Code*:*

```
def adjustShipPosition(self):
    # I'm sure there's a better way to do this, possibly
by equating the position of the ships to the center of
the grid
    # but I took a more manual approach, I could later
change this to make the program more future proof
```

```
    if self.Image == self.vImage:
        if self.name == 'patrol boat':
            self.rect.x += 15  # Adjust the x position
for the patrol boat
        elif self.name == 'rescue ship':
            self.rect.x += 15  # Adjust the x position
for the rescue ship
        elif self.name == 'cruiser':
            self.rect.x += 5  # Adjust the x position for
the cruiser
        elif self.name == 'destroyer':
            self.rect.x += 10  # Adjust the x position
for the destroyer
        elif self.name == 'submarine':
            self.rect.x += 10  # Adjust the x position
for the submarine
        elif self.name == 'battleship':
            self.rect.x += 5  # Adjust the x position for
battleship

    elif self.Image == self.hImage:
        if self.name == 'patrol boat':
            self.rect.y += 15  # Adjust the y position
for the patrol boat
        elif self.name == 'rescue ship':
            self.rect.y += 15  # Adjust the y position
for the rescue ship
        elif self.name == 'cruiser':
            self.rect.y += 5  # Adjust the y position for
the cruiser
        elif self.name == 'destroyer':
            self.rect.y += 10  # Adjust the y position
for the destroyer
        elif self.name == 'submarine':
            self.rect.y += 10  # Adjust the y position
for the submarine
        elif self.name == 'battleship':
            self.rect.y += 5  # Adjust the y position for
the battleship
```

Explanation*:*

This function is quite straight-forward, the ships are positioned a few pixels away from the top-left position of the grid depending on their name to make it seem like they're in the center of the grid.

Code*:*

```
def returnToDefaultPosition(self):
    # Returns the ship to its default position
    self.rect.topleft = self.pos
    self.hImageRect.center = self.vImageRect.center = 
self.rect.center
    # adding this part made it stop going off the grid 
because it's set to its default vertical position
    self.rotation = False
    self.switchImageAndRect()
```

Explanation*:*

This function was seen earlier when I was explaining the snapping feature, this function is used to return the ship to its default position, this is done by setting the current position of the ship to its original position, and to prevent the ship from being returned to its default position horizontally, I return the ship as its vertical image and position every time this function is called.

Code*:*

```
def offTheGrid(self, gridCoords):
    grid_start_x = gridCoords[0][0][0]
    grid_start_y = gridCoords[0][0][1]
    grid_end_x = gridCoords[-1][-1][0] + 50  # Adjust for 
the cell size
    grid_end_y = gridCoords[-1][-1][1] + 50  # Adjust for 
the cell size

    if not (grid_start_x <= self.rect.left < grid_end_x 
and
            grid_start_y <= self.rect.top < grid_end_y 
and
            grid_start_x < self.rect.right <= grid_end_x 
and
            grid_start_y < self.rect.bottom <= 
grid_end_y):
        self.returnToDefaultPosition()
```

Explanation:

The offTheGrid function checks whether the ship is within the grid or out of bounds, to check if the ship is within the grid, the top-left starting and ending x and y values are stored in their respective variables, the ending grid cell position is adjusted by 50 pixels, this is to account for width of the final grid cell. The four corners of the ship is checked with the value stored in the variables to see if it's within the grid. If the four corners are greater than the starting position of the grid and less than the final bottom right position of the grid that would mean it's within the grid. If these requirements are not met the ship is rotated to its vertical position and placed at its starting default position.

Code:

```
def checkShipCollision(self, shipList):
    # make a copy of pFleet because I don't wanna ruin
that one yet
    # if it isn't copied the original list item is
removed not the copy
    sList = shipList.copy()
    sList.remove(self)
    for item in sList:
        if self.rect.colliderect(item.rect):
            return True
    return False
```

Explanation:

This function has been made to detect whether a ship is colliding with another ship when being manually placed, a copy of the ship list has been made, because if the original list is modified the ships would disappear from the game on start-up, since the list would be empty. The ship that has been selected is removed from the list because it must not check for collisions with itself or else, I would not be able to place the ship. All the ships within the list are checked with the selected ship within a loop, if the positions of the ships are within each other then, the function returns true and whilst the value of the function is true the ship will not be allowed to snap onto the grid in the main game loop, however, if a ship does not collide or has stopped overlapping with another ship then the False Boolean value is returned which means the criteria has been fulfilled to place the ship.

Code:

```
def draw(self, window):
    #draws the ships onto the screen
    window.blit(self.Image, self.rect)
    for guns in self.gunList:
        guns.drawGuns(window, self)
```

Explanation:

The final function within the ship class will be used to contentiously draw the ships and guns within the main game loop, this function displays the appropriate image at its current position, the window parameter represents the main game screen, since all the guns were appended to a list earlier, they must be drawn through the usage of a loop, the drawGuns function is within another class.

Initialization Of The Guns Class

Code:

```
class Guns:
    def __init__(self, gunImage, pos, size, offset):
        self.gunImage = loadImage(gunImage, size, True)
        self.offset = offset
        self.gunImageRect = self.gunImage.get_rect()
```

Explanation:

All the attributes are initialized and declared here such as; the image of the gun, the offset from the center of the ship and the position of the guns.

The Functions of the Guns Class

Code:

```
def update(self, ship):
    # Updates the gun position on the ship
    self.gunImageRect.center = (ship.rect.centerx,
ship.rect.centery + (ship.Image.get_height() // 2 *
self.offset))
```

Explanation:

The ship position updates according to the position of the ship, these positions, the gun is continuously placed at the center of the ship with an offset in the y-axis.

Code:

```
def drawGuns(self, window, ship):
    self.update(ship)
    window.blit(self.gunImage, self.gunImageRect)
```

Explanation:

This function was used in the ship class to draw the ships onto the grid, the update function is called to update the ships position according to the current center position of the ship, then the ship is placed onto the grid with its updated position.

Initialization Of The Buttons Class

Code*:*

```
class Buttons:
    def __init__(self, Img, size, pos, msg):
        self.pos = pos
        self.buttonImg = loadImage(Img, size)
        self.buttonRect = self.buttonImg.get_rect()
        self.ImageLarger =
pygame.transform.scale(self.buttonImg, (size[0] + 10,
size[1] +10))
        self.buttonRect.topleft = pos
        #centerizes the text with the buttonrect so it
don't look like it's in the corner
        self.msgClick = msg
        self.msg = self.addText(msg)
        self.msgRect = self.msg.get_rect(center =
self.buttonRect.center)
```

Explanation:

The buttons class will be used to create, place and provide functionality for the buttons on the game screen. The attributes declared within this class are; the position of the buttons, the image and size of the button which will remain unchanged constant for all the buttons, from the image the rectangle is gathered, this rectangle will be used to interact with the buttons, a larger image is created from the original image of the button, this image is 10 pixels larger than the original button in both the x and y axis, the larger image will be used as a UI feature that makes it clear to the user which button they are hovering over. The position of the button is set to the position provided in the argument of a button object; this position will be unchanged throughout execution of a game-cycle. The msgClick attribute was created to distinguish between which button was clicked, the msg attribute could not be used for this purpose as it is used to process the message string into text that can be displayed on the grid, and lastly the msgRect attribute will be used to position the text correctly on the button.

The Functions of the Button Class

Code*:*

```
def addText(self, msg):
    """add font of the image button"""
    font = pygame.font.SysFont('Stencil', 22)
    message = font.render(msg, 1, (255, 255, 255))
    return message
```

Explanation:

This function will be used to render the message onto the pygame surface, I have first created a font, I chose the stencil font as it is bold by default which makes it easier to read on the game screen and a size seemed suitable as it did not seem overly large on the game screen. The message is rendered using the font I've just created within the function and the render method is called to make the message placeable on the game screen, I've enabled anti-aliasing as which is seen in the second argument being set to true, this has been done to make the message seem less pixelated, and the color I chose for the text is white. This message will be displayed on the game screen in the main file.

Code*:*

```
def focusOnButton(self, window):
    if
self.buttonRect.collidepoint(pygame.mouse.get_pos()):
        window.blit(self.ImageLarger, (self.buttonRect[0]
- 5, self.buttonRect[1] - 5, self.buttonRect[2],
self.buttonRect[3]))
    else:
        window.blit(self.buttonImg, self.buttonRect)
```

Explanation:

When the player hovers over the button the larger image will be displayed, once the button is enlarged the position of the text on the button should change to keep the button in the center of the enlarged button, this has been done by shifting the message 5 pixels upwards and to the left, whilst the width and height of the message stay the same, the new button is then displayed on the grid.

Code*:*

```
#Game status indicates whether or not a game has started
    def updateButton(self, gameStatus):
        if self.msgClick == 'Reset' and gameStatus ==
False:
            self.msgClick = 'Quit'
        if self.msgClick == 'Quit' and gameStatus ==
True:
            self.msgClick = 'Reset'
        elif self.msgClick == 'Deployment' and gameStatus
== False:
            self.msgClick = 'ReDeploy'
        elif self.msgClick == 'ReDeploy' and gameStatus
== True:
            self.msgClick = 'Deployment'
        elif self.msgClick == 'Randomize' and gameStatus
== False:
            self.msgClick = 'Radar Scan'
        elif self.msgClick == 'Radar Scan' and gameStatus
== True:
            self.msgClick = 'Randomize'
        self.msg = self.addText(self.msgClick)
        self.msgRect = self.msg.get_rect(center =
self.buttonRect.center)
```

Explanation:

The gameStatus parameter is in function indicates whether a game has started or not, the text on the buttons is dependent on the gameStatus, there are 6 buttons within the game, 3 of these buttons will be displayed on the grid at a time, if the game is in its pre-game state the three buttons that will be displayed will be deployment, randomize and reset. The deployment button will be used to start the game, once this button is clicked all pre-game functions like placing and randomizing the ships will be inaccessible and the shooting mechanic will be validated, this button will also trigger all the other buttons to change. The randomize button will be used to randomize the ship positions for both the player and opponent fleets, the reset button will be used to reset the players ship positions to their default positions and randomizes the opponent/computers ships. Once the game has been started the text on the three buttons will be changed to ‘Redeploy’, ‘Radar Scan’ and ‘Quit’. The redeploy button clears the explosions on the game grid, set the ships back to their default position and randomize the opponent’s fleets, it practically works like a reset button, the final function this button performs is switching the texts on the buttons back to their pre-game state. The radar scan button triggers the radar functionalities which is used to find a ship on the grid. The quit buttons terminates the game window. The message that is being rendered changes to the new message and the new text is centered on the button image.

Code*:*

```
def randomizeShipPositions(self, shipList, gameGrid,
randomizePos):
    randomizePos(shipList, gameGrid)
```

Explanation:

This function within the button class uses randomize ship function in the main game loop to randomize the ship positions when the randomize button is clicked.

Code*:*

```
def resetPosition(self, shipList):
    for ship in shipList:
        ship.returnToDefaultPosition()
```

Explanation:

This function will be used to reset the ships positions when the game has been restarted or when the player chooses to reset his ships positions from the grid.

Code*:*

```
def draw(self, window):
    self.focusOnButton(window)
    window.blit(self.msg, self.msgRect)
```

Explanation:

The draw function will draw the message (not the button) onto the grid, with a suitable position. The focusOnButton function is called within this functions to adjust the position of message when the player hovers over a specific button.

Initialization Of The Player Class

```
class Player:
    def __init__(self):
        self.turn = True
```

Explanation:

The player class will handle the shooting mechanic for the player, it consists of one attribute and that attribute will be used to validate whether it's the players turn.

The Functions of the Player Class

Code*:*

```
def playerShoot(self, coordGrid, gameLogic, EXPLOSIONS):
    posX, posY = pygame.mouse.get_pos()
    #first one is the original bottom left pos, second on
is 1 cell up and next is right bottom last is top right
this makes up the grid
    #tbh I don't know what these represent
    if posX >= coordGrid[0][0][0] and posX <=
coordGrid[0][-1][0] + 50 and posY >= coordGrid[0][0][1]
and posY <= coordGrid[-1][0][1] + 50:
        for i, rowX in enumerate(coordGrid):
            for j, colX in enumerate(rowX):
                if posX >= colX[0] and posX < colX[0] +
50 and posY >= colX[1] and posY <= colX[1] + 50:
                    # Checks for a hit or if a ship is
there in a specific cell
                    if gameLogic[i][j] != ' ':
                        if gameLogic[i][j] == 'O':
                            gameLogic[i][j] = 'T'
                            self.turn = False
                            print('Hit')

EXPLOSIONS.append(Explosion(redExplosion,
coordGrid[i][j], 'Hit', None, None, None))
                    else:
                        gameLogic[i][j] = 'X'
                        print('Miss')

EXPLOSIONS.append(Explosion(blueExplosion,
coordGrid[i][j], 'Hit', None, None, None))
                        self.turn = False
    return EXPLOSIONS
```

Explanation:

Since the player class has a single purpose and that is to shoot on the opponent's grid, it is also composed of one function. When the game has started, if the player clicks on the grid that mouse position is retrieved, that position represents the player shooting at a grid, this value is checked with the top-left, top-right and the bottom-left of the opponent's grid to make sure the player has clicked within the boundaries of the grid or else the shooting function will not be performed. If the player has clicked within the grid all the grid cells are searched to figure out which cell the click lies within. The gameLogic grid is checked since once the game starts its updated with the ship's positions, the entire gameLogic grid is searched and if the position the player has shot contains a ship 'O' that value is changed to a T to indicate a hit and the word 'Hit' is output to the console for debugging purposes, the Boolean value self.turn is set to false indicating the players turn is over and the explosion animation is played. Contrastingly, if the position on the gameLogic grid is set to ' ' which represents are empty grid space, the value on the logic grid is changed to 'X' indicating a miss, the word miss is output to the console, furthermore the miss animation is played at that position and the players turn is over until the computer shoots which will be implemented in the main game loop. There is an explosions list in the main game loop which contains all explosions with their positions, the value being returned is assigned to it.

Initialization Of The Computer Class

Code*:*

```
class EasyComputer:
    def __init__(self):
        self.turn = False
        self.status = self.computerStatus('Thinking')
        self.name = 'Easy Computer'
```

Explanation:

The turn in the EasyComputer class is initially set to false whilst the turn in the player class is initialized to true is because the player shoots at computers grid first. The value within the self.status variable will be displayed on the game screen when it's the computers turn to shoot, this is done to show that the shooting mechanic is executing properly, I've also initialized a name for the computer.

The Functions of the Computer Class

Code*:*

```
def computerStatus(self, msg):
    image = pygame.font.SysFont('Stencil', 22)
    message = image.render(msg, 1, (0, 0, 0))
    return message
```

Explanation:

This function processes and renders the text 'Thinking' and returns the processed value.

Code*:*

```
def computerShoot(self, gameLogic, pGameGrid,
EXPLOSIONS):
    validChoice = False
    while not validChoice:
        rowX = random.randint(0, 9)
        colX = random.randint(0, 9)

        if gameLogic[rowX][colX] == ' ' or
gameLogic[rowX][colX] == 'O':
            validChoice = True

        if gameLogic[rowX][colX] == 'O':
            gameLogic[rowX][colX] = 'T'
            print('Hit')
            EXPLOSIONS.append(Explosion(redExplosion,
pGameGrid[rowX][colX], 'Hit', fireExplosionList,
explosionList, None))
            self.turn = False
        else:
            if gameLogic[rowX][colX] != 'T':
                gameLogic[rowX][colX] = 'X'
                print('Miss')

EXPLOSIONS.append(Explosion(blueExplosion,
pGameGrid[rowX][colX], 'Hit', None, None, None))
                self.turn = False
```

Explanation:

Within this function a random row and column value are found using the random function, this is within 0 to 9, since the grid is a 10x10. In order to prevent the computer from shooting at the same position twice, I've set a condition for shooting and that condition is that random position chosen for shooting must be a ship or an empty space, not a hit ship or a missed cell. If the random position is a ship ('O') the value is set to 'T' indicating a hit, additionally the hit animation is played and the computers turn is over. If the position is a miss not a ship position and not a hit ship position the value within that grid is set to 'X', the miss animation is played and the computers turn is over.

Code*:*

```
def draw(self, window, cGameGrid):
    cellSize = 50
    if self.turn:
        window.blit(self.status, (cGameGrid[0][0][0] - 
cellSize, cGameGrid[-1][-1][1] + cellSize))
```

Explanation:

This function is used to draw the text onto the grid when it's the computers turn to shoot, the position where the message is displayed. The image is displayed 50 pixels below the bottom-left position of the grid.

Initialization Of The Explosions Class

Code*:*

```
class Explosion:
    #This class will animate the explosition in the cell 
which got hit or missed factors determine the image shown
    def __init__(self, image, pos, action, imageList = 
None, explositionList = None, soundFile = None):
        self.image = image
        self.pos = pos
        #set up rect with topleft because that's where 
the grid is
        self.rect = self.image.get_rect(topleft=pos)
        self.imageList = imageList
        self.action = action
        self.timer = pygame.time.get_ticks()
        self.imageIndex = 0
        self.explosionList = explositionList
        self.explosionIndex = 0
        self.explosion = False
```

Explanation:

The explosions class has seven attributes these attributes consist of: the path of the image, the position where the image is to be displayed, the position of the rectangle is set to the top-left position which has been passed into the function, this puts the correct explosion in the right grid position. The image list will contain the animation image, the images will be displayed sequentially to provide the fire animation, the action attribute will be given a 'Hit' or a 'Miss' value and depending on this value, an image will be shown, I've added the timer to prevent the animation from seeming instant, I will use this function to provide a small delay before the next animation image is shown, this makes the animation seem smoother and blend in. The imageIndex will be used to keep track of which animation image in imageList is being displayed or will be displayed next. The explosionList and explosionIndex follow the same logic as the imageList, the index will be used to keep track of which explosion will be displayed next. The difference between the explosionList and the imageList is that the imageList will display the fire image, this animation will run continuously until the game has ended whilst the explosion animation will occur once a player has shot a ship and once this explosion animation is finished the fire image will continuously animate in a loop. The Boolean flag self.explosion will, explosion Boolean value will be used to determine when an explosion is active or not and the soundFile is not a priority yet so it will be ignored for now.

Functions Of The Explosions Class

Code*:*

```
def animateExplosion(self):
    self.explosionIndex += 1
    if self.explosionIndex < len(self.explosionList):
        return self.explosionList[self.explosionIndex]
    else:
        return self.animateFire()
```

Explanation:

The initial explosion (not the fire) will be animated within this function, every time this function is called the explosionIndex is increased by one to show the next animation image, to not increase to an index position that is outside of the list the index position is required to be less than the list of explosions, if it's less than the length of the explosion list the explosion is animated, once it exceeds the value of the explosion list a function to animate the fire is animated instead.

Code*:*

```
def animateFire(self):
    # Gives room to the animation does not look instant
    if pygame.time.get_ticks() >= self.timer >= 100:
        self.timer = pygame.time.get_ticks()
        self.imageIndex += 1
    """makes it so the image iterates with the amount of
images"""
    if self.imageIndex < len(self.imageList):
        return self.imageList[self.imageIndex]
    else:
        # once the animation is finished the animation
occurs again
        self.imageIndex = 0
        return self.imageList[self.imageIndex]
```

Explanation:

This function will be used to animate the fire which will be shown after the explosion, to not make the animation seem instant the animation index for the fire is increased every 100 milliseconds, the imageIndex is validated to be in the bounds of the fire list and then the next image in the fire list is returned, once the index exceeds the length of the fire list, the index value is reset and the cycle repeats.

Code:

```
def draw(self, window):
    # if the value is still none
    if not self.imageList:
        window.blit(self.image, self.rect)
    else:
        self.image = self.animateExplosion()
        self.rect = self.image.get_rect(topleft=self.pos)
        self.rect[1] = self.pos[1] - 10
        if pygame.time.get_ticks() >= self.timer >= 100:
            window.blit(self.image, self.rect)
```

Explanation:

To display all the animation images the draw function is required, within this function it's first checked whether a fire list is provided or not because the hit and miss indicators for the player are not animated, if is there is a single image provided then that image is shown because there has been no animation list provided, however, if a fire animation list is provided an explosion or fire image is gathered from the animationExplosion function and the position of this fire or explosion is adjusted to be within the grid and is then displayed after a 100 milliseconds to not make the animation seem instant.

Main.py

With the added context of the GameClasses.py file, it will now be easier to understand some of the other functions within the first prototype of my battleship game.

Declaring Static Explosion

Code:

```
redExplosion =
loadImage("assets/images/tokens/redtoken.png", (cellSize,
cellSize))
blueExplosion =
loadImage("assets/images/tokens/bluetoken.png",
(cellSize, cellSize))
```

```
greenExplosion =
loadImage("assets/images/tokens/greentoken.png",
(cellSize, cellSize))
```

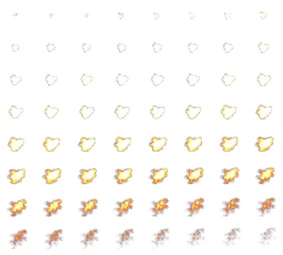
Explanation:

These variables contain the static explosion and miss images for the player and computer.

Code Dealing With The Explosion

Code*:*

```
def seperateExplosionImages(explosionSheet, rows, cols,
newSize, size):
    image = pygame.Surface((128, 128))
    # position on the sheets is topleft 0, 0 and the
place where you get the picture from comes after
    image.blit(explosionSheet, (0, 0), (rows * size[0],
cols * size[1], size[0], size[1]))
    # transforms it into the size of the grid
    image = pygame.transform.scale(image, (newSize[0],
newSize[1]))
    # removes the background from the spreadsheet
selected image
    image.set_colorkey((0, 0, 0))
    return image
```



Explanation:

The explosion file I was provided from the assets folder has an 8x8 sheet that is 1024 x 1024 pixels in size, which means 1 explosion image within the explosion sheet file is 128 x 128 pixels in size, the initial position given to the image is the top-left of the game screen, this position is arbitrary as it is later changed to its proper position on the grid, this function will be iterated within a function with 8 rows and 8 columns, this is to separate each image. The row and cols corresponds to the image that is to be separated, since each image is 128 x 128 pixels wide, the size that they will be extracted with would be the same, once an image is separated it's scaled down to 50 x 50 pixels to fit within a cell on the grid. The image I provided away seems white because word usually removes the background from an image but, the sprite sheet I was given was black so, in order to remove the background from the image I have made the black background transparent so the background isn't visible when the image is animated, once the image is processed it is returned and added to a list which will later be displayed on the grid sequentially.

Code*:*

```
fireExplosionList =
loadAnimationImages('assets/images/tokens/fireloop/fire1_
', (cellSize, cellSize), 13)

# loads in a collective of 128 by 128 images
explosionSheet =
pygame.image.load("assets/images/tokens/explosion/explosi
on.png").convert_alpha()
explosionList = []

for row in range(8):
    for col in range(8):

explosionList.append(seperateExplosionImages(explosionShe
et, col, row, (cellSize, cellSize), (128, 128)))

EXPLOSIONS = []
```

Explanation:

The 13 animation images are first loaded in as a list using the loadAnimationImages function, these images don't require any processing since, they are given individually so the processing with the loadAnimationImages function is enough. The explosion sheet is loaded in and an explosionList which is declared. The explosion list will be iterated over 64 times to store all the animation images with their proper cell size (the explosion list is within the loop, even though it may not look like it above) , this list will later be displayed within a function in the main game loop, and finally a list called EXPLOSIONS is declared, this list will be used to create explosion objects, this list will also contain the fire animation and the hit or miss explosions.

Code Dealing With The Radar

Code*:*

```
def displayRadarScanner(imageList, indNum, SCANNER):
    if SCANNER == True and indNum <= 359:
        image = increaseAnimationImage(imageList, indNum)
        return image
    else:
        return False
```

Explanation:

The scanner Boolean flag is used to validate the activation of the scanner, indNum represents the current index value of the scanner image being displayed, this number should not exceed 359 since, that is the number of scanner images within the list, if the index number has not exceeded the number of radar images the radar animation image is returned, opposingly if it does exceed the number of radar images False is returned and the animation ends.

Code*:*

```
RADARGRIDIMAGES =
loadAnimationImages("assets/images/radar_base/radar_anim"
, (ROWS * cellSize, COLS * cellSize), 360)
# blip needs to be in cell so 50, 50
RADARBLIPIMAGES =
loadAnimationImages("assets/images/radar_blip/Blip_",
(cellSize, cellSize), 11)
RADARGRID =
loadImage('assets/images/grids/grid_faint.png', (ROWS *
cellSize, COLS * cellSize))
```

Explanation:

The reason there are 360 radar grid images that are loaded in within this list is because the radar image will animate in a circular motion, and since a circle is 360 degrees, each angle contains 1 animation frame. The radar blip images will be a small green dot to indicate a ships position, this animation image consists of 11 images, the radar grid will be a faint static outline of the computers/opponent's grid, this image will be on the opponent's grid at all times.

Code*:*

```
def increaseAnimationImage(imageList, ind):
    if ind <= 359:
        return imageList[ind]
```

Explanation:

This function returns a radar image depending on the current index position of the radar, this index position increases when the scanner Boolean flag is active and resets when it's complete.

Code*:*

```
def displayRadarBlip(num, position):
    if SCANNER:
        image = None
        if position[0] >= 5 and position[1] >= 5:
            if num >= 0 and num <= 90:
                image =
increaseAnimationImage(RADARBLIPIMAGES, num // 10)
        elif position[0] < 5 and position[1] >= 5:
            if num >= 90 and num <= 180:
                image =
increaseAnimationImage(RADARBLIPIMAGES, (num // 2) // 10)
        elif position[0] < 5 and position[1] < 5:
            if num >= 180 and num <= 270:
                image =
increaseAnimationImage(RADARBLIPIMAGES, (num // 3) // 10)
        elif position[0] >= 5 and position[1] < 5:
            if num >= 270 and num <= 360:
                image =
increaseAnimationImage(RADARBLIPIMAGES, (num // 4) // 10)
        return image
```

Explanation:

The num parameter within the displayRadarBlip function contains the current radar angle/index position and the position parameter encompasses the x and y coordinate of a random ship position on the grid. The scanner value must be validated to true before the radar is displayed. The calculations taken to display the blip is different depending on which quadrant the random location of the blip is in, if the blip is within the top-right quadrant coordinate, it would mean that the animation image has not passed the 90

degree mark, in this case, the index position of the radar is divided by 10 and that index position of the blip is shown on the grid, the reason I divide the radar index by 10 and provide it as the blip index is because I want the blip to animate with the movement of the radar, this index is divided by 2 in different quadrants so that I provide a valid index value for the blip as the blip only consists of 13 images. The blip is displayed depending on which quadrant its position is in, for example if the radar image enters the bottom right quadrant which is the 180–270-degrees section and the position is greater than 5 on the y-axis and less than 5 on the x-axis, the radar will first have to enter that quadrant before the animation for the blip starts. Once the blip image that is to be displayed is determined it is returned and displayed on the screen.

Functions With Added Context

Code*:*

```
def deploymentPhase(deployment):
    if deployment == True:
        return True
    else:
        return False
```

Explanation:

This function is responsible for assigning a variable with the status of the game, if the game has not started yet the deployment status will be true whilst if the deployment button is clicked the game has started and the deployment status becomes true.

Code*:*

```
def takeTurns(p1, p2):
    if p1.turn == True:
        p2.turn = False
    else:
        p1.turn = False
        p2.turn = True
        while p2.turn:  # Ensure the computer takes its
turn until it misses or changes turn
            p2.computerShoot(pGameGrid, pGameLogic,
EXPLOSIONS)
        if p2.turn == False:
            p1.turn = True
```

Explanation:

The take turns function will be used to determine whose turn it currently is, if the attribute turn is true with the player 1 object, player 2's turn will be set to false, since prototype 1 does not consist of the hard computer or local co-op I'll refer to player 2 as computer, once the players turn is set to false the computers turn is set to true and the computer gets to shoot, with prototype 1 there's no multi-shot mode so, with the given scenario, once the computer shoots the computers turn is set to false and once it's set to false the player's turn is set to true and the player gets to shoot, this cycle continues until the game ends.

Function For Displaying Onto the Screen

Code*:*

```
def updateGameScreen(GAMESCREEN):
    GAMESCREEN.fill((0, 0, 0))
    showGridOnScreen(GAMESCREEN, cellSize, pGameGrid,
cGameGrid)
    for ship in pFleet:
        ship.draw(gameScreen)
        ship.snapShiptoGrid(pGameGrid)
    for ship in cFleet:
        ship.draw(gameScreen)
        ship.snapShiptoGrid(cGameGrid)
    #    GAMESCREEN.blit(computerGridImage,
(cGameGrid[0][0][0], cGameGrid[0][-1][-1]))
    for button in BUTTONS:
        button.updateButton(deploymentStatus)
        button.draw(gameScreen)
        radarScan = displayRadarScanner(RADARGRIDIMAGES,
INDNUM, SCANNER)
        if radarScan == False:
            pass
        else:
            GAMESCREEN.blit(radarScan,
(cGameGrid[0][0][0], cGameGrid[0][-1][-1]))
        RBlip = displayRadarBlip(INDNUM, BLIPPOSITION)
        if RBlip:
            GAMESCREEN.blit(RBlip,
(cGameGrid[BLIPPOSITION[0]][BLIPPOSITION[1]][0],

cGameGrid[BLIPPOSITION[0]][BLIPPOSITION[1]][1]))
        for explosion in EXPLOSIONS:
            explosion.draw(gameScreen)
#       computer.draw(gameScreen, cGameGrid)
    updateGameLogic(pGameGrid, pFleet, pGameLogic)
    updateGameLogic(cGameGrid, cFleet, cGameLogic)
    pygame.display.update()
```

Explanation:

I have created the updateGameScreen function to provide all the visual effects on the game screen, this function is constantly iterated within the main game loop. The screen is first cleared and given a black background, then the 2 game grids are drawn to screen using the showGridOnScreen function. The player and computer ships are drawn and updated on the grid; they are also provided with the functionality of snapping to the grid. I was initially planning on hiding the computers grid, however, for the first prototype my main focus was functionality so I did not see the point of hiding the grid therefore, I commented it out. The buttons are displayed at their default position just like the ships and are updated depending on the status of the game. The radar is displayed within the button object as the radar functionality occurs when the radar scan button is pressed. The radar scan button contains the 1 of the 360 radar images that are constantly updated once the radar button is clicked the radar is shown on the animated on the top-left portion of the computer grid and once the radar passes the ship at the blips position, the blip is animated with the ships position. All the explosion objects within the EXPLOSIONS list are displayed, after that, the game logic for both the computer and player grids are updated, this happens towards the end after all the processing has been done and finally, the game screen is refreshed to display all the newly drawn elements.

Main Game Loop

Code*:*

```
"""Main Game Loop"""
# code determines the status of pygame and ends the session if
event type is quit
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                if deploymentStatus == True:
                    for ship in pFleet:
                        if
ship.rect.collidepoint(pygame.mouse.get_pos()):
                            sortFleet(ship, pFleet)
                            if not
ship.checkShipCollision(pFleet):
                                ship.active = True
                                ship.shipMove(pFleet)
                else:
                    if player1.turn:
                        player1.playerShoot(cGameGrid,
cGameLogic, EXPLOSIONS)
                        takeTurns(player1, computer)  # Switch
turns after player shoots
                    while computer.turn:  # Ensure the computer
takes its turn until it misses or changes turn
                        computer.computerShoot(pGameGrid,
pGameLogic, EXPLOSIONS)
                    takeTurns(player1, computer)  # Switch turns
after computer shoots

                for button in BUTTONS:
                    if
button.buttonRect.collidepoint(pygame.mouse.get_pos()):
                        # if button.msgClick == 'ReDeploy':
                        #    deploymentStatus =
button.redeploy(deploymentStatus)
                        if button.msgClick == 'Randomize':
                            if deploymentStatus == True:

button.randomizeShipPositions(pFleet, pGameGrid,
randomizeShipPositions)

button.randomizeShipPositions(cFleet, cGameGrid,
randomizeShipPositions)
                        elif button.msgClick == 'Reset':
                            if deploymentStatus == True:
                                # calling it twice cuz calling
it once don't seem enough to return it to the proper position
                                button.resetPosition(pFleet)
                                button.resetPosition(pFleet)
                        elif button.msgClick == 'Deployment':
                            deploymentStatus =
deploymentPhase(False)
                        elif button.msgClick == 'ReDeploy':
                            deploymentStatus =
deploymentPhase(True)
                            for ship in pFleet:
                                ship.returnToDefaultPosition()
                                ship.returnToDefaultPosition()
                            EXPLOSIONS.clear()
                            button.msgClick = 'Deployment'
                        elif button.msgClick == 'Quit':
                            running = False
                        elif button.msgClick == 'Radar Scan':
                            SCANNER = True
                            INDNUM = 0
                            BLIPPOSITION =
pickRandomShipPosition(cGameLogic)


            elif event.button == 3:  # Right click
                for ship in pFleet:
                    if deploymentStatus == True:
                        ship.rotateShip()

            elif event.button == 2:
                printGameLogic()

        elif event.type == pygame.MOUSEBUTTONUP:
            for ship in pFleet:
                if event.button == 1:
                    sortFleet(ship, pFleet)
                    ship.offTheGrid(pGameGrid)
                    if not ship.checkShipCollision(pFleet):
                        ship.active = False

    for ship in pFleet:
        if deploymentStatus == True:
            ship.shipMove(pFleet)

    updateGameScreen(gameScreen)
    if SCANNER == True:
        INDNUM += 1

    takeTurns(player1, computer)

pygame.quit()
```

Explanation:

The variable running is declared that controls whether the main game loop continues to run, this variable is initially set to true when the I run the game, once it's set to false the game ends as the main game loop is terminated. When the player clicks the window close button on the top-right end of the screen the game loop is terminated. However, if the game is running and the player uses the left mouse button to click on a ship and the game is in its ship placement stage, the ships get sorted within the fleet and checked for collisions, then activated to move on the grid. After the player chooses to end the ship placement phase and start the game, the player has to click on the opponents ship to shoot at the opponent's fleet, once the player has shot the players turn is set to false and it becomes the computer's turn to shoot, this processing is done within the takeTurns function, the same operation is occurred for the computer and this continues until the game ends. To undertake the button functionality, within the game loop is any button is clicked different actions are performed such as randomizing the ships, resetting the ship positions, starting/ending the game, initializing the radar scan and closing the game window. When the randomize button is clicked the positions for both the player and computer fleets are randomized, the resetPosition function and returnToDefaultPosition functions are called twice within their button because when I choose to return the ships to their starting positions some of them don't return at the exact position but when I call it twice, they return without fail. If the game has started and the player decides to reset the game, they click on the ReDeploy button, this button resets all the buttons to default text, clears the explosions on the grid, randomizes the computer fleets and resets the fleets to their default position. The index number is reset to 0 when the radar scan button is clicked because the animation is supposed to be displayed again. When the player clicks the right button whilst a ship is selected in its pre-game state the ship is rotated but once it's placed it cannot be rotated until it's selected again. The mouse wheel when pressed prints the game logic, this is used for debugging and provides an accurate state of each cell on the grid. To deselect a ship, I've added a mouse up event, this event checks to see if the left mouse button is released, if it is released the fleets are sorted and checked to be on the grid, a check is made to prevent the ships from being placed if they overlap, if they do not overlap the ships are deselected and snapped onto the grid. The position of the player ships are updated during the deployment phase and all the visual effects are constantly updated within the main game loop using the updateGameScreen function, if the scanner flag is set to true and the radar scan button has been pressed the index number is increased by 1 until it reaches the end of the radar images list. Finally, the take turns function is called to correctly handle the turn management mechanic between the player and the computer and once the loop ends which means running is set to false all the pygame resources are released to properly close the game.

circularImport.py

Code*:*

```
import pygame

pygame.init()

cellSize = 50
screenWidth = 1260
screenHeight = 960
ROWS = 10
COLS = 10

gameScreen =
pygame.display.set_mode((screenWidth,screenHeight))

# loads the images
def loadImage(path, size, rotate=False):
    img = pygame.image.load(path).convert_alpha()
    img = pygame.transform.scale(img, size)
    # This will be used later once I create the rotate
functionality
    if rotate == True:
        img = pygame.transform.rotate(img, -90)
    return img
```

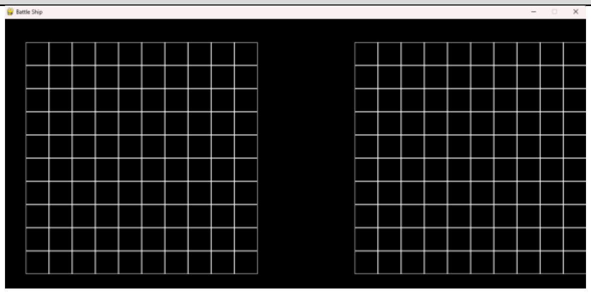
```
redExplosion =
loadImage("assets/images/tokens/redtoken.png", (cellSize,
cellSize))
blueExplosion =
loadImage("assets/images/tokens/bluetoken.png",
(cellSize, cellSize))
greenExplosion =
loadImage("assets/images/tokens/greentoken.png",
(cellSize, cellSize))

def seperateExplosionImages(explosionSheet, rows, cols,
size, newSize):
    image = pygame.Surface((128,128))
    #position on the sheets is topleft 0, 0 and the place
where you get the picture from comes after
    image.blit(explosionSheet, (0, 0), (rows * size[0],
cols * size[1], size[0], size[1]))
    #transforms it into the size of the grid
    image = pygame.transform.scale(image, (newSize[0],
newSize[1]))
    #removes the background from the spreadsheet selected
image
    image.set_colorkey((0, 0, 0))
    return image

def loadExplosionImages(size):
    imageList = []
    for imageIndex in range(0,13):
        if imageIndex < 10:

imageList.append(loadImage(f'assets/images/tokens/fireloo
p/fire1_ 00{imageIndex}.png', size))

        elif imageIndex < 20:

imageList.append(loadImage(f'assets/images/tokens/fireloo
p/fire1_ 0{imageIndex}.png', size))
    return imageList

fireExplosionList = loadExplosionImages((cellSize,
cellSize))
```

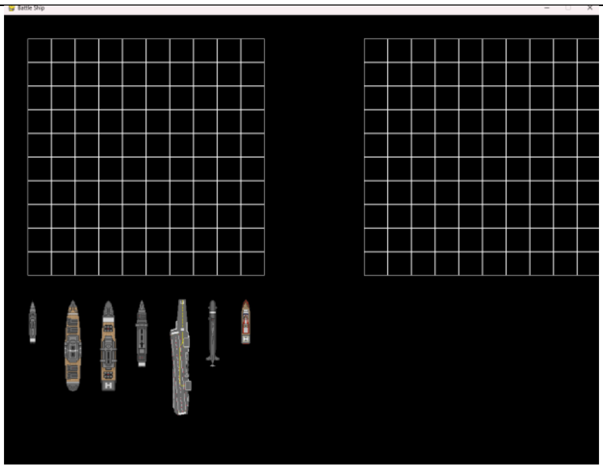
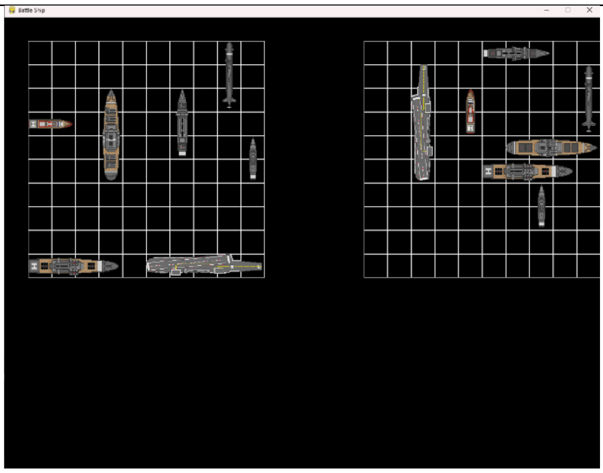
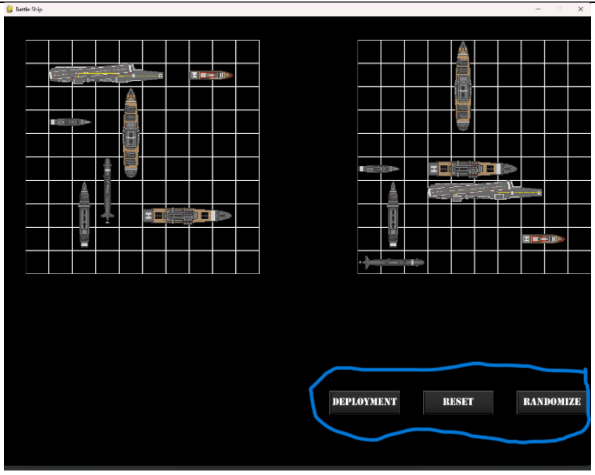
```
#loads in a collective of 128 by 128 images
explosionSheet =
pygame.image.load("assets/images/tokens/explosion/explosi
on.png").convert_alpha()
explosionList = []

for row in range(8):
    for col in range(8):

explosionList.append(seperateExplosionImages(explosionShe
et, col, row,  (cellSize, cellSize), (128, 128)))
```

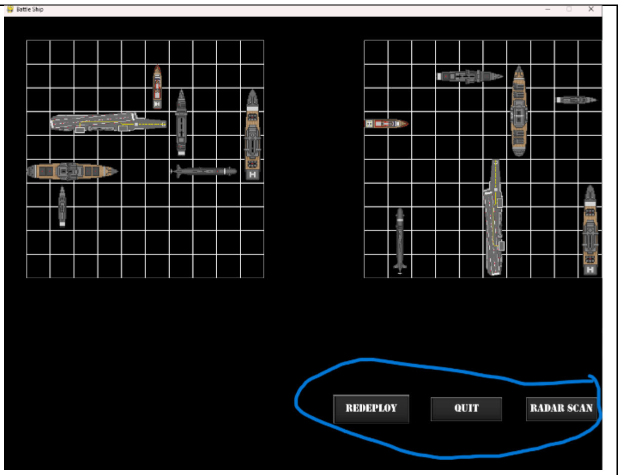
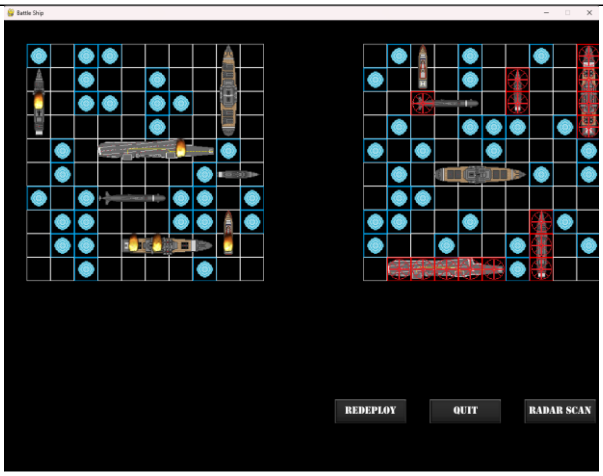
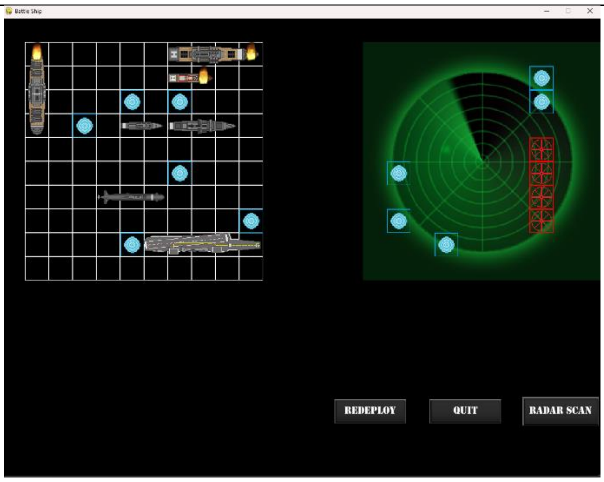
Explanation:

I've created this file as to store all the shared utility functions within GameClasses and Main.py, the only new addition within the file is the loadImage function, the other code has been explained. Within the loadImage function the image is first loaded in and optimized for blitting using the convert_alpha method, it is then transformed with the size provided within the parameters, this function is also present in the other files in different forms so I've also checked for rotation, if the ship is rotated the image is transformed at 90 degrees counter-clockwise, once the image is completely processed it is returned.

UI Elements for Prototype 1

UI Identification	Application
The 2 empty player and computer grids.	

The player fleets at their default position.	
The player fleets manually placed on the grid and the computer fleets randomized onto the grid.	
The buttons before the game starts.	

The buttons that the default buttons change to when the game starts.	
The explosions and the randomized turn-based shooting mechanic on the grid.	
The radar scanner sweeping the grid and a small green blip within the 270- and 360-degree quadrant on the grid.	

Prototype 1 Testing

The way I have evaluated and tested prototype 1 is by providing a copy to each of my stakeholders and allowing them assess my coded solution, the testing method I have implemented with them has been black box alpha testing, once my stakeholders have completed testing prototype 1, they will be provided with another survey, within this survey they will mention what they are satisfied with, what they want me to work on, any errors they have encountered and what they want to see in the next prototype.

Gathering Stakeholder Feedback

I have asked my stakeholders a series of questions and collected their responses, these responses will serve as the building blocks for prototype 2.

The questions I had asked were:

Q1: What aspects of the game are you most satisfied with?

Q2: What areas of the game need the most improvement?

Q3: Have you encountered any errors or bugs when testing the game? If so, please describe them.

Q4: What features of changes would you like to see in the next prototype?

The responses I have gotten were as follows:

Name	Q1	Q2	Q3	Q4
Jack	The radar animation works surprisingly well, the blip on the radar does not linger on for too long whilst simultaneously providing a substantious amount of time to spot the blip.	The user interface within this prototype is subpar, however that is to be expected as your main focus should be on functionality rather than the graphics, however just add a little more to make it more user-friendly.	I've tried every possible scenario I could possibly think of but I am pleased to say that I haven't found a single error.	A better AI, as I'd mentioned within the initial survey, I would like a complex AI that seems like challenging and fun to play against.

Lily	The visual fidelity of the fire that is pre-faced by an explosion seems like a great user-interface addition.	The user interface is quite stale, there should be an interactable home screen and a winner needs to be declared once all the fleets on either end of the board are destroyed.	None in the first prototype.	haven't yet seen the 2x2 bomb feature that I had requested in the first prototype implemented, I would like another button for the bomb and a hidden grid on the computer's end.
Omar	The ship placement mechanic seems so smooth and intuitive, I love the little attention to detail given to the placement mechanic, example when I've rotated the ship within the grid and I move it out of the grid the ship is rotated back to its default rotation and I must go back into the grid to rotate it.	All the features that you've implemented into the game have been implemented without any flaws, so for the features in prototype 1 I wouldn't say there are any improvements required.	None	definitely a better AI, the AI within this prototype does not provide a challenge.
Oscar	The ship placement mechanic and the radar worked better than I'd thought.	The AI is quite lackluster and easy to win against, and the turn-based mode is quite dated.	None	I would like a better AI that seems slightly less randomised and more precise.

Breakdown of the Responses & What I Learned

There seems to be an overwhelming number of responses requesting a better AI, so that will be the top priority within my next prototype, thankfully none of my stakeholders have found a bug/error so it would not be necessary to debug my code to make it error free.

Below is a complete breakdown of all the responses:

- **Jack:** Jack has praised the way I have implemented the radar functionality, however in contrast, he believes my user-interface is quite lackluster and requires some work done to make it more interactive and provide further functionality. It seems Jack carried out white box testing to identify an error, since he claims he went through every single scenario to find an error, this further enforces my belief debugging not being required for prototype 1 when moving onto prototype 2. Jack has alluded to his previous response when asked about what he would like to see in prototype 2, Jack would like a more complex AI in the second iteration, I have already gathered plans for developing a more complex AI based on these responses and it has to do with checking the spaces around a hit ship for other positions containing a ship and shooting at those positions.

- **Lily:** Lily seems to be impressed with how smooth the visual effects for the explosions seem but, is not too fond of the user-interface, this seems to be a recurring concern as Jack had also mentioned it within his responses. No improvements is required to any of the existing objects according to lily which provides a greater insight, this could do to the fact she has not found an error. Lily seems to be the only respondent who has not commented on the AI but has rather commented on the 2x2 bomb she had requested in the previous survey, she has provided instructions on she would like this to be implemented and it is through an additional button when the game starts, she would also like a hidden grid which is a must for prototype 2.
- **Omar:** Omar has been expressed admiration for the ship placement mechanic so, it would seem there is no need to amend this functionality, he has not requested any improvements and has not found any errors. Omar has followed the general consensus for an improvement in the AI which will be worked on.
- **Oscar:** The radar and ship mechanic has yet again been praised by Oscar. Oscar seems to not like the turn-based mode which seems consistent to his previous response, I could satisfy this demand by creating a multi-shot mode which is to be prompted right after the home screen and the AI seems to not be challenging to Oscar. No bugs have been found and Oscar would like to see a more precise and less randomized AI in the next iteration, this request in consist with the idea I have for a hard mode.

Additions in Prototype 2

- ❖ **Hard Computer**: The way the hard computer will work is as follows, once the computer shoots and if it's a hit, the 4 squares surrounding the hit position(top, bottom, left and right) will be checked for a ship, once a ship is found in any of these cells the computer will shoot at it and repeat this process until the entire ship is destroyed and there are no ships that are found surrounding the hit position.
- ❖ **Home Screen**: The home screen will be used to select which mode the player would like to play in, the user or users will be provided with 3 modes these modes will be as follows: Easy Computer, Hard Computer and Player vs Player (Local co-op), once the player chooses a mode that mode will be used to represent the opponents shooting mechanic, and the user will be directed to a turn-based screen.
- ❖ **Turn-based Screen:** The turn-based screen will be used to select whether the player wants to play turn-by-turn or the multi-shot mode, the multi-shot mode will allow the user or computer to shoot until they miss, once the player chooses the mode, they want to play in the will be directed to the main game screen and this is where the game will be played.
- ❖ **Hidden grid & Grid Background:** Within the assets folder there is a hidden grid for the computer and a background for the user grid that does not need to be hidden when playing against a computer, these will be added once the project is completed and all the other features are added.

- **Bomb:** I will create a bomb button; this button will shoot at a random 2x2 position on the opponents grid.
- **End Screen:** The end screen will declare the winner, the winner will be decided by which players fleets were destroyed first and the end screen will contain the same buttons from the Home screen to choose which mode they would like to play in next time around, once the user presses any of these modes the game will start with the opponent the user chose to play against.

Breakdown and Explanation of Prototype 2

There will be no new files implemented in prototype 2, however there will be multiple amendments to the functions and classes created in prototype 1 and the classes created in prototype 1 will be used to create new objects for prototype 2, the code that has been added to a class or function will be highlighted with a bright colour, all the other code that has not been highlighted can be ignored, however, for the classes or functions that are entirely new I will not be highlighting them as no changes have occurred within them.

Main.py

Initialization

Code*:*

```
"""Modules and Initialization"""
import pygame
from GameClasses import Ships, Buttons, Player,
EasyComputer, HardComputer
from circularImport import loadImage
import random

pygame.init()
pygame.mixer.init()
```

Explanation:

The HardComputer class have been created within the GameClasses.py file, this class will be used to create the hard computer object.

Game Setting & Variables

Code*:*

```
# GAME SETTING AND VARIABLES
# The height and width of the screen canvas
screenWidth = 1260
```

```
screenHeight = 960
ROWS = 10
COLS = 10
cellSize = 50
buttonImg = 'assets/images/buttons/button.png'
pGameGridImg =
loadImage('assets/images/grids/player_grid.png', ((ROWS +
1) * cellSize, (COLS + 1) * cellSize))
cGameGridImg =
loadImage('assets/images/grids/comp_grid.png', ((ROWS +
1) * cellSize, (COLS + 1) * cellSize))
deploymentStatus = True
SCANNER = False
INDNUM = 0
textFont = pygame.font.SysFont('Stencil', 60)
# 1 ship position
BLIPPOSITION = None
mainMenuScreenAsset =
loadImage('assets/images/background/Battleship.jpg',
(screenWidth // 3 * 2, screenHeight))
endScreenAsset =
loadImage('assets/images/background/Carrier.jpg',
(screenWidth // 3 * 2, screenHeight))
turnScreenAsset =
loadImage('assets/images/background/Submarine.jpg',
(screenWidth // 3 * 2, screenHeight))
MENU = True  # Added global menu variable
easyStatus = True
computer = HardComputer()
endStatus = False
turnScreenStatus = False
turnBasedStatus = True
playerPlaying = False
```

Explanation:

The stencil font variable that I have created contains the font for the text that will be used to declare the winner. The home screen, turn screen and the end screen background assets are loaded in with a width that takes up 2/3rds of the screen, the rest of the space will be used to place the buttons. The easyStatus flag will be used to determine whether the easy AI should be used or the Hard AI, if easyStatus is set to false the computer will be set the HardComputer class, however, if it’s true the computer object will be created using the EasyComputer class. The MENU, turnScreenStatus and endStatus will be used to determine which screen should be displayed, if MENU is set to true the home screen will be displayed, that's why it's initially set to true as the first screen that should be displayed on boot-up should be the home screen, turnScreenStatus will be used to show the screen where the player chooses if they want to play using the 'shoot until miss' mode or the 'turn-based' mode, the turnBasedStatus flag will be used to determine which mode the user decided to play using, if the user decided to play with the turn-by-turn mode this flag is set to true, however, if the player decides to play using the multi-shot mode this flag is set to false, the endStatus Boolean value will be used to show the end screen once all the opponents or players fleets are destroyed and the playerPlaying variable will be set to true if the user decides to play against another human player, if not this flag is set to false.

Game Setting & Variables

Code*:*

```
# Game Lists
BUTTONS = [
    Buttons(buttonImg, (150, 50), (1100, 800),
'Randomize'),
    Buttons(buttonImg, (150, 50), (900, 800), 'Reset'),
    Buttons(buttonImg, (150, 50), (700, 800),
'Deployment'),
    Buttons(buttonImg, (150, 50), (500, 800), 'Bomb'),
    Buttons(buttonImg, (250, 100), (900, screenHeight //
2 - 150), 'Easy Computer'),
    Buttons(buttonImg, (250, 100), (900, screenHeight //
2 + 150), 'Hard Computer'),
    Buttons(buttonImg, (250, 100), (900, screenHeight //
2 - 150), 'Turn-by-Turn'),
    Buttons(buttonImg, (250, 100), (900, screenHeight //
2 + 150), 'Multi-shot'),
    Buttons(buttonImg, (250, 100), (900, screenHeight //
2), 'Player vs Player'),
]
```

Explanation:

A new bomb button is created that is 200 pixels away in the x-axis from the previous button and is the same size as the buttons from prototype 1. The 'Easy Computer', 'Player vs Player' and 'Hard Computer' buttons will be displayed on the home screen and end screen, these buttons are sized and placed relative to the home screen and end screen background images, these buttons are separated by 150 pixels on the y-axis and are 250 pixels wide and 100 pixels long.

Screen Functions

Code*:*

```
def mainMenuScreen(gameScreen):
    # creates illusion of switching game screens
    global MENU
    global easyStatus
    global computer
    global turnScreenStatus
    global playerPlaying
    if turnScreenStatus == False:
        if MENU == True:
            gameScreen.blit(mainMenuScreenAsset, (0, 0))
            for button in BUTTONS:
                if button.msgClick == 'Easy Computer' or button.msgClick == 'Hard
Computer' or button.msgClick == 'Player vs Player':
                    button.draw(gameScreen)
            for event in pygame.event.get():
                if event.type == pygame.MOUSEBUTTONDOWN:
                    if event.button == 1:
                        for button in BUTTONS:
                            if button.buttonRect.collidepoint(pygame.mouse.get_pos()):
                                if button.msgClick == 'Easy Computer':
                                    MENU = False
                                    easyStatus = True
                                    turnScreenStatus = True
                                    computer = EasyComputer()
                                    playerPlaying = False
                                elif button.msgClick == 'Hard Computer':
                                    MENU = False
                                    easyStatus = False
                                    turnScreenStatus = True
                                    playerPlaying = False
                                    computer = HardComputer()
                                elif button.msgClick == 'Player vs Player':
                                    MENU = False
                                    turnScreenStatus = True
                                    playerPlaying = True
                                return
```

Explanation:

I've made this portion of the code smaller because if it's too large the lines break into each other which makes it difficult to understand within this document. I have accessed a series of global variables within this function, these global variables will be changed if a specific action is performed. If the MENU flag is set to true this means the home screen is to be displayed so, the screen is cleared and the main menu image is displayed from the top-left of the screen and the easy, hard and player buttons are drawn, these buttons will be used to select the mode the user wants to play in. Once any of the buttons are clicked the home screen image and buttons are no longer drawn on the screen and it switches to the turn-based screen where the user chooses what style they want to play the game with. The unique properties of each button are as follows: the easy button will set the opponent as the easy AI, whilst the Hard button will set the opposer as the Hard AI and the player vs player button will set opponent as another player and set the playerPlaying flag that is used to indicate another user is playing to True, these changes are returned.

Code*:*

```
def endScreen(gameScreen):
    # creates illusion of switching game screens
    global endStatus
    global easyStatus
    global MENU
    global turnScreenStatus
    global computer
    global pGameLogic
    global cGameLogic
    global turnScreenStatus
    global deploymentStatus
    gameScreen.blit(endScreenAsset, (0, 0))
    gameWinners = endGameLogic(pGameLogic, cGameLogic)

    if not gameWinners[0]:
        drawText('Computer won', textFont, ('green'), 150, 150)

    elif not gameWinners[1]:
        drawText('Player won', textFont, ('green'), 150, 150)

    for button in BUTTONS:
        if button.msgClick == 'Easy Computer' or button.msgClick ==
'Hard Computer' \
                or button.msgClick == 'Player vs Player':
            button.draw(gameScreen)
    for event in pygame.event.get():
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                for button in BUTTONS:
                    if
button.buttonRect.collidepoint(pygame.mouse.get_pos()):
                        if button.msgClick == 'Easy Computer':
                            endStatus = False
                            easyStatus = True
                            MENU = False
                            turnScreenStatus = True
                            cGameLogic = createGameLogic(ROWS, COLS)
                            pGameLogic = createGameLogic(ROWS, COLS)
                            EXPLOSIONS.clear()
                            for ship in pFleet:
                                ship.returnToDefaultPosition()
                                ship.returnToDefaultPosition()
                            deploymentStatus = True
                            computer = EasyComputer()
                        elif button.msgClick == 'Hard Computer':
                            endStatus = False
                            easyStatus = False
                            MENU = False
                            turnScreenStatus = True
                            cGameLogic = createGameLogic(ROWS, COLS)
                            pGameLogic = createGameLogic(ROWS, COLS)
                            EXPLOSIONS.clear()
                            for ship in pFleet:
                                ship.returnToDefaultPosition()
                                ship.returnToDefaultPosition()
                            deploymentStatus = True
                            computer = HardComputer()
                        elif button.msgClick == 'Player vs Player':
                            endStatus = False
                            easyStatus = True
                            MENU = False
                            turnScreenStatus = True
                            cGameLogic = createGameLogic(ROWS, COLS)
                            pGameLogic = createGameLogic(ROWS, COLS)
                            EXPLOSIONS.clear()
                            for ship in pFleet:
                                ship.returnToDefaultPosition()
                                ship.returnToDefaultPosition()
                            deploymentStatus = True
                            computer = Player()
                        return
```

Explanation:

The endScreen function will change screens once all the fleets on either side of the board are destroyed. The end screen will feature an image at the same position as the main menu screen and the same buttons, these buttons provide the same functionality as the buttons on the main menu screen except they change different flags and the game logic is reset and the winner is decided using the endGameLogic function, the winner is displayed using large green text at the end screen. When any of the buttons is clicked all the screens are set to false except the turn-based screen which will be used to allow the player to decide which mode they want to play in, all of the buttons reset the game logic for both the grids, the buttons, explosions and ships, the enemy fleet is randomized. The main difference between the buttons is that computer object is set to the EasyComputer when the easy button is clicked, the hard button sets the computer object to HardComputer and the player button sets it to the Player class to create a player object and the updated values are returned.

Code*:*

```
def endGameLogic(PGAMELOGIC, CGAMELOGIC):
    global endStatus

    # Initialize flags to track if any ships are left
    playerShipsLeft = False
    computerShipsLeft = False

    # Check if there are any 'O's in the game logic grid
    for row in PGAMELOGIC:
        for cell in row:
            if cell == 'O':
                # If 'O' is found, then there are still
ships left
                playerShipsLeft = True
                break
        if playerShipsLeft:
            break

    for row in CGAMELOGIC:
        for cell in row:
            if cell == 'O':
                # If 'O' is found, then there are still
ships left
                computerShipsLeft = True
                break
        if computerShipsLeft:
            break

    # If no 'O' was found, set endStatus to True
    if not playerShipsLeft or not computerShipsLeft:
        endStatus = True
    else:
        endStatus = False

    winner = [playerShipsLeft, computerShipsLeft]
    return winner
```

Explanation:

The endGameLogic function will be used to decide who the winner at the end of a game cycle. This function creates 2 Boolean variables, these Boolean variables will be used to decide whether there are any ships left in the game. Both the computer and player logic grids are searched, if a ship ('O') is found the variables are set to true and the loop is broken because if it keeps searching the grid an empty space or a hit/miss ship would be found which would pre-maturely end the game, this will go on until a ship cannot be found on either grid, in which case the endStatus will be set to true, this flag will trigger the end screen and the winner will be decided depending on which Boolean value is left true, the Boolean value that is set to false will be the loser and the value set to true will be the winner, the winner value is returned and is used to display the winner on the end screen.

Code*:*

```
def turnBasedScreen(gameScreen):
    # creates illusion of switching game screens
    global endStatus
    global easyStatus
    global turnScreenStatus
    global turnBasedStatus
    gameScreen.blit(turnScreenAsset, (0, 0))
    for button in BUTTONS:
        if button.msgClick == 'Turn-by-Turn' or
button.msgClick == 'Multi-shot':
            button.draw(gameScreen)
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    if
button.buttonRect.collidepoint(pygame.mouse.get_pos()):
                        if button.msgClick == 'Turn-by-Turn':
                            turnScreenStatus = False
                            turnBasedStatus = True
                        elif button.msgClick == 'Multi-shot':
                            turnScreenStatus = False
                            turnBasedStatus = False
                        return
```

Explanation:

This function will be used to display the turn-based screen where the player will choose if they want to play using the turn-by-turn mode or the multi-shot mode. The turn-based asset will be at the same position and size as the other screen assets, and the 'Multi-shot' and 'Turn-by-Turn' button will be display on this screen as oppose to the computer and player buttons on the home and end screen. When either of these buttons are pressed the turn screen is no longer displayed and the main game screen is displayed where the player gets to play. When the turn-by-turn button is pressed turnBasedStatus is set to true, this allows the shooting to be alternated between the player and the opponent, whilst the multi-shot button sets the turnBasedStatus to false, this allows the player or the computer to shoot until they miss.

Code*:*

```
def drawText(msg, font, text_col, x, y):
    global gameScreen
    img = font.render(msg, True, text_col)
    gameScreen.blit(img, (x, y))
```

Explanation:

This function processes the winner text and displays it onto the grid at a position where it can be seen, I chose this position to be over the end screen asset.

Loading the Sound

```
#Loading in the sound
hitSound =
pygame.mixer.Sound('assets/sounds/explosion.wav')
hitSound.set_volume(0.05)
shotSound =
pygame.mixer.Sound('assets/sounds/gunshot.wav')
shotSound.set_volume(0.05)
missSound =
pygame.mixer.Sound('assets/sounds/splash.wav')
missSound.set_volume(0.05)
```

Explanation:

This function will be used to load in and set the volume of the miss, hit and explosion assets, these assets will play when either the computer or player shoots at the opponents grid.

Turn Function

Code*:*

```
def takeTurns(p1, p2):
    if p1.turn == True:
        p2.turn = False
    else:
        p1.turn = False
        p2.turn = True
        while p2.turn:  # Ensure the computer takes its
turn until it misses or changes turn
```

```
            if playerPlaying == False:
                p2.computerShoot(pGameLogic, pGameGrid,
EXPLOSIONS, turnBasedStatus)
            else:
                p2.playerShoot(pGameGrid, pGameLogic,
EXPLOSIONS, turnBasedStatus)
        if p2.turn == False:
            p1.turn = True
```

Explanation:

When it’s player 2’s turn, if either of the computers are playing the computer gets to shoot, however, if the player is playing the playerShoot method from the player class is used and the player shoots on the main players grid.

Game Variables

Code*:*

```
# LOADING GAME VARIABLES
# p stands for player and the grid in which the player
has is initialised here
pGameGrid = createGameGrid(ROWS, COLS, cellSize, (50,
50))
pGameLogic = createGameLogic(ROWS, COLS)
pFleet = createFleet()

# c stands for computer and the grid in which the
computer has is initialised here
cGameGrid = createGameGrid(ROWS, COLS, cellSize,
(screenWidth - (ROWS * cellSize), 50))
cGameLogic = createGameLogic(ROWS, COLS)
cFleet = createFleet()
randomizeShipPositions(cFleet, cGameGrid)

player1 = Player()
player2 = Player()
```

Explanation:

The player2 object representing the second player is created, this object encompasses all player1's attributes except, it will be used on player1's grid.

Displaying the Changes

Code*:*

```
def updateGameScreen(GAMESCREEN):
    if MENU == True and endStatus == False and turnScreenStatus
== False:
        mainMenuScreen(GAMESCREEN)

    if turnScreenStatus == True:
        GAMESCREEN.fill((0, 0, 0))
        turnBasedScreen(GAMESCREEN)

    if MENU == False and endStatus == True and turnBasedStatus
== False:
        GAMESCREEN.fill((0, 0, 0))
        endScreen(gameScreen)

    if MENU == False and endStatus == False and turnScreenStatus
== False:
        GAMESCREEN.fill((0, 0, 0))
        showGridOnScreen(GAMESCREEN, cellSize, pGameGrid,
cGameGrid)
        GAMESCREEN.blit(pGameGridImg, (0, 0))

        for ship in pFleet:
            ship.draw(gameScreen)
            ship.snapShiptoGrid(pGameGrid)

        for ship in cFleet:
            ship.draw(gameScreen)
            ship.snapShiptoGrid(cGameGrid)
        GAMESCREEN.blit(cGameGridImg, (cGameGrid[0][0][0] - 50,
cGameGrid[0][0][1] - 50))
        #    GAMESCREEN.blit(computerGridImage,
(cGameGrid[0][0][0], cGameGrid[0][-1][-1]))

        for button in BUTTONS:
            button.updateButton(deploymentStatus)
            #        button.redeploy(deploymentStatus)
            if button.msgClick != 'Easy Computer' and
button.msgClick != 'Hard Computer'  and \
                    button.msgClick != 'Turn-by-Turn' and
button.msgClick != 'Multi-shot' and \
                    button.msgClick != 'Player vs Player':
                button.draw(gameScreen)


            radarScan = displayRadarScanner(RADARGRIDIMAGES,
INDNUM, SCANNER)
            if radarScan == False:
                pass
            else:
                GAMESCREEN.blit(radarScan, (cGameGrid[0][0][0],
cGameGrid[0][-1][-1]))

            RBlip = displayRadarBlip(INDNUM, BLIPPOSITION)
            if RBlip:
                GAMESCREEN.blit(RBlip,
(cGameGrid[BLIPPOSITION[0]][BLIPPOSITION[1]][0],

cGameGrid[BLIPPOSITION[0]][BLIPPOSITION[1]][1]))

            for explosion in EXPLOSIONS:
                explosion.draw(gameScreen)

#         computer.draw(gameScreen, cGameGrid)

        updateGameLogic(pGameGrid, pFleet, pGameLogic)
        updateGameLogic(cGameGrid, cFleet, cGameLogic)
    pygame.display.update()
```

Explanation:

Within the updateGameScreen function, I have added some conditions to determine which screen is displayed on the main game screen. To display the main menu screen the MENU flag should be set to true and the other two flags that are used to display the other 2 screens should be set to false, once this criteria the home screen is show, the same verification is done to display the turn based screen, except instead of requiring the MENU flag to be set to true the turnScreenStatus must be set to true instead, once this variable is set to true the other two are automatically set to false and the screen menu screen is no longer iterated within the main game loop and is filled in with a black background to display the new turn-based screen. Once all the fleets on either side of the board are completely destroyed the endStatus flag is set to true, and the other two are set to false, these combinations of Boolean values will display the end screen. The main game screen is displayed after the turn-based screen, this is because when a button is pressed on the turn-based screen at three of the screen variables are set to false and the main game UI elements are drawn to the screen and given their functionality. Within the main game portion within the function, I have added a background for the player grid and hidden the computer grid. All the buttons that are not required for the main game functionalities such as ‘Easy Computer’ and ‘Multi-shot’ have been hidden and a new button has been added alongside my existing buttons within the main game screen, this button is called ‘Bomb’ and will be used to shoot at a random 2x2 position on the enemies grid.

Changes in the Turn Functionality

Code*:*

```
def takeTurns(p1, p2):
    if p1.turn == True:
        p2.turn = False
    else:
        p1.turn = False
        p2.turn = True
        while p2.turn:  # Ensure the computer takes its
turn until it misses or changes turn
            if playerPlaying == False:
                p2.computerShoot(pGameLogic, pGameGrid,
EXPLOSIONS, turnBasedStatus)
            else:
                p2.playerShoot(pGameGrid, pGameLogic,
EXPLOSIONS, turnBasedStatus)
        if p2.turn == False:
            p1.turn = True
```

Explanation:

I have made some small changes within the takeTurns function which handles how the turns are distributed between the player and opponent. The playerPlaying variable is used to determine if the opponent is a computer or a real human, if the opponent is a computer the computerShoot function which is within both the hard and easy computer is called, however, if a human is playing then the Boolean variable is set true and the playerShoot function is called again against the main player’s grid so that player 2 can shoot at player 1’s grid.

Changes in Main Game Loop

Code*:*

```
# code determines the status of pygame and ends the session if
event type is quit
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                if deploymentStatus == True:
                    for ship in pFleet:
                        if
ship.rect.collidepoint(pygame.mouse.get_pos()):
                            sortFleet(ship, pFleet)
                            if not
ship.checkShipCollision(pFleet):
                                ship.active = True
                                ship.shipMove()
                else:
                    if player1.turn:
                        player1.playerShoot(cGameGrid,
cGameLogic, EXPLOSIONS, turnBasedStatus)
                        takeTurns(player1, computer)  # Switch
turns after player shoots
                    while computer.turn:  # Ensure the computer
takes its turn until it misses or changes turn
                        computer.computerShoot(pGameGrid,
pGameLogic, EXPLOSIONS, turnBasedStatus)
                        takeTurns(player1, computer)  # Switch
turns after computer shoots

                for button in BUTTONS:
                    if
button.buttonRect.collidepoint(pygame.mouse.get_pos()):
                        #if button.msgClick == 'ReDeploy':
                        #deploymentStatus =
button.redeploy(deploymentStatus)
                        if button.msgClick == 'Randomize':
                            if deploymentStatus == True:

button.randomizeShipPositions(pFleet, pGameGrid,
randomizeShipPositions)

button.randomizeShipPositions(cFleet, cGameGrid,
randomizeShipPositions)
                        elif button.msgClick == 'Reset':
                            if deploymentStatus == True:
                                # calling it twice cuz calling
it once don't seem enough to return it to the proper position
                                button.resetPosition(pFleet)
                                button.resetPosition(pFleet)
                        elif button.msgClick == 'Deployment':
                            deploymentStatus =
deploymentPhase(False)
                        elif button.msgClick == 'ReDeploy':
                            deploymentStatus =
deploymentPhase(True)
                            for ship in pFleet:
                                ship.returnToDefaultPosition()
                                ship.returnToDefaultPosition()
                            EXPLOSIONS.clear()
                            button.msgClick = 'Deployment'
                        elif button.msgClick == 'Quit':
                            running = False
                        elif button.msgClick == 'Radar Scan':
                            SCANNER = True
                            INDNUM = 0
                            BLIPPOSITION =
pickRandomShipPosition(cGameLogic)
                        elif button.msgClick == 'Bomb' and
deploymentStatus == False:
                            button.gridExplosion(cGameGrid,
cGameLogic, EXPLOSIONS)
                        if MENU == True:
                            if button.msgClick == 'Easy
Computer':
                                MENU = False
                                easyStatus = True
                                computer = EasyComputer()
                                playerPlaying = False
                                turnScreenStatus = True
                            elif button.msgClick == 'Hard
Computer':
                                MENU = False
                                easyStatus = False
                                turnScreenStatus = True
                                playerPlaying = False
                                computer = HardComputer()

                            elif button.msgClick == 'Player vs
Player':
                                MENU = False
                                easyStatus = False
                                turnScreenStatus = True
                                playerPlaying = True
                                computer = player2

                        if turnScreenStatus == True:
                            if button.msgClick == 'Turn-by-
Turn':
```

```
                                turnScreenStatus = False
                                MENU = False
                                endStatus = False
                                turnBasedStatus = True

                            elif button.msgClick == 'Multi-
shot':
                                turnScreenStatus = False
                                turnBasedStatus = False
                                MENU = False
                                endStatus = False


            elif event.button == 3:  # Right click
                for ship in pFleet:
                    if deploymentStatus == True:
                        ship.rotateShip()

            elif event.button == 2:
                printGameLogic()

        elif event.type == pygame.MOUSEBUTTONUP:
            for ship in pFleet:
                if event.button == 1:
                    sortFleet(ship, pFleet)
                    ship.offTheGrid(pGameGrid)
                    if not ship.checkShipCollision(pFleet):
                        ship.active = False

    for ship in pFleet:
        if deploymentStatus == True:
            ship.shipMove()

    updateGameScreen(gameScreen)
    if SCANNER == True:
        INDNUM += 1

    if playerPlaying == True:
        takeTurns(player1, player2)
    else:
        takeTurns(player1, computer)

    if deploymentStatus == False:
        endGameLogic(pGameLogic, cGameLogic)

pygame.quit()
```

Explanation:

I have added the home screen, end screen and turn screen buttons to the main game loop in case they fail or do not work within their functions, and have provided functionality for the new bomb button, the bomb button can only be pressed when the game has started, it is displayed during the pre-game state but cannot be used, when clicked it gridExplosion function which shoots at a random 2x2 position on the opponents grid. Towards the end of the main game loop the playerPlaying function decides whether to call the takeTurns function with the opponent as another human player or a computer, if playerPlaying is true the function is called using a human player, if this not the case the function is called using a computer. The final change has to do with ending the game, once the game has started the endGameLogic continuously checks the fleets on the both the grids, and once both the ships are destroyed the endGameLogic ends the game and declares a winner.

GameClasses.py

Changes within the Imports in GameClasses.py

Code*:*

```
import pygame
import random
from circularImport import redExplosion, blueExplosion,
fireExplosionList, explosionList, hitSound, shotSound,
missSound
```

Explanation:

From the circular import class, the 3 more variables are imported, these variables handle the sound functionality so they are named hitSound, shotSound and missSound.

Initialization of the Hard Computer

Code*:*

```
class HardComputer(EasyComputer):
    def __init__(self):
        # inherits easy computers attributes
        super().__init__()
        self.moves = []
```

Explanation:

To initialize the hard computer class, I have used the EasyComputer as a parent class and have inherited all the attributes and functions from it, the main attribute inherited from EasyComputer is self.turn which handles the turn functionality, the computerShoot function will be changed in the HardComputer class but the draw and computerStatus function will stay the same. An empty list called self.moves is created which all the possible places the computer can shoot after a successful hit.

Functions of the Hard Computer

Code*:*

```
def computerShoot(self, gameLogic, pGameGrid, EXPLOSIONS,
turnBased):
    if len(self.moves) == 0:
        validChoice = False
        while not validChoice:
            rowX = random.randint(0, 9)
            rowY = random.randint(0, 9)

            if gameLogic[rowX][rowY] == ' ' or
gameLogic[rowX][rowY] == 'O':
                validChoice = True

            if gameLogic[rowX][rowY] == 'O':
                gameLogic[rowX][rowY] = 'T'
                print('Hit')
                shotSound.play()
                hitSound.play()
                EXPLOSIONS.append(
                    Explosion(redExplosion,
pGameGrid[rowX][rowY], 'Hit', fireExplosionList,
explosionList, None))
                self.generateMoves((rowX, rowY),
gameLogic)
                if turnBased == True:
                    self.turn = False
                else:
                    self.turn = True
            else:
                if gameLogic[rowX][rowY] != 'T':
                    gameLogic[rowX][rowY] = 'X'
                    print('Miss')
                    shotSound.play()
                    missSound.play()
```

```
EXPLOSIONS.append(Explosion(blueExplosion,
pGameGrid[rowX][rowY], 'Hit', None, None, None))
                    self.turn = False
                else:
                    self.turn = True

    elif len(self.moves) > 0:
        rowX, rowY = self.moves[0]
        EXPLOSIONS.append(
            Explosion(redExplosion,
pGameGrid[rowX][rowY], 'Hit', fireExplosionList,
explosionList, None))
        gameLogic[rowX][rowY] = 'T'
        self.moves.remove((rowX, rowY))
        if turnBased == True:
            self.turn = False
        else:
            self.turn = True

    return EXPLOSIONS
```

Explanation:

The parameters within the computerShoot class has seen a new inclusion with the turnBased parameter, this parameter will be used to decide whether the multi-shot mode is being used or the turn-by-turn mode. If there are no potential moves within the moves list the computer shoots at a random position within the grid, same as the EasyComputer, however, once a move is found which would result in hitting a ship once a ship has been successfully hit a hit and the sound of a shot being fired is played, the grid logic grid updates the hit position with a 'T' and the Explosion visual effect is displayed at that position. A function within the HardComputer class is called when a ship is hit, this function is called generateMoves and this function checks the four grid cells surrounding the hit cell for another grid cell containing a ship and once the ship cell is found it is shot at until no grid cells containing around a hit cell containing a ship is found. When the turnBased Boolean value is set to true the game plays out as the usual turn-by-turn way however, if it's set to true which means the player chose to use the multi-shot functionality, the player gets another shot once a ship has been hit until they miss. If self.moves has a position with a ship within it that position is shot at by the computer and that position with the ship is removed so that it is not shot at again, and the same turn based logic for multi-shot and turn-by-turn is used within this portion of the code, explosions is returned to the main file and displayed onto the grid.

Code*:*

```
def generateMoves(self, coords, grid, lstDir=None):
    x, y = coords
    nx, ny = 0, 0
    for direction in ['North', 'South', 'East', 'West']:
        if direction == 'North' and lstDir != 'North':
            nx = x - 1
            ny = y
            if not (nx > 9 or ny > 9 or nx < 0 or ny <
0):
                if (nx, ny) not in self.moves and
grid[nx][ny] == 'O':
                    self.moves.append((nx, ny))
                    self.generateMoves((nx, ny), grid,
'South')

        if direction == 'South' and lstDir != 'South':
            nx = x + 1
            ny = y
            if not (nx > 9 or ny > 9 or nx < 0 or ny <
0):
                if (nx, ny) not in self.moves and
grid[nx][ny] == 'O':
                    self.moves.append((nx, ny))
                    self.generateMoves((nx, ny), grid,
'North')

        if direction == 'East' and lstDir != 'East':
            nx = x
            ny = y + 1
            if not (nx > 9 or ny > 9 or nx < 0 or ny <
0):
                if (nx, ny) not in self.moves and
grid[nx][ny] == 'O':
                    self.moves.append((nx, ny))
                    self.generateMoves((nx, ny), grid,
'East')

        if direction == 'West' and lstDir != 'West':
            nx = x
            ny = y - 1
            if not (nx > 9 or ny > 9 or nx < 0 or ny <
0):
                if (nx, ny) not in self.moves and
grid[nx][ny] == 'O':
                    self.moves.append((nx, ny))
                    self.generateMoves((nx, ny), grid,
'East')

    return
```

Explanation:

This function will be used to check the cells around a hit cell and append the list self.moves with that position which will later be shot at and removed. x, y represents the current hit position on the grid, and nx, ny represent the x and y value the cell block that is being checked for a ship, for north this would be one cell block under the current cell block, for south it would be one cell block above the current cell block, for east it would be one cell block to the right the current cell block, and for west it would be one cell block to the left of the current cell block. For north the cell block that is being checked must obey 2 conditions, one of these conditions is that it must be within the 10 by 10 grid which means the value must be more than 0 but less than 9, the second condition is that it must not already be within self.moves and should contain a ship which means its value on the logic grid must be ‘O’, it all these conditions are met the value is added to the self.moves list. The lstDir parameter prevents the function from immediate reversing the direction that has to be checked, if this preventive measure was not put in place the function would check back and forth between the same positions eventually leading to an infinite loop, so once a direction is checked the opposite direction is called recursively in order to prevent the function from checking the same position again and adding it to the list. This process is repeated for all south, east and west positions and the list with the updated positions is shot at within the computerShoot function.

Changes To The Easy Computer Function

Code*:*

```
def computerShoot(self, gameLogic, pGameGrid, EXPLOSIONS,
turnBased):
    validChoice = False
    while not validChoice:
        rowX = random.randint(0, 9)
        colX = random.randint(0, 9)

        if gameLogic[rowX][colX] == ' ' or
gameLogic[rowX][colX] == 'O':
            validChoice = True
        if gameLogic[rowX][colX] == 'O':
            gameLogic[rowX][colX] = 'T'
            print('Hit')
            shotSound.play()
            hitSound.play()
            EXPLOSIONS.append(
                Explosion(redExplosion,
pGameGrid[rowX][colX], 'Hit', fireExplosionList,
explosionList, None))
            if turnBased == True:
                self.turn = False
            else:
                self.turn = True
        else:
            if gameLogic[rowX][colX] != 'T':
                gameLogic[rowX][colX] = 'X'
                print('Miss')
                shotSound.play()
                missSound.play()

EXPLOSIONS.append(Explosion(blueExplosion,
pGameGrid[rowX][colX], 'Hit', None, None, None))
                self.turn = False
            else:
                self.turn = True

    return EXPLOSIONS
```

Explanation:

The changes to the computerShoot function within the EasyComputer, are as follows: the turn-based status Boolean flag decides whether the multi-shot functionality is used depending on the mode the user chose to play with when there is a hit, and when there is a hit a hit and shot sound play, for a miss a miss and shot sound is played.

Changes Explosion Functionality Within The Buttons Class

Code*:*

```
def gridExplosion(self, coordGrid, gameLogic,
EXPLOSIONS):
    # Choose a random cell in the grid
    randomRow = random.randint(0, len(coordGrid) - 2)
    randomCol = random.randint(0, len(coordGrid[0]) - 2)

    # Create a 2x2 area to shoot
    for i in range(2):
        for j in range(2):
            cellX, cellY = randomRow + i, randomCol + j
            if gameLogic[cellX][cellY] != ' ':
                if gameLogic[cellX][cellY] == 'O':
                    gameLogic[cellX][cellY] = 'T'

EXPLOSIONS.append(Explosion(redExplosion,
coordGrid[cellX][cellY], 'Hit', None, None, None))
                else:

EXPLOSIONS.append(Explosion(blueExplosion,
coordGrid[cellX][cellY], 'Miss', None, None, None))
            else:
                if gameLogic[cellX][cellY] != 'T':
                    gameLogic[cellX][cellY] = 'X'

EXPLOSIONS.append(Explosion(blueExplosion,
coordGrid[cellX][cellY], 'Miss', None, None, None))

    return EXPLOSIONS
```

Explanation:

This function within the buttons class will be used to create a random 2x2 explosion on the grid. A random row and random column value is found using the randint functionality and limiting the range of values to the end of the row and column minus 2, the reason I have subtracted 2 from the size of the grid is because if an explosion is shot at the right or bottom edge of the grid the explosion hit markers would appear outside to grid so, to prevent the explosion from not fitting into the grid the size of the grid is subtracted by 2. A for loop has been used to iterate over a 2x2 area starting from the cell that is selected at random and moving over to the next cell within the row and column. If that area contains a ship (‘O’) the game logic updates that position to a hit (‘T’) and an empty position on the logic grid is updated to a ‘X’ indicating a miss, the explosions are appended at the shot positions and returned to be displayed within the main game screen.

CircularImport.py

Changes To the Shared Utility Variables

```
import pygame

pygame.init()
pygame.mixer.init()

cellSize = 50
screenWidth = 1260
screenHeight = 960
ROWS = 10
COLS = 10
hitSound =
pygame.mixer.Sound('assets/sounds/explosion.wav')
hitSound.set_volume(0.05)
shotSound =
pygame.mixer.Sound('assets/sounds/gunshot.wav')
shotSound.set_volume(0.05)
missSound =
pygame.mixer.Sound('assets/sounds/splash.wav')
missSound.set_volume(0.05)
```

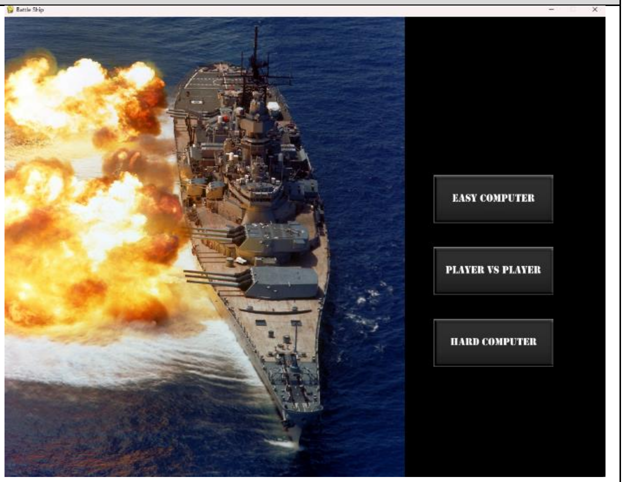
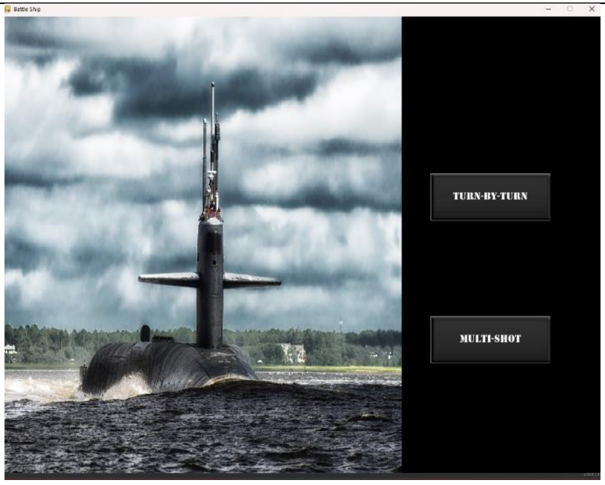
Explanation:

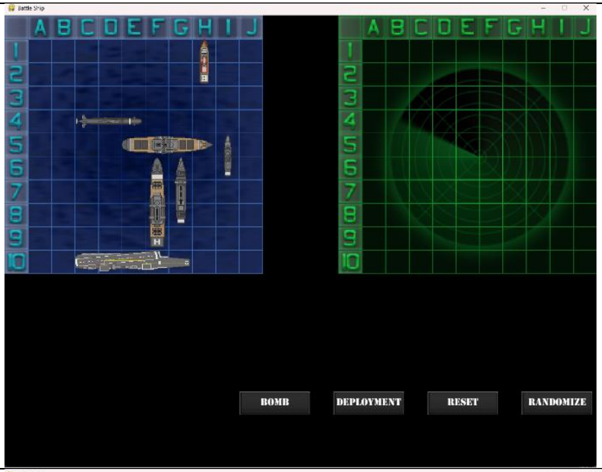
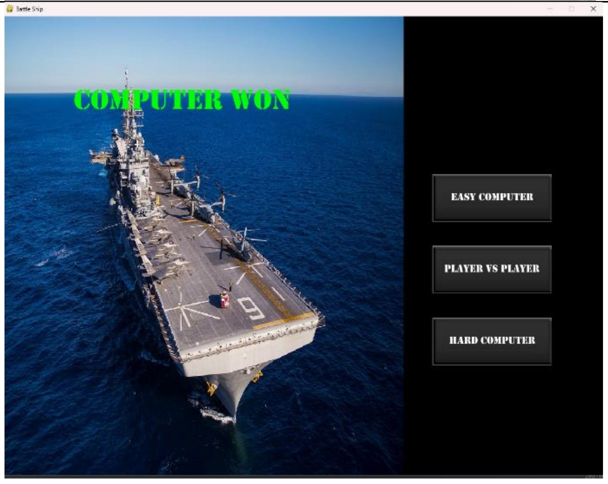
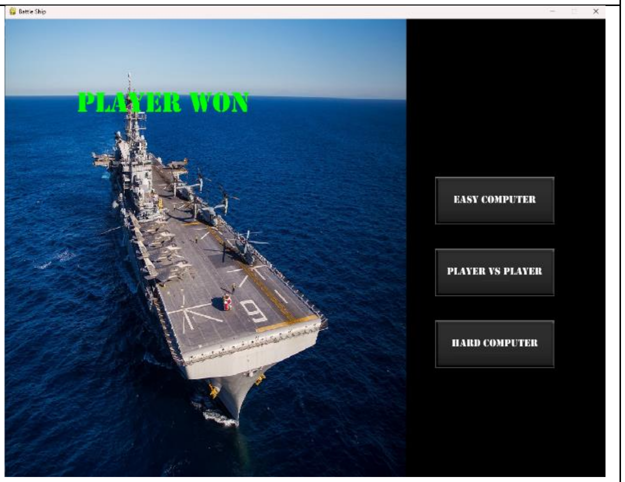
The only new additions within the circularImport file is the inclusion of a shot, hit and miss sound and the volume is adjusted to a suitable value.

UI Elements of Prototype 2

There have been a considerable number of changes that I have made to the UI from prototype 1, these changes I have made has given the user more choices on how they want to play the game.

The Additions to the UI can Be Seen Below:

UI Identification	Application
This is the home screen, this is where the user chooses whether they want to play with the easy computer, another human or the hard computer.	 Battle Ship EASY COMPUTER PLAYER VS PLAYER HARD COMPUTER
This is the turn-based screen, this is where goes after choosing who they want to play at the home screen, the user can choose to either play 'turn-by-turn' against their opponent or play with a 'shoot until miss mode' against their opponent.	

I have added a background image to both the player and computer grid, the background can be used to get the coordinates of the ships, and for the computer grid the ships are hidden using a faint static radar image. A new button has also been created the 'Bomb' button, this button cannot be used until the ship deployment phase is over.	 Battle Ship A B C D E F G H I J 1 2 3 4 5 6 7 8 9 10 BOMB DEPLOYMENT RESET RANDOMIZE
I have created an end screen where the winner is announced and the user can choose to play again against the same opponent or a different one, the turn-based screen is shown after this one and then the main game screen.	 Battle Ship COMPUTER WON EASY COMPUTER PLAYER VS PLAYER HARD COMPUTER
This is how the winner is announced on the end screen when the player wins.	 Battle Ship PLAYER WON EASY COMPUTER PLAYER VS PLAYER HARD COMPUTER

Prototype 2 Testing

The way I will test prototype 2 will slightly differ from how I tested prototype 1. I will still use a survey however, I will ask my stakeholders to evaluate all the UI features, functions and responses within my game and provide a comment to understand why they chose to evaluate a specific part of my battleship solution the way they did, I have requested my stakeholders to label any bugs they find based on its severity, a bug can range from minor to critical, and the bug or issue will be pre-faced with its label before it is explained.

The Table Below Provides All The Features that are Being Evaluated

Classification	Feature
Robustness	Explosion smoothness
Robustness	Switching screens
Usability	Home screen
Usability	Turn-screen
Usability	End screen
Usability	UI Satisfaction
Functionality	Ship mechanic
Functionality	Shooting mechanic
Functionality	Radar power-up
Functionality	Bomb power-up
Functionality	Buttons
Functionality	Easy computer
Functionality	Hard computer
Functionality	Local CO-OP
Functionality	Turn-based mode
Functionality	Multi-shot mode
Functionality	Game ending logic

The responses I have gotten from each respondent will be shown individually below:

- **Oscar's Feedback**

Feature	Does it work as intended?	Feedback & Stakeholder Comment
Ship mechanic	Yes	It's smooth and snappy, no errors can be identified with this mechanic.

Shooting mechanic	Yes	It provides an accurate and clear hit and miss indicator.
Explosion Smoothness	No	A new bug has emerged that was not present in prototype 1 and that is the ships fire and explosions are being animated too quickly, the explosions cannot be seen when a ship is hit.
Buttons	Yes	All the post and pre-game buttons work properly, however there is an issue with the player vs player mechanic when I choose that mode.
Switching Screens	No	CRITICAL BUG MUST FIX: Sometimes when I choose a mode on the home-screen too quickly the turn screen does not appear and at other times it does.
Home screen	Yes	It has a nice polished/stylish finish and all the modes I'd hoped to be within this prototype are present.
Turn-screen	Yes	When I choose a mode between the turn-by-turn mode and the multi-shot mode, I get what I'm expecting every time.
End screen & Game ending logic	Yes	The end screen shows without fail when all the fleets on a grid are destroyed and the winner is declared in large visible text, the buttons on the end screen work as intended.

Radar power-up	Yes	It works perfectly fine just as prototype 1.
Bomb power-up	No	The bomb power-up works fine for the most part, but as a minor bug and that bug is that it shoots at spaces that were already shot at and updates a hit ship with a missed marker if it hits the same ship twice.
Easy computer	Yes	The easy computers works just as well as it did in prototype 1.
Hard computer	Yes	The hard computer provides a much larger challenge than the easy computer and feels especially tense in the multi-shot mode, the AI for this game provides a fun yet challenging game.
Local CO-OP	No	CRITICAL BUG: It does not work and crashes once the first shot has been shot
Turn-based mode	Yes	The turn-based mode switches from the player to the computer and from the computer to the player after each turn without any issues or errors.

Multi-shot mode	Yes	There's no errors to it, it allows you to shoot until you miss and allows the same condition for your opponent.
UI Satisfaction	Yes	The UI is smooth and provides a much more interactive experience when compared to prototype 1.

- **Lily's Feedback**

Feature	**Does it work as intended?**	**Feedback & Stakeholder Comment**
Ship mechanic	Yes	All the ships can be placed without any bugs
Shooting mechanic	Yes	The proper turn is allocated after every shot, I have not identified any errors with the shooting mechanic.
Explosion Smoothness	No	MINOR BUG: The explosion mechanic works fine however, there seems to be little to no delay between each animation image for both the explosion and the fire.
Buttons	Yes	All the buttons execute the functionality that I would expect from them.

Switching Screens	No	CRITICAL BUG: the turn-based screen does not show up every time after I choose which opponent I want to play against, it instantly directs me to the main game without having me choose the mode I want to play with. MINOR BUG: The buttons on the turn-based screen have to sometimes be clicked multiple times to start the game.
Home screen	No	MINOR DISSATISFACTION: The home screen UI does not provide any explanation on how the game is to be played and what each mode game mode consists of, this could especially be confusing to a new player who does not know how to play battleship.
Turn-screen	Yes	The turn-screen is pretty self-explanatory so the functionality provided by it seems fine, however, I do not like that the buttons have to be clicked multiple times to finally change to the main game screen.
End screen & Game ending logic	Yes	The game seems to end properly whenever all the fleets within a grid are destroyed, and the winner announced is accurate and clear every time, and I like that there's an option to play again that works properly.
Radar power-up	Yes	No changes were made to it from prototype 1 which is fine, as it seemed to work as intended within prototype 1.
Bomb power-up	No	MINOR BUG: The bomb power-up seems to update hit ships on the grid with a miss indicator, once it comes into contact with it, and grid cells that have been shot at should not be shot at again at random by the explosion, other than these 2

	🟥	issues it seems fine and works as I had requested.
Easy computer	🟩 Yes	It has not changed for prototype 1, and the AI within prototype 1 worked just fine.
Hard computer	🟩 Yes	Prototype 1's computer seemed stale and boring, however the difficulty provided by the hard computer makes the game both engaging and fun, the hard computer definitely increases the stakes when the game is played.
Local CO-OP	🟥 No	CRITICAL BUG: the player vs player mode causes the game to freeze once a shot is fired, this should not be present if it's decided to release the game to the public as this bug would make the game unplayable with a human opponent.
Turn-based mode	🟩 Yes	The turns between the player and computer are transitioned very smoothly, however it would be nice it there was a delay before the computer shoots.
Multi-shot mode	🟥 No	MINOR BUG: When 2 ships are connected, it misses a position sometimes, this could be due to ships being in both the cells surrounding a hit position and it gives one priority and ignores the other.

UI Satisfaction	No	It seems there could be more additions to be more user-friendly, because if this version is released to the public, it may be difficult for some users to understand how to play the game.

- **Jack's Feedback**

Feature	Does it work as intended?	Feedback & Stakeholder Comment
Ship mechanic	Yes	The functionality seems work just as well as it within prototype 1.
Shooting mechanic	Yes	It updates the correct grid position with a new accurate status.
Explosion Smoothness	No	MINOR BUG: The explosion and fire animation does not seem to have any delay between each frame, this makes the explosion instant, which means it cannot be seen and the fire seems to animate too quickly.
Buttons	Yes	The buttons in-game seem to work fine and seem to be error free.
Switching Screens	No	CRITICAL BUG: Turn mode selection screen does not show up when the opponent is selected too quickly, I have to wait a few seconds before selecting which opponent I want to play against for the turn-based screen to show up.

Home screen	Yes	The UI elements are placed in an orderly fashion which makes all the buttons clear and visible, and the UI seems to be visually appealing.
Turn-screen	No	MINOR BUG: I have to consistently click on the turn buttons multiple times to redirect myself to the actual game-screen, this problem is especially present with the multi-shot button, and could make certain users impatient.
End screen & Game ending logic	Yes	The feature that announces the winner and loser seems like a good addition and the end-screen shows and does exactly what I'd expect from it.
Radar power-up	Yes	The radar does precisely what it was designed to do, it searches and displays a ship in a satisfactory and clear manner.
Bomb power-up	Yes	The bomb explosive power-up shoots at a random 2x2 position on the grid, this purpose is implemented successfully.
Easy computer	Yes	It does as you'd expect and does not bring up any errors.

Hard computer	Yes	The AI for the hard computer has been implemented brilliantly, there seems to be no errors and paired with the multi-shot mode provides an extremely difficult experience which is to be expected from the hard computer, there seems to be a good amount of complexity when compared to the easy computer.
Local CO-OP	No	CRITICAL BUG: The game crashes whenever I shoot at my opponents grid, this an urgent issue that must be fixed immediately as this error could prove as a major deterrent to the public if the game is released.
Turn-based mode	Yes	I could not find any errors with the turn switching mechanic so, I will approve of this feature.
Multi-shot mode	Yes	I was told this mode would allow you to shoot until you miss, this is exactly how it works and there seems to be new errors that arise from this game mode.
UI Satisfaction	Yes	The interactive UI has large buttons that are clearly visible and the text provides context to what entails the player when they click on the button.

- **Omar's Feedback**

Feature	**Does it work as intended?**	**Feedback & Stakeholder Comment**
Ship mechanic	Yes	Seems to work like version 1, so I wouldn't expect any errors.
Shooting mechanic	Yes	The turns switch properly, and seem to be without error.
Explosion Smoothness	Yes	The hit and missed ships seem to feature a sound now, this sound makes the game more user-friendly and more interactive when played.
Buttons	No	CRITICAL ERROR: The buttons carry out the correct functionality except the player vs player button, this button crashes the game when a shot is played.
Switching Screens	No	MINOR BUG: The screen where you choose whether you want to play using the turn-by-turn mode or multi-shot mode does not appear half the time.
Home screen	Yes	All the buttons work fine and the UI seems fine.

Turn-screen	Yes	The buttons work as intended and the UI buttons have clear text indicating what they'd do.
End screen & Game ending logic	Yes	A winner is declared in an clear fashion, however I feel as if the loser should also be declared if the player vs player mode worked.
Radar power-up	Yes	No changes were made to it since prototype 1.
Bomb power-up	No	MINOR BUG: Hit positions are updated with a miss marker/explosion, this does not affect the outcome of the game however, does provide an incorrect visual indicator as they overlap over each other.
Easy computer	Yes	The easy computer seems to be what the AI from the first prototype 1, this rebrand seems accurate as it was too easy to defeat.
Hard computer	Yes	The hard computer adds to the difficulty of the game which seemed to be missing in prototype 1, within the multi-shot mode if you line your ships next to each other you could lose in 1 turn, this seems to make the hard computer too difficult, in some cases however as far as functionality goes it works fine.

Local CO-OP	No	CRITICAL BUG: I cannot play with another human nor shoot at the grid on the left as it crashes once the first shot is fired, the game cannot be released without fixing this issue in future prototypes.
Turn-based mode	Yes	Turns are switched successfully without any faults or errors.
Multi-shot mode	Yes	There seems to be no error and it works as anticipated.
UI Satisfaction	No	The Player vs Player mode when selected causes the game to crash when a shot is fired, but apart from that it pretty much works fine.

Validation For The Key Elements

Feature	Elements Being Validated	Status of The Validation	Comment
Explosion smoothness	The player and computer hit/miss indicators and animation.	Mostly Validated: There is a new minor issue, that does not drastically change the game.	Animated related issues have been found that require further testing.
Switching screens	The transition between one screen and the next.	Not Validated: There seems to be a common issue that occurs often.	Critical issue: some screens do not appear every time.
Home screen	The UI elements, usability and satisfaction with the home screen.	Mostly Validated: stakeholders are not satisfied with the content of the home screen.	The home screen must be more user-friendly in future prototypes.
Turn-screen	The UI elements, usability and satisfaction with the turn-screen.	Validated: There are no issues and the screen works perfectly fine.	The turns buttons and UI carry out their intended purpose.
End screen/ Game ending logic	The UI elements, usability and the transition from the game to the end screen.	Validated: The contents of the end screen seem adequate and the winner declaration feature is satisfactory to the stakeholders.	The game ends properly, and provides the correct winner, the end screen works as intended.
UI Satisfaction	The overall contentment with the UI.	Partially Validated: Some UI features do not work as intended.	More testing is required to fix the UI.
Ship mechanic	The movement/placement/overlapping of the ships.	Validated: The ship mechanic provides no errors with overlapping or positioning.	The ships intractability works as intended.
Shooting mechanic	Switching between the player and opponents turns.	Validated: The shooting mechanic provides accurate indicators.	The shooting mechanic works as intended.
Radar power-up	The radar's animation and functionality.	Validated: the radar is up to my stakeholders standard and does not have any errors.	The radar animation and functionality seems fine.

Bomb power-up	The overlapping of hit/miss positions and any errors found with the bomb.	Not Validated: The bomb shoots at positions that are already shot at and updates cells with an incorrect indicator.	The bomb power-up has some critical issues that must be fixed.
Buttons	The responsiveness and feedback of the buttons.	Mostly Validated: The buttons provide an expected result expect for the player vs player buttons.	Some of the buttons do not provide a desired outcome.
Easy computer	The shooting logic of the easy computer.	Validated: No issues were found with the easy computer.	Easy computer's rebrand is acceptable.
Hard computer	The shooting logic of the hard AI.	Validated: The logic of the hard computer is sufficient.	The hard computer is up to standard.
Local CO-OP	The status of the player vs player mode and how the game plays out when it's used.	Not Validated: Does not work at all.	Critical error, fixing this error should be a top priority.
Turn-based mode	Switching between the player and opponents turns.	Validated: Turns are switched accurately.	The turn-based mode without errors.
Multi-shot mode	Switching between the player and opponents turns.	Validated: Turns are switched accurately.	The multi-shot mode works as intended.

Prototype Review at Each Iterative Stage

The table below indicate the validation of each feature based on the color of the text, the color of the validation matches the validation status of the table above, I will review the changes each feature has gone through from the first prototype to the next and comment on it for future reference.

Feature	Status of The Feature for Prototype 1	Status of The Feature for Prototype 2	Comment
Explosion smoothness	There was a smooth explosion and fire explosion, and each frame was animated properly.	A new minor bug which has occurred to the animation of the explosion and fire seem to fire had been created.	This error could possibly be fixed if a larger delay is added to each frame.
Switching screens	Was not implemented yet.	The screen switching works for the most part however if the program isn't given adequate time to load the turn-screen won't show up.	A small delay can occur after the game starts to load the assets properly.
Home screen	Did not exist.	The home screen works fine however, it is lacking in a few features.	A tutorial menu could be added to guide new players.
Turn-screen	Did not exist.	The turn-screen works fine when it shows up and carries out its functionality properly.	The turn-screen does not require any further research.
End screen/ Game ending logic	Did not exist.	The end screen and the logic to get to the end screens works flawlessly as of prototype 2.	The game is ended properly and does not require any changes.
UI Satisfaction	The UI was not up to my stakeholders standards and had to be polished.	Most of my stakeholders find it satisfactory however, they have brought up a few minor complaints with it.	Further research needs to be done to make the UI slightly better.
Ship mechanic	The ship mechanic that I had created during this stage worked perfectly fine without any errors.	No changes were made to the ship mechanic in prototype 2, so it works fine.	No more research is required as no new errors have occurred in prototype 2.

Shooting mechanic	The shooting mechanic provided accurate hit and miss indicators.	There have been so changes to the shooting mechanic in prototype 2.	It works fine and no changes need to be made.
Radar power-up	The radar power-up's animation worked fine and the blip that appeared on the screen was accurate.	No new errors have made an appearance for the radar power-up in prototype 2.	The number of usages the radar has could be limited in a future prototype.
Bomb power-up	Did not exist.	The bomb power-up contains a few minor errors.	The bomb power-up should shoot slightly further from the previous position that was shot at by the bomb.
Buttons	All the buttons worked as intended.	The addition of the 'Player vs Player' has created major error as it does not provide the right functionality.	The buttons 'Player vs Player' could be removed in the issue is not resolved in future iterations.
Easy computer	The computer was too easy and did not provide a challenge for my stakeholders.	No changes were made from prototype 2 except for rebranding, so it works fine.	The easy computer could be slightly more difficult but has fulfilled what my stakeholders wanted to see in it.
Hard computer	Did not exist.	The hard computer has satisfied my stakeholders so no changes are to be made to it and it is complete.	The hard computer has been approved by my stakeholders and does not require any changes.
Local CO-OP	Did not exist.	Major critical error: The local co-op as of prototype 2 does not work as intended and crashes the game.	The 'Player vs Player' mode should be given priority in future iterations.
Turn-based mode	The default shooting mode was set to the turn-based mode for prototype 1, as I had not created any other shooting modes yet.	The turn-based mode is optional within prototype 2 as a new mode has been introduced.	The turns are switched successfully has no changes are required.
Multi-shot mode	Did not exist.	The multi-shot mode works perfectly fine but can make a game a little difficult at times.	The multi-shot mode does not need as debugging as it has been integrated successfully.

Evaluation

Reviewing the Success Criteria

If I had ended the project with prototype 1, I would consider it a success however, once I delved deeper within prototype 2 it seems the battleship solution has not been 100% successful, but what I sought out to do when I came up with the battleship idea was to make a difficult AI which provides re-playability and this criteria has been fully met, this means I am sort of satisfied with the final solution however, I would not be confident to release it to the general public, there are definitely points where the battleship game can be improved and these will be stated later on within the evaluation.

Below is a review of the success criteria with each prototype version:

Classification	Success Criteria	Status in Prototype 1	Status in Prototype 2	Further Development Notes
Robustness	The functions must successfully work with each other.	Criteria Met: All the functions and features work alongside and collaborate with each other to provide a working solution.	Criteria Partially Met: Some of the features can lead to a major error that crashes the game.	For the future versions a warning could be added below the buttons to prevent a user from encountering a crash, or the Player vs Player feature could be completely removed until it is fixed.
Robustness	The game should not provide any errors.	Criteria Met: None my errors had discovered any errors when testing prototype 1 of the game.	Criteria Not Met: There are some major critical errors present within prototype 2 which makes it unsuitable to be released to the general public.	The crashing error could lie within the turn-based mechanic or an improper opponent assignment, these 2 mechanics should be fully tested with the 'Player vs Player' mechanic in the future to encounter where the error is caused.
Usability	The UI must be user-friendly	Criteria Mostly Met: A few of my stakeholders provided a minor complaint with the UI however, nothing too major was addressed.	Criteria Partially Met: Most of my features were labelled accurately however, some new users may have some difficultly grasping the concept of the game.	A new tutorial button could be created that explains how the game works for users who are new or unfamiliar with battleship.
Usability	Outputs must be accurate to the inputs.	Criteria Met: When an input is provided by a user an accurate and expected output is received by them.	Criteria Not Met: With the addition of many new features some interactable features do not provide their desired output.	The user should not be able to provide an input to the features that do not provide a proper output until their functionality is fully fixed.

Usability	Visual indicators should be clear and understandable.	Criteria Met: All of my visual indicators were clear, accurate and smooth.	Criteria Mostly Met: Some of the visual indicators such as the explosion is clear however, seems too rapid and fast to understand for some users.	Since the issue lies within the animation of the explosion and not the accuracy of the visual indicators, the way the animation frames are shown should be debug.
Functional	A winner and a loser must be provided upon ending the game.	Criteria Not Met: There was no end screen so I could not provide a winner or loser when all the fleets are destroyed on a grid.	Criteria Met: The winner is declared in large bold green text upon ending the game, so the criteria is met.	No further research or changes are required in future prototypes since this feature works as expected.
Functional	The AI must not be predictable.	Criteria Met: The AI is randomized in prototype 1, so it does not follow a set pattern.	Criteria Met: The same randomized logic for the AI was used in prototype 2 as prototype 1 but a larger punishment is given when a ship is hit.	The AI might be too unpredictable currently to feel like a real human being, I believe a few of the patterns of the AI should change in future prototypes to give it a more ‘human feeling’.
Functional	Ship placement should not leave the boundaries of the grid.	Criteria Met: The ship is returned to its original position when placed outside the grid.	Criteria Met: No changes were made to the boundary detection mechanic for the ships as it worked fine in prototype 1.	The ship placement seems fine and should not be changed in future prototypes.
Functional	AI must have a sense of difficulty.	Criteria Not Met: The AI did not have any complexity other than the randomized shooting so it was too easy to defeat for most players.	Criteria Met: The AI in prototype 2, destroys an entire ship once it hits a ship, this makes the game several times for difficult.	The easy AI might be too easy and the hard AI might be too difficult for some users, I believe in future iterations there should be an AI which is in the middle-ground of these two.

Prototype Reviewing The Functional Requirements

Most of the functional features I had brought up in the success criteria in the analysis section have been successfully met. Below I have outlined each success criteria and assessed them by stating if they have been met and why I feel the way I do about the functional criteria.

Ship Placement

The ship placement mechanic has by far exceeded my expectations, prior to programming the game I had not been much aware of validated an element in a game, however after I had made my game, I had perfected by ship placement mechanic as to not cause any errors and take care of minor annoyances such as ships overlapping each other when moved around the grid, the criteria has been fully met without any issues.

Shooting Mechanic

The shooting mechanic I had created for my game had been incredibly precise and did not cause any false shots or shoot at the wrong positions on the grid or shoot outside the grid. I was initially worried about how the math would be performed to determine a position that had been shot, however upon developing this mechanic I had learned to breakdown the math into smaller pieces to understand it and create an accurate shooting mechanic that only contained the shots within their selected grid cells, the criteria was successfully implemented.

Randomize ship Positions

The randomization of the ships has been added to the game through the usage of buttons, the two main rules I had to overcome when making my ship randomization mechanic was to not allow the ships to overlap and keeping the ships within the grid, these 2 objectives had to be achieved to make the randomize mechanic work, I had to manually work out the math to prevent the ships from overlapping, once the overlapping problem had been fixed the other issue of keeping the ships within the grid was quite simple to achieve, so with the competition of the second validation I had successfully met this criteria.

Reset ships

I had created initial positions for each of the elements of the game as to set all the elements to their default position when the game is reset, these initial positions had come in handy later on as it made it quite easy to make the reset functionality, all I had to do to implement the reset functionality was clear the grid and set all the elements to their default positions so, this functional criteria had been fully met.

Game Start initializer

In order to initialize the start of the game I had to separate the code into pre-game and post-game start segments, through this distinction I had to create a Boolean flag that could smoothly switch between each game state, once this game flag was created and added properly, I just had to trigger it on and off through a series of buttons, this approach I had taken to initialization of the game had been it easier to implement this functionality than I had initially projected. This criteria had fully met.

Quit Game

I had tackled quitting the game functionality quite later on in my development cycle as it was not a priority at first, since you can close a game window by clicking the cross button on the top right of the screen, however my stakeholders had recommended to create finish off this feature as it gave the game more of a polished look, the creation of this feature had been straightforward, all I had to do was disable the running flag and the pygame utility functions to make it work, once I had learned this it was clear on how I had to implement this feature. The quit game criteria had been fully met.

End game/Restart

Toggling between the main game screen and the end screen had been quite difficult as it required grasping the status of each ship and ending once a certain game state had been achieved, keeping track of the game state had been quite troublesome and took a decent amount of debugging to finally implement, once I had found out how to end the game and declare a winner switching game windows was quite straightforward as I had already learned it with the home screen, however resetting the game did not prove this difficult as it required me to change the state of a few game assets. Once I had done this the criteria had been successfully met.

Radar Scan Button

The game radar was incredibly challenging to implement as this was my first-time handling and working with the pygame animation mechanic, integrating animation with functionality was an entirely new concept to me as I had never done it before. I had used a few tutorials and websites to learn how to integrate seem-less animation with functionality in order to make it work smoothly and the results are better than I had projected when I had come up with the idea. The criteria was met successfully.

Ship Rotation

Rotating the ships is a simple concept by itself, however when it came to placing it on the grid without making it leave the boundaries of the grid I encountered some difficulty initially but as I gathered a larger understanding of how ships responds to the values within the grid it became increasingly easier to keep the ship within the grid and eventually the ship stayed within the grid without fail, however I did notice a small problem with the ship rotation that my stakeholders did not see, the problem has to do with the guns being displayed when the ship is rotated horizontally, the guns disappear, however the problem is so miniscule that it can go without notice. The criteria was mostly met.

Status of Ships

For prototype 1 I did not store nor update the status of the ships other than resetting the game, because I had not yet created the end game logic which checks all the fleets on both the grids to determine a winner, so keeping up with the status of ships did not seem necessary in prototype 1, however as I went on to develop prototype 2 it had become more evident that I needed the status of the ships to completely reset the game and declare a winner, once I had achieved in developing this functionality the criteria was met.

The AI opponents

The AI opponent was not a challenge in creating for prototype 1, however for prototype 2 I had to increase the complexity of the AI by several factors, so I brain stormed the idea of destroying the ships when a hit was found, and this is how I felt when creating both the AI's:

- **Easy Computer:** The easy computer acted sort of as a placeholder opponent in prototype 1, I used it as the building blocks for a more complex hard AI, however later on I liked the concept of having multiple modes so I kept the easy computer as an opponent for new players to understand how the game works whilst not making the game too difficult, the criteria for the easy computer was met in prototype 1.
- **Hard Computer:** The hard computer was an improved version its predecessor the easy computer, once I had come up with the idea of destroying all the ships when the AI had found one implementing it was not as easy as I had expected, I had encountered multiple errors and had an increasingly hard time creating it with an error that repeatedly checked the same grid cell infinitely, once I had spent some time debugging the code I had learned to exclude certain grid cells while others were being checked and this fixed a major error with the hard computer, once this error was fixed the criteria for the hard computer was fully met.

All of my functional success criteria were fully met as they had been made for the base plate of the game and not prototype 2, if my functional success criteria had included prototype 2, I would not have a met all of my success criteria and may have also have some criteria's that would not have been met.

Prototype Reviewing The Usability Feature Requirements

Most of the usability features have already been tackled and explained within the success criteria table however, I have left some out as I had given the idea for them during the development of prototype 2.

One of the usability features I had not met had to do with the game not being user-friendly, for instance if a new player were to play my game with no prior understanding of battleship, they would not understand the mechanics or how to play it, for future development version I would like to add a tutorial screen and a back button on each of the screens, as my game currently stands once a player has chosen a mode they want to play with they cannot go back to the prior screen to change the mode they want to play in until the game has ended or until they have closed the window and reset the game, this reduces the user-friendliness for some users and should be worked on in future prototypes.

The way the ship snaps to the grid has far exceeded what I had initially projected from the final outcome, the ship snaps with precision and accuracy, this makes it quite difficult to mistakenly place your ship at the wrong position and another feature that had a few errors until the end of the development of prototype 1 was to not allow the ships to be placed on top of each other, this feature seemed quite simple to add in theory but had proved to be difficulty when I started to implement it. This usability feature does not require any future research as it works just fine.

The turn-screen has been created properly but an 'annoying' bug with the turn-screen is that it does not show up most the time, this bug is quite difficult to debug as it is not necessarily an error but does feel like one, I could possibly make the turn-screen show up every time in future prototypes by providing a delay for some assets and make the turn-screen load in first but I believe that could not be a fool proof solution as the error could lie somewhere else, but whatever the case this bug should be given priority when moving onto future iterations. This feature has been partially met as it does technically work but does not work every time, so I cannot say I have met this usability criteria.

Limitations

There are quite a few limitations to my current battleship solution, these limitations include not enough user-friendliness, turn-screen not showing up, player vs player mode not working, no intermediate level difficulty to the AI, fast explosion animation that reduces the fidelity of the game and no mini-max algorithm to the hard AI as I believe this could increase the replay value of the game.

Why are these Limitations Present?

These limitations are largely present because of time constrictions, especially with the mini-max algorithm, the complexity with the current hard AI seemed enough and took a large amount of time to implement as a large amount of research had to be done to create it, the intermediate level has not been created as I envision a intermediate level would look like a combination of the randomized shooting of the easy computer and a mini-max algorithm, so this would not be possible with my current level of knowledge on a mini-max algorithm. I have spent a large amount of time debugging the player vs player mechanic, and just could not pinpoint the error, however I have a few hunches on what it could be. The user-friendliness of the solution is definitely a weak point as new players will not be able to understand how the game is played, the reason I did not implement a tutorial is because my stakeholders had not requested one initially however, as time went on and most the functionalities were developed in prototype 1 the focus had slightly shifted to the UI in prototype 2, so I did not have enough time to create a tutorial. As for the turn-screen not showing up half the time, the reason this could not be fixed is similar to why I couldn’t finish developing the player vs player mechanic and that reason is that I could not find the root cause of the error as it did not necessarily present itself as an error so, I had a harder time debugging this bug. And for my last limitation the explosion animation, the reason this could not be fixed is because I did not notice that the animation did not seem right until my stakeholders pointed it out during the testing phase of prototype 2.

Potential Improvements to my Limitations

For future iterations I could fix most these issues by conducting further research and broadening my understand of game development with pygame, for the player vs player mode not working the crashing cause could potentially lie within the turn mechanic, as I have set different rules when a player is playing, but these rules could possibly be incorrect and different criteria’s need to be put in place to make it work, another cause could be that the computer objects may still be assigned to some of the variables when the player mode is played. To create a mini-max intermediate mode, I could research and learn how to implement a mini-max algorithm into my battleship solution, however this should be secondary priority in future versions, fixing the bugs must come first and another bug that is quite bothersome is the turn-screen not showing up, this bug could be fixed by adding some extra validation before the game screen is shown and testing different screen scenarios to figure out what scenario works. To create a tutorial in future version I need to communicate with my stakeholders on how they would like to see this UI feature implemented and create a back button and tutorial button with their guidance.

The Development Stages

Prototype 1

Before Developing:

Whilst I was breaking down the functionalities within the game and gathering feedback on how I should implement them, I envisioned the final product to have more power-ups and less bugs however, this seems unrealistic now as I did not have much experience with pygame, but I was quite vary of the UI, since I had little to no experience with front-end development of a game, as I had stated within my analysis section, but I was not that worried when it came to programming the functionality of the game as I had had previous experience with back-end development, this was also another reason I had done a lot of research when it came to game power-ups, however I did not take into account that back-end development with other types of software would not translate to back-end experience with game development, this would later prove detrimental as I had encountered a bunch of errors during the development process, however by the end of the development phase, I had solved most of them.

During Development

When I began developing the game it seemed quite daunting to program an entire game from scratch, however once I had found an asset folder for my battleship solution, the final vision of the game had become more clear, and I became more confident on meeting my stakeholders criteria's. During the development of the ship mechanics, I had not expected to have as many rules to the ships as I had by end of developing prototype 1, it felt as if once I had fixed a certain problem with the ship placement there were 10 problems awaiting me, but I eventually overcame this hurdle just to encounter another problem with the development of another feature such as the explosion or the radar, the explosions did not provide too many errors during the development phase, however this was because a lot of planning had gone into developing them as it was no easy feat and the radar had too many elements it, specifically the blip, the blip functionality of the blip which was finding a random ship position was quite simple to create after making the randomize ship mechanic as it followed the same logic, however animating it to make it look visible and not have it drag out for too long at the same time was quite difficult to implement, but by making it complement the animation of the radar and using the radar animation as a measure of how long the blip should show on the grid it became slightly easier to animate and the final product seemed to work as intended. Smaller problems such as initializing the start of the game and changing the buttons did not prove too difficult since most of them required a single Boolean flag to determine the game status, however making the deployment and redeploy button affect the status of the other buttons was quite bothersome as I had treated all the buttons individually initially, later on I learned that I needed to treat them as a hierarchy with the deployment and the redeploy buttons at the top and the others below them to change the status of each of these buttons with a change in the status of the deployment and redeploy buttons.

After Developing:

I was astonished with how error-free the first prototype was once I had completed it and handed it over to my stakeholders to test, I did not however anticipate the number of features that were missing from prototype 1 as my stakeholders had pointed out at the simplicity of my AI and the lack of UI features, once I had gathered the feedback for my first prototype I had noticed made a list of the missing features and had planned on added them one by one until the list was complete, these features missing features would be created in prototype 2.

Prototype 2

Before Developing:

Before the testing of prototype 1 I had expected prototype 2 to have less features and just build onto the features of prototype 1, however this prophecy would prove to be wrong as the list of features for prototype 2 that I had retrieved from my stakeholders seemed to be much longer than I had expected, with the base plate of prototype 1 these features seemed easier to implement than prototype 1 as most the features had to do with the screens and the UI and less with the overall game, but there was still a larger feature that I had not introduced yet and that was the hard computer, I expected the logic of the hard computer to be much easier to create than the easy computer as I believed all I had to do was built onto prototype 1 with a few extra features, this sentiment was partially right but not quite correct.

During Development:

I had treated the development phase as a check-list of all the features within my list, I had given the hard AI priority as it was the most requested feature that my stakeholders wanted to see in prototype 2. The hard AI was quite challenging to develop as I had initially gone into it thinking that all I had to do was check the 4 grid cells around the hit ship, this thinking was partly correct but I did not think about which direction I would give priority and how I would prevent an infinite grid checking problem, to understand this problem further I had looked up how all the grid cells around a chess piece are checked in python online, and I had used this as reference to check all the cells around my the hit ship cell, this approach to the development of the hard AI had proved sufficient as no errors presented themselves anymore with the hard AI and the hard AI was complete by fixing this bug. During the development of the interactive screens, I had to very minimal research to create them, the research that I did had to do with how to switch screens in pygame as there is no functional way of creating new screens, so I had to become a bit creative with the switching screen mechanic by covering up the prior screen and using some validation to decide which screen needs to be shown, once I had developed the home screen the other screens were quite easy to develop but the logic to go from the main game screen to the end screen was a slight challenging but once I had made it, I realized I could use it to provide a winner so I leveraged the end game logic to declare a winner, the 2x2 bomb was quite easy to create as it I could just use my prior randomized shooting mechanic and increase it to a 2x2 scale to create a button that shoots a random 2x2 position, I was not expecting to create the bomb functionality that easily, however a few problems had presented themselves with the development of the bomb mechanic and that problem is that since the bomb works at a random 2x2 position there is no way of preventing it from shooting at positions that are already shot at, this problem could possibly be solved by setting a couple to rules to the bomb that checks the 4 cells that are going to be shot at making sure they are empty, however if I had gone through with this approach there would be another issue that would present itself and that if 3 of the 4 squares have not already been hit, the 4th square would prevent the bomb mechanic from going off in that position, this could drastically limit the positions the bomb can be used and so I decided against adding it, there are probably better ways of overcoming this problem but more testing would be required to figure out a solution for it. The player vs player mode had not been given a priority during the development of prototype 2, since my stakeholders had requested a larger focus on the AI, since the AI for version 1 had been lacking, so I painstakingly tested the player vs player mode during the end of development but couldn't pinpoint an error, so I had concluded more research would be required to fix this problem in future prototypes.

After Development

Once I had finished developing prototype 2, my stakeholders found more bugs within it than they did within prototype 1, this is because several new features were added that I did have much prior experience in developing. With the finalization of prototype 2, I had become more confident with creating applications with pygame, but I believe there is a lot more for me to learn when it comes to UI development with pygame, since I feel as if I had just about reached the bar when it came to the UI, but there are also a lot more improvement I could do to it to make it look and feel nicer. A new error that I had not expected had shown up once my stakeholders had tested my solution and that was the explosion animation error, it is quite minor but should still be fixed in future prototypes and I had known my turn screen does not show up sometimes but I didn't expect it to be as frequently as my stakeholders had told me, at first I thought it was a minor issue that shows up 20-30% of the time but after I got the stakeholder feedback the figure seemed closer to 50-60%, this drastic increase in the bug had made me label this turn-screen not showing up bug as critical bug and should be given priority when fixing the bugs in future prototypes.

Further Maintenance

I believe my battleship base plate has decent potential once more features are added, so I decided to make the maintenance of the code quite easy for future iterations by annotated segments of the code with what the segment does and have explained how some complex parts of the code work with the usage of comments.

Below are some of the ways the code has been easy to maintain for future prototypes:

- Each portion of the code has been labelled properly, this makes the structure of the code easier to understand if another developer were to look into it, and I have given each variable, data structure, class and function a suitable name that makes their purpose quite simple to understand.
- The code is modular in nature, this makes it seem less cluttered, and new elements such as screens can be created quite easily by using functions that were created in previous prototypes, I have made it easy to understand what the purpose of each function is by providing comments to some of them that may be difficult to understand and giving all of them suitable names, the

usage of functions has also made the code much more efficient, this reduces the hardware requirements needed to test and play the game for future prototypes.

- I have separated the classes from the main file by keeping the classes in a different file and have put the shared utility functions in a file of their own, this prevents the circular import issue in future iterations if a function or variable within some of the files needs to be used in another, separating the files reduces the amount of clutter in a single file, and if a new object needs to be created such as an intermediate AI, navigating within the file has been made easier by separating the class and main file.

Maintenance Issues

A few maintenance issues I could have with my code has to be with the fact I cannot pinpoint some the errors within my code, this could prove fatal in future prototype versions as I do not have starting point on where to tackle the current issues such as the player vs player mechanic, not knowing the cause of an error means I cannot provide comments on how to deal with the current problems in the future, this means a considerable amount of time would be required to pinpoint the issue and fix it, this would not be the case if I could find where the current issues originate and provide instructions on how to fix them.