Battleships Video Game Design

by Louis Bennette - 200801270

1 Summary

Battleships is a two-player table-top board game. Both players have a hidden hand consisting of two grids, the first being the placement of their battleships and the other being a grid of the attempts they have made to attack their opponent. The game requires that the players take turns at making guesses as to where the opponent's ships are. If the guess is a hit, a miss or a ship sinking, the opponent must announce it. More specifically for this project I will be using the official Hasbro rule set [4]. I will be transferring all the concepts and rules over into a playable interactive video game.

1.1 Aims and Objectives

I intend to reproduce the Battleships board game as a video game which can be played offline and online.

1. Singleplayer is the offline mode where the player will play against an AI opponent.
2. Multiplayer is the online mode where player will be paired with another player opponent by connecting to a game hosting server.

The game will have images rendered as graphics to represent all of the graphical user interface elements.

1.2 Changes to the original specification

The original specification was a board view of what would be featured in the game. There are few new decisions that need to be added as well as some decisions that have changed.

The confirmed language that will be used is Java as it has very good server socket control [5] which will be very useful for by the client server model that will be implemented. Java is multi-platform which will make the game more accessible to a wider range of users. The game is expected to run on individual's home and personal computers. There a fragmentation issue due to the inevitability of different screen sizes. This is a problem because the game will run full screen and utilise all of the screen. As a simple solution I have chosen to only deal with 16:9 and 4:3 aspect ratios. If a user does not have a screen with the required aspect ratio the game will be windowed and will run in scaled 16:9 mode.

There are also a few features that I will implement into the game if I have enough time. All of the core features have not been changed of had anything added to them. The following are optional features only:

- I will make the program available on apple OSX computers. The libraries that are being used make this feature feasible as I only need to package the correct operating system native files.

  This therefore changes the deliverables at the end of the project by adding an OSX version of the client.

- I will implement multi-language options into the game, I will do so using dictionaries in both the client and the server. The client dictionaries will hold language specific information for components such as button names, button text, label text, ship names. These are all that singe-player mode requires. Multi-player uses all of these but will use the connection to the server to retrieve multi-player specific text.

- I will implement an in-game chat room for multi-player games so the user can talk with their opponent.

The game will run without an internet connection. The user will be able to fully play the game and use the single-player and options features. However, if the user is not connected to the internet and attempts to use the multi-player, the scene will load but there will be a message stating that the user needs to be connected to the internet. An internet connection is required for the multi-player game.

The libraries I had chosen in the specification will still be used. LWJGL [1] will be my graphics to screen manipulation using openGL. Slick2D [7] will be used for sounds and texture loading and binding.

1.3 Research that has been conducted

For one week after my specification was handed in I was conducting test programs.

The first of these programs was testing a small client-server based system to understand how the server socket programming in java works. For the socket programming practice I used Jones's BSD Sockets programming from a multi-language perspective [5] The system was running on two machines on a local area network and was successfully making a connection, sending and receiving data and closing the connection safely. This first project taught me the risks involved with server connections and also a better from of practice to use for the listeners. The listeners will have their own thread and will call back to the respective class with the message that is required. More on this in the Multiplayer, Server Login and Matchmaking section 2.4 on page 10. The risks here are that catching brute loss of connections is difficult and can quickly cause a chain effect that crashes the client and/or the server.

The second test I conducted was a simple texture and text renderer. My program would load up textures and rectangles with textures bound to them. This test also had a small bit of

text with a personalised font that printed to screen the current mouse's x and y coordinates. This was a crucial test as these were all the basic requirements that my program needed. Based on this test I can now create buttons with text and have them be click-able. I can also draw a ship to screen and have drag and drop features using these simple principles. I learnt that I should use an update-and-draw system where at every iteration of the game loop I update the elements I need to update and draw the elements I need to draw to screen. This means every loop represents one frame in the game. More on this in the Scene Views and Components section 2.2 on page 4.

2 Design

2.1 Overview

The game will run on a simple update-and-draw loop principle. The GameLoop class will be the main controller for the game. In this class there will be a loop designed to loop indefinitely until the game close is requested. The game loop creates and controls the appropriate scene views and the scene views them selves create and control their respective Components. The game loop calls the *draw*() and *update*() functions. The SceneViews call the *draw*() and *update*() on their Components. Finally the Components draw to screen and update their location and state. The game loop makes use of the GameState class which holds different enumerations for different states. Specifically the game loop makes use of the General state to know what scene view it needs to call update and draw on. These simple building blocks are the basic structure of the game. The implementation will be using object oriented design patterns such as singleton classes and factory classes [6].

There are a few factory classes that are used for system wide access to their functions. By system wide I mean that any class in the client has access to their methods and fields (through getters and setters). The system wide classes are:

- OpenGLManager: used to initialise openGL and to draw rectangles to screen
- TextureManager: used to return a texture given a string.
- FontManager: singleton class used to provide the game fonts to components
- ResolutionManager: singleton class used to provide margins and pre-calculated positions for game objects dependent on the aspect ratio
- Window: used to control the window and update the window for every frame.
- Input: Keeps track of all the keyboard and mouse inputs.
- Time: Keeps tract of the time every frame lasts.

For my Input and Time I will be taken most of the code from Benny Bogoganoosh's Input.java [2] for the Input class and his Time.java [3] for the Time class. This is because these classes do exactly what I needed them to, as the project is a large body of work using these simple classes will aid me greatly.

2.2 Scene Views and Components

The scene views are used to hold components. The game will specifically consist of four scene views. Singleplayer, Multiplayer, Options and Menu. Components are the items that are graphically represented and manipulated by update calls these are for instance; Buttons, Ships or Grids.

Scene Views:

The scene views are responsible for initialising their components. When the game loop updates and draws the view, the view must in turn update and draw all of the relevant components. The views make use of the GameState class to know information about the Player state or the General state of the system. The Singleplayer and Multiplayer classes inherit from the AbstractGame class, this provides the user the ability to get game options such as return to menu (Quit) or to change the settings (Options).

• *Menu*: The menu is the first view that the user will see as they load the game. The menu will be the view from which the user can navigate to other views. Menu will hold a button registry and an *onClick(Button b)* function to handle click events from the buttons belonging to menu. These buttons' particular funtionality is to allow the user to navigate to other scene views. There will be a single player button, Multiplayer button, Options button and an Exit button.

• *Singleplayer*: This view will inherit from the abstract game class. The view will hold the players as one user player and as one AI computer opponent. The view will also have a button registry with the same functionality as in the Menu. The player will hold it's own components such as ships and grids so the player classes also need the update-and-draw functions. The view will start by letting the player place their ships anyway they like,. When the user is ready the view will randomly choose a player to start first, then the game will start. When the game is won by someone the scene view will inform the players who has won and provide buttons as option for what to do next.

• *Multiplayer*: This view is very similar to the Singleplayer view. This view has a button registry as well, it also has a user player. The opponent is an online player. The user will initially see a text field where they will be prompted to enter a username. This will be sent to the server to establish a connection. Then the user will be randomly allocated an opponent for them to battle with. The rest of the game from that point will be the same as the single player game.

• *Options*: This view is where the user will choose configurations for the game such as mute options, or resolution options. It's worth noting that much of the options view's

functionality is of lower priority and is therefore treated as optional feature sets.

Components:

The components are the interface the game has with the user. The components provide a graphical representation on screen for the user. They also provide interactions to the user by, for example, allowing the user to drag and drop ships or click coordinates on a map or buttons, etc..

• *Buttons*: This component simply provides the user the ability to click the button and have that button correspond to a particular slice of code that the game will execute. The buttons require an IClickable as a parameter. An IClickable is typically a scene view. It holds the ability to deal with button events. When button in a clickable view is pressed the the *onClick(Button b)* function of that view is called and based on the parameter *b*'s name the correct bit of code is executed.

• *Ships*: The ships are components that belong to the player class. The ships are initialised when the player class is initialised. The ships can be dragged and dropped onto the grid during the placement phase and can be rotated. The ships are actually made up of five classes, one for each ship, Carrier, Battleship, Cruiser, Submarine and Destroyer. This is because in one game we will always have two instances of each ship. The ships are also constant in their sizes.

• *Grids*: The grids are a representation of the battle field but the can also act as a container for the location of the ships. The grids are used to determine what cells have been hit and where those hits missed or succeeded.

• *Labels*: This is a container for printing text out onto an area of the screen. Labels can not be updated they are used by the game as a type of status bar. A label can be marked as a title so that when it is drawn, the font used will be the title font.

• *Text Areas*: This is container that allows the user to type in a string of text which allows editing and keyboard commands. The Text area will have an action assigned to it. For instance, the chat text area in a multiplayer game will perform a broadcast action with the string written by the user and empty the current text field's contents. The text area will have a isBeingEdited field. This field is flag that allows the update method to append direct input from the user to the text area.

2.3 AI Opponent

The AI will be used to play against the player in single player mode. The role of the AI is to provide a challenge to the user by using a heuristic. To tackle this problem I have devised a solution that would make the game still be possible to win but also allow the AI a fighting chance to beat the user.

The AI will hold its own 2 grids, they will be used the same way as the user's, one for the AI's ships and one for the attempts on the opponent. Perhaps the AI will also require a sleep timer because the response of such an AI will be very fast and could make the user feel un-immersed.

The heuristic will be simplistic and straight forward, it consists of two states, search and kill. Meaning simply that the AI will search semi-randomly across the board and when it hits a ship it will attack until it has destroyed the ship.

More specifically when an enemy ship has been hit we will add the points of the hit into a list. Regardless of the state the AI is in, every time it makes a successful hit, it will add that hit's coordinates to the list. When a ship has been sunk the respective points of that ship are removed from the list. This implies that any points in the list belong to a ship that has not been sunk. Only when the list is empty the AI can resume the search state. As well as updating the list we add every attempt the AI has made to the targetGrid array. This is to avoid the AI choosing the same point twice.

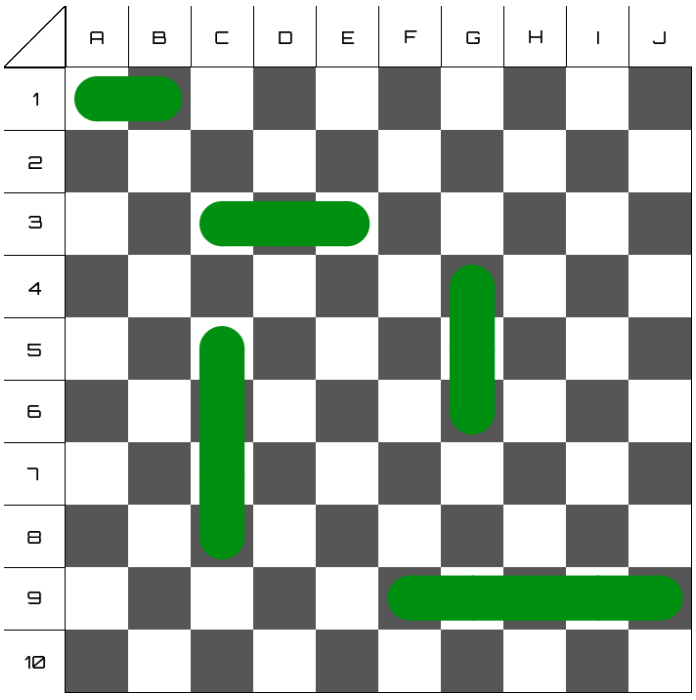


Figure 1: Every ship has to exist on a black and a white cell of the grid regardless of placement and orientation.

Search State:

The semi-random behaviour was a behaviour I came up with over the summer, the idea was reinforced when I read an article expressing the same behaviour. That is, if we assume the game grid is like a chess board with checkered black and white tiles (see figure 1). The AI only needs to search in the black squares to find all of the ships. This is because all the ships are 2 or more squares in length meaning they have to appear in both black and white regions in any placement. Using this will shorten the maximum time it takes to explore the whole board by half. (Providing we stay in the search state) Another thing that the search state will do is avoid redundant cells. In figure 2 we can see that If the above behaviour chose the ? cell it would be a waste of a turn as this cell could never possibly have a ship in it. The algorithm would ignore this cell and make an attempt on another cell.

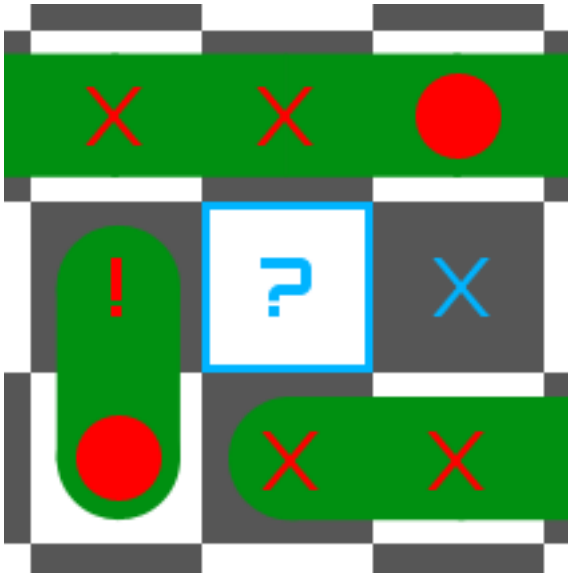


Figure 2: • = Origin, × = Hit, ! = Sinking hit, × = Miss. Example of a redundant cell where the cell '?' is the cell in question.

Kill State:

The kill state is an original creation based on tackling the possibility that two ships are adjacent and making sure the AI can account for all the identified hits before switching back to search mode. There are 3 sub states: Identify, Horizontal Attack, Vertical Attack. It does so by holding and analysing a list of all the hits it has made. When an enemy ship has been hit we will add the points of the hit into a list. The first point of that list will be the "origin" of the kill state. Next turn the ai will randomly chose a point adjacent to the origin to fire.

Identify is the initial internal step of the kill state. This step requires the AI to 'identify' the orientation of the ship it is hunting. The heuristic here is to assume that if we have a point in the list then there will be at least one adjacent point to it. The AI therefore starts by randomly firing at any of the adjacent points in the grid. If there is a miss we stay in

the identify state. If there is a successful hit in the cell above and below the origin we enter vertical mode. If the hit is to the left or right cell we enter horizontal mode.

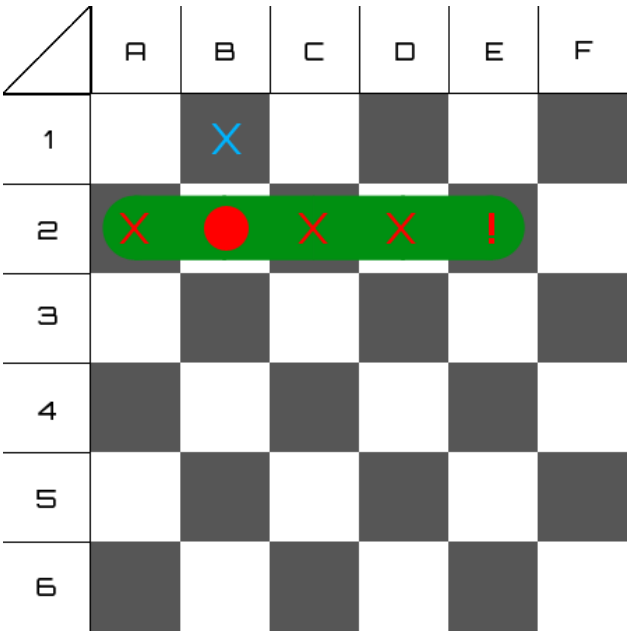


Figure 3: • = Origin, × = Hit, ! = Sinking hit, × = Miss.

In figure 3 we see that the AI started in it's identify state with origin $B2$, the AI made an attempt at $B1$ but it missed and remained in the identify state only to attack $A2$ and then move to the horizontal mode.

Horizontal and vertical mode are essentially the same mode except for their respective orientations. For now we will discuss these modes by using horizontal mode in the examples. It's simply worth noting that vertical mode will work the same way.

If we look at figure 3 The AI successfully entered horizontal mode when it hit $A2$. On the next turn the AI will attempt to continue in the **direction of it's last successful hit** but if an attempt has already been made in that cell or if the cell is out of bounds then we remain in horizontal mode but change direction. This is what is happening in figure 3 after the hit in $A2$ the AI changes direction and hits $C2$ and so on.

$list_4$	$E2$
$list_3$	$D2$
$list_2$	$C2$
$list_1$	$A2$
$list_0$	$B2$

Figure 4: Representation of the list that would be generated by the action made in figure 3

In figure 4 we can see that the items are added to list in order that they were hit and not in their correct order on the grid, this makes removing a ship a little bit more tricky. However due to the steps we take there is an inevitable situation that will always arise. That is, that the sinking blow will always be on the extremity of the ship.

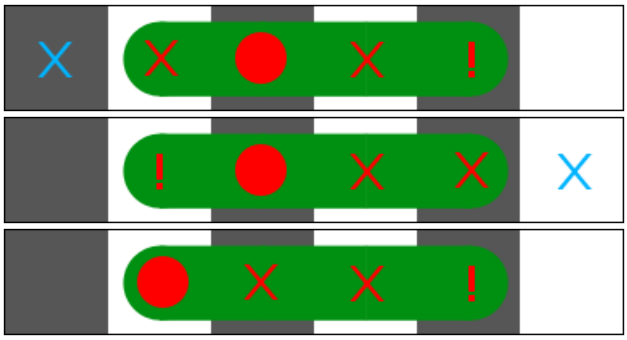


Figure 5: • = Origin, × = Hit, ! = Sinking hit, × = Miss. This is the iteration of the kill algorithm over the three possible states for a ship of size 4.

Figure 5 is a representation of the possible outcomes the algorithm would have on a ship of size 4. Although in figure 5 we only show for a ship of size 4 this rule applies to a ship of any length, providing the length is a positive integer. The situations in figure 5 are unique. Although they can be mirrored, in essence these are the only situations that can occur.

We can prove that these four situations are the only unique situations that can arise for a ship of size 4. To do this I will prove this to be true for a ship of size n. My definition here of a unique situation is; an outcome where none of the outputted situations are mirrors of each other. Firstly for a ship of size n where n is even and $n > 1$: We take

$$uniqueStarts = n \div 2$$

where $uniqueStarts$ represents the number of possible cells that the origin can start without those cells being mirror images of each other. Then we take

$$possibleFollowActions = (uniqueStarts - 1) \times 2 + 1$$

where $possibleFollowActions$ represents the number of possible cells that will be hit after the origin has been hit and those cells are cells that belong to the ship. This is important because if the next cell after the origin is a miss then we are back in the identify mode. But here we ignore all the searching and we only want information about the ship being hit. $(uniqueStarts - 1)$ here represents all the cells that are not on the extremities of the ship. The cells on the extremities only have one possible successful hit and that's moving towards the centre of the ship, this is why we add one.

Secondly for a ship of size n where n is odd and $n > 1$: We take

$$uniqueStarts = (n - 1) \div 2 + 1$$

we can assume that the central point of the ship is unique as it has no mirror images. So we always ensure that the central point is a possible unique start point for the algorithm.

Once the AI has hit it's 'secondary' cells, it will iterate the algorithm for every turn and move in the same direction then finish. Or it will miss, turn around, move in the other direction then finish. Regardless of encountering other ships the AI once it misses will go to the origin change directions and reach the end of a ship.

In other words the *possibleFollowActions* represents the number of unique situations that can arise from the searching and identify states once in the kill states. For any ship of size n if we iterate the algorithm for any start point on the ship, the sinking hit or last hit will always be on an extremity of the ship.

Because we know that a sinking hit ship will always be on an extremity when a sink hit is announced. The AI will also receive the length of the ship. We also know the direction in which we were travelling along the ship just before sinking it. We can work back from that point, the AI will take the coordinates of the sinking hit then using the reverse direction, the orientation and the size of the ship the AI will make a small new list with all the points corresponding to the ship that has just been sunk.The AI will quickly check to see if all the points in the new list exist in the hit list. If they do the AI will remove all those points from the hit list as all the hits are now accounted for. If the origin was one of the points that was removed then we make the first point on the list the new origin and return to the identify mode. If the hit list is empty then we simply return to the search state. Finally, it's worth noting that if the identify mode can't attack an adjacent square then we change the origin to the next point on the hit list and so on. If everything is done well there will always be a point on the hit list that is eligible to become the new origin.

2.4 Multiplayer, Server Login and Matchmaking

During a multiplayer game the player will be playing against an opponent online via a server which is relaying messages to and from each client. The server here will be designed to minimise the amount of memory it uses. To do this the server will just act as a relay, sending messages to and from the clients.

The multiplayer game will be sending the messages to the server, there will also be a message listener attached to the muliplayer game. This simply listens to any message that will be sent by the server. When a message is received it's placed in a list. The multiplayer game will check to see if there are any messages in the list. If there are it will send the message to a parsing function. Once parsed the appropriate output will execute the following correct piece of code.

The client will need to distinguish between the commands. I will be using a code based system. The code is inserted internally and can not be done manually by the user.

- '0' will be a string message to be displayed to the player's text area. Here the '0' will be followed by any message, for example "0Hello World!" The '0' is removed from the string and the rest is printed.

- '1' will be a client command. The command will be a 3 letter string which will refer to a particular piece of code to be executed. Anything after the 3 letters will be used as parameters if needed. For example "1try1,5" will be taken in by the opponent as a try (fire) command where "1,5" are the coordinates. The opponent will respond with either miss, hit or sink. So "1ansm", ans (answer) command with m (miss) as a variable.

- '2' will be server commands for example "2dco" to disconnect the user. Or "2req" to re queue the user for another game. These commands are not relayed to the users they are internally managed by the server.

When the user starts a multiplayer game, the game will in turn initialise an user-player and a online-player. The online-player will be a representation of the opponent's state but will not hold any data about the grid. The user will be expected to position their ships and say when they are ready. When the user is ready their personal state is updated. The game will also send a message to the server which will relay the message to the other connected client. The other client will then update the state of it's online-player. This structure is how all the messages are transferred to each other.

The online user will hold very little information about the opponent. The online user will also hold simple information about the opponent however it will not hold information about the opponent's ships or their position on the grid. It will however hold data about the opponent's state and about what ships are still on the field. This is for simple game checking. If we send a try coordinate command and our answer is "1snk1" then we know that the ship '1', which is a Battleship, has been sunk. The game therefore updates it's instance of online-user. When there are no more ships in the online-user instance, we wont need to ask the opponent via the server. In this way both connected clients will be updated simultaneously.

The multiplayer game will also need to have a small inbuilt parser. The parser will take the first character in the string to know what type of message it is. Then depending on the type we will take the next part of the message and perform an action or more checks. This system is so the user will just need to click a location on the grid and the hit commands will be automated without explicit communication between the users themselves.

The Server:

The server will be multi-threaded, this is because we need several instances of a game to run at once. We also need the server to keep adding new users and listeners for incoming messages. The Server Listener will be listening to any new connections and adding them to a login thread which will handle their username and add them to the online users.

Online users is the list of currently connected users. The user class will hold data on the user's socket and their name. The user also has a listener associated with it. This listener is waiting for messages from the client that it is associated to. When a message is recieved the string will be added to a list. The game instance will check the lists and parse any message for both users in every loop. If there are 2 or more users in the Online Users list then we

take the first two of the list and add them to a game. When the game instance is started up it is expected to receive two users. At this point the message listeners for each user are started up and the game can begin.

It's also worth noting that the server will be run from the command line and therefore will not have any graphical interface. The server will be logging all important activity occurring in the system this is to help debug in case of server crashes.

2.5 Client GUI

The following are simplistic views. The views are not finished images that the user will see. They are spacial representations of where components will be. The colours represent different components: Blue = Grid, Green = TextArea, Red = Button, Yellow = Ship, Purple = Label.

Figure 6: Menu: The black border represents the space where a graphic could be used for the title screen. The title of the game will run along the top on the label.

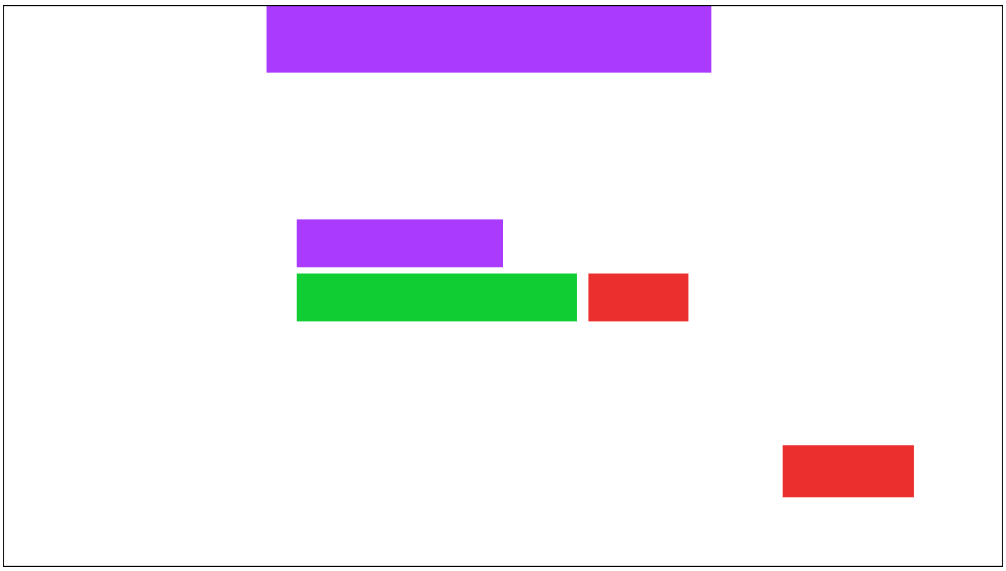


Figure 7: Multiplayer: This is what the user will see when they are asked to enter their username. The back button is in the bottom right.

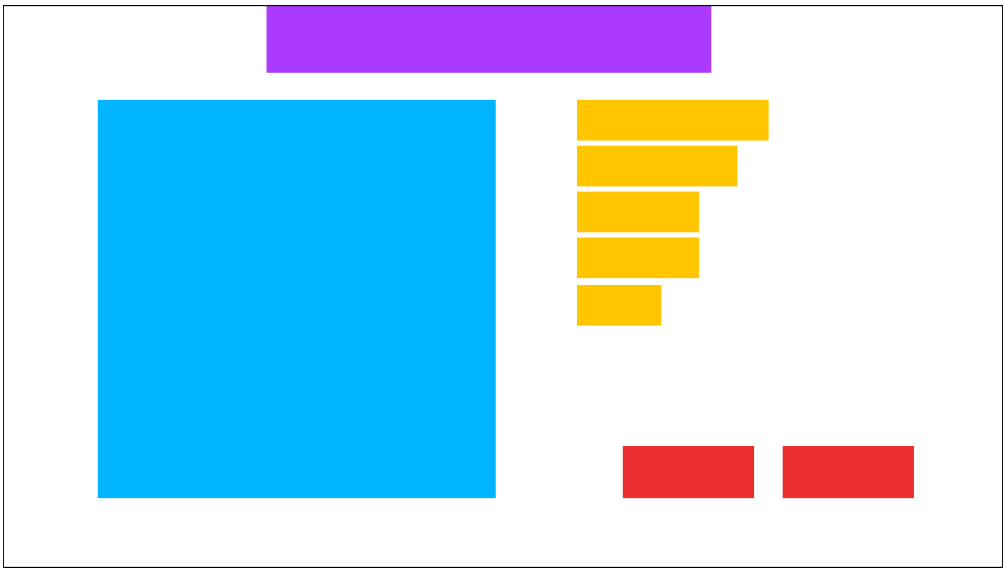


Figure 8: Multiplayer and Singleplayer: This is what players will see when they are expected to position their ships onto the grid. The buttons are the back button and the ready button.

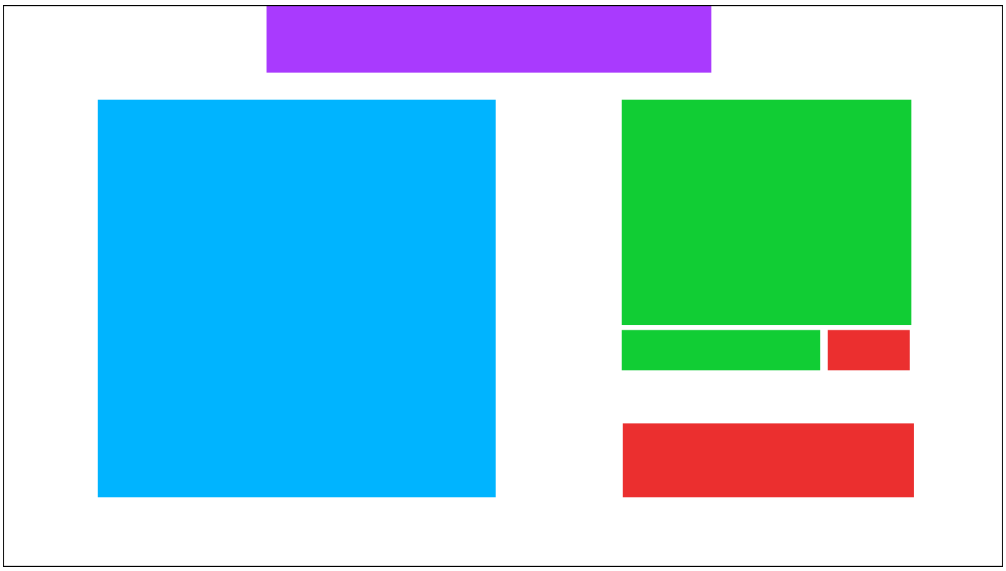
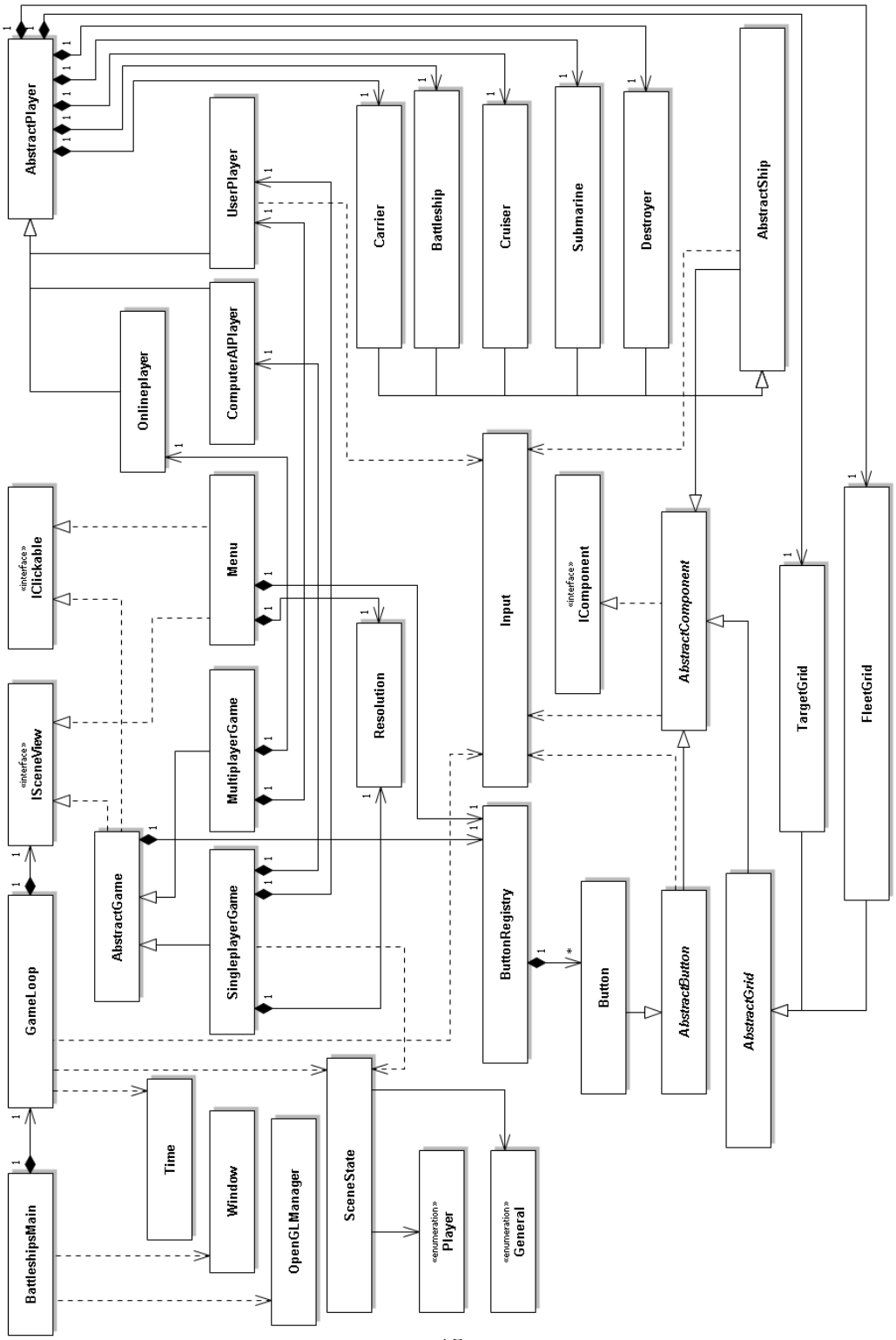


Figure 9: Multiplayer and Singleplayer:

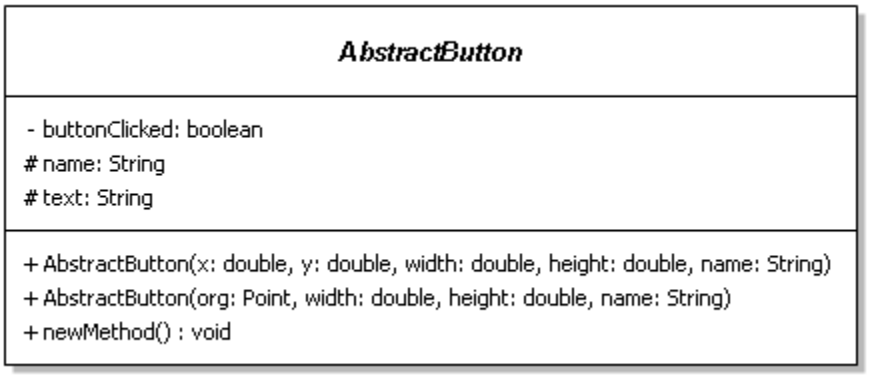
2.6 Client Structure

Figure 10: Battleships client class diagram.

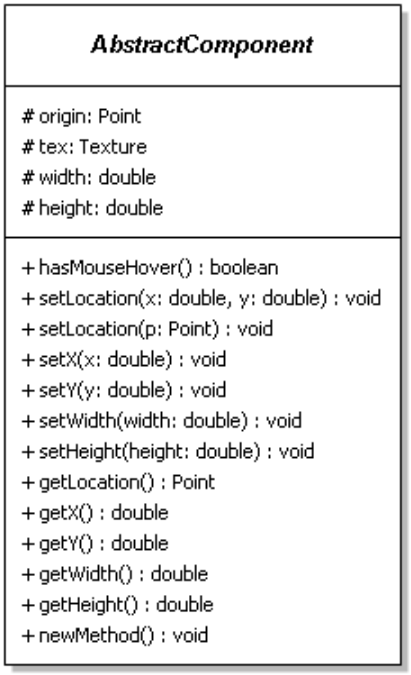


The following is the detailed version of the classes used in the class diagram represented above in figure 10

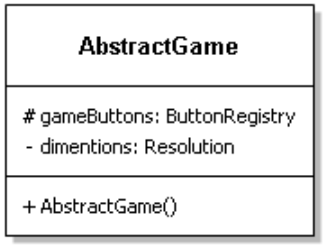
AbstractButton



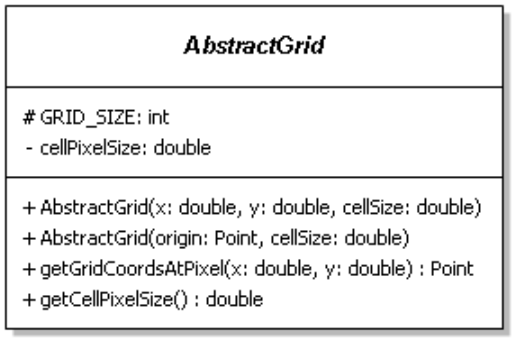
AbstractComponent



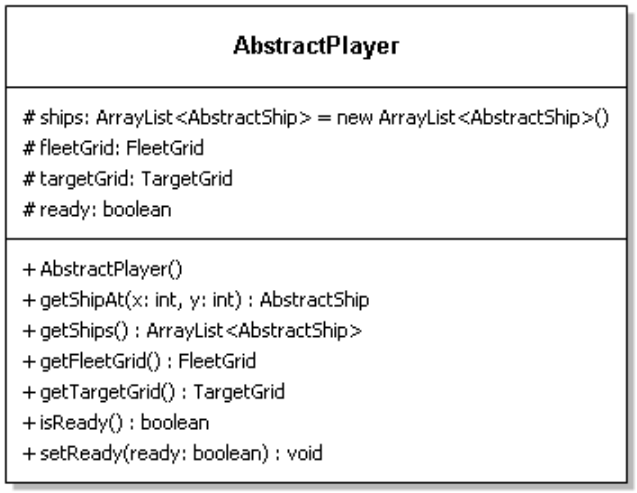
AbstractGame



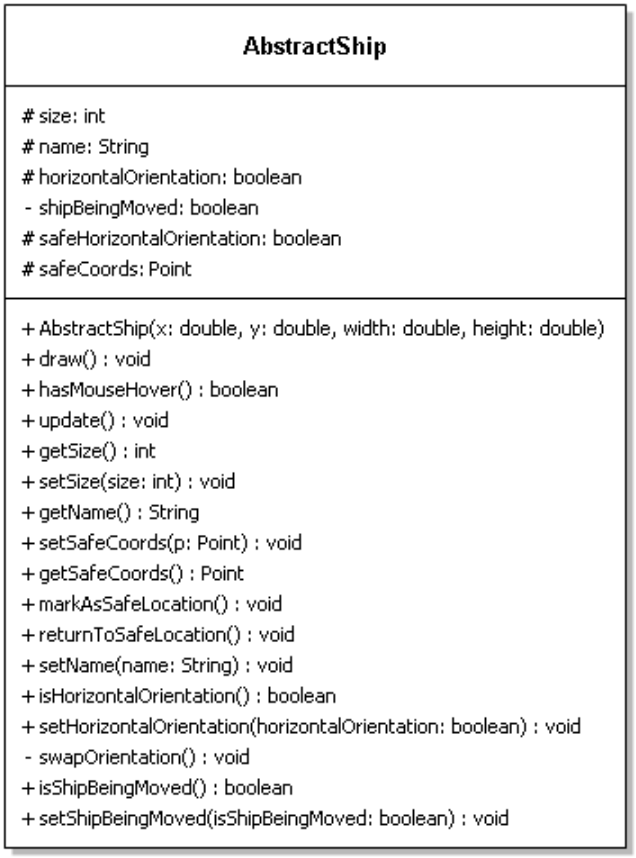
AbstractGrid



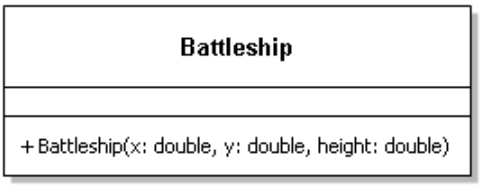
AbstractPlayer



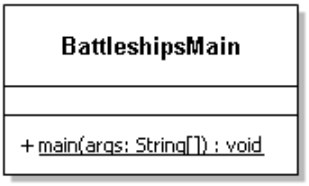
AbstractShip



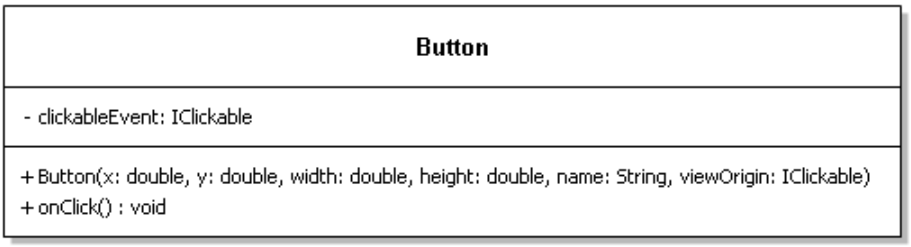
Battleship



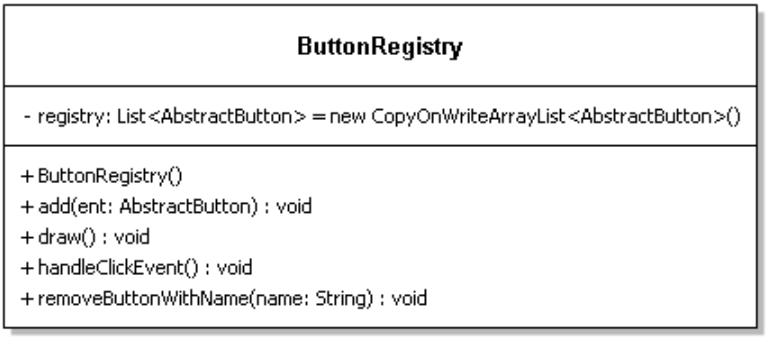
BattleshipsMain



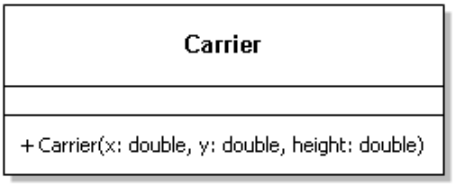
Button



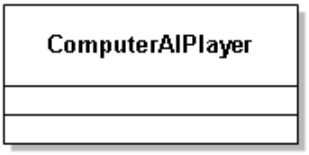
ButtonRegistry



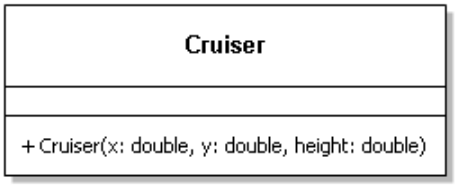
Carrier



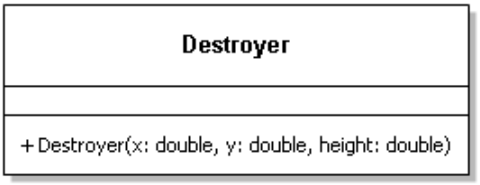
ComputerAIPlayer



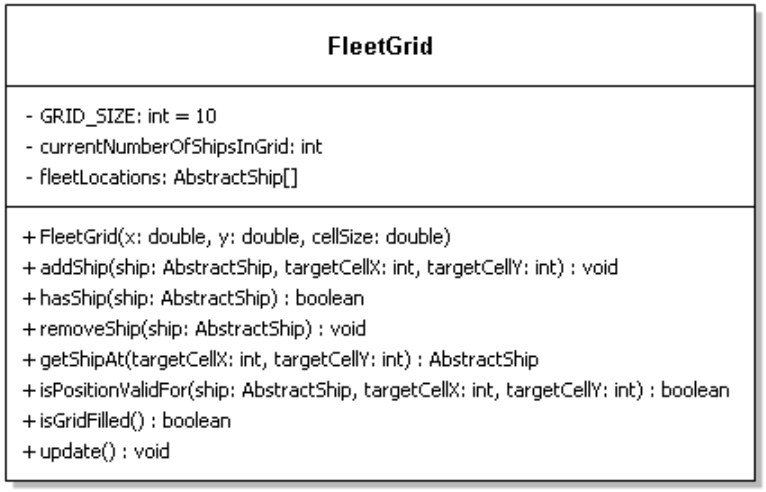
Cruiser



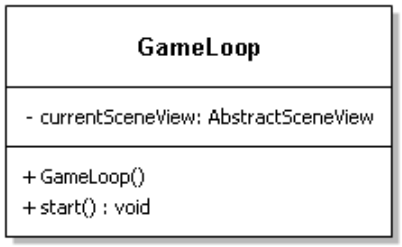
Destroyer



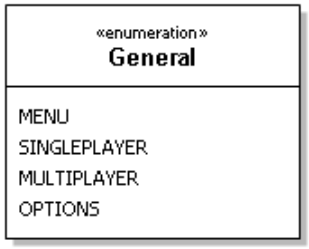
FleetGrid



GameLoop

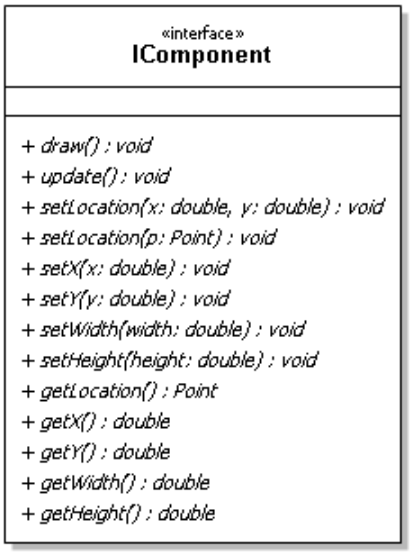


General

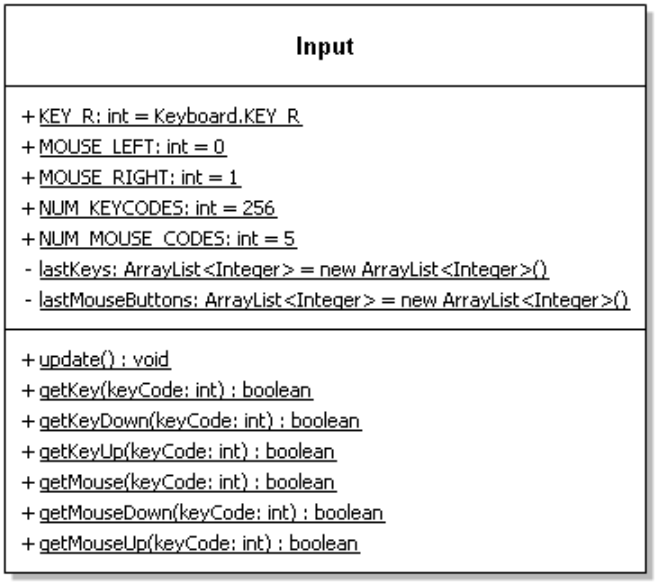


IClickable

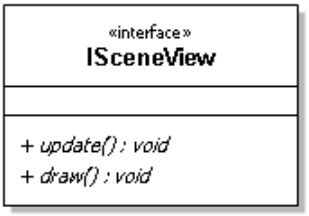
IComponent



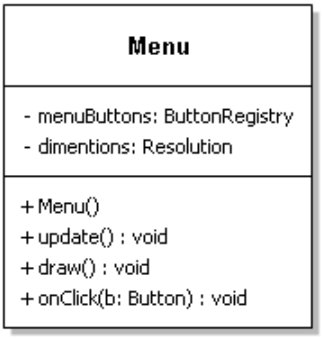
Input



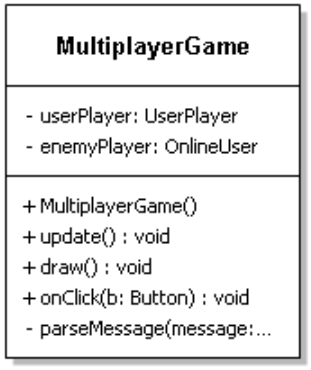
ISceneView



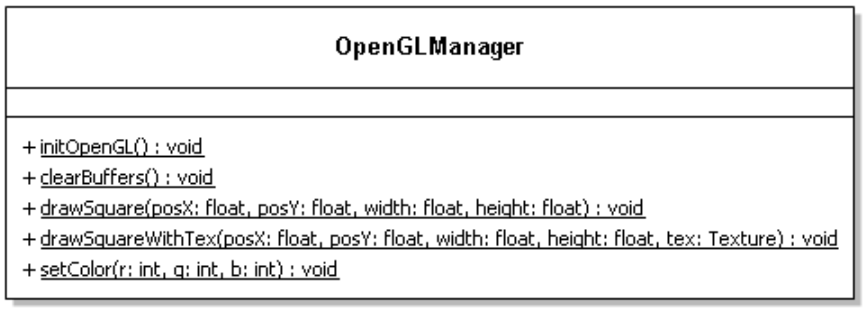
Menu



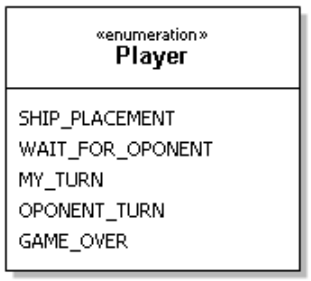
MultiplayerGame



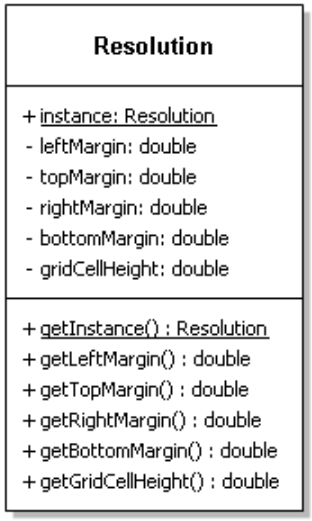
OpenGLManager



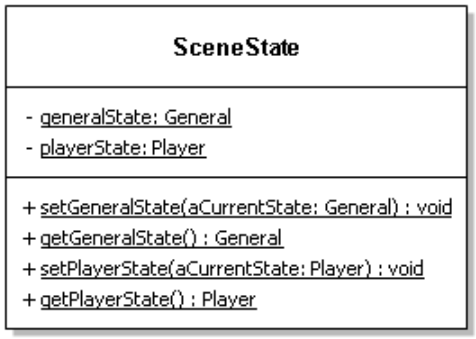
Player



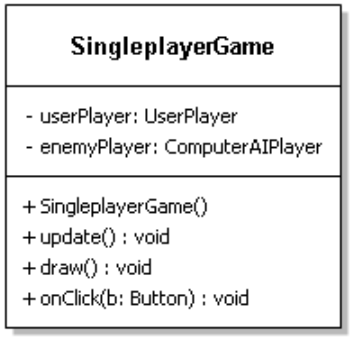
Resolution



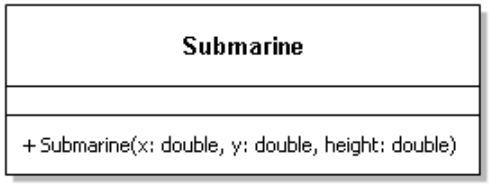
SceneState



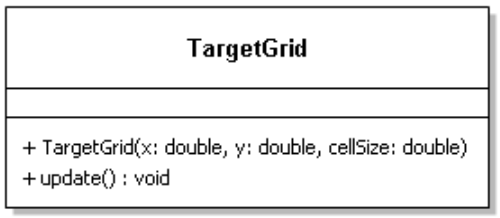
SingleplayerGame



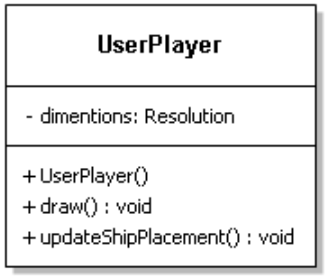
Submarine



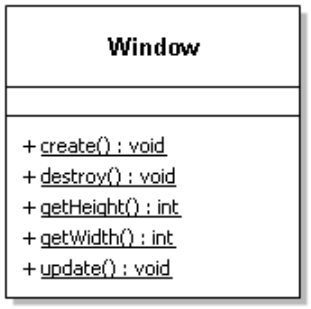
TargetGrid



UserPlayer

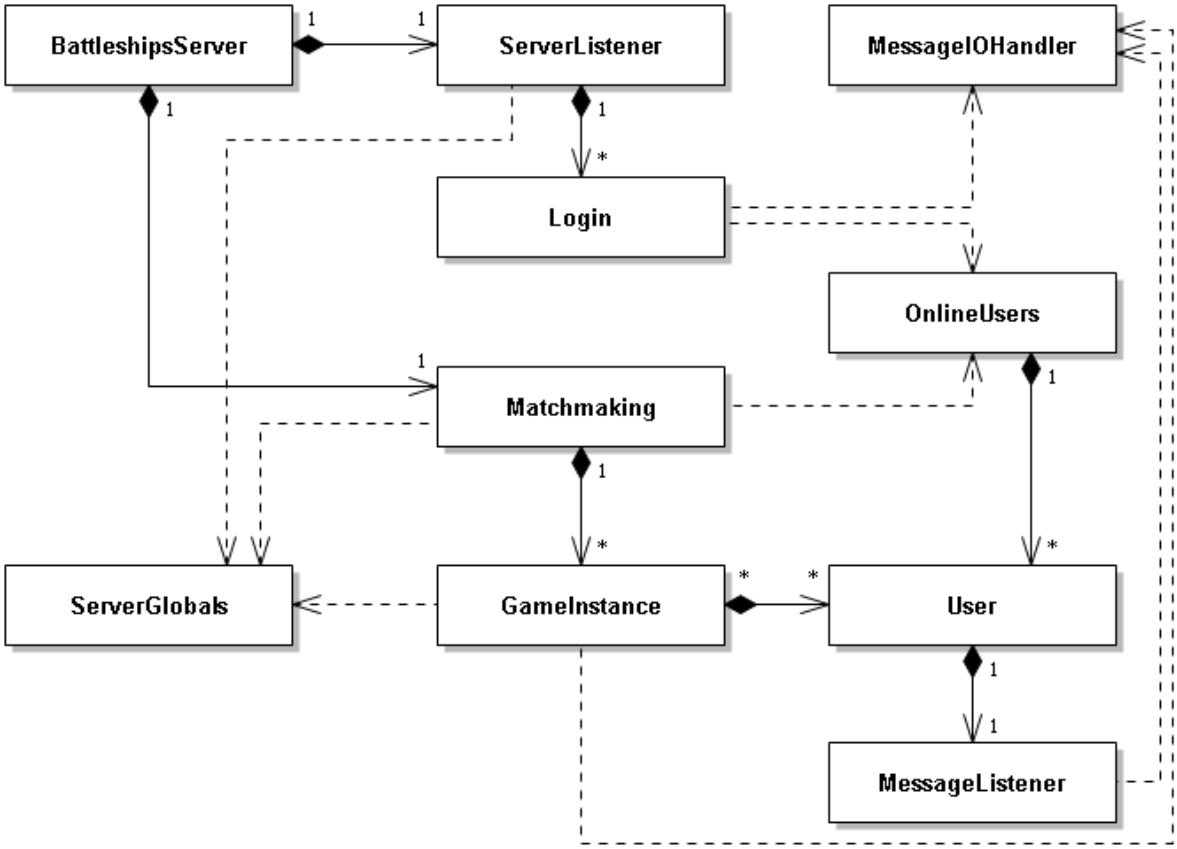


Window



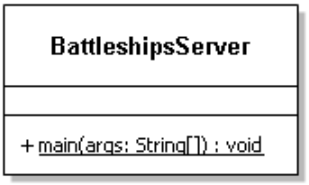
2.7 Server Structure

Figure 11: Battleships server class diagram.

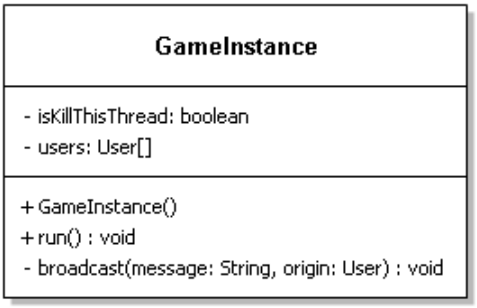


The following is the detailed version of the classes used in the class diagram represented above in figure 11. Note that the class diagram does not tell what methods are synchronized or not. The public static methods in MessageIOHandler and in OnlineUsers are actually public static synchronized methods.

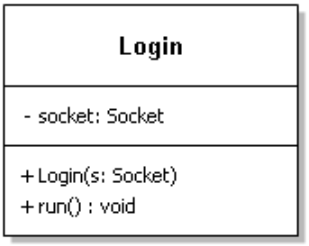
BattleshipsServer



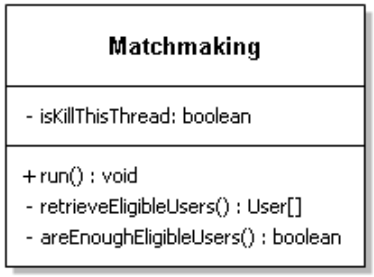
GameInstance



Login

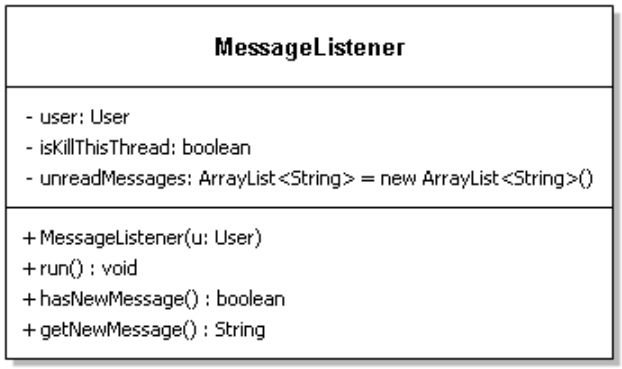


Matchmaking



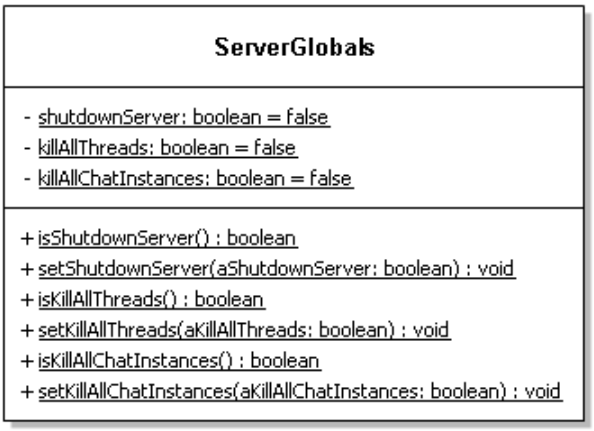
MessageIOHandler

MessageListener

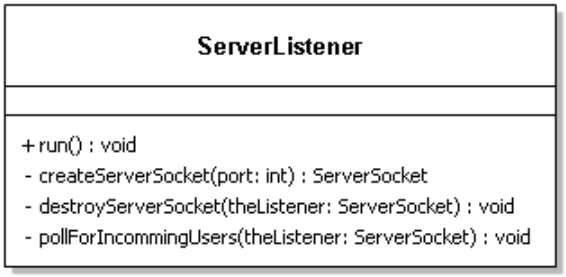


OnlineUsers

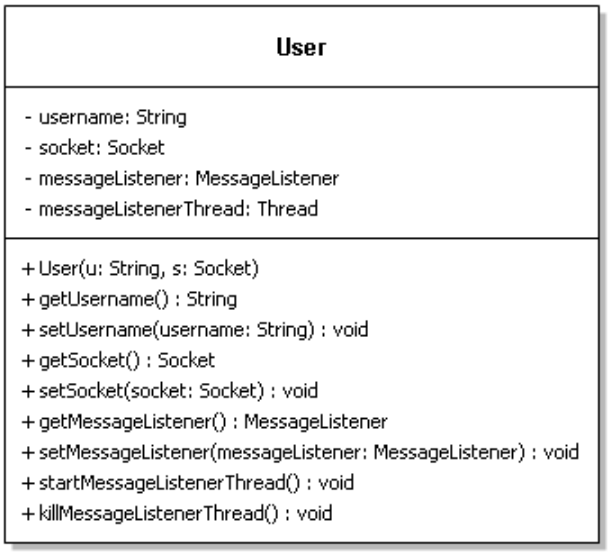
ServerGlobals



ServerListener



User



3 Review

The gantt chart specified durring the specification phase has been altered. So far, in a general perspective I have achieved what I was expecting to this far along the project. There have been a few things that have been done differently during the design phase. I decided that my project design did not need as many diagrams to express how it would work. I also spent more time in my research phase as I ended up coding a substantial amount during this testing. Some of the code I tested will be reused in the implementation itself especially the work done on the server socket tests.

The following summarises what has been delivered and what is expected to be delivered in the future.

Delivered:

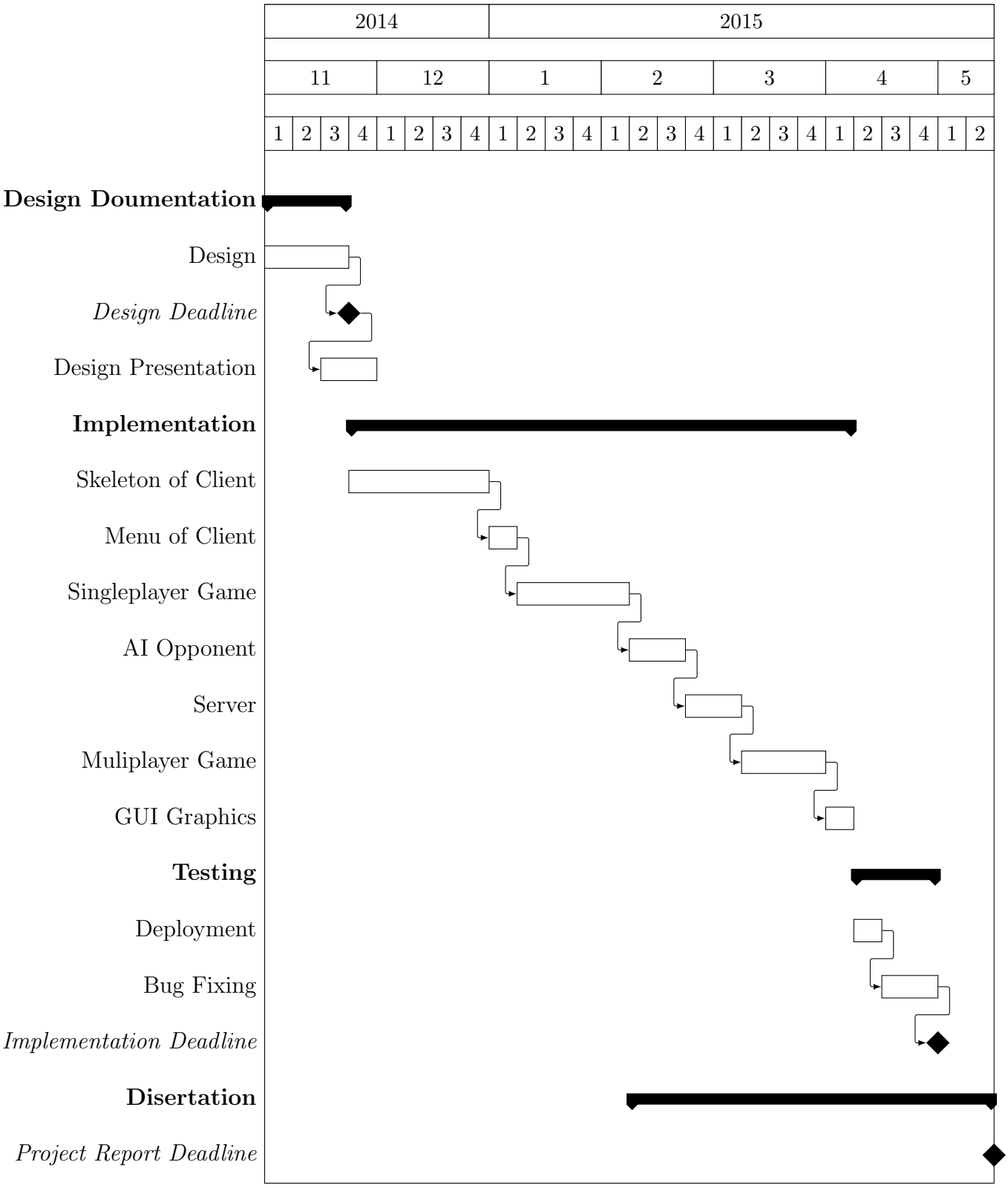
- Specification document
- Design document
- Design presentation slides

To be delivered:

- Interim progress report
- A working game client
- A working game server
- Implementation presentation slides
- Project Report document

In order to evaluate the software I must re-iterate what the aims of the project are and what will be delivered at the end of the project. I plan to have a client to project the game to the user via a clear and easy to use interface. The other part of my system is a server which will be expected to be running and serving the clients for several days on end. So the evaluation of the project will be; how well do users respond to navigating and playing the game? How clear are the images? Is the game of high standard and enjoyable? Is the game easy to play but still engaging? For the server my evaluation will be; how robust is the software for the client and the server? How responsive and fast are the messages dealt with? How easy is it for the user to connect to the server? The actual evaluation of the game will be done when the implementation of the software is completed and has been deployed and tested.

The following is a revised version of the gantt chart presented in the specification document. This gantt chart is in less detail as the project itself is agile and will require reworking.


2014
2015
11
12
1
2
3
4
5
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2
Design Doumentation
Design
Design Deadline
Design Presentation
Implementation
Skeleton of Client
Menu of Client
Singleplayer Game
AI Opponent
Server
Muliplayer Game
GUI Graphics
Testing
Deployment
Bug Fixing
Implementation Deadline
Disertation
Project Report Deadline

References

[1] Lwjgl, 2013.

[2] Benny Bobaganoosh. 3d game engine input.java, 2014.

[3] Benny Bobaganoosh. 3d game engine time.java, 2014.

[4] Milton Bradley Co. Batteship for 2 players, 1990.

[5] M. Tim Jones. *BSD Sockets programming from a multi-language perspective [electronic book] / M. Tim Jones.* Online access with subscription: Ebrary. Hingham, Mass. : Charles River Media, c2004., 2004.

[6] Partha Kuchana. *Software architecture design patterns in Java [electronic book] / Partha Kuchana.* Online access with purchase: Taylor & Francis. Boca Raton, Fla. : Auerbach Publications, 2004., 2004.

[7] New Dawn Software. Slick2d java documentation, 1999.