

Assignment 1 –  
*Battleships* Design Document

AC11001 Introduction to Software Development

Jodie Laurenson

# 

# Statement of requirements

#### ***Description of purpose***

The purpose of the solution is to provide a piece of software, developed using the programming language Java, which will allow users to play a version of the game *Battleships*. The program will randomly place ships on a grid and then ask the user to input grid coordinates. There will be by default a total of ten ships per game, of differing sizes, which will be placed at random on the grid at the start of the game.

The software will allow the user to save an in-progress game to an external file, with the option to load from this file instead of starting a new game.

#### ***End users***

The target audience will be young children between the ages of 8–11 who are learning basic arithmetic skills. The game will explore the concept of strategy within a game.

##### Assumptions made

* The game should be clear and easy to use,
* The user is able to use a keyboard and monitor
* The user will have the required hardware and software to run the game

#### ***Inputs to the program***

* On the first menu, typing 1 results in a new game being initialised
* On the first menu, inputting 2 results in a saved game being loaded
* Typing 3 on the first menu results in a new multiplayer game being created
* On the second menu, inputting 2 results in the current game being saved
* Entering coordinates using the keyboard and pressing enter button – results in the program checking whether or not the target coordinates are occupied or not, and marking the cell as appropriate on the grid. This is carried out after the user types 1 on the second menu
* Typing 0 on either menus results in the program terminating

#### ***Outputs from the program***

* Updating of the player’s grid when they fire
* Allocating position of the ships
* Displaying the grid showing where has been fired at
* Display the amount of shots taken once all ships are sunk and after every turn
* Saving game details to an external file
* Displaying the position the computer has fired (only in multiplayer)

### 

### 

### 

### 

### Requirements

#### ***Functional requirements***

**R1.** The program will allow the player to **input the coordinates** of their attacks using the keyboard.

**R2.** The program will **start a new game** of *Battleships* when the user enters “1” button on the main menu.

**R3.** The program will **load a previous game** of *Battleships* when the user enters “2” button on the main menu.

**R4.** The program will **start a new multiplayer game** of *Battleships* when the user enters “3” button on the main menu.

**R5.** The program should **close** when the user enters “0” on either menus.

**R6.** The program will allow the user to **save** an in-progress game to an external file.

**R7.** The program will **display whether or a ship has been hit.**

**R8.** The program should **check that a valid integer** has been inputted**.**

**R9.** The program will **update the board** with “X” or “O” to show where the user has fired**.**

**R10.** The program will display whether or not a **ship has been sunk.**

**R11.** The program will **randomly place ships** on the grid at the beginning of each game.

**R12.** The program will **place a ship again** if it is an invalid position.

**R13.** The program will **increment the shots taken** every time and display on the console**.**

#### ***Non-functional requirements***

**R14.** The program will have an intuitive, easy-to-understand interface.

**R15.** The program will comply with the Copyrights, Designs and Patents Act.

**R16.** The program will not store any information on its users, and as such will comply with the Data Protection Act.

**R17.** The program will require a Java virtual machine to be installed on the computer system in order to run.

**R18.** The program will be robust, able to detect when erroneous input is given and prevent the player from doing so.

**R19.** The scope of the game development must be within the timeline of the project.

### 

### User interface

* The menu should be easy to understand so the user knows which option they are selecting
* The initial menu should present the player with the option to start a new game or load an older game.
* The game board will be simple and easy to read.
* The grid axis will be labeled 1 to 10 horizontally and vertically

### Use cases

1. In this example of using the battleships program, the user initiates a new game. The system begins by randomly placing the ships on the grid for both itself and the user. The user then enters the coordinates they want to fire at and the system detects whether they have hit a ship or missed. The system then generates a coordinate at random, detects if it has fired at that location before and whether it hit the user's ships. The user then saves the game and the system writes a text file.

|  |  |  |
| --- | --- | --- |
| New Game —> Save Game | | |
| 1 | User | Selects ‘new game’ from the menu. |
| 2 | System | Initialises a fresh grid, placing the ships at random. |
| 3 | User | Enters the coordinates they wish to fire at. |
| 4 | System | Looks up the coordinates the player has chosen. |
| 5 | System | Checks if there is a ship present at the player's coordinates. |
| 6 | Sytem | Marks the coordinates with either a cross or a circle accordingly. |
| 7 | System | Generates a coordinate to fire at. |
| 8 | System | Checks if position has already been fired at, if it has, get a new coordinate |
| 9 | System | Updates grid depending on if ship is present or not |
| 10 | User | Saves the game. |
| 11 | System | Writes out current game conditions to text file for later retrieval. |

1. In this example the user loads a previously saved game file and continues to play the game until the user destroys all of the opponent's ships, when the system displays the win message, with the score displayed. The user then starts a new game.

|  |  |  |
| --- | --- | --- |
| Load Game —> Win Screen —> New Game | | |
| 1 | User | Selects load game from the menu |
| 2 | System | Imports game file to board |
| 3 | User | Enters coordinate to fire at in the fire box |
| 4 | System | Looks up the coordinates chosen by the player |
| 5 | System | Detects the cell at chosen coordinate as having a ship in it |
| 6 | System | Detects the cell as the final cell with an intact ship in it |
| 7 | System | Displays score to user |
| 6 | User | Selects new game from the menu |
| 7 | System | Initialises a fresh grid, placing the ships at random. |

1. In this example the user selects a new game. The user then enters a coordinate and the system checks and confirms that it is indeed valid, so the user then enters another coordinate. The system checks to see if the coordinate is valid; it isn't, so the system sends an error message and prompts the user to enter new coordinates.

|  |  |  |
| --- | --- | --- |
| New Game —> Repeated Shot | | |
| 1 | User | Selects ‘new game’ from the menu. |
| 2 | System | Initialises a fresh grid, placing the ships at random. |
| 3 | User | Enters the coordinates they wish to fire at. |
| 4 | System | Looks up the coordinates the player has chosen. |
| 5 | System | Checks if there is a ship present at the player's coordinates. |
| 6 | Sytem | Marks the coordinates with either a cross or a circle accordingly. |
| 7 | User | Enters coordinate to fire at in the console |
| 8 | System | Looks up the coordinates chosen by the player |
| 9 | System | Detects cell as having previously been fired upon |
| 10 | System | Displays dialog box prompting player to choose new coordinates |

### 

### 

### Classes

#### ***Candidate classes***

|  |  |  |
| --- | --- | --- |
| **Candidate classes** | **Accepted / rejected** | **Reason for rejection/acceptance** |
| BattleshipsGame | Accepted | Top-level coordinator class. Also contains the information on a particular game of Battleships, including the Grid objects and Player objects, and miscellaneous information such as whose turn it is. |
| loadGame | Rejected | Method of BattleshipsGame |
| saveGame | Rejected | Method of BattleshipsGame |
| newGame | Rejected | Method of BattleshipsGame |
| gameStatus | Rejected | Field of BattleshipsGame |
| playerType | Rejected | Field of Player |
| playerName | Rejected | Field of Player |
| shotsFired | Rejected | Field of Player |
| hasWon | Rejected | Field of Player |
| playerTurn | Rejected | Field of Player |
| HumanPlayer | Accepted | Contains methods for allowing the human player to play the game. |
| CPUPlayer | Accepted | Contains methods for allowing the computer opponent to make guesses. |
| Grid | Accepted | Contains the Ship objects on a particular grid, as well as which cells have been hit. |
| shipObjects | Rejected | Array of Ship objects, field of Grid. |
| cellStatuses | Rejected | Array of booleans, field of Grid. |
| Ship | Accepted | Contains information such as type of ship and status. |
| shipName | Rejected | Field of Ship containing ship type e.g. ‘Submarine’, as a string. |
| shipSize | Rejected | Field of Ship containing number of cells taken up. |
| cellStatuses | Rejected | Array of booleans, field of Ship. |

#### 

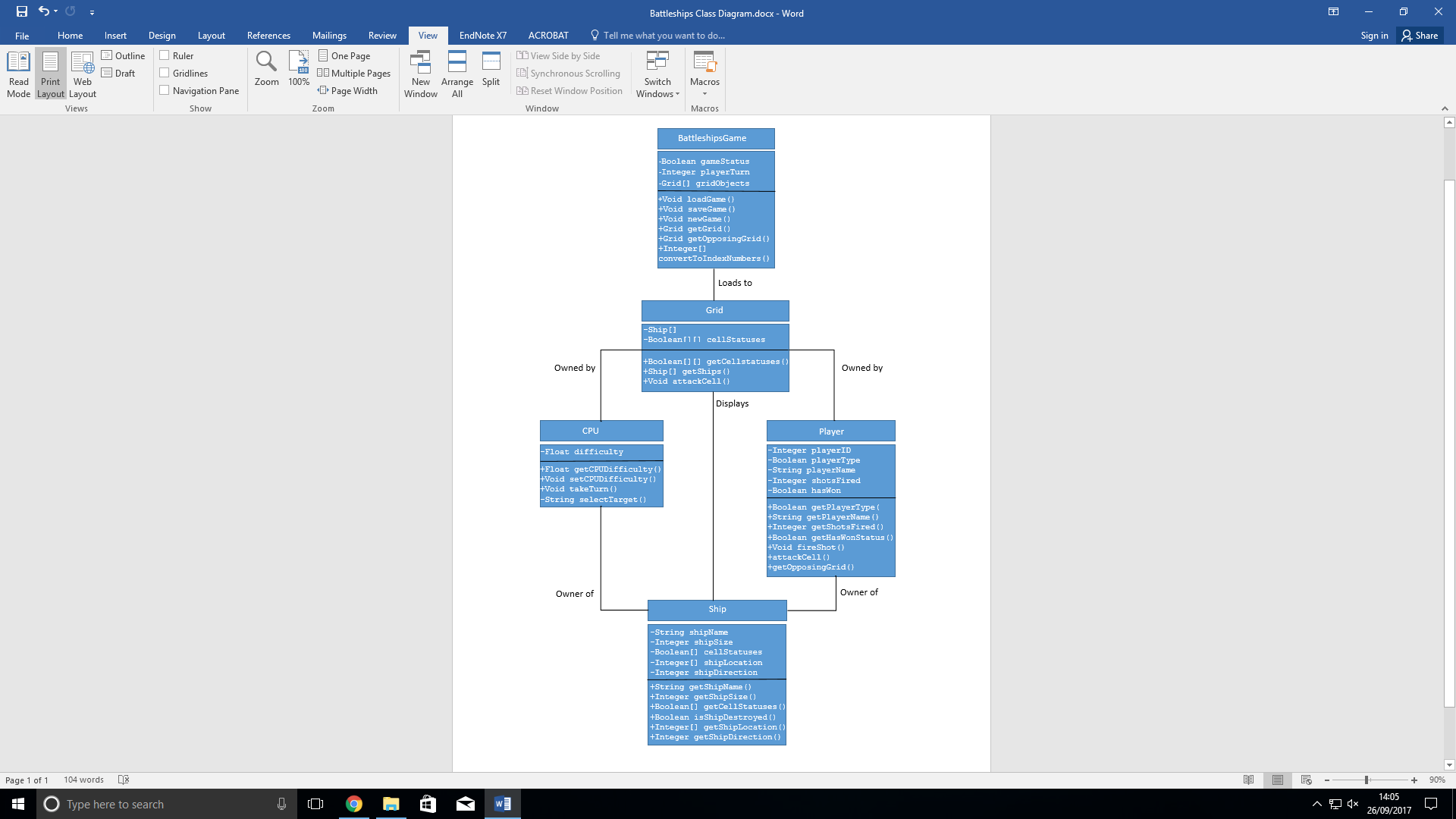
#### 

#### ***Class descriptions***

* **BattleshipsGame –** Top-level coordinator class that contains methods for the menus. Also contains the information on a particular game of Battleships, including the Grid objects and Player objects, and miscellaneous information such as whose turn it is.
  + **Methods**
    - **Void main() –** The main method that calls the first menu and creates a game object
    - **Void mainMenu() –** Gives the user a choice between starting a new game, load game or new multiplayer game and calls methods corresponding to the choice made.
    - **Void gameMenu() –** asks user for a choice between fire shot which asks the user for coordinates, save game which calls the saveGame() method in the grid class or the user can close the program
    - **Void gameMenuMP() -** Almost the exact same as the game menu class but it is a multiplayer version. I has an if statement which decides whos turn it is
* **Player** – Contains information such as name, score and ship location. This class is not used directly, but is inherited by HumanPlayer and CPUPlayer.
  + **Attributes**
    - **Integer shotsFired** – Integer containing the player’s score.
    - **Boolean hasWon** – Boolean containing whether the player has won or not. (true if the player has won, false otherwise)
  + **Methods**
    - **Void incrementShotsFired()** – increases the shots fired by 1 after every shot.
    - **Void decreaseShotsFired()** – decreases shots fired back to previous value if invalid position entered.
    - **Integer getShotsFired()** – Returns player score as a integer
    - **Boolean getHasWon()** – Returns the player status as a boolean.
* **Computer** – Controls the actions of the computer-controlled player.
  + **Attributes**
    - **Ship[] ships - an array of the computer's ships and positions**
  + **Methods**
    - **Computer()** – Constructor method for when starting a new game.
    - **Void fireRandomShot()** – Randomly chooses a position on the grid and checks if it is valid or not and then calls method to update its own grid
* **Grid** – Contains the ship objects on a particular grid, as well as a record of which cells have been attacked.
  + **Attributes**
    - **String[] shipGrid** – 2D Array of Ship positions
    - **Int row - amount of x positions**
    - Int column - amount of y positions
  + **Methods**
    - **Grid()** – Constructor method, creates a new empty grid.
    - **Void displayGrid** – Displays the grid as “X”s, “O”s and “-”s to the user.
    - **Ship[] placeShips()** – Calls methods that creates ships, displays information about the ships and checks if it is a valid position
    - **Void updateShipGrid()** – This method goes through every position on the ship and checks if it is out of bounds and calls a method to check if it is overlapping other ships
    - **Void clearOverlappedShip()** – Resets overlapped ships grid to previous ship code
    - **Ship createShips()** – Used to create an array of ship objects by calling constructor from Ship class
    - **Boolean attackCell()** – Converts the input target coordinates to index numbers then checks to see if the cell has already been attacked, returning true or false depending on this. each ship on the grid is checked to see if it occupies the cell.
    - **String updateCellStatus()** – increases number of cells hit and prints message if a ship has been sunk
    - Boolean checkIfShipPresent() - Checks if a ship is present where the target coordinates are
    - Void saveGame() - Writes ship objects and coordinates to an external text file
    - Void loadGame() - Reads a file and parses it into arrays which is assigned to the objects, then the coordinates below are read in to figure out what positions have been hit
    - Void checkIfEndGame() - increments the number of ships sunk variable if a new ship is sunk and if shipsSunk = 9 then the game is over and asks user if they want to start a new game
* **Ship** – Contains information on individual ships such as name, size, status, location, and direction.
  + **Attributes**
    - **String shipName** – The name of the ship (eg. ‘Cruiser’, ‘Battleship’)
    - **Integer shipSize** – The length of the ship as an integer
    - **Integer shipX** – The x coordinate of the ship
    - **Integer shipY** – The y coordinate of the ship
    - **Integer shipDirection** – The direction that the ship is orientated . (0=horizontal, 1=vertical)
    - **integer cellStatuses** – the amount of parts of the ship that are sunk
    - String shipCode - The abbreviation of the ship name e.g. C1, S3
  + **Methods**
    - **Ship()** – Constructor method for when starting a new game. Creates a random coordinate and orientation
    - Void increaseCellStatuses() - Increments the the parts of the ship sunk by 1
    - **String getShipName()** – Returns the name of the ship.
    - **String getShipCode()** – Returns the code of the ship.
    - **Integer getShipSize()** – Returns the size of the ship.
    - **int getCellStatuses()** – Returns the statuses of the ship’s cells.
    - **Integer getShipX()** – Returns the ship’s x position.
    - **Integer getShipY()** – Returns the ship’s y position.
    - **Integer getShipDirection()** – Returns the ship’s direction.

#### 

#### ***Class diagram***



#### 

#### 

#### 

#### 

#### 

#### ***Pseudocode***

**BattleshipsGame.mainMenu()**

**{**

**Boolean exit=false**

**DO**

**{**

**displayMenuChoices()**

**GET choice**

**IF choice=1{**

**displayGrid()**

GET VALID x ;

GET VALID y;

myGrid.attackCell(x,y);

} ELSE IF choice=2{

myGrid.saveGame(ships);

}ELSE IF choice = 0{

exit=true;

} while(exit=false);

Grid.fireShot()

{

boolean attacked = getCellStatus(x,y);

If attacked=true{

Display error message;

battleshipsGame.mainMenu();

}

ELSE IF attacked = false AND shipGrid[x][y]=empty{

Display “miss”;

shipGrid[x][y]=”X”;

}

ELSE IF attacked = false AND shipGrid[x][y]!=empty{

Display “hit”;

updateCellStatus(x,y);

shipGrid[x][y]=”O”;

}

displayGrid();

Return true;