Welcome to Battleship

This is a text oriented game based on the ever popular Battleship game by Milton Bradley. There are a few differences.

# Object of the Game

The object of the game is to sink all 5 ships in as little turns as possible with a max score of 20 the goal is to get a ratio of 1, score 20 divided by turns 20, this is a perfect score for the game.

## Game Modes

There are three game modes:

1. The first is a single player mode with unlimited turns seeing how close to a perfect score you can get.
2. The second is a single player mode where you only have 50 turns to sink all battleships, if not complete, THE BATTLESHIPS SINK YOU! Again the goal is to get the perfect score in 20 turns.
3. The third mode is a two player mode where each player has a board of 5 ships, the goal is to sink all 5 battleships before the other player, and of course to get the perfect score so you can brag to all your friends of how good you are!

## Ships

There are 5 ships to destroy in the game each taking a different amount of hits to sink:

* Carrier: 6 hits to sink and is displayed as so ######
* Battleship: 5 hits to sink and is displayed as so #####
* Cruiser: 4 hits to sink and is displayed as so ####
* Submarine: 3 hits to sink and is displayed as so ###
* Destroyer: 2 hits to sink and is displayed as so ##

## Board

The layout of the board consists of a 10x10 grid where the columns are indicated by letters A-J and the rows are indicated by a number 0-9. The ships are randomly generated onto the boards where the ships are indicated by a string of # according to how long they are. The ships can be placed horizontally or vertically in any way on the board and can be right next to each other. An example of a randomly generated board is shown in Figure 1.

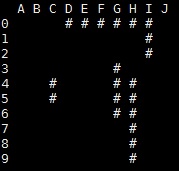


Figure 1: Board Example

## Playing the Game

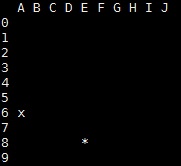


Figure 3: Board in play example

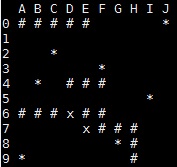


Figure 2: Board surrendered example

Once you start the game the board is randomly generated and is not shown to the player. The player will continuously choose locations on the board to attempt to attack a ship, if the hit is a success the system will prompt “Hit!” and show an ‘x’ on the location the player chose to attack. If the hit is a failure then the system will prompt “Miss!” and show a ‘\*’ on the location the player chose to attack, an example of a board in play is shown in Figure 2. If a player choses to surrender the game in the attack phase the player enters ‘S’ followed by any integer and the system will print out the surrendered board with the placement of all the ships along with the previous hits and misses which is shown in Figure 3.

Class Descriptions

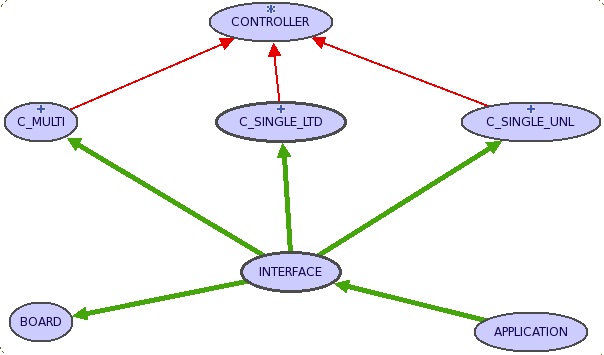


Figure 4: BON diagram for battleship

There are three main classes that make up the entirety of the Battleship game which are BOARD, INTERFACE and CONTROLLER. The BON diagram for the Battleship game system is shown in Figure 4, where all the relationships can be seen.

# BOARD

The BOARD class is an ADT and is the Model class. It has all the features that change the state of the board and is controlled by the INTERFACE class. The board is a 2-D array using the ARRAY2 abstract class of class CHARACTER and has a make which instantiates the object and creates the board filling it with empty spaces, ‘ ‘.

## Features

The BOARD class has many features new\_board, display, display\_solution, random\_int, random\_char, draw\_ship, check\_one, check\_two, check\_three, check\_four, attack, and toint. The features that change the state of the board are new\_board, draw\_ship, and attack. The rest of the features are used in the ones to change state of the board.

### new\_board

This feature simply fills the board with its proper columns and rows where the columns are A-J and the rows are 0-9 and the top right corner of the 2-D array is an empty spot. Then the feature calls the random\_char feature to generate random placements for the ships.

### random\_char

This feature randomly generates the placement of the head of a ship and ensures that the ship will be confined to the size of the 2-D array. This feature calls the random\_int feature to get the head placement of the ship, and then calls the draw\_ship feature to draw the rest of the ship, this feature loops until all ships are randomly generated and drawn.

### random\_int

This feature generates a random number which will be an INTEGER coordinate value factor to be returned. The value returned is an INTEGER greater than or equal to zero. This is the post-condition contract that must be ensured when returning the value:

ensure

result\_check: Result >= 0

--Check to ensure result is integer greater than 0

The returned value is checked against in random\_char to see if it is in bounds of the 2-D array if not it is called again.

### draw\_ship

This feature takes in an INTEGER as a parameter of which is the size of the ship to be drawn. The pre-condition contract for this feature is that the size of the ship must be greater than or equal to one and less than or equal to five and is as shown below:

require

min\_max\_ship\_size: i >= 1 and i <= 5

--Requires that size of the ships are between 2 and 6 units

The feature then loops and gets a random INTEGER generated by random\_int that determines the orientation of the ship depending on whether the ship will be oriented down, right, up, or left the feature calls check\_one, check\_two, check\_three, check\_four respectively.

### check\_one

This feature takes in an INTEGER that is the size of the ship and ensures that it can be drawn downwards. It has the same pre-condition contract as draw\_ship as it is using the same value when it is passed from draw\_ship:

require

min\_max\_ship\_size: i >= 1 and i <= 5

--Requires that size of the ships are between 2 and 6 units

The feature returns a BOOLEAN value that tells draw\_ship whether the ship can be drawn downwards or not and has a post-condition contract that the returned value reference is not void as so:

ensure

--Return value must not be void

result\_check: Result /= void

### check\_two

This feature is the exact same as check\_one in pre and post conditions and has similar implementation but this feature returns a BOOLEAN value as to if the ship can be drawn rightward.

### check\_three

Again this feature is the same as the two above in pre and post conditions and similar implementation but returns a BOOLEAN value as to if the ship can be drawn upward.

### check\_four

Once again the same as above but returns a BOOLEAN value as to if the ship can be drawn leftward.

### display

This feature displays the contents of the current board to the players with the ships hidden, this is called anytime an attempt to attack is made to show the current state of the board.

### display\_solution

This feature displays the contents of the board to players after they have won, been defeated or surrendered. The board displayed has all attempts to attack, hits and misses, and the location of the remaining ships if any.

### attack

This feature performs the attack command and changes the state of the board. The feature takes in two parameters and INTEGER which is the X coordinate on the board and a CHARACTER which is the Y coordinate on the board and the Y coordinate is transferred to an INTEGER by the toint feature that maps the character to its respective coordinate on the board. The pre-condition contracts for this feature are that the INTEGER parameter must be greater than or equal to two and less than or equal to eleven, this is due to the domain range of the board, the second is that the CHARACTER value must be an alphabetic character this is due to the domain range A-J. The pre-condition is as so:

require

--Parameter checks to ensure they are valid

check\_c1: c1 >= 2 and c1 <= 11

check\_c2: c2.is\_alpha

The feature returns a STRING that describes the result of the attempted attack, if the attempt is a success the return value is “Hit!” and is placed accordingly on the board. If the attempt is a failure the return value is “Miss!” and is placed accordingly on the board. If the attempt is on a location on the board that has been previously chosen for an attack then the return value is "You have already targeted this cell!" and the player loses a turn. Finally if the attempt is an invalid location on the board the return value is "Invalid cell!" and the player loses a turn. Finally we must ensure the post-condition is met, that the return value must not be a void reference as so:

ensure

--Ensure that the return value is not void

result\_check: Result /= void

### toint

This feature converts the CHARACTER coordinate value to its respective place on the board. It takes in a CHARACTER value as a parameter which is the column value on the board A-J and returns an INTEGER value that is its respective place on the board. The pre-condition contract is that the CHARACTER parameter must be an alphabetic character as so:

require

--Require that input parameter is a character

check\_char: c.is\_alpha

This feature has a post-condition such that the result cannot be a void reference and that the return value is of the proper domain range of the board being, result must be greater than or equal to two and less than or equal to eleven, even though a negative value is returned else the character is not found from previous conditions the value should never be a negative value. The post-condition is described as so:

ensure

--Ensure that result is not void

result\_check: Result /= void

--Ensure result is in range

range\_check: Result >= 2 and Result <= 11