Battleship

EECS3311 Project Report

David Iliaguiev, 201479830, davidili
Li Yin, 211608973, yinl1
Ting Fai Cheung, 211067162, tfcheung

8/3/2016

Contents

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 obtaining a high score, for every successful attack the player gains 10 points, for every miss the player gets -1 points.

Game Modes

There are three game modes:

1. The first is a single player mode with unlimited turns seeing how high of a score the player 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 highest score.
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 highest score so you can brag to all your friends of how good you are!

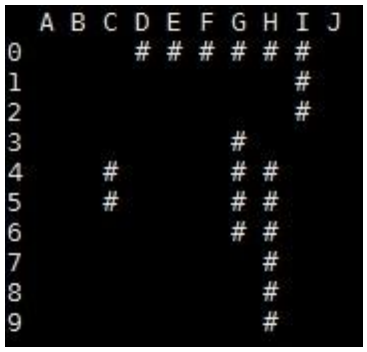
Ships

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

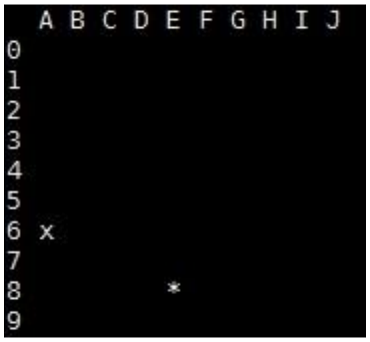
The layout of the board consists of a 10x10 grid where the columns are indicated by letters A-J and the rows are indicated by a number 0-9. The ships are randomly generated onto the boards where the ships are indicated by a string of # according to how long they are. The ships can be placed horizontally 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.



```
  A B C D E F G H I J
0       # # # # # #
1                 #
2                 #
3             #
4     #       # #
5     #       # #
6             # #
7               #
8               #
9               #
```

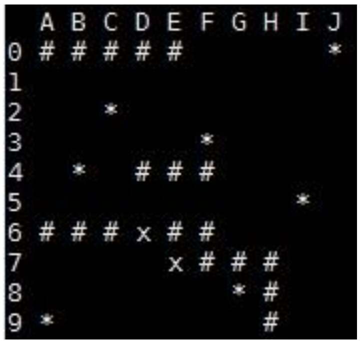
Figure 1: Board Example

Playing the Game



```
  A B C D E F G H I J
0
1
2
3
4
5
6 x
7
8         *
9
```

Figure 3: Board in play example



```
  A B C D E F G H I J
0 # # # # #         *
1
2     *
3           *
4   *   # # #
5                 *
6 # # # x # #
7         x # # #
8             * #
9 *               #
```

Figure 3: 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’ or ‘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

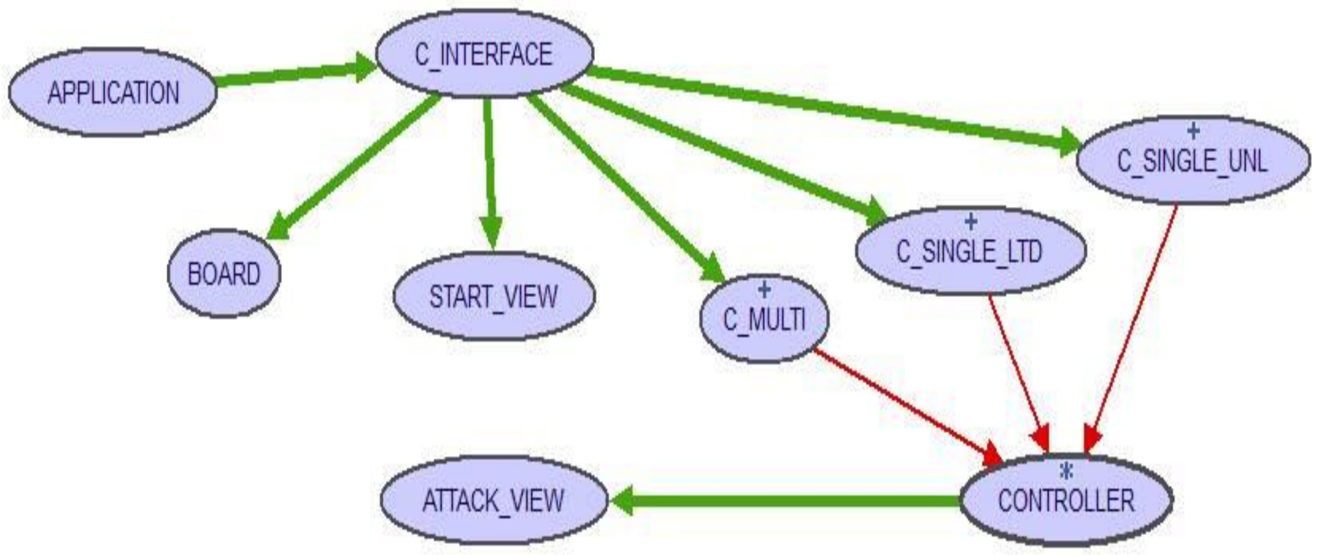


Figure 4: BON diagram for battleship

There are three main classes that make up the entirety of the Battleship game which are BOARD, C_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
```

Controller Family

The CONTROLLER is a deferred class and a parent to its separate game mode controllers C_SINGLE_UNL the single player unlimited mode, C_SINGLE_LTD the single player limited mode with 50 turns, C_MULTI the two player mode and the BON diagram for the family tree is shown in Figure 5.

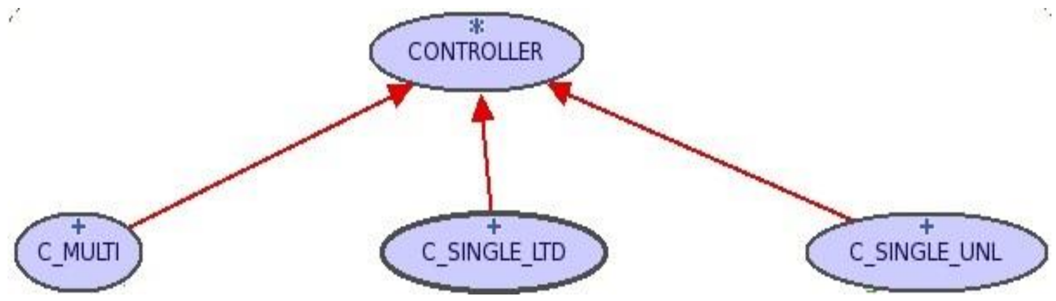


Figure 5: BON diagram for the CONTROLLER family

CONTROLLER

The CONTROLLER class is deferred it has three deferred attributes done, score1, and score2 and has three effective features that are inherited by its children check_input, refresh, and display_result.

Features

done

The done attribute is a BOOLEAN value that determines the status of the current game being played.

score1

The score1 attribute is an INTEGER value that represents the number of successful attacks the single player or player 1 in two player modes have gotten.

score2

The score2 attribute is an INTEGER value that represents the number of successful attacks player 2 has gotten in two player mode.

check_input

The check_input feature takes in two parameters, a CHARACTER being the column and an INTEGER being the row to attack. This feature is called in the effective child classes to error check and prompt that the location chose is valid or invalid. The feature then returns a BOOLEAN value to tell the child classes whether it can continue or try inputting another location to attack. This feature does not have any pre-conditions as it is purely to error check and prompt the player to reselect. On the other hand it is important to ensure the post-condition contract that the returned value reference is not void as so:

```
ensure
        --Ensure that result is not void
        result_check: Result /= void
```

refresh

The refresh feature resets the score1 to 0, score2 to 0, and done to false. This is done so that if a game is completed and a new game is started in the same mode, the object will clear and start from the beginning for a new game.

display_result

The display_result feature takes in two parameters, score that is an INTEGER and turn that is an INTEGER. This feature is used by the effective children to display the current score and turn after an attempt to attack. The pre-condition for the parameters are that score and turn references are not void and that both are greater than or equal to zero and is shown as so:

```
require
        --Requires that the input parameters are not void and >= 0
        score_check: score /= void and score >= 0
        turn_check:     turn /= void and turn >= 0
```

The post-condition contracts are also for the parameters and ensure that again the same condition as the precondition is met and that neither the score nor the turn has changed and are as so:

```
ensure
        --Ensure that no changes have been made to score and turn when displaying result
        score_unchanged: score = old score
        turn_unchanged: turn = old turn
        --Requires that the input parameters are not void and >= 0
        score_check: score /= void and score >= 0
        turn_check:     turn /= void and turn >= 0
```

The CONTROLLER class has class invariants that are inherited by its children as well, on is the done check that done equals true implies exit current game and done equals false implies that current game is in progress. The other is a check for the scores that score1 and score2 are greater than or equal to zero, the invariants are as so:

```
invariant
        --done_check: done = true -> exit current game AND done = false -> current game in progress
        --Ensure player scores are greater than or equal to 0
        score_check:    score1 >= 0 and score2 >= 0
```

C_SINGLE_UNL

The C_SINGLE_UNL class is a controller for the single player unlimited mode. This class inherits all the attributes and features of the CONTROLLER class and creates an effective feature interact.

Features

The features of this class are those inherited from CONTROLLER and interact.

interact

The interact feature implements the simple player unlimited game itself, it takes in a BOARD as a parameter, this parameter is given by the interface controller C_INTERFACE when a board is created then interact plays the game. The pre-condition contracts for this feature are that the board must not be a void reference and checking that the score1 was reset to zero being that score1 must be greater than or equal to zero. The pre-conditions are as so:

```
require
        --Requires that board is not void
        board_not_null: g /= VOID
        --Ensure that score is refreshed for each new game
```

The feature uses the ATTACK_VIEW class to prompt the player to enter a coordinate and takes in the column and row value, all error checking is done within the loop. The game continues until ‘s’ or ‘S’ is inputted followed by an integer or the player has sunk all battle ships and done is changed to true and the game is exited and control goes back to the interface controller class C_INTERFACE. The post-condition for this feature is that done is equal to true after execution and is as so:

```
ensure
        --Ensure that done is true if game exits
        done_check: done = true
```

The class inherits its class invariant from its parent class CONTROLLER.

C_SINGLE_LTD

This class is the controller for the single player limited game where the player has 50 turns to sink all ships otherwise it is game over and is the child of the CONTROLLER class.

Features

The feature for this class are inherited by the CONTROLLER class and also has its own interact feature.

interact

The interact feature is the same as in C_SINGLE_UNL, the same pre-conditions and the same class invariant. The only differences are that this interact feature maintains a counter on the amount of turns left and prompts the player so, the other is the post-condition for this is feature is that done must equal true at the execution or the turns left counter equals zero and is as so:

```
ensure
        --Ensure that done is true or counter is 0 if game exits
        --done_check: done = true OR  counter = 0
```

C_MULTI

This class is the controller for the two player mode the players get two randomized boards and go turn by turn until one has sunk all the battleships and wins or a player has surrendered.

Features

This class' features are the ones inherited from the CONTROLLER class and its own interact feature much like the other two game mode interact features.

interact

This interact feature acts just as above with a little more complexity having to double check all locations and board prompting and that this interact feature takes in two parameters, two BOARDs one for each player. This feature uses both the score1 and score2 that are inherited from the CONTROLLER class, one for each player. Therefore the pre-condition is a little different than that of the previous interact features, the pre-condition contracts are that both boards cannot be a void reference and that both score1 and score2 has been properly reset to zero, the contracts are as so:

```
require
        --Requires that both gameboards are not void
        boards_not_null: g1 /= VOID and g2 /= VOID
        --Ensure that score is refreshed for each new game
        score_refresh: score1 = 0 AND score2 = 0
```

This interact feature acts in the same way as the others, it uses the ATTACK_VIEW to prompt and take input of the locations for both players, error checks them and prompts accordingly. The post-condition contract is the same as the C_SINGLE_UNL post-condition, that the done attribute must be true at the end of execution and is as so:

```
ensure
        --Ensure that done is true if game exits
        done_check: done = true
```

The class invariant for this class is again like all the rest of the mode controllers and is inherited from the CONTROLLER class.

ATTACK_VIEW

The ATTACK_VIEW class is a view class that is used by the game mode controllers and prompts the users for input and takes in the input and send its back to the game mode controllers for error checking and processing.

Features

The features for this class are display_attack, display_attackp1, display_attackp2, get_char, get_int, and convert_int. The displays are for prompting the players for input, the gets are to get the location inputted by the players.

display_attack

The display_attack feature is used for the single player unlimited and limited modes and just prompts the player for attack location input.

display_attackp1

The display_attackp1 feature is used for the two player mode and just prompts player 1 for attack location input.

display_attackp2

The display_attackp2 feature is used for the two player mode and just prompts player 2 for attack location input.

get_char

The get_char feature is called by the controller to get the column location input in range A-J, the error checking is done in the game mode controller classes. The feature returns a CHARACTER value that is checked against in the post-condition contract that the return value reference is not void and the return value is alphabetic and is as so:

```
ensure
        result_not_void:        Result /= VOID
--      result_is_alpha:        Result.is_alpha
--      This ^ is ensured by the controller on return
```

get_int

The get_int feature is again called by the game mode controller but this time for the row location input in range 0-9, the error checking is done by the conver_int feature converting a string into a character then integer. The feature returns an INTEGER value that is checked against in its post-condition contracts that the return value reference is not void and the return value is in the range zero to nine and is as so:

```
ensure
        result_not_void:        Result /= VOID
--      result_range:           Result >= 0 and Result <= 9
--      This ^ is checked by controller on return
```

convert_int

The feature convert_int is used for the get_int feature above. It takes a CHARACTER as a parameter and returns an INTEGER corresponding to the proper integer value in the rows domain range of the board 0-9. It has the same post-condition as the get_int feature as well that the return value is not void and is in the range 0-9 and is shown as so:

```
ensure
        result_not_void:        Result /= VOID
        --      result_range:   Result >= 0 and Result <= 9
        --      This ^ is checked by controller on return
```

INTERFACE_ACCESOR

This class is a deferred class to provide unique access to the game controller C_INTERFACE

Features

This class has one object attribute that is of type C_INTERFACE and one feature is_real_interface.

is_real_interface

This feature checks the Singleton status of the C_INTERFACE object, that there is only one, and returns a BOOLEAN value if there is indeed only one object.

The class invariant for this class is to ensure that there is only on instance of the interface and is as so:

```
invariant
        --Ensure that there is only one instance of interface
        real_interface: is_real_interface
```

NEW_INTERFACE

This class is a child of INTERFACE_ACCESSOR and inherits its class invariant to ensure that there is only one instance of the game controller E_INTERFACE. This is the class that is created by the APPLICATION class to start the system and the game.

Features

This class' only feature is the_interface, this feature creates the one and only instance of the game controller E_INTERFACE and returns the reference to the E_INTERFACE object.

C_INTERFACE

The C_INTERFACE class is the main game controller class that is created from the APPLICATION class. It starts the game and let the player select the option of different game modes. It uses the START_VIEW class to prompt the player to choose the game and get the input to decide what mode is played. It instantiates two BOARD objects and three CONTROLLER objects, one for every game mode, so a C_SINGLE_UNL, C_SINGLE_LTD, and C_MULTI objects.

Features

The C_INTERFACE class only has a make feature and the rest are attributes for instantiation of objects. The only other feature is the exit attribute which is a BOOLEAN to decide whether the player exits the game or not.

This class has some class invariants to maintain the state of the game controller, these invariants are that neither of the game boards should be a void reference, none of the game mode controllers should be a void reference and the exit equals false implies that the game is still being played, the invariants are shown as so:

```
invariant
        --Class invariants to ensure that gameboards and control modules are not void
        board_not_null: gameboard1 /= VOID and gameboard2 /= VOID
        controllers_not_null: cmulti /= VOID and csingleltd /= VOID and csingleunl /= VOID
        not_exit: -- exit = false implies the game is still being played
```

START_VIEW

The START_VIEW class is used by the C_INTERFACE class to get the player's input and contains the texts for the menus. It assists C_INTERFACE to get the right thing that the player requires.

Features

The features of the START_VIEW class are get_start, display_welcome, and display_instructions these features are used for the interface to the player.

get_start

The get_start feature is requested by the game controller class to get the input from the player to determine the game mode or to exit. This feature returns an INTEGER value that is in the range of 1-5 depending on which game mode the player wishes to start. The feature has post-condition contracts to verify the input of the player such that the return value is not a void reference and that the return value is greater than or equal to one and less than or equal to 5 and are as so:

```
ensure
        result_not_void:        Result /= VOID
        --      result_range:   Result >= 1 and Result <= 5
        --      This ^ should be ensured but is error checked in C_INTERFACE
```

display_welcome

This feature simply prompts the player with the game welcome message and the instructions on what values to input for choosing a game mode or to exit.

display_instructions

This feature simply prompts the player with the game instructions, how the game is played and the ultimate goals

Design Patterns

The design patterns that we chose are the Singleton pattern as it is required for the game controller but in the game controller another structural pattern emerges as well. Another pattern chosen is the MVC, Model View Controller, pattern that is the backbone of our system, and lastly the Observer design pattern for the state changes of the BOARD objects.

Singleton

The objective of the Singleton pattern is to ensure that each class has exactly one instance and that they are accessible globally from known locations. This design pattern is the most obvious that is needed for our system design as the APPLICATION class creates one instance of the game controller class which is C_INTERFACE. There is ever only one instance of a C_INTERFACE object that is created and it is the object that starts the game and ends the game when the player chooses to exit via the ‘5’ command input. In addition, the single instance of this object is the underlying main class that is responsible for ensuring the successful implementation of all other classes.

To preserve singleton status, we have 5 participating components to this singleton pattern. The C_INTERFACE class is the class for which we want to make a singleton for the purpose of this game. It also serves as the single instance class as it creates a unique instance of itself and constantly checks that this instance of C_INTERFACE is indeed the same instance as the one created initially when the instance of the class is first created. INTERFACE_ACCESSOR is our singleton accessor as it provides us access to a unique instance of the C_INTERFACE class by declaring an access point. In addition, the INTERFACE_ACCESSOR deferred class also ensures that there is only one instance of interface in our entire game. NEW_INSTANCE class acts as an access point to C_INTERFACE, inheriting directly from INTERFACE_ACCESSOR. This class uses a once function to ensure that exactly one C_INTERFACE is created and subsequent calls to this access point will not result in the creation of a new C_INTERFACE instance. Finally, the user of this INTERFACE_ACCESSOR is APPLICATION itself. As soon as the game starts, APPLICATION will create an access point to C_INTERFACE and the series of classes above will ensure that only one interface will be created each time the game is run.

This design pattern is important in our system otherwise there would be multiple games started and considering that this is a text-based game played in the prompt, the terminal would be overloaded with prompts and the input to start the game would have many errors as there would be multiple instances of the game system unable to interact with any of them. In the case with our program, the pattern is applied immediately as soon as the game starts. Other cases of singleton pattern occur elsewhere in the game, in particular the game mode controller classes.

There are other singleton design patterns that are used in C_INTERFACE as well, such as only having one instance of each game mode controller. The C_SINGLE_UNL, C_SINGLE_LTD, and C_MULTI are ever only instantiated once, when the game controller gets input from the START_VIEW it decides what game mode controller to use, once the game mode controller is finished execution it gets reset and is possibly used again respective to the players input thereafter.

Implementation of the singleton pattern in Eiffel is simple and in the example with our game, it does support its objective of creating just one instance of a necessary class. In a sense this is a structural design pattern called Proxy, the game mode controllers are created no matter what they are surrogates or placeholder for the game controller to choose which to use based on the players input of which game mode he/she wishes to play.

Model View Controller (MVC)

This design pattern was chosen because it is ideal with user interactive systems such as a text-based input game. Another reason is that due to its inherent divide and conquer style it was easy to divide up work amongst our group members and bring our classes together to just plug and play once completed. It also made it easy to do testing and error checking for input values as the view would take in input that was requested by the controller, check for the correct ranges and contracts then report back to the controller, if the controller needed processing done it would send data to the model which would then process and either return a value or report back. In our case our model was the game board itself BOARD, this class did all state changes to the board anytime the controller asked to do something.

The relationships between the 3 modules are also quite straightforward. When changes to the state of the game occur, they are immediately handled by the Controllers. The INTERFACE initially deals with user selection at the welcome screen and is the master controller for the entire game. After selection occurs, control is passed on to corresponding game mode controllers which deal with the specified game modes. This signifies a relationship between various controllers. The controller updates the Model, which in our case is the game BOARD, with data and crucial information about what is required for each specific game modes. The controller also updates the view, which includes the START_VIEW and ATTACK_VIEW, presenting users with vital information and constant updates about further instructions and user inputs. This information will ensure that users understand that there is progress and the game is shifting from one state to the next. It also ensures that there is constant user interaction, which is crucial for an interactive game experience. Finally, the views also directly communicate with the model, requesting further data not provided by the controller.

The MVC diagram in Figure 6 shows the participants of MVC pattern and shows the flow of our system requesting and processing and reporting back, the system works as so:

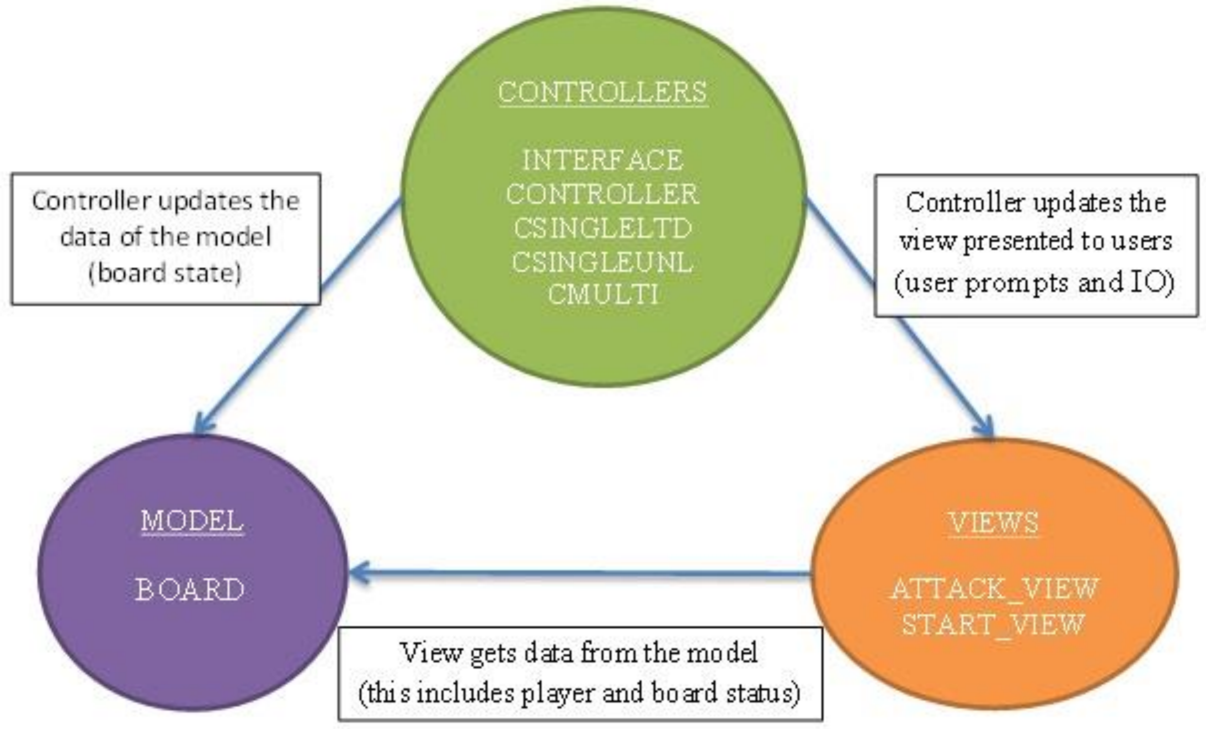

CONTROLLERS
INTERFACE
CONTROLLER
CSINGLELTD
CSINGLEUNL
CMULTI
Controller updates the data of the model (board state)
Controller updates the view presented to users (user prompts and IO)
MODEL
BOARD
VIEWS
ATTACK_VIEW
START_VIEW
View gets data from the model (this includes player and board status)

Figure 6: MVC diagram for Battleship system

- Model
    - The BOARD class is the model which contains all the features that controls the board, creating a new randomized board which is requested by the main game controller E_INTERFACE when a mode is chosen, when any actions such as attack is requested by the game mode controller from the CONTROLLER family the game mode controller requests from the model BOARD to change the state of the board
- Controller
    - There is the main game controller which is the E_INTERFACE class this game controller connects the whole system together, it requests its view, START_VIEW, to prompt the player for input the requests again to the view for the input from the player, it then requests the model BOARD to create standard empty game boards so that they can be ready for when the game mode controller randomly generates a board for play. Once the game controller determines what game mode the player wishes to play it requests the respective game mode controller to begin the game, once the game is finished the game mode controller is reset and returns to the game controller to either start a new game or exit the game entirely.
    - The game mode controllers are instances of the CONTROLLER family, the CONTROLLER class is deferred, for the sake of cleanliness and attractiveness the diagram shown in Figure 6 only has the parent CONTROLLER class shown but it is implied through inheritance that the game controller E_INTERFACE is actually working with CONTROLLERs child classes C_SINGLE_UNL, C_SINGLE_LTD, and C_MULTI as these are the game modes to be played. The game mode controller interacts with the BOARD to alter its states every time an attack is issued when the player gets input from the game mode controller's view ATTACK_VIEW.

- View
    - There is the START_VIEW, this view interacts with the main game controller and prompts the player for game mode input, the game controller then requests the input from this view to determine the game mode, and this view is the games main interface and interaction with the player.
    - There is the ATTACK_VIEW, this view interacts with the game mode controller, it prompts the user to input the location of the attack he/she wants to attempt, the game mode controller then requests error checks the location and if valid requests the attack to be processed from the model, the view then indirectly interacts with the model to prompt the player of the current state of the board and the players score, turn, and turns left.

Observer

The final design pattern that we chose is the Observer behavioural pattern, which states that if one subject changes its state, then all observers are immediately notified of these changes and updated accordingly. This pattern was chosen because the BOARD object that is created by the game controller has to be seen by almost all the classes, in particular the crucial game mode controllers and the views, thus each instance of a BOARD has a one-to-many relationship with all the other classes.

The two main participants of the Observer pattern are the subject, which is the game BOARD itself, and the Observer, which consists of START_VIEW and ATTACK_VIEW. Concrete subject are represented by the instances of BOARD, used for each designated game mode. These objects store the current state of the BOARD, which will be used by the Concrete Observers. Concrete Observer is represented by the three game mode classes, which store BOARD state information and constantly maintains reference to the concrete subject. These classes are: C_SINGLE_LTD, C_SINGLE_UNL, and C_MULTI.

The Observer pattern achieves its objectives by supporting broadcast communication by ensuring that all relevant classes are kept up to date on the state of the game BOARDS. In addition, multiple boards in multiplayer modes ensure that observers can depend on more than one subject at one time. Observers at the same time are allowed to change the state of the BOARDS based on user input while changes to BOARDS are immediately updated to the VIEW modules.

The game controller creates the instances of the object, the game mode controller interacts with the object and its class for processing, and the view class indirectly interact with the object and its model class to present to the player and for the game mode controller to continue to make attack operations on the object. The game controller will create 2 instances of the BOARD class at a time for play on for each player and all the connected classes need to see the changes in their states especially for the two player mode controller C_MULTI. This behavioural pattern is actually closely connected to the State behavioural pattern where the game mode controller changes the state of the object such as when an attack attempt is made and adds a hit or miss character to the board which alters the behaviour of the object as it now has to avert these new locations on the board, this makes the object appear to change its class to another.