EECS3311-W2019 — Project Report

Submitted electronically by:

Team members	Name	Prism Login	Signature
Member 1:	Haider Altahan	mrfirete	
Member 2:	Yunho Nam	namyun	
*Submitted under Prism account:		namyun	

* Submit under **one** Prism account only

Also submit a printed version with signatures in the course Drop Box

Contents

Documentation must be done to professional standards. See OOSC2 Chapter 26: *A sense of style*. Code and contracts must be documented using the Eiffel and BON style guidelines and conventions. *CamelCase* is used in Java. In Eiffel the convention is *under_score*. Attention must be paid to using appropriate names for classes and features. Class names must be upper case, while features are lower case. Comments and header clauses are important. For class diagrams, use the BON conventions, and use clusters as appropriate. Use the EiffelStudio document generation facility (e.g. text, short, flat etc. RTF views), suitably edited and indented to prevent wrapping, to help you obtain appropriately documentation (e.g. contract views). Each diagram must be at the appropriate level of abstraction. Use Visio for the BON class diagrams.

Your signature attests that this is your own work and that you have obeyed university academic honesty policies. Academic honesty is essentially giving credit where credit is due, and not misrepresenting what you have done and what work you have produced. When a piece of work is submitted by a student it is expected that all unquoted and uncited ideas and text are original to the student. Uncited and unquoted text, diagrams, etc., which are not original to the student, and which the student presents as their own work is considered academically dishonest.

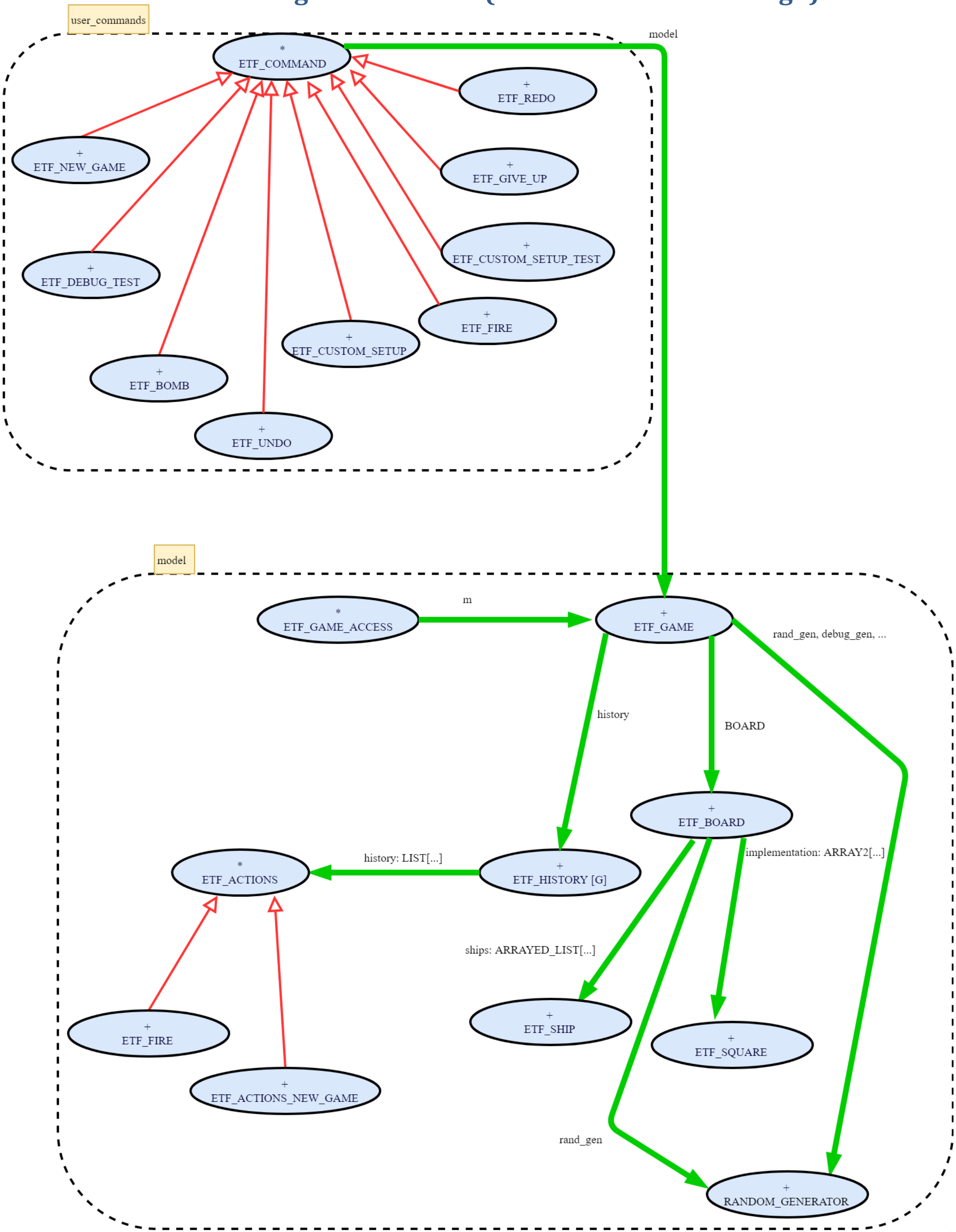
1. Requirements for Project Battleship

We were given the following requirements for our Battleship project:

The Battleship is a single-player game played against the computer. The computer randomly places the ships and it is the goal of the Player to sink the ships. The Player is given a set of shots and bombs. The game ends when either all the ships are sunk (Player wins) or all the shots and bombs have run out (Player loses). The game can be played normally using levels to determine the size of the board, the number of the ships, and the number of bombs and shots. The game can also be customized to allow board sizes anywhere from 4×4 to 12×12. The Player is given the option to ‘undo’ or ‘redo’ each move, and give up the ongoing game. This project also comes with debugging features that allow either the normal game or the custom game types to be played using ‘the debug mode’ to allow the Player to see where the ships are located and the ships are placed in a predetermined way so that acceptance tests can be used to spot any error. The project also allows the Player to give up any currently ongoing game.

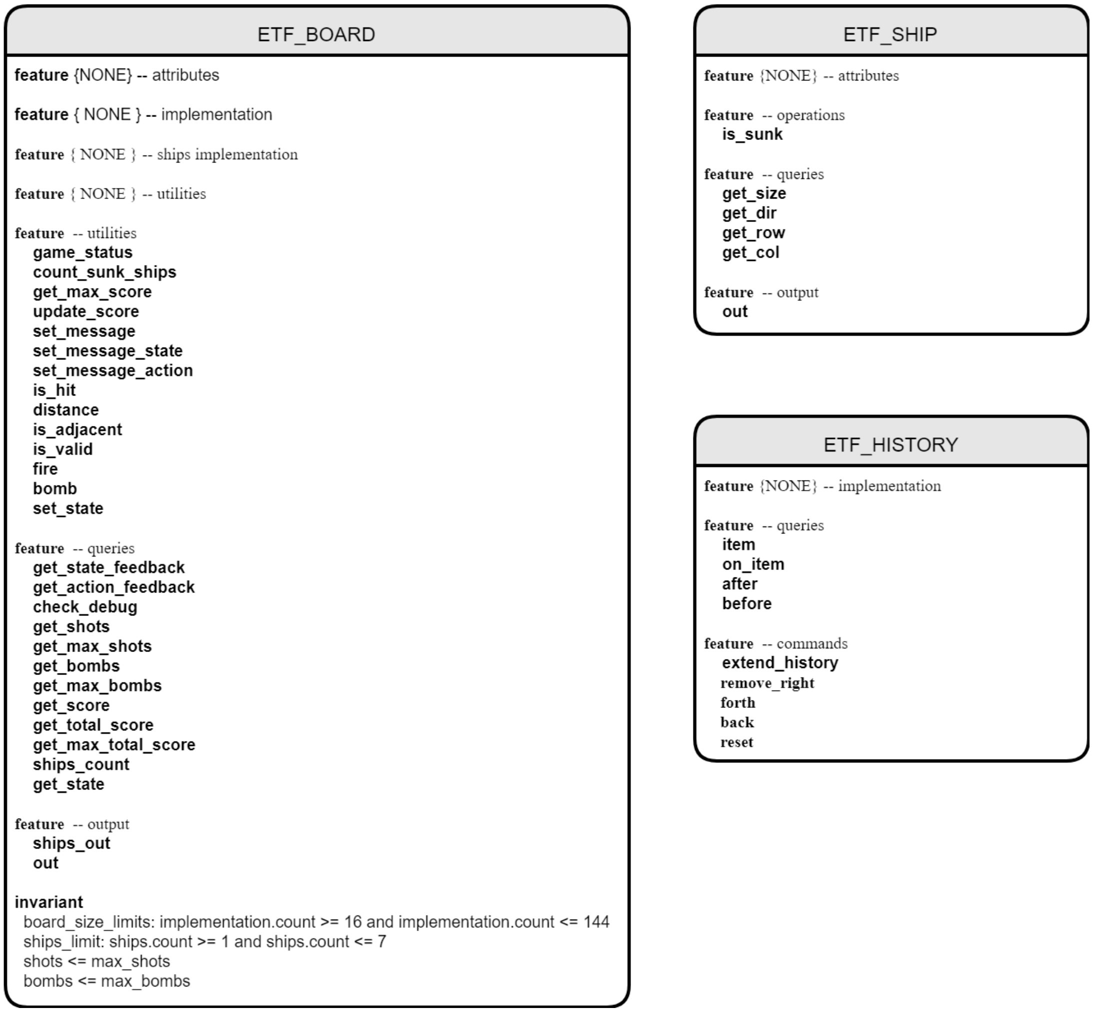
See battleship.ui.txt in the doc for the grammar of the user interface. The acceptance tests *at001.expected.txt*, *at002.expected.txt*, *at101.expected.txt*, *at102.expected.txt*, and *at103.expected.txt* describe some of the input-output behavior at the console.

2. BON class diagram overview (architecture of the design)



For the design choice, we wanted to keep it as generic as we possibly could. We used the singleton pattern, where each command used by user_command classes rely on model attribute from ETF_COMMAND. Then model is used to access the modules, so that if this design needs to accommodate any additional game, or any additional functionality, a new game or a new functionality can be added without any need to make changes to the design.

We have also consistently used the information hiding principle. As seen below in our UML diagram of encapsulated classes, any class that can be encapsulated are done so. The client of these classes have no access to the implementation nor the attributes, and modifying the implementations will not require the client to change any of their code. The modules that were not encapsulated are done so to maintain the singleton pattern, so any other module that uses them can have access to those modules.



For reusability and extendibility, we made our History module's type as ETF_ACTIONS. Our undo/redo module is designed, so that our History module can store any action for any game. Our undo/redo module is designed so that it can be inherited for any game and any additional functionality. This is further explicated in the 4th section of this document.

3. Table of modules — responsibilities and information hiding

1	ETF_Game	**Responsbility**: Set up ETF_Board when a new game is created and output appropriate string of the board states.	**Alternate**: None
	Concrete	**Secret**: None	

2.1	ETF_Board	**Responsbility**: Record the data of ongoing game's states.	**Alternate**: None
	Concrete	**Secret**: Implemented via ships (see 2.1.1) and squares (see 2.1.2)	

2.1.1	ETF_SHIP	**Responsbility**: Record of data of an existing ship on the board.	**Alternate**: None
	Concrete	**Secret**: None	

2.1.2	ETF_SQUARE	**Responsbility**: Record of a square representation on the board.	**Alternate**: None
	Concrete	**Secret**: None	

3	ETF_HISTORY[G]	**Responsbility**: Ordered collection of any actions used for undo/redo design pattern.	**Alternate**: None
	Concrete	**Secret**: Implemented via actions. See 4.	

4	ETF_ACTIONS	**Responsbility**: Abstract class for any actions used in any game.	**Alternate**: None
	Abstract	**Secret**: None	

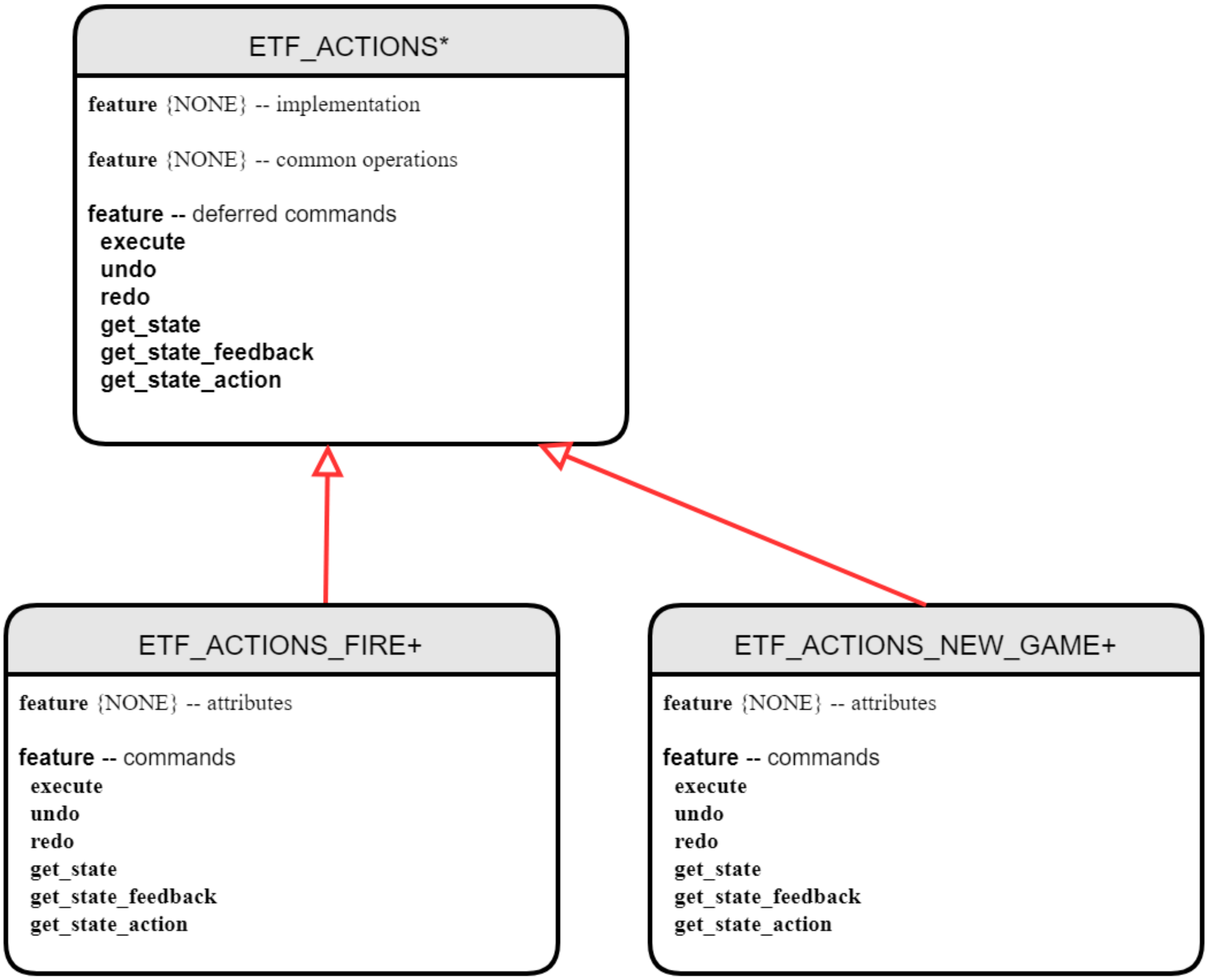
4.1	ETF_ACTIONS_FIRE	**Responsbility**: Record of board state before and after firing action.	**Alternate**: None
	Concrete	**Secret**: None	

4.2	ETF_ACTIONS_NEW_GAME	**Responsbility**: Record of board state before and after game creation.	**Alternate**: None
	Concrete	**Secret**: None	

5	RANDOM_GENERATOR	**Responsbility**: Randomly generates coordinates of ships from the input number of ships.	**Alternate**: None
	Concrete	**Secret**: None	

4. Expanded description of design decisions

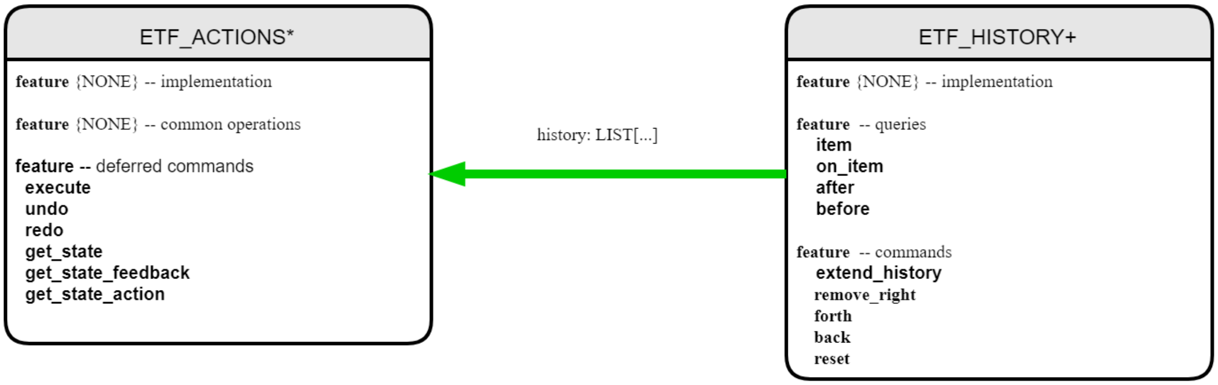
We believe that the most important module is the undo/redo module. This module is the subsystem that allows the game to undo/redo their actions. It is comprised of ETF_ACTIONS, ETF_ACTIONS_FIRE, and ETF_ACTIONS_NEW_GAME. As a part of the requirements of this project, we believe that this module in particular was important because its functionality is independent from other modules in the system. We provide below the UML overview of our undo/redo module:



ETF_ACTIONS_FIRE stores the state of the board before and after execution of firing. ETF_ACTIONS_NEW_GAME stores the state of the board before and after any attempts at creating a new board. The actual execution of the actions that changes the board occurs outside of these classes and the execute feature of this module only serves to store the board state. So for any client to use this module, all they have to know is how ETF_ACTION's deferred features work regardless of how many similar 'actions' are added to the module.

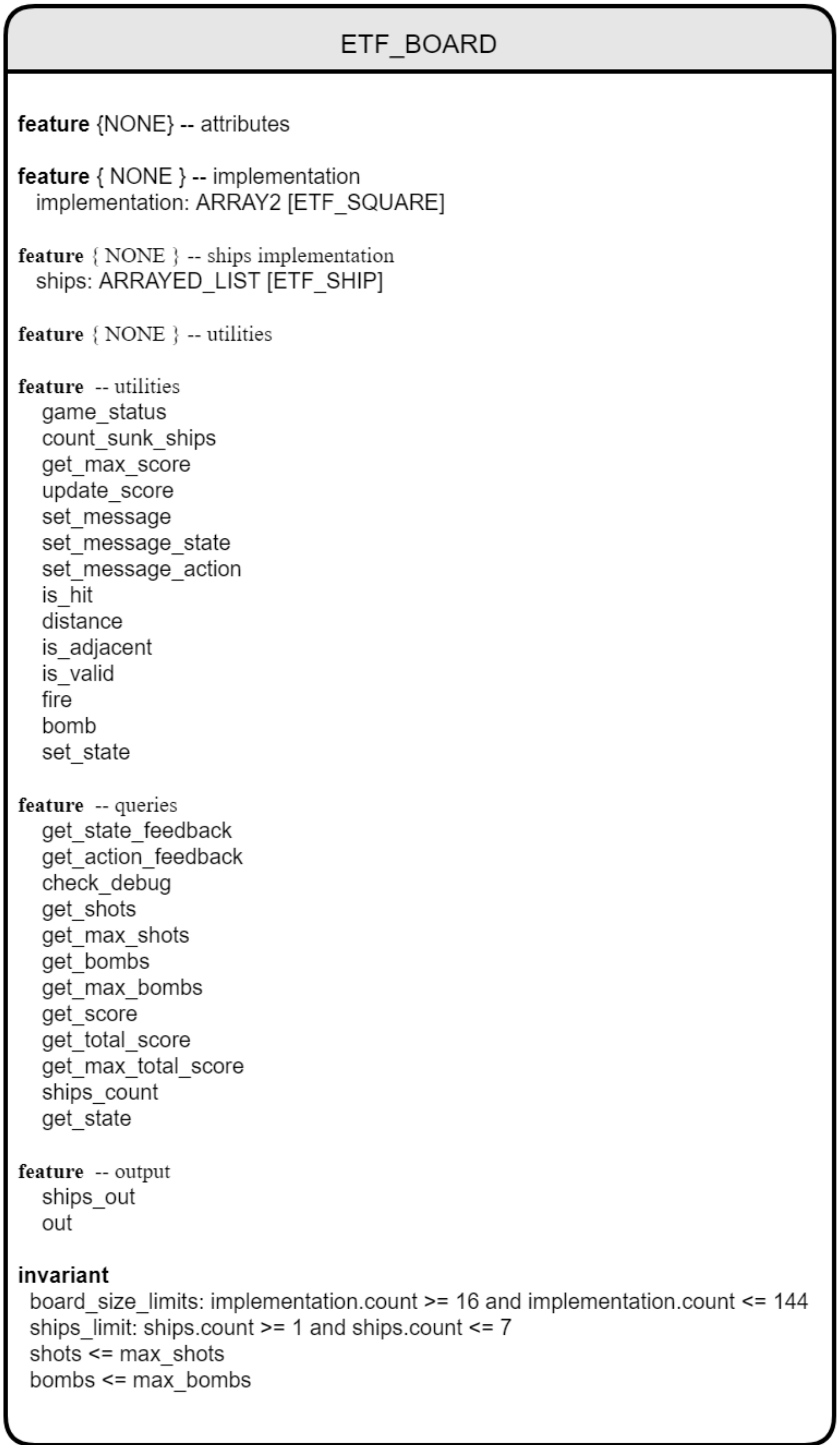
Furthermore, the implementation of ETF_ACTIONS is implemented using the singleton pattern of this system by referencing ETF_GAME. So any class that inherits from ETF_ACTION can have access to whatever game and whatever board that is available in the system. This completes the modularity design of ETF_ACTIONS so that we can use this module with any other system specification that is added to the system.

Since we are using polymorphism, we wished to avoid any error from dynamic binding. We will use the following UML diagram to illustrate how this was done:

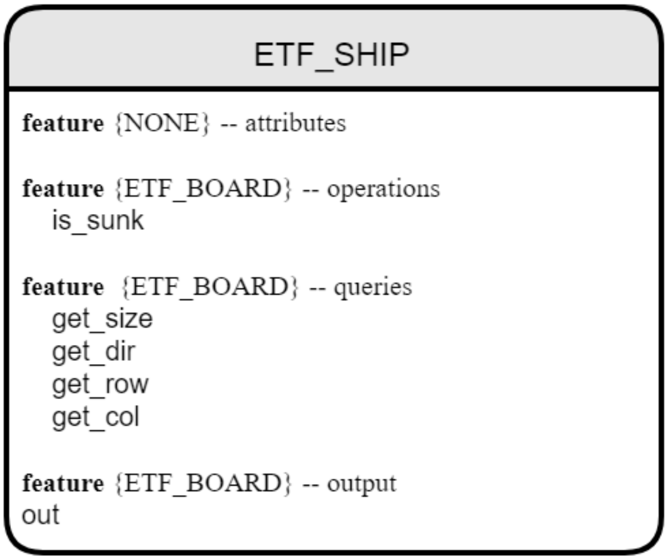


This shows the relationship between the undo/redo module and the ETF_HISTORY module. The implementation of ETF_HISTORY is: history: LIST[G]. G is set to type ETF_ACTIONS. To avoid any error from dynamic binding, the generic type ETF_ACTIONS is stored in the implementation's data structure, so that ETF_HISTORY is able to store any object classes that inherit from ETF_ACTIONS. This ensures that we will never have any issues with dynamic binding, because any object stored in ETF_HISTORY's implementation will inherit from ETF_ACTIONS.

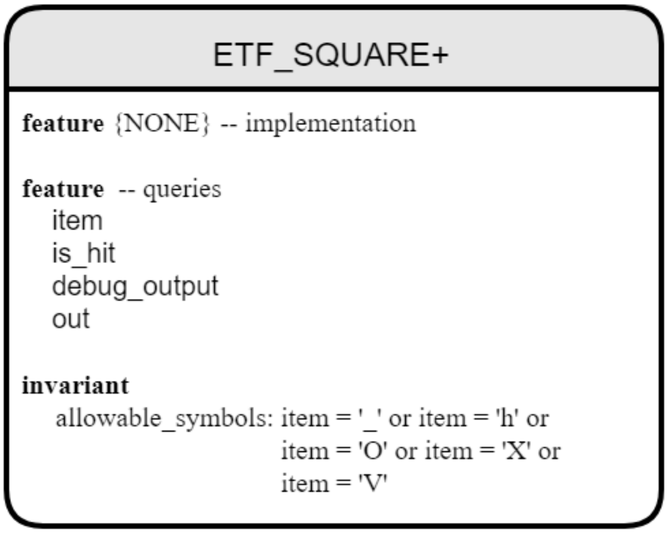
5. Significant Contracts (Correctness)



```
ETF_BOARD

feature {NONE} -- attributes

feature { NONE } -- implementation
  implementation: ARRAY2 [ETF_SQUARE]

feature { NONE } -- ships implementation
  ships: ARRAYED_LIST [ETF_SHIP]

feature { NONE } -- utilities

feature  -- utilities
   game_status
   count_sunk_ships
   get_max_score
   update_score
   set_message
   set_message_state
   set_message_action
   is_hit
   distance
   is_adjacent
   is_valid
   fire
   bomb
   set_state

feature  -- queries
   get_state_feedback
   get_action_feedback
   check_debug
   get_shots
   get_max_shots
   get_bombs
   get_max_bombs
   get_score
   get_total_score
   get_max_total_score
   ships_count
   get_state

feature  -- output
   ships_out
   out

invariant
 board_size_limits: implementation.count >= 16 and implementation.count <= 144
 ships_limit: ships.count >= 1 and ships.count <= 7
 shots <= max_shots
 bombs <= max_bombs
```



```
ETF_SHIP

feature {NONE} -- attributes

feature {ETF_BOARD} -- operations
   is_sunk

feature  {ETF_BOARD} -- queries
   get_size
   get_dir
   get_row
   get_col

feature {ETF_BOARD} -- output
out
```



```
ETF_SQUARE+

feature {NONE} -- implementation

feature  -- queries
   item
   is_hit
   debug_output
   out

invariant
    allowable_symbols: item = '_' or item = 'h' or
                       item = 'O' or item = 'X' or
                       item = 'V'
```

Our board module has the most significant contracts in our system. The above UML diagrams show the hidden implementations and the invariant of ETF_BOARD. As the supplier, we want to make sure that our client can freely use our code without worrying about its implementations. That is why implementation and ships use the information hiding principle. Normally, they would not be included in a UML diagram as they are supposed to be hidden, but for this case, we revealed their implementation to visually illustrate our point.

implementation: ARRAY2[ETF_SQUARE]

is a private feature. Its information can never be accessed by its client. In addition, ETF_SQUARE's features can only be used by ETF_BOARD.

ships: ARRAYED_LIST[ETF_SHIP]

is private as well. Also, ETF_SHIP's features can only be used by ETF_BOARD.

This way, our contract with the client for this module can never be violated. Even if we make major changes to our implementations, the client will never know it and they do not need to be alerted to this change.

In addition, ETF_BOARD's invariant is important as it defines a substantial part of our design. This module is made specifically for our project specifications. Its contract shows that it is made exclusively for the Battleship game. The invariant specifies the precise range for the number of ships, number of shots, number of bombs, and the size of the board restricted to the requirements of the game. That means any game that is a part of this system will require its own set of module using the singleton design to allow command code to access them.

6. Summary of Testing Procedures

Test file	**Description**	**Passed**
at1.txt	Error case: Test scenario where wrong inputs are used for custom_setup and custom_setup_test	✓
at2.txt	Normal case: Test undo/redo for normal games and custom games	✓

7. Appendix (Contract view of all classes)

```
note
        description: "Business model of the Battleship game."
        author: "Jackie Wang"
        date: "$Date$"
        revision: "$Revision$"

class interface
        ETF_GAME

create {ETF_GAME_ACCESS}
        make

feature -- model attributes

        state_counter: INTEGER_32

        game_counter: INTEGER_32

        board: ETF_BOARD

        history: ETF_HISTORY [ETF_ACTIONS]

        gave_up: BOOLEAN

        is_custom: BOOLEAN

        last_board: ETF_BOARD

feature -- model operations

        default_update
                        -- Perform update to the model state.

        reset
                        -- Reset model state.

        reset_history

        new_game (level: INTEGER_64; is_debug_mode: BOOLEAN)
                        --create new board

        custom_game (dimension, ships, max_shots, num_bombs: INTEGER_32; is_debug_mode: BOOLEAN)
                        --create new board

        give_up

feature -- queries

        get_is_cusom: BOOLEAN

feature -- actions commands

        set_board (a_board: ETF_BOARD)
                        -- set current board to a_board

        set_last_board (a_board: ETF_BOARD)
                        -- set previous board to a_board

feature -- queries

        out: STRING_8
                        -- Returns string representation of game state

end
```

```
note
        description: "Board of various sizes that represent the state of the Battleship game."
        author: "JSO"
        date: "$Date$"
        revision: "$Revision$"

class interface
        ETF_BOARD

create
        make,
        make_empty

feature -- game info

        game_status: INTEGER_32
                        -- 0: Game is RUNNING
                        -- 1: Game is LOST
                        -- 2: Game is WON
                        -- 3: Game has not started
                        -- 4: Gave up

        count_sunk_ships: INTEGER_32
                        -- Returns the number of ships sunk

        get_max_score: INTEGER_32
                        -- Returns maximum score possible

        update_score (i: INTEGER_32)
                        -- update total_score of current board

        set_message (a_state, a_action: STRING_8)
                        -- set messages

        set_message_state (a_state: STRING_8)
                        -- set messages for states

        set_message_action (a_state: STRING_8)
                        -- set messages for actions

        is_hit (row, col: INTEGER_32): BOOLEAN
                        -- return True if coordinate hits a ship

        distance (coordinate1: TUPLE [row: INTEGER_64; column: INTEGER_64]; coordinate2: TUPLE [row:
INTEGER_64; column: INTEGER_64]): INTEGER_32
                        -- return distance between coordinate1 and coordinate2

        is_adjacent (coordinate1: TUPLE [row: INTEGER_64; column: INTEGER_64]; coordinate2: TUPLE
[row: INTEGER_64; column: INTEGER_64]): BOOLEAN
                        -- returns true if coordinate1 and coordinate2 are adjacent; false otherwise
                ensure
                        is_adjacent: distance (coordinate1, coordinate2) ~ 1 implies Result = True
                        is_not_adjacent: distance (coordinate1, coordinate2) /~ 1 implies Result =
False

        is_valid (coordinate: TUPLE [row: INTEGER_64; column: INTEGER_64]): BOOLEAN
                        -- Is this a valid position given borad size

        fire (row, col: INTEGER_32)

        bomb (coordinate1: TUPLE [row: INTEGER_64; column: INTEGER_64]; coordinate2: TUPLE [row:
INTEGER_64; column: INTEGER_64])

        set_state (i: INTEGER_32)

feature -- queries

        get_state_feedback: STRING_8
                        -- returns feedback of current state

        get_action_feedback: STRING_8
```

```
			-- returns feedback of current 'action'

	check_debug: BOOLEAN
			-- returns indication if board is using debugging mode

	get_shots: INTEGER_32
			-- returns number of shots used

	get_max_shots: INTEGER_32
			-- returns max number of shots available

	get_bombs: INTEGER_32
			-- returns number of bombs used

	get_max_bombs: INTEGER_32
			-- returns max number of bombs available

	get_score: INTEGER_32
			-- returns current score of all sunk ships

	get_total_score: INTEGER_32
			-- returns total score of all shots that were hit

	get_max_total_score: INTEGER_32
			-- returns max score

	ships_count: INTEGER_32
			-- returns number of ships

	get_state: INTEGER_32
			-- returns current state

feature -- out

	ships_out: STRING_8
			-- Returns string representation of ships

	out: STRING_8
			-- Return string representation of current game.
			-- You may reuse this routine.

invariant
	board_size_limits: implementation.count = board_size * board_size
	shots_limit: shots <= max_shots
	bombs_limit: bombs <= max_bombs

end
```

```
note
	description: "Summary description for {ETF_SHIP}."
	author: ""
	date: "$Date$"
	revision: "$Revision$"

class interface
	ETF_SHIP

create
	make,
	make_empty

feature -- constructor

	make_empty

	make (a_size, a_row, a_col: INTEGER_32; a_dir: BOOLEAN)

end
```

```
note
	description: "The game square position with char"
	author: "JSO"
	date: "$Date$"
	revision: "$Revision$"

class interface
	ETF_SQUARE

create
	make

invariant
	allowable_symbols: item = '_' or item = 'h' or item = 'v' or item = 'O' or item = 'X'

end
```

```
note
        description: "[
                Abstract move of Battleship.
        ]"
        author: "JSO"
        date: "$Date$"
        revision: "$Revision$"

deferred class interface
        ETF_ACTIONS

feature -- deferred commands

        execute

        undo

        redo

        get_state: TUPLE [oldp: INTEGER_32; newp: INTEGER_32]

        get_state_feedback: TUPLE [oldp: STRING_8; newp: STRING_8]

        get_state_action: TUPLE [oldp: STRING_8; newp: STRING_8]

end
```

```
note
        description: "Summary description for {ETF_ACTIONS_FIRE}."
        author: ""
        date: "$Date$"
        revision: "$Revision$"

class interface
        ETF_ACTIONS_FIRE

create
        make

feature -- Initialization

        make (new_board: ETF_BOARD)

feature -- commands

        execute

        undo

        redo

        get_state: TUPLE [oldp: INTEGER_32; newp: INTEGER_32]

        get_state_feedback: TUPLE [oldp: STRING_8; newp: STRING_8]

        get_state_action: TUPLE [oldp: STRING_8; newp: STRING_8]

end
```

```
note
	description: "Summary description for {ETF_ACTIONS_NEW_GAME}."
	author: ""
	date: "$Date$"
	revision: "$Revision$"

class interface
	ETF_ACTIONS_NEW_GAME

create
	make

feature -- Initialization

	make (new_board: ETF_BOARD)

feature -- commands

	execute

	undo

	redo

	get_state: TUPLE [oldp: INTEGER_32; newp: INTEGER_32]

	get_state_feedback: TUPLE [oldp: STRING_8; newp: STRING_8]

	get_state_action: TUPLE [oldp: STRING_8; newp: STRING_8]

end
```

```
note
        description: "History operations for undo/redo design pattern."
        author: "JSO"
        date: "$Date$"
        revision: "$Revision$"

class interface
        ETF_HISTORY [G -> ETF_ACTIONS]

create
        make

feature -- queries

        item: G
                        -- Cursor points to this user operation
                require
                                on_item

        on_item: BOOLEAN
                        -- cursor points to a valid operation
                        -- cursor is not before or after

        after: BOOLEAN
                        -- Is there no valid cursor position to the right of cursor?

        before: BOOLEAN
                        -- Is there no valid cursor position to the left of cursor?

feature -- comands

        extend_history (a_op: G)
                        -- remove all operations to the right of the current
                        -- cursor in history, then extend with a_op
                ensure
                                history [history.count] = a_op

        remove_right
                        --remove all elements
                        -- to the right of the current cursor in history

        forth
                require
                                not after

        back
                require
                                not before

        reset
                ensure
                        check_is_empty: history.is_empty

end
```

```
note
        description: "[
                The RANDOM_GENERATOR class is used to generate
                coordinates to place ships on the board. Each
                set represents a new ship and can be attained
                by calling forth.
        ]"
        author: "Joshua Phillip"
        date: "June 18th, 2018"
        revision: "1"

class interface
        RANDOM_GENERATOR

create
        make_debug,
        make_random

feature -- queries

        column: INTEGER_32
                        -- returns a random variable used to generate column coordinates

        row: INTEGER_32
                        -- returns a random variable used to generate row coordinates

        direction: INTEGER_32
                        -- returns a random variable used to generate direction

feature -- commands

        forth
                        -- sets the row, column and direction variables forward
                        -- should be called for a new ship or if there is a collision

end
```