EECS 3311 – Lab 4

Student #1

Name: Rajkumar Balakrishnan Lakshmi

Prism login: kumarraj

Student Number: 213141197

Signature: 

Student #2

Name: Philip D'Aloia

Prism login: pdaloia

Student Number: 213672431

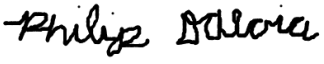
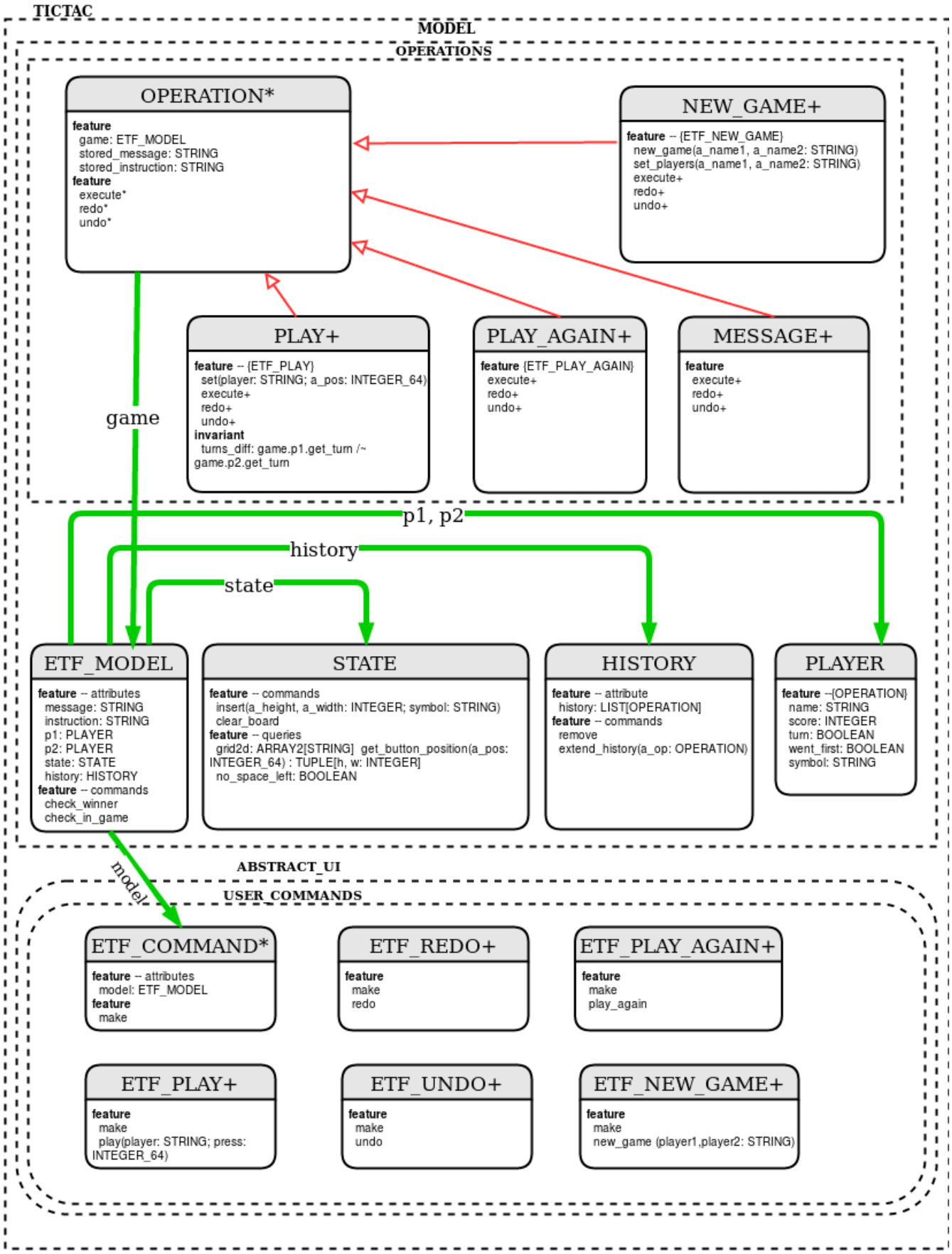
Signature: 

Table of Contents


TICTAC
MODEL
OPERATIONS
OPERATION*
feature
game: ETF_MODEL
stored_message: STRING
stored_instruction: STRING
feature
execute*
redo*
undo*
NEW_GAME+
feature -- {ETF_NEW_GAME}
new_game(a_name1, a_name2: STRING)
set_players(a_name1, a_name2: STRING)
execute+
redo+
undo+
PLAY+
feature -- {ETF_PLAY}
set(player: STRING; a_pos: INTEGER_64)
execute+
redo+
undo+
invariant
turns_diff: game.p1.get_turn /~ game.p2.get_turn
PLAY_AGAIN+
feature {ETF_PLAY_AGAIN}
execute+
redo+
undo+
MESSAGE+
feature
execute+
redo+
undo+
game
p1, p2
history
state
ETF_MODEL
feature -- attributes
message: STRING
instruction: STRING
p1: PLAYER
p2: PLAYER
state: STATE
history: HISTORY
feature -- commands
check_winner
check_in_game
STATE
feature -- commands
insert(a_height, a_width: INTEGER; symbol: STRING)
clear_board
feature -- queries
grid2d: ARRAY2[STRING] get_button_position(a_pos: INTEGER_64) : TUPLE[h, w: INTEGER]
no_space_left: BOOLEAN
HISTORY
feature -- attribute
history: LIST[OPERATION]
feature -- commands
remove
extend_history(a_op: OPERATION)
PLAYER
feature --{OPERATION}
name: STRING
score: INTEGER
turn: BOOLEAN
went_first: BOOLEAN
symbol: STRING
model
ABSTRACT_UI
USER_COMMANDS
ETF_COMMAND*
feature -- attributes
model: ETF_MODEL
feature
make
ETF_REDO+
feature
make
redo
ETF_PLAY_AGAIN+
feature
make
play_again
ETF_PLAY+
feature
make
play(player: STRING; press: INTEGER_64)
ETF_UNDO+
feature
make
undo
ETF_NEW_GAME+
feature
make
new_game (player1,player2: STRING)

The deferred OPERATION class is an abstract class. It has access to the current instance of the game (of type ETF_MODEL) as well as two attributes, stored_message and stored_instruction (noth of type STRING). It also has two undefined features called execute, redo, undo. The classes PLAY, PLAY_AGAIN, NEW_GAME, and MESSAGE all inherit from OPERATION and redefine the execute, undo, and redo features to do what is necessary for an object of its' type.

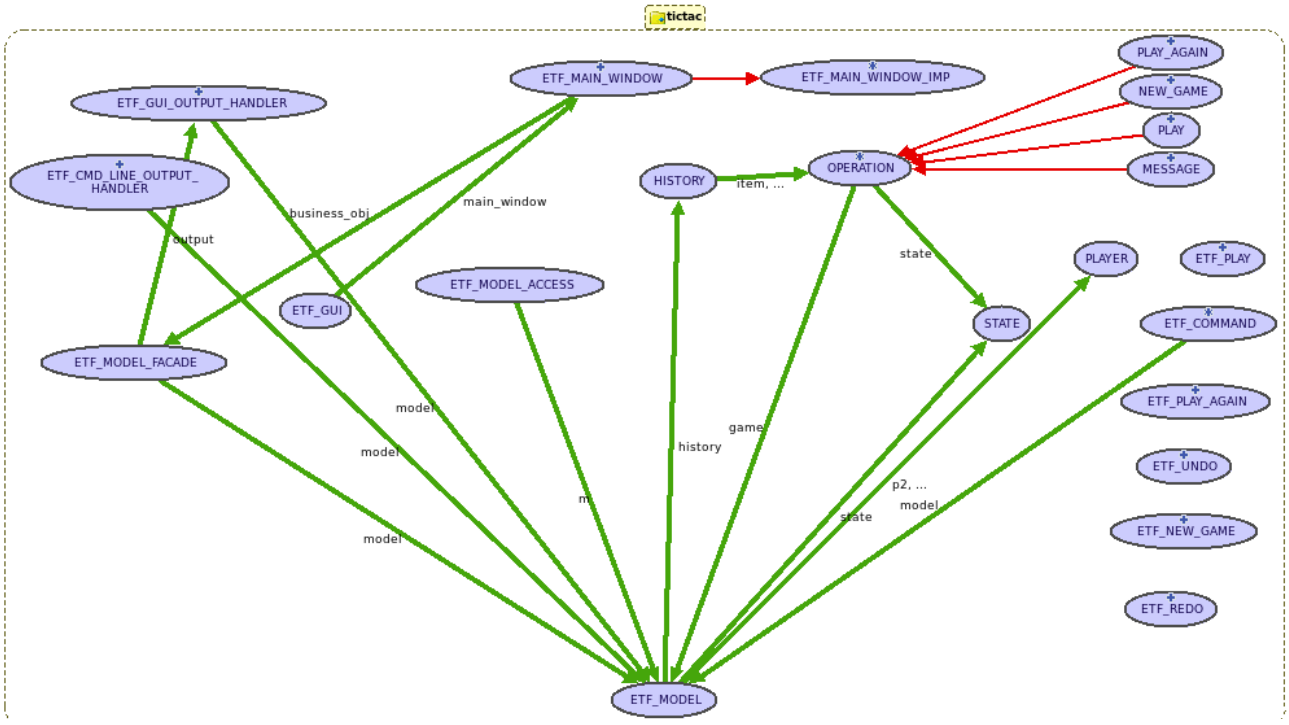
ETF_MODEL is the class which defines an object of the tictactoe game. It is the client in the client-supplier relation with the STATE, HISTORY, and PLAYER classes. It is the supplier in the client-supplier relationship with ETF_COMMAND and OPERATION.

ETF_MODEL takes one instance of STATE to define the tictactoe board, it takes one instance of HISTORY to store operations and messages, and takes two instances of PLAYER (one for each player in the tictactoe game).

OPERATION takes the current instance of ETF_MODEL as a model of the current game.

ETF_COMMAND also take the current instance of ETF_MODEL for the same reason as OPERATION.

BON diagram generate by Eiffel Studio IDE:



Module	Description	Information Hiding
ETF_MODEL	The ETF_MODEL class is a module used to define the titactoe game. It stores necessary information for the game such as the history of operations and messages for the undo/redo mechanism, the state which is the object used to store the information about the current game board, and one player object for each of the players in the game. It has two attributes, message and instruction, which are of type STRING. They are used to store and print the current message and instruction to the players of the game after every input. There are also another two attributes name is_winner and in_game. is_winner is a BOOLEAN attribute used to determine if there is a winner in the game. in_game is also a BOOLEAN attribute which is used to determine if the current game is still in progress.	- create exported to {ETF_MODEL_ACCESS} - make exported to {NONE}
HISTORY	The HISTORY module defines an object in which stores a LIST that holds objects of type OPERATION. It has features such as extend_history to add an OPERATION to the LIST, remove which removes all items in the LIST, remove_right which removes all items after the item that the cursor is pointing to.	- make exported to {STATE}
STATE	The STATE module defines an object for the tictactoe game board. It has an attribute called grid2d which is of type ARRAY2[STRING] which is used to store the positions of player 1's X selection and player 2's O selections as well as underscored used to indicate a free space on the board. Features are included such as insert to insert and "X" or "O" in an appropriate board position, no_space_left to determine if the board is filled with "X" and "O", and clear_board to reset the board (make every position an underscore).	- create exported to {ETF_MODEL} - make exported to {NONE}
PLAYER	The PLAYER module defines an object for a player in the tictactoegame. An object of type PLAYER has attribute name, score, symbol (for if the player is "X" or "O"), turn (if it is currently their turn), went_first (for if the player went first in the current game).	- create exported to {ETF_MODEL} - make exported to {NONE} - attributes exported to {OPERATION} - commands exported to {OPERATION, ETF_MODEL}

OPERATION	The module OPERATION is used to define a deferred class. This deferred class has attributes stored_message and stored_instruction. It also has undefined features execute, undo, and redo. OPERATION also has a feature to access the current ETF_MODEL of the game.	Not applicable
PLAY	The PLAY module inherits from OPERATION. It implements the features execute, redo, and undo to handle a successful play operation from a user and what must be done to redo and undo that operation. It also has a set feature which sets the play operation's symbol (depending on which player's turn it is) finds the position in the board which will be changed.	- make exported to {NONE} - attributes exported to {NONE} - commands exported to {ETF_PLAY}
PLAY_AGAIN	The PLAY_AGAIN module inherits from OPERATION. It implements the features execute, redo, and undo to handle a successful play_again operation from a user and what must be done to redo and undo that operation.	- make exported to {NONE} - attributes exported to {NONE} - commands exported to {ETF_PLAY_AGAIN}
NEW_GAME	The NEW_GAME module inherits from OPERATION. It implements the features execute, redo, and undo to handle a successful new_game operation from a user and what must be done to redo and undo that operation. It also has a feature called set_players which sets the player's names in a new game.	- make exported to {NONE} - attributes exported to {NONE} - commands exported to {ETF_NEW_GAME}
MESSAGE	The MESSAGE module inherits from OPERATION. This module is used to define a message if another operation is unsuccessful. It implements the features execute, redo, and undo to handle a successful message operation from a user and what must be done to redo and undo that operation.	- make exported to {NONE} - attributes exported to {NONE}
ETF_COMMAND	The ETF_MODEL attribute is a class that defines a singleton object which is used to access the current instance of ETF_MODEL (the current game). Other modules such as ETF_PLAY, ETF_PLAY_AGAIN, and ETF_NEW_GAME inherit from classes that inherit from ETF_COMMAND which define what must happen when the user inputs a certain operation and what must be done to handle an error or incorrect input. ETF_UNDO and ETF_REDO also indirectly inherit from ETF_COMMAND to define what must happen when an undo or redo input is entered.	- make exported to {NONE}

Undo / Redo Design

The undo / redo design that we used in this lab is very similar to the design pattern described in the book and illustrates the use of polymorphism, static typing and dynamic binding in object-oriented design. The design was achieved by having one deferred class OPERATION and the corresponding game operations NEW_GAME, PLAY, PLAY_AGAIN and MESSAGE inherit from the deferred class. The deferred class itself consists of three deferred features execute, undo and redo. The operation classes of tic-tac-toe game that needed to make use of undo and redo functionality simply inherited from OPERATION and had the deferred features effective. The deferred class also has two attributes stored_message and stored_instruction in order to store the current message and instruction of the game.

The current state of the game at every user input is stored to a Linked list of the type OPERATION in a separate class called HISTORY. The class HISTORY has the feature extend_history to add an OPERATION to the LIST and remove feature to remove all items from the LIST. In routine {ETF_NEW_GAME} new_game, if the input from the user is valid , then the board of the game is cleared and set to default state. The items stored in LIST of class HISTORY is removed and the execute feature of class NEW_GAME is executed . In routine {ETF_PLAY} play, if the input from the user is valid , then symbol of input player is inserted at the appropriate position by calling the execute feature of class PLAY. The execute feature of PLAY module also checks if there is a winner at current state of the game, if not checks for a tie in the game and sets the message and instruction appropriately. All these operations of the current instance is stored to the HISTORY's LIST as an item of type OPERATION. In routine {ETF_PLAY_AGAIN} play_again, if the input from the user is valid (i.e., if there is a tie in the game or there is a winner and if the players wish to play again) , then the board of the game is cleared and set to default state and the items in LIST of class HISTORY is also removed. Thus every change made to the game at a given instance is stored in the list and can be redone or

undone easily by accessing OPERATION items from the list. A code snippet that uses polymorphism and dynamic binding is shown below :

```
play_again
    local
        play_again_op: PLAY_AGAIN
        message_op: MESSAGE
    do
        if not model.in_game then
            create message_op.make (model.message)
            model.history.extend_history (message_op)
            message_op.execute

        elseif not model.is_winner and not model.state.no_space_left then
            create message_op.make (model.err_game_not_finished)
            model.history.extend_history (message_op)
            message_op.execute
        else
            -- perform some update on the model state
            model.default_update
            model.set_is_winner (False)
            model.state.clear_board
            create play_again_op.make

            play_again_op.execute
            model.history.remove

        end
        etf_cmd_container.on_change.notify ([Current])
    end
```

In routine {ETF_UNDO} undo, we first check if the cursor is on an item, this to make sure that the history is not empty and that cursor is not pointing to before. If the cursor points to an item, then the undo feature of that item(OPERATION) is called and executed accordingly by the use of dynamic binding. After successful completion of undo procedure the cursor is moved to the previous position. Similarly for the routine {ETF_REDO} redo, we first check if the cursor is on an item, this to make sure that the history is not empty. If the cursor points to an item, then the redo feature of that item(OPERATION) is called and executed accordingly by the use of dynamic binding. The redo feature of every operation simply carries out the execute feature of that operation by making a call to it. After successful completion of redo procedure the cursor is moved to the next position. A code snippet that shows the implementation of the undo using dynamic binding is shown below :

```
undo
    local
        explain: BOOLEAN
        message_op: MESSAGE
    do
        -- perform some update on the model state
        model.default_update

        if model.history.on_item then
            model.history.item.undo
            model.history.back
        end

        etf_cmd_container.on_change.notify ([Current])
    end
```

Detecting a Winning Game

In the class ETF_MODEL, we include an attribute names is_winner which is a BOOLEAN attribute and a feature (in specific, a command) named check_winner. The check_winner command is used to manipulate whether is_winner is true or false. Whenever is_winner is true, it means there is a winner in the current game, and when it is false it means there is no winner currently. The way check_winner determines whether is_winner should be changed to the value true or false when the feature is called is through checking all possible winning scenarios on the current tictactoe board. This is done by checking each of the rows, columns, and diagonals, and then seeing if all of the values in the row/column/diagonal match. If all three match, it also checks to make sure one value is not an underscore (a blank space in the board). If this is satisfied in any of the rows, columns or diagnoals, is_winner is set to be true and the game is over. If it is not satisfied, is_winner is set to false and the game progresses normally. This describes how the feature check_winner manipulates the signal variable is_winner to show whether there is a winner or not, but there is still design decision on when check_winner should be called and where it must be called.

The method of detecting whether there is a winner in the current game of tictactoe is called only after a valid play operation is carried out. This means a winner is checked if the player whose turn it is plays in a vacant spot on the current tictactoe board. When this occurs, the ETF_PLAY module creates an operation of type PLAY which executes the turn carried out by the player in turn, and in the execute feature is where it checks to see if there is a winner through a call to the check_winner feature in the ETF_MODULE module. After checking through all winning scenarios to see if the new PLAY operation has created a winner, it will change the is_winner attribute (which is of type BOOLEAN) to the appropriate value of True or False.

If there is already a winner in the current tictactoe game when a valid play operation is input by the user, it will be detected by ETF_PLAY. Once an operation of type PLAY has been executed which results in either of the two players winning the game, nothing else can be done to change the state of

the game, but the game will still let the user enter inputs (which will not be executed). This results in a necessary check to see if there is a winner when ETF_PLAY is called. ETF_PLAY calls the feature check_winner in ETF_MODEL, and if there is a winner, it creates a new operation of type MESSAGE in which is to let the user know that the game is finished due to a player already winning the tictactoe game.