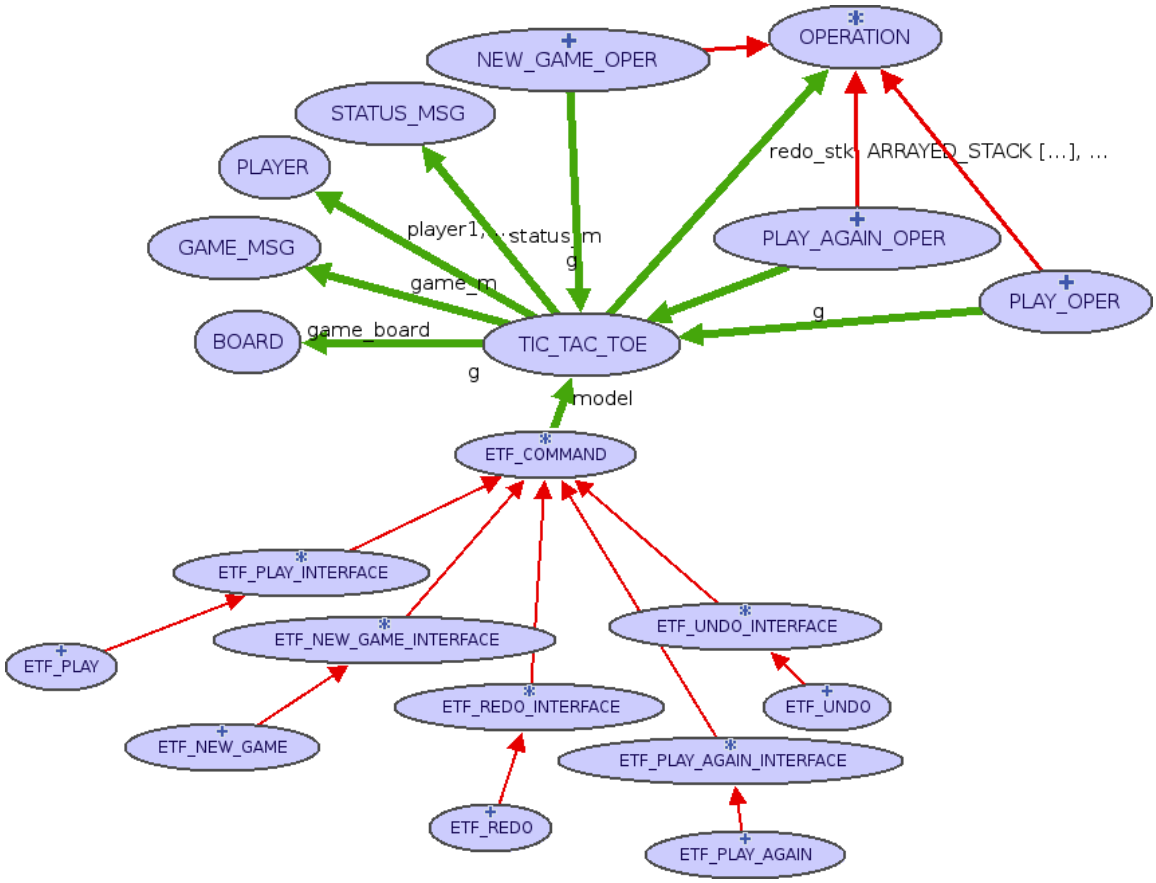
# EECS 3311 Lab4 Report: Tic Tac Toe

Amanda D'Errico (213007554, aderrico)

# Table of Contents

# Top Level view




The program is being manipulated through the ETF Command and ETF Redo/Undo classes which all call the model class (Tic_Tac_Toe). Tic_Tac_Toe has a singleton instance so the game is only created once for users to play. Tic_Tac_Toe is the class used as the supplier to all the classes that use a game instance. In the bon diagram above, the Operation class doesn't use a game instance, but its effective classes do. Effective classes New_Game_Oper, Play_Oper and Play_Again_Oper use a game instance to manipulate the state of the game by updating the messages stored in strings. When the Tic_Tac_Toe class runs, the new string messages are updated in the out function. Operation is used for the undo and redo implementation in Tic_Tac_Toe to preserve polymorphism. Additional classes include the Board, Player, Game_Msg and Status_Msg classes which are explained in the following page.

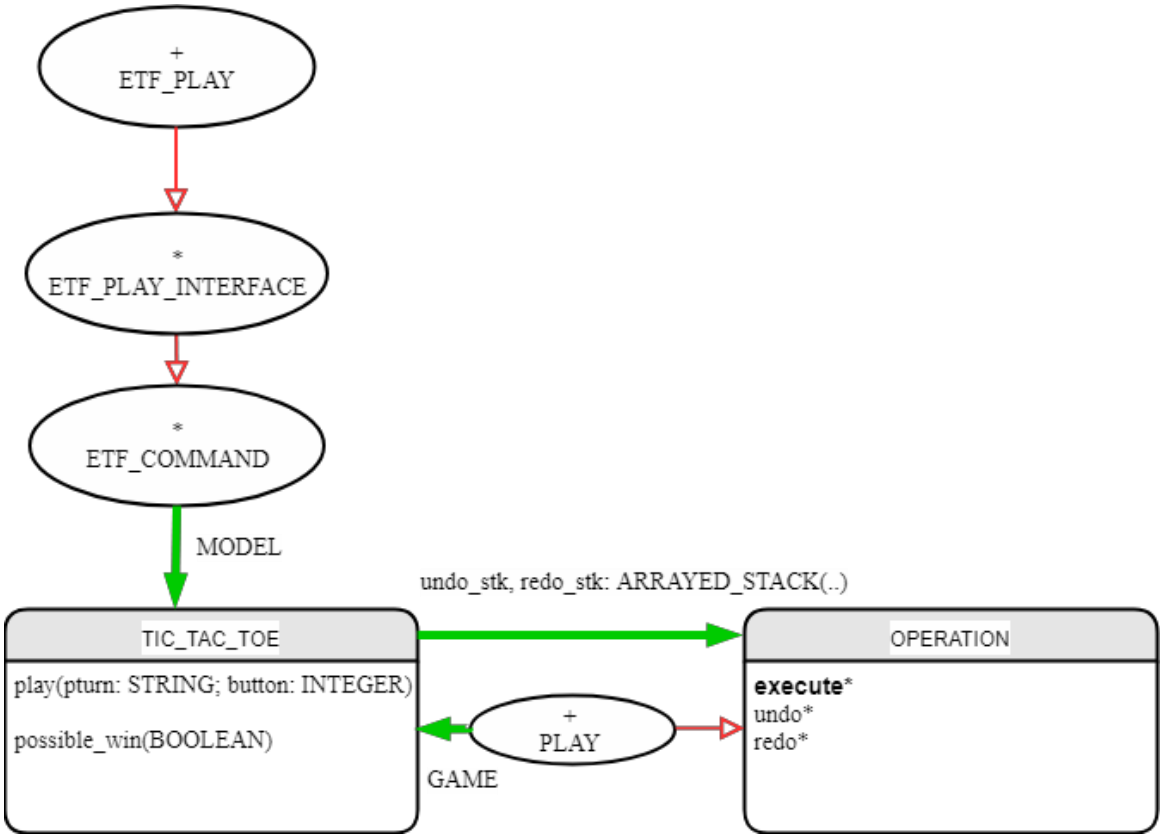
# Significant modules

| Class | Description | Design |
|---|---|---|
| Board | The game board for tic_tac_toe. It initializes the values to "_" and as operations are called, places either "X" or "O" in its place instead. | Allows only a square board. Varies in size in future designs. Also contains functions used in Play_Oper (almost_cap) and update_board (Tic_Tac_Toe's redo function.) |
| Operation | It is for undo/redo design. Parent of other operation classes New_Game_Oper, Play_Oper and Play_again_Oper. | Old status, player and game messages must be redefined in its effective classes so when calling the undo and redo function, the game and status string messages reset to the messages associated with that action. |
| Play_again_oper | It is for playing the game again, and makes the next player be whoever did not play first for the last game played. | Refill board with "_" and reset game_over and game_start booleans to false and true, respectively. This is done only if a winner or tie is detected. |
| New_game_oper | Starts a new game, initializes to default status messages and game | Set the player's turn and first player, fill the board with "_", initialize each player's score to 0, and |

| | | |
|---|---|---|
| | message describing which player plays next. | reset game_over and game_start booleans to true and false, respectively. |
| Play_oper | Player plays a move. It will change messages depending if it performs normally or there is an error message in the ETF corresponding class. | The status and game messages are determined on whether the last move played results in a win, a tie, or the next move played. |
| Status_msg | This class stores all game status messages. Some examples include default message “ok”, “not this player’s turn”, “button already taken”, etc. | Initializes various status messages. If messages are not default they are checked in the ETF classes for operations and redo undo. |
| Game_msg | This class stores all game messages. Examples include “start new game”, “play again or start new game” and “<player> plays next”. | Initializes game messages. Function player_next finds the player’s name to determine who’s turn it is next. |
| Player | Information about the player. Initialized name to an empty string to define later in tictactoe. Used single responsibility principle to encapsulate the class and ensure a player’s main responsibility is providing its name to tictactoe. Tictactoe is | Storing information of players, defining its name, updating the name, providing getters for other classes to use. |

| | | |
|---|---|---|
| | responsible for the player's score and turn. | |
| Tic_tac_toe | The class that supplies to every class implemented. Board, player1 and player2, messages, stacks for redo/undo, new_game_oper, play_oper and play_again_oper are all included in tictactoe. | Store main information such as players, board, messages associated with status and game messages, player scores, etc. Undo and redo are implemented in tictactoe for any operation. |

# Detecting a Winning Game




Below is the execute function in Play_Oper that detects a winning game:

```
execute
    do
        -- checks if the button the player plays will guarantee a win
        if g.possible_win (g.get_pt_letter, g.button_played) then
            -- place the letter in winning spot
            g.game_board.board.enter(g.get_pt_letter, g.button_played)
            -- set the status and game messages
            g.set_sts_str (g.status_m.win)
            g.set_msg_str (g.game_m.new_or_again)
            -- set new board string
            g.set_board_str (g.game_board.out)
            -- increase that player's score
            g.increase_pscore (g.player_turn)
            -- game is now over and now new game is started
            g.set_game_over (true)
            g.set_game_start (false)
        -- after checking if there's no possible win state, and the board is full, must be a tie
        elseif g.game_board.almost_cap then
            -- place the letter in tie spot
            g.game_board.board.enter(g.get_pt_letter, g.button_played)
            -- set the status and game messages
            g.set_sts_str (g.status_m.tie)
            g.set_msg_str (g.game_m.new_or_again)
            -- set new board string
            g.set_board_str (g.game_board.out)
            -- swap the players for the next time a game is played
            g.swap_players
            -- game is now over and now new game is started
            g.set_game_over (true)
            g.set_game_start (false)
        -- the board is not full, and you can place your letter in the appropriate spot of your choice
        else
            -- place letter in empty spot
            g.game_board.board.enter(g.get_pt_letter, g.button_played)
            -- swap players and set the next player to go next
            g.swap_players
            g.set_sts_str (g.status_m.default_message)
            g.set_msg_str (g.game_m.player_next (g.player_turn))
            -- set new board string
            g.set_board_str (g.game_board.out)
        end
        -- reset the old_status, game message and button associated with state so we can access them later in tictactoe
        set_old_sts(g.sts_str)
        set_old_gmsg(g.msg_str)
        set_old_p1_msg(g.player_out (g.player1))
        set_old_p2_msg(g.player_out (g.player2))
        set_old_board(g.board_str)
```

A query possible_win in Tic_Tac_Toe checks all possible scenarios to guarantee a win for player1 or player2 before placing the player's turn's letter in the appropriate space. If none of the possible winning scenarios are met, then the player does not win the game and the win status message and corresponding game message are not updated. Otherwise, the player places its corresponding letter in the spot on the board and status and game messages are updated. If the operation doesn't generate a winning game, then you must call almost_cap from the Board class to check if the board has one space left for the player to place their letter. Since it's already guaranteed that the player's who is currently

playing did not win, they must have tied instead, and the status and game messages are updated. Otherwise, the player places their corresponding letter on the board and update their game and status messages to the default status message and game message that represents who plays next.

```
class
    ETF_PLAY
inherit
    ETF_PLAY_INTERFACE
        redefine play end
create
    make
feature -- command
    play(player: STRING ; press: INTEGER_64)
        require else
            play_precond(player, press)
        local
            p: PLAY_OPER
        do
        -- here we do all error cases -> defensive programming
            if model.is_game_finished then
                model.set_sts_str (model.status_m.game_finished)
            elseif not model.is_pexists (player) then
                model.set_sts_str (model.status_m.no_player)
            elseif not model.is_pturn (player) then
                model.set_sts_str (model.status_m.not_turn)
            elseif model.is_btaken (press.as_integer_32) then
                model.set_sts_str (model.status_m.taken)
            else
                model.play (player, press.as_integer_32)
            end

            create p.make_play(model, press.as_integer_32, model.sts_str, model.msg_str)

            if not model.redo_stk.is_empty then
                if model.undo_stk.is_empty then -- indicates that new_game is in redo
                    model.undo_stk.put (model.redo_stk.item)
                    model.redo_stk.remove
                else
                    model.clear_redo
                end
            end

            model.set_undo (p)

            etf_cmd_container.on_change.notify ([Current])
        end

end
```

In the ETF_PLAY class all the error cases are checked in priority of the oracle before a move is played, in order to set error messages instead of playing a move. The error cases are checked through Boolean functions located in the model class (Tic_Tac_Toe). This defensive programming logic is also used for ETF_NEW_GAME and ETF_PLAY_AGAIN for classes New_Game_Oper and Play_Again_oper, respectively.

# Undo/Redo Design

The undo/redo design was implemented through two stacks – one representing the undo stack and one representing the redo stack. In each operation, its respective status message and game message must be updated (named old_status and old_msg, respectively). Each time an operation is performed, it is pushed onto the undo stack with its corresponding messages. Each time undo is called, the top item is removed from the undo stack and pushed onto the redo stack. The game returns the board associated with the previous state. In Play_Oper, this is before the player performed a move. This is represented by placing "_" where the old button was played. Otherwise, in New_Game_Oper, you can just re- execute New_Game_Oper by initializing its variables again. If an Operation is called after an undo command, then the redo stack is cleared. If the Operation is new_game or play_again then the undo stack also becomes cleared. Players are swapped if the old_status is the default message. When a game is over, the undo stack's top's old_status remains the value.

For redo, if the redo stack isn't empty then push back onto the undo stack. Redo is re-executed if the Operation is New_Game_Oper or Play_Again_Oper. Otherwise, the game board is updated to the board within the top item in the redo stack. The staus and game messages get set as the messages associated with that state and the board string gets updated. Players are swapped if the old_status is the default message. If the redo stack is empty, the code within Tic_Tac_Toe doesn't get executed, and the messages remain the same as the previous state.