Tic Tac Toe

Student Name: Tao Wang
Student Number: 214672547
Prism Login: kevin89
Signature: Tao Wang

Contents

1. Requirements for Tic-Tac-Toe

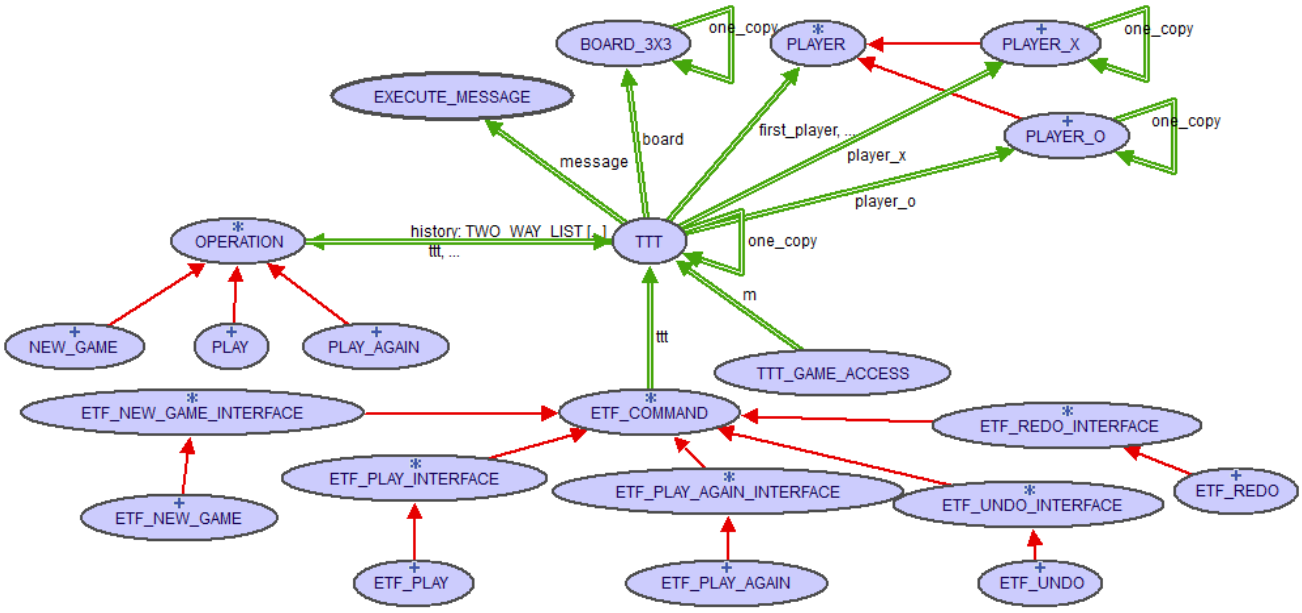
Based on the description for Tic-Tac-Toe on Wilipedia, Tic-tac-toe (also known as noughts and crosses or Xs and Os) is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. The following example game is won by the first player, X: Game of Tic-tac-toe, won by X

This project provides three basic commands, “new_game”, “play” and “play_again”, for two players to play games. On top of them, the commands “undo” and “redo” are added into this game to let the game more flexible and beginner-friendly.

The grammar of the user interface for the commands above is as follows:

```
type NAME = STRING
type BUTTON = 1..9

new_game(player1: NAME; player2: NAME)
  -- add players `player1' and `player2'
  -- `player1' starts X

play(player: NAME; press: BUTTON)

play_again

undo
  -- last action while in play
  --otherwise no effect

redo
  -- last action while in play
  -- otherwise no effect
```

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



The key module is TTT one instance of which represents one Tic-Tac-Toe(TTT) game.

The user interface gets a TTT object (singleton) by using TTT_GAME_ACCESS

3. Table of modules — responsibilities and information hiding

1	TTT	**Responsibility**: A representation of a Tic-Tac-Toe game which stores the game state	**Alternative**: none
	Concrete	**Secret**: The game state includes following information: 3x3 game board, player x, player o, game state message, game history, next turn, first player, game over.	

1.1	BOARD_3x3	**Responsibility**: A Tic-Tac-Toe game board	**Alternative**: none
	Concrete	**Secret**: implemented via an array of size 9	

1.2	EXECUTE_MESSAGE	**Responsibility**: A message after execution of one game operation. It is either a success message or an error message.	**Alternative**: none
	Concrete	**Secret**: Just a wrapper class for a message string	

1.3	PLAYER	**Responsibility**: A Tic-Tac-Toe player	**Alternative**: none
	Abstract	**Secret**: none	

1.3.1	PLAYER_X	**Responsibility**: A Tic-Tac-Toe player who plays “X”	**Alternative**: none
	Abstract	**Secret**: none	

1.3.2	PLAYER_O	**Responsibility**: A Tic-Tac-Toe player who plays “O”	**Alternative**: none
	Abstract	**Secret**: none	

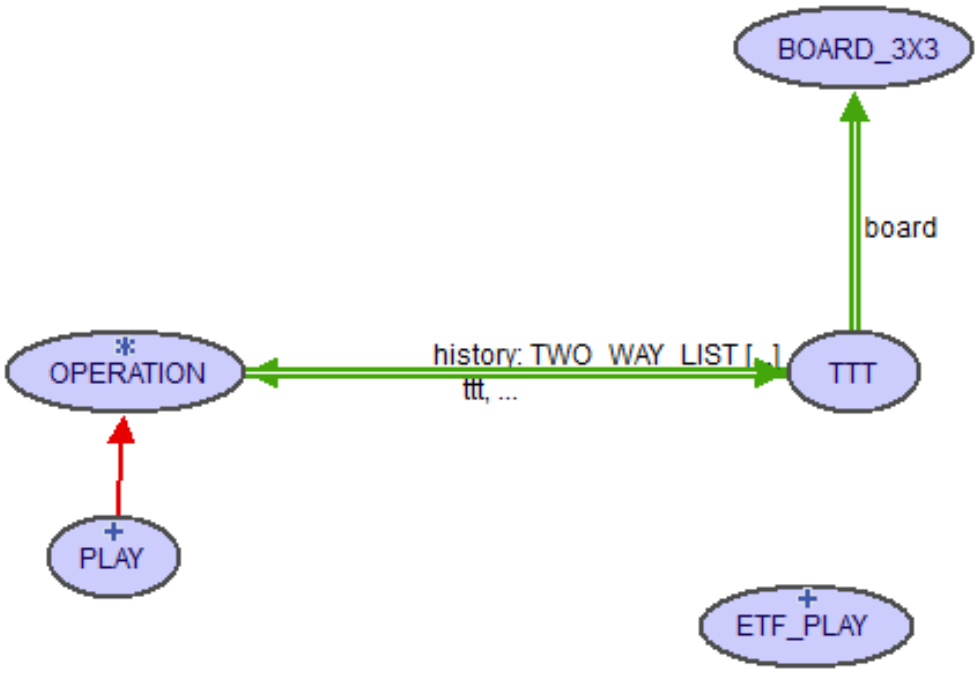
2	OPERATION	**Responsibility**: An operation on a Tic-Tac-Toe game	**Alternative**: none
	Abstract	**Secret**: none	

2.1	NEW_GAME	**Responsibility**: An operation that starts a new tic-tac-toe game.	**Alternative**: none
	Concrete	**Secret**: none	

2.2	PLAY	**Responsibility**: An operation that places “X” or “O” on a specified spot	**Alternative**: none
	Concrete	**Secret**: none	

2.2	PLAY_AGAIN	**Responsibility**: An operation that resets the game	**Alternative**: none

4. How to detect a winning game



In order to detect a winning game, main classed involved are TTT, PLAY, BROAD_3x3

When playing using the command line user interface, the player directly communicates with one instance of the class ETF_PLAY. ETF_PLAY internally interacts with one singleton instance of the class TTT, and it invokes the command “play” of TTT

```
note
	description: "TTT represents one tic-tac-toe game"
	author: ""
	date: "$Date$"
	revision: "$Revision$"

class interface
	TTT

feature --Commands

	play (player_name: STRING_8; position: INTEGER_32)
```

Since undo/redo design pattern is used in this project, inside the command above, it creates an object of the operation PLAY, and execute it, i.e. invoking the command execute of PLAY object.

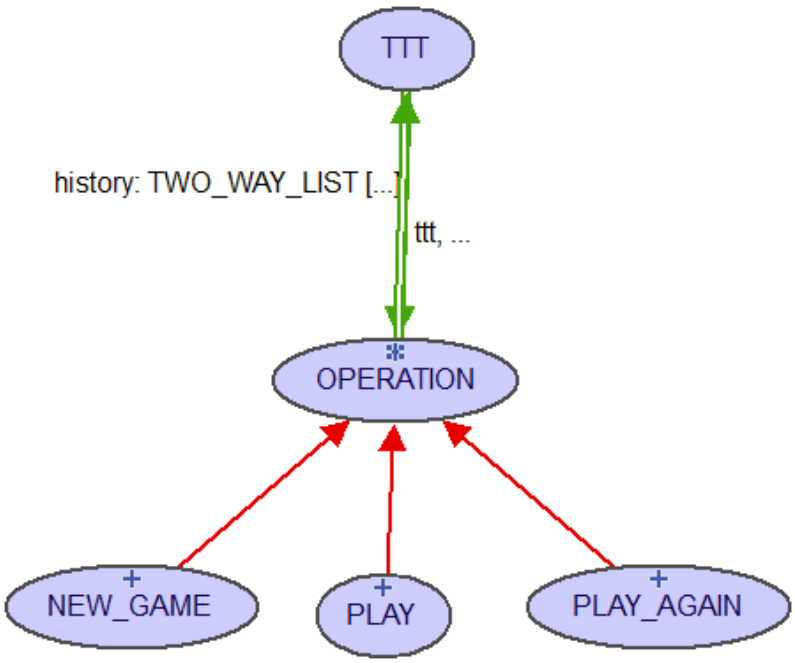
```
note
	description: "The operation PLAY"
	author: ""
```

```
        date: "$Date$"
        revision: "$Revision$"

class interface
        PLAY

feature --Commands

        execute

invariant
        position_within_bounds: position >= 1 and position <= 9
```

Inside the command execute, if passed player name and position are valid, then after put “X” or “O” onto the specified position, it checks the state of the current game board, if it’s “player x wins”, or “player o wins”, “draw” or “in progress”. For determining the state of the board, the query check_win in the class BOARD_3x3 is used.

```
note
        description: "A 3x3 game board for Tic-Tac-Toe"
        author: ""
        date: "$Date$"
        revision: "$Revision$"

class interface
        BOARD_3X3

feature --Queries

        check_win: INTEGER_32
                        --      returns a constant associated with the state of the game
                        --              If PLAYERX wins, returns 1.
                        --              If PLAYERO wins, returns 2.
                        --              If game is drawn, returns 3.
                        --              If game is in progress, returns 4.
```

5. Undo/redo design



OPERATION is a deferred class that has three commands, “execute”, “undo” and “redo”.

Since it is a small game, I decide to store all the game state information, such that the implementations of “undo” and “redo” are the same for all descendant concrete classes. To avoid code being “smelly”, “undo” and “redo” are implemented in OPERATION, and “execute” is deferred. “undo” simply restores the old state to the current game, and “redo” simply calls the command “execute” to re-execute. All descendent classes of OPERATION implement their own “execute” based on their roles on the game.

```
note
        description: "A deferred operation on the game Tic-Tac-Toe"
        author: ""
        date: "$Date$"
        revision: "$Revision$"

deferred class interface
        OPERATION

feature --Attrinutes

        old_ttt: detachable TTT

        has_error: BOOLEAN

        ttt: TTT
                        -- points to the singleton TTT object(model)

feature

        execute
```

```
	undo
			-- Revert to the state before the execution of an operation
		require
			has_been_executed: old_ttt /= Void

	redo
			-- by default "redo" just repeats execution

end
```

To implement multi-level undo/redo, TTT must have an attribute, “history”, to store executed operations. TWO_WAY_LIST[OPERATION] is used to implement “history” with which I can move the cursor back and forth to go to different game states. The cursor is manipulated inside two commands of TTT, “undo” and “redo”, and new operations are appended to the end of the history when “new_game”, “play” and “play_again” are called.

```
note
	description: "TTT represents one tic-tac-toe game"
	author: ""
	date: "$Date$"
	revision: "$Revision$"

class interface
	TTT
feature --Commands

	new_game (playerx: STRING_8; playero: STRING_8)

	play (player_name: STRING_8; position: INTEGER_32)

	play_again

	undo

	redo
```