EECS3311-Lab5-TicTacToe

Name: Mikhail Gindin
Signature: Mikhail Gindin
Prism: mikegin

Contents:

Top Level View – pg. 2

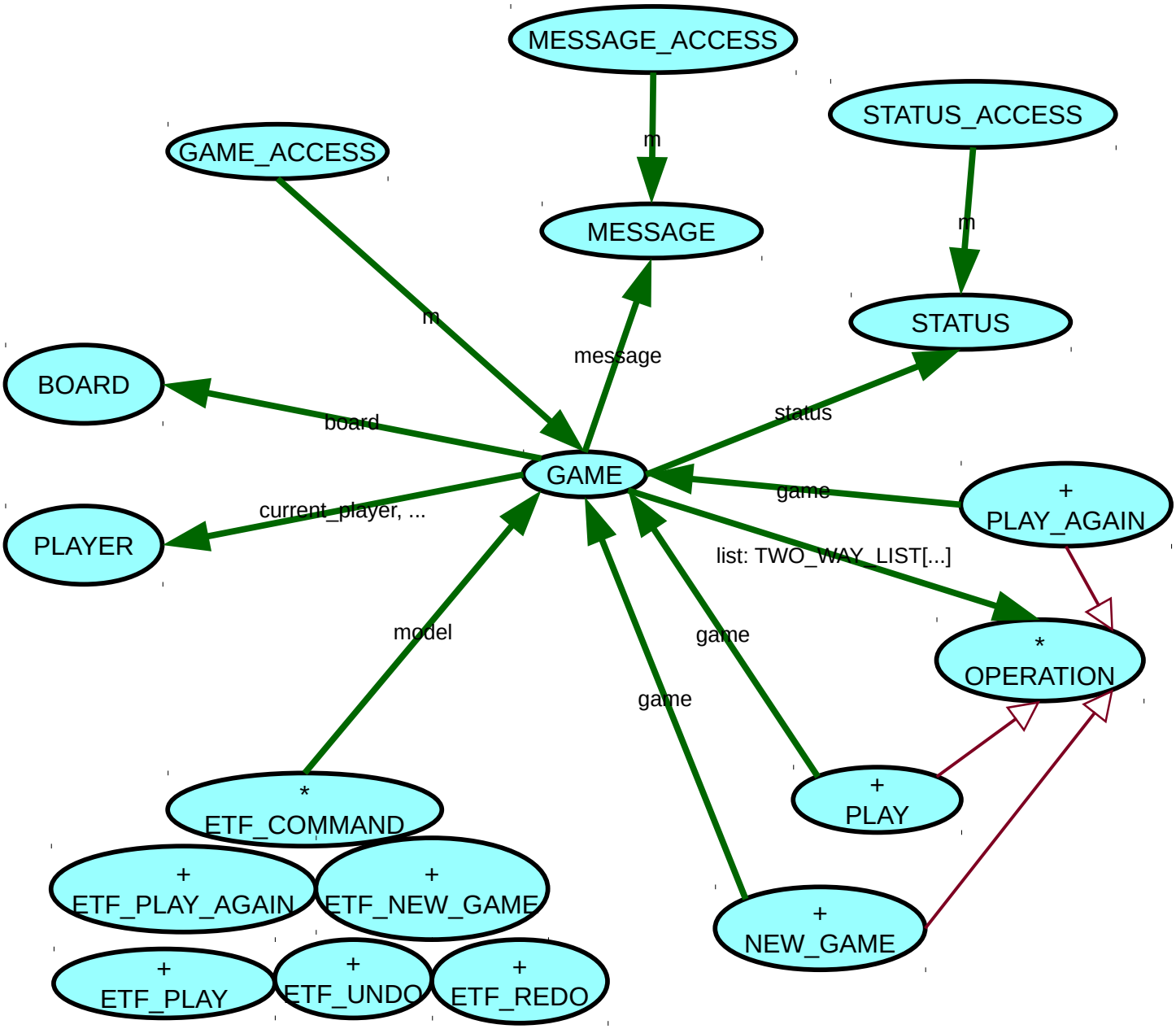
Significant Modules – pg. 3

How to detect a winning game – pg. 5

Undo/Redo design – pg. 6

Top Level View

The UI, in this case the ETF_COMMAND classes call the GAME class. GAME has corresponding methods for each command type. GAME stores the game board (from the BOARD class), an operations (“history”) list, and the players. After a UI call, GAME then creates OPERATION classes, and in case of PLAY, stores them in the history list. The OPERATION classes do validity checking and set the status and message accordingly. If the UI does an undo or redo call, then GAME just uses the correct operation in the history list and calls the undo or redo of the OPERATION object.



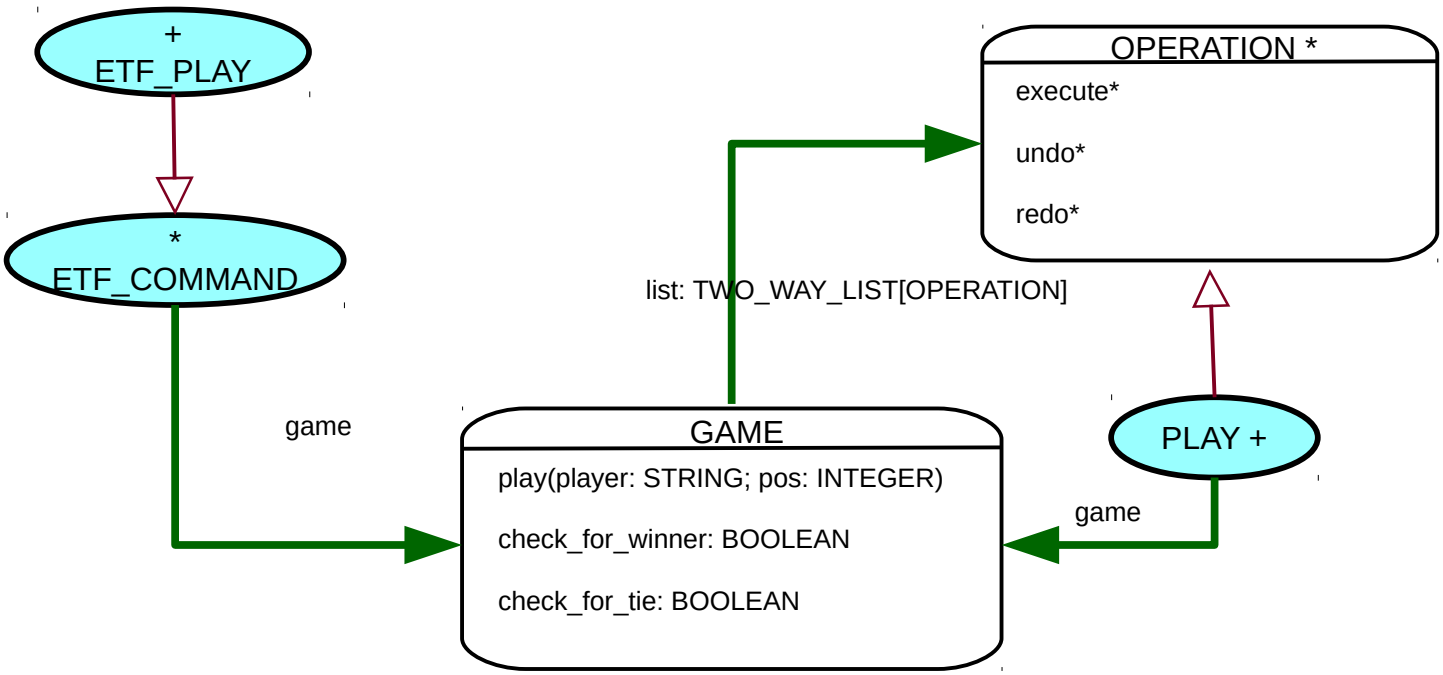
Significant Modules

Class	Description	Design Decision
BOARD	Class that represents a generic game board, has commands and queries to set and get characters on the board, clear the board, check if the board is full, and more.	This class was created to specifically deal with the game board. It was designed to be flexible; allow for any size and not be specific to a TicTacToe board.
PLAYER	Class that represents a player. Has attributes like the player's name, score, and game character (X/O).	This class was created to house attributes associated with a player.
STATUS	Class that houses various game statuses.	This class was created to house statuses. Statuses could change separately from messages.
MESSAGE	Class that houses various game messages.	This class was created to house messages. Messages could change separately from statuses.
GAME	Class that represents the TicTacToe game. The UI calls this class. Here the messages/status, operations history, game board, and players are stored. This class is responsible for creating the OPERATION classes, keeping track of the operations history, and creating the output. Also some game logic is done (i.e checking the board for a winner, converting button number to board coordinates).	This class was created to store the main information of the game. The players, the history of operations, and the output. The UI calls this class, and then this class either creates the operation (if its a play, play_again, or new_game) or uses an already created operation (if its an undo or redo) from its history list.
OPERATION	Deferred class that represents a user operation. Can be executed, undone, or redone.	Created since there are many type of operations all which do similar commands.
PLAY	Class that represents a play operation. Does validity checking, state changes (i.e switching turns, increasing scores), and message setting accordingly.	Created to handle play command logic.
PLAY_AGAIN	Class that represents a play_again operation. Cannot be undone or redone. Does validity	Created to handle play_agian logic.

	checking, state changes (i.e clearing the game board/history), and message setting accordingly.	
NEW_GAME	Class that represents a new_game operation. Cannot be undone or redone. Does validity checking, state changes (i.e setting the players), and message setting accordingly.	Created to handle new_game logic.

How to detect a winning game

After every valid play command, a check is done to see if there is a winner. So first the UI calls the GAME class. The GAME class creates a PLAY object and calls it's execute command. The PLAY object first checks if the command is valid. If it is, the PLAY object calls the check_for_winner query in GAME, which does the required logic on the board to see if a winner exists. If it does, the game is set to be finished, the player's score is incremented, and the required messages are put in place.



Undo/Redo design

First it is key to understand the following:
-Each PLAY object has an undo and a redo command
-A PLAY object's undo command is just the opposite of its execute (it resets the character it placed to the default character).
-A PLAY object's redo command just calls the PLAY object's execute command.
-Each PLAY object also stores the current status and message upon creation, before it's execution and status changing is done.

The GAME class stores PLAY objects in a "history" list, since play operations are the only ones that can be undone/redone. The lists cursor follows the "current" PLAY object. So when the user does an undo command, the PLAY object that the history list's cursor is pointing at does it's undo command. The undo command, upon also doing the proper undo execution, restores the game's status to the status that the PLAY object had saved upon it's creation. After the undo command, the history cursor moves back (if possible) to the previous play operation.

For a redo command, if possible, the history list cursor moves right and executes that PLAY object's redo command, which just calls the PLAY object's execute command.

When a PLAY object is created, before appending the PLAY object to the end of the history list, all PLAY objects to the right of the cursor (if there are any) are removed from the list. This is because if after some undo commands, a play operation is executed, the user cannot then redo the play operation that was to the right of the cursor, since it would not make sense.