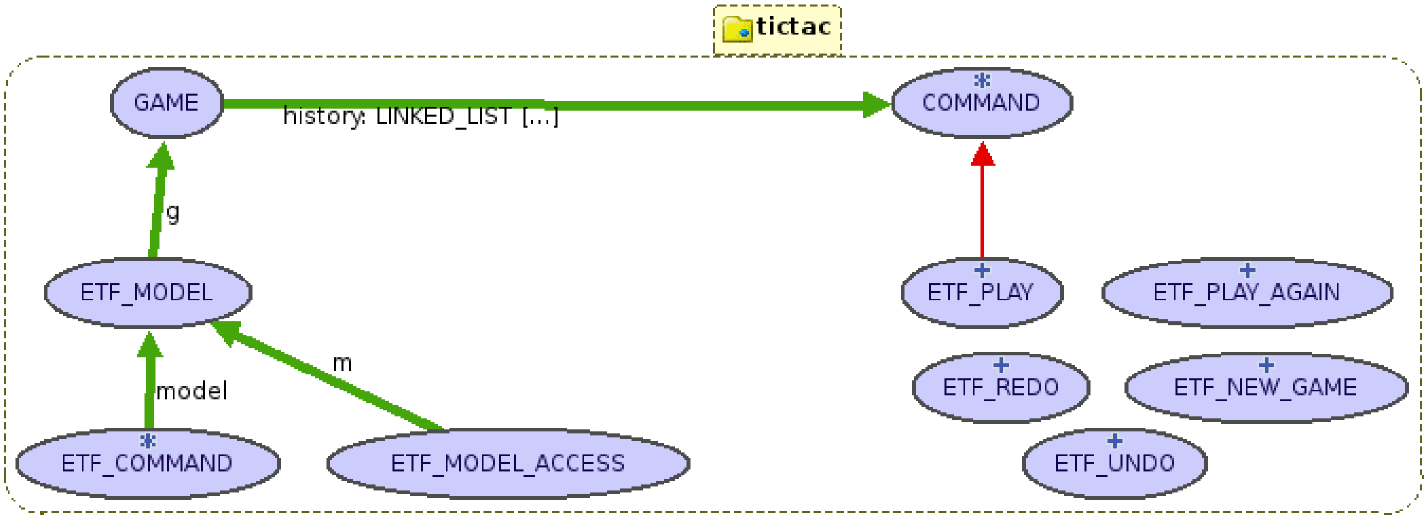
EECS3311 Lab 5 Report

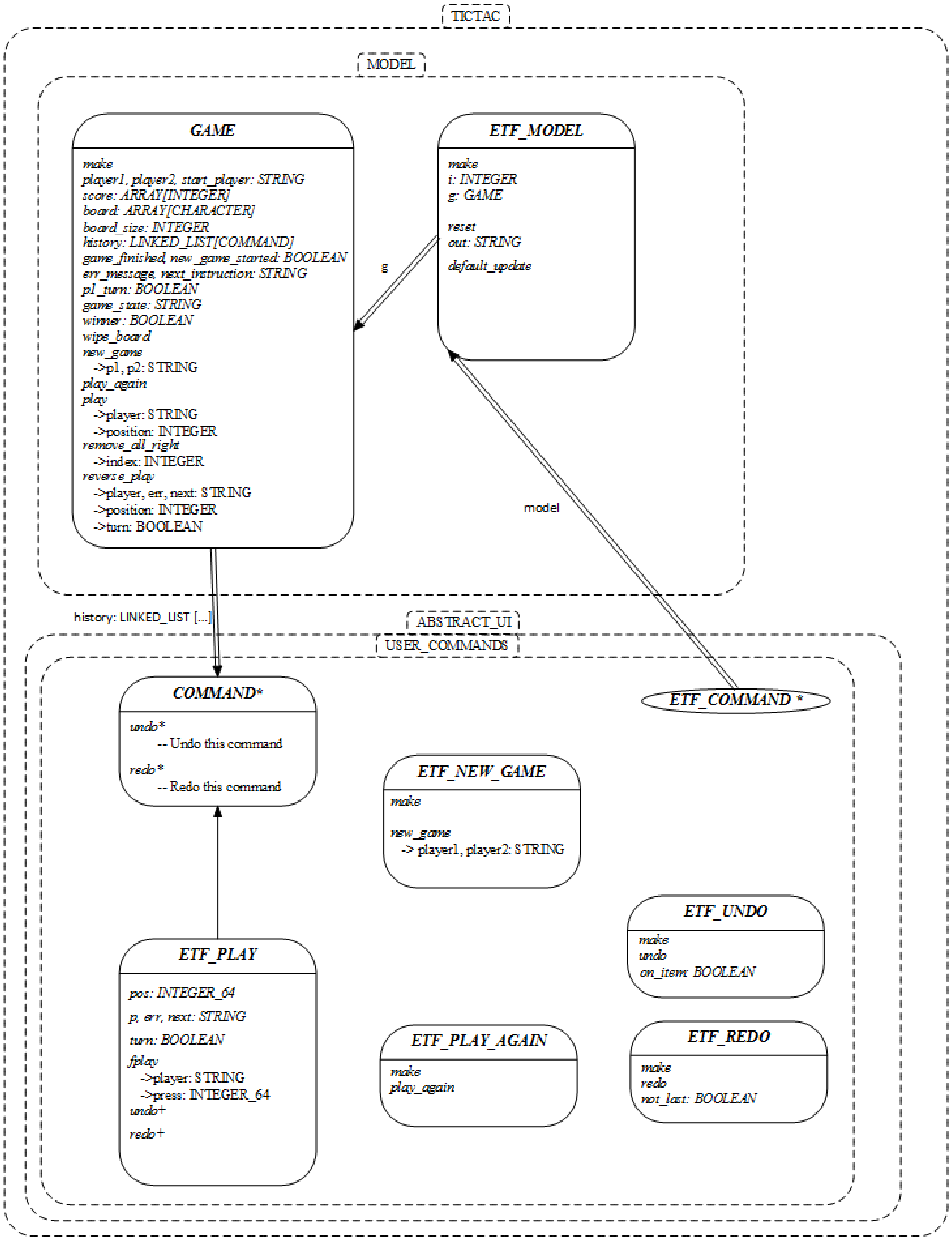
By: Matthew MacEachern (Prism: mcmaceac)

Table of Contents

1. Top Level View of the Design



(detailed view)



2. Significant Modules

Module Name	Description
GAME	The game class is the main core of my design and the engine of the program that performs all the core features and stores all the essential information about the state of the game. This class stores many attributes pertaining to the game such as the names of the two players (player1 and player2). A string variable called start_player is used to which player went first in each round to properly change the starter player after every round. A boolean variable p1_turn is used to indicate if it is currently player1's turn or not. This is used in several of the game's routines to be discussed shortly. An integer array score is used to track the score of the two players, index 1 stores the score for player 1 and index 2 stores the score for player 2. The character array board stores the contents of the game board, which means it will hold 9 characters in total for our 3x3 game. These attributes mentioned are exported to NONE and are therefore private to the game class to ensure that proper information hiding is in place. The linked_list of commands called history is a storage area for the commands used in the game, and is what provides our game with undo and redo functionality. This is discussed in more detail in the undo / redo design section of this report. The variables err_message and next_instruction are used to give instruction via command line output to the user on what is happening after each command they enter. These attributes are exported to ETF_COMMAND to ensure that the commands can get proper information about the game and know what is happening with it. Next are the game's core routines that allow it to function. The wipe_board routine simply inserts '_' characters into all the positions on the game board. The new_game makes use of this wipe_board, but not after performing some error checking and updating the error message and next_instruction accordingly. The play_again routine is like new_game except the current player information and scores are saved. The play routine is the core routine of the game. It places the appropriate mark for the player at the specified position, but not before performing error checking first. Within the play routine after a play has been made it checks if there is a winner, a routine that is discussed in detail in the win detection section. The remove_all_right and reverse_play routines are used for the undo and redo functionality which is discussed in the undo/redo design section. These routines are exported to ETF_COMMAND to ensure that information hiding is preserved.
COMMAND	The command class is a simple deferred class that has two deferred routines undo and redo. This class is used to facilitate the command design pattern and two ensure that undo and redo work properly for the program. This is discussed in more detail in the undo/redo design section of this report.

ETF_PLAY	The etf_play class is important in that it is the only command in the current design that inherits from the command class and therefore is the only command with the undo/redo functionality (discussed in detail in the undo/redo design section).
ETF_UNDO	The etf_undo class implements the design discussed in the textbook for the command design pattern with multiple level undos. This class is essential to providing the undo functionality that my design has.
ETF_REDO	The etf_redo class implements the design discussed in the textbook for the command design pattern. This class is essential to providing the redo functionality that my design has.

3. Win Detection

My win detection design does not try to be more complicated than it needs to be. The detection that a win has occurred resides in the winner routine within my GAME class. This routine consists of a brute force check of all the possible wins. That is, it checks to see the three possible horizontal line wins, the three possible vertical line wins, and the two possible cross board wins. If the marks on my board in any of these eight situations equal each other (and the marks on the lines are not the '_' mark indicating an empty position, since all positions start off this way), then a line has been achieved by one of the players.

Wins are checked only when needed. The only time that a win is possible is directly after a player has made a valid play (i.e. they have successfully placed their mark on the board). This means that the game only needs to check if there is a win after this has happened, which occurs inside my play routine within GAME.

My win routine keeps things simple. It does not return who has won, since this can be checked in the play routine by seeing who has made the play. This makes the routine simple and reusable for other routines. For example, the game_finished routine is a BOOLEAN routine that makes use of the winner routine. This is useful for outside classes to be able to check if a game is still running or if the game is finished and a play_again is valid.

4. Undo / Redo Design

The undo and redo design I used follows the design specified in the textbook closely, and satisfies the design goals G1 through G4 mentioned in the lab specification. I only needed to create one additional class to facilitate these design goals in full. This one class, COMMAND is a deferred class that has two deferred features: undo and redo. Game commands that wish to have the undo and redo functionality must inherit from this COMMAND class.

Within the GAME class itself, there is a list of COMMAND elements called history[COMMAND]. I chose to make the list a LINKED_LIST to facilitate arbitrary-level undos and I would not need to resize the structure for every entry into it. This ensures that only enough storage space is used to hold the exact number of commands in the list, not more. In ETF_UNDO and ETF_REDO, the power of this list structure is shown. ETF_UNDO and ETF_REDO both do not inherit from COMMAND since in this specific domain it does not make sense for undo to have an undo routine, and similarly with redo. In ETF_UNDO, in the undo routine, I first check if the cursor for the history list is on an item. That is, it checks to see if the list is not empty and the cursor is not in the before position. If the cursor is on an item, then that item's undo routine is called, which uses the power of dynamic binding to ensure that the proper undo for the proper command is called. The cursor is then moved back in the list to the previous command, which is the same as the textbook's implementation of the command design pattern. In ETF_REDO, a similar set of instructions is used, except the redo routine checks that the list is not empty and the cursor is not on the last position in the list (meaning that the cursor is on the latest command entered, and the redo should do nothing). After this check, the item's redo function is called and dynamic binding is again used to ensure that the proper redo is called.

In my design, the only command that has undo / redo functionality is the play command. From testing with the oracle, I noticed that when a new game is successfully started, the undo command does not do anything. This was similar with the play again command. This led me to make the decision of having only the play command inherit from command, and having new game and play again wipe the history list when successfully executed. Within ETF_PLAY, I implement undo and redo since it inherits from COMMAND. Also within ETF_PLAY, I store three attributes when the play command is executed within the command state: the player name p, the position pos the command is trying to execute on, and the current error message err that the game is holding. These attributes are to ensure that the undo executes properly. The actual undo routine calls a custom routine for play within the GAME class called reverse_play. This routine clears the board at the position that is stored in the attribute for that specific ETF_PLAY command. Similar customized routines will need to be created for additional commands that may need to have undo functionality in the future, but this is ok as stated in the textbook. The redo routine in ETF_PLAY simply executes the play routine with the stored attributes mentioned previously.

When several undo commands are executed followed by a play command, the redo command should not do anything to change the game state. As mentioned in the textbook, this would make no sense. Therefore, when play is executed all the commands to the right of the current cursor position are cleared from the command history list, and then the current

ETF_PLAY command is added to the end of the history list. This only happens if the current game is not finished. If the game is finished, the history list is cleared, since performing undo / redo when a game has finished does not do anything, as confirmed by the oracle.

This mechanism is flexible, which makes it a powerful tool when it comes to object oriented design, and enables new commands to make use of it easily.