TicTac Lab Documentation

Mehrzad Bazhdanzadeh (*mehrzadb*)

Table of content

Aim of the Lab

In this Lab we were assigned to focus on creating a class or classes which would help client to be able to save data. This generally would be adapted in various different games or applications to be able to recall the previous states in which the application has been. As an example in TicTac players will play a hand and that hand will be changed by a certain move on a three by three board, making it nine possible positions. How to save information is necessary in making a decision on how to implement the following game game.

Classes and their use

In this Lab I have created three additional classes in order to make my game. First class is called “SAVE_POINT” in which is responsible to save one type of data information in to a list of string. Class “PLAYER” was made to create as game’s object player. And finally “BOARD” as our other game’s object type would be our final class. None of the above are singletons as there was no such need to create a singleton object within this project.

What does SAVE_POINT do

Imagine a game is required to keep track of scores, points, gold, items or etc. The class SAVE_POINT was implemented as a stack like list object to be able to recreate a save mechanism for any possible type of save information necessary within the game. However, it does not save the state of an object. Rather it saves primitive data in to a list by converting it to a string. Thus users of SAVE_POINT are required to call a specific function while calling SAVE_POINT to be able to input their desired primitive type. However, after loading the values, client may realize that the save information is of type STRING. Thus they would need to convert the information to an appropriate type by calling the right feature call on the STRING object.

Creation

The creation is done by calling “make”. It would initialize the object’s state.

Commands

All the following commands are used to add information to a SAVE_DATA object:

- ✔ save_string
- ✔ save_char

- ✔ save_int
- ✔ save_boolean
- ✔ save_double

And in order to reset the whole object, client may use the following command:

- ✔ data_reset

Queries

In order to load the recent data, client may be able to use the "load" command. In addition to pop the recent data, client can simply use "pop." Notice that if list is empty a special string of "*!NAN!*" will be outputted to the client. This indicates that no saves exists within the object's states.

To also add a test feature to our class I have added "to_string" which will print out all the saved information by most recent.

What does player do

Player was created in order to save basic information about the player. The player will be having three key values:

- ✔ Name
- ✔ Mark
- ✔ Score

These values are used to keep track of players and to perform further actions within game. It is best to note that "Mark" is used specific to the design specification of the game and it will strictly be 'X' or 'O' which will be later explained.

Creation

In order to create a player, we need to specify their names and the mark which is assigned to them in the game of TicTac. As such make(name, mark) command will initialize the object's state.

Commands

The "increase_score" is the only command within the class which would increase the score of the player by one.

Queries

In order to retrieve the player’s name or mark, we have specified certain queries such as the following:

- ✔ get_name: STRING
- ✔ get_mark: CHARACTER

In addition “to_string” query will return us a string in the following format:

“ [score]: score for “[name]” (as [mark])”

What does BOARD do

BOARD simply represents a game board for the TicTac game. As such it would be in the form of:

```
_ _ _
_ _ _
_ _ _
```

Where each position will later be filled with ‘X’ or ‘O’. The board will also contain the game logic of win and draw condition based on ‘X’ or ‘O’ inputs.

Creation

By calling “make”, the constructor call, the board will be set to a base set viewed above. I have represented my BOARD as an ARRAY of nine fields.

Commands

There are two distinct commands within the board:

- ✔ move(mark, position)
  where mark is ‘X’ or ‘O’ and position is previously identified as greater than 0 and less than 10
- ✔ reset_move(position)
  where it will set the position back to ‘_’

Queries

The following are the queries which will check winning conditions:

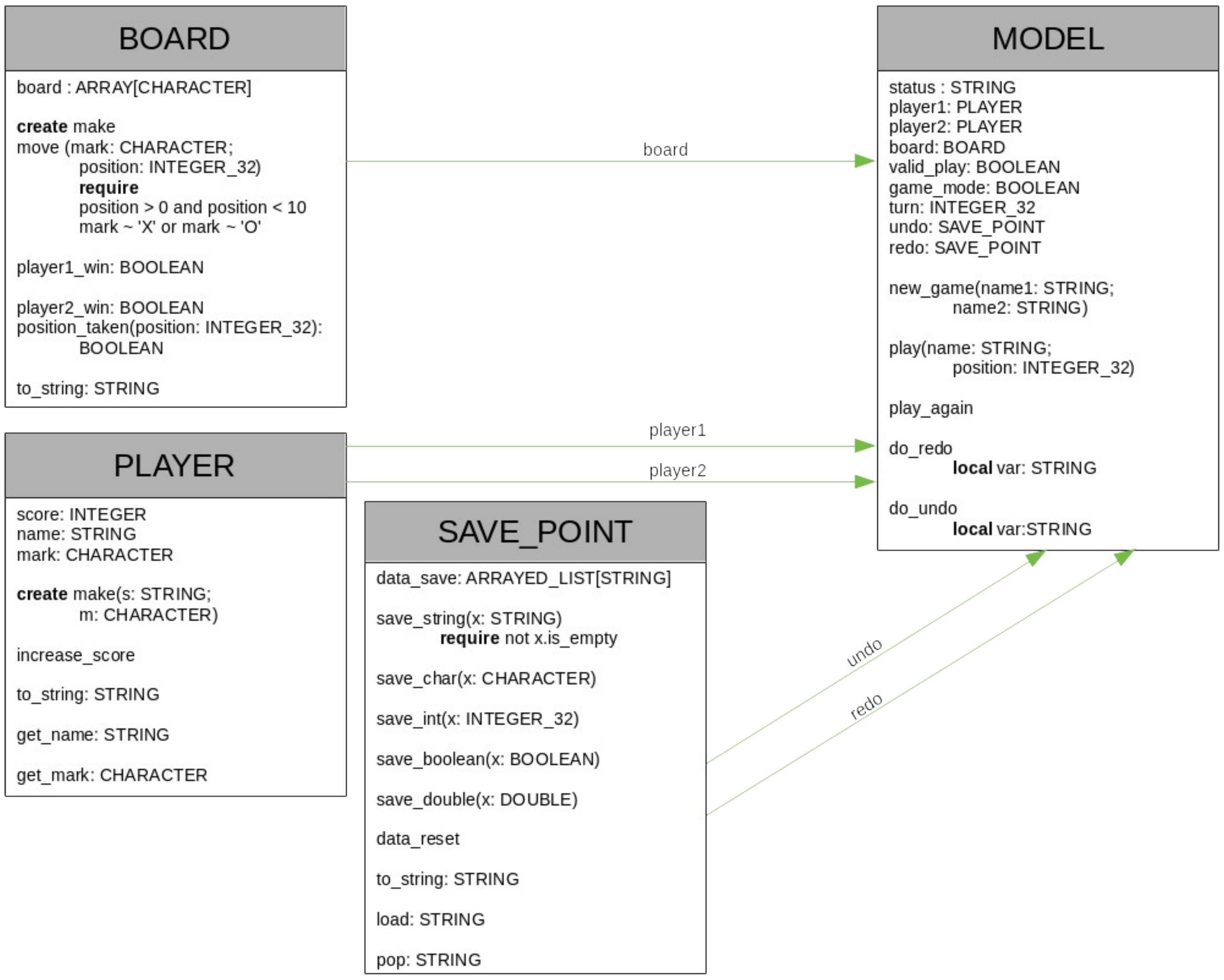
- ✔ player1_win: BOOLEAN
- ✔ player2_win: BOOLEAN

- ✔ full: BOOLEAN
  for draw

There are more queries to check the state of the board and printing the board. These are:

- ✔ position_taken(position): BOOLEAN
  tells if a position was previously played
- ✔ to_string: STRING
  will return out the updated board.

UML DIAGRAM



For better view you can open it within docs folder.

Tweaking MODEL

MODEL is where I have applied the game logic to the following program. Mainly during the command call of “play.” However, I would like to explain all commands step by step.

Commands

Before I dive in to commands, I would like to point out the use of “status: STRING.” This variable will be used to output the result of using a command, whether it was used properly or not, to our command line/GUI.

- *new_game*

To be able to start a game player will input new_game(“xxx”, “yyy”) in to the game input listener. This would take us to “new_game” command. In this command we will first check if the names are the same, then whether either starts with an integer, and if it passes all stages it is then we create first player “xxx” with mark of ‘X’ and second player of “yyy” with mark of ‘O.’ The reason this command is important is of two conditions which will follow to be true. First is “valid_play: BOOLEAN” which indicates it is a valid match between two people with two acceptable names. Next is “game_mode: BOOLEAN” in which we will choose to either start playing the game, by setting it to true, or not, as false.

- *play*

In order to be able to play the game we must ensure that we have “valid_play” and “game_mode” to both be true. Then we check (a) the turn is right, (b) the move the players are supposed to make is correct, and (c) player name is correct. If either of them is wrong we will indicate them by outputting the appropriate message by attaching it to status.

If all the above was correct and players make their move, we would then check for win condition by calling “BOARD’s” win queries. If we have one of them as right we increase scores appropriately, and we will set our “game_mode” to be false, as there needs to be a reset in game.

- *play_again*

This command requires the “valid_play” to be true. If not, no changes will happen within the game.

- *do_redo* and *do_undo*

Both commands are only valid when are playing the game only. “game_mode” and “valid_play” are true.

If we undo a game we will save that to redo if we need to change it again, and the same for redo. When we do either commands, if the game is on and there is possible values within “SAVE_POINT” object, we will change turns accordingly. Thus if we undo and it was “xxx” turn, now it is “yyy” turn to play.

Both redo and undo are using “SAVE_POINT” object type, thus we are able to check if we have possible changes to be done according to our data. If no data exist we are prompted with “*!NAN!,*” there are no data to be changed.

It is also important to know when we undo and play redo will be reset to empty again.

Testing

To be able to test our undo and redo, I have implemented the output in a way that when two players with “test101” and “test202” exist, the output will print out the information which is held by undo and redo for every turn.

In addition, after compiling the project, you can easily test the differences by going in to ./Testing folder and executing ./bat to check at2.txt test cases which uses meld.