Yusuf Corr

11/22/2019

Documentation for Tic-Tac-Toe Game Creation

My initial thoughts were to break the game down into parts (objects) that would interact with each other. This is ultimately the way I formed the source code for the final game.

**IDE**

* SPYDER v4

I chose to use SPYDER due to its interactive terminal which I thought to be useful to test the game creation process.

**OBJECT CREATION**

* TPlayer (name/title, score, symbol) – player object used to create player data for game, interacts with board (in a set way). Inherits from Player class.
* Board (9X9 Grid, data) – what players are playing “on” (placing symbols on). Inherits from Grid class.
* Tic-Tac-Toe (game, win, players 1 & 2) – a class that defines (more specifically) the functionality of how players interact with the board and returns if a win has occurred. Creates both Board and TPlayer instances.

Using ideas of Object Oriented Programming, I created a custom library (so to speak) with abstract Grid and Player classes (so to be used in the future in other text-based game projects).

These Classes were set to be public and could easily be extended on in the creation of future subclasses that were more specific for Tic-Tac-Toe.

SPECIFICALLY, for the Board class (subclass of Grid), I left the underlying data to be INT only, that way to check if a spot was empty would be checking if the underlying data at that spot was TYPE INT. I also used a DICT for the underlying Data Structure due to it’s ability to be quickly updated and accessed.

**LIBRARY CREATION**

After creating initial objects, I decided to create a separate file in which the base abstract objects could be inherited to create specific classes and subclasses for my Tic Tac Toe game. This mainly focused on cutting and pasting original code into a new file. This left me with the following classes in my custom library:

* Player
* Grid

**METHOD CREATION**

After the initial objects were created with correct subclass attributes, I started to create functions based on how each object would interact and to meet guideline criteria. To do this I broke down the game play in steps players could take each turn:

If the board is not full, players could:

1. Choose (empty) location to place symbol
2. Place symbol on board

I put methods that would check if position was empty (if underlying data was and int), place a given player’s symbol on an empty location **(\_isPositionEmpty**) via entering a number position (**\_place**). After the ability to place a symbol was possible, I then worked on the following:

* End game when board is full
* Parameters around placing symbol

This led me to testing my code in the SPYDER interactive console. Since players can only enter int data types, it made parameter setting really easy (a general while loop with many if-elif statements). I also used the **\_isPositionEmpty** function here in an elif block to make sure a symbol could not be over-written. Lastly, using knowledge on % (and knowing that %2 is 0 for even numbers and 1 for odd numbers) I created the iterator turn (initialized at 0) and added 1 to it each time, then set the player for the next turn equal to players[turn%2], players = [player1, player2] (where player 1 and 2 are Player objects with a specific symbol and designated titles).

I then implemented an update method, to update the row, column, and diagonal attributes. I also implemented a **\_hasWon** function that checked if in each row, column or diagonal had 3 matching symbols STARTING ONLY AT TURN 5(or when turn was greater than or equal to 4). The reason for this is that before the 5th turn, there are only 2 or each players symbols on the Board.

I ENDED UP getting rid of the row, column, and diag attribute since it took up space and needed more supporting code. Instead, I created a **\_check\_vector** function to check a given vector dictionary (i.e. all horizontal vectors or rows dict) and created local variables for that function that worked the same.

Instead of terminating the game if the board was full and nobody had won, I decided to cap a turns limit to 9 (otherwise Board is full after 9 turns and no one had won yet).

The main thing I encountered when editing my code was encapsulating code to make it reusable and efficient.

**TEST**

For testing purposes, I had an idea to write test code, but instead used SPYDER’s interactive terminal to test my game and specific attributes and methods. Though not efficient, I was able to sort out many parameters that could cause a bug in my game and used a while loop with conditions to check user input.

Big O was taken into consideration to a lesser extent but seeing how this game does not require much data it seemed to work in my favor.

I plan on continuing to learn both Big O and specifics on how to write test code so that I can create features more efficiently.

**IDEAS**

Sorting out how my game would be won was a little tricky. Initially I was going to have sequential data saved so that if a players “placement history” matched a specific sequence, they win the game. This code quickly got out of hand, and I had to step away and revisit a different way to solve the problem.

This led me to this video which suggested a different way to solve the solution:

* <https://www.youtube.com/watch?v=BHh654_7Cmw&t=1965s>

I only used the video to look at a different way to solve how players won. Because I used a dictionary for the underlying data structure, I decided to have local variables called columns, rows, and diag to contain the correct key sequence for each row, column and so on. This then allowed me to check if a row, column or so on had 3 matching symbols.

Problem solved.

After this I had many people test and play my game, thus finding more bugs and fixing them.

Below are some of the idea’s I white-boarded in the process of making the game.





