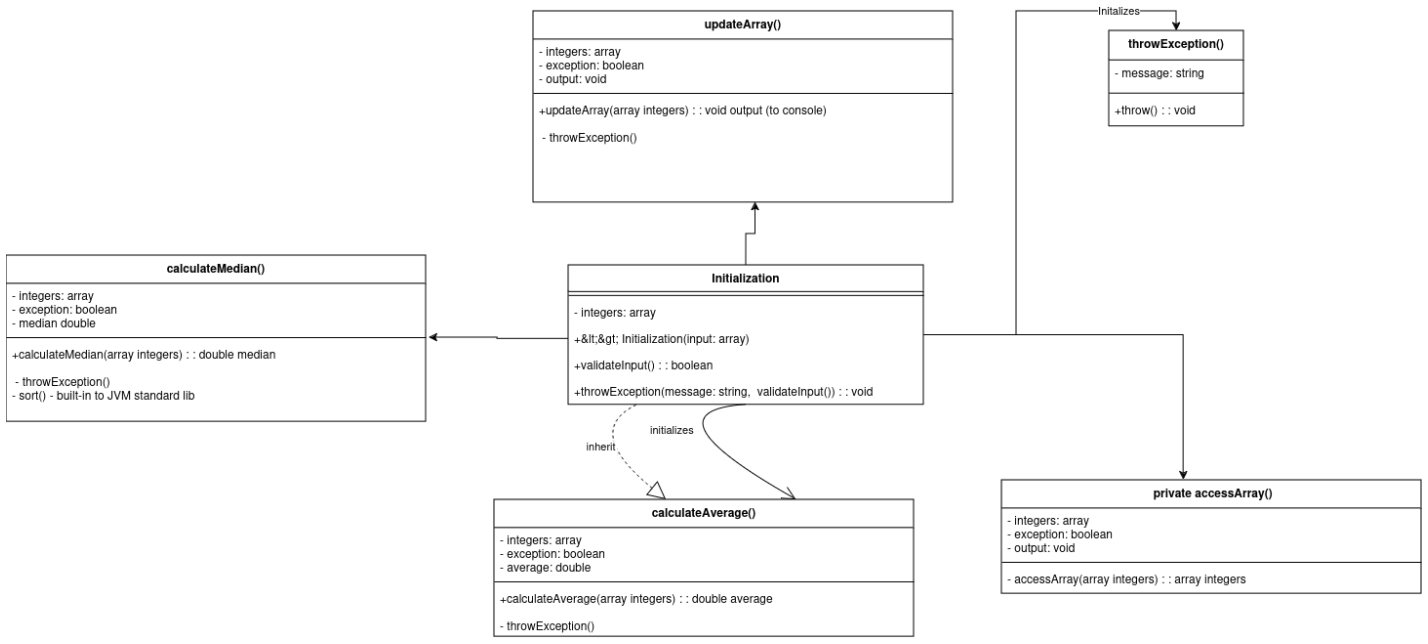
Assignment06 Report:

UML Diagram:



Q1:

My design reflects the encapsulation principle by keeping each function relevant to the skeleton of the game *enclosed* in each of their respective functions. This is represented via a counter-example, as we could have initialized the grid and game as a single function called ***initialize(),*** however this would be against the Encapsulation SOLID / OOP principle.

Q2:

The game flow reflects good design principles via having a single void: play() function that runs within a while loop and calls various functions associated with external classes to retain a modular design that handles all exceptions and cases.

The challenges I faced with handling user input was resource leaks, as in one of my functions, specifically Player.makeMove() I cannot close my scanner after each call or else it will never take input again.

The fix to this would be to pass in scanner with System.in as a parameter to makeMove() but this is confusing visually and so I opted to leave it as is, as it doesn't cause any errors.

Q3:

I implemented polymorphism by having a single Player class and adapting its name, piece and a boolean variable that determines if we are initializing a player as a computer or a real player.

The boolean: isComputer affects the makemove(..) function, so instead of having a makeMovePlayer or makeMoveComputer we have a single function that adjusts based on the isComputer vaue..

Dynamic dispatch improves the flexibility via the Main Class which utilizes dynamic dispatch when initializing player1 and player2 with different parameters at runtime.

Q4:

I checked for the winning condition by returning a boolean either true or false, based on whether the diagonals of the board are all the same, this is hardcoded for a 3×3 grid but for the horizontal winning conditions we have a for-loop that iterates the I variable checking board[i][0] & board[i][1] & board[i][2] or for the vertical winning conditions we check: board[0][i] & board[1][i] board[2][i] are the same.

I structured the game to allow for restarts without duplicating code, by having the play() function in a while loop with a condition that is adjusted based on the boolean restart() returns value, which returns a boolean value that determines if the play loop continues to run or not.

Q5:

The refactoring changes I made was to make drawBoard() into a function and resetBoard() functions. resetBoard() is now called if restart() is going to return True.

It adheres to OOP principles as I have 3 classes with various single-responsibility functions and only share as much information within the classes as necessary.

Mainly I made sure that all functions and variables follow the self-documentation principle, the Game class follows the interface principle, and within each class the interface principle is followed by sharing just as much information as necessary to perform the correct functionality.

Q6:

The OOP principles that helped me reuse the previously implemented classes was single-responsibility, divide and conquer,and encapsulation. Since the code was well distributed and broken up into single purpose functions and classes that contained code specific to its purpose served in the TicTacToe game, it allowed me to only adjust the code within those previously defined functions to allow for dynamic allocation of the grid and win conditions, I did not have to reinvent the wheel.

The root cause of those difficulties was mainly in checking the win condition and redrawing the board to fit any size, I tackled those difficulties by creating iteration tables and parsing the code multiple times to find where exactly the bug is occurring in the logic.