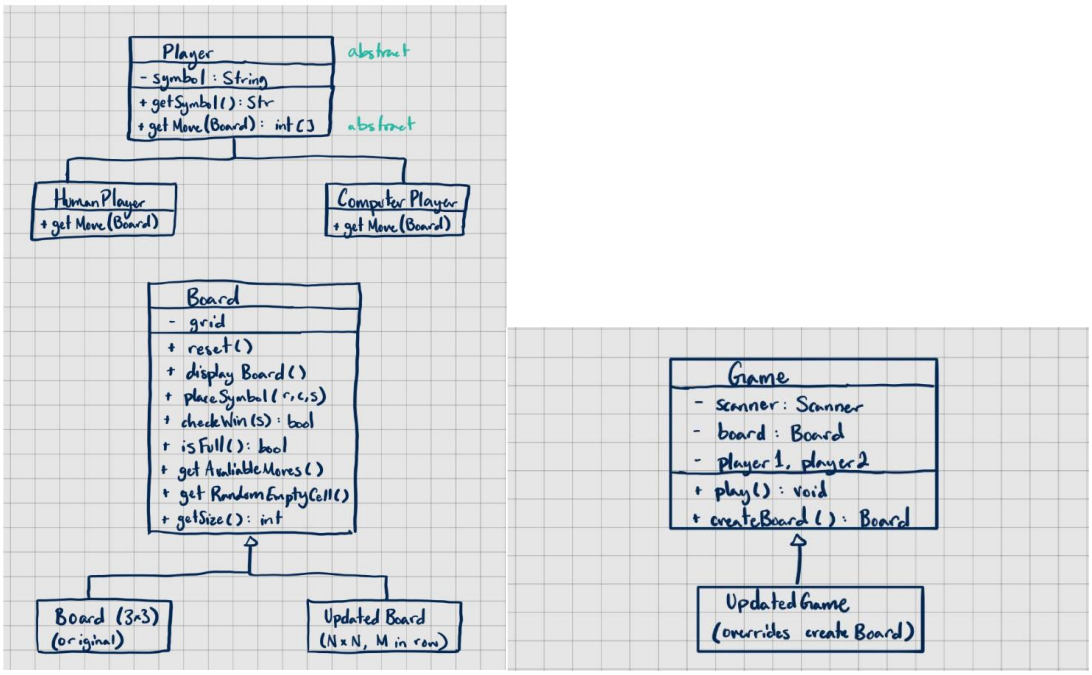
Assignment 06 – Report

Lakshmi Saranya Alamanda

Step 1

Draw the UML of each core class and explain your design.



How does your design reflect the encapsulation principle?

My code encapsulates data inside classes so that each class controls its own internal state:

- Game never modifies the board directly, it uses Board's public methods
- Player stores its own symbols and hides how moves are produced (ex: human input or random)
- Board hides the underlying grid and only exposes safe methods (ex: placeSymbol, getCell)

Step 2

How does the game flow reflect good design principles?

The game flow implements good design principles including display and state are handled by Board, input and move decisions by Player subclasses, and main functions by Game. The functions of Game include display the board, request a move from the current player, attempts to place the move, and checks for win/draw conditions. This structure applies to the Single Responsibility Principle (each class handles one concern), uses clear method contracts and avoids code duplication by centralizing the loop in Game.

What challenges did you face in handling user input?

I had scanner issues as I used nextInt() and nextLine() together, which made the leftover newline character get stored in the nextLine(). The solution I found was to use a blank nextLine() and parse with Integer.parseInt() to deliberately consume the newline character after nextInt(). The second issue I had was the user inputting the coordinates in the form (2,3). Originally I misread the question and took the coordinates as 2 separate inputs, I changed this by using a string to store the input in the form (2,3), and used trimming and splitting at the comma into an array that is then used to store the row and column values. Additionally, I needed to do -1 on both coordinates to convert it to a 0-based indexes, to void an ArrayIndexOutOfBoundsException.

Step 3

How did you implement polymorphism in your player classes?

Polymorphism is achieved by declaring an abstract method, public abstract int[] getMove(Board board) in Player. HumanPlayer and ComputerPlayer override this method with distinct implementation. HumanPlayer.getMove reads and validates console input. ComputerPlayer.getMove returns a random empty cell obtained from the board. When Game is run it stores players in variables typed as Player, which calls to current.getMove(board) and overrides the method appropriate to the objects concrete class.

How does dynamic dispatch improve the flexibility of your game?

Dynamic dispatch allows Game to operate on the abstract Player type, the concrete behaviour is selected at runtime. This results in new player types being added without touching the Game logic, mode selection requires only creating the appropriate player objects, and looping the game is still generic and reuseable, reducing duplication.

Step 4

How did you check for winning condition?

The method checkWin in Board class goes through every grid cell, checking for matching symbols. It checks outward along four directions; right, down, down-right, and down-left. For each direction it checks the next cell, first verifying that it is in bounds then it accesses the grid. If consecutive matching is found, the method returns true. Bounds are enforced to avoid indexing errors.

How did you structure the game to allow for restarts without duplicating code?

The game was built so that the control flow for running one match is contained entirely inside a single loop in the Game class. All steps including, displaying the board, receiving moves, placing symbols, checking for wins, checking for draws and switching players, are performed inside this loop and not duplicated anywhere else. Because the loop is self-contained, restarting the game does not require rewriting or copying the logic. You can restart the project by simply resetting the board and run the same loop again. The player objects and game structure do not need to be recreated, as the logic doesn't depend on any external state outside of the board.

Step 5

What improvements did you make during refactoring?

Some improvements that I made include single scanner usage by moving Scanner into Game. Modularization, I kept the board logic entirely into Board and player input into HumanPlayer, reducing Game complexity. I also used a lot of helper methods in Board, including getAvailableMoves(), getRandomEmptyCell(), and getSize() which help keep higher-level logic clean. These refactoring's improved readability, maintainability, and reduced bug surface.

How does your final codebase adhere to the OOP principles?

Encapsulation: Board hides its grid, player hides move selection logic, public methods form the exposed contract.

Abstraction: Player and Board define concise interfaces; implementations vary behind those interfaces.

Inheritance & Polymorphism: HumanPlayer and ComputerPlayer inherit from Player. Polymorphic calls (getMove, checkWin) allow flexible behaviour.

Single Responsibility: Each class has a focused responsibility (control flow, state, input).

Open–Closed Principle: New player/board types can be added by extending base classes; existing classes do not need modification.

Information Hiding: Internal grid representation is not exposed; interaction is through methods.

Step 7

What OOP principles help you reuse the previously implemented classes in

developing the upgraded game? Why do the OOP principles help?

Inheritance: UpdatedBoard extends the base Board (or implements the same interface), making it substitutable wherever a Board is expected; UpdatedGame extends Game and only overrides

createBoard() to inject the upgraded board. This preserves the contract between classes and enables reuse.

Polymorphism & Dynamic Dispatch: Game calls methods on Board and Player abstractly; concrete implementations can vary. This decouples Game from the precise board representation.

Encapsulation & Interface Contracts: Because Board and Player expose stable methods, UpdatedBoard can adhere to that contract and be used without changing Game or Player.

These principles together allow the upgraded project to reuse previously tested logic and to minimize changes to existing artefact

If you encounter difficulties in developing the upgraded game, what are the root

cause of those difficulties? How do you tackle the difficulties?

Some difficulties that I encountered include the scanner buffering issue, which was caused by mixing nextInt() and nextLine(). Mutating loop variables in algorithms, in checkWin mutated outer loop variables inside inner loops, producing out-of-bounds errors. These problems were solved by using nextLine() consistently and Integer.parseInt() for numeric parsing; or consumed newline after nextInt() when necessary. And for the loop variables I rewrote checkWin() to compute currRow = start + step * stepSize rather than incrementing the outer-loop variables