For the design of this program, my main aim was to emphasize SOLID principles. I upheld the Single Responsibility Principle by ensuring that all game logic was placed in the Game class, while the GamePlayer class only controlled the mechanics of running the game and providing the interface to the user. This meant that for example, no print statements exist within the Game class, as printing to the screen is solely a user interface action. In addition, anyone wishing to play a game with modified rules only needs the GamePlayer class to do so, ensuring this class is responsible to the player, while the Game class remains responsible to only the programmer.

I tried to make it so that the program was open to extension and followed the Open-Closed Principle. I separated out the different checks for a win into private helper methods for better readability and so that they can be switched in and out if possibly, in the future one wants to implement different combinations of streaks to count as a win. Any logic that should stay consistent across these changes is kept in isPlaying, such as the updating of the winner. This is to ensure a lack of coupling or code duplication.

I maintained the Dependency Inversion Principle by making sure there were no dependencies on hard-coded numbers. This not only made it so that there weren't dependencies on anything concrete, but also allows the player to have more freedom over the game by giving them the ability to change the connect goal, dimensions of the board, and number of players. Additionally, I had it so that the Game class didn't depend on the GamePlayer class and ensured that source code dependencies only point to the inner software layers of the Clean Architecture. This has the benefit of being able to easily switch out the interface if we were to implement a GUI in the future or if we simply wanted to use the Game class in another program without the GamePlayer class.

Finally, a note on runtime. For the win check helper methods, my initial thought was to simply store where the last slot was filled and have the program iterate through the entire matching row, column, and diagonals to see if there was a streak of four adjacent matching filled slots. However, this proved to be unnecessary as the program only needs to check slots adjacent to the slot that was just filled. By changing my algorithm to start at the last slot filled, branch out from there, count the length of the streak, and stop once the streak is broken, I was able to simplify the runtime of the program from $\Theta(max(numRows,\ numCols))$ complexity to $\Theta(min(connect,\ max(numRows,\ numCols)))$ complexity, which can be much smaller (An example would be Game(connect: 4, numRows: 1000, numCols: 1000)).