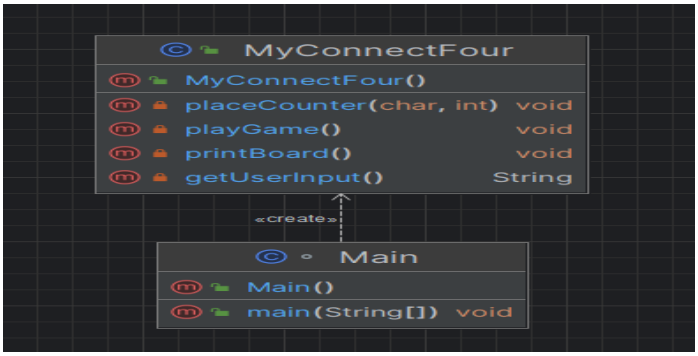
Requirement 2: How to Restructure the Connect4 Game:

Current Condition of the Connect4 Game.

- UML Diagram of Connect4 Game after solving bugs.



As we can see in the diagram above, the Main class is directly related to the MyConnectFour class. In terms of high cohesion and loose coupling, this is not the best application design. Let me briefly explain high cohesion and loose coupling. High cohesion is somewhat akin to the Single Responsibility Principle (SRP) of the S.O.L.I.D approach. To achieve high cohesion, each class or function needs to have a clear and specific purpose.

On the other hand, loose coupling is nearly like the O (Open/Closed Principle) and D (Dependency Inversion Principle) parts of the S.O.L.I.D principles. In brief, to achieve loose coupling, your classes should not depend on each other directly. Instead, classes need to depend on abstractions.

In the provided Connect4 application, the MyConnectFour class attempts to handle all the game logic within one class and runs the application in the main class. As a result, this setup causes low cohesion and tight coupling. Firstly, we need to understand the purpose of each method in this class before creating a new UML diagram for this application.

Problems:

1. **Constructor Method Problem:** The MyConnectFour constructor initializes the play game function, as well as the board and input variables. In general, it is recommended to initialize variables, classes, objects, and interfaces using the dependency injection principle. Ideally, gameplay and the board should be in separate classes, each depending on their respective interfaces. These interfaces, in turn, should be initialized through constructor dependency injection.
    - **PlaceCounter Method Problem:** There are several significant issues associated with this method.
      Wrong Loops: The method contains two incorrect loop implementations. One of them leads to an infinite loop, and the other loop is flawed.
      Wrong Method Creation: This method aims to record **player movements** on the **board** using variables. Ideally, the **player**, **player movement** (gameplay), and **board** should be encapsulated within separate classes, each having its own interface. When the board size, player names, or gameplay style changes, relying on a single method becomes problematic. It is advisable to adopt a more object-oriented approach, distributing responsibilities among classes. This way, modifications to specific aspects won't necessitate rewriting the entire method.
2. **PlayGame Method Problem:** This method is completely wrong in terms of Single Responsibility. Let me illustrate this in a humorous way. Imagine this method aspiring to be Steven Gerrard for Liverpool or Lionel Messi for Barcelona. It's like a Swiss army knife trying to be everything and expecting to handle every task thrown at it.

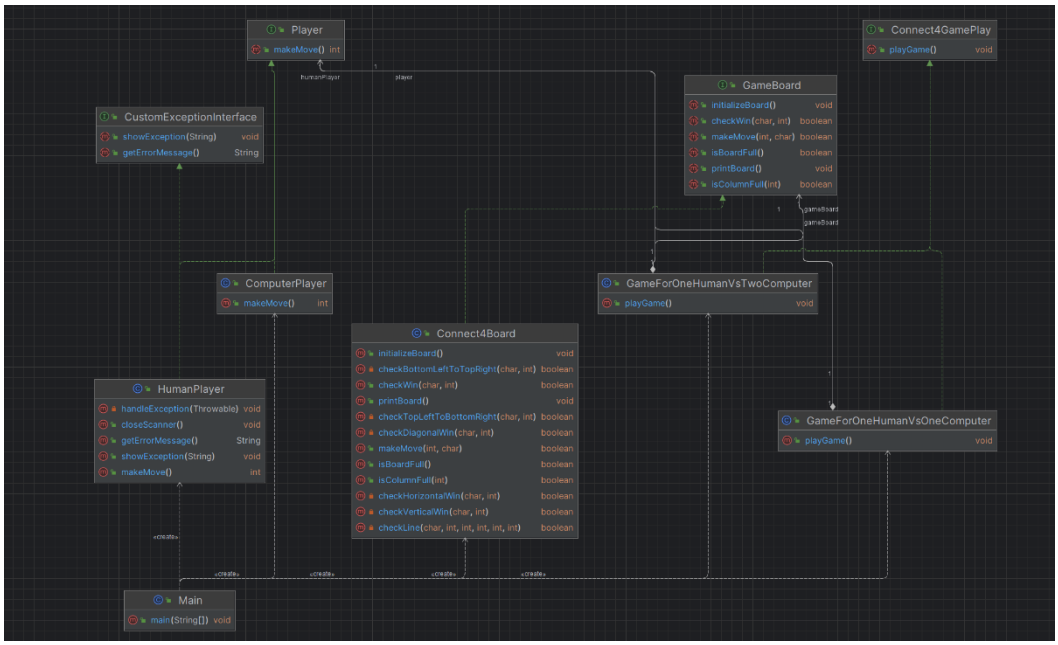
- **Polymorphism Problem**
  The method currently uses separate checks for vertical and horizontal wins for two players. Consider consolidating these under a generic **checkWin** method. Also, the checkDiagonalWin, checkBottomLeftToTopRight, checkBottomLeftToTopRight, and checkLine methods lack comprehensive winning condition checks. Extend these methods to cover all necessary conditions.
- **Players Problem:** This method handles user input and implements game logic based on player actions. For flexibility, consider creating separate classes for human and computer players, each inheriting from a common player interface. This way, changes in player names or additions won't directly impact the method, promoting a more modular and adaptable design.
- **Magic Number 4 Problem:** The method currently relies on a fixed magic number (4) to determine the winning condition, suitable for Connect4. However, this approach isn't flexible for ConnectN, where the winning condition can vary between 2 and 7. Consider modifying the method to handle dynamic winning conditions for more adaptable gameplay, accommodating different variants of the ConnectN game.
- **Println Problem:** This functionality can be encapsulated in another class with its own interface. An Animation interface could include methods like welcome, congratulations, and commiserations. These methods can then be implemented in a dedicated Animation class, adhering to the Animation interface. This modular approach promotes code separation and enhances extensibility.
- **PrintBoard Method Problem:** The printBoard() method, despite some omissions and potential bugs affecting board size as mentioned in Requirement 1, is appreciated for its functionality. Consider refactoring it into a separate class and interface for the game. In my solution's UML diagram, I will illustrate this proposed structure.

5.GetUserInput Method Problem:

- When you enter another character, the Connect4 game crashes. This method needs to prevent this situation and prompt the player to provide appropriate input by offering suggestions without crashing the game.

New Version of the Connect4 & ConnectN Game.

- UML Diagram of Connect4 & ConnectN Game



Pros:

- I researched that there are 8 different versions of loose coupling, including (No, Minimal, Temporal, Control, Content, Common, External, Semantic) coupling. I am not absolutely sure which one best describes my design. However, I have succeeded in making classes independent from each other. They can derive their signature from their interfaces. Ultimately, I have mitigated the tight coupling problem within the application.
- I created a **CustomExceptionInterface** and applied it to the **HumanPlayer**, as I assumed that the computer would never input incorrect data. Initially, I considered using an abstract class instead of an interface for the human player. However, this choice led to issues in my **GameForOneHumanVsOneComputer** and **GameForOneHumanVsTwoComputer** classes. Consequently, I opted for interfaces to address these problems.
- Additionally, I researched that there are 6 different types of cohesion: Functional, Sequential, Communicational, Procedural, Temporal, and Coincidental. I believe I have succeeded in reducing code repetition by employing inheritance, polymorphism, and dependency injection in my design.
- I successfully developed a ConnectN game for one human player versus two computer players.
- I enhanced the user interface, making it more useful and colorful.

Cons:

- To successfully create the ConnectN game, I initially duplicated classes based on the number of players. This was necessary as the addition of a second computer player didn't work seamlessly due to the design of the gameboard, which was originally suitable for Connect4, allowing only two players.

  To address this, I redesigned the **playGame**() method, customizing it for ConnectN. However, this adjustment didn't fully meet the requirements for ConnectN, especially considering the winning condition where N is between 2 and 7.

  As a temporary solution, I added an integer parameter to the **checkWin** method in the **GameBoard** interface. Its default value is set to 4 for One vs One. For One vs Two, the input is taken from the user, and the gameplay implements winning logic according to the specified value of N entered by the user in the **GameForOneHumanVsTwoComputer** class. This can be considered a brute force solution to adapt to the ConnectN requirements.

Caution: I did not want to use **encapsulation** intentionally by creating **getter** and **setter** methods because it can sometimes cause tight coupling. Let me explain with an example. If the project were related to a banking application that has an object for customer information containing 100 different variables, you could write getter and setter methods, or you could use a record instead of a class. However, for a Connect4 game, where there are only a few columns in classes, I didn't want to introduce tight coupling due to getter methods. Instead, I opted to use **dependency injection**.