OOP design patterns and principles used in this implementation:

* **Abstraction**:
  + The Player class is abstract, providing a common interface for different player types.
  + Why: It allows for different implementations of players while ensuring they all have the necessary methods.
* **Encapsulation**:
  + Each class encapsulates its data and behavior.
  + Why: It helps in maintaining the integrity of the data and provides a clear interface for interaction.
* **Inheritance**:
  + HumanPlayer and ComputerPlayer inherit from the Player class.
  + Why: It promotes code reuse and allows for specialized implementations of different player types.
* **Polymorphism**:
  + The GetMove method is implemented differently in HumanPlayer and ComputerPlayer.
  + Why: It allows for different behavior based on the player type while using a common interface.
* **Single Responsibility Principle**:
  + Each class has a single, well-defined responsibility (e.g., Board manages the game board, Game manages the game flow).
  + Why: It makes the code more modular, easier to understand, and easier to maintain.
* **Open/Closed Principle**:
  + The design allows for easy addition of new player types or game variations without modifying existing code.
  + Why: It makes the system more extensible and reduces the risk of introducing bugs in existing functionality.
* **Strategy Pattern**:
  + Different player types (Human and Computer) implement different strategies for making moves.
  + Why: It allows for easy swapping of player types and addition of new player strategies.
* **Composition**:
  + The Game class composes Board, Player, and Piece objects.
  + Why: It creates a flexible structure that can be easily modified or extended.
* **Factory Method (simplified)**:
  + The Game class creates player objects based on user input.
  + Why: It centralizes object creation and allows for easy modification of the creation process.
* **Iterator Pattern (implicit)**:
  + The use of List<Board> and iterating over it in various methods.
  + Why: It provides a standard way to access elements of the collection without exposing its underlying structure.

These patterns and principles work together to create a flexible, maintainable, and extensible design for the Notakto game. They allow for easy modifications, such as adding new player types or changing game rules, while keeping the core structure intact.

---

Here's an analysis of the design principles and patterns used:

**Design Principles**

* **Single Responsibility Principle (SRP)**:
  + Each class has a specific responsibility. For example, NotaktoBoard handles board-related operations, Player manages player actions, and Game controls the game flow.
* **Open/Closed Principle (OCP)**:
  + The Player class is abstract and can be extended (e.g., HumanPlayer and ComputerPlayer) without modifying existing code.
* **Liskov Substitution Principle (LSP)**:
  + Subclasses like HumanPlayer and ComputerPlayer can be used interchangeably with the Player base class.
* **Interface Segregation Principle (ISP)**:
  + While not explicitly using interfaces, the classes have focused methods that align with this principle.
* **Dependency Inversion Principle (DIP)**:
  + High-level modules (like Game) depend on abstractions (like Player) rather than concrete implementations.

**Design Patterns**

* **State Pattern**:
  + Participating Classes: GameState, Game
  + Important Operations: PauseGame(), ResumeGame()
  + Justification: Manages the game's state (paused, resumed) and allows for easy state transitions.
* **Strategy Pattern**:
  + Participating Classes: Player (abstract), HumanPlayer, ComputerPlayer
  + Important Operations: GetMove()
  + Justification: Allows for different player strategies (human input vs. computer algorithm) to be interchangeable.
* **Command Pattern**:
  + Participating Classes: Move, Game
  + Important Operations: Undo(), Redo()
  + Justification: Encapsulates move actions as objects, allowing for easy undo/redo functionality.
* **Template Method Pattern**:
  + Participating Classes: Board (abstract), NotaktoBoard
  + Important Operations: PlacePiece(), CheckForWin()
  + Justification: Defines the skeleton of board operations in the abstract class, with specific implementations in the concrete class.
* **Singleton Pattern** (implied):
  + Participating Classes: Board (with static AllBoards list)
  + Important Operations: AddBoard(), ResetAllBoards()
  + Justification: Maintains a single collection of all boards across the game.
* **Memento Pattern** (partially implemented):
  + Participating Classes: GameState, Game
  + Important Operations: SaveGame(), LoadSavedGame()
  + Justification: Allows for saving and restoring game state.

**Classes/Interfaces to be Reused from Existing Libraries and Frameworks**

* **System Collections:**
  + List<T>
  + Stack<T>
* **System IO:**
  + File
* **System Text Json:**
  + JsonSerializer
* **System:**
  + Console
  + Random
  + Serializable attribute

These classes and interfaces are used for data structures, file operations, JSON serialization, console I/O, and random number generation, providing essential functionality for the game implementation without reinventing the wheel.