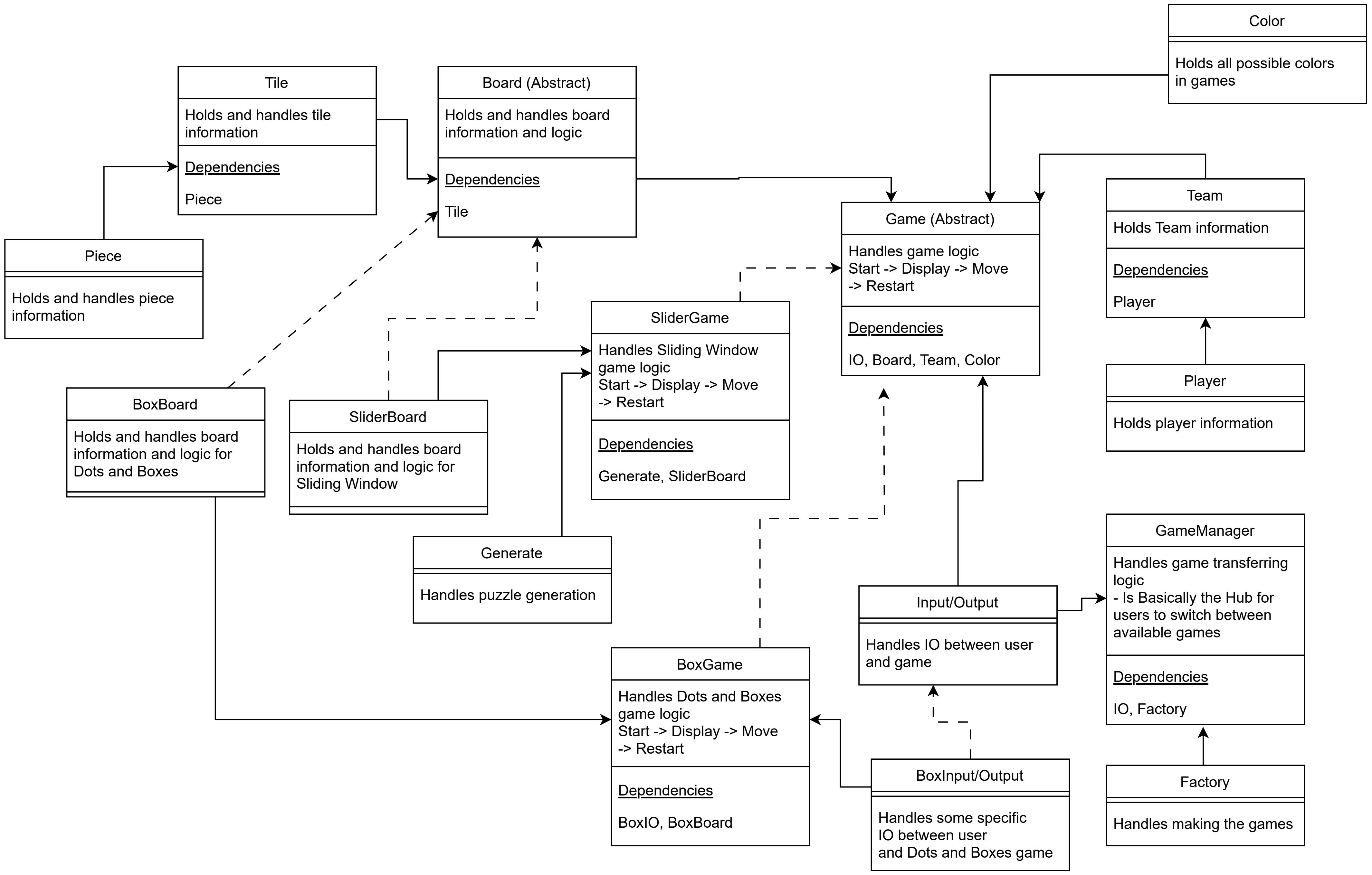
Color
Holds all possible colors in games
Tile
Holds and handles tile information
Dependencies
Piece
Board (Abstract)
Holds and handles board information and logic
Dependencies
Tile
Game (Abstract)
Handles game logic
Start -> Display -> Move -> Restart
Dependencies
IO, Board, Team, Color
Team
Holds Team information
Dependencies
Player
Piece
Holds and handles piece information
SliderGame
Handles Sliding Window game logic
Start -> Display -> Move -> Restart
Dependencies
Generate, SliderBoard
Player
Holds player information
BoxBoard
Holds and handles board information and logic for Dots and Boxes
SliderBoard
Holds and handles board information and logic for Sliding Window
Generate
Handles puzzle generation
GameManager
Handles game transferring logic
- Is Basically the Hub for users to switch between available games
Dependencies
IO, Factory
Input/Output
Handles IO between user and game
BoxGame
Handles Dots and Boxes game logic
Start -> Display -> Move -> Restart
Dependencies
BoxIO, BoxBoard
BoxInput/Output
Handles some specific IO between user and Dots and Boxes game
Factory
Handles making the games

Explains class structure at a high level through UML:
Everything starts at GameManager that initalizes the general IO, and with this IO, has Factory make games of type Game. This also means that the specific games, in this case BoxGame and SliderGame, are extensions of Game. Each type Game would have the components Board, Team, Color, and the IO passed to it via Factory. Not all Game have to Generate a puzzle, so currently only SliderGame has this component. Like before, each game has a specific board of Board: BoxBoard and SliderBoard extends Board. Each Board then has the component Tile and each Tile has the component Piece. Each Team has component Player. Color class is used by Game to color code each Team. Lastly, the IO passed to Game is also extendable for specific needs of each game: this, like all other games is done in the Factory.

How your design allows for both scalability and extendibility:
My design demonstrates scalability and extendibility through its isolationist structure and well-defined abstractions. By organizing the code into separate classes responsible for specific aspects of the game, such as Game, Board, and IO, I've created a foundation that can easily accommodate the addition of new features or games. The use of abstract classes like Game and Board defines common behaviors and structures shared across different games, allowing new games to be implemented simply by creating classes that adhere to these interfaces. The Factory class further enhances scalability by providing a centralized mechanism for instantiating game objects based on user input. This allows the program to accommodate new games without requiring much modifications to existing code. Additionally, my design ensures that each class encapsulates specific functionality, minimizing the impact of changes on other parts of the system. Extending the IO class for specific games, such as BoxIO for Dots and Boxes, allows for tailored input/output behavior while maintaining a consistent IO interface for user interaction. (Please note, this interface doesn't mean the Java interface, it's just what people call the thing that gives and receives user feedback.) Overall, my design fosters scalability and extendibility by providing a flexible architecture that can adapt to future requirements and additions with ease. Proven by how GameManager can allow the users to play two different games given that each game is created as the same type within the Factory.

Any changes from past assignments (for future assignments):

One of the first things that changed from the past assignment was the abstraction of Board.java and Game.java so that they can be extended to any game instead of a concrete Game or Board class for specifically Sliding Window. These two .javas are also now abstract classes to accommodate this abstraction. Second thing I changed was that I've added a way for users to quit the game they are playing at anytime. Third was the addition of a GameManager.java, Factory.java, Color.java, and Team.java. The addition of both GameManager and Factory is to allow users the ability to change their board game as long as the program is still running. Color and Team were added to help users better distinguish between ally and foe when playing games with two or more parties. (This is my way to excusing why I didn't add color to Sliding Window) Fourthly, I made the decision to slightly abstract out the IO so that it is initiated at the GameManager and passed around to each subsequent object as needed. Of course, depending on the game, a new, more specific object of type IO might need to be passed instead. This is the case for Dots and Boxes since it's getPieceMove method behaves differently to general IO.