**FINAL – Yelgazy Nurai SE-2221**

**Tic-Tac-Toe game**

**Project Overview**

**Project Idea:** The main idea of this project is to create a modular and extensible Tic-Tac-Toe game in Java, showcasing the implementation of various design patterns. It provides a solid foundation for future expansion and customization of the game.

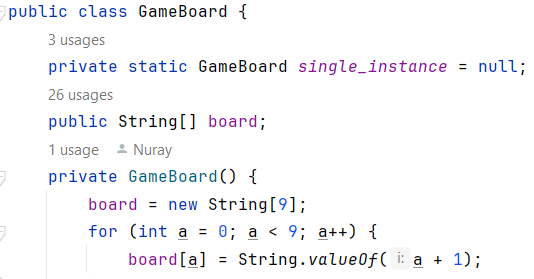
**Performance Goal:** To ensure smooth and responsive gameplay experience for users, with no noticeable delays or lag during player moves, AI calculations (if implemented), and UI interactions (if applicable), even on low-end hardware.

**Performance Objectives:**

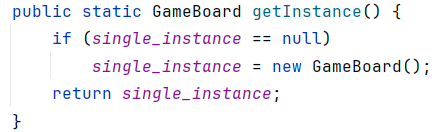
1. Ensure that the game responds promptly to user inputs, with no noticeable delay between the time a user makes a move and the corresponding action on the game board.
2. Minimize unnecessary memory consumption to ensure the game runs smoothly on systems with limited resources.
3. Validate user inputs effectively to prevent incorrect moves or unexpected behavior.
4. Implement robust error handling to gracefully handle unexpected situations or edge cases, ensuring the game remains stable.

**Main body**

**The Singleton Pattern** is a design pattern used to ensure that a class has only one instance throughout the entire lifecycle of a program, and provides a global point of access to that instance. This can be useful when you want to restrict the instantiation of a class to a single object, which is shared across different parts of a program.

****

GetInstance() Method provides the global point of access to the single instance of the GameBoard. It checks if an instance already exists, and if not, it creates one. It then returns the instance.

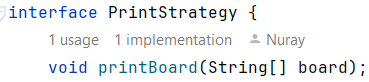


In the Main class, the GameBoard is obtained using GameBoard.getInstance(). This ensures that only one instance of the GameBoard is created and shared across the entire program.

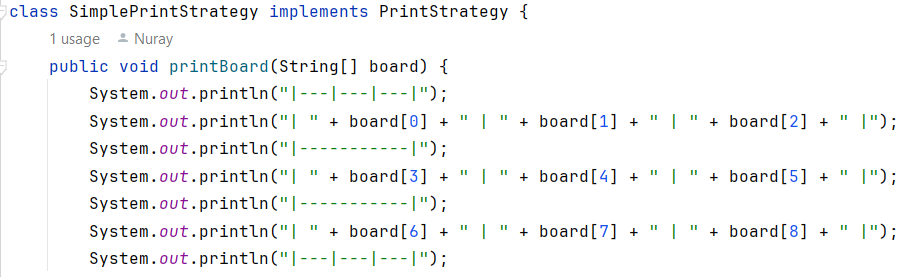


**The Strategy Pattern** is a design pattern that allows you to define a family of interchangeable algorithms, encapsulate each one, and make them interchangeable. This pattern enables the algorithm to vary independently from the context that uses it.

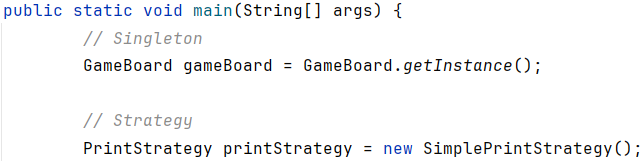
The PrintStrategy interface defines a contract for printing the game board. It acts as a blueprint for different strategies that can be used to print the board.



SimplePrintStrategy is a concrete implementation of the PrintStrategy interface. It provides a specific algorithm for printing the game board in a simple text format.

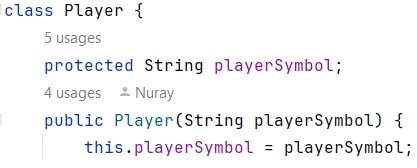


In the Main class, an instance of PrintStrategy is created and used to print the game board.

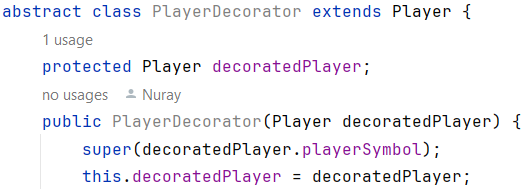


**The Decorator Pattern** is a structural design pattern that allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.

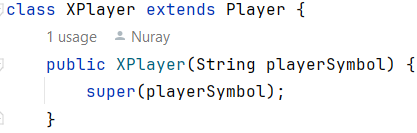
The Player class is an abstract class that serves as a base for different types of players in the game. It has a playerSymbol attribute to represent the symbol ('X' or 'O') associated with the player.



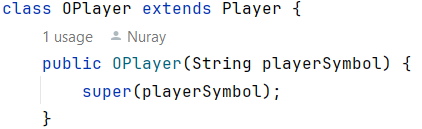
PlayerDecorator is an abstract class extending the Player class. It is intended to serve as a base class for potential player decorators.



XPlayer is a subclass of Player and represents a player with the symbol 'X'. It overrides the makeMove() method to always return "X".

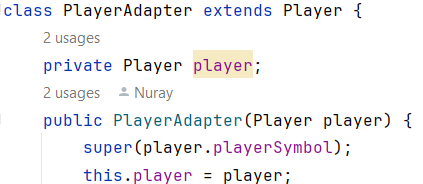


OPlayer is another subclass of Player and represents a player with the symbol 'O'. Like XPlayer, it overrides the makeMove() method to always return "O".

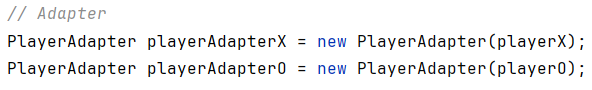


**The Adapter Pattern** is a structural design pattern that allows the interface of an existing class to be used as another interface, effectively allowing incompatible interfaces to work together.

PlayerAdapter is a class that adapts the existing Player objects to fit into the game logic. It extends the Player class and overrides the makeMove() method.



In the Main class, PlayerAdapter instances are created to adapt Player objects. This allows for a modified output format while using the existing player objects.

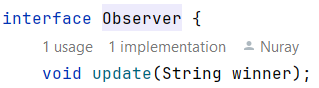


The playerAdapterX and playerAdapterO objects can now be used in place of regular Player objects.

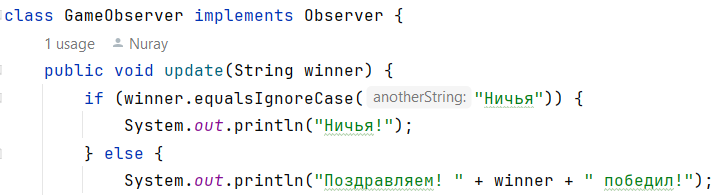


**The Observer Pattern** is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

The Observer interface defines a method update(String winner) which is used to notify observers of changes in the game state.

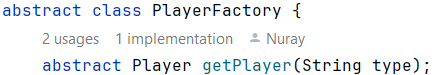


GameObserver is a concrete class that implements the Observer interface. It is responsible for observing the game and responding to changes in the game state.

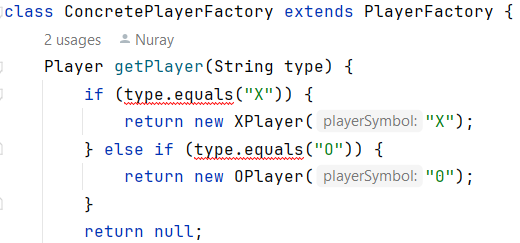


**The Factory Method Pattern** is a creational design pattern that provides an interface for creating instances of a class, but allows subclasses to alter the type of objects that will be created.

PlayerFactory is an abstract class that declares a method getPlayer(String type), which is intended to create instances of Player subclasses.

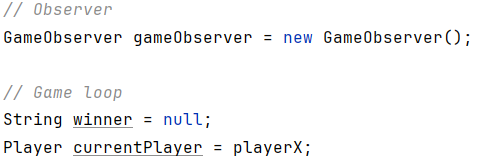


ConcretePlayerFactory is a concrete subclass of PlayerFactory. It overrides the getPlayer(String type) method to create specific types of players based on the input parameter type.

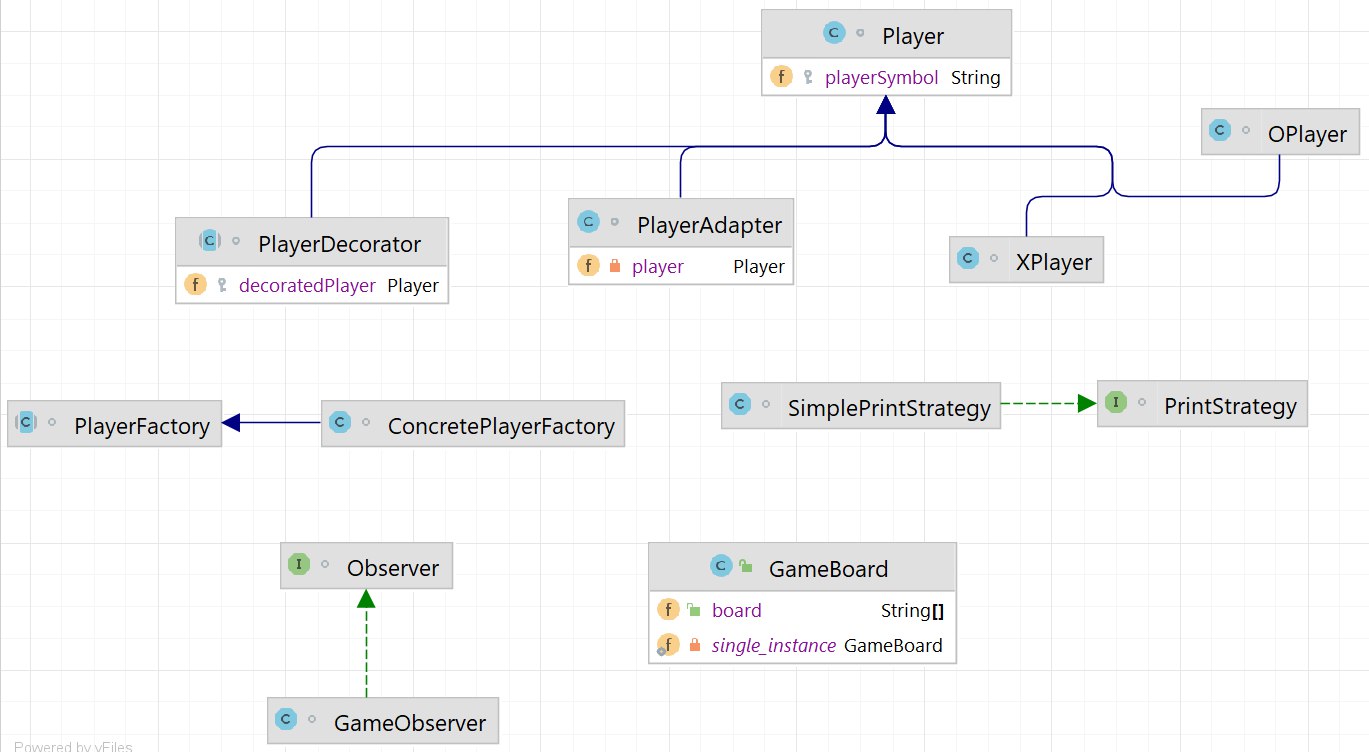


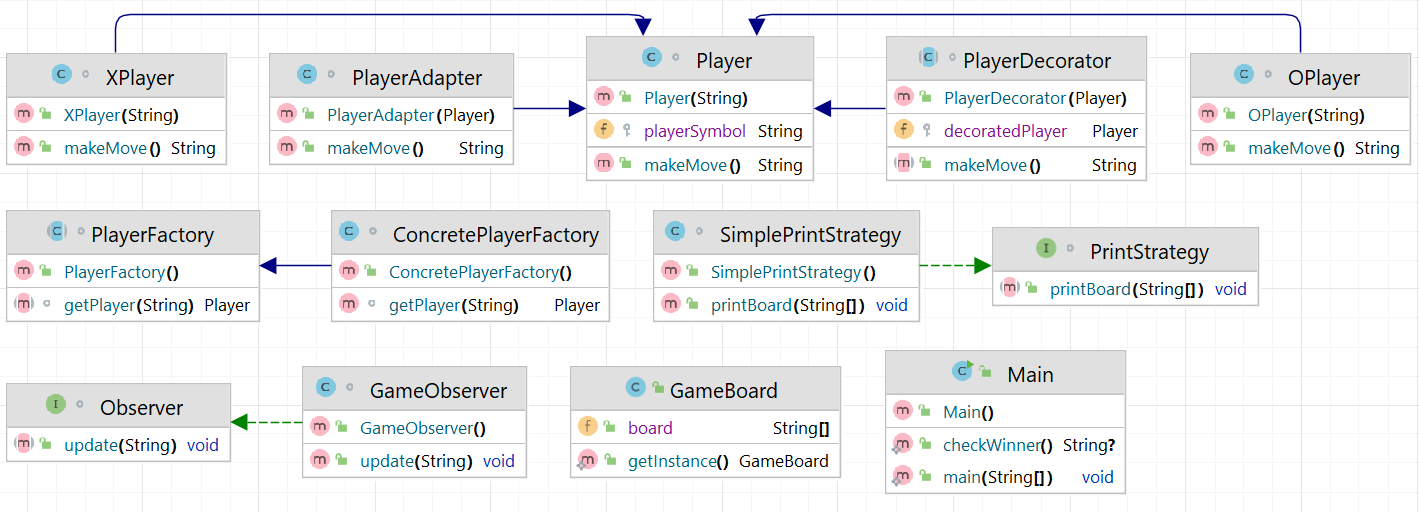
In the Main class, an instance of GameObserver is created to observe the game.

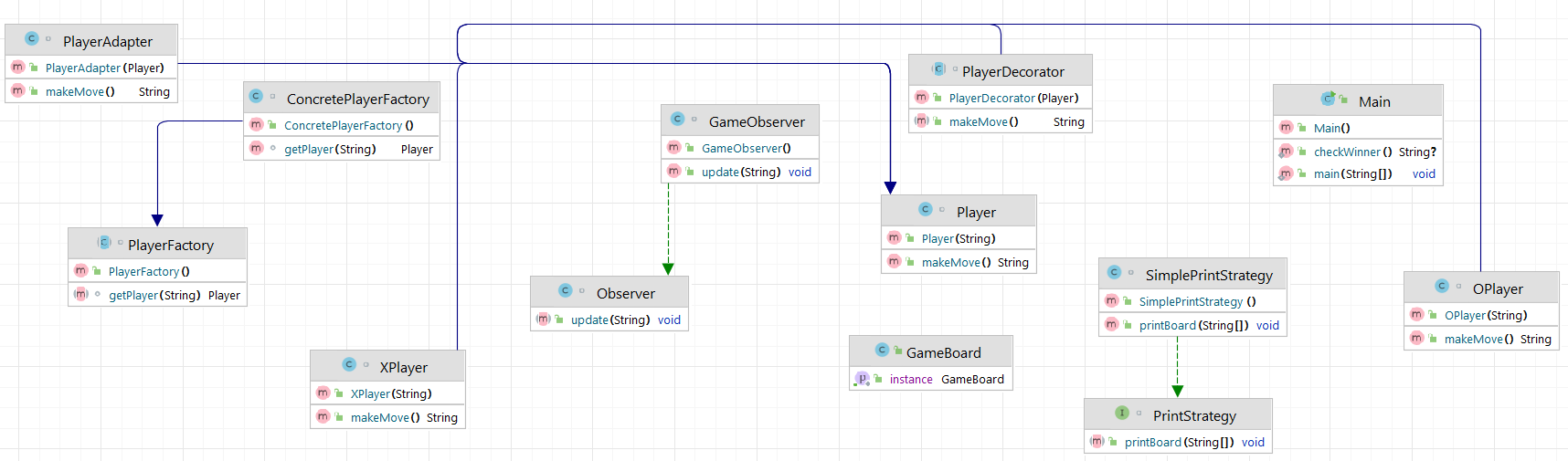
Later in the game loop, when a winner is determined or the game ends in a draw, the update method is called to notify the observer.

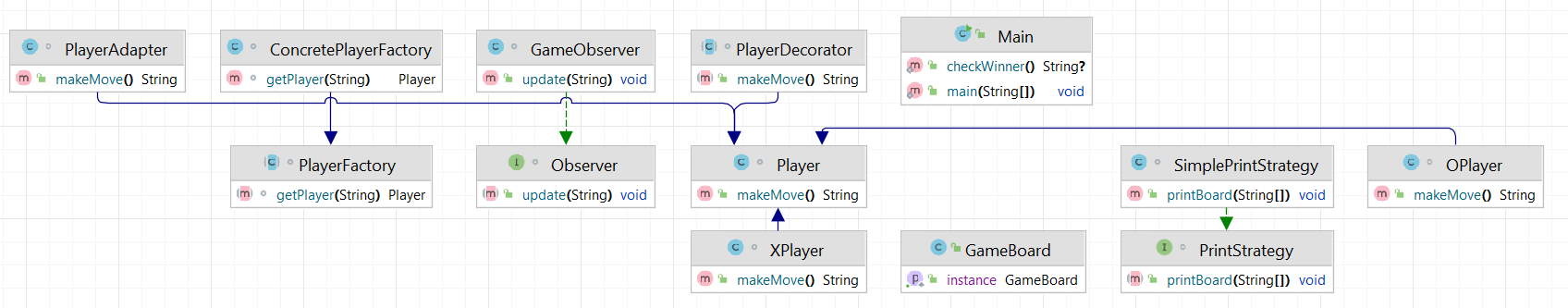


**UML Diagram**



****

****

****

**Conclusion**

In conclusion, this code exemplifies the application of various design patterns to enhance the modularity, flexibility, and maintainability of a simple Tic-Tac-Toe game. Each pattern addresses specific concerns, such as object creation, behavior modification, and notification handling, contributing to a well-structured and extensible codebase. By implementing these patterns, the code exhibits best practices in software design and serves as a solid foundation for potential future enhancements or variations of the game.

**Challenges Faced**

**Database Interaction:** If you decide to extend the game to include features like storing player statistics, game history, or user accounts, integrating a database can be challenging. This involves tasks like setting up database connections, writing SQL queries, and handling data retrieval and storage.

**Pattern Implementation:** Implementing design patterns correctly and effectively can be challenging, especially if you're not familiar with them. It requires a solid understanding of the patterns' concepts and how they can be applied in your specific context.

**Error Handling:** Proper error handling is crucial for robust software. It involves anticipating potential issues, defining clear error messages, and implementing mechanisms to gracefully handle exceptions or unexpected scenarios.

**Code Structure:** Organizing your code effectively is essential for maintainability and readability. It involves creating well-defined classes, methods, and modules, as well as establishing clear relationships between different components.