**21CS704**

**Design Patterns**

**2022-2023 Even Semester**

**Case Study Report**

**Name:** Shivang Modi

**Roll No.:** CB.EN.P2AIE22007

**Topic:** Tic-Tac-Toe Game

**Table of Contents**

[**1. Problem Statement 1**](#_9qqkextufe8e)

[1.1 Introduction 1](#_1qdyygi373r)

[1.2 Problem Description 1](#_c10daykjpbnb)

[**2. Design Patterns to Solve the Problem 3**](#_vyznzo9qad2l)

[2.1 MVC (Model-View-Controller) Pattern 3](#_elk3ibsu4eew)

[2.2 Factory Pattern 3](#_qzlsj2e6aayf)

[2.3 Strategy Pattern 3](#_gmy7b21liqh8)

[2.4 Observer Pattern 4](#_9kayyksyjwym)

[2.5 Singleton Pattern 4](#_hv8f77moriet)

[2.6 Command Pattern 4](#_c4vn2cvsa6xg)

[**3. UML Diagram 5**](#_iwpcfc57mi8a)

[**4. Conclusion 6**](#_nd2of77qbjwc)

# **1. Problem Statement**

# 

## **1.1 Introduction**

Tic-Tac-Toe is a classic two-player game played on a grid of 3x3 squares. The game's objective is to be the first player to form a horizontal, vertical, or diagonal line with three of their own marks, either “X” or “O”, on the grid. The game is played in turns, with each player taking their move one at a time.

Traditionally, the Tic-Tac-Toe game is played between two human players sitting face to face, but in modern implementations, the opponent can also be an AI player. This follows players to enjoy the game even when there is no human opponent available.

When playing against a human opponent, both players take turns entering their desired move by selecting an empty square on the grid. The game progresses until one player achieves a winning combination or all the squares on the grid are filled, resulting in a draw.

In the case of playing against an AI opponent, the AI is programmed to make strategic moves based on predefined rules or algorithms. The AI player analyzes the current state of the game, evaluates possible moves, and selects the best move to maximize its chances of winning or forcing a draw. The AI opponent aims to provide a challenging and engaging gameplay experience for the player.

The choice between playing against a human or an AI opponent depends on the player’s preference and the availability of other players. Playing against a human opponent adds an element of competition, social interaction, and strategy, the ability to practice and improve skills, and the opportunity to challenge oneself against different difficulty levels.

Whether playing against a human or an AI, Tic-Tac-Toe remains a fun and entertaining game that tests players’ logical thinking, strategic planning, and decision-making abilities. It is a perfect choice for quick gameplay sessions or as a recreational activity to enjoy with friends, family, or even on your own.

## **1.2 Problem Description**

You are tasked with developing a Tic-Tac-Toe game that allows the player to choose between playing against a human opponent or an AI opponent. The game should provide an interactive and engaging experience for the player, offering the flexibility to play against different opponents based on their preferences.

The game should be played on a 3x3 grid, where the player’s objective is to form a horizontal, vertical, or diagonal line with three of their own marks, either “X” or “O” before their opponent does. The player should be able to select their mark (either “X” or “O”) at the beginning of the game.

If the player chooses to play against a human opponent, the game should alternate turns between the two players. Each player should be promoted to enter their move by selecting an empty square on the grid. The game should validate the move and update the grid accordingly. The game should continue until one player achieves a winning combination or all the squares on the grid are filled, resulting in a draw. The outcome of the game (Win, Lose, or Draw) should be displayed at the end, and the players should have the option to restart the game.

If the player chooses to play against an AI opponent, the game should provide a challenging gameplay experience. The AI opponent should be capable of making strategic moves based on predefined rules or algorithms. It should analyze the current state of the game, evaluate possible moves, and select the best move to maximize its chances of winning or forcing a draw. The AI opponent’s moves should be displayed on the grid, and the game should continue until there is a winner or a draw. At the end of the game, the outcome should be displayed, and the players should have the option to restart the game.

The game should have a user-friendly interface that displays the current star of the grid, prompts the player for their moves, and provides feedback on the game's progress. It should handle invalid moves, such as selecting an occupied square or entering incorrect input, and provide appropriate error messages to the player.

Overall, the goal is to create an enjoyable and interactive Tic-Tac-Toe game that allows the player to choose their opponent, whether it be another human player or an AI player. The game should provide a seamless experience, allowing the player to have fun and test their skills in this classic game of strategy and logic.

# **2. Design Patterns to Solve the Problem**

# 

## **2.1 MVC (Model-View-Controller) Pattern**

The MVC pattern is used to separate the concerns of the game into three distinct components: Model, View, and Controller.

* **Model:** The TicTacToeModel class represents the game’s state and logic. It manages the game board, tracks the current player, checks for winning conditions, and handles moves.
* **View:** The TicTacToeView class is responsible for displaying the game’s state to the user. It provides methods to update and display the game board and messages to the console.
* **Controller:** The TicTacToeController class acts as an intermediary between the model and view. It handles user input, updates the model based on user actions, and updates the view to reflect the current game state.

The MVC pattern helps to separate the concerns of data management, user interface, and control logic, allowing for better code organization, modularity, and maintainability.

## **2.2 Factory Pattern**

The Factory pattern is used to create player objects based on the player type specified by the user (user or AI).

* The PlayerFactory class provides a factory method createPlayer() that creates the appropriate player object based on the player type specified. It encapsulates the object creation logic, allowing for easy extension and the addition of new player types in the future.

The factory pattern simplifies the object creation process, centralizes the creation logic, and decouples the client code from the actual object creation.

## **2.3 Strategy Pattern**

The Strategy pattern is used to implement different strategies for the user and AI opponent’s move selection.

* The MoveStrategy interface defines the common methods getNextMove() that different user and AI move strategies must implement.
* The UserMoveStrategy class implements the MoveStrategy interface and provides a strategy to select valid moves on the board for the user.
* The RandomMoveStrategy class implements the MoveStrategy interface and provides a strategy to select a random valid move on the board for the AI.

The Strategy pattern allows for interchangeable AI move strategies, making it easy to add new strategies in the future and providing flexibility in customizing the AI’s behavior.

## **2.4 Observer Pattern**

The Observer pattern is used to notify the view whenever there is a change in the model’s state.

* The Observer interface defines the update() method that the view implements. It allows the model to notify the view whenever a change occurs.
* The TicTacToeModel class maintains a list of observers and notifies them whenever there is a change in the game state.
* The TicTacToeView class implements the Observer interface and updates the view whenever notified by the model.

The Observer pattern establishes a one-to-many relationship between the model and the view, ensuring that the view is always up-to-date with the current game state.

## **2.5 Singleton Pattern**

The Singleton pattern is used to ensure that only one instance of the TicTacToeModel class exists throughout the game.

* The TicTacToeModel class uses a private constructor and a static method getInstance() to provide a single instance of the class.

The Singleton pattern guarantees that all components of the game access the same instance of the model, ensuring consistency and avoiding unnecessary object creation.

## **2.6 Command Pattern**

The Command pattern is used to encapsulate the game actions as objects, allowing for their execution and undoing.

* The Command interface defines the execute() method that encapsulates a specific action.
* The MakeMoveCommand and RestartGameCommand classes implement the Command interface and encapsulate the actions of making a move and restarting the game, respectively.
* The TicTacToeController class uses commands to execute game actions based on the user input.

The Command pattern provides a flexible and extensible way to encapsulate actions as objects, enabling the controller to handle user commands dynamically and facilitating easy undo or redo functionality if needed.

# **3. UML Diagram**

# 

## 

# **4. Conclusion**

# 

In conclusion, we have successfully developed a Tic-Tac-Toe game that offers the flexibility to play against either a human opponent or an AI opponent. The game incorporates several design patterns to enhance its structure, modularity, and user experience.

The **Model-View-Controller (MVC)** pattern is implemented to separate the game's components into three distinct entities. The model handles the game's logic, the view displays the game state to the user, and the controller acts as an intermediary between the two, managing user input and updating the model and view accordingly. This separation of concerns promotes code organization and maintainability.

The **Factory pattern** is utilized to create player objects based on the player type chosen by the user. The PlayerFactory encapsulates the object creation logic, allowing for easy extension and the addition of new player types in the future. This pattern enhances code flexibility and extensibility.

The **Strategy pattern** is employed to implement different AI move strategies. The MoveStrategy interface defines a common method, getNextMove(), which various AI strategies can implement. In the provided example, the RandomMoveStrategy randomly selects an available move. This pattern allows for interchangeable AI strategies and easy customization of the AI's behavior.

The **Observer pattern** is utilized to notify the viewer whenever there is a change in the model's state. The model maintains a list of observers and notifies them of any updates. This pattern ensures that the view is always up-to-date with the current game state and facilitates a dynamic and responsive user interface.

The **Singleton pattern** ensures that only one instance of the TicTacToeModel class exists throughout the game. This guarantees consistency and avoids unnecessary object creation, promoting efficient memory usage and ensuring that all components of the game access the same instance of the model.

The **Command pattern** is employed to encapsulate game actions as objects, allowing for their execution and potential undoing. The commands are executed based on user input, providing flexibility and enabling the controller to handle user commands dynamically.

In summary, the implemented design patterns contribute to a well-structured, maintainable, and extensible Tic-Tac-Toe game. The game provides a user-friendly interface, allowing players to choose between playing against a human or an AI opponent. Players can enjoy the strategic challenge of playing against an intelligent AI or engage in a competitive match against a friend. With its modular design and use of various design patterns, the game can be easily expanded with new features, additional player types, or enhanced AI strategies.