Name: Nguyen Duc Tam

Student ID: 104311214

**Design Report**

**Project Overview**

This Tetris game project demonstrates effective object-oriented design and implementation. The game includes various features such as Tetromino spawning, grid management, score calculation, level progression, and sound effects. The design showcases the use of multiple classes, interfaces, and enumerations to encapsulate different aspects of the game logic.

**How the Game Works**

The Tetris game begins with a start menu, where the player can press a key to begin a new game. Once the game starts, Tetromino pieces start falling from the top of the grid. The player can move these pieces left, right, or down, and can also rotate them to fit them into empty spaces in the grid. The objective is to complete horizontal lines with no gaps. When a line is completely filled, it clears from the grid, and the player earns points. The more lines cleared at once, the higher the score. As the player clears more lines, the game speed increases, making the Tetrominos fall faster and increasing the challenge. The game continues until the Tetromino pieces stack up to the top of the grid, at which point the game is over. Background music plays during gameplay, and sound effects provide audio feedback for actions like clearing lines, rotating pieces, and game over events. The player can then choose to restart the game from the game over screen.

**List of Classes**

1. **GameManager**
2. **TetrominoManager**
3. **GridManager**
4. **ScoreManager**
5. **InputManager**
6. **DrawManager**
7. **Tetromino**
8. **TetrominoFactory**
9. **TetrominoSpawner**
10. **IRotationStrategy**
11. **DefaultRotationStrategy**
12. **Enum**

**Main Methods and Design Features**

**GameManager**

* **RunGame()**: The main game loop, responsible for processing events, updating game state, and rendering the game.
* **StartNewGame()**: Initializes the game state, resetting scores, grid, Tetrominos, and timers.
* **UpdateLevelAndSpeed(bool forceLevelUp)**: Updates the game level and falling speed based on lines cleared or forced level up.

**TetrominoManager**

* **TetrominoManager()**: Constructor that initializes the current and next Tetromino.
* **SpawnNextTetromino(int[,] grid)**: Spawns the next Tetromino and checks if it can be placed on the grid.
* **Reset()**: Reinitializes both current and next Tetrominos for a new game.

**GridManager**

* **PlaceTetromino(Tetromino tetromino)**: Places the current Tetromino on the grid.
* **ClearCompleteLines()**: Clears complete lines from the grid and returns the number of lines cleared.
* **Reset()**: Resets the grid for a new game.

**ScoreManager**

* **UpdateScore(int linesCleared)**: Updates the score based on the number of lines cleared.
* **Reset()**: Resets the score for a new game.
* **SaveHighScore()**: Saves the high score to persistent storage.

**InputManager**

* **HandleInput(Tetromino currentTetromino, int[,] grid)**: Processes user inputs to move or rotate the current Tetromino.
* **ExitRequested**: Property that checks if the exit command has been requested.
* **DebugLevelUpRequested**: Property that checks if a debug level up has been requested.

**DrawManager**

* **Draw(Tetromino currentTetromino, Tetromino nextTetromino, int[,] grid, Dictionary<int, Color> colors, int score, int highScore, int level, int linesCleared)**: Draws the current state of the game.
* **DrawStartMenu()**: Draws the start menu screen.
* **DrawGameOver(int score, int highScore)**: Draws the game over screen.

**Tetromino**

* **Rotate(int[,] grid)**: Rotates the Tetromino if the rotation is valid.
* **CanMoveLeft(int[,] grid)**: Checks if the Tetromino can move left.
* **CanMoveRight(int[,] grid)**: Checks if the Tetromino can move right.
* **CanMoveDown(int[,] grid)**: Checks if the Tetromino can move down.
* **MoveLeft(int[,] grid)**: Moves the Tetromino left.
* **MoveRight(int[,] grid)**: Moves the Tetromino right.
* **MoveDown(int[,] grid)**: Moves the Tetromino down.
* **CanBePlaced(int[,] grid)**: Checks if the Tetromino can be placed at its current position.

**TetrominoFactory**

* **CreateTetromino(TetrominoType type)**: Creates a new Tetromino based on the given type.

**TetrominoSpawner**

* **GetNextTetromino()**: Returns the next Tetromino to be spawned.

**IRotationStrategy**

* **Rotate (Tetromino tetromino)**: Interface method to rotate the Tetromino according to its strategy.

**DefaultRotationStrategy**

* **Rotate (Tetromino tetromino)**: Rotates the Tetromino blocks 90 degrees and updates the rotation state.

**Design Features**

* **Object-Oriented Design**: The project employs a clear separation of concerns, with each class handling specific aspects of the game logic.
  + **Abstraction**: Each class represents a clear abstraction (e.g., TetrominoManager handles Tetromino-related operations, GridManager handles grid operations).
  + **Encapsulation**: Data and methods are encapsulated within classes, providing a modular and maintainable codebase.
  + **Inheritance and Polymorphism**: The use of interfaces (e.g., IRotationStrategy) allows for flexible and interchangeable behavior.
  + **Cohesion and Coupling**: Each class is conceptually coherent with good cohesion and appropriate coupling. Methods are small, targeted, and perform meaningful actions without unnecessary code duplication.
* **State Management**: The GameManager class manages the overall game state, including transitions between start menu, running game, and game over states.
* **Real-time Input Handling**: The InputManager class processes user inputs in real-time, enabling responsive gameplay.
* **Sound Management**: The game includes background music and sound effects, managed centrally to enhance the gaming experience.

**Design Patterns**

 **Factory Pattern**: The TetrominoFactory demonstrates the factory design pattern, creating objects based on type.

 **Singleton Pattern**: The GameManager employs the singleton pattern to ensure only one instance of the game manager exists.

 **Strategy Pattern**: The use of IRotationStrategy and DefaultRotationStrategy demonstrates the strategy design pattern, allowing different rotation strategies for Tetrominos.

**Class Details**

**GameManager Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Main game loop | Method | RunGame() handles the main game loop and state transitions. |
| Game state management | Method | StartNewGame() and UpdateLevelAndSpeed(bool forceLevelUp) manage game initialization and level progression. |

**TetrominoManager Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Tetromino management | Constructor, Method | Initializes and manages the current and next Tetromino with TetrominoManager(), SpawnNextTetromino(int[,] grid), and Reset(). |

**GridManager Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Grid operations | Method | PlaceTetromino(Tetromino tetromino), ClearCompleteLines(), and Reset() handle grid-related operations. |

**ScoreManager Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Score management | Method | UpdateScore(int linesCleared), Reset(), and SaveHighScore() manage scoring and high score saving. |

**InputManager Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Input handling | Method, Property | HandleInput(Tetromino currentTetromino, int[,] grid), ExitRequested, and DebugLevelUpRequested handle user inputs. |

**DrawManager Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Drawing operations | Method | Draw(...), DrawStartMenu(), and DrawGameOver(...) handle rendering of different game states. |

**Tetromino Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Tetromino behavior | Method | Rotate(int[,] grid), CanMoveLeft(int[,] grid), CanMoveRight(int[,] grid), CanMoveDown(int[,] grid), MoveLeft(int[,] grid), MoveRight(int[,] grid), MoveDown(int[,] grid), and CanBePlaced(int[,] grid) handle Tetromino actions. |

**TetrominoFactory Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Tetromino creation | Method | CreateTetromino(TetrominoType type) creates Tetrominos based on type. |

**TetrominoSpawner Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Tetromino spawning | Method | GetNextTetromino() returns the next Tetromino to be spawned. |

**IRotationStrategy Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Rotation strategy | Method | Rotate(Tetromino tetromino) defines the interface method for rotating a Tetromino. |

**DefaultRotationStrategy Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Default rotation logic | Method | Rotate(Tetromino tetromino) rotates Tetromino blocks 90 degrees. |

**Enum Details**

|  |  |  |
| --- | --- | --- |
| **Responsibility** | **Type Details** | **Notes** |
| Various game states and types | Enumeration | Used to define game states |

**Conclusion**

I believe my Tetris game project successfully demonstrates the ability to design and implement a program of reasonable complexity with effective use of object-oriented design concepts. The classes are well-abstracted, methods are focused and meaningful, and the data and objects are managed intelligently to create a cohesive and well-structured application.