VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AN ENGINEERING

Object-Oriented Programming

ITIU069

Instructor: Tran Thanh Tung

FINAL REPORT

TETRIS

By Group 03 hay 4 – Member List

Lê Huỳnh Thiện	ITCSIU23036 (Team Leader)
Lý Gia Bảo	ITDSIU23034 (Team Member)
Đoàn Xuân Cao	ITITIU23004 (Team Member)

Table Of Contents

Chapter 1: INTRODUCTION

1. ABSTRACT

This project develops **Tetris Game**, which is a classic and widely recognized puzzle video game. The game revolves around the placement of geometric shapes, known as **tetrominoes**, onto a grid-based playfield. These tetrominoes fall from the top of the playfield, and players must manipulate their position and orientation to create horizontal lines without gaps. Completed lines disappear, earning the player points and providing more space to continue playing.

Moreover, the challenge increases as the game progresses, with tetrominoes falling at faster speeds, requiring quick thinking, spatial awareness, and precision. The game ends when the stacked blocks reach the top of the playfield. Tetris is renowned for its simplicity, accessibility, and addictive gameplay, making it one of the most popular and enduring video games of all time.

2. CONTRIBUTION AND TASK ALLOCATION:

A. CONTRIBUTION

Name	Student ID	Contribution
Lê Huỳnh Thiện	ITCSIU23036	35%
Đoàn Xuân Cao	ITITIU23004	32%
Lý Gia Bảo	ITDSIU23034	33%

B. **TASK ALLOCATION**

STAGE	ACTION	MEMBER	WEEK
PLANNING AND PREPARATION	Researching about the project topic, analyzing the project requirements	Thiện	**1-2**
	Determining research scopes and the objective of the project	Thiện	
	Constructing a timeline for the project	Thiện	
	Determining the tools for the project	Bảo, Thiện	
	Write Project Report	**All member**	
DESIGNING UML	Analyzing the classes, objects	Thiện	**3-4**
	Design UML	Cao, Thiện	

CREATE APPLICATION INTERFACE	Developing application interface	Thiện	**6**
	Developing GameBoard	Thiện	
	Developing Minos	Bảo, Thiện	
DEVELOPING FUNCTIONS FOR THE APPLICATION	Finding the sound games	Cao	**7-9**
	Developing sound games	Bảo	
	Developing scoring system, keyHandler	Bảo, Thiện	
TESTING AND COMPLETING THE REPORT	Fixing bugs on Java	Thiện, Bảo	**10**
	Testing functions of application and optimizing the application performance	**All member**	
	Writing Final Report	**All member**	
	Making slides for the presentation	**All member**	

	Rehearsing the presentation and reviewing the overall project	**All member**	
PRESENTATION		**All member**	**11**

3. THE TECHNIQUES AND TOOLS

A. Framework technologies:

- **Java Swing:** provides a powerful framework for creating graphical user interfaces (GUI), making it an ideal choice for developing games like Tetris. Using Swing, the game board can be implemented as a **`JPanel`**, where the **`paintComponent`** method dynamically renders tetrominoes and updates the grid. A **`javax.swing.Timer`** manages the game loop, controlling the movement of falling pieces and checking for collisions. Player input, such as moving or rotating blocks, can be handled with a **`KeyListener`**, ensuring smooth interaction. With its lightweight components and robust event-handling, Swing enables the creation of an engaging and visually appealing Tetris game.

B. Develop Environment:

- **IntelliJ IDEA and Visual Studio Code:** an integrated development environment, to streamline the development process by providing code

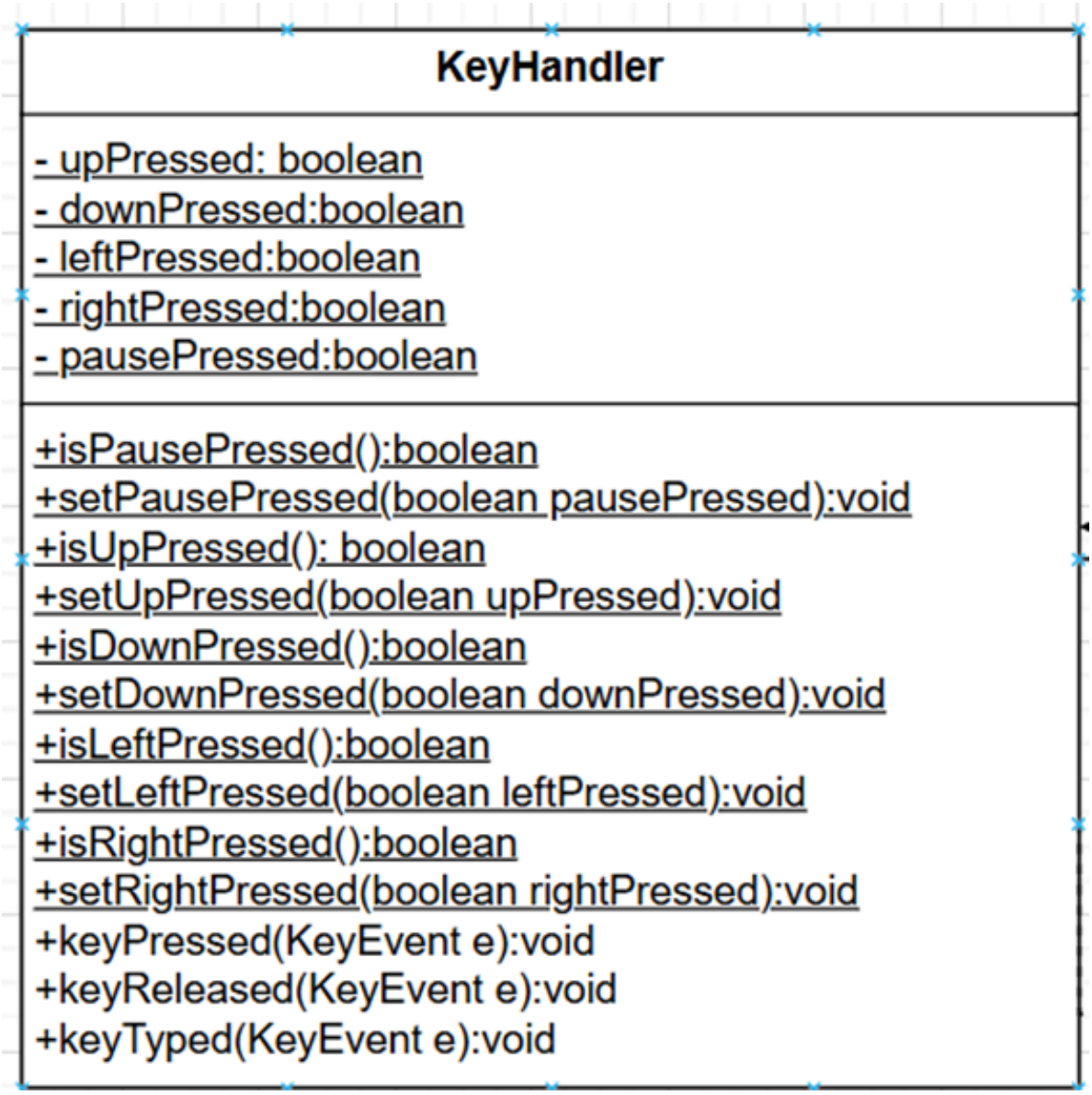
editing, build automation, and debugging functionalities.

- **GITHUB:** Multiple developers can work on a project simultaneously, using features like pull requests, branches, and code reviews to coordinate and ensure code quality.

Chapter 2: CLASS DIAGRAM AND GAME RULES

2.1. Class Diagram:

2.1.1 KeyHandler Class:



Role:

- The KeyHandler class serves as the system's interface for handling user input. It monitors key presses and releases, updating the state of relevant attributes that dictate gameplay actions.
- The class ensures encapsulated access to input states and translates user actions into game events.

Attributes:

1. **private static boolean upPressed**:
    - **Description**: Tracks whether the "up" key is currently pressed.
    - **Encapsulation**: Marked private to restrict direct access; static to ensure a single state is shared across the system.
2. **private static boolean downPressed**:
    - **Description**: Tracks whether the "down" key is pressed, typically used for accelerating the fall of a Tetris piece.
    - **Encapsulation**: Ensures controlled access through setter and getter methods.
3. **private static boolean leftPressed**:
    - **Description**: Indicates whether the "left" key is pressed, which moves the Tetris piece leftward.
4. **private static boolean rightPressed**:

- **Description**: Indicates whether the "right" key is pressed, which moves the Tetris piece rightward.

5. **private static boolean pausePressed**:
    - **Description**: Tracks the state of the "pause" key, allowing the game to be paused.

Methods:

1. **Setter Methods**:
    - **Examples**:
        - public static void setUpPressed(boolean upPressed)
        - public static void setDownPressed(boolean downPressed)
        - Similar methods for leftPressed, rightPressed,and pausePressed,
        - **Description**: Update the state of the corresponding key.
    - **Encapsulation**: Encapsulates the modification of key states, ensuring that changes are made only through controlled methods.
2. **Getter Methods**:
    - **Examples**:
        - public static boolean isUpPressed()
        - public static boolean isDownPressed()
        - Similar methods for leftPressed, rightPressed, and pausePressed
        - **Description**: Return the current state of the corresponding key.
    - **Encapsulation**: Restricts access to key states by exposing read-only functionality through getters.

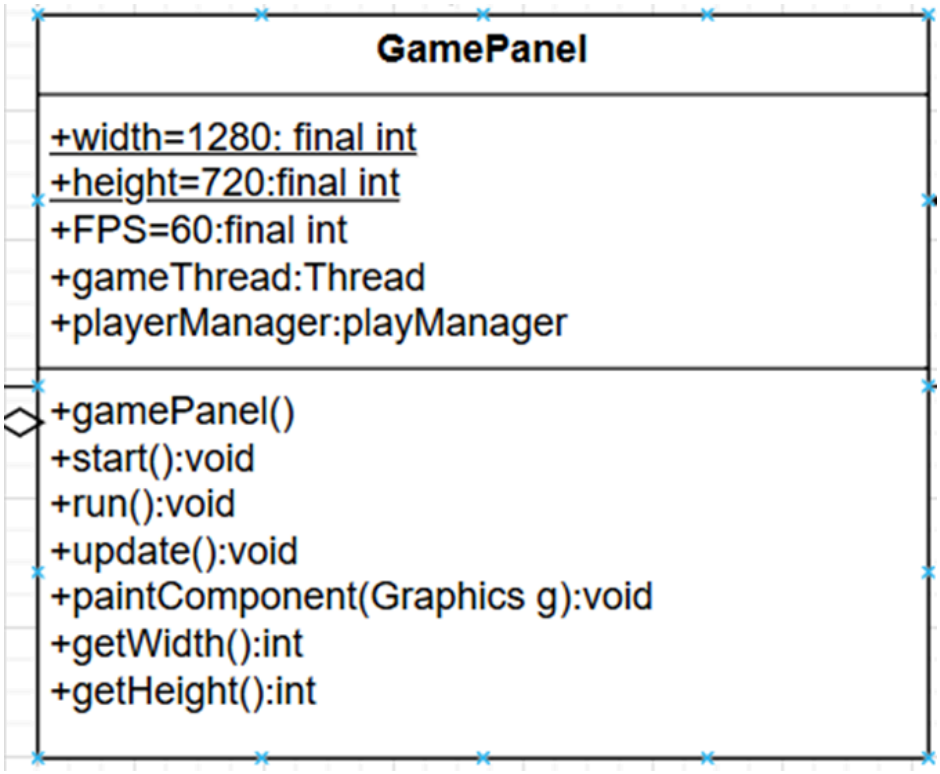
3. **public void keyPressed(KeyEvent e)**:
    - **Description**: Invoked when a key is pressed. Identifies the key and updates the corresponding state attribute.
    - **Responsibilities**:
        - Maps key codes to logical actions (e.g., arrow keys, spacebar).
    - **Abstraction**: Simplifies the process of handling key press events.
4. **public void keyReleased(KeyEvent e)**:
    - **Description**: Invoked when a key is released. Updates the corresponding key state attribute to false.
5. **public void keyTyped(KeyEvent e)**:
    - **Description**: Not used in this implementation, as the game primarily relies on key press and release events.

Design Principle Of KeyHandler:

- **Encapsulation**:
    - All key state attributes are private, ensuring controlled access and modification through setters and getters.
- **SRP**:
    - Focuses exclusively on user input management, separating it from gameplay logic.
- **Abstraction**:

- Hides the complexity of key event handling, providing a clean interface for accessing key states.

2.1.2 GamePanel Class:



Role:

The GamePanel class serves as the core component of the graphical user interface (GUI). It is responsible for managing the game loop, rendering game components, and interacting with the gameplay logic encapsulated in the PlayManager class. This class integrates other system components, acting as the main entry point for the visual and interactive aspects of the game.

Attributes:

1. **public static final int width:**
    - **Description**: Represents the fixed width of the game panel, set to 1280 pixels. This value is constant and shared across the entire system**.**
    - **Encapsulation:** Marked as final to ensure immutability, adhering to good design practices by preventing accidental modification**.**
    - **Usage**: Used in rendering operations and collision boundary checks within the game**.**
2. **public static final int height:**
    - **Description:** Represents the fixed height of the game panel, set to 720 pixels. Like width, it is immutable and ensures consistency throughout the game logic**.**
    - **Encapsulation:** Restricts direct access by marking it as public static final, ensuring encapsulated configuration**.**
3. **final int FPS:**
    - **Description:** Frames per second, set to 60, which determines the speed at which the game loop executes**.**
    - **Encapsulation:** Declared final to ensure that the frame rate remains constant, preventing fluctuations that might disrupt the game's performance**.**
4. **public static Sound music:**
    - **Description:** A static reference to the Sound class that handles the playback of background music and sound effects**.**

- **Encapsulation:** Although public, the Sound class itself encapsulates the implementation details of audio playback, abstracting the complexities from the GamePanel.

5. **Thread gameThread:**
    - **Description:** A thread responsible for running the game loop. It separates game logic from the UI thread, ensuring smooth gameplay and preventing performance bottlenecks.
    - **Usage:** The thread executes the run() method, which updates and renders the game components**.**
6. **PlayManager playManager:**
    - **Description:** An instance of the PlayManager class, which centralizes gameplay logic, including the movement of Tetris pieces, collision detection, and score tracking**.**
    - **Encapsulation**: Delegates gameplay logic to a dedicated class, adhering to the Single Responsibility Principle (SRP**).**

Methods:

1. **public GamePanel():**
    - **Description:** Constructor that initializes the game panel and its dependencies.
    - **Responsibilities:**
        - Instantiates the PlayManager.
        - Configures the Sound instance for background music.

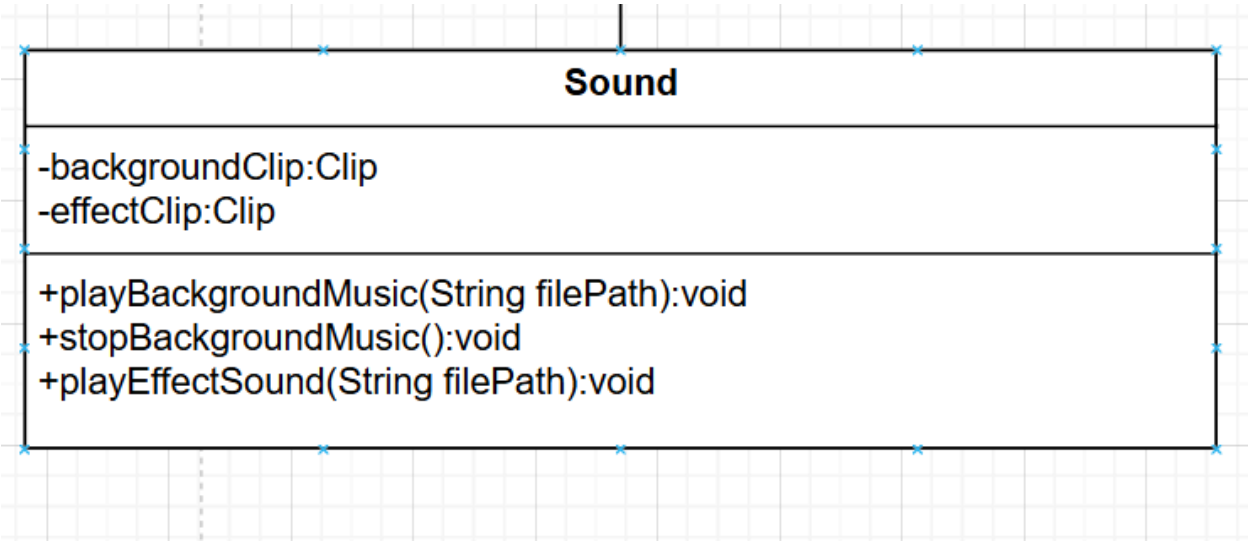
        - Prepares the game thread.
    - **Encapsulation:** Hides the complexity of initialization from the rest of the application.
2. **public void start():**
    - **Description:** Starts the game by initializing the gameThread and playing the background music**.**
    - **Responsibilities:**
        - Ensures the game loop begins execution.
        - Delegates audio playback to the Sound class.
    - **Abstraction:** Abstracts the game thread and music playback mechanics.
3. **public void run():**
    - **Description:** Implements the core game loop.
    - **Responsibilities:**
        - Calls update() to handle game state changes.
        - Calls repaint() to trigger graphical rendering.
    - **Polymorphism:** Interacts with various game objects (Block, Mino) using their abstract or overridden methods.
4. **public void update():**
    - **Description:** Updates the game state by delegating to the PlayManager.
    - **Responsibilities:**
        - Ensures game objects are updated at each frame.
    - **Encapsulation:** Encapsulates gameplay logic within PlayManager.

5. **protected void paintComponent(Graphics g):**
    - **Description:** Handles rendering of all game elements.
    - **Responsibilities:**
        - Invokes draw() methods of individual game components.
        - Ensures consistent rendering by managing graphical context (Graphics g).
    - **Polymorphism:** Leverages overridden draw() methods in Block and Mino**.**
6. **@Override public int getWidth() and @Override public int getHeight():**
    - **Description:** Return the panel's dimensions, adhering to the contract defined by the parent class.

Design Principles in GamePanel

1. **Encapsulation**:
    - Critical attributes (width, height, gameThread) are either final or delegated to other classes to limit direct access and modification.
2. **SRP**:
    - Handles only GUI-related tasks, delegating gameplay logic to PlayManager and music playback to Sound.
3. **Abstraction**:
    - Hides implementation details of game logic (PlayManager) and music (Sound).
4. **Polymorphism**:
    - Interacts with game objects (Block, Mino) through abstract methods without knowing their specific implementations.

2.1.3 Sound Class:



Role:

The Sound class centralizes audio management for the game, including background music and sound effects. By encapsulating audio playback logic, it simplifies the integration of sound into the game.

Attributes:

Attributes:

1. **private Clip backgroundClip**:
    - **Description**: Represents the audio clip used for looping background music.
    - **Encapsulation**: Marked private to restrict direct access, ensuring controlled playback.
2. **private Clip effectClip**:

- **Description**: Represents the audio clip used for single-use sound effects.
- **Encapsulation**: Similar to backgroundClip, encapsulates sound effect logic.

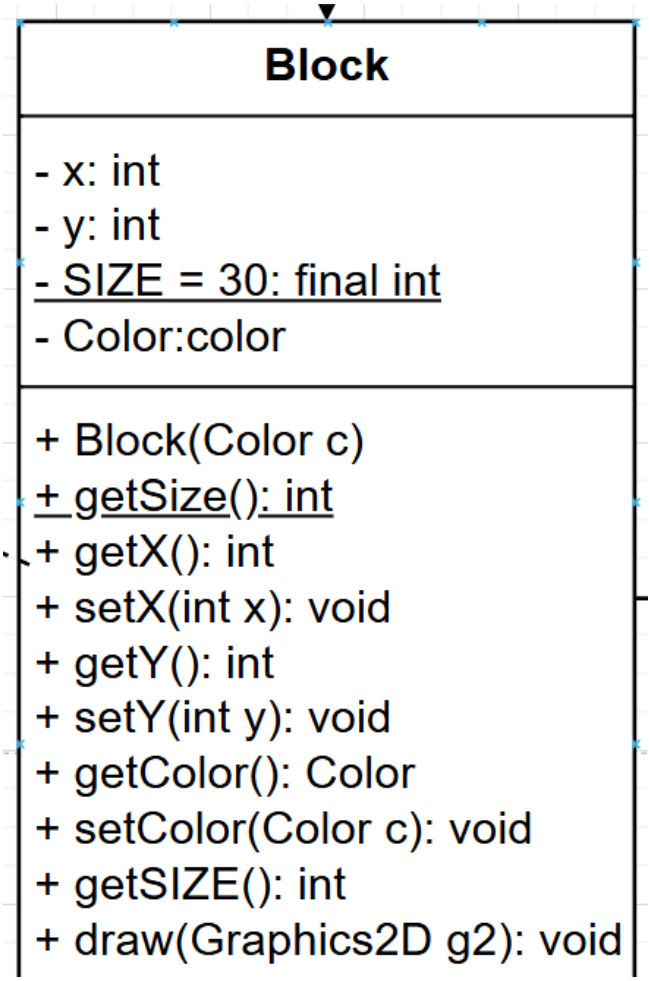
Methods:

1. **public void playBackgroundMusic(String filePath)**:
   - **Description**: Plays looping background music.
   - **Responsibilities**:
     - Loads the audio file from the specified filePath.
     - Starts playback in a loop.
   - **Abstraction**: Hides the complexity of audio file handling.
2. **public void stopBackgroundMusic()**:
   - **Description**: Stops the background music.
   - **Responsibilities**:
     - Ensures proper closure of the audio stream.
   - **Encapsulation**: Prevents external control of audio state.
3. **public void playEffectSound(String filePath)**:
   - **Description**: Plays a single-use sound effect.
   - **Responsibilities**:
     - Loads and plays an audio file from the specified filePath.
   - **Abstraction**: Simplifies the playback of sound effects.

Design Principles in Sound:

- **Encapsulation**:
    - Audio playback logic is encapsulated within the class, exposing only public methods for controlled usage.
- **SRP**:
    - Dedicated solely to audio management, separating it from gameplay logic.
- **Abstraction**:
    - Hides low-level details of working with audio files and streams.

2.1.4 Block Class:



Role:

The Block class represents the fundamental building block of all Tetris pieces. Each Tetris piece (Mino) is composed of multiple Block instances. The class encapsulates properties such as position, size, and color, and provides methods for rendering and updating blocks.

Attributes:

1. **private int x**:
    - **Description**: Represents the x-coordinate of the block on the game grid.
    - **Encapsulation**: Marked private to restrict direct access, with getter and setter methods for controlled manipulation.
2. **private int y**:
    - **Description**: Represents the y-coordinate of the block on the game grid.
    - **Encapsulation**: Similar to x, encapsulated to ensure data integrity.
3. **private static final int SIZE**:
    - **Description**: The fixed size of a block, set to 30 pixels.
    - **Encapsulation**: Marked final to prevent changes, ensuring uniformity across all blocks.
4. **private Color color**:
    - **Description**: Specifies the color of the block, used for rendering.
    - **Encapsulation**: Controlled access through getter and setter methods.

Methods:

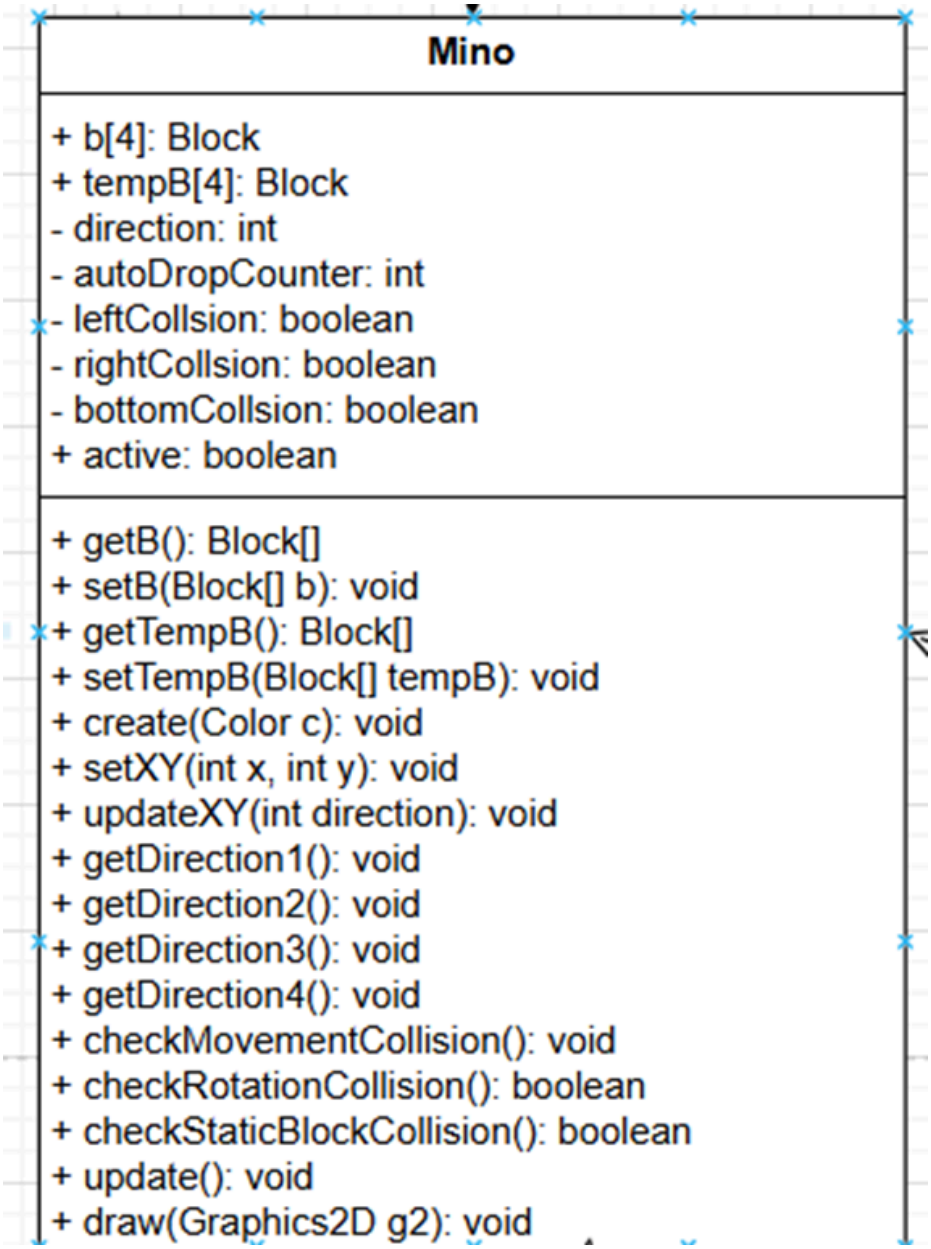
1. **public Block(Color c)**:
    - **Description**: Constructor that initializes the block with a specific color.
    - **Responsibilities**:
        - Assigns the color attribute.
        - Initializes the position to default values.

2. **Getter and Setter Methods**:
    - Examples:
        - public int getX() / public void setX(int x)
        - public int getY() / public void setY(int y)
        - public Color getColor() / public void setColor(Color color)
    - **Encapsulation**: Provides controlled access to block attributes.
3. **public static int getSize()**:
    - **Description**: Returns the fixed size of the block.
4. **public void draw(Graphics2D g2)**:
    - **Description**: Renders the block on the game panel using the specified Graphics2D context.
    - **Polymorphism**: Can be overridden or extended if needed for specialized block types.

Design Principles in Block:

- **Encapsulation**:
    - Attributes are private, with controlled access via getters and setters.
- **SRP**:
    - Manages the properties and rendering of a single block.
- **Abstraction**:
    - Hides rendering details behind the draw() method.

2.1.5 Mino Class:



Role:

The Mino class represents a generic Tetris piece composed of multiple Block instances. It serves as the base class for all specific Tetris piece shapes (e.g., Mino_Bar, Mino_L1).

Attributes:

1. **protected Block[] b**
    - **Description**: Represents the blocks that form the Tetris piece.
    - **Purpose**: Tracks the position and layout of the piece on the grid.
    - **Encapsulation**: Protected to allow access in subclasses while restricting external access.
2. **protected Block[] tempB**
    - **Description**: Temporarily stores the positions of the blocks during rotation calculations.
    - **Purpose**: Allows rotation logic to validate changes without directly modifying the main blocks.
3. **protected int direction**
    - **Description**: Tracks the current rotation state of the Tetris piece (values: 0–3).
    - **Purpose**: Indicates the orientation of the piece, allowing rotation logic to determine the next state.
4. **protected boolean collisionLeft**
    - **Description**: Tracks whether the piece collides with the left wall of the grid.
    - **Purpose**: Ensures the piece does not move outside the grid's boundaries.

5. **protected boolean collisionRight**
    - **Description**: Tracks whether the piece collides with the right wall of the grid.
    - **Purpose**: Restricts the piece from exceeding the grid's right boundary.
6. **protected boolean collisionBottom**
    - **Description**: Tracks whether the piece collides with the bottom boundary or static blocks.
    - **Purpose**: Stops the piece when it lands on the bottom or other blocks.
7. **protected boolean active**
    - **Description**: Indicates whether the piece is actively controlled or has become static.
    - **Purpose**: Differentiates between moving pieces and landed pieces.

Methods:

1. **public void create(Color c)**
    - **Description:** Initializes the Tetris piece with the specified color.
    - **Responsibilities:**
        - Assign colors to all blocks in the piece.
        - Ensure visual uniformity of the piece during gameplay.
2. **public void setXY(int x, int y)**
    - **Description:** Sets the initial position of the piece on the game grid.
    - **Parameters:**

        - x: The x-coordinate of the piece.
        - **y:** The y-coordinate of the piece.
    - **Responsibilities:**
        - Align the piece within valid boundaries when it spawns.
3. **public void updateXY(int direction)**
    - **Description:** Updates the positions of the blocks in the piece based on the new rotation direction.
    - **Parameters:**
        - **direction:** The new rotation state (0–3).
    - **Responsibilities:**
        - Rotate the piece while maintaining its layout integrity.
4. **public boolean checkMovementCollision()**
    - **Description:** Detects collisions with grid boundaries or static blocks during movement.
    - **Responsibilities:**
        - Prevent invalid movements to occupied or out-of-bound areas**.**
5. **public boolean checkRotationCollision()**
    - **Description:** Validates whether the piece can rotate without collisions or out-of-bound errors**.**
    - **Responsibilities:**
        - Ensure rotation only occurs if the new state is valid.
6. **public void update()**

- **Description:** Updates the position and state of the piece based on user input or game logic**.**
- **Responsibilities:**
  - Handle movement and rotation logic.
  - Detect and respond to collisions during gameplay.

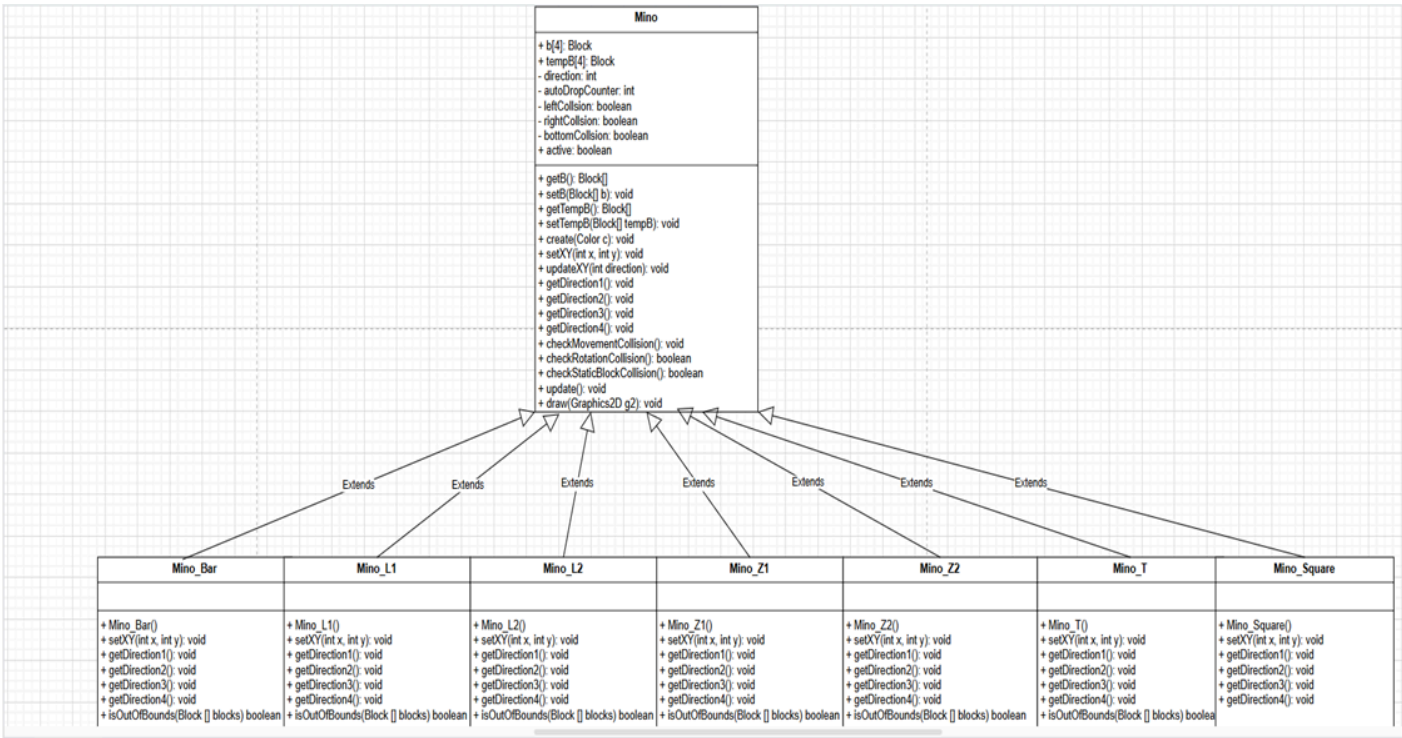
7. **public void draw(Graphics2D g2)**
   - **Description:** Renders the Tetris piece on the game board using the graphics context.
   - **Responsibilities:**
     - Draw all blocks in the piece.
     - Maintain consistency in rendering across all Tetris pieces.

Design Principles in Mino:

- **Encapsulation**:
  - Attributes like collisionLeft, collisionRight, and collisionBottom are protected, restricting direct access.
  - Movement and rotation logic are encapsulated within the update and checkCollision methods.
- **Abstraction**:
  - The Mino class abstracts shared behavior, allowing subclasses to focus on unique shapes.

- **Inheritance**:
    - Serves as the parent class for specific Tetris piece shapes (e.g., Mino_Bar, Mino_T).
- **Polymorphism**:
    - Subclasses can override methods like setXY to define unique configurations for their shape.

2.1.6 Mino Variants Class:



Role: The Mino class is extended by specific subclasses that define the shapes and behaviors of individual Tetris pieces. Each variant inherits common attributes and methods from Mino but customizes the piece layout and behavior.

Mino_Bar class:

Role:

The Mino_Bar class represents the "I" shaped Tetris piece. It extends the Mino class and provides specific behavior for its movement, rotation, and rendering.

Attributes:

Inherits all attributes from the Mino class without adding new attributes.

Methods:

- **public Mino_Bar()**

  Description: Constructor initializes the "I" shaped piece's structure.

  Responsibilities:

  - Sets the initial block layout for the piece.

- **public void setXY(int x, int y)**

  Description: Sets the initial position of the "I" piece on the grid.

  Responsibilities:

  - Aligns the blocks in a straight line**.**

- **public void getDirection0()**

  Description: Defines the horizontal layout of the "I" piece**.**

  Responsibilities:

  - Updates block positions to form a horizontal line**.**

- **public void getDirection1()**

  Description: Defines the vertical layout of the "I" piece**.**

  Responsibilities:

  - Updates block positions to form a vertical line.

- **public void getDirection2()**

  Description: Resets the piece to a horizontal line in the opposite direction.

- **public void getDirection3()**

  Description: Resets the piece to a vertical line in the opposite direction**.**

- **public boolean isOutOfBounds(Block[] blocks)**

  Description: Checks if any block of the "I" piece is out of the grid boundaries.

Mino_L1 class:

Role:

The Mino_L1 class represents the "L" shaped Tetris piece facing right. It extends the Mino class and provides rotation logic specific to its shape.

Attributes:

Inherits all attributes from the Mino class.

Methods:

- **public Mino_L1()**

  Description: Constructor initializes the "L" shaped structure.

- **public void setXY(int x, int y)**

  Description: Aligns the blocks in the default "L" shape at the given position.

- **public void getDirection0()**

  Description: Sets the default "L" shape layout.

- **public void getDirection1()**

  Description: Rotates the "L" piece clockwise to face upwards.

- **public void getDirection2()**

  Description: Rotates the "L" piece to face left.

- **public void getDirection3()**

  Description: Rotates the "L" piece to face downwards.

- **public boolean isOutOfBounds(Block[] blocks)**

  Description: Ensures all blocks of the "L1" piece are within boundaries after rotation.

Mino_L2 class:

Role:

The Mino_L2 class represents the mirrored "L" shaped Tetris piece facing left. It mirrors the logic of Mino_L1.

Attributes:

Inherits all attributes from the Mino class.

Methods:

- **public Mino_L2()**

  Description: Constructor initializes the mirrored "L" shape.

- **public void setXY(int x, int y)**

  Description: Aligns the blocks in the mirrored "L" shape at the given position.

- **Methods (getDirection0 through getDirection3):**

  Each defines rotation states for the mirrored "L" piece.

- **public boolean isOutOfBounds(Block[] blocks)**

  Description: Ensures valid boundary conditions for the "L2" piece.

Mino_Z1 and Mino_Z2 classes:

Role:

Mino_Z1 represents the standard "Z" shaped piece, while Mino_Z2 represents its mirrored counterpart.

Attributes:

Inherits all attributes from the Mino class.

Methods :

- Define initial layout (setXY).
- Rotate between horizontal and vertical states (getDirection0 and getDirection1).
- Check boundary violations after transformations (isOutOfBounds).

Mino_T class:

Role:

The Mino_T class represents the "T" shaped Tetris piece.

Attributes:

Inherits all attributes from the Mino class.

Methods:

- Define rotations for all four directions (getDirection0 through getDirection3).
- Check for boundary violations with isOutOfBounds.

Mino_Square class:

Role:

The Mino_Square class represents the "O" shaped piece. It does not rotate.

Attributes:

Inherits all attributes from the Mino class.

Methods:

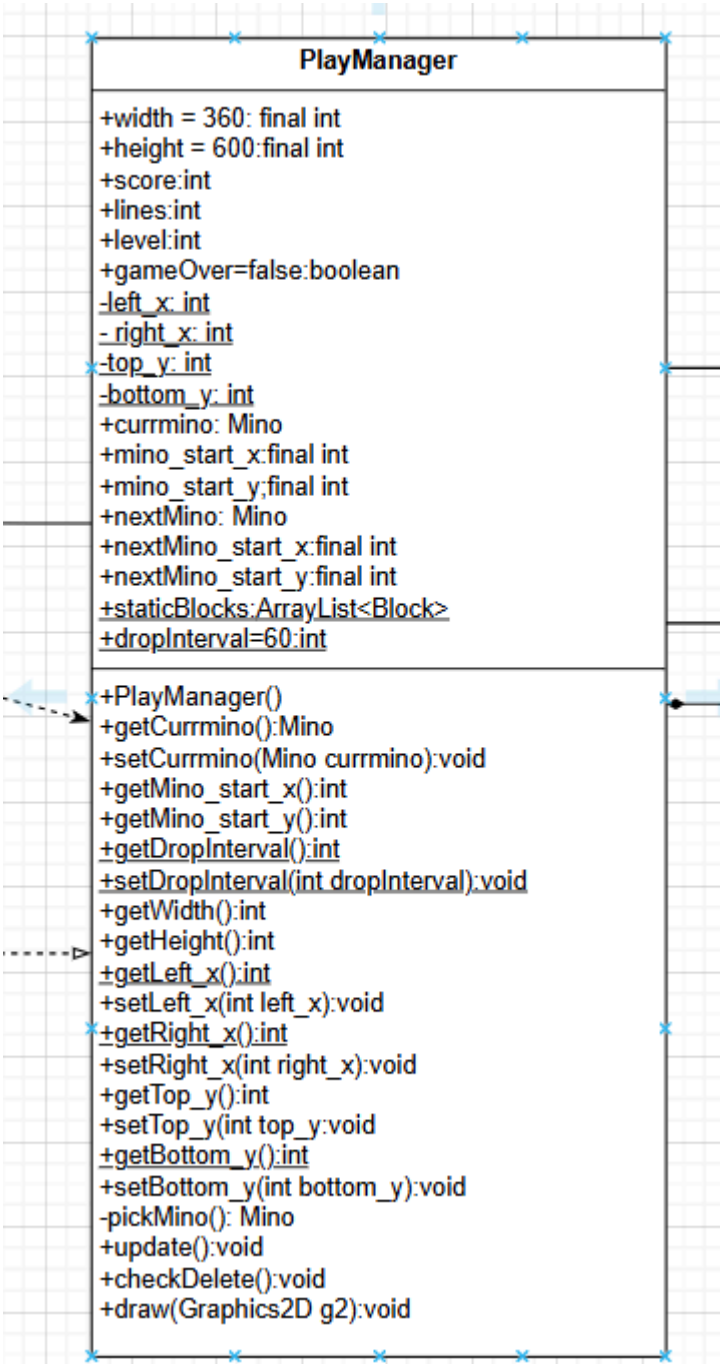
- **public Mino_Square()**

  Description: Initializes the "Square" piece structure.

- **public void setXY(int x, int y)**

  Description: Aligns the blocks in a 2x2 grid.

2.1.7 PlayManager Class:



Role: The PlayManager class manages the core gameplay mechanics. It is responsible for handling active and upcoming Tetris pieces, detecting and resolving collisions, managing static blocks, and clearing completed lines.

Attributes:

1. **final int width**
    - Description: Specifies the width of the playable grid (360 pixels).
    - Encapsulation: Marked final to prevent modification.
2. **final int height**
    - Description: Specifies the height of the playable grid (600 pixels).
    - Encapsulation: Marked final to ensure immutability.
3. **private static int left_x**
    - Description: Represents the x-coordinate of the left boundary of the game grid.
    - Encapsulation: Static and private for controlled access.
4. **private static int right_x**
    - Description: Represents the x-coordinate of the right boundary of the game grid.
    - Encapsulation: Static and private for consistency.
5. **private static int top_y**
    - Description: Represents the y-coordinate of the top boundary of the game grid.
    - Encapsulation: Static and private for controlled use.
6. **private static int bottom_y**
    - Description: Represents the y-coordinate of the bottom boundary of the game grid.
    - Encapsulation: Static and private to ensure boundary integrity.

7. **Mino currmino**
    - Description: Stores the current falling Mino.
    - Encapsulation: Default (package-private) to allow access within the package.
8. **final int mino_start_x**
    - Description: Specifies the initial x-coordinate where the Mino spawns.
    - Encapsulation: Marked final for immutability.
9. **final int mino_start_y**
    - Description: Specifies the initial y-coordinate where the Mino spawns.
    - Encapsulation: Marked final to prevent modification.
10. **Mino nextMino**
    - Description: Stores the next Mino to be spawned.
    - Encapsulation: Default for package-level access.
11. **final int nextMino_start_x**
    - Description: Specifies the x-coordinate where the next Mino is displayed.
    - Encapsulation: Marked final for consistency.
12. **final int nextMino_start_y**
    - Description: Specifies the y-coordinate where the next Mino is displayed.
    - Encapsulation: Marked final for immutability.
13. **public static ArrayList<Block> staticBlocks**
    - Description: A collection of Block objects representing blocks that are locked in place.

- Encapsulation: Public and static to allow access from outside and shared across instances.

14. static int dropInterval

- Description: Specifies the interval in frames between automatic downward movements of the current Mino.
- Encapsulation: Static to ensure consistency across the game.

15. int score

- Description: Tracks the player's score.
- Encapsulation: Default to allow package-level access.

16. int lines

- Description: Tracks the number of cleared lines.
- Encapsulation: Default to allow access within the package.

17. int level

- Description: Represents the current level of the game, which affects difficulty.
- Encapsulation: Default for package-level access.

18. boolean gameOver

- Description: A flag indicating whether the game has ended.
- Encapsulation: Default for internal package-level access.

Methods:

1. **public Mino getCurrmino()**
    - Description: Returns the current falling Mino.
2. **public void setCurrmino(Mino currmino)**
    - Description: Sets the current falling Mino.
3. **public int getMino_start_x()**
    - Description: Returns the starting x-coordinate of the current Mino.
4. **public int getMino_start_y()**
    - Description: Returns the starting y-coordinate of the current Mino.
5. **public static int getDropInterval()**
    - Description: Returns the interval between downward movements of the current Mino.
6. **public static void setDropInterval(int dropInterval)**
    - Description: Updates the interval between downward movements.
7. **public int getWidth()**
    - Description: Returns the width of the playable grid.
8. **public int getHeight()**
    - Description: Returns the height of the playable grid.
9. **public static int getLeft_x()**
    - Description: Returns the x-coordinate of the left boundary.

10. public void setLeft_x(int left_x)

- Description: Updates the x-coordinate of the left boundary.

11. public static int getRight_x()

- Description: Returns the x-coordinate of the right boundary.

12. public void setRight_x(int right_x)

- Description: Updates the x-coordinate of the right boundary.

13. public int getTop_y()

- Description: Returns the y-coordinate of the top boundary.

14. public void setTop_y(int top_y)

- Description: Updates the y-coordinate of the top boundary.

15. public static int getBottom_y()

- Description: Returns the y-coordinate of the bottom boundary.

16. public void setBottom_y(int bottom_y)

- Description: Updates the y-coordinate of the bottom boundary.

17. public PlayManager()

- Description: Constructor that initializes the game grid, Mino positions, and boundaries.

18. private Mino pickMino()

- Description: Randomly selects and returns a new Mino from the available types.

19. public void update()

- Description: Updates the game state, including moving Mino, handling row deletions, and checking for game-over conditions.

20. public void checkDelete()

- Description: Checks for completed rows and deletes them, updating score and level as needed.

21. public void resetGame()

- Description: Resets the game state, clearing blocks, resetting score, and restarting gameplay.

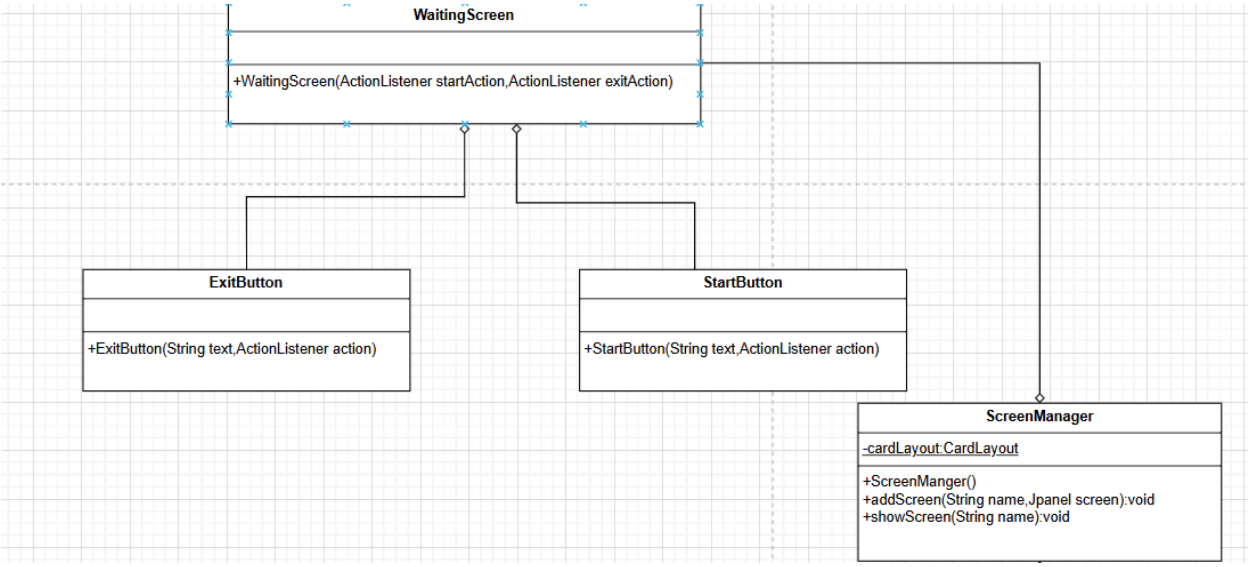
22. public void draw(Graphics2D g2)

- Description: Renders the game elements, including the grid, current and next Mino, score, level, and paused or game-over states.

Design Principles in PlayManger:

1. **Encapsulation:** All gameplay attributes are private and accessed through methods.
2. **Abstraction**: Hides complex logic like piece movement, collision detection, and scoring from external classes.
3. **Inheritance**: Reuses functionality from base objects like Mino and Block.
4. **Polymorphism**: Supports dynamic behaviors for different Tetris pieces or custom drawing methods.
5. **SOLID Principles:**
   - SRP: Focuses on managing gameplay mechanics only.
   - OCP: Easily extendable for new mechanics, such as power-ups or multiplayer.
   - LSP: Works seamlessly with graphical components like Graphics2D

2.1.8 User Interface Classes:



StartButton Class:

Role:

The StartButton class provides a customized button used to start the Tetris game. It offers a specific appearance and behavior tailored for the game's interface, emphasizing clarity and user-friendliness.

Methods:

1. **public StartButton**(String text, ActionListener action):
    - **Description:** Constructor that initializes the button with specified text and an action listener.
    - **Responsibilities**:
        - Sets the button text, font, background color, and text color.

- Adds an action listener to handle button clicks.

Design Principles in StartButton:

1. **Single Responsibility Principle (SRP):**
    - Focuses exclusively on creating a start button with specific styling and functionality.
2. **Abstraction:**
    - Abstracts the complexity of button styling and configuration within the constructor.

ExitButton Class:

Role:

The ExitButton class provides a customized button to exit the game. It ensures a distinct appearance and behavior to communicate its purpose effectively to the user.

Methods:

1. **public ExitButton**(String text, ActionListener action):
    - Description: Constructor that initializes the button with specified text and an action listener.
    - Responsibilities:
        - Sets the button text, font, background color, and text color.
        - Adds an action listener to handle button clicks.

Design Principles in ExitButton:

1. **Single Responsibility Principle (SRP**):
    - Handles only the creation and styling of an exit button.
2. **Abstraction:**
    - Hides the complexity of button configuration within the constructor.

WaitingScreen Class:

Role:

The WaitingScreen class represents the initial screen shown to the player, containing a title and buttons for starting or exiting the game.

Methods:

1. **public WaitingScreen(ActionListener startAction, ActionListener exitAction):**
    - **Description:** Constructor that initializes the layout, title label, start button, and exit button.
    - **Responsibilities:**
        - Configures layout using GridBagLayout.
        - Adds the title, start, and exit buttons to the screen with spacing and alignment.

Design Principles in WaitingScreen:

1. **Single Responsibility Principle (SRP):**
    - Dedicated to displaying the initial game interface with title and buttons.
2. **Abstraction:**
    - Hides the layout management details within the constructor.

ScreenManager Class:

Role:

The ScreenManager class manages transitions between different screens in the game, such as the WaitingScreen and gameplay screen.

Attributes:

1. **private CardLayout cardLayout:**
    - **Description:** Manages screen transitions within the panel.
    - **Encapsulation:** Ensures the layout logic is hidden from external access.

Methods:

1. **public ScreenManager():**
    - Description: Constructor that initializes the CardLayout and sets it as the layout for the panel.
    - Responsibilities:
        - Prepares the panel for adding and switching between screens.
2. **public void addScreen(String name, JPanel screen):**
    - Description: Adds a new screen to the manager, associated with a unique name.
3. **public void showScreen(String name):**
    - Description: Switches to the screen identified by the given name.

Design Principles in ScreenManager:

1. **Encapsulation:**
    - The CardLayout and screen management logic are private and abstracted away.
2. **Single Responsibility Principle (SRP):**
    - Responsible for adding and managing screen transitions only.
3. **Abstraction:**

Simplifies screen transitions through a clear API (addScreen, showScreen)

Chapter 3: JAVA APPLICATION

3.1 UI Design

3.1.1 Game board

The game board is the central component of the Tetris application. It serves as the canvas where Mino blocks are displayed during gameplay. The **GamePanel** class is responsible for managing the game board's layout. Below is the implementation:

```
public GamePanel(){  1 usage  oniic +1
    this.setPreferredSize(new Dimension(width, height));
    this.setBackground(Color.BLACK); //Black background
    this.setLayout(null);

    //Implement KeyListener
    this.addKeyListener(new KeyHandler());
    this.setFocusable(true);

    playerManager = new PlayManager();
}
```

```
public class PlayManager {  34 usages  huynhthienle +2

    final int width = 360;  7 usages
    final int height = 600;  3 usages
```

- The constructor of GamePanel responsible for setting dimension, layout and KeyListener.

3.1.2 Scoreboard

The scoreboard is integrated within the user interface to display the player's current score, number of deleted lines and levels. It updates dynamically during gameplay based on the number of lines cleared and the player's progress.

```
// Draw Score Frame
g2.setColor(Color.GREEN);
g2.drawRect(x, top_y,  width: 300,  height: 200);
g2.drawString( str: "LEVEL: " + level,  x: x + 50,  y: top_y + 50);
g2.drawString( str: "SCORE: " + score,  x: x + 50,  y: top_y + 100);
g2.drawString( str: "LINES: " + lines,  x: x + 50,  y: top_y + 150);
```

- Scoreboard is written in the **draw()** method of the **PlayManager** class.
- These lines of code are responsible for drawing the grid of the scoreboard, setting its position, and displaying level, score, and the number of deleted lines continuously.

3.2 Basic Components

3.2.1 Overview

The Tetris game is built on a modular architecture, where the functionality is divided into distinct components to ensure clarity, reusability, and ease of maintenance. Each component is responsible for a specific part of the game, such as managing the game logic, rendering the interface, handling user input, and maintaining the flow of the game.

At its core, the game comprises the following foundational elements:

- Game Board: Represents the grid where Tetriminoes (shapes) fall, align, and interact. The board tracks the occupied and empty spaces using a 2D array structure.

- Tetrominoes: These are the blocks that the player controls, each with unique shapes (e.g., I, O, T, L, J, Z, S). They move, rotate, and interact with the grid.
- Game State Manager: Controls the progression of the game, including score calculation, level changes, and detecting the game over state.
- Input Handler: Captures user input (keyboard controls) to move and rotate Tetriminoes.
- UI and Graphics: Responsible for rendering the grid, Tetrominoes, and other visual elements such as the score, level, and game over screen.

These components interact seamlessly to deliver a smooth gaming experience. The modularity of the design ensures that individual components, such as the scoring system or user input, can be updated or extended without disrupting the entire game.

3.2.2 Key Classes

1. **GamePanel**

Purpose: Acts as the central game engine that manages the game's graphical interface, background music, and game loop.

Responsibilities:

➔ Defines the game screen dimensions (WIDTH and HEIGHT).

➔ Runs the game loop to update game logic and repaint graphics at a consistent frame rate (FPS).

➔ Handles user interactions by adding a KeyListener for keyboard inputs.

➔ Plays background music using the Sound class.

Key Methods:

- start(): Initializes and starts the game thread and the music.

```
//start() method
public void start(){  1 usage   oniic +1
    gameThread = new Thread( task: this);
    gameThread.start();

    music.playBackgroundMusic( filePath: "/music.wav");
}
```

- update(): Updates the state of the game objects (via PlayManager).

```
//update() method
public void update(){  1 usage  oniic
    playerManager.update();
}
```

- paintComponent(Graphics g): Renders all visual components, such as the Tetris blocks and game area, on the screen.

```
//paintComponent() method
public void paintComponent(Graphics g){  oniic
    super.paintComponent(g);

    Graphics2D g2 = (Graphics2D)g;
    playerManager.draw(g2);
}
```

2. **PlayManager**

Purpose: Controls the core gameplay mechanics, including managing the movement and behavior of Tetris blocks.

Responsibilities:

➔ Handles the current and next Tetris pieces, including their spawning and movement.

➔ Manages the playfield boundaries and the logic for clearing full rows.

➔ Tracks game statistics, such as score, level, and lines cleared.

➔ Checks for game-over conditions and resets the game state when necessary.

Key Methods:

- update(): Check the game state, including block movements and collisions.

```java
// update() method
public void update() {  1 usage   huynhthienle +2
    if (gameOver) {
        music.stopBackgroundMusic();
        if (KeyHandler.isSpacePressed()) {
            resetGame();
        }
        return;
    }

    if (KeyHandler.isPausePressed()) {
        return; // Pause the game
    }
```

Code: Check the state of the game (game over and pausing)

```java
if (!currmino.active) {
    staticBlocks.add(currmino.b[0]);
    staticBlocks.add(currmino.b[1]);
    staticBlocks.add(currmino.b[2]);
    staticBlocks.add(currmino.b[3]);

    // Check for game over condition
    for (Block block : staticBlocks) {
        if (block.getY() <= top_y) {
            gameOver = true;
            return;
        }
    }
```

Code: Generate next mino block

- checkDelete(): Detects and clears completed rows, adjusts the score, and speeds up the game.

```
public void checkDelete() {  1 usage   Baogankteam
    int x = left_x;
    int y = top_y;
    int blockCount = 0;

    while (x < right_x && y < bottom_y) {
        for (int i = 0; i < staticBlocks.size(); i++) {
            if (staticBlocks.get(i).getX() == x && staticBlocks.get(i).getY() == y) {
                blockCount++;
            }
        }

        x += Block.getSize();

        if (x == right_x) {
            if (blockCount == 12) {
                for (int i = staticBlocks.size() - 1; i > -1; i--) {
                    if (staticBlocks.get(i).getY() == y) {
                        staticBlocks.remove(i);
                    }
                }

                // Music when a row is deleted
                music.playEffectSound( filePath: "/clear.wav");
```

```
                lines++;
                score += 50;

                // Change drop speed based on level
                if (lines % 5 == 0 && dropInterval > 1) {
                    level += 1;

                    if (dropInterval % 5 == 0 && dropInterval > 10) {
                        dropInterval -= 10;
                    } else {
                        dropInterval -= 1;
                    }
                    setDropInterval(dropInterval);
                }

                // move lines above the deleted line down
                for (int i = 0; i < staticBlocks.size(); i++) {
                    if (staticBlocks.get(i).getY() < y) {
                        staticBlocks.get(i).setY(staticBlocks.get(i).getY() + Block.getSize());
                    }
                }
            }
            blockCount = 0;
            x = left_x;
            y += Block.getSize();
        }
    }
}
```

- resetGame(): Resets all game variables to restart the game.

```java
// Add a method to reset the game state
public void resetGame() {  1 usage   huynhthienle +1
    staticBlocks.clear();
    score = 0;
    lines = 0;
    level = 0;
    gameOver = false;
    dropInterval = 60;

    currmino = pickMino();
    currmino.setXY(mino_start_x, mino_start_y);
    nextMino = pickMino();
    nextMino.setXY(nextMino_start_x, nextMino_start_y);
    music.playBackgroundMusic( filePath: "/music.wav");
}
```

-

- draw(Graphics2D g): Renders the play area, current/next blocks, and game statistics.

```java
// draw() method
public void draw(Graphics2D g2) {  1 usage   huynhthienle +2
    // Draw Play Area Frame
    g2.setColor(Color.WHITE);
    g2.setStroke(new BasicStroke( width: 4f));
    g2.drawRect( x: left_x - 4,  y: top_y - 4,  width: width + 8,  height: height + 8); // orderlist (x,y,width,height)

    // Draw Next Mino Frame
    int x = right_x + 100;
    int y = bottom_y - 200;
    g2.drawRect(x, y,  width: 200,  height: 200);
    g2.setFont(new Font( name: "Arial", Font.PLAIN,  size: 30));
    g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    g2.drawString( str: "Next",  x: x + 60,  y: y + 60); // order list (x,y)

    // Draw Score Frame
    g2.setColor(Color.GREEN);
    g2.drawRect(x, top_y,  width: 300,  height: 200);
    g2.drawString( str: "LEVEL: " + level,  x: x + 50,  y: top_y + 50);
    g2.drawString( str: "SCORE: " + score,  x: x + 50,  y: top_y + 100);
    g2.drawString( str: "LINES: " + lines,  x: x + 50,  y: top_y + 150);

    // Draw the current mino
    if (currmino != null) {
        currmino.draw(g2);
    }

    // Draw nextMino
    nextMino.draw(g2);
```

```java
// draw() method
public void draw(Graphics2D g2) {  1 usage  huynhthienle +2
    // Draw Play Area Frame
    g2.setColor(Color.WHITE);
    g2.setStroke(new BasicStroke( width: 4f));
    g2.drawRect( x: left_x - 4,  y: top_y - 4,  width: width + 8,  height: height + 8); // orderlist (x,y,width,height)

    // Draw Next Mino Frame
    int x = right_x + 100;
    int y = bottom_y - 200;
    g2.drawRect(x, y,  width: 200,  height: 200);
    g2.setFont(new Font( name: "Arial", Font.PLAIN,  size: 30));
    g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    g2.drawString( str: "Next",  x: x + 60,  y: y + 60); // order list (x,y)

    // Draw Score Frame
    g2.setColor(Color.GREEN);
    g2.drawRect(x, top_y,  width: 300,  height: 200);
    g2.drawString( str: "LEVEL: " + level,  x: x + 50,  y: top_y + 50);
    g2.drawString( str: "SCORE: " + score,  x: x + 50,  y: top_y + 100);
    g2.drawString( str: "LINES: " + lines,  x: x + 50,  y: top_y + 150);

    // Draw the current mino
    if (currmino != null) {
        currmino.draw(g2);
    }

    // Draw nextMino
    nextMino.draw(g2);
```

```
    // Draw staticBlocks
    for (Block b : staticBlocks) {
        b.draw(g2);
    }

    // Draw pause
    g2.setColor(Color.yellow);
    g2.setFont(g2.getFont().deriveFont( size: 50f));
    if (KeyHandler.isPausePressed()) {
        FontMetrics fm = g2.getFontMetrics();
        String pauseText = "PAUSE";
        int textWidth = fm.stringWidth(pauseText);
        int textX = left_x + (width - textWidth) / 2;
        g2.drawString(pauseText, textX,  y: top_y + 300);
    }

    // Draw game over
    if (gameOver) {
        g2.setColor(Color.RED);
        g2.setFont(g2.getFont().deriveFont( size: 50f));
        FontMetrics fm = g2.getFontMetrics();
        String gameOverText = "GAME OVER";
        int textWidth = fm.stringWidth(gameOverText);
        int textX = left_x + (width - textWidth) / 2;
        g2.drawString(gameOverText, textX,  y: top_y + 300);
    }
}
```

3. **KeyHandler**

Purpose: Manages keyboard input for controlling gameplay.

Responsibilities:

➔ Listens for key presses and releases to handle player input (e.g., moving pieces left, right, rotating, pausing).

➔ Updates boolean flags (upPressed, downPressed, etc.) based on key events, allowing other classes to respond accordingly.

➔ Toggles game pause state and plays/stops background music.

Key Methods:

- keyPressed(KeyEvent e): Processes key inputs like arrow keys and spacebar for controlling Tetris blocks.

- keyReleased(KeyEvent e): Resets the corresponding flags when keys are released.\

```java
public void keyReleased(KeyEvent e) {
    int code = e.getKeyCode();

    switch (code) {
        case KeyEvent.VK_UP:
        case KeyEvent.VK_W: // W key for "up"
            upPressed = false;
            break;
        case KeyEvent.VK_DOWN:
        case KeyEvent.VK_S: // S key for "down"
            downPressed = false;
            break;
        case KeyEvent.VK_LEFT:
        case KeyEvent.VK_A: // A key for "left"
            leftPressed = false;
            break;
        case KeyEvent.VK_RIGHT:
        case KeyEvent.VK_D: // D key for "right"
            rightPressed = false;
            break;
        case KeyEvent.VK_SPACE:
            spacePressed = false;
            break;
    }
}
```

- Static utility methods (e.g., isPausePressed(), isSpacePressed()) provide access to the current state of key presses.

```java
public static boolean isUpPressed() { return upPressed; }

public static boolean isDownPressed() { return downPressed; }

public static boolean isLeftPressed() { return leftPressed; }

public static boolean isRightPressed() { return rightPressed; }
```

3.3 Main methods

3.3.1 Game Initialization

The game initializes by setting up the window, play area, and essential components.

```java
public class Main {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Tetris Game");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setResizable(false);


        GamePanel gamePanel = new GamePanel();

        frame.add(gamePanel);

        frame.pack();


        frame.setLocationRelativeTo(null);

        frame.setVisible(true);
```

```
        gamePanel.start();
    }
}
```

3.3.2 Game Loop

The game loop updates and renders the game at 60 FPS.

```
@Override
public void run() {
    double drawInterval = 1000000000/ FPS;
    double delta = 0;
    long lastTime = System.nanoTime();
    long currentTime;

    while (gameThread != null){
        currentTime = System.nanoTime();
        delta += (currentTime - lastTime)/drawInterval;
        lastTime = currentTime;

        if (delta >= 1){
            update();
            repaint();
            delta--;
        }
    }
}
```

3.3.3 Tetrimino Controls

Handles player input for moving and rotating Tetriminos.

```
@Override
public void keyPressed(KeyEvent e) {
   int code = e.getKeyCode();


   switch (code) {
       case KeyEvent.VK_UP:
       case KeyEvent.VK_W: // W key for "up"
           upPressed = true;
           break;
       case KeyEvent.VK_DOWN:
       case KeyEvent.VK_S: // S key for "down"
           downPressed = true;
           break;
       case KeyEvent.VK_LEFT:
       case KeyEvent.VK_A: // A key for "left"
           leftPressed = true;
           break;
       case KeyEvent.VK_RIGHT:
       case KeyEvent.VK_D: // D key for "right"
           rightPressed = true;
           break;
       case KeyEvent.VK_P:
           pausePressed = !pausePressed;
```

```
            if (pausePressed) {
                music.stopBackgroundMusic();
            } else {
                music.playBackgroundMusic("/music.wav");
            }
            break;
        case KeyEvent.VK_SPACE:
            spacePressed = true;
            break;
    }
}

@Override
public void keyReleased(KeyEvent e) {
    int code = e.getKeyCode();

    switch (code) {
        case KeyEvent.VK_UP:
        case KeyEvent.VK_W: // W key for "up"
            upPressed = false;
            break;
        case KeyEvent.VK_DOWN:
        case KeyEvent.VK_S: // S key for "down"
            downPressed = false;
            break;
        case KeyEvent.VK_LEFT:
        case KeyEvent.VK_A: // A key for "left"
```

```
            leftPressed = false;
            break;
        case KeyEvent.VK_RIGHT:
        case KeyEvent.VK_D: // D key for "right"
            rightPressed = false;
            break;
        case KeyEvent.VK_SPACE:
            spacePressed = false;
            break;
    }
}
```

3.3.4 Game State Management

Manages states like running, paused, and game-over.

```
if (gameOver) {
    music.stopBackgroundMusic();
    if (KeyHandler.isSpacePressed()) {
        resetGame();
    }
    return;
}
if (KeyHandler.isPausePressed()) {
    return; // Pause the game
}
```

3.4 Features and Implementation

3.4.1 Scoring and Level Progress System

The Scoring and Level Progress System calculates the player's score based on the number of cleared lines, tracks the current level, and adjusts the game's drop speed as the level increases. Clearing multiple lines at once rewards the player with additional points. As levels progress, the game becomes more challenging by increasing the Tetrimino drop speed.

Key Features:

- Score Calculation:
    - 50 points are awarded for each cleared line.
    - Score increases when multiple lines are cleared simultaneously.

- Level Progression:
    - The level increases every 5 lines cleared.
    - Each level increment reduces the Tetrimino drop interval, making the game faster.

- Speed Adjustment:
    - Drop speed reduces by 10 or 1 frame based on the level progression, creating a more challenging gameplay experience.

```
public void checkDelete() {
   int x = left_x;
   int y = top_y;
```

```
    int blockCount = 0;


    while (x < right_x && y < bottom_y) {
        for (int i = 0; i < staticBlocks.size(); i++) {
            if (staticBlocks.get(i).getX() == x && staticBlocks.get(i).getY() ==
y) {
                blockCount++;
            }
        }

        x += Block.getSize();

        if (x == right_x) {
            if (blockCount == 12) {
                for (int i = staticBlocks.size() - 1; i > -1; i--) {
                    if (staticBlocks.get(i).getY() == y) {
                        staticBlocks.remove(i);
                    }
                }

                // Music when a row is deleted
                music.playEffectSound("/clear.wav");

                lines++;
                score += 50;

                // Change drop speed based on level
```

```
                if (lines % 5 == 0 && dropInterval > 1) {
                    level += 1;

                    if (dropInterval % 5 == 0 && dropInterval > 10) {
                        dropInterval -= 10;
                    } else {
                        dropInterval -= 1;
                    }
                    setDropInterval(dropInterval);
                }
                // move lines above the deleted line down
                for (int i = 0; i < staticBlocks.size(); i++) {
                    if (staticBlocks.get(i).getY() < y) {
                        staticBlocks.get(i).setY(staticBlocks.get(i).getY() +
Block.getSize());
                    }
                }
            }
            blockCount = 0;
            x = left_x;
            y += Block.getSize();
        }
    }
}
```

3.4.2 Sound effects and music

The Sound Effects and Music system enhances the player experience by adding immersive background music and sound effects for various game events like clearing lines or game over.

Key Features:

- Background Music:
    - Plays a continuous loop of background music during gameplay.
    - Stops the music when the game is paused or over.

- Sound Effects:
    - Provides one-time sound effects for game events, such as clearing a line or pressing a button.
    - Ensures sound effects do not overlap by stopping the previous clip before playing a new one.
- Dynamic Playback Control:
    - Allows starting, stopping, and looping audio files.

```
public class Sound {

   private Clip backgroundClip; // Clip for background music
```

```
    private Clip effectClip;      // Clip for one-time sound effects

    public void playBackgroundMusic(String filePath) {

        try {

            if (backgroundClip == null) {

                // Load and initialize the background music clip

                 AudioInputStream audio =
AudioSystem.getAudioInputStream(getClass().getResource(filePath));

                backgroundClip = AudioSystem.getClip();

                  backgroundClip.open(audio);

                backgroundClip.loop(Clip.LOOP_CONTINUOUSLY); // Loop background
music

            }



            // Start background music only if it's not already playing

            if (!backgroundClip.isRunning()) {

                backgroundClip.start();

            }

        } catch (Exception e) {

            e.printStackTrace();
```

```
        }

    }

    public void stopBackgroundMusic() {

        if (backgroundClip != null && backgroundClip.isRunning()) {

            backgroundClip.stop();

        }

    }

    public void playEffectSound(String filePath) {

        try {

            // Stop and release the previous effect clip (if any) to avoid
overlap

            if (effectClip != null && effectClip.isRunning()) {

                effectClip.stop();

            }



            // Load a new sound effect

            AudioInputStream audio =
AudioSystem.getAudioInputStream(getClass().getResource(filePath));

            effectClip = AudioSystem.getClip();
```

```
            effectClip.open(audio);



            // Play the sound effect once

            effectClip.start();

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}
```

Chapter 4: CONCLUSION

1. Achieved Goals

In this project, the implementation of the classic Tetris game was successfully completed using the principles of Object-Oriented Programming (OOP). The following key goals were achieved:

1. **Game Functionality:**
    - The core mechanics of Tetris, including piece generation, rotation, movement, and line clearing, were implemented effectively.
    - Collision detection was handled to ensure seamless gameplay.
2. **User Interface:**
    - A graphical interface was developed using Java Swing (or the relevant library), providing players with an engaging and interactive experience.
    - A scoring system and a real-time display of upcoming pieces were integrated to enhance user engagement.
3. **OOP Design Principles:**
    - Key OOP concepts such as encapsulation, inheritance, and polymorphism were applied to design a modular and reusable codebase.
    - Classes such as Tetrimino, GameBoard, and GameController were created to ensure a clear separation of concerns and maintainability.

4. **Game Logic and Mechanics:**
    - Implemented features like increasing difficulty levels, automatic dropping of pieces, and a game-over condition when the board fills up.

Overall, the project achieved its primary goal of creating a functional and enjoyable Tetris game while reinforcing OOP concepts learned during the course.

2. Future Works

While the current implementation provides a complete and functional Tetris game, several areas for improvement and expansion have been identified:

1. **Enhanced Graphics and Animations:**
    - Improve the visual appeal by adding animations for piece movement, line clearing, and game-over screens.
    - Implement themes or customizable designs for the game board and pieces.

2. **Multiplayer Mode:**
    - Introduce a competitive multiplayer mode where two players can compete, with mechanics like sending "garbage lines" to the opponent.

3. **Online Leaderboard:**
    - Implement an online leaderboard to allow players to upload their scores and compete globally.

4. **AI-Powered Opponent:**
    - Develop an AI opponent for single-player versus mode, using algorithms to simulate intelligent gameplay.

5. **Portability:**
    - Extend the game's compatibility to mobile devices or create a web-based version for a broader audience.

By incorporating these future enhancements, the Tetris project can evolve into a more robust, engaging, and feature-rich application.

Chapter 5: REFERENCE

Java Application

1. **Books**:
    - Schildt, H. *Java: The Complete Reference*. McGraw-Hill Education. Detailed guide to Java programming, including JDBC.
    - Earman, D. *Java Database Best Practices*. O'Reilly Media. Best practices for database integration in Java applications.