SWINBURNE UNIVERSITY OF TECHNOLOGY

COS20007

Object Oriented Programming

Learning Summary Report

VU PHAN
104222099
104222099@student.swin.edu.au

I. Introduction

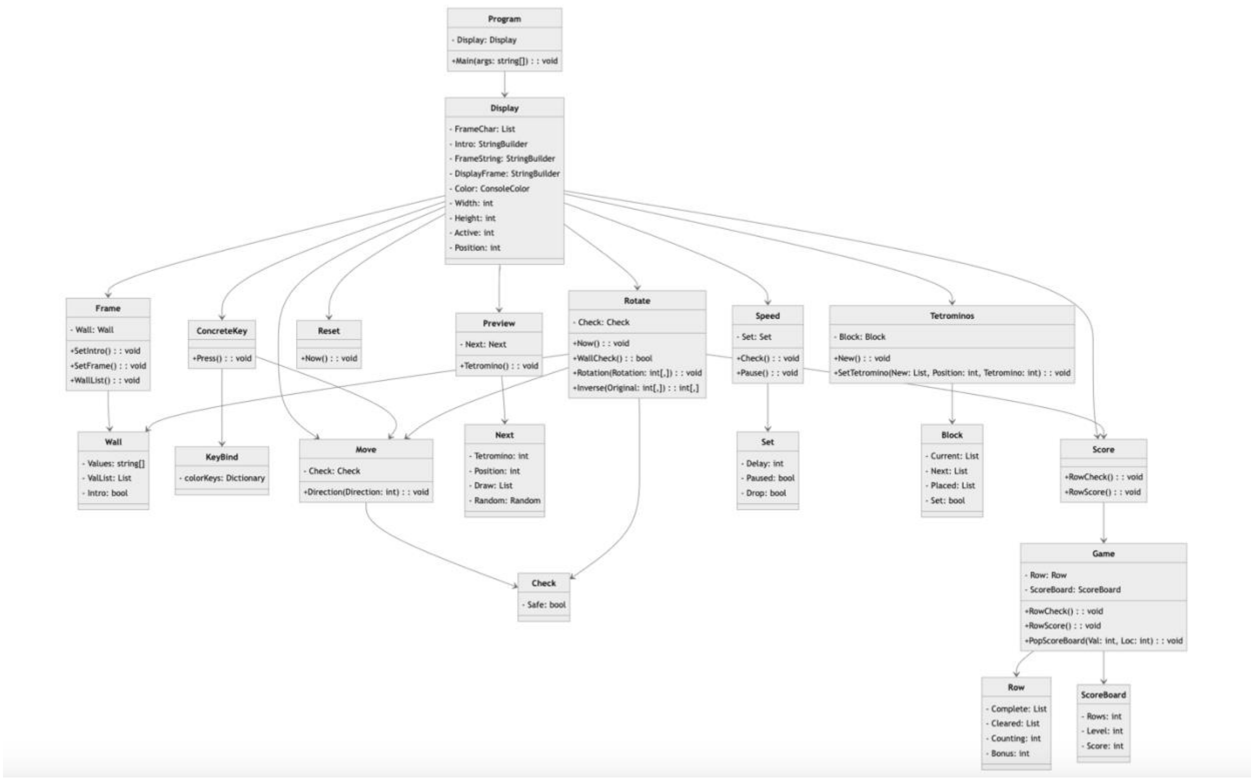
TetrisV is an ode to the classic Tetris game, aiming to recreate the nostalgic gameplay experienced on the Gameboy console and the original 1984 version. It encompasses various components, including a display manager, frame organizer, user controls, scoring system, and Tetromino manager. The game aims to engage users in an interactive Tetris experience, manipulating falling Tetrominos to complete rows and achieve high scores.

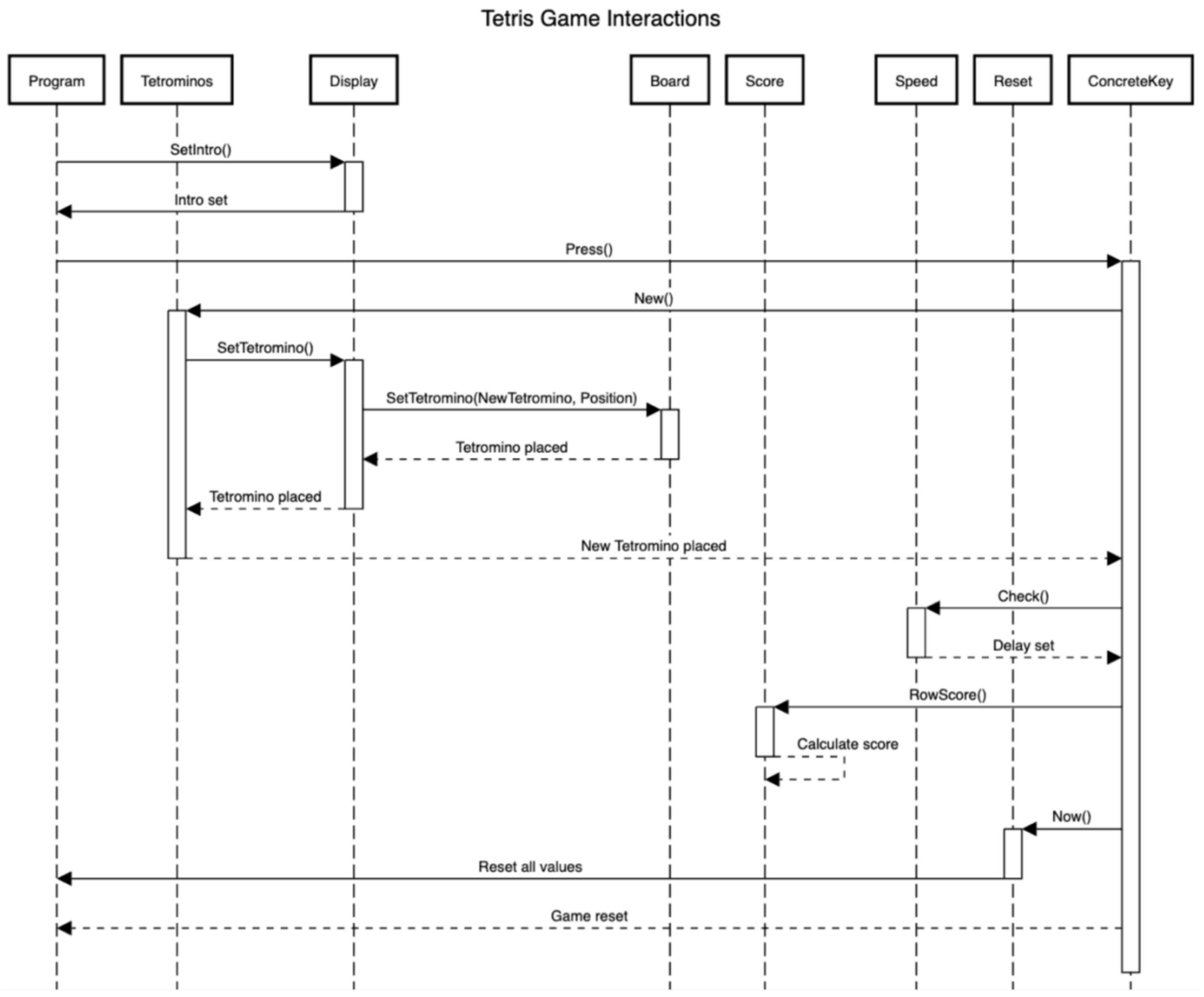
II. Submission Details

a. Code Files:

https://drive.google.com/drive/folders/1jBq7hSPkSS75KIpX91-iuQLalk5MDklU?usp=sharing

b. UML Class Diagram:





III. Design Overview

a. Key Feature.

The custom program demonstrates the following features:

1. Tetromino Manipulation: Players can rotate and move falling tetromino shapes to fit into available spaces, aiming to create solid rows without gaps.

2. Scoring System: The game keeps track of completed rows, awarding points for each row filled. Clearing multiple rows simultaneously earns bonus points.

3. Increasing Difficulty: As players progress, the game speeds up, challenging them to react faster and make decisions more quickly.

4. Display Manager: Manages the game screen, presenting the falling tetromino and the game board. It showcases essential information, such as the next tetromino.

5. Controls: Allows players to interact with the game using keyboard controls, enabling movement, rotation, and pause functionality.

6. Leveling System: As players achieve higher scores or complete rows, they advance to higher levels, increasing the game's speed and difficulty.

7. Game Over Condition: When tetrominoes reach the top of the game board, the game ends. It displays the final score and allows players to restart.

b. Classe segmentation.

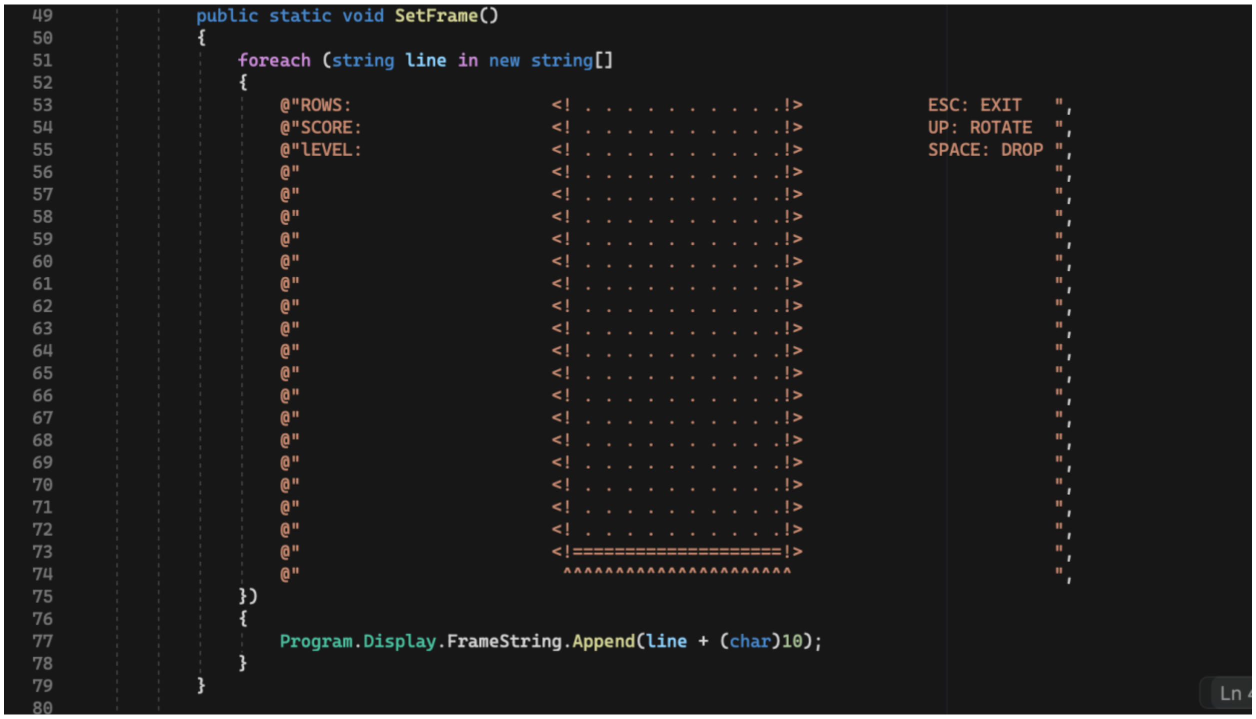
Class	Description	Responsibilities
Display	Manages screen output	- Display frames - Set frame elements - Manage game screen dimensions - Handle active state and position
Frame	Organizes game frames	- Manages walls in the frame - Sets introduction text - Lists all walls in the frame
Wall	Handles wall elements within frames	- Manages array of wall values - Handles wall lists - Controls introduction state
Key	Manages user controls	- Maps key-color pairs for controls - Handles key press events
Move	Controls movement direction	- Manages movement directions
Check	Manages the checking mechanism	- Indicates safe condition
Preview	Handles the next tetromino info	- Generates the next tetromino
Reset	Handles the reset event	- Resets the game
Rotate	Manages the checking mechanism for rotation	- Handles rotation events - Checks wall boundaries for rotation - Controls rotation process - Inverts rotation for calculations

Class	Description	Responsibilities
Score	Manages rows and scoring	- Manages completed rows - Calculates scores for rows - Updates the scoreboard
Row	Handles completed and cleared rows	- Manages completed row indices - Counts completed rows - Manages additional score bonuses
ScoreBoard	Handles scoring information	- Manages completed rows, current level, and overall score
Speed	Manages speed settings	- Checks speed conditions - Handles pause functionality
Set	Manages game settings	- Manages time delay for movements - Controls pause and tetromino dropping
Tetrominos	Handles block arrangements in the game	- Generates new tetrominos - Sets tetrominos on the board
Block	Manages tetromino arrangements	- Manages current, next, and placed tetromino arrangements - Controls arrangement setting

c. Methods

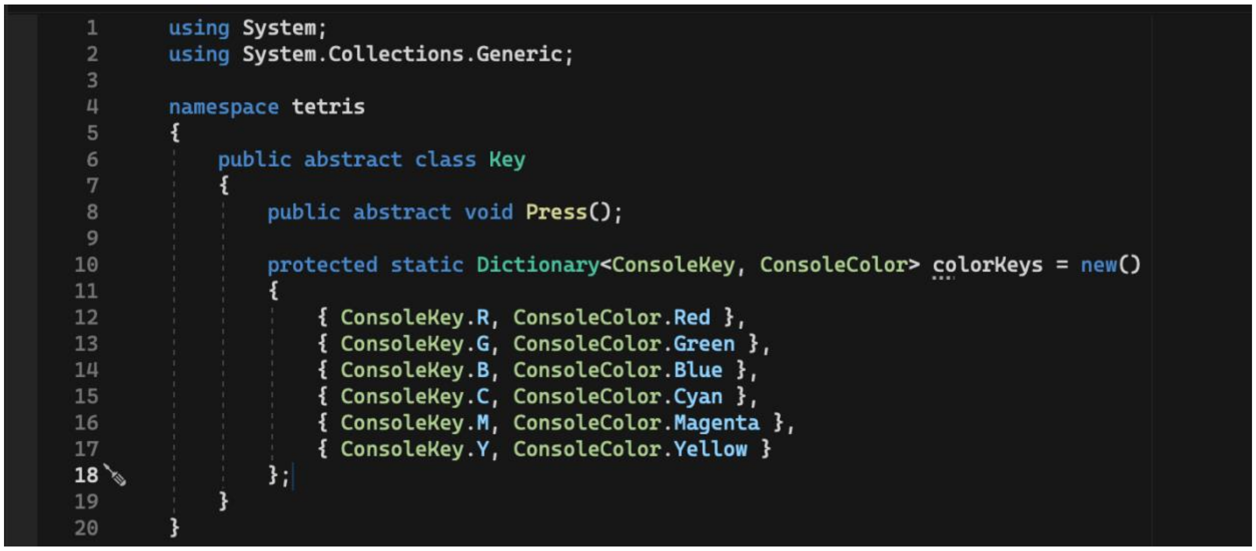
Main method:

1. **SetFrame():**
    - **Class: Frame**
    - **Description:** Sets up the game frame elements including walls and visual components essential for gameplay.



```
public static void SetFrame()
{
    foreach (string line in new string[]
    {
        @"ROWS:                     <! . . . . . . . . . . .!>          ESC: EXIT   ",
        @"SCORE:                    <! . . . . . . . . . . .!>          UP: ROTATE  ",
        @"LEVEL:                    <! . . . . . . . . . . .!>          SPACE: DROP ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <! . . . . . . . . . . .!>                      ",
        @"                          <!======================!>                      ",
        @"                           ^^^^^^^^^^^^^^^^^^^^^^                         ",
    })
    {
        Program.Display.FrameString.Append(line + (char)10);
    }
}
```

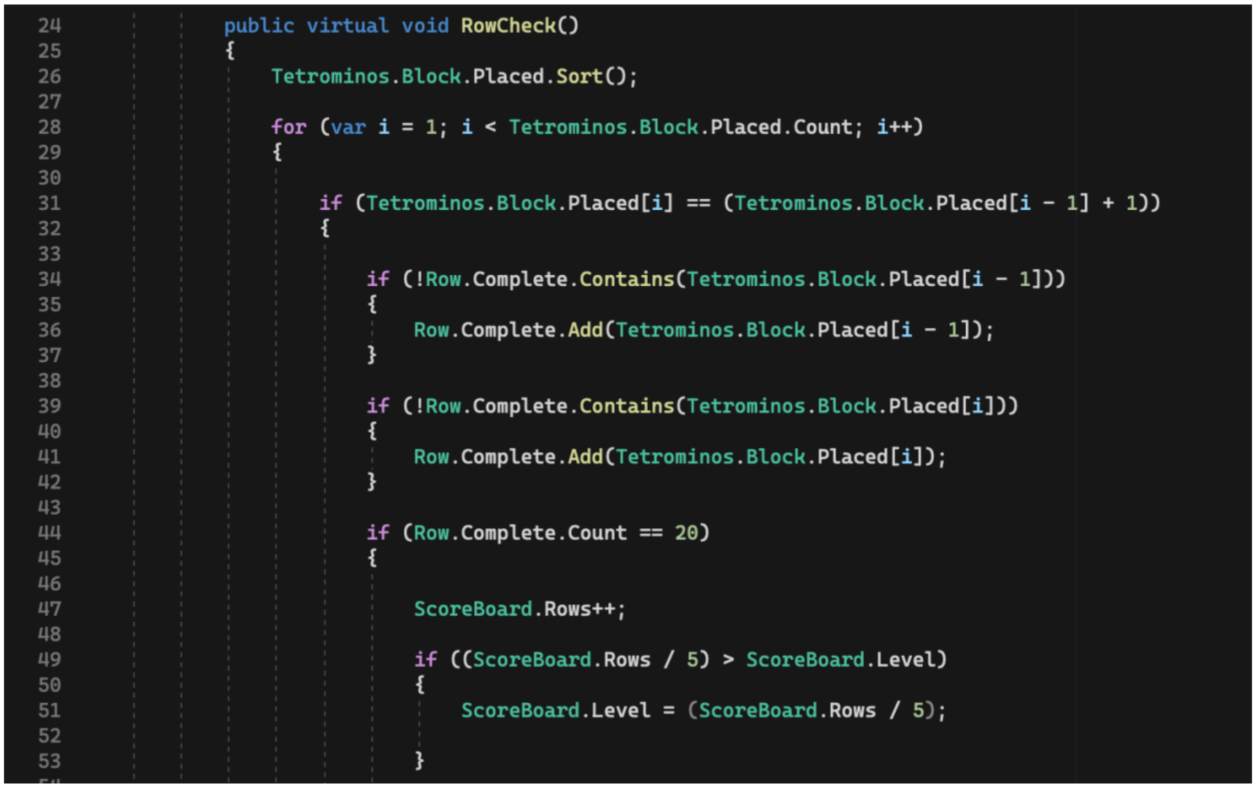
2. **Press():**
   - **Class:** Key
   - **Description:** Handles key press events, allowing control over game elements like movement or rotation of tetrominos.



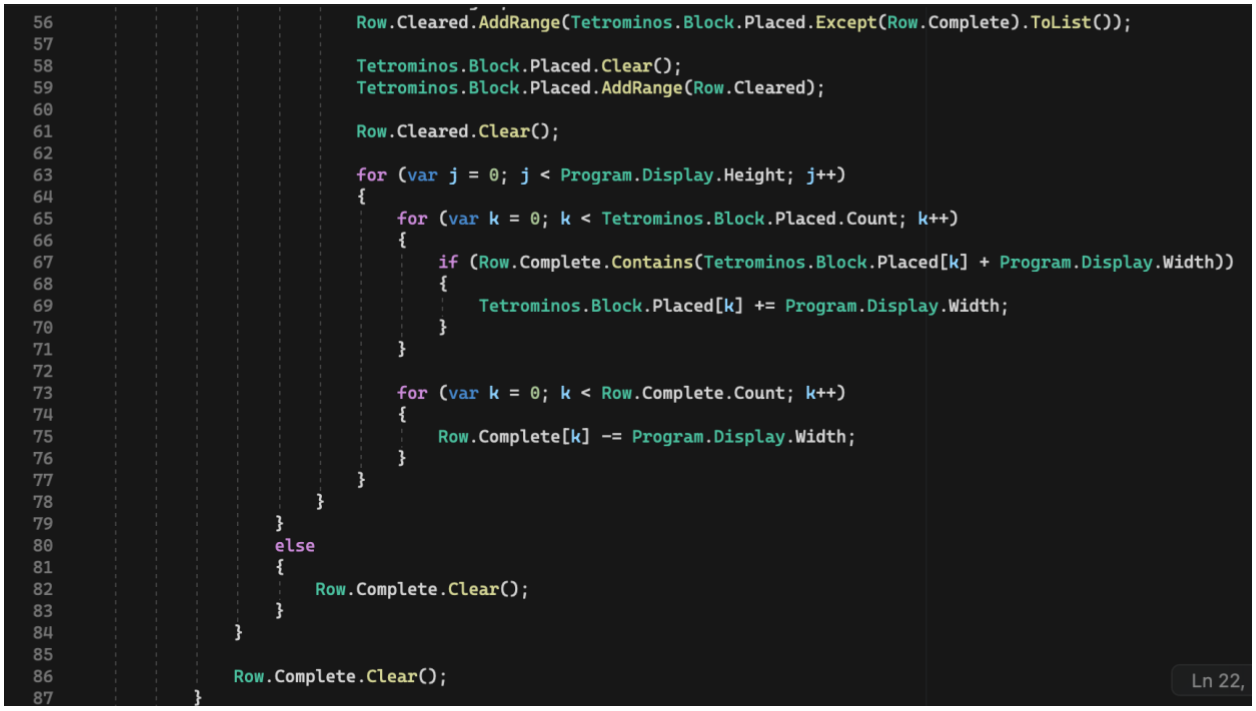
```
using System;
using System.Collections.Generic;

namespace tetris
{
    public abstract class Key
    {
        public abstract void Press();

        protected static Dictionary<ConsoleKey, ConsoleColor> colorKeys = new()
        {
            { ConsoleKey.R, ConsoleColor.Red },
            { ConsoleKey.G, ConsoleColor.Green },
            { ConsoleKey.B, ConsoleColor.Blue },
            { ConsoleKey.C, ConsoleColor.Cyan },
            { ConsoleKey.M, ConsoleColor.Magenta },
            { ConsoleKey.Y, ConsoleColor.Yellow }
        };
    }
}
```

3. **RowCheck():**
   - **Class:** Row

- **Description:** Checks the game grid for completed rows, triggering actions upon completion.

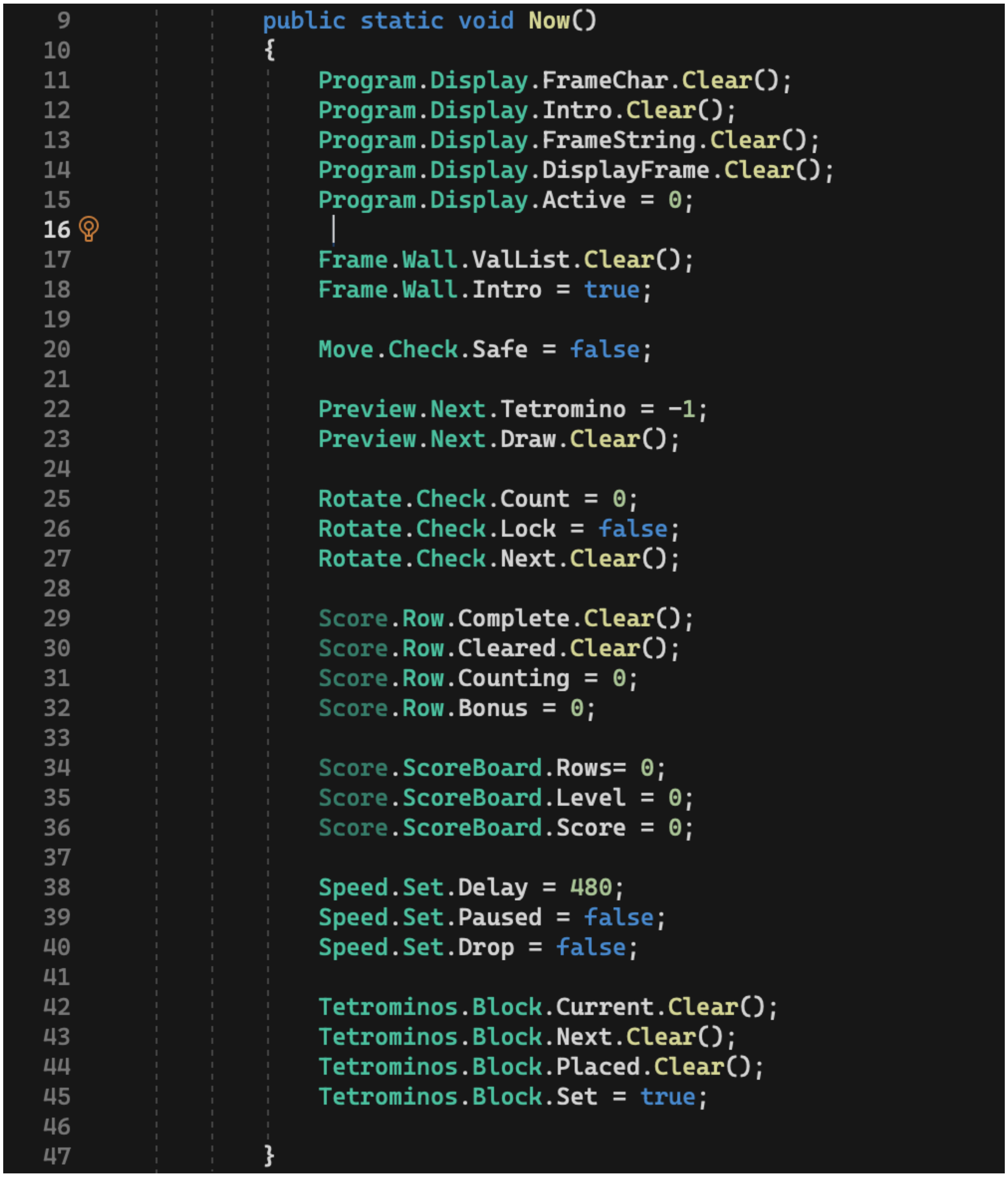


```
public virtual void RowCheck()
{
    Tetrominos.Block.Placed.Sort();

    for (var i = 1; i < Tetrominos.Block.Placed.Count; i++)
    {

        if (Tetrominos.Block.Placed[i] == (Tetrominos.Block.Placed[i - 1] + 1))
        {

            if (!Row.Complete.Contains(Tetrominos.Block.Placed[i - 1]))
            {
                Row.Complete.Add(Tetrominos.Block.Placed[i - 1]);
            }

            if (!Row.Complete.Contains(Tetrominos.Block.Placed[i]))
            {
                Row.Complete.Add(Tetrominos.Block.Placed[i]);
            }

            if (Row.Complete.Count == 20)
            {

                ScoreBoard.Rows++;

                if ((ScoreBoard.Rows / 5) > ScoreBoard.Level)
                {
                    ScoreBoard.Level = (ScoreBoard.Rows / 5);

                }
```



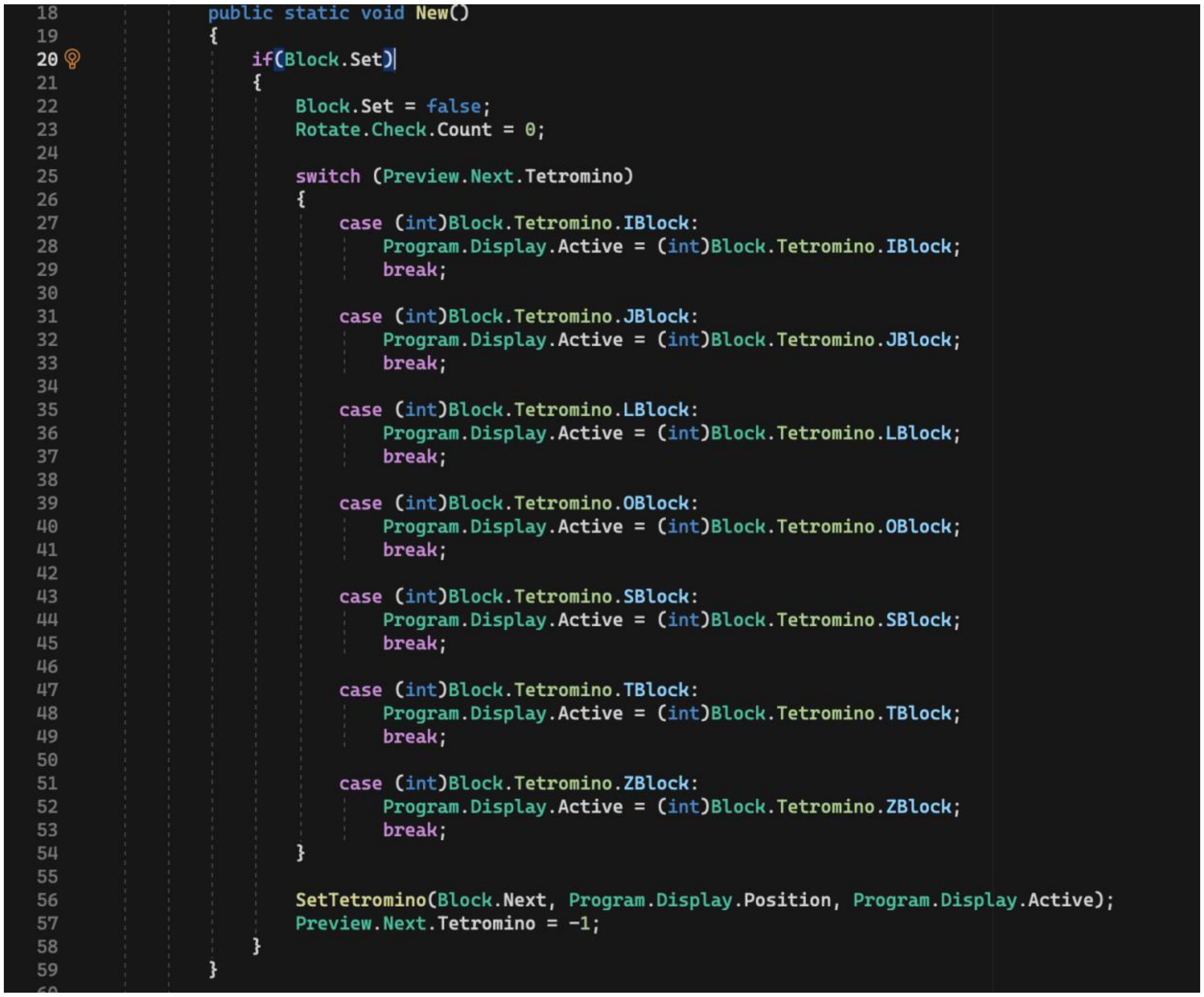
```
                Row.Cleared.AddRange(Tetrominos.Block.Placed.Except(Row.Complete).ToList());

                Tetrominos.Block.Placed.Clear();
                Tetrominos.Block.Placed.AddRange(Row.Cleared);

                Row.Cleared.Clear();

                for (var j = 0; j < Program.Display.Height; j++)
                {
                    for (var k = 0; k < Tetrominos.Block.Placed.Count; k++)
                    {
                        if (Row.Complete.Contains(Tetrominos.Block.Placed[k] + Program.Display.Width))
                        {
                            Tetrominos.Block.Placed[k] += Program.Display.Width;
                        }
                    }

                    for (var k = 0; k < Row.Complete.Count; k++)
                    {
                        Row.Complete[k] -= Program.Display.Width;
                    }
                }
            }
        }
        else
        {
            Row.Complete.Clear();
        }
    }

    Row.Complete.Clear();
}
```

4. **Now():**
    - **Classes:** Reset, Rotate, Set, and potentially others
    - **Description:** Executes specific actions like game resets, rotations, and settings management.



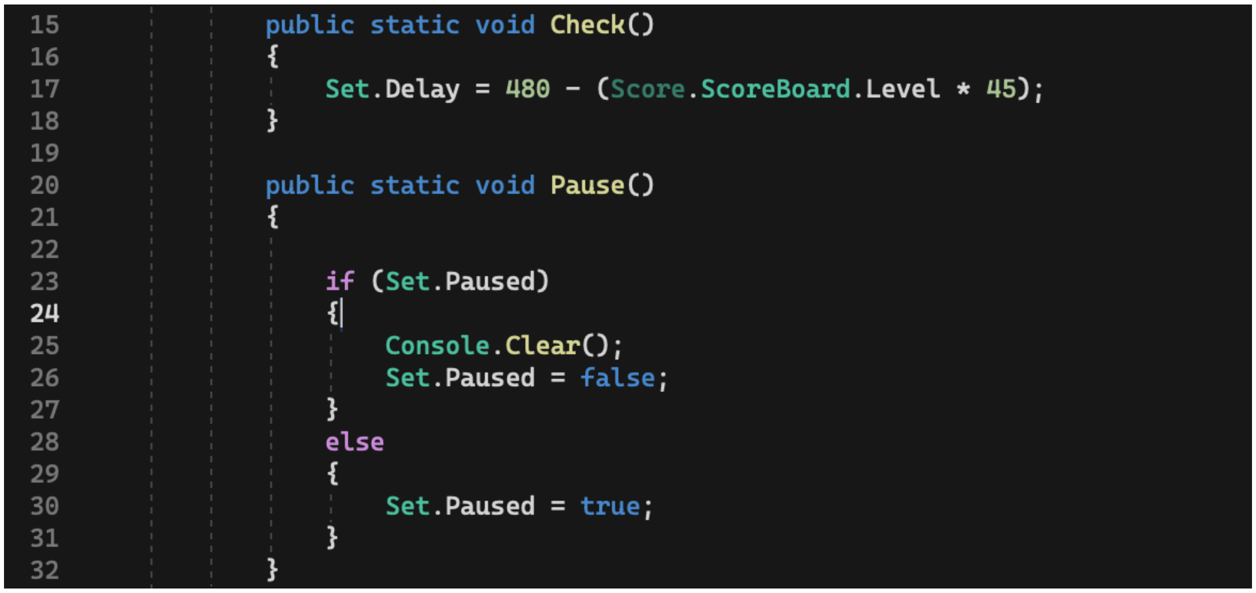
```
public static void Now()
{
    Program.Display.FrameChar.Clear();
    Program.Display.Intro.Clear();
    Program.Display.FrameString.Clear();
    Program.Display.DisplayFrame.Clear();
    Program.Display.Active = 0;

    Frame.Wall.ValList.Clear();
    Frame.Wall.Intro = true;

    Move.Check.Safe = false;

    Preview.Next.Tetromino = -1;
    Preview.Next.Draw.Clear();

    Rotate.Check.Count = 0;
    Rotate.Check.Lock = false;
    Rotate.Check.Next.Clear();

    Score.Row.Complete.Clear();
    Score.Row.Cleared.Clear();
    Score.Row.Counting = 0;
    Score.Row.Bonus = 0;

    Score.ScoreBoard.Rows= 0;
    Score.ScoreBoard.Level = 0;
    Score.ScoreBoard.Score = 0;

    Speed.Set.Delay = 480;
    Speed.Set.Paused = false;
    Speed.Set.Drop = false;

    Tetrominos.Block.Current.Clear();
    Tetrominos.Block.Next.Clear();
    Tetrominos.Block.Placed.Clear();
    Tetrominos.Block.Set = true;

}
```

5. **New():**
   - **Class:** Tetrominos
   - **Description:** Generates new tetrominos, crucial for gameplay advancement.



```
        public static void New()
        {
            if(Block.Set)
            {
                Block.Set = false;
                Rotate.Check.Count = 0;

                switch (Preview.Next.Tetromino)
                {
                    case (int)Block.Tetromino.IBlock:
                        Program.Display.Active = (int)Block.Tetromino.IBlock;
                        break;

                    case (int)Block.Tetromino.JBlock:
                        Program.Display.Active = (int)Block.Tetromino.JBlock;
                        break;

                    case (int)Block.Tetromino.LBlock:
                        Program.Display.Active = (int)Block.Tetromino.LBlock;
                        break;

                    case (int)Block.Tetromino.OBlock:
                        Program.Display.Active = (int)Block.Tetromino.OBlock;
                        break;

                    case (int)Block.Tetromino.SBlock:
                        Program.Display.Active = (int)Block.Tetromino.SBlock;
                        break;

                    case (int)Block.Tetromino.TBlock:
                        Program.Display.Active = (int)Block.Tetromino.TBlock;
                        break;

                    case (int)Block.Tetromino.ZBlock:
                        Program.Display.Active = (int)Block.Tetromino.ZBlock;
                        break;
                }

                SetTetromino(Block.Next, Program.Display.Position, Program.Display.Active);
                Preview.Next.Tetromino = -1;
            }
        }
```

6. **Check():**
   - **Class:** Speed
   - **Description:** Verifies speed conditions, potentially controlling game pace or tetromino falling speed.

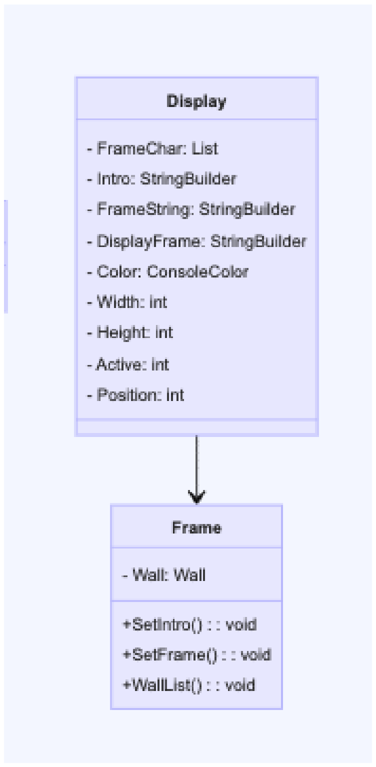


```
        public static void Check()
        {
            Set.Delay = 480 - (Score.ScoreBoard.Level * 45);
        }

        public static void Pause()
        {

            if (Set.Paused)
            {
                Console.Clear();
                Set.Paused = false;
            }
            else
            {
                Set.Paused = true;
            }
        }
```

d. Design features

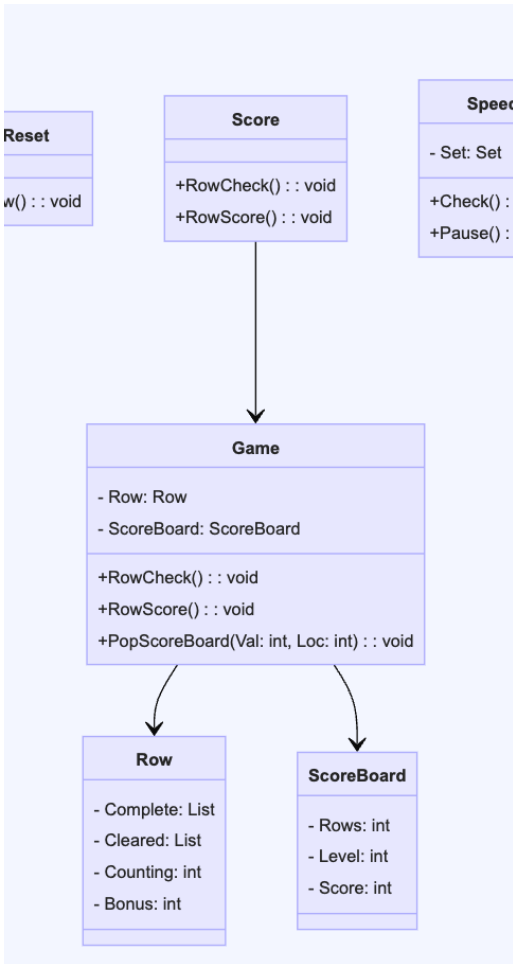
1. Frame Management:

- **Classes:** Frame, Display
- **Description:** Organizes and manages the game display, responsible for rendering and updating visual components.

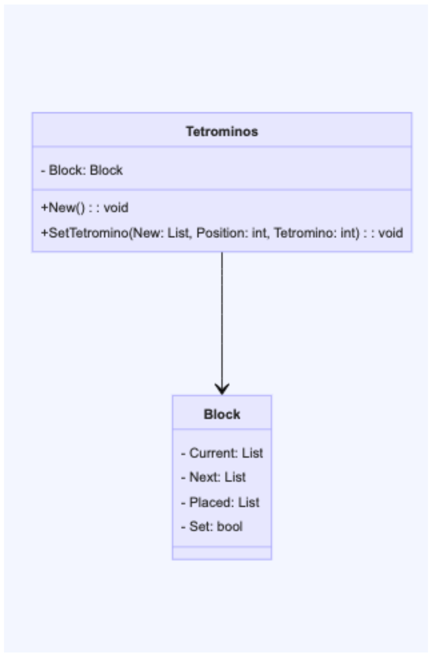


2. **Control Handling:**
   - **Class:** Key
   - **Description:** Allows user interaction via keyboard inputs, controlling various game aspects such as movement and rotation.

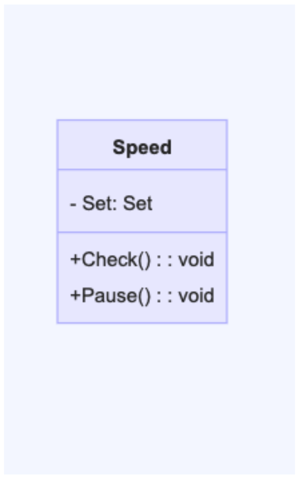
3. **Scoring Mechanism:**
   - **Classes:** Score, Game
   - **Description:** Manages and updates the player's score based on completed rows or other game achievements.



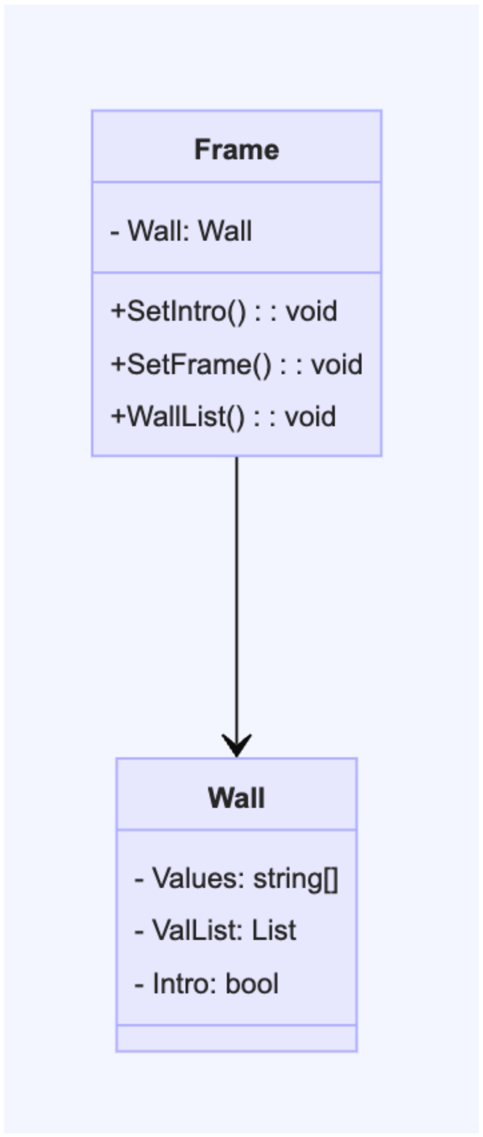
4. **Tetromino Manipulation:**
   - **Class:** Tetrominos
   - **Description:** Governs tetromino behavior including movement, rotation, and placement within the game grid.



5. **Game Speed Control**
    - **Class:** Speed
    - **Description:** Controls the pace or difficulty of the game, potentially affecting tetromino falling speed or other game dynamics.



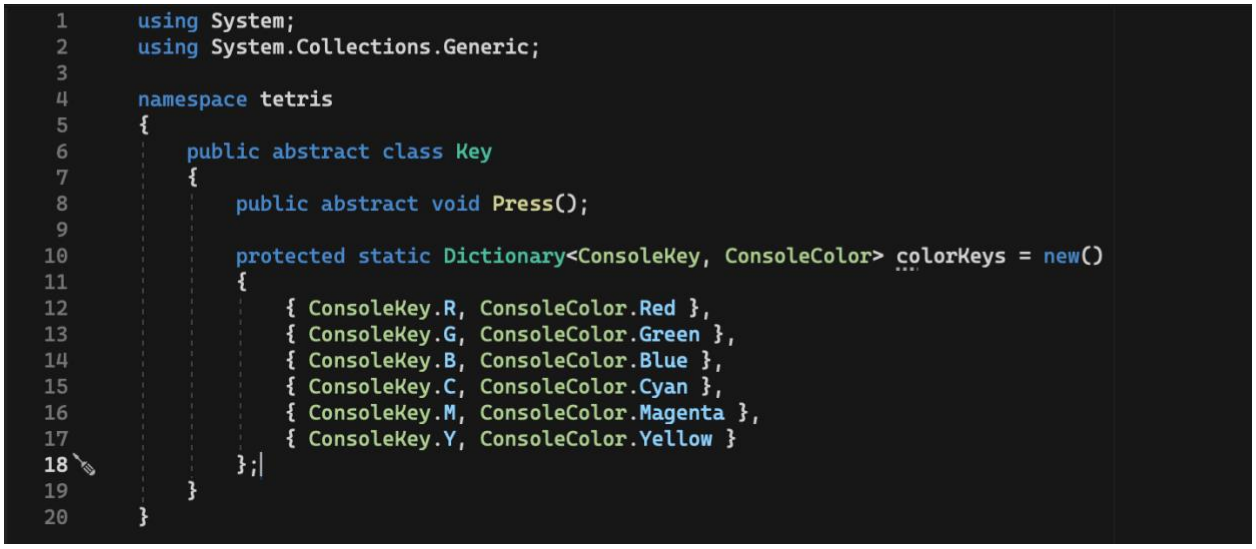
6. **Wall and Boundary Management**
    - **Classes:** Wall, Frame
    - **Description:** Ensures tetrominos adhere to game boundaries, preventing invalid movements or placements outside the game grid.



e. Concept of object-oriented programming

1. Abstraction

Abstraction is the concept of hiding the implementation details and showing only the necessary features of an object. In my code, the use of abstract classes signifies abstraction, particularly in Key.cs and Game.cs.

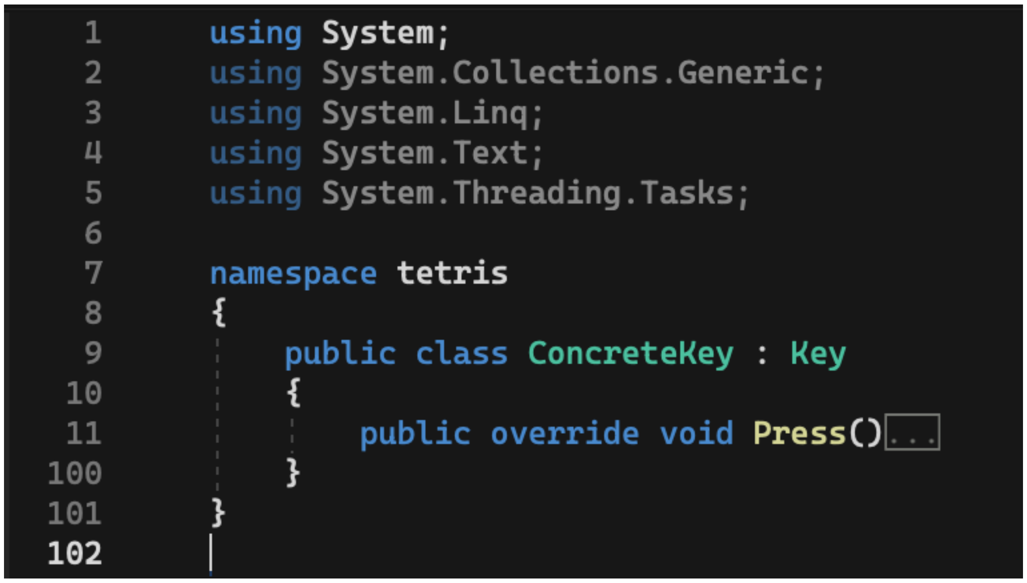


```
using System;
using System.Collections.Generic;

namespace tetris
{
    public abstract class Key
    {
        public abstract void Press();

        protected static Dictionary<ConsoleKey, ConsoleColor> colorKeys = new()
        {
            { ConsoleKey.R, ConsoleColor.Red },
            { ConsoleKey.G, ConsoleColor.Green },
            { ConsoleKey.B, ConsoleColor.Blue },
            { ConsoleKey.C, ConsoleColor.Cyan },
            { ConsoleKey.M, ConsoleColor.Magenta },
            { ConsoleKey.Y, ConsoleColor.Yellow }
        };
    }
}
```

Here, **Key** is an abstract class with an abstract method **Press()**. The **Key** class is intended to be a base class where specific key presses are handled in derived classes. The **Press()** method is defined as abstract, which means any class inheriting from **Key** must implement its own version of **Press()**.

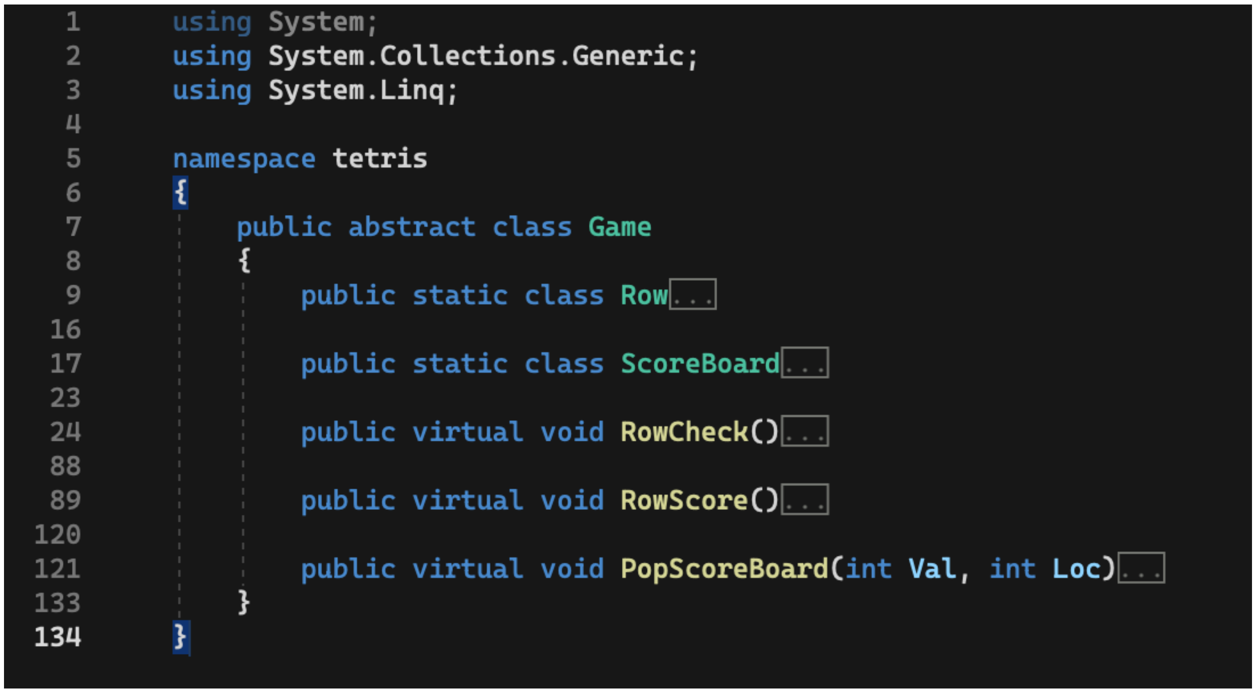
2. Inheritance

Inheritance is the mechanism where one class acquires the properties and behaviors of another class. You've used inheritance in several places, particularly in classes inheriting from the base class **Key**.



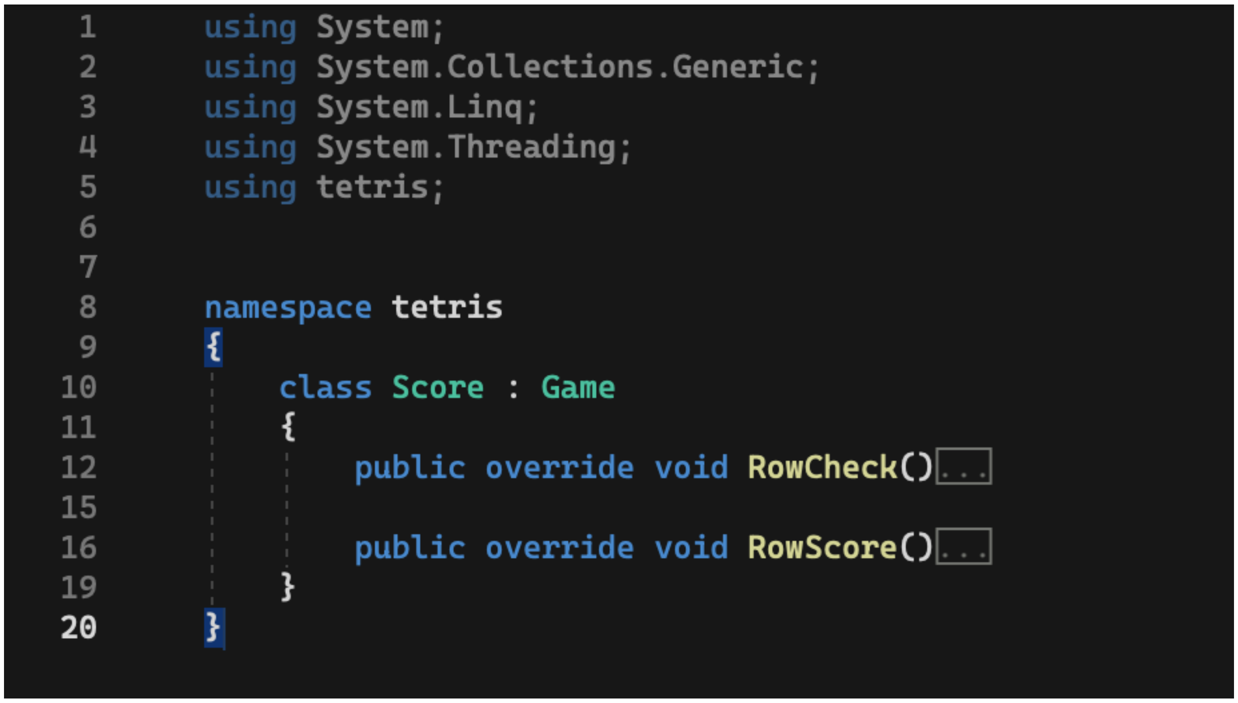
```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace tetris
{
    public class ConcreteKey : Key
    {
        public override void Press()...
    }
}

```

In **ConcreteKey.cs**, the class **ConcreteKey** inherits from the **Key** class using the **: Key** syntax. It then overrides the abstract **Press()** method, providing its own implementation for handling key presses.



```csharp
using System;
using System.Collections.Generic;
using System.Linq;

namespace tetris
{
    public abstract class Game
    {
        public static class Row...

        public static class ScoreBoard...

        public virtual void RowCheck()...

        public virtual void RowScore()...

        public virtual void PopScoreBoard(int Val, int Loc)...
    }
}
```

The **Game** class is an abstract class containing abstract and virtual methods. This allows subclasses to override these methods to provide their own specific functionality while inheriting the common features defined in the **Game** class. In this case:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using tetris;


namespace tetris
{
    class Score : Game
    {
        public override void RowCheck()[...]

        public override void RowScore()[...]
    }
}
```

IV. Conclusion

The custom program aligns with the objectives of showcasing a comprehensive understanding and application of object-oriented programming concepts. It exemplifies adherence to coding standards, effective use of UML diagrams, and successful program implementation that demonstrates the desired complexity and functionality.