

School of Science, Computing and Engineering Technologies

COS20007- Object Oriented Programming



D Level Custom Program : UML Design Level, Implementation, and User Documents

Project Title: Block Drop Challenge (Tetris Game)

Student Name: Truong Ngoc Gia Hieu

Student ID (Swinburne): 105565520

Student ID (Vietnam): SWS01217

Date: July 31th, 2025

Part A: Introduction

Brief Introduction

First and foremost, the report not only indicates the code design, structure, and UML class diagram for my custom program developed in the course Object-Oriented Programming (COS20007) but also showcases what Swinburne students have learned during three-year academic performance in the Experience Days (ExDays). Furthermore, the unit aims to introduce four important concepts in the Object-Oriented Programming (OOP) field including Abstraction, Encapsulation, Polymorphism, and Inheritance as reflected in the Unit learning Outcomes (ULOs). Notably, this course offers great opportunities for me to enhance my soft skills including complex problem-solving skill and logical thinking. In summary, participating in the course provides a comprehensive foundation in OOP, significantly boosting both theoretical knowledge and applying practical skills into my custom program.

Game Overview

Based on the requirements for a Distinction level, I have decided to utilize four different concepts in my custom program that showcase my understanding and well-structured design. While the custom program features minor real-world relevance, I believe that creating the "Tetris Game" is an interesting project for a beginner. It will allow me to enhance my problem-solving skills and critical thinking while gaining a deeper understanding of programming concepts such as event handling, collision detection, and user input management. Additionally, the custom program features a friendly UI design, elements, attributes, constructors, methods, important classes, well-organized and clear UML class diagram, and complicated gameflow by using OOP principles. In summary, this practice ensures that each class in the game is clear and manageable, making the game maintainable, easy to understand and update.

Part B: Object-Oriented Programming (OOP) Principles Application

In the context of game development, the Unit Learning Outcomes (ULOs) for Object-Oriented Programming (COS20007) have been directly addressed through my custom program. Notably, this program demonstrates proficiency in several key areas:

- **Explaining OOP Principles:** I have effectively applied core concepts such as Encapsulation, Abstraction, Polymorphism, and Inheritance to enhance the design and functionality of my game
- **Developing OOP Programs:** Utilizing the C# programming language, I created a functional game that leverages object-oriented techniques to manage game entities and behaviors.
- **Designing, Developing, Testing, and Debugging:** I employed OOP principles within an integrated development environment to build a robust solution, ensuring that the game operates smoothly and efficiently.
- **Communicating Solutions:** I constructed appropriate diagrams and

textual descriptions, including a UML Class Diagram in Part C, to elucidate the static structure and dynamic behavior of the object-oriented design

- **Reflecting on Good Practices:** I have described and explained factors that contribute to a good object-oriented solution, drawing upon my practical experience in the game development.

This foundation sets the stage for a deeper exploration of how OOP principles are applied throughout the game workflow, enhancing both the development process and the final product.

2.1 Abstraction

Abstraction is used to define a common interface for all types of blocks without getting into the specific details of each one. Specifically, I use an abstract class (***Block.cs***) to define important behaviors, including movement and rotation, as well as abstract properties (***Id***, ***Tiles***, and ***StartOffset***). Furthermore, the following elements showcase what every concrete block class must implement, allowing the game logic to control all blocks in the same way, regardless of their specific shapes. By integrating with the abstract ***Block*** class and Abstraction principle, I believe that other parts of the game, like ***GameState***, can work with any shape without needing to know its characteristics. In summary, the abstract properties and methods are a contract that all concrete block types must follow.

```
public abstract class Block
{
    //Two dimensional position array which contains tile positions in four rotation states-//
    10 references
    protected abstract Position[][] Tiles { get; }
    //A start offset which decides where the block spawns in the grid
    11 references
    protected abstract Position StartOffset { get; }
    //Id of blocks-//
    14 references
    public abstract int Id { get; }
    //Stores the current rotationstate and current offset-//
    private int rotationState;
    private Position offset;
```

2.2 Inheritance

In my custom program (Tetris game), I utilize the Inheritance principle to generate distinct block types including ***IBlock***, ***JBlock***, ***ZBlock***, ***TBlock***, ***SBlock***, ***OBlock***, and ***ZBlock*** classes which inherit from the abstract ***Block*** class. Specifically, this approach means automatically get all the methods and properties defined in ***Block*** class, and they are required to provide their own implementation for the abstract properties, defining their unique ***shape***, ***Id***, and ***starting position***.

```
public class SBlock : Block
{
    private readonly Position[][] tiles = new Position[][]
    {
        new Position[] { new(0,1), new(0, 2), new(1, 0), new(1, 1), },
        new Position[] { new(0,1), new(1, 1), new(1, 2), new(2, 2), },
        new Position[] { new(1,1), new(1, 2), new(2, 0), new(2, 1), },
        new Position[] { new(0,0), new(1, 0), new(1, 1), new(2, 1), }
    };

    8 references
    public override int Id => 5;

    5 references
    protected override Position StartOffset => new Position(0, 3);

    4 references
    protected override Position[][] Tiles => tiles;
}
```

2.3 Encapsulation

Encapsulation is one of the important OOP principles and also builds data and methods operate on data into a single unit (a class) and controlling access to that data. In simple terms, encapsulation is used to protect the internal state of key game components. For instance, the GameGrid class's core data is "**private readonly int[,] grid**". In details, to interact with this array, other classes cannot access it directly. Instead, they must use the public indexer **this[int r, int c]** or methods like ClearFullRows(). As a result, this practice ensures that the grid data is always manipulated in a controlled and predictable way, preventing accidental corruption of the game board.

```
5 references
public class GameGrid //--Change from internal to public--
{
    private readonly int[,] grid;
    7 references
    public int Rows { get; } //--First diemnsion is row--
    9 references
    public int Columns { get; } //--Second dimension is column--
    //--The indexer below allows to easy access to the array--
    //--The indexer directly on a gamegrid object--
    2 references
    public int this[int r, int c]
    {
        get => grid[r, c];
        set => grid[r, c] = value;
    }
}
```

2.4 Polymorphism

Polymorphism allows objects of different classes that are related by inherit to be treated as objects of a common base class. Moreover, the meaning of “Polymorphism” means “many forms” that makes my game more logic, UI user-friendly, and well-structured code. In details, the **BlockQueue** and **GameState** classes are designed to work with objects of type **Block**, but at runtime, these objects can be any of the specific inherited block types (**IBlock**, **JBlock**, **LBlock**, etc.). Additionally, when calling a method like **CurrentBlock.RotateCW()**, the system automatically invokes the correct rotation logic for the specific block instance currently stored in **CurrentBlock**, allowing to write generic, flexible code that can handle all block types without complex conditional statements.

```
3 references
public class GameState
{
    //-Properties-//
    private Block currentBlock;

    23 references
    public Block CurrentBlock
    {
        get => currentBlock;
        private set
        {
            currentBlock = value;
            currentBlock.Reset();

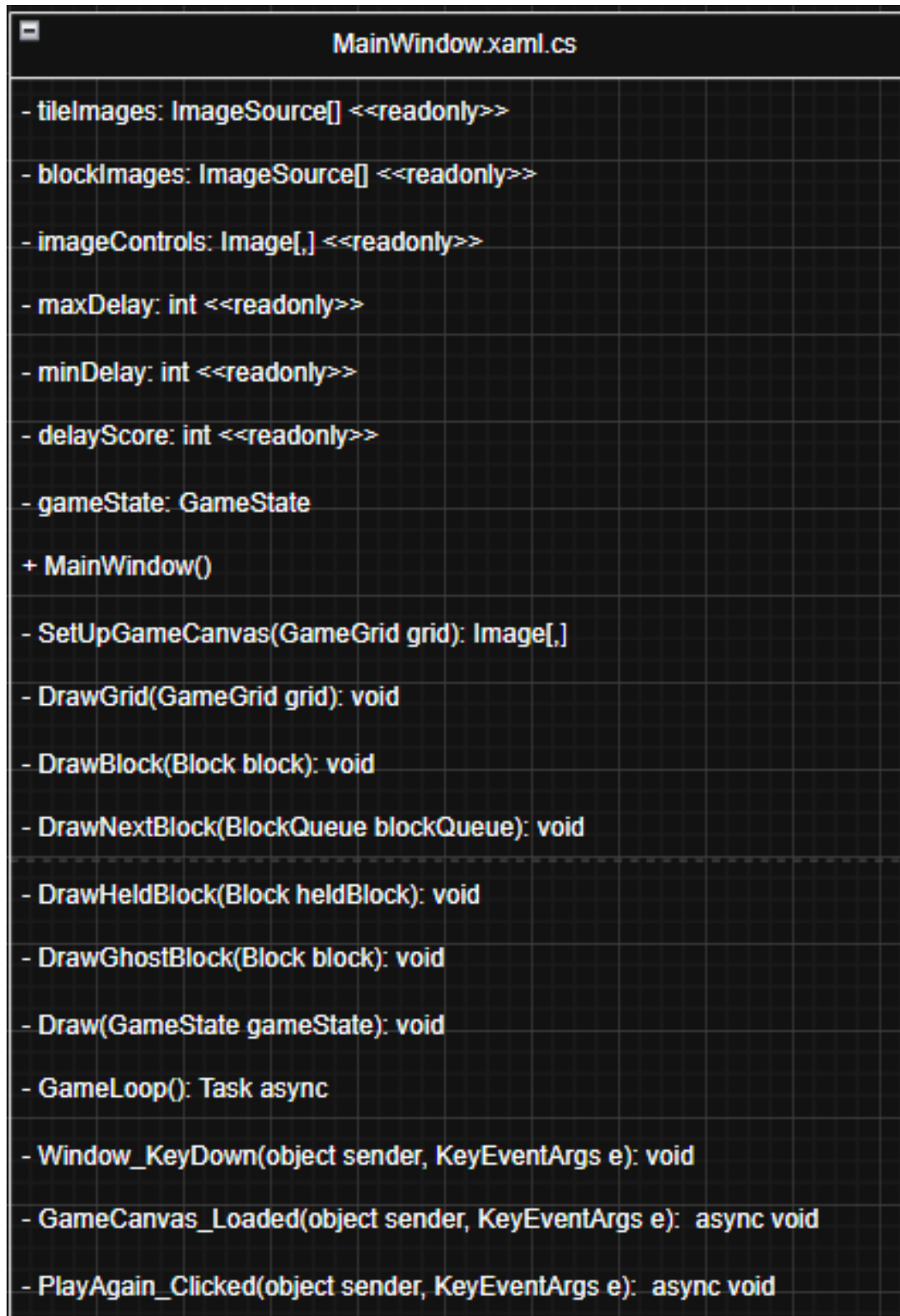
            for (int i = 0; i < 2; i++)
            {
                currentBlock.Move(1, 0);

                if (!BlockFits())
                {
                    currentBlock.Move(-1, 0);
                }
            }
        }
    }
}
```


Part C: Classes and Responsibilities

Classes	Responsibility
GameState	<ul style="list-style-type: none">- The brain of the game- Manages the entire game state , including the current block, the player's score, and whether the game is over.- Indicates all game rules and interactions.
GameGrid	<ul style="list-style-type: none">- This is the playing field- Displays 2D grid where the blocks fall and are stored.- Handles the core mechanics of checking for and clearing completed rows.
Block	<ul style="list-style-type: none">- An abstract class base for all block types- Defines the common attributes and behaviors for any Tetris piece including its shape, Id, and methods for movement and rotation.
IBlock	Class for the block I-Shaped
JBlock	Class for the block J-Shaped
ZBlock	Class for the block Z-Shaped
TBlock	Class for the block T-Shaped
OBlock	Class for the block O-Shaped
SBlock	Class for the block S-Shaped
LBlock	Class for the block L-Shaped
BlockQueue	<ul style="list-style-type: none">- The block generator- Responsible for generating the next block to be used in the game, ensuring a random but fair sequence.
Position	<ul style="list-style-type: none">- A simple utility class.- This class is used to store/manipulate the (row, column) coordinates of a tile on the game grid.

Part C: UML Class Diagram



```
MainWindow.xaml.cs

- tileImages: ImageSource[] <<readonly>>
- blockImages: ImageSource[] <<readonly>>
- imageControls: Image[,] <<readonly>>
- maxDelay: int <<readonly>>
- minDelay: int <<readonly>>
- delayScore: int <<readonly>>
- gameState: GameState
+ MainWindow()

- SetUpGameCanvas(GameGrid grid): Image[,]
- DrawGrid(GameGrid grid): void
- DrawBlock(Block block): void
- DrawNextBlock(BlockQueue blockQueue): void
- DrawHeldBlock(Block heldBlock): void
- DrawGhostBlock(Block block): void
- Draw(GameState gameState): void
- GameLoop(): Task async
- Window_KeyDown(object sender, KeyEventArgs e): void
- GameCanvas_Loaded(object sender, KeyEventArgs e): async void
- PlayAgain_Clicked(object sender, KeyEventArgs e): async void
```

