

Name: Trung Kien Nguyen

Student ID: 104053642



COS20007

Object-Oriented Programming

Custom Program – Gibbous Tetris



TABLE OF CONTENTS

<i>Table of contents.....</i>	<i>2</i>
<i>Introduction.....</i>	<i>3</i>
<i>Overview of the program.....</i>	<i>3</i>
<i>Some program's sequences.....</i>	<i>6</i>
<i>Design patterns' using.....</i>	<i>9</i>
<i>Conclusion.....</i>	<i>10</i>
<i>References.....</i>	<i>10</i>

INTRODUCTION

Gibbous Tetris (using C# and SplashKit library)

TRUNG KIEN NGUYEN – 104053642

Bachelor of Computer Science – AI major

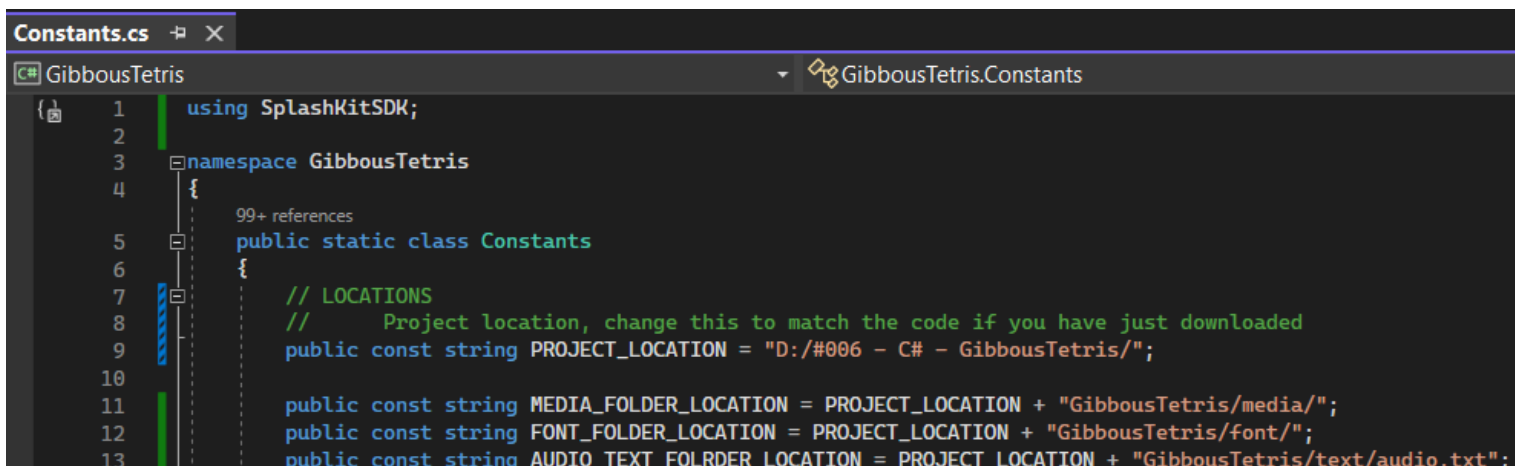
Swinburne University of Technology – Hawthorn Campus – Semester 1 – Year 2023

This program is the Custom Program for the Unit COS20007 – Object Oriented Programming. Based on the original Tetris game developed by Atari Games and published by Atari and Tengen in 1988 and 1989, my program incorporates object-oriented programming principles to create a modular and extensible program. I represented the game's elements, such as the game board, tetromino shapes, scoring mechanism, and user input handling, through the use of classes and objects, providing an engaging and interesting gaming experience, while ensuring the quality of the program according to the OOP's principle.

OVERVIEW OF THE PROGRAM

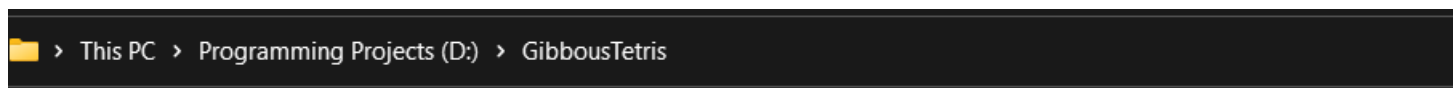
The program is written in C# programming language and the SplashKit library. It includes a total of 25 C# scripts files in the “script” folder, 24 png image files, 6 mp3 music files, and 4 wav sound effect files in the “media” folder, 6 otf and ttf font file for text drawing, and 4 txt text files for saving and loading.

To run this program correctly, after downloading and extracting the file “GibbousTetris.zip”, it is required to **change the path leading to the Folder "GibbousTetris" in the Script "Constants.cs"** (the public constant PROJECT_LOCATION),



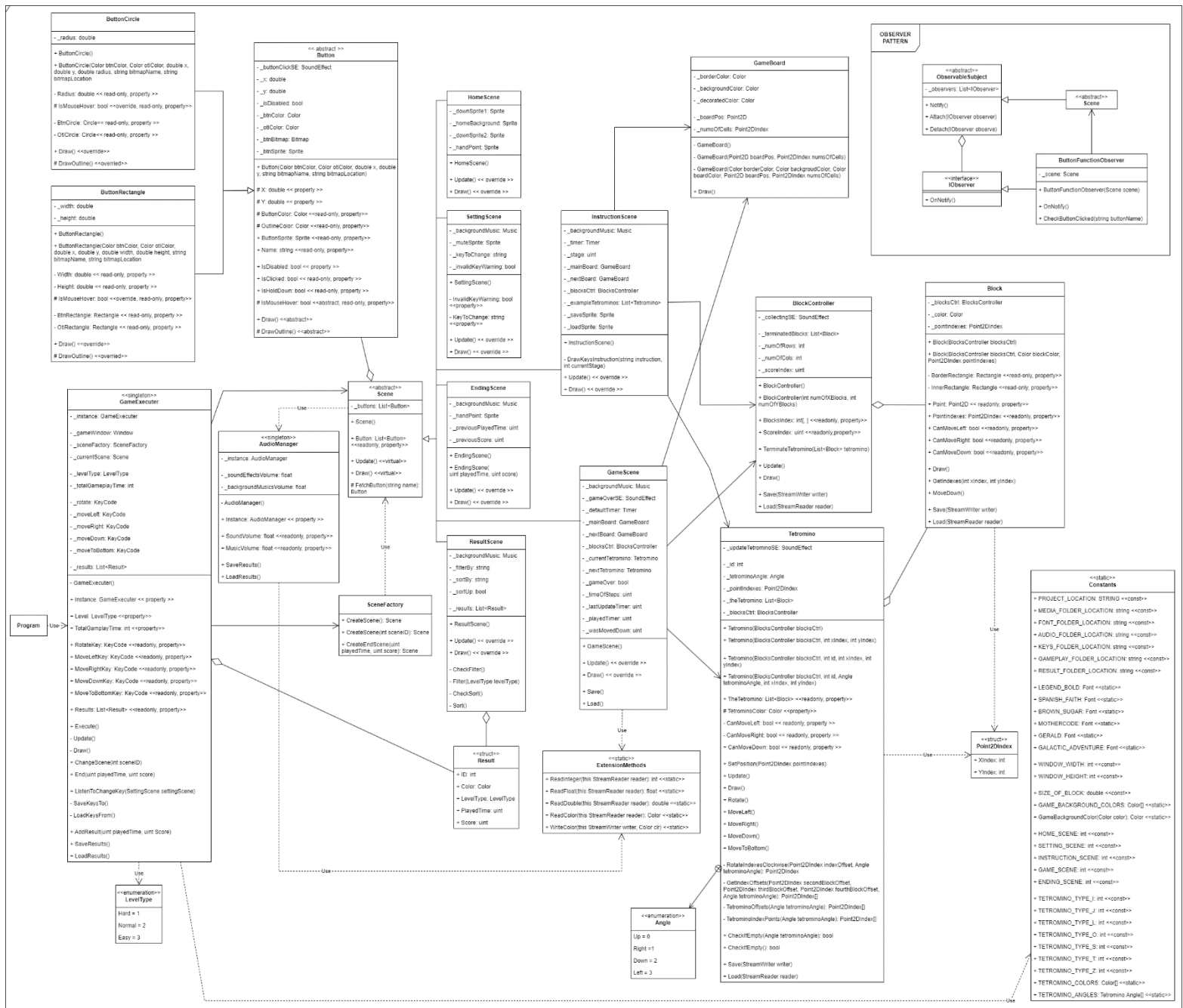
```
1 using SplashKitSDK;
2
3 namespace GibbousTetris
4 {
5     99+ references
6     public static class Constants
7     {
8         // LOCATIONS
9         // Project location, change this to match the code if you have just downloaded
10        public const string PROJECT_LOCATION = "D:/#006 - C# - GibbousTetris/";
11
12        public const string MEDIA_FOLDER_LOCATION = PROJECT_LOCATION + "GibbousTetris/media/";
13        public const string FONT_FOLDER_LOCATION = PROJECT_LOCATION + "GibbousTetris/font/";
14        public const string AUDIO_TEXT_FOLDER_LOCATION = PROJECT_LOCATION + "GibbousTetris/text/audio.txt";
```

or simply place the extracted project folder directly in the D: disk:



(This process is to make sure the program can find the needed media, font and text files to run)

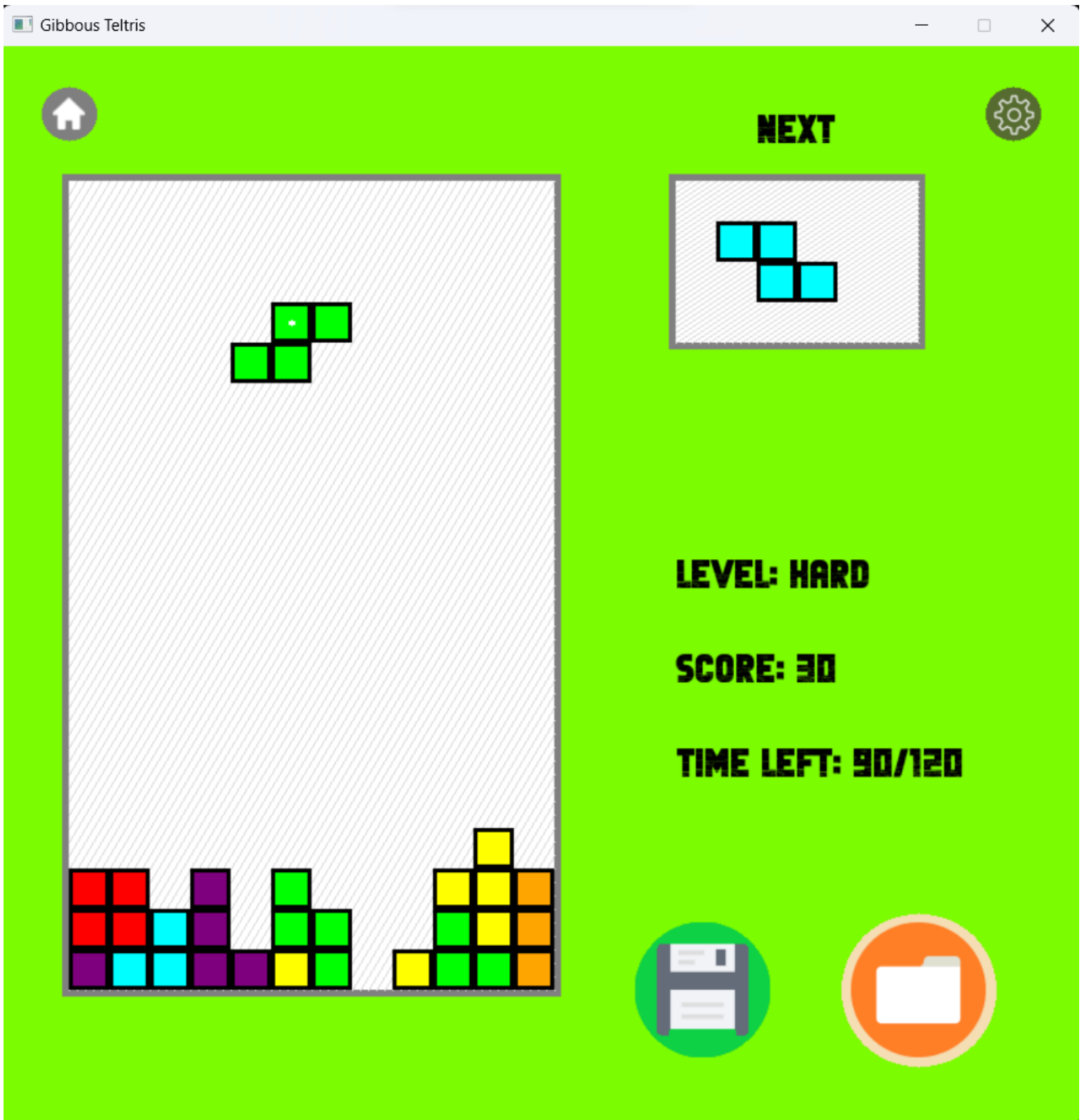
Below is the UML Class Diagram of my Custom Program:



The game has a total of 6 scenes, executing by the **GameExecuter**, including:

- **HomeScene**: To choose the level (Easy/Normal/Hard), total gameplay time (1-600 secs). This also has buttons to go to other scenes, including **GameScene**.
- **SettingScene**: To modify the volume of background music and sound effects. You can also change the function keys used in the **GameScene** to manipulate the tetromino (rotate, move)
- **InstructionScene**: An overall tutorial of the game step by step.
- **GameScene**: This is where the player enjoys the game, it includes a main gameplay board, a sidebar for the next tetromino, displays of level type, current scores, total and remaining times. When the game is over, it takes you to the **EndingScene**, while your play is recorded.
 - The current tetromino in the mainboard is automatically moved down by 1 unit after each 3, 2 or 1 second, depending on the level type.
 - Player uses Up Key, Left Key, Right Key, Down Key or Spacebar (or alternative keys as changed in **SettingScene**) to rotate, or move the current tetromino respectively.

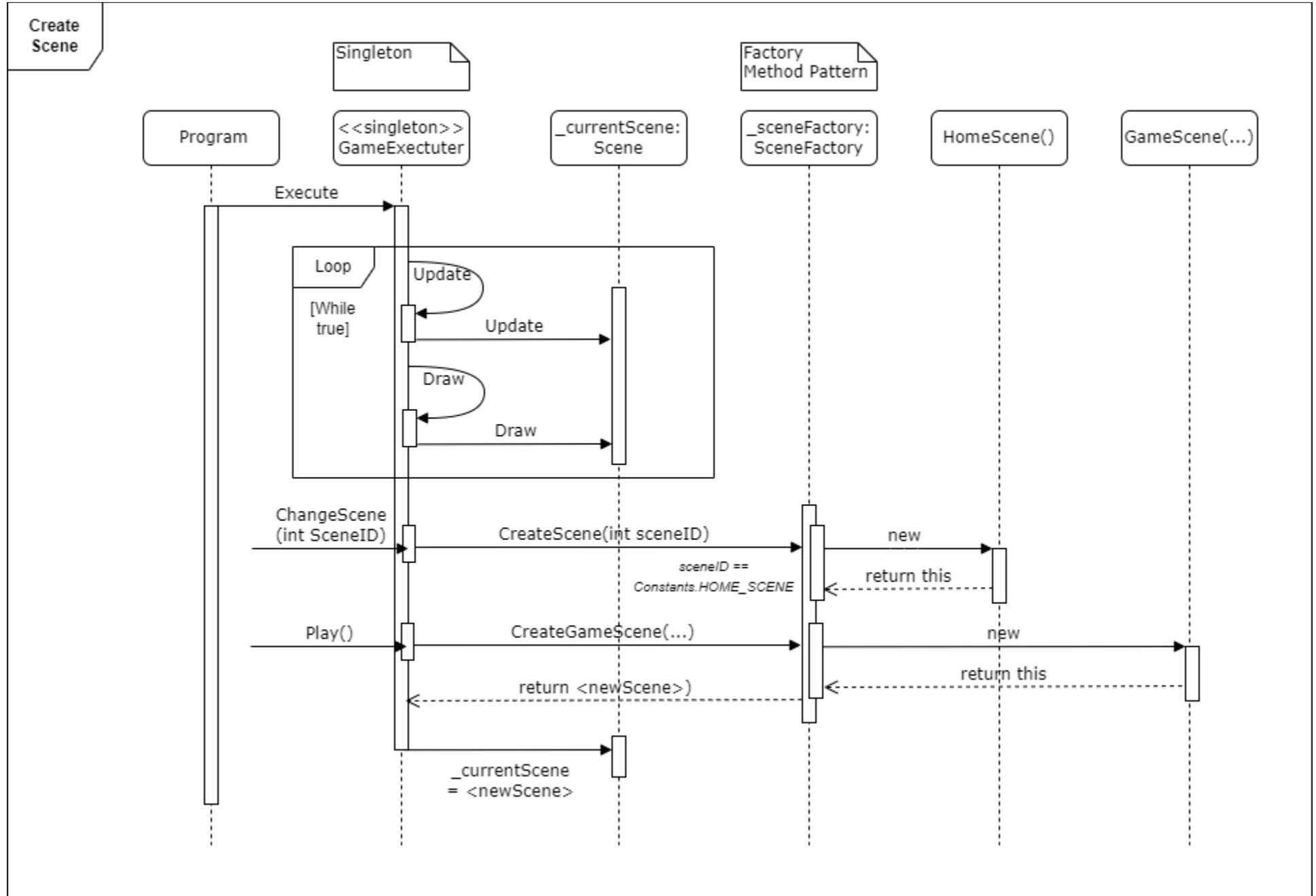
- When the current tetromino cannot move down anymore, it is terminated and the next one is added to the mainboard.
- When a line of terminated blocks is complete, the player earns some points. The number of earned points depends on the level type (the time of each step) and the total time of the game (for each line, the player's score increases by $(3 / \text{TimeOfSteps}) * (600 / \text{TotalGameplayTime}))$)
- In this screen, player can save the current play, or load the saved play anytime.



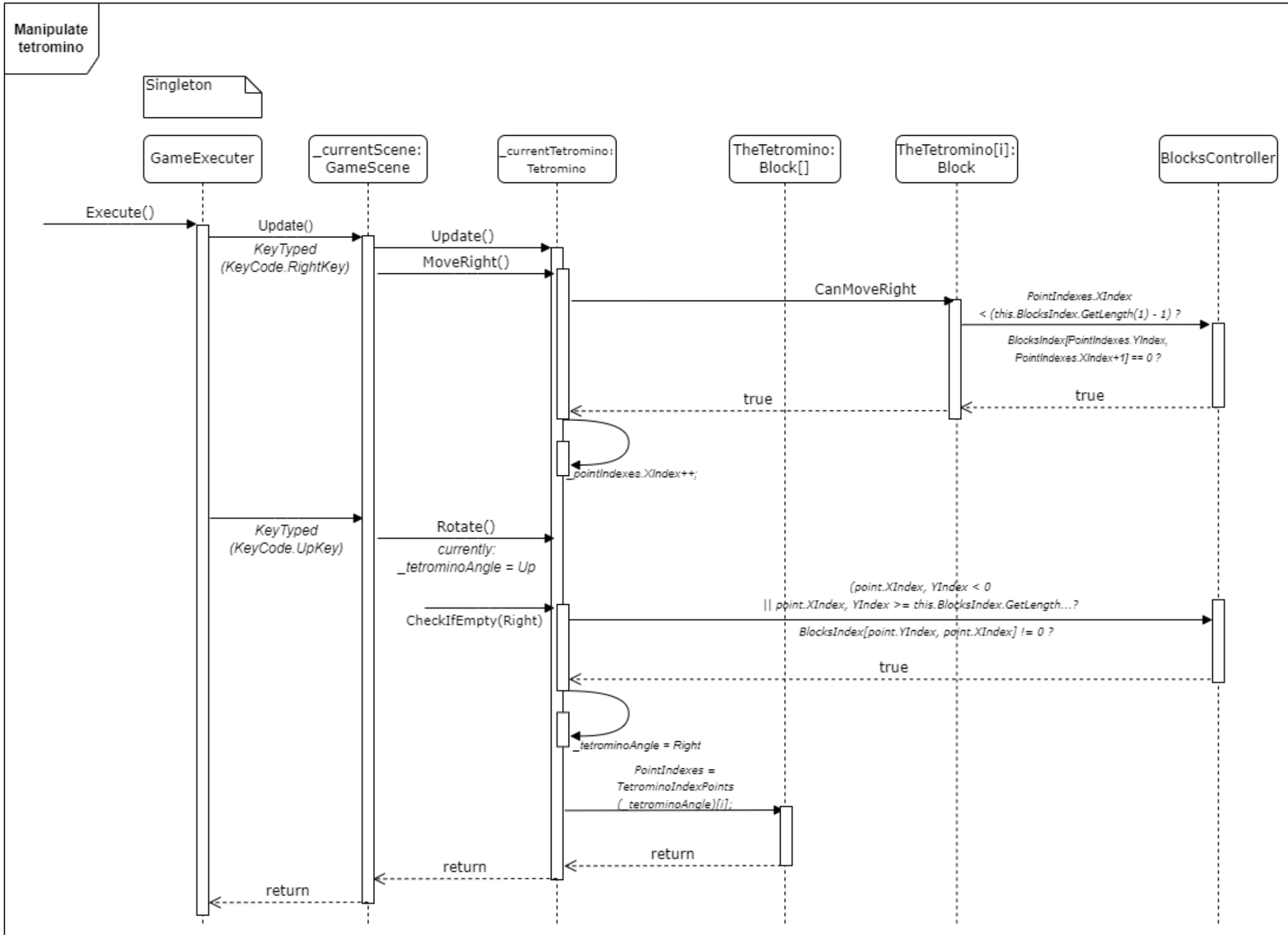
- **EndingScene:** Just to show your effort, with total score, play time and related stuff. From there you can choose to go to Home, replay or view all your previous plays.
- **ResultScene:** Show all your 11 latest plays' information, including Level Type, Score, Played Time and Total Gameplay Time. You can manually sort and filter the data as you want.

SOME PROGRAM SEQUENCES

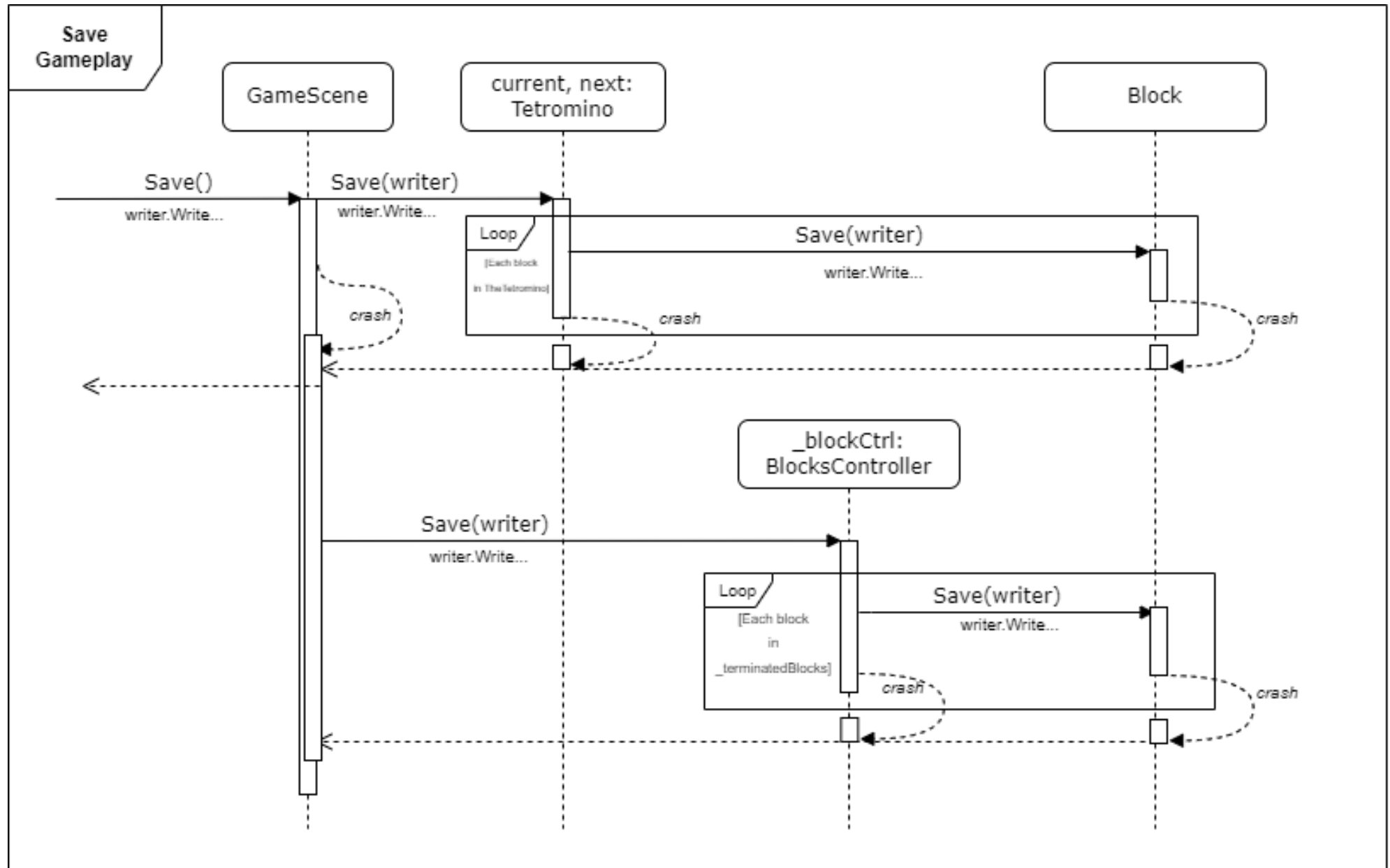
Creating Scene Process:



Manipulating Tetromino Process:

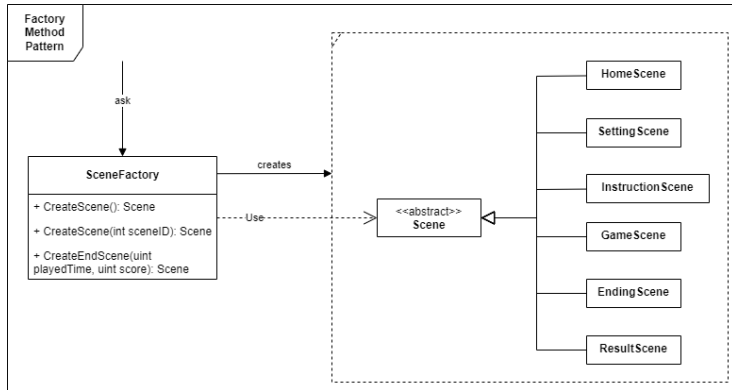


Save a Gameplay Process:



DESIGN PATTERNS' USING

1. Factory Method Pattern:



In the program, I have designed that 6 different scene classes are inherited from the abstract class `Scene`. Normally, when I want to create any scene according to a specific condition, I would write a statement of “if...else if...else”, or “switch...case...”. This will be fine if I only need to create in one place in my codebase, however, if the program requires creating such Scenes in many different classes, there is a duplication code issue. Then, every time I need to update, for example to add a new scene type, along with its creation condition, I would have to fix it all over the place in the codebase. That is why I need to encapsulate the object creation logic in one or more separate method, called the factory method(s).

The `SceneFactory` class has two main factory methods: `CreateScene` (two times of overloading) and `CreateEndScene`.

Then I can create different types of scenes in my program, such as in the `GameExecuter`:

```
5 references
public void ChangeScene(int sceneID)
{
    _currentScene = _sceneFactory.CreateScene(sceneID);
}

1 reference
public void End(uint playedTime, uint score)
{
    _currentScene = _sceneFactory.CreateEndScene(playedTime, score);
}
```

2. State pattern

This design pattern is also presented in this scenes model. Specifically, the `GameExecuter` plays the role of “context” class that executes the program's scene. The “states” are represented by separate scene classes that are inherited from the abstract class `Scene`. The `GameExecuter` class has the methods `Update` and `Draw` which delegate the behavior to the current scene.

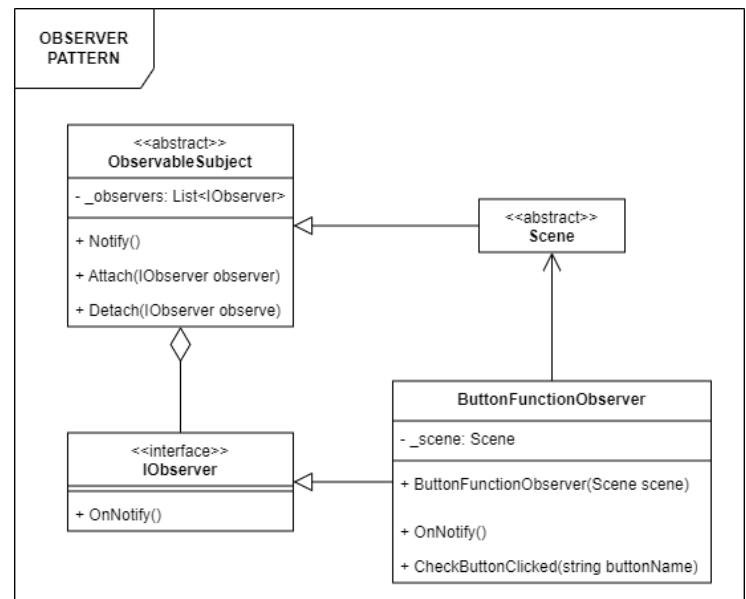
Initially, the `GameExecuter` object's current scene is the home one. When the method `Update` or `Draw` is called, it delegates the behavior to the current scene (`HomeScene`), telling it to `Update` or `Draw` respectively. Also, we can use the methods

`ChangeScene` or `End` to switch between scenes at runtime. By using the State pattern, I can easily add new scenes without modifying `GameExecuter`'s code, but to create new scenes inherited from the abstract class `Scene`.

3. Observer pattern

Some scenes have the same function buttons (Home button to return home, play and replay buttons to change to the `GameScene`). To check if these buttons is clicked, and implement their functions, it is not recommended to bring that conditions code to the `Update` method of each scene, as it will increase the code duplication and coupling. Instead, I implemented the `IObserver` interface to the `ButtonFunctionObserver` class, which registers itself as an observer of a specific scene by calling `scene.Attach(this)` in its constructor. When the subject (`Scene`) calls the `Notify` method, it triggers the `OnNotify` method in each observer (`ButtonFunctionObserver`). In the `OnNotify` method, the observer performs specific actions based on the state of the buttons in the scene. For example, it checks if a button is held down or clicked and executes the corresponding logic.

By using this pattern, the code achieves loose coupling between the scene and the button functionality. The scene doesn't need to have direct knowledge of the observers or their specific actions. In the future, new observers can be added or existing ones can be removed without affecting the scene or other observers. This promotes modularity, reusability, and easier maintenance of the codebase.



CONCLUSION

This program is one of my first 2D games based on OOP principles, so I believe it is very simple and has many shortcomings. In the near future, specifically the next related units, I will consider trying out some more complex and advanced areas of OOP.

Thank you for reading these words.

If you have any questions or comments about this program, including errors or misunderstandings, please let me know by emailing 104053642@student.swin.edu.au.

REFERENCES

[1] Splashkit Documents. Retrieved from

<https://splashkit.io/api/windows/>

[2] ChatGPT. Retrieved from <https://chat.openai.com/>

[3] Factory Method Pattern – Giải thích đơn giản, dễ hiểu.

Retrieved from <https://topdev.vn/blog/factory-method-pattern-giai-thich-don-gian-de-hieu/>

[3] Game programming patterns with C# - Observer Pattern.

Retrieved from

<https://www.habrador.com/tutorials/programming-patterns/3-observer-pattern/>

[4] Tetris game (which my program is based on). Retrived

from <https://www.freetetris.org/game.php>