Refactoring Documentation for Project “Labyrinth-7”

Team “Labyrinth-7”

1. Redesign of the project structure:

   1.1. Adding GameHandler project which according to the current game state handles the initialization (start, restart), the update, the drawing or termination of any game that implements the IGame interface:

      1.1.1. Creating GameState enumeration;
      1.1.2. Defining IGame interface;
      1.1.3. Initialization of the DrawEngine.

   1.2. Adding all other classes to a new project LabyrinthGameEngine.
   1.3. Renaming the class Table -> Player.
   1.4. Renaming the class Game -> Labyrinth.
   1.5. Renaming the class Labyrinth -> LabyrinthGame.
   1.6. Moving Main method to a new static class GameEntry.
   1.7. Extracting logic for Top 5 players in a new Singleton class RankingTopPlayers.
   1.8. Adding a Labyrinth Factory class which hides the logic for creating and testing the Labyrinth.

2. Reformatting of the source code:

   2.1. Removing all unnecessary empty lines in files Labyrinth.cs and Game.cs:

      2.1.1. between fields declaration, e.g.:

```
public static bool flag;

public static bool flag2;

public static bool flag3;

public static bool flag4;
```

```
public static bool flag;
public static bool flag2;
public static bool flag3;
public static bool flag4;
```

      2.1.2. after opening brackets and before closing brackets of body parts, e.g.:

```
static void Main(string[] args)
{




        positionX = positionY = 3;
// player position
```

```
static void Main(string[] args)
{
        positionX = positionY = 3;
// player position
```

2.1.3. between mutually connected by logic parts of the code, e.g.:

```
case "top":

        Table_(scores);
        Console.WriteLine("\n");
        DisplayLabyrinth(labyrinth);

        break;
```

```
case "top":
        Table_(scores);
        Console.WriteLine("\n");
        DisplayLabyrinth(labyrinth);
        break;
```

2.1.4. multiple empty lines.

2.2. Removing unnecessary comments (relying on self-documentation), e.g.:

```
// bachka!! yesss
```

```
//used for adding score only when game is finished naturally and not by the restart command.
```

2.3. Moving usings statements after namespace declaration:

```
namespace LabyrinthGameEngine
  {
        using System;
        using System.Collections.Generic;
        using System.Linq;
```

2.4. Split the lines containing several statements into several simple lines:

```
if (scores.Count == 0) { Console.WriteLine("The scoreboard is empty! "); }
```

// TO DO

3. Renaming variables:

In Game.cs:	flag -> isLabyrinthValid
	flag2 -> isInLabyrinth
	flag3 -> isPlaying
	flag4 -> isEscapedNaturally
In Labyrinth.cs	flag_temp -> isInLabyrinth

4. Applying the DRY principle when extracting all common logic from method Main(string[] args) concerning the change of the game state when the player escapes from the labyrinth:

```
private static bool SuccessfulEscape(int totalMoves)
{
	Console.WriteLine("\nCongratulations you escaped with {0} moves.\n", totalMoves);
	bool IsInLabyrinth = false;
	isEscapedNaturally = true;
	return IsInLabyrinth;
}
```

5. Add access modifiers to all classes and methods.

6. Implementing Game Handler:

6.1. Removing wrong inheritance of class Game *(new Labyrinth)* by class Program *(new LabyrinthGame)* and changing access modifiers from protected to public.

```
class Program : Game
```

```
public class LabyrinthGame : IGame
```

6.2. LabyrinthGame inherits and implements Interface IGame from GameHandler class library. Implements game state.

6.3. Move Main() method's content temporarily to LabyrinthGame constructor to make sure everything continues to work after the Game Handler implementation.

6.4. Move Main() method to newly created GameEntry class, declare the user interface and run the **game handler'**s class Game (passing LabyrinthGame type):

```
public static class GameEntry
{
	public static void Main()
	{
		Type gameType = new LabyrinthGame().GetType();
		UserInterface userInterface = UserInterface.Console;

		Game.Instance.Run(gameType, userInterface);
	}
}
```

6. Moving fields declaration from Labyrinth to LabyrinthGame class where they are actually used and refactor accordingly.

7. Creating IPlayer Interface and implementing it by class Player. Modifying Player:

7.1. Changing public fields to private and exposing them using public properties. Changing references. Adding default values of fields:

(in LabyrinthGame.cs)

```
public static int positionX
public static int positionY
```

```
this.Player.PositionX
this.Player.PositionY
```

(in Player.cs)

```
private int moves = 0;

public int Moves
{
        get
        {
                return this.moves;
        }
        set
        {
                this.moves = value;
        }
}
```

7.5. Creating an instance of class Player in LabyrinthGame's constructor passing as arguments initial x and y position of Player (in the middle of the labyrinth):

```
public LabyrinthGame()
{
        ...
        int initialPlayerPositionX = labyrinthCols / 2;
        int initialPlayerPositionY = labyrinthRows / 2;

        Player = new Player(initialPlayerPositionX, initialPlayerPositionY);
        ...
}
```

8. Introduced constants:

LABYRINTH_ROWS = 7

LABYRINTH_COLS = 7

NUMBER_OF_TOP_PLAYERS = 5;

9. Total refactoring of SolutionChecker method (test for exit from the labyrinth):
    9.1. Renamed as CheckIfAnyExit and placed in LabyrinthFactory class.
    9.2. Usas recursion to check all possible ways out of the labyrinth. Receives a deep copy of the current labyrinth as an argument (which is created by the class ObjectCopier and Labyrinth class serialization).