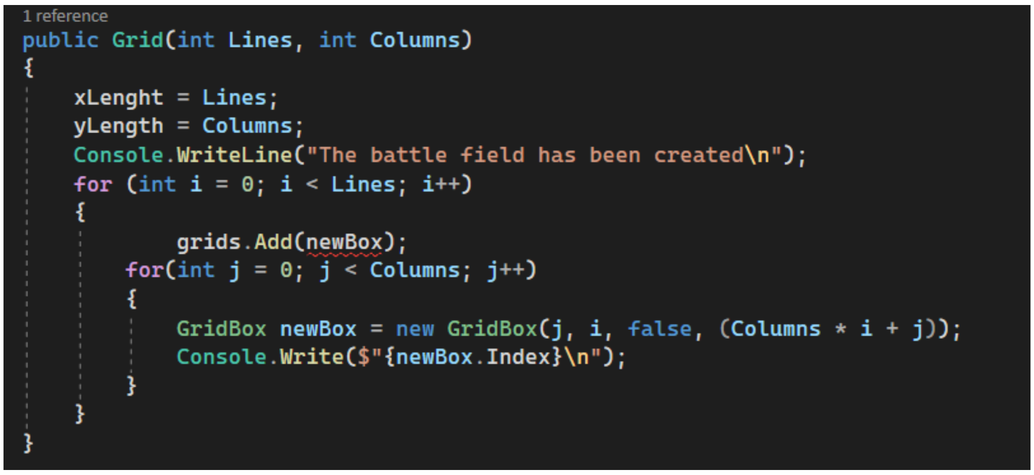
Auto battle code refactor documentation

Author: Bruno Locha Gomes Oliveira

Date: 24/11/2022

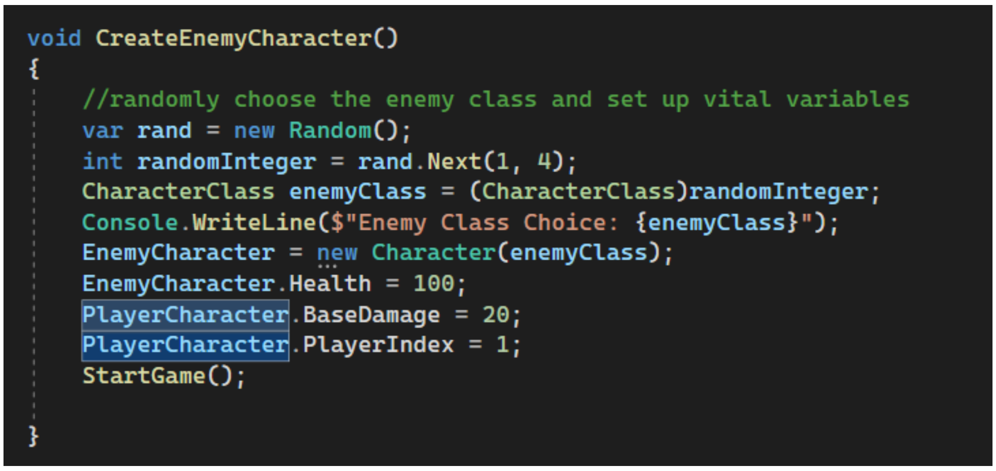
1- There were two scripts (Character.cs and Grid.cs) in the base folder of the project and, consequently, out of the project. I deleted both.

2- There was an compilation error in line 21 of Grid.cs, which was caused by an variable usage on the wrong line. I fixed it by moving the newBox call to the correct context.



```
1 reference
public Grid(int Lines, int Columns)
{
    xLenght = Lines;
    yLength = Columns;
    Console.WriteLine("The battle field has been created\n");
    for (int i = 0; i < Lines; i++)
    {
            grids.Add(newBox);
        for(int j = 0; j < Columns; j++)
        {
            GridBox newBox = new GridBox(j, i, false, (Columns * i + j));
            Console.Write($"{newBox.Index}\n");
        }
    }
}
```

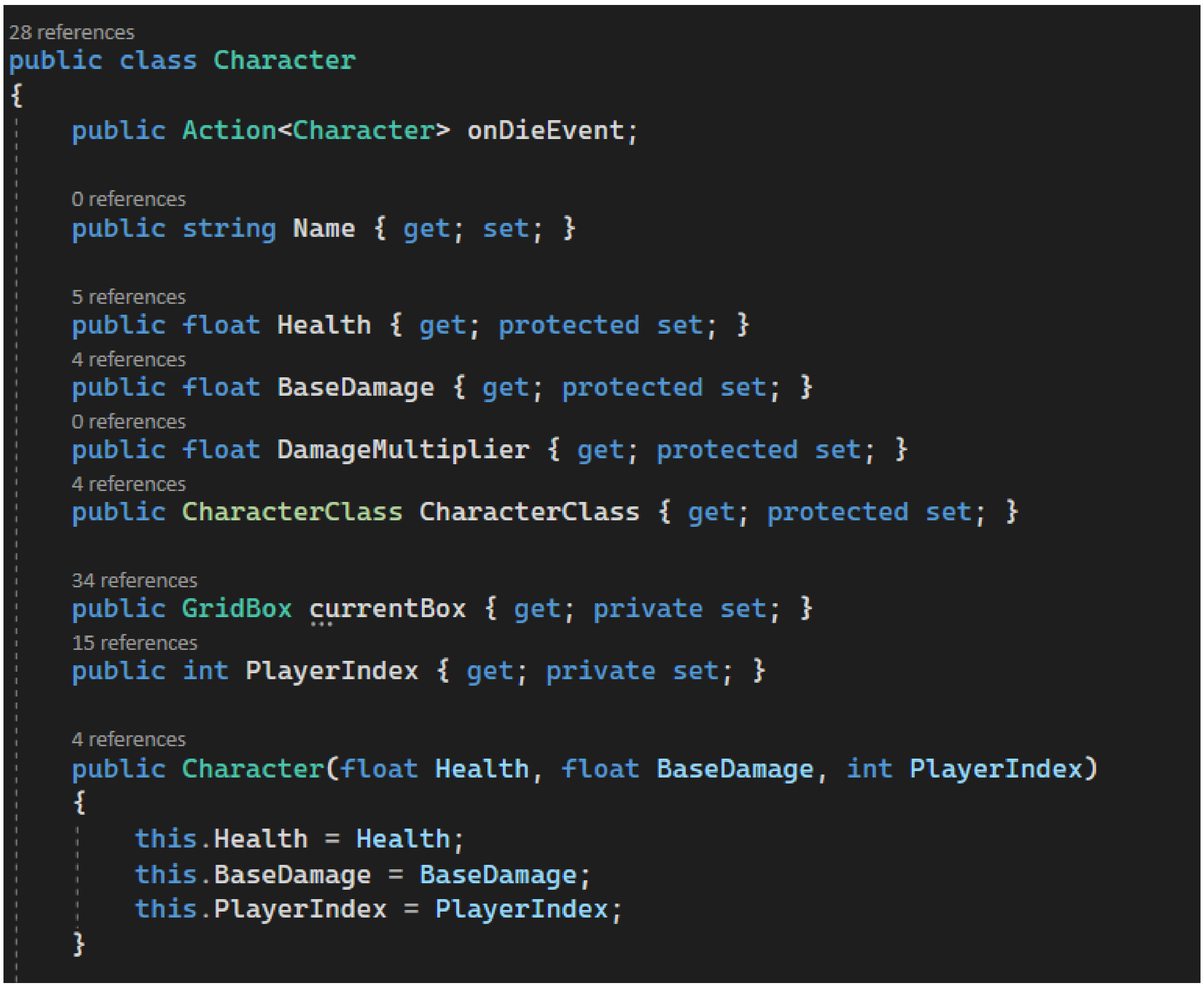
3- On the method CreateEnemyCharacter, the Player character was being wrongly modified.



```
void CreateEnemyCharacter()
{
    //randomly choose the enemy class and set up vital variables
    var rand = new Random();
    int randomInteger = rand.Next(1, 4);
    CharacterClass enemyClass = (CharacterClass)randomInteger;
    Console.WriteLine($"Enemy Class Choice: {enemyClass}");
    EnemyCharacter = new Character(enemyClass);
    EnemyCharacter.Health = 100;
    PlayerCharacter.BaseDamage = 20;
    PlayerCharacter.PlayerIndex = 1;
    StartGame();

}
```

4- To avoid the error appointed in 3, those character properties (Health, BaseDamage, PlayerIndex), must be encapsulated in the Character class and the properties of that class must have access modifiers to ensure that. Also, the

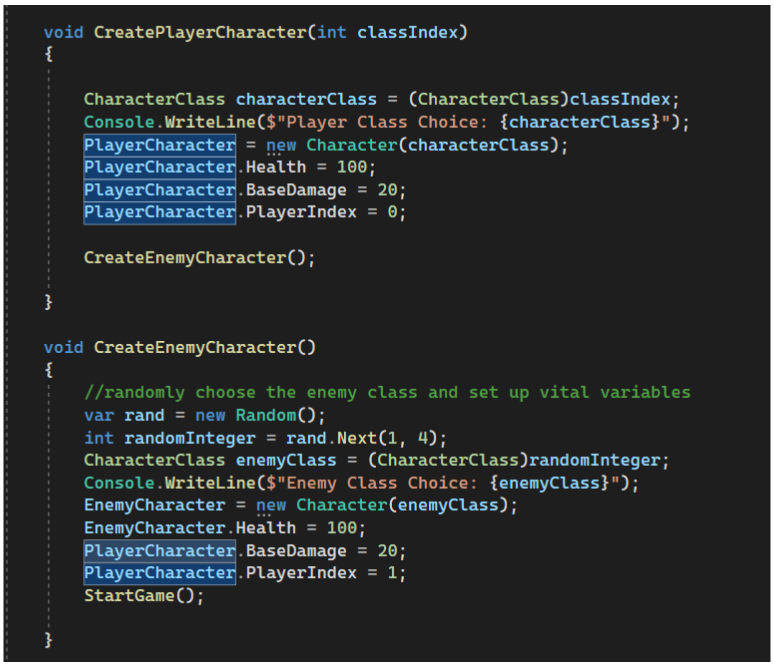
CharacterClass passed to the constructor was stored nowhere, so I created a field for that.



```
28 references
public class Character
{
    public Action<Character> onDieEvent;

    0 references
    public string Name { get; set; }

    5 references
    public float Health { get; protected set; }
    4 references
    public float BaseDamage { get; protected set; }
    0 references
    public float DamageMultiplier { get; protected set; }
    4 references
    public CharacterClass CharacterClass { get; protected set; }

    34 references
    public GridBox currentBox { get; private set; }
    15 references
    public int PlayerIndex { get; private set; }

    4 references
    public Character(float Health, float BaseDamage, int PlayerIndex)
    {
        this.Health = Health;
        this.BaseDamage = BaseDamage;
        this.PlayerIndex = PlayerIndex;
    }
```

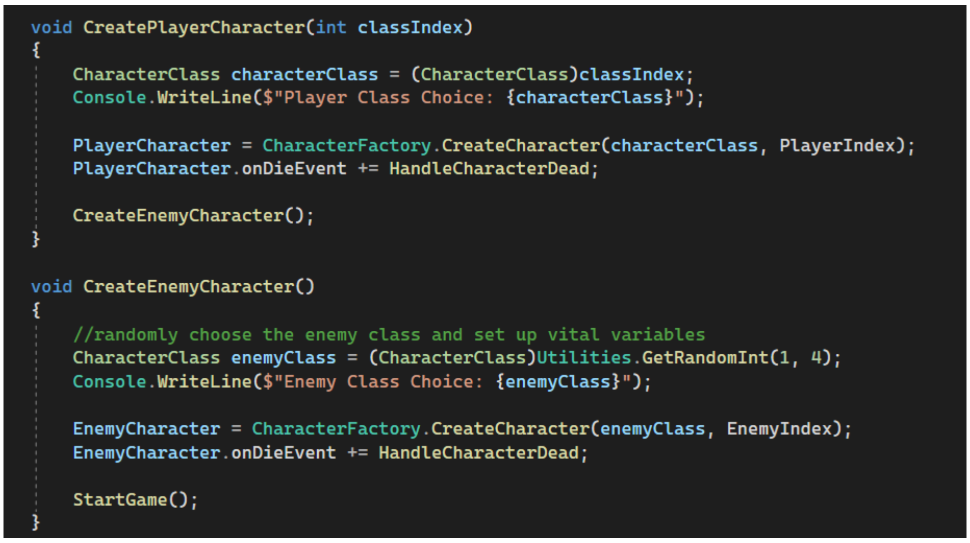
5- Along with 4, I moved the Character creation to a factory class to, in the future, create different types of characters following the Open Closed and Liskov Substitution principles of SOLID, so we can get the maximum code flexibility. I also refactored the Creation code to avoid unnecessary repetition.

Before:



```
void CreatePlayerCharacter(int classIndex)
{

    CharacterClass characterClass = (CharacterClass)classIndex;
    Console.WriteLine($"Player Class Choice: {characterClass}");
    PlayerCharacter = new Character(characterClass);
    PlayerCharacter.Health = 100;
    PlayerCharacter.BaseDamage = 20;
    PlayerCharacter.PlayerIndex = 0;

    CreateEnemyCharacter();

}

void CreateEnemyCharacter()
{
    //randomly choose the enemy class and set up vital variables
    var rand = new Random();
    int randomInteger = rand.Next(1, 4);
    CharacterClass enemyClass = (CharacterClass)randomInteger;
    Console.WriteLine($"Enemy Class Choice: {enemyClass}");
    EnemyCharacter = new Character(enemyClass);
    EnemyCharacter.Health = 100;
    PlayerCharacter.BaseDamage = 20;
    PlayerCharacter.PlayerIndex = 1;
    StartGame();

}
```

After:



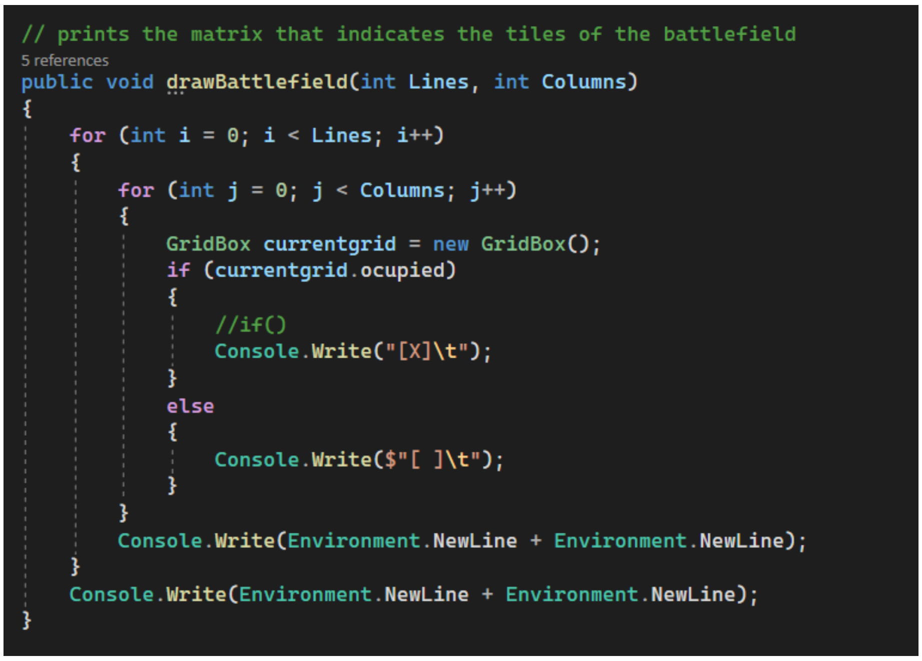
```
void CreatePlayerCharacter(int classIndex)
{
    CharacterClass characterClass = (CharacterClass)classIndex;
    Console.WriteLine($"Player Class Choice: {characterClass}");

    PlayerCharacter = CharacterFactory.CreateCharacter(characterClass, PlayerIndex);
    PlayerCharacter.onDieEvent += HandleCharacterDead;

    CreateEnemyCharacter();
}

void CreateEnemyCharacter()
{
    //randomly choose the enemy class and set up vital variables
    CharacterClass enemyClass = (CharacterClass)Utilities.GetRandomInt(1, 4);
    Console.WriteLine($"Enemy Class Choice: {enemyClass}");

    EnemyCharacter = CharacterFactory.CreateCharacter(enemyClass, EnemyIndex);
    EnemyCharacter.onDieEvent += HandleCharacterDead;

    StartGame();
}
```

This approach also improved the code readability by a lot and was successfully replicated to other parts of the project.

6- In Program.cs, the type Grid was being imported without any need.

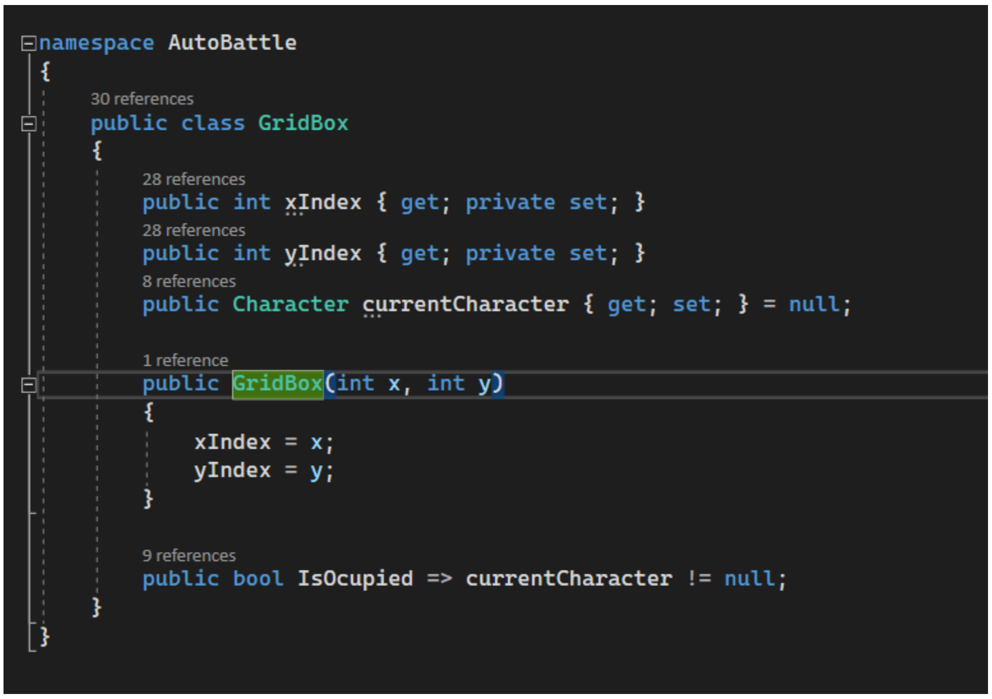
7- The grid search and modification actions were being made in the Character.cs class. This broke the Single Responsibility principle of solid, so I moved all this logic to the Grid.cs.

8- In the Grid.cs, the grids attribute was stored as a List<GridBox>. The battlefield it's in a Matrix structure, so I used private GridBox[,] grids instead of a list, to save the objects. This makes search and general operations much easier and flexible.
9- The DrawBattlefield method on Grid.cs had an error: it was allocating a new object instead of read the existing ones, so the console info was wrong.



```
// prints the matrix that indicates the tiles of the battlefield
5 references
public void drawBattlefield(int Lines, int Columns)
{
    for (int i = 0; i < Lines; i++)
    {
        for (int j = 0; j < Columns; j++)
        {
            GridBox currentgrid = new GridBox();
            if (currentgrid.ocupied)
            {
                //if()
                Console.Write("[X]\t");
            }
            else
            {
                Console.Write($"[ ]\t");
            }
        }
        Console.Write(Environment.NewLine + Environment.NewLine);
    }
    Console.Write(Environment.NewLine + Environment.NewLine);
}
```

10- The GridBox entity was declared as an struct, but this is not a good practice, because it has a constructor and mutable fields. So I moved that from the Types class to an isolated one and modified the struct to a class. Also, the isOcupped field is better as a method that checks if a Character is allocated in the block. This approach facilitates future search type algorithms.

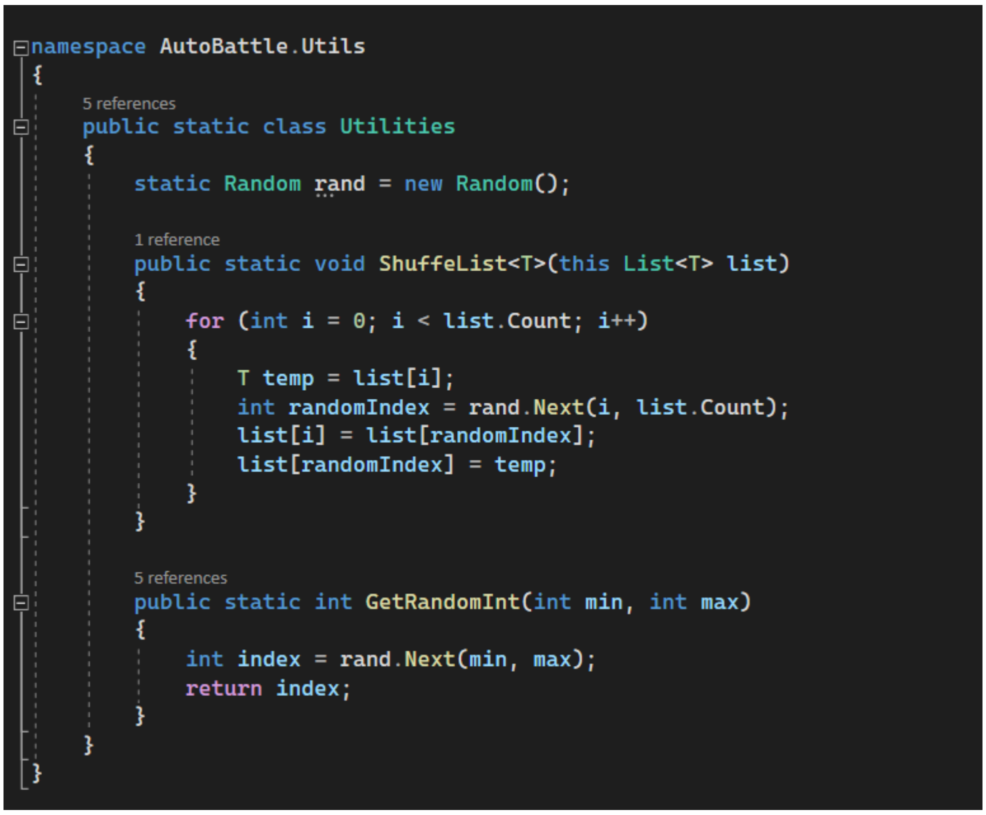


```
namespace AutoBattle
{
    30 references
    public class GridBox
    {
        28 references
        public int xIndex { get; private set; }
        28 references
        public int yIndex { get; private set; }
        8 references
        public Character currentCharacter { get; set; } = null;

        1 reference
        public GridBox(int x, int y)
        {
            xIndex = x;
            yIndex = y;
        }

        9 references
        public bool IsOcupied => currentCharacter != null;
    }
}
```

11- Instead of using a single setted target, the Character must try to find any enemies on the board and try to attack/move to the closest one. This approach opens ways to team and royale battles features.

12- The StartTurn method of Character.cs had all kinds of logic errors, like misplaced return statements and wrong assignments. I refactored it completely using the new Grid approach discussed in 7.

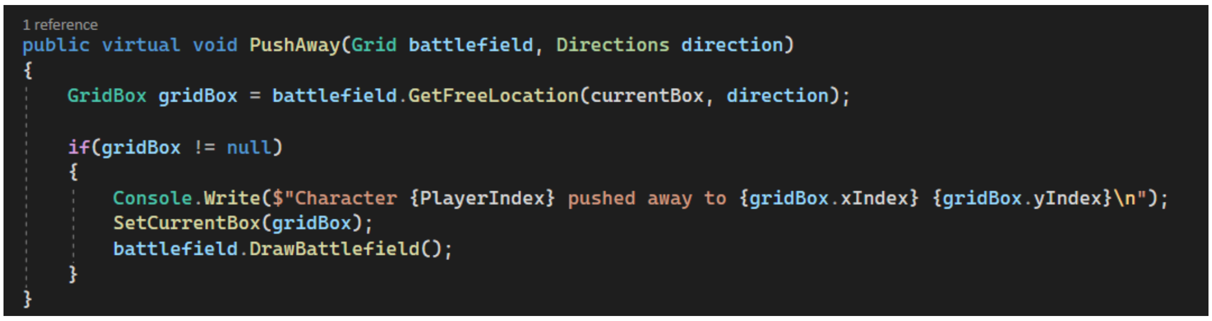
13- There was a method GetRandomInt on the Program.cs. I moved this to an static Utility class, where the Random object was previously stored, so in every operation that requires it, we don't need to create a new one. In the same class, I created a GenericListExtension method to help in shuffle operations.



```
namespace AutoBattle.Utils
{
    5 references
    public static class Utilities
    {
        static Random rand = new Random();

        1 reference
        public static void ShuffeList<T>(this List<T> list)
        {
            for (int i = 0; i < list.Count; i++)
            {
                T temp = list[i];
                int randomIndex = rand.Next(i, list.Count);
                list[i] = list[randomIndex];
                list[randomIndex] = temp;
            }
        }

        5 references
        public static int GetRandomInt(int min, int max)
        {
            int index = rand.Next(min, max);
            return index;
        }
    }
}
```

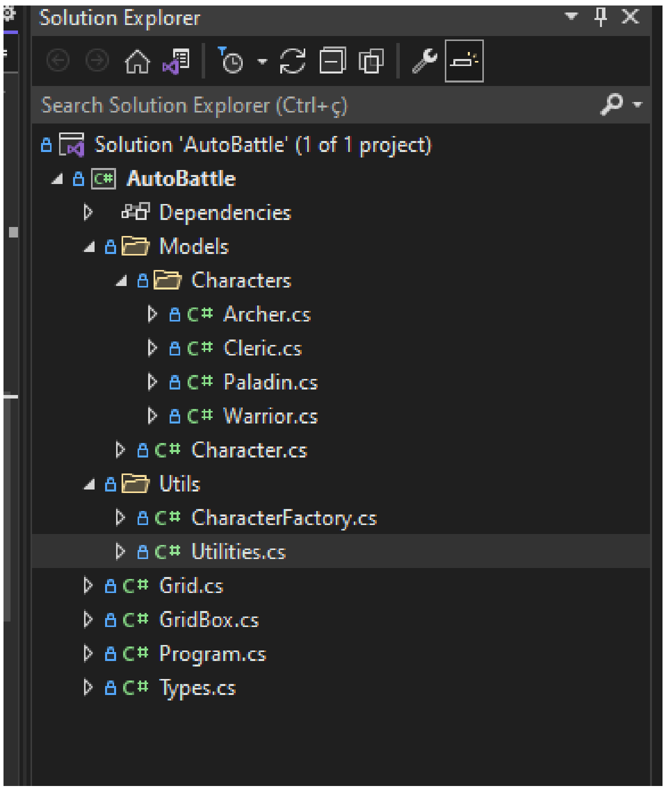
14- In the game combat, all the player turn was processed before check if the was a game over. I fixed this by creating an event on Character.cs, called onDieEvent, that got every observer method stored in it and triggered a callback when the player died. This approach made it possible to control the game using an enum called GameState, which makes the mode more flexible.

15- As requested, I implemented the May~July feature that gives the successful attacks a random chance to push away the target. This method also works with an direction system to enforce naturality in this action.



```
1 reference
public virtual void PushAway(Grid battlefield, Directions direction)
{
    GridBox gridBox = battlefield.GetFreeLocation(currentBox, direction);

    if(gridBox != null)
    {
        Console.Write($"Character {PlayerIndex} pushed away to {gridBox.xIndex} {gridBox.yIndex}\n");
        SetCurrentBox(gridBox);
        battlefield.DrawBattlefield();
    }
}
```

16- Last, I improved the project structure with new namespaces and folders for better organization.


Solution Explorer

Search Solution Explorer (Ctrl+ç)

```
Solution 'AutoBattle' (1 of 1 project)
  AutoBattle
    Dependencies
    Models
      Characters
        Archer.cs
        Cleric.cs
        Paladin.cs
        Warrior.cs
      Character.cs
    Utils
      CharacterFactory.cs
      Utilities.cs
    Grid.cs
    GridBox.cs
    Program.cs
    Types.cs
```