**Refactoring documentation for project ''Minesweeper-5''**

1. **Redesigned the project structure:**

* Renamed the solution to Minesweeper-5.
* Renamed the project to Minesweeper.
* Extracted each class in separate file with name answering to class's purpose. Examples: Scoreboard.cs, BoardManager.cs, Renderer.cs
* Ordered classes in folders depending on their functionality. Examples: Data, Logic, Enums, Interfaces
* Added unit tests to the project.

## Extracted new methods with single, clear purpose. Examples:

## IsMineInPosition() from the method ScanSurroundingFields().

* + CountOpenFields() from the method CheckIfWin().
* Extracted enumerations.

Examples: BoardStatus.cs, FieldStatus.cs

* Extracted interfaces.

Examples: IBoard.cs, IBoardManager.cs, IPlayer.cs

* Implemented design patterns:
  + Creational:

Singleton pattern - used for class Game.cs in order to prevent the existence of two or more instances of the game core.

Prototype pattern - used with IClonable interface and the class Field to create the board(which consists of fields).

* + Structural:

Facade pattern - class Scoreboard plays the facade role. It has methods Save() and Load() and behind that the source of the scoreboard could be easily changes(text file, dictionary..)

Command pattern - used to wrap player's command as object and evaluate it . Represented as class PlayerCommand.cs

* + Behavioral:

Visitor pattern - created IAcceptable and IVisitor interfaces, inherited by Board : IAcceptable and MineSetterVisitor : IVisitor. Used to set the mine status on each field.

Strategy pattern - used with interface IRandomGenerator inherites by class RandomGenerator which is the concrete algorythm to generate random number. An instance of IRandomGenerator is passed to the MineSetter class to determine the logic for generating number.

* Redesigned the structure of the project in order to make the code testable and understandable. Used best practices from OOP, SOLID and DRY principles.

1. **Reformatted the source code**

* Removed all unnecessary empty lines
* Inserted empty lines where needed.
* Removed all unnecessary comments.
* Added descriptive comments where needed.
* Remove unused directives.
* Put all using directives inside of the namespace.
* Added access modifiers to all classes and methods.
* XML documentation where needed.
* StyleCop - no violations!
* Validation and exception handling.
* All variables named in camelCase. All classes, constructors, properties, enumerations, interfaces named in PascalCase. All constants named with capitals and underscores e.g MAX\_ROWS.
* Short life span of the variables.

1. **Renamed variables**

* **str**  renamed to **gameState**
* **name** renamed to **playerName**
* **score** renamed to **playerScore**
* **choosenRow** renamed to **chosenRow**

1. **Renamed methods**

* **menu()** renamed to **Run()**
* **top()** renamed to **ShowTopPlayers()**

1. **Introduced constants**

* GAME\_BOARD\_SIZE = 4
* SCORE\_BOARD\_SIZE = 5
* SPACE = " "
* NO\_PLAYERS\_MESSAGE = "There is still no TOP players!"
* SCORE\_BOARD\_TITLE = "Scoreboard"

1. **New features**

* Simple menu
* Scoreboard
* Visual changes
* A bit more colorful

1. **SOLID & DRY**

* Single Responsibility - The code is designed so that every class and method has strong cohesion and loose coupling. Examples: Renderer.cs, RandomGenerator.cs, IsMineInPosition()
* Open/Closed Principle - The project relies on abstraction in order to make extending the project easier. Used multiple Interfaces and patterns providing abstraction. Examples: IPlayer.cs, IBoardManager.cs, Facade Pattern, Strategy Pattern.
* Liskov Substitution - Almost none in depth inheritance and examples of Liskov Substitution. Basically there are interfaces inherited by concrete implementations.
* Interface Segregation - Used small, cohesive and focused interfaces in order to reduce the existence of unnecessary methods and decrease coupling. Examples: IBoard.cs, IBoaardManager.cs, IRandomGenerator.cs, IHighscore .cs
* Dependency Inversion - Used constructor injection and property injection to make the design more flexible. Examples: Field.cs constructor changed (Constructor injection), MineSetter.cs constructor is passed a random generator(Constructor injection)
* DRY(Don't Repeat Yourself) - Extracted methods and classes to reduce code repetition.

Extracted constants, no magic strings and values. Unit tests written, no manual tests.

1. **Unit Testing**

Written 46 unit tests which test project's functionality and proper behavior.

Code Coverage: 65.96%

1. **More**

These are some examples of the unsorted changes from the project which don't repeat the ones above.

* Property for MaxTopPlayersCount added to Highscore, used in test. (Make it testable)
* Sorting highscore tested
* New class implemented only for testing purposes. RandomGeneratorForTesting.cs; some more tests added for Board and BoardManager;
* Bug Found and Fixed – Game.cs constructor removed, all initializations for current game, have to happen in a new game cycle
* Remove ShowTopPlayers() method from Game.cs, created PrintTopPlayers() in Renderer.cs instead.
* When listing empty players list exception fixed. Add Game constructor. Renderer.cs left with default constructor (used parameter injection instead)
* Field.cs constructor changed (Constructor injection)
* Replace logic for ScanSurroundingFields from Board to the new BoardScanner class. It depends on the private IsMineInPosition() method, so it goes to the new class too.
* Made all fields in Game.cs non-static
* Made code in Game.cs work with non-static methods (including keyword this)
* Remove InitializeGameBoard method (not needed anymore)
* Jagged array replaced by matrix, and made all code working with it.
* Setters added in class Playerfor name and score
* Validation and exception handling added in Player.cs
* Override Equals() for Field.cs, due to some unit testing

\* Not all of the uses of the upper mentioned topics are given as examples.