# Teamwork Project Assignment for the [High-Quality Code Course @ SoftUni](https://softuni.bg/courses/high-quality-code/)

You are given a C# software project designed to implement some of the following **console-based games**:

* [Balloon Pops](Baloons-Pop/High-Quality-Code-Team-Project-Balloon-Pops.docx)
* [Battle Field](Battle-Field/High-Quality-Code-Team-Project-Battle-Field.docx)
* [Bulls and Cows](Bulls-and-Cows/High-Quality-Code-Team-Project-Bulls-and-Cows.docx)
* [Game 15](Game-Fifteen/High-Quality-Code-Team-Project-Game-Fifteen.docx)
* [Hangman](Hangman/High-Quality-Code-Team-Project-Hangman.docx)
* [King Survival](King-Survival/High-Quality-Code-Team-Project-King-Survival.docx)
* [Labyrinth](Labyrinth/High-Quality-Code-Team-Project-Labyrinth.docx)
* [Minesweeper](Minesweeper/High-Quality-Code-Team-Project-Minesweeper.docx)

The project consists of one or more **source code** files and sometimes contains other files and is provided as ZIP archive. You need to **refactor** the project in order to **improve its quality** following the best practices learned in the course "[**High-Quality Programming Code**](https://softuni.bg/courses/high-quality-code/)", apply SOLID and DRY principles, OO design patterns, and **implement unit tests** that ensure that the code has correct behavior.

## Detailed Assignment Description

In order to ensure the high quality of the project you need to fulfill the following tasks:

1. **Perform refactoring** of the **entire project** (its directory structure, project files, source code, classes, interfaces, methods, properties, fields and other class members and program members and its programming logic) in order to **make the code "high quality"** according to the best practices introduced in the course "High-Quality Programming Code". The obtained **refactored code** should conform to the following characteristics:
   * **Easy to read**, **understand** and **maintain** – the code should be well structured; should be easy to read and understand, easy to modify and maintain; should follow the concept of **self-documenting code**; should use **good names** for classes, methods, variables, and other identifiers; should be consistently formatted following the best formatting practices; should have **strong cohesion** at all levels (modules, classes, methods, etc.); should have **loose coupling** between modules, classes, methods, etc.; should follow the best practices of organizing programming logic at all levels (classes, methods, loops, conditional statements and other statements); should follow the best practices for working with variables, data, expressions, constants, control structures, exceptions, comments, etc.
   * **Correct behavior** – the project should fulfill correctly the requirements and to **behave correctly** in all possible use cases. This means that all bugs or other problems in the project (e.g. performance or usability issues) should be fixed and any unfinished or missing functionality should be completed. The code should be very well tested with properly designed **unit tests**.
2. **Design and implement unit tests** covering the entire project functionality. To ensure the project works correctly according to the requirements and behaves correctly in all possible use cases, design and implement unit tests that cover all use cases and the entire program logic. If needed, first redesign the program logic to **make the code testable**. Test the normal expected behavior (correct data) and possible expected failures (incorrect data). Put special attention to the border cases. The **code coverage** of the unit tests should be **at least 80%**.
3. **Implement design patterns** – redesign the project to apply a fewof the classical OO design patterns:
   * **Structural patterns** – adapter, aggregate, bridge, composite, decorator, extensibility, façade, etc.
   * **Behavior patterns** – chain of responsibility, command, interpreter, iterator, mediator, observer, etc.
   * **Creational patterns** – abstract factory, builder, factory method, singleton, prototype, etc.
4. **Follow the SOLID and DRY principles**
   * Single responsibility principle
   * Open / close principle
   * Liskov substitution principle
   * Interface segregation principle
   * Dependency inversion principle
   * Don't repeat yourself (DRY) principle
   * Redesign the project to fulfil the **SOLID** and **DRY** principles – each principle should be implemented at least once
5. **Briefly document the refactorings** you have performed in order to improve the quality of the project. Use English or Bulgarian language and follow the sample (see below).

## Deliverables

1. The **original source code** (project files, .cs files) without executables.
2. The **refactored source code** (project files, .cs files) without executables.
3. The **unit tests** – source code (project files, .cs files) without executables.
4. The **refactoring documentation**.

**Pack the project deliverables** in a single ZIP archive. Be sure to **avoid including large unused** **files** in the archives (e.g. compilation binaries and well-known libraries). Your archive should be up to **8 MB**. Each team member should submit the same archive as a homework.

## Teamwork Guidelines

* Use **GitHub** (<http://github.com>) as project hosting, source control and team collaboration environment.
* Each team member should have **commits in at least 3 different days**.

## Public Project Defense

Each team will have to deliver a **public defense** of its work in front of the other students, trainers and assistants. Teams will have **only 10 minutes** for the following:

* Demonstrate the **refactored code** (very shortly).
* Explain what **refactoring has been performed** and why. The documentation will definitely help you.
* Demonstrate the **unit tests** and code coverage.
* Show the **source code** and explain how it works.
* Explain how each team member has **contributed**: display the change logs in GitHub.
* Optionally you might prepare a **presentation** (3-4 slides).

Please be **strict in timing**! On the 10th minute you **will be interrupted**! It is good idea to leave **the last 2 minutes for questions** from the other students, trainers and assistants.

Be **well prepared** for presenting maximum of your work for minimum time. Bring your own laptop. Test it preliminary with the multimedia projector. Open the project assets beforehand to save time.

## Assessment Criteria

* **Refactored code quality** (easy to read, understand and maintain, well-structured code, split into loosely coupled methods / classes / files, correctly used OOP, with good naming, formatting, correctly applied SOLID and DRY principles, etc.) – **0…5**
* **Unit tests** (code made testable, well written unit tests, 80% code coverage or more) – **0…6**
* **Correct behavior** (working project according to the problem description) – **0…2**
* **Design patterns** (2 design patterns implemented) – **0…2**
* **Documentation** (2 design patterns implemented) – **0…2**
* **Teamwork\*** (GitHub used; each team member contributed in 5 different days; distribution of tasks) – **0…3**
* **Bonus** – **0…2**

\* If not all team members have contributed to the project, this **does not affect** the Teamwork points.

## Give Feedback about Your Teammates

You will be invited to **provide feedback** about all your teammates, their attitude to this project, their technical skills, their team working skills, their contribution to the project, etc. The feedback is important part of the project evaluation so **take it seriously** and be honest.

|  |  |  |  |
| --- | --- | --- | --- |
| Sample Refactoring Documentation for Project "Game 15"  1. Redesigned the project structure:    * Renamed the project to **Game-15**.    * Renamed the main class **Program** to **GameFifteen**.    * Extracted each class in a separate file with a good name: **GameFifteen.cs**, **Board.cs**, **Point.cs**.    * … 2. Reformatted the source code:    * Removed all unneeded empty lines, e.g. in the method **PlayGame()**.    * Inserted empty lines between the methods.    * Split the lines containing several statements into several simple lines, e.g.:  |  |  |  | | --- | --- | --- | | **if (input[i] != ' ') break;** | ➔ | **if (input[i] != ' ')**  **{**  **break;**  **}** |  * Formatted the curly braces **{** and **}** according to the best practices for the C# language * Put **{** and **}** after all conditionals and loops (when missing). * Character casing: variables and fields made **camelCase**; types and methods made **PascalCase** * Formatted all other elements of the source code according to the best practices introduced in the course "High-Quality Programming Code". * …  1. Renamed variables:    * In class **Fifteen: number** ➔ **numberOfMoves**.    * In **Main(string[] args): g** ➔ **gameFifteen**. 2. Introduced constants:    * **GameBoardSize = 4**    * **ScoreBoardSize = 5** 3. Extracted the method **GenerateRandomGame()** from the method **Main()**. 4. Introduced class **ScoreBoard** and moved all related functionality in it. 5. Moved method **GenerateRandomNumber(int start, int end)** to separate class **RandomUtils**. 6. … |

/// <summary>

/// Property of the X coordinate

/// </summary>

/// <summary>

/// Property for the Y coordinate

/// </summary>

/// <summary>

/// Method that check the coordinates and whether there are existing coordinates

/// </summary>

/// <param name="input">The input string with elements</param>

/// <param name="result">The coordinates where the player is staying</param>

/// <returns></returns>