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

You are given the library **KitchenPC** which is an open-source framework written in C# for working with recipes, shopping lists, and menus.

**Note:** The code you are given has been adapted to the High-Quality Code course. Please **do not download** the official code repository. Use the provided code instead.

## Official Documentation

The documentation of the library is located here: <https://github.com/KitchenPC/core>.

## Code Architecture. Getting to Know the Project

Research the project. Read the documentation (or at least get to know how the project works). Build class diagrams to see the various classes inside the project and the relations between them.

Get acquainted with the abstract structure of the code: what subsystems it contains, how these subsystems are related to each other, what is the purpose of interfaces and abstract classes, what design patterns does the project use, what design decisions have been made and what are the motivations for them, etc.

Run the unit tests (some of them will fail, you will have to worry about this later). The unit tests are also good sources of documentation, so read some of them and see how the system works together.

You should know the architecture of the **entire project** before you start refactoring it.

## Refactoring – 8 points

**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".

Please use the code conventions described in the lectures, if possible. For example, according to these conventions, there should be no underscore in front of private fields (**private int \_field**). However, it is more important to stick to **consistent conventions** throughout the 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**.

## Bug Fixing – 2 points

There are several bugs in the code. There are some issues in the GitHub repository of the project, but you **don't need to fix them**. There are some other bugs in the code which make some of the unit tests fail.

Describe the bugs using **GitHub Issues**. Open separate issues for all bugs and close them when the bugs have been fixed. You can optionally add comments to the issues.

Find and remove the bugs. Write some more unit tests in order to be sure the bug will not persist in future versions of the code. Write **at least one unit test** for each bug you have found (and removed).

## Unit Testing – 6 + 2 points

Write unit tests for the following classes:

* **Weight.cs**
* **Fluent\Recipes.cs**
* **Recipes\RecipeQuery.cs**
* **Modeler -> UnitTests\Modeler.cs**
* **Tags -> UnitTests\Tags.cs** (consider refactoring the test cases, they are too long)

There are currently some tests, but you should add some on your own. There are some test stubs (look for TODOs in the **Tests** project) but feel free to add more tests.

Some tests are empty. Consider writing them.

Follow the patterns of the tests which have already been written.

The code coverage for your tests should be at least 80% (for the methods you have tested).

You will need to use mocking for some of the unit tests. You can use existing mocks and write more on your own.

## Code Documentation – 5 points

Add XML documentation for some of the classes, properties, methods, etc. You may optionally create a Sandcastle documentation file.

## SOLID, DRY and Other Principles – 5 points

Document at least two instances of each of the **SOLID** principles.

If you have implemented some of the principles yourself (and they have not been in the code before), document this too. Include this in the refactoring documentation.

You can also document any **best practices** you like in the code.

## Design Patterns – 4 points

Document at least one instance of a design pattern from each of three design pattern categories: **Creational**, **Structural** and **Behavioral**.

If you have implemented some of the design patterns yourself (and they have not been in the code before), document this too. Include this in the refactoring documentation.

## Refactoring Documentation

Create a document (free text, either a Word / OpenOffice document, or a text document) describing all refactorings in the code. Refer to the sample below.

|  |  |  |  |
| --- | --- | --- | --- |
| Sample Refactoring Documentation for Project "…" **Code Refactoring:**   1. Redesigned the project structure:    * Renamed the project to **…** 2. Reformatted the source code:    * Removed all unneeded empty lines, e.g. in the method …**()**.    * 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, for example:    * 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. …   **Bug Fixing and Unit Tests:**   1. Fixed bug with …, wrote GitHub issue, wrote unit tests in class … 2. …   **SOLID, DRY and Other Principles. Best Practices:**   1. Single Responsibility Principle:    * …    * … 2. Open / Closed Principle:    * …    * … 3. …   **Design Patterns:**   1. Creational Design Patterns    * … 2. Structural Design Patterns:    * … 3. Behavioral Design Patterns:    * … |