**High-Quality Programming Code – Team Labyrinth-7**

**Overview**

The **Labyrinth-7** team is given a **C# software project** designed to implement the interactive **console-based game “Escape from Labyrinth**”.

In this game the player should escape from a labyrinth of size 7x7 cells.

The project was implemented according to theoverall High-Quality Programming Code team projects assignment, and the special instructions given in the Quality-Code-Team-Project-Labyrinth.docx file.

Тhe works on the project included full refactoring aiming to **improve its quality** following the best practices learned in the course “[High-Quality Programming Code](http://telerikacademy.com/Courses/Courses/Details/244)” and **implemented unit tests** to ensure that the code has correct behavior.

\*\*\*\*\*\*\* Със зелено са отбелязани извършени работи по време на екипната работа \*\*\*\*\*\*

Документацията трябва да бъде на английски, затова тук мисля да записваме изречения на английски включително. Файлът е във формат .docx, но има конвертори към .md и накрая ще конвертираме.

*Това са направени досега – 22 юли неща от Петър, от мен и от Димитър (в Гит коментарите не винаги са пълни, или може би там да ги даваме пълни на английски?). Надявам се който има възможност да се включи, дори и ако не знае C# (каквото можем, ще помагаме), но ако следи курса и си предава домашните навреме, за да се яви на изпит:*

Дешифрирани наименования като flag, използван е дебъгера и проследени няколко метода, в единия са променени имената с по – адекватни, за да се разбира какво прави.

Направени началните размери на лабиринта (дължина на редовете/колоните) да са константи, защото бяха директно написани с число.

Променени имената на две от променливите flag,\_temp и flag4 на имена, които съответстват на действията им.

Направени 2 малки промени по имената на методи и променливи, в комита съм описал кои. Останаха още малко неща за преименуване. След това трябва да видим кои методи може да се редактират или да се разделят на по малки парчета.

Module Labyrinth.cs renamed to LabyrinthEngine.cs; added Singleton pattern in module LabyrinthEngine; removed Flag3 - used while(true) instead.

\*\*\*\*\*

Using Design patterns:

1. Creative design patterns
   1. Singleton
   2. Lazy Initialization

The Labyrint class (module LabyrintEngine) controls the game Labyrinth using one instance at a time by the means of Singleton creative design pattern combined with the "Lazy Initialization" creative design pattern.

The explicit static constructor tells C# compiler not to mark type as beforefieldinit; this provides for using the "Lazy Initialization" pattern, because the laziness of type initializers is only guaranteed by .NET when the type isn't marked with a special flag called "beforefieldinit".

\*\*\*\*\*

Method LabyrinthGenerator refactored: changed names and types of the variables representing the coordinates of the cells. Similar changes made throughout all the code where the coordinates of the labyrinth cells have been used.

Removed spaghetti code from the method SolutionChecker (additional code should be added to check if there is at least one path out of the maze). ?????????

Лабиринтът е променен да се рендерира само веднъж на сеанс, а не при всяка стъпка на движение; движението се извършва със стрелките, а за TOP, RESTART и EXIT се въвеждат съответно T, R и E.

Добавен нов цикъл, който да рисува реда на лабиринта, вместо 7-те реда в оригиналния код.

\*\*\*\*\*\*\*\*\*\*\*\*\*Със син фонт по-надолу е отбелязано оригиналното задание\*\*\*\*\*\*\*\*\*\*\*

**Detailed Assignment Description**

In order to ensure the high quality of the assigned 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](http://telerikacademy.com/Courses/Courses/Details/244)”. 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.
   * All **default StyleCop rules should pass** without any warning.
   * **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. **Implement design patterns** – redesign the project to fulfil 9 of the [Software Design patterns](http://en.wikipedia.org/wiki/Software_design_pattern)
   * **Structural patterns** – implement **at least 3** of the structural design patterns (adapter, aggregate, bridge, composite, decorator, extensibility, façade, etc…)
   * **Behavior patterns** – implement **at least 3** of the behavioral design patterns (chain of responsibility, command, interpreter, iterator, mediator, observer, etc…)
   * **Creational patterns** – implement **at least 3** of the creational design patterns (abstract factory, builder, factory method, singleton, prototype, etc…)
3. **Follow the SOLID and DRY principles** – Single responsibility, Open/close, Liskov substitution, Interface segregation, Dependency inversion, Don't repeat yourself
   * Redesign the project to fulfil the [SOLID](http://en.wikipedia.org/wiki/Solid) and [DRY](http://en.wikipedia.org/wiki/Don%27t_repeat_yourself) principles – each principle should be implemented at least once
4. **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 90%.
   * Use unit testing framework of your choice (e.g. Visual Studio Team Test, NUnit, MbUnit or other).
   * At least 10 of your tests should use **mocking** (Moq, JustMock or other).
5. Documentation and comments
   * **Document the refactorings** you have performed in order to improve the quality of the project. Use English language and follow the sample (see below). The documentation must be in .md (GitHub Markdown) format.
   * Document all public members and classes using **XML documentation** within the code and generate **CHM documentation**.
   * Add **code comments** where appropriate
6. **Add new functionalities** to the game. The more functionalities you add the more points you will get.
   * Follow all principles and unit test you new game functionalities.

**Deliverables**

1. The **original source code** (project files, .cs files) without executables (in folder /before/).
2. The **refactored source code** (project files, .cs files) without executables (in folder /src/).
3. The **unit tests** – source code (project files, .cs files) without executables (in folder /tests/).
4. The **refactoring documentation**. (README.md file)

**Team Work Requirements**

* Obligatory use **Git** as source code repository and **GitHub** (<http://github.com>) as project hosting and team collaboration environment. SVN or TFS are **not** allowed for this project.
  + Take advantage of the GitHub issues for project management
* **Each team member** should have contributions to the project and **commits in the source control repository in 7 different days**. We acknowledge that this requirement seems a bit unnatural, but we want to track **how the team collaborates over the time** and that the **project is developed incrementally**, not in the “last minute”.

**Other Requirements**

* Every team member should send the project in the students system <http://telerikacademy.com>.
  + Pack the project deliverables in a **single ZIP archive**.
  + Be sure to avoid including large unused files in the archives (e.g. compilation binaries).
  + Your archive should be up to 8 MB.
  + Each team member should submit the same archive as a homework.
* Be prepared as a team to **defend your project** in front of the course lecturers. You should be able to explain what refactorings have been performed and why. The documentation will definitely help you. Be prepared to **demonstrate how the unit tests cover the project’s functionality**. Preferably bring your own laptop to reduce the effort to setup your development environment and project workspace.
* Be prepared to **explain the used patterns and SOLID principles**.
* Be prepared to **show the commit logs** from the source control system to demonstrate how the project development efforts are shared between the team members and over the time.

**Discussion Forum**

* You can freely discuss the course projects and ask questions in the official discussion forum of the course: <https://telerikacademy.com/Forum/Category/19/c#-qpc>

**Sample Refactoring Documentation for Project “Game 15”**

1. Redesigned the project structure: Team “…”
   * 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.:

Before:

if (input\[i\] != ' ') break;

After:

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](http://telerikacademy.com/Courses/Courses/Details/244)”.
  + …

1. Renamed variables:
   * In class Fifteen: number to numberOfMoves.
   * In Main(string\[\] args): g to gameFifteen.
2. Introduced constants:
   * GAME\\_BOARD\\_SIZE = 4
   * SCORE\\_BOARD\\_SIZE = 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. …