# Exercises: High-Quality-Code-and-Refactoring

Problems for exercises and homework for the "Software-Engineering" course from the official "Applied Programmer" curriculum.

## Rotating Walk in Matrix

### Understand the Task of the Program

We are given a **square** **matrix** of **n x n** cells. A **rotating** **walk** in the matrix is a walk that **starts** from the **top** **left** **corner** of the matrix and goes in the **down-right** **direction**. When **no continuation** is **available** at the **current** **direction** (either the matrix wall or non-empty cell is reached), the **direction** is **changed** to the **next** **possible** **clockwise**. The **eight** **possible** **directions** are as follows:

8-directions

When **no empty cell** is **available** in **all directions**, the **walk is restarted** from an **empty** **cell** at the **smallest** **possible** **row** and as **close** as **possible** to the **start** of this **row**. When **no empty cell** is **left** in the **matrix**, the walk is finished.

Your task is to write a program that reads from the console an **integer number** n (1 ≤ **n** ≤ 100) and displays the **filled matrix on the console**.

#### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Explanation** |
| **3** | **1 7 8**  **6 2 9**  **5 4 3** | **Starts** from the **top** **left** **corner** and **move** **along** the **diagonal** **(1-2-3)**. After reaching the **bottom** **right** **corner** the **direction** is **changed** **clockwise** and the **matrix** **filling** **continues** **(4-5)**. The **direction** **changes** **again** until it **fills** the **matrix** **(6-7-8-9)**. |
| **6** | **1 16 17 18 19 20**  **15 2 27 28 29 21**  **14 31 3 26 30 22**  **13 36 32 4 25 23**  **12 35 34 33 5 24**  **11 10 9 8 7 6** | **Starts** from the **top** **left** **corner** and **move** **along** the **diagonal** **(1-2…6)**. After reaching the **bottom** **right** **corner** the **direction** is **changed** **clockwise** and the **matrix** **filling** **continues** **(7-8…11)**. The **direction** **changes** **again** until **no empty cell** is **available** at **all directions** **(12-13…15)**, **(16-17…20)** … **(30)**. When **no empty cell** is **available** in **all directions**, the **walk is restarted** from an **empty** **cell** at the **smallest** **possible** **row** and as **close** as **possible** to the **start** of this **row (31)**. After finding an empty cell the matrix filling continues until it fills the matrix **(31-32…36)**. |

### Input

The input is a **single positive integer** – n – **number of cells** [0…100].

### Output

**Print** on the console **the filled matrix**. Each **element** (number) in the matrix should be in a **4-character field**:



## Refactor and Clean the Smelly Code

Use the Visual Studio **solution** "Matrica.sln".Improve its **internal quality**. You might follow the following steps**:**

* **Open the project** in VS and **examine the classes**, **project structure**, etc.
* **Run** the **code** to see if it **compiles** **correctly**.
* See if there are **any** **differences** **between** your **console** **output** and the **example** **output**.
* Start **reading** the **code**.
  + Find out **what** the **methods** **do**.
  + Find out **what** the **variables** **do**.
  + **Debug** **code** if necessary.
* Think about **what** **changes** you **can** **make**.
  + Take **notes** if necessary.
* **Refactoring** the code:
* **Reformat** the code.
* **Rename** the **badly named variables**.
* **Search** for **magic** **numbers**.
* **Rename** the **badly named methods**.
* **Extract** **duplicate** **code** in **methods**.
* **Extract** **pieces** of **code** that **work** **together** in **new** **methods**.
* **Order** the **methods** **correctly**.
* **Split** **complex** **boolean** **expressions** into **several** **expressions** or **methods**.
* **Add** **comments** to **parts** of the **code** that are **difficult** to **understand**.
  + Make the **code testable**.
* **Write unit tests. Fix** any **bugs** found while **testing**.
  + **Test** all **edge** **cases** of the **methods**.
* Think about **how** to **test** **console-based** **input**/**output**.

**Refactor** the **code** **following** the **guidelines** from this lecture. Do it **step** **by** **step**. **Run** the **unit** **tests** after each **major** **change**.

### Rename Project Files

The first thing you should do is to look at the "Matrica.sln" **project structure**:



As you can see, we have a **badly named project and class**. Think of how to change the "Matrica" and "Program.cs" **file names**, so that they follow **good naming practices**. Don't forget to **change the namespaces**, as well.

### Refactoring Examples

**Refactoring hints** for Matrica.sln code.

#### Badly Initialized and Named Variables

Look at this code snippet:



As you can see, we should **refactor the code**, as it is **not readable** and it is not a good practice to **define many variables at once**.

In addition, our **variables are named badly** (you cannot tell what the **k** variable keeps as a value, as its name doesn't suggest it).

The **code** **may be** **refactored** like this:

Graphical user interface, text

Description automatically generated

#### Magic Numbers

In badly written code we often have some **magic numbers** (a direct usage of a number in the code):



This is bad, because it is **not clear what these numbers are**. To improve code, we should move these numbers to **variables** or **constants** and use them as shown below:

 🡪



#### Badly Named Method

In the below example we have a **method**, which **name** is not correct, according to the **C#'s standart of naming methods**. Also, it has no **explicitly set access modifier**, which means it is an internal method:

A picture containing icon

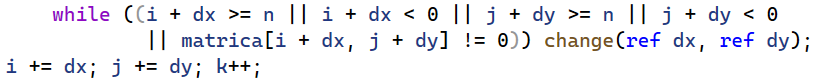
Description automatically generated

To fix this, we should change the **name of the method** to be in PascalCase and make it public to be visible to all other classes (for example, test classes):



#### Duplicated Code and Complex Boolean Expressions

In the example below, we have **duplicated code** and **complex Boolean expressions**:



We should **clear the code** and **refactor it**, **without changing its logic**. This is done by **separating the repeated code** to **methods** and **reusing them**. Finally, it may look like this:

Graphical user interface, text, application

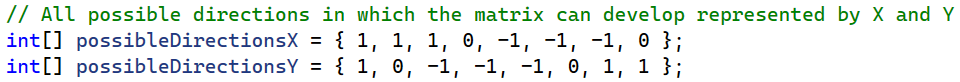
Description automatically generated with medium confidence

#### Comments

We have some places in the code where it is **not easy to understand** the app logic or what some **variables** are:



In this case, it is a **good idea to add a comment** to explain the code simply, with one sentence:



Also, you should **delete bad and useless comments**:



### Unit Tests Example

Don't forget that you should write **unit tests** to **test your app logic** and if it **works correctly**.

To do so, you should first make sure that **all your methods and code are testable** – methods should have clear input and output.

Then, **write positive and negative test cases** for **each method** and make sure they **run sucessfully**. Also, **test all edge cases** and think about how to **test console-based input/output** (you can find an easy way here: <https://www.pmichaels.net/2022/05/26/unit-testing-a-console-application/>).

For example, we should have **4 test methods**, testing if we go out of the **matrix in the 4 directions**, and at least one **negative test case**:

