TP 14 C# : Sudoku Solver

Handing in the practical

At the end of the practical, your git repository must respect the following architecture:

```
tp14-sherlock.holmes/
|-- README
|-- .gitignore
|-- SudokuSolver/
    |-- SudokuSolver.sln
    |-- MiniTests/
        |-- Everything except bin/ et obj/
    |-- SudokuSolver/
        |-- Everything except bin/ et obj/
    |-- SudokuSolverTests/
        |-- Everything except bin/ et obj/
```

Make sure you follow these guidelines before handing in your tp:

- Replace `sherlock.holmes` with your login.
- The file `README` is mandatory.
- No `bin` or `obj` in the project.
- Strictly abide by the given prototype.

README

In this file you must write any and all of your commentary on the TP, your work or anything that comes to mind. An empty `README` will result in a penalty.

1 Introduction

The goal of this practical is to introduce you to the usage and writing of unit tests. To do this, after getting used to those, you will have to implement a Sudoku and its solver, while being assisted by tests. Some of them are provided, but you will have to write some yourself.

1.1 Unit tests ?

You should already have heard about unit tests this year. Unit tests are procedures (functions, for example) ensuring that a part of your program is working as it should be (for example, checking that a function returns the right result when given certain parameters).

When using tests, the goal is to make enough so that most of your program is tested. The proportion of lines of your project executed when running your tests is called the **test coverage**. The more your coverage is close to 100%, the more you're continuously ensuring that your project is working well. When configuring GitHub, for example to run your test suite after each commit, you will be automatically notified if one of your edits has broken something in your project.

It is common to write tests for a functionality before implementing it, this is called **T.D.D.: Test Driven Development**, which allows you to know if something is working right after you implemented it, and to improve it while continuously checking that it still returns the right result. This method also guarantees a high test coverage.

1.2 NUnit

To write unit tests in C#, one of the most popular libraries is **NUnit**. The easiest way to add tests to a project is to use the **Unit Test Project** template in the New Project Wizard in Rider. This will create a project with all the required dependencies.

Tip

If you already have a Main function in your project, you should create a new one for your tests. This is because NUnit already provides a Main function to run your test, which will conflict with yours.

To write a test with NUnit, you just need to create a function (in any class) with the **Test** attribute. You can then use the **Assert** class to check some results:

```
1 [Test]
2 public void TestAdd()
3 {
4     Assert.AreEqual(0, Add(0, 0));
5     Assert.AreEqual(7, Add(5, 2));
6     Assert.AreEqual(-1, Add(3, -4));
7 }
```

To prevent code repetition, you can use multiple **TestCase** attributes instead to list inputs that will be given to your function as parameters. Each case corresponds to a test.

```
[TestCase(0, 0, 0)]
[TestCase(5, 2, 7)]
[TestCase(3, -4, 1)]
public void TestAdd(int a, int b, int result)
{
    Assert.AreEqual(result, Add(a, b));
}
```

To use a range as input, you can use the **Range** attribute (with the **Test** attribute too this time):

```
[Test]
public void TestAddOpposites([Range(0, 10)] int x)
{
    Assert.AreEqual(0, Add(x, -x));
}
```

Here, 11 tests will be generated with the values between 0 and 10 (both included).

There are plenty of ways to write tests with NUnit, and also a lot more functions other than **AreEqual** in the **Assert** class. You can, for example, check that a function is throwing a certain exception, check that a list contains an element, and more! We invite you to check the official NUnit documentation to learn more.

Warning

When using a function from the **Assert** class, the expected result is always the first parameter, while the one you obtain should be the second one.

1.3 Run tests on Rider

To run your tests on Rider, you can use the button in the margin next to your test functions or the class containing them. The 'Unit Tests' tab on the bottom left will allow you to get the list of the tests of your project to run them individually, or all together.

2 Mini-tests

This section will get you used to unit tests.

2.1 Suspicious functions

In this section, you must fix the functions provided in the **SuspiciousFunctions** so that they pass all the tests from the **SuspiciousFunctionsTests** class.

2.1.1 Fibonacci

```
1  public static int Fibonacci(int n)
```

The **Fibonacci** function returns the **n**-th element of the Fibonacci sequence.

2.1.2 ViceMax

```
1  public static int ViceMax(int[] array)
```

The **ViceMax** function returns the second biggest element of an array. You don't have to handle the case where the array is empty, or filled with the same value.

2.2 Empty functions

In this section, you must implement the functions in the **EmptyFunctions** class so that they pass all the tests from the **EmptyFunctionsTests** class.

2.2.1 GetDigit

```
1  public static int GetDigit(int n, int digit)
```

The **GetDigit** function returns the **digit**-th digit of the number **n** starting from the right.

2.2.2 IsNumberPalindrome

```
1  public static bool IsNumberPalindrome(int n)
```

The **IsNumberPalindrome** function check if **n** is a palindrome, i.e. that it can be read both forwards and backwards.

3 Sudoku

The goal of this section is for you to implement the Sudoku game as well as its solver.

Sudoku is a logic based combinatorial number-placement puzzle originating from Japan and popularized in the 19th century by French newspaper featuring such games on their papers. Wikipedia article for those who wants to go further https://en.wikipedia.org/wiki/Sudoku

4 Rules and goal

The goal of the game is to fill a board of 9 by 9 squares with numbers between 1 and 9. For the board to be valid, the player must abide by a few simple rules.

- Each line must have 9 different numbers.
- Each column must have 9 different numbers.
- Each square must have 9 different numbers.
- You can only use a number between 1 and 9.
- Each case must be filled with a number.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

As per the picture above, in pink : a square, in blue : a line and in green a column

5 A few pointers

5.1 Provided code

To make your life easier, we have given you a pretty-printer for the Sudoku Boards.

As you may have noticed this practical insists heavily on the use of testing, to stay in the mood we gave you the SudokuTests.cs file. It is mostly empty for now and up to you to complete it as you advance in this practical.

5.2 Our implementation

Our implementation is quite simple, we have a single attribute: **Board**. It is represented using a 2 dimension array of int. The possible values are between 1 and 9 if a value has been set, 0 if it hasn't been set.

6 Code to write

6.1 First constructor

```
1       public Sudoku(int[,] board)
```

First constructor for the Sudoku class takes a board as parameter.

6.2 IsBoardValid()

```
1       public bool IsBoardValid()
```

Returns true if the board does not infringe any rules, false otherwise.

Tip

At this stage, the board is able to contain any number of 0.

6.3 IsSolved()

```
1       public bool IsSolved()
```

Returns true if the board is solved, false otherwise.

Tip

At this stage, no 0 should be present on the board.

6.4 Solve()

```
1       public bool Solve()
```

As you might have guessed, the goal of this function is to solve a Sudoku. There are many ways to approach this problem, the easiest by using backtracking.
Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, which incrementally builds candidates to the solutions, and abandons each partial candidate

("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution. This gif is a great example.

In other words, backtracking consists in going through every single possibility as long as the predicate (i.e., rules) are not infringed. Once they are, you simply revert to a previous state and take another possible solution. In our case, you use a different number in a certain square. To go further : https://en.wikipedia.org/wiki/Backtracking

The function must return **true** if the board has been solved, **false** if not.

6.5 Second Sudoku constructor

```
1 public Sudoku(int difficulty)
```

Implement a constructor that will create a new Board. It will contain **difficulty** values, placed randomly.

Warning

Be careful, the generated board must be valid and solvable.

6.6 Play()

```
1 public static void Play()
```

It is now time to let a user play the game. The **Play** function will first ask the user for the difficulty at which he desires to play between **Easy**, **Medium** and **Hard**, by displaying '**Choose a level of difficulty (Easy, Medium, Hard):** '.

It will then generate a new object of class Sudoku using the difficulty (see Second Sudoku constructor), the difficulty values being:

- Easy : 45 cases filled
- Medium : 30 cases filled
- Hard : 20 cases filled

Then, until the player has solved the Sudoku, the grid must be displayed (using the **Print** method), and the user input must be prompted using '**Coordinates and value:** ', and read using format **X Y DIGIT**.

If the format is not respected, you must print '**Invalid input**' in the standard error stream, and prompt the user again for coordinates (first displaying the grid again).

Finally, when the user has solved the grid, you must display '**You won**'.

Warning

When prompting the user for input, add a space after the colon, but no line break.

There is nothing more deceptive,

than an obvious fact.