Assignment 3 – General Description, Requirements and Guidance.

1. Objectives

The main objectives of this assignment are:

- To develop a Windows Forms-based desktop application with graphical user interface (GUI).
- To implement input validation to ensure accurate results and prevent unexpected program termination.
- To implement getter and setter methods, along with parameterized methods and methods with return values, to facilitate interaction between objects from different classes.
- To create and utilize enumerations to bundle related constant values for better code structure and readability

2. Introduction:

Up until now, we have been working with console applications to learn and test the basic syntax and structures of the C# language. Now, it's time to move on to more practical cases and start creating applications with a graphical user interface (GUI). We start by using some of the most commonly used Windows Form controls.

A key objective of this assignment is to establish a clear separation between the presentation layer (GUI) and the logic (calculation and data manipulation). In previous assignments, user interactions occurred within the same class. However, moving forward, all user interactions (input and output) will be handled exclusively by the GUI class (Form object). Meanwhile, the logic for processing input data and generating output will reside in separate classes, which the GUI will utilize as needed.

In our console applications, we had classes responsible for both calculations and user communication. This was necessary because, at the time, we were just starting to learn the language's syntax and programming fundamentals, making it impractical to separate these concerns.

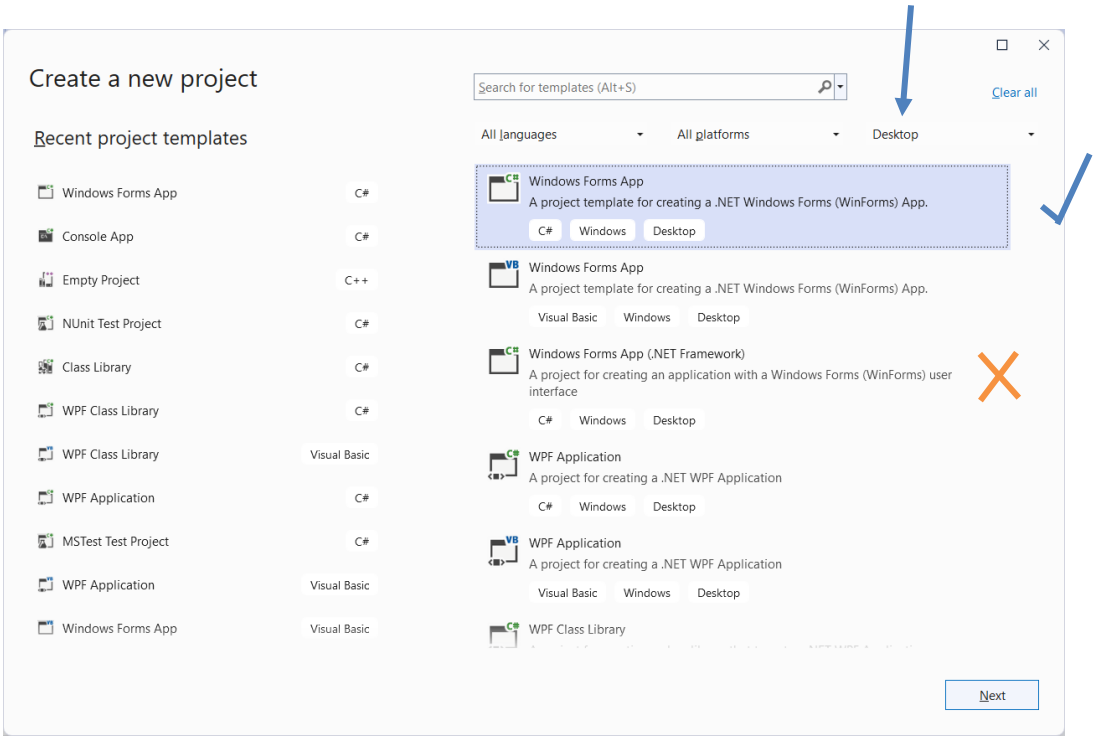
However, a more object-oriented approach is to have a user interface class (UI) manage all user interactions, while other classes focus on different tasks like calculations or data processing. Now that we are ready for this transition, we will begin creating a Form object (MainForm) to serve as the main user interface. This will allow us to decouple the logic from the user interactions, ensuring the Form class is solely responsible for communicating with the user, while other classes handle specific responsibilities.

Design your classes to be as self-contained and independent as possible. Each class should receive input, process data, and provide output exclusively through methods, without any direct user interaction. This ensures that the classes focus on their specific responsibilities and

remain decoupled from the user interface, promoting better modularity and maintainability in your code.

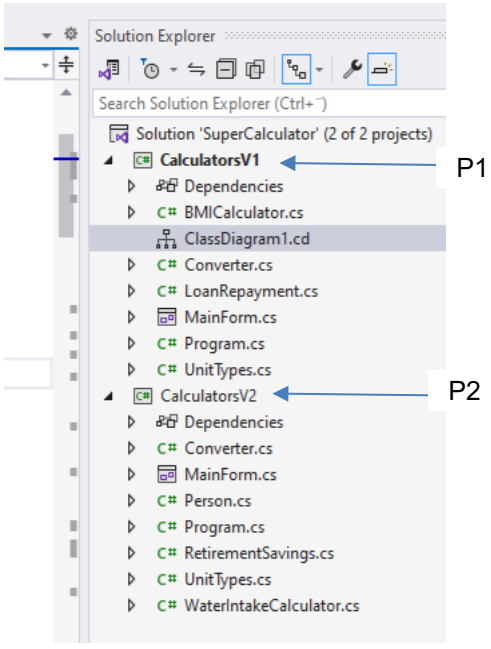
3. Create a Windows Form Application

Start by launching your development environment, Visual Studio (VS), and creating a new project for this assignment. Select **.NET Desktop App** as the project type, then choose **Windows Forms App** as the template for building your graphical user interface (GUI).:



3.1 Solution and projects

When creating a new Windows Forms project, Visual Studio (VS) automatically generates a new solution along with a new project. It also prepares a startup form in the project, where you can design your graphical components (commonly referred to as "controls"). This form is typically assigned a default name, such as Form1 (or MainForm in the provided image).



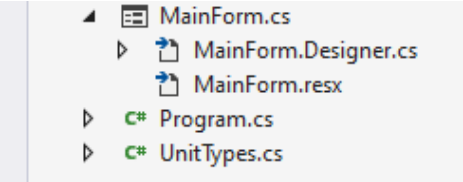
A solution can contain one or more projects. In the example, the solution includes two projects. You can combine different types of projects within the same solution, such as a Console project and a Windows

Each form in Visual Studio (VS) has two parts: a visual part and a code part. The visual part is where

you design the graphical user interface (GUI) using tools like the **Toolbox**, **Properties** box, and other graphical utilities available in VS. The code part is used to program the behavior and functionality of the GUI.

VS automatically generates a file named xxx.Designer.cs, where xxx is the name of the form (e.g., MainForm.Designer.cs). This file is used by VS for generating the code behind the visual design, such as creating components, initializing them, and handling events.

Important: Do not modify the code in the Designer file unless you encounter compilation errors related to that file, as changes in this file can disrupt the auto-generated code and may cause issues with the form's design.



The Form class should handle all user interactions within the application, including both input and output. All data processing and manipulation should be encapsulated within their respective classes to maintain a clear separation of concerns.

For clarity and better organization, it is recommended to give all classes, including forms, meaningful and descriptive names. For instance, you can rename Form1 to MainForm to signify that it is the startup form of the application. If additional forms are needed, name them according to their specific purpose, making the project structure more perceptive and easier to manage.

Important hints:

- Renaming a Form: In recent versions of Visual Studio, renaming a form can sometimes cause unexpected behavior, such as GUI elements disappearing (likely due to a bug). To avoid this, it's recommended to rename the form before adding many controls.
- Closing the Form Designer: It is good practice to regularly close the Form Designer, especially if it fails to load the form correctly. Make sure to close both the Form Designer and the associated code file (the .cs file) to prevent loading issues and improve performance.

When you are ready to design the GUI, start by considering the input parameters your application requires, as these will typically be provided by the user. Additionally, determine the output your application will calculate and display on the GUI. Keeping these input and output requirements in mind will help you design your classes.

Use appropriate Windows Forms controls available in the Toolbox in Visual Studio to create an intuitive and functional interface.

TextBox vs Label

It's important to note that **TextBoxes** are intended for user input, whereas **Labels** are meant for displaying read-only data. All output displayed by the program should be read-only, preventing users from altering the results. Additionally, **TextBoxes** have a higher overhead compared to **Labels**, making **Labels** the preferred choice for displaying output.

RadioButtons and GroupBox

When using **RadioButtons**, it is important to encapsulate each set within a container component, such as a **GroupBox**. This is necessary because, if multiple sets of **RadioButtons** are placed directly on the same Windows Form without a container, they will be treated as a single group. As a result, selecting one **RadioButton** will automatically deselect any other **RadioButton** in that group.

Although your current GUI design may only include one set of **RadioButtons**, it is a good practice to always enclose them in a **GroupBox** to ensure correct behavior and maintain consistency in future designs.

ComboBox

A **ComboBox** consists of two parts: a text box and an arrow for selecting from a predefined list of values. If you are not handling input for the text box part of the **ComboBox**, it should be set to **read-only** to prevent the user from entering text manually. Instead, the user should be restricted to selecting values from the drop-down list.

To make a **ComboBox** read-only, simply change its **DropDownStyle** property to **DropDownList** in the Properties window in Visual Studio. This ensures that users can only choose from the predefined options.

Two Alternatives for filling a ComboBox with values from an enum::

Consider the enum **ActivityLevel**:

```
public enum ActivityLevel
{
    Low,
    Medium,
    High
}
```

Alternataive 1:

```
cmbActivityLevel.DataSource = Enum.GetNames(typeof(ActivityLevel));
```

Alternative 2:

```
cmbActivityLevel.Items.AddRange(Enum.GetNames(typeof(ActivityLevel)));
```

In both cases, it's a good idea to select a default option so that the ComboBox is ready to use from the start:

```
cmbActivityLevel.SelectedIndex = 1; //ActivityLevl.Medium
```

A better alternative:

```
cmbActivityLevel.SelectedIndex = (int)ActivityLevel.Medium;
```

4. General requirements valid for all versions of the assignment

The calculator classes should not interact with the user in any way, including displaying messages via MessageBox. These classes should receive input through setter methods, perform the necessary calculations, and provide output via return values or out-parameters of methods. Getter methods can also be used to retrieve data from the class objects.

4.1 Input control

When converting the contents of text boxes to numerical values, use the **TryParse** method instead of **Parse** of the numerical type. This ensures that the program will not crash if the user enters a value in an incorrect numerical format. The TryParse method allows for safe conversion by returning false if the input is invalid, which enables you to handle the error properly.

```
private bool ReadLoanPeriod()
{
    int value = 0;
    bool ok = int.TryParse(txtPeriod.Text, out value);
    if (ok && (value > 0))
        loanRepayment.SetPeriodInYears(value);
    else
        MessageBox.Show("Invalid value for the loan period; value must be an integer > 0!");

    return ok;
}
```

For converting a **string** to a **double** value, use **double.TryParse**. The following code can be used to convert a string (e.g., txtHeight.Text) to a double, provided the string represents a valid numerical format.

```
private double ReadDouble(string str, out bool success)
{
    double value = -1.00;
    success = false;
    if (double.TryParse(str, out value))
        success = true;

    return value;
}
```

The method **string.IsNullOrEmpty**(string) is very useful for validating whether a string is either empty or uninitialized. The following code snippet demonstrates how to read the user's name from a textbox on the form, named **txtName**

```
private void ReadName ( )
{
    txtName.Text.Trim ( ); //Delete spaces at the beginning and end of the string
    if (!string.IsNullOrEmpty ( txtName.Text ))
        bmiCalc.Name = txtName.Text;  //no need for any type conversion
    else
        bmiCalc.Name = "Unknown";
}
```

4.2 Displaying output

Formatting values with the $ symbol in a string literal is a convenient way to automatically convert numerical values into their textual representations. Here is an example

```
private void DisplayLoanRepaymentResults()
{
    double value = loanRepayment.GetMonthlyRepayment();
    lblMonthlyTotal.Text = $"{value:N2}";  //value.ToString("0.00");
    Continue with the rest

}
```

The "N2" format specifier is used to display a value with two decimal places and separate the thousands with commas (or the appropriate delimiter based on regional settings)

“N2” is a formatting specifier that will display the value with 2 decimal places separating the thousands. “N0” can be used when no decimal places are required.

Here is another example:

```
double totalAmount = 1234.56;
lblTotalAmount.Text = $"Total: {totalAmount:C}";
```

In this example, the $ symbol is used for string interpolation, and the **:C** format specifier converts the numerical value to a currency format, adding the appropriate currency symbol (e.g., kr for SEK, $ for USD), based on the regional setting in your operating system.

4.3 Fields

Declare instance variables only for user input, and ensure that all instance variables are **private**. Use setter and getter methods to provide controlled access to these fields. Strive after keeping the number of fields as low as possible and avoid using instance variables to store output data if it can be returned by a method.

In this assignment, instance variables for output values are unnecessary, as output will be returned by methods. However, you are encouraged (and it may often be necessary) to use local variables (variables inside a method) as needed. You can use the same variable types and names in multiple methods, as appropriate

Calculator classes should be focused purely on processing input and returning results, while MainForm is responsible for gathering user input, displaying output, and interacting with the user.

The MainForm class (GUI) should only include instances of the calculator classes and any values that are not part of those classes. For example, user-specific data, such as the user's name in the context of a BMI calculator, can be handled within the MainForm class, because this data does not inherently belong to the BMI calculator class. This ensures a clear separation between the GUI and the logic of the calculations. However, if necessary, it is acceptable to move the declaration of the user's name (or similar variables) to a calculator class if it simplifies the design or makes sense within the context of that class.

```
public partial class MainForm : Form
{
    private string name = "NoName";

    //Declare and create an instance of the BMI Calculator
    private BMICalculator bmiCalc = new BMICalculator();

    //Declare and create an instance of the LoanRepayment Calculator
    private LoanRepayment loanRepayment = new LoanRepayment();

    1 reference
    public MainForm()
    {
        InitializeComponent();
        InitializeGUI();      //Your initializations.
    }
```

Note: The code snippet above is just an example, but it illustrates a pattern that should be followed in all versions of the assignment.

The first method in the **MainForm** class is its constructor, where you will find a call to the method **InitializeComponent**. This method is generated by VS. Any additional initialization code or method calls should be inserted **after** this statement to ensure that the form is properly set up before it is displayed. For example, you might write and utilize a method named **InitializeGUI** to handle various initialization tasks, such as setting default values, configuring controls, or preparing data.

To modify the properties of a control in Visual Studio, first view the form and click on the component you wish to adjust to ensure it is selected in the **Properties** window. You can then make changes to the control's properties. For example, clicking on an empty space on the form will select the form itself, allowing you to modify the following properties of **MainForm**:

- **FormBorderStyle**: Choose one of the fixed types from the available options. This setting will define the border style of your Form.
- **MaximizeBox**: Set this property to **false**. This change will disable the maximize button on your Form, fixing its size.
- **StartPosition**: Select **CenterScreen**. This setting ensures that your Form will be centered on the screen when the form is loaded.
- **Text**: Provide a title for your program that includes your name, for example, "Super Calculator by [Your Name]". This title will be displayed in the title bar of the Form.

Most of the properties of the controls can also be modified programmatically in the code-behind class. This allows you to dynamically adjust control properties at runtime, providing greater flexibility and control over your application's behavior.

For example, you can set properties like FormBorderStyle, MaximizeBox, StartPosition, or Text in the constructor or other methods (InitializeGUI in the code below) of your MainForm class:

```
private void InitializeGUI()
{
    this.FormBorderStyle = FormBorderStyle.FixedSingle;
    this.MaximizeBox = false;
    this.StartPosition = FormStartPosition.CenterScreen;
    this.Text = "Super Calculator by Apu";

    lblAmount.Text = string.Empty;
    //other initializations
}
```

This method is called from the constructor of the form.

```
//MainForm's constructor
1 reference
public MainForm()
{
    InitializeComponent();
    InitializeGUI();
}
```

Good Luck!

Farid Naisan

Course Responsible and Instructor