Isaac Basque-Rice

12basque-rice.i@thestudio-liverpool.net

Microsoft Calculator Documentation

A Calculator to demonstrate the Event Driven Programming paradigm

[Introduction 2](#_Toc7730637)

[Demonstrating the use of EDP features and functions 2](#_Toc7730638)

[Explaining the importance of accessing database structures to solve set problems 2](#_Toc7730639)

[Interpreting the correct usage of a complex third-party function 3](#_Toc7730640)

[Designing an EDP Application 5](#_Toc7730641)

[Selecting an appropriate design tool and planning a solution 5](#_Toc7730642)

[Interpreting a design to deduce the correct event driven controls and handlers 8](#_Toc7730643)

[Implementing the EDP Application 9](#_Toc7730644)

[Carrying out EDP development with a range of IDE facilities 9](#_Toc7730645)

[Using appropriate features and functions to carry out EDP using correct coding standards and conventions 30](#_Toc7730646)

[Using appropriate debugging tools to identify and resolve programming faults 39](#_Toc7730647)

[Documenting an event driven programming solution 41](#_Toc7730648)

[Deploying an EDP application as a professional installation 51](#_Toc7730649)

[Reviewing an EDP solution against the client specification 56](#_Toc7730650)

[Evaluating the EDP Solution using client feedback to calculate an effective maintenance and update strategy 57](#_Toc7730651)

# Introduction

In order to show that my programming skills are up to scratch, as it were, with regards to programming standards and conventions, Microsoft have tasked me with developing a small and extremely simple event-driven application, a calculator. This calculator must have functionality including, but not limited to, adding, subtracting, multiplying, and dividing numbers, but will almost certainly contain functionality to clear, backspace, and do arithmetic with both positive and negative numbers.

For this application I will be using Visual Studio as my IDE, C# as my logic language, and WPF/XAML for the front end. This setup gives an excellent environment for developing this application in the method I am used to and seems to work well for me.

# Demonstrating the use of EDP features and functions

## Explaining the importance of accessing database structures to solve set problems

In the instance of a calculator, accessing database structures could theoretically be fairly important, in several more advanced calculators, real-world scientific and otherwise, the function for accessing previously stored results tends to be in a button labelled ANS, the Microsoft standard calculator goes a step further and, unless it is cleared, will store the entire history of formulae and result in a separate box, able to be revisited at any time.

This, no doubt, can be achieved by making use of some form of database structure, the storing of data detached from the application itself in a connected yet separate structure results in a significant increase in performance as far as RAM usage is concerned, as without the use of these structures all data relating to the calculator must be stored in RAM.

In addition, the nature of RAM as a volatile data storage method means that once the app is closed or the device is shut off, all data relating to the app is gone, so in a more complex calculator application allowing users to store and come back to a formula later my serve as extremely useful.

More generally, the use of accessing a database structure could, in theory, be the accessing of an account for an application. The accessing of this account could lead to some features which may or may not be available to subscribers or members of the application only, the accessing of the structure could serve as a method of authenticating these premium users, subscribers, and so on and so forth.

## Interpreting the correct usage of a complex third-party function

For this solution the majority of the backend content was much too simple to comfortably deploy a complex third party function, as a result of this, I have decided the best use of a third party function in this case would be MaterialDesignInXAML made by several members of the GitHub community, available here: <https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit>

Below is an example of how it has been used in-application, it (fairly helpfully, I might add) highlights the button the user has pressed through the use of Google’s trademarked way of highlighting in material design, a sort of wave of light radiating from the point that the user has clocked, as so:

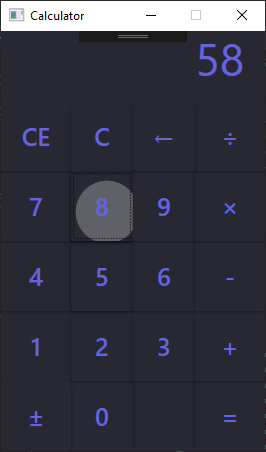


Figure 1, the version of the app that has material design loaded, note the highlighting of the number 8

In order to make use of this, I had to import the toolkit using the ResourceDictionary tags in the App.xaml file in the solution, specifically, I imported the dark theme and the default themes, which were required for the toolkit to function properly, below is a copy/paste of what I did:

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="pack://application:,,,/MaterialDesignThemes.Wpf;component/Themes/MaterialDesignTheme.Dark.xaml" />

<ResourceDictionary Source="pack://application:,,,/MaterialDesignThemes.Wpf;component/Themes/MaterialDesignTheme.Defaults.xaml" />

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Application.Resources>

Figure 2, my app.xaml page after using a library

Additional content this third party function ads is relatively sparse, at least as far as this app is concerned, however it does allow for significant aesthetic improvements which grants my app a sort of leg up with regards to competitors, additionally, from a usability and accessibility standpoint it allows for those users who may not be especially dextrous to see more clearly what they have pressed, through the highlighting function, as well as those who are visually impaired due to the stark contrast between the dark background and the light highlight colour.

Third part functions for deployment specifically are ClickOnce, XCopy, and the Windows Installer the latter of which is shown later on in the section entitled “Deploying an EDP application as a professional installation”, as such, the former two will be explained here in some detail:

### ClickOnce

ClickOnce (<https://docs.microsoft.com/en-us/visualstudio/deployment/clickonce-security-and-deployment?view=vs-2019>) Is a deployment technology made for use primarily with Windows Forms and Windows Presentation Foundation software that allows the developer to deploy and update their application as they so choose, it is a component of the .NET Framework and was introduced in version 2.0 in order to aide with deployment. Below is an image of a ClickOnce installation process

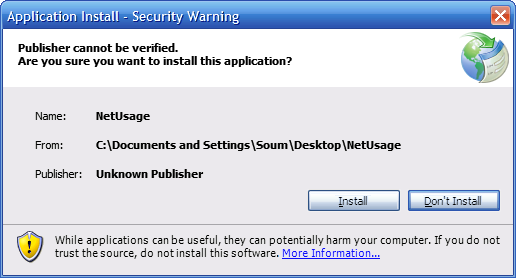


Figure 3, An image of a ClickOnce installation process

This form of deployment is primarily used for web-based deployment of non-web apps, it can be published to or downloaded from a web or file server.

### XCopy

XCopy is a command-line program most suitable for smaller programs that do not require updating, that is, that there is a single upload date and that’s the end of the project. Applications must adhere to the following guidelines:

* It must be self-contained and not require a separate client to run
* Files must be moved from location A (the developers’ computer or code repo) to location B (a server, hard disk, and so on)
* The application cannot require shell integration, such as start menu shortcut, desktop icon, etc.

# Designing an EDP Application

## Selecting an appropriate design tool and planning a solution

In order to carry out my task successfully and with as little issue as possible and with the highest level of efficiency and effectiveness, it is necessary for me to implement the use of a wide variety of design tools. These design tools do not just encompass front end design, or how the app looks to the user, as one may assume it does, it will also include flow charts, storyboards, and a variety of other tools that will aid me in envisioning a functioning app.

The first tool I will be using to design the solution will be GIMP (<https://www.gimp.org/>), the GNU Image Manipulation Program, a free alternative to Adobe’s Photoshop/Illustrator programs. I intend for the front-end design to have the following two main colours:

|  |  |  |
| --- | --- | --- |
| Colour | Hex Value | Use |
|  | #282833 | Background |
|  | #6564DB | Text (output, buttons, and so on) |

I intend for the calculator itself to be 280px\*420px and to be divided into 24 segments, each 70px\*70px and each housing an individual control (except for the output window, which will take up 4 along the top).

I will need 10 number buttons (0-9), 5 operator buttons (+, -, ×, ÷, and =), a button that allows the user to choose between positive and negative numbers, and 3 clear buttons (CE, C, and Backspace). This will add to 19 buttons total, plus the four spaces needed for output along the top, this will add to 23, leaving one space open.

The following was the result of the front-end design effort, note the guidelines, which were placed at intervals of 70px beginning at 0 both horizontally and vertically, this resulted in the desired grid effect that will serve to emulate the grid framework in the application itself:

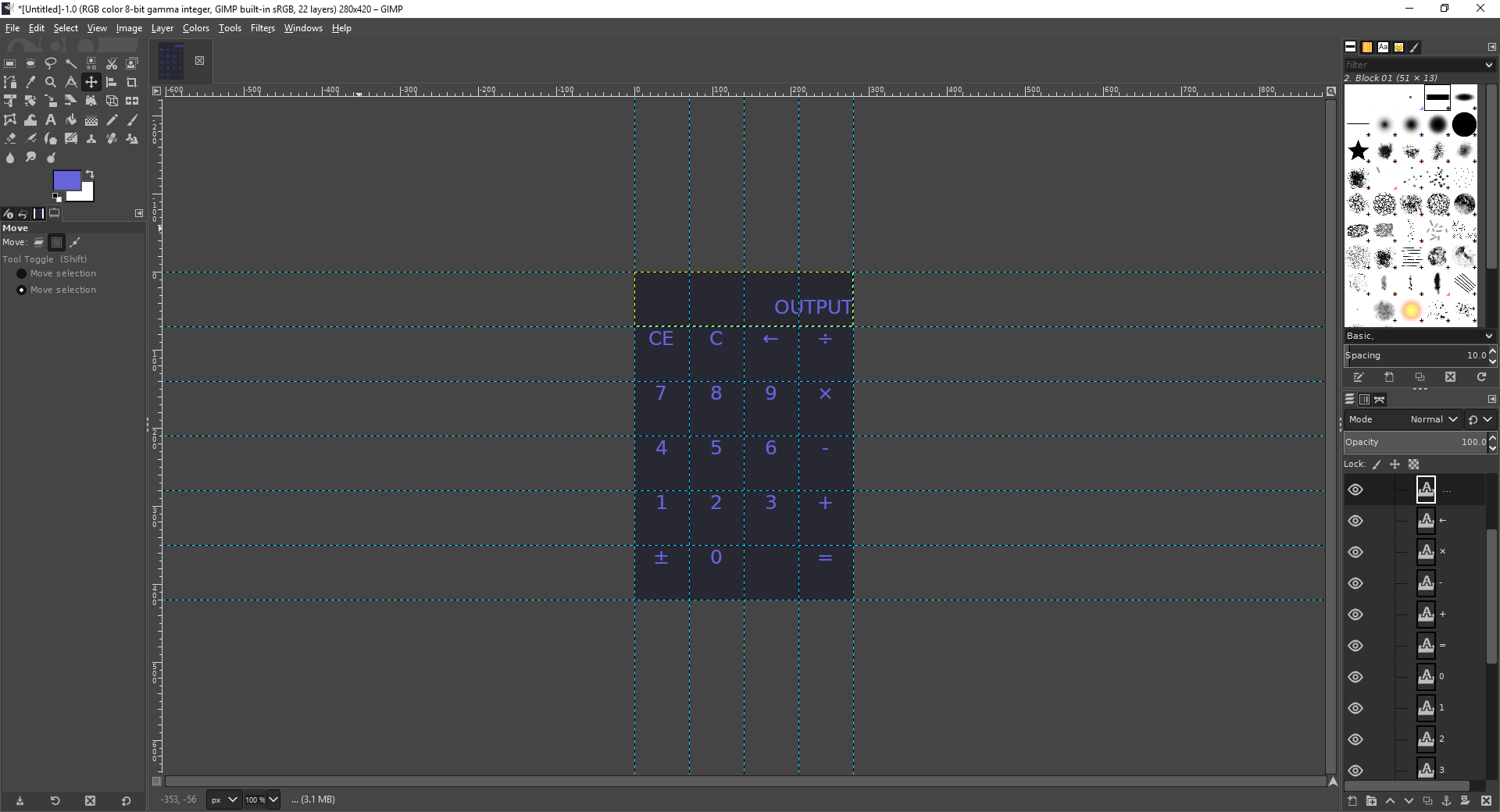


Figure 4, a screenshot of the calculator as it was immediately after being built in GIMP

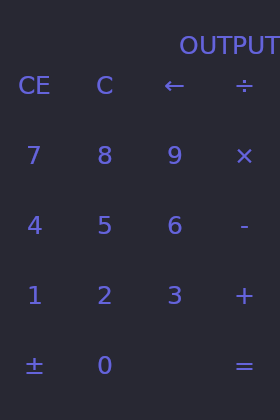


Figure 5, the calculator's front end as it will be when built

Further designs, such as flow charts, story boards, and so on, are unnecessary due to the fact that EDP is, by design, non-linear and cannot be storyboarded, a user can press any number of buttons in any order they wish, theoretically. Additionally, due to the simplistic and pre-defined nature of the application (in that everybody knows what a calculator does), there is no further design consideration necessary.

## Interpreting a design to deduce the correct event driven controls and handlers

Now it is important that the events are designed in a way that allows for me, as the developer, to implement them fairly seamlessly into the solution, to achieve this I will be making use of tables that correspond to the events I will be programming in the final solution.

The following is the table that corresponds to each of the buttons in the solution, note that I will also be creating three variables called number1, number2, and op, the former two will be the two numbers the user will input into the solution and the latter, “op” being the operator (plus, minus, etc.)

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Category | Description | Handler |
| Btn\_NumX | Numbers | If there is no operator, take number1, multiply it by 10 and add X, else, do the same with number2, this will make it appear as if a number is being added to the end of the output window (2, 23, 234, and so on) | OnClick |
| Btn\_OpX | Operators | Sets the value of op to be equal to either +, -, ×, or ÷, remembers that, and then sets the display value to 0 | OnClick |
| Btn\_OpEquals | Operators | Depending on the value of op as determined by btn\_OpX, will either add, subtract, multiply, or divide number1 and number2, and output the result to the display box | OnClick |
| Btn\_ClrClearEntry | Clear | If the op has no value take number1 and set the value to 0, else, set number2 to 0, display 0 | OnClick |
| Btn\_ClrClearAll | Clear | Set number1, number2, and display to 0, set op to nothing, clears everything from the memory of the calculator | OnClick |
| Btn\_ClrBack | Clear | Divides the number by 10, due to the fact we are dealing exclusively with whole numbers this will have the effect of just removing the most recent number as far as the user is concerned | OnClick |
| Btn\_SymPositiveNegative | Symbols | Multiplies the number by -1, resulting in the number being negated, 5 becomes -5, -2 becomes 2, etc. | OnClick |

# Implementing the EDP Application

## Carrying out EDP development with a range of IDE facilities

At this point it is necessary to actually go through with the development of the application, I will be making use of a number of Integrated Development Environment (IDE) facilities, as any developer is wont to do. The IDE in particular I will be using, as mentioned previously, is Microsoft’s Visual Studio 2017 (<https://visualstudio.microsoft.com/>), I will be making use of the WPF framework (<https://docs.microsoft.com/en-us/dotnet/framework/wpf/getting-started/>), the UI framework that is part of the .NET stack, C# which serves as the back-end of said stack, and the integrated facilities used to accommodate all of this.

### Integrated Development Environment (IDE)

The facilities I will be using in the IDE all fall into one of three categories, the code editor, the debugger, and the designer, what follows is a more comprehensive look at the three.

The code editor is perhaps the most familiar aspect of the IDE to the average person, this is the part of the IDE where the code is written and edited, this part is where the majority of my time will most likely be spent during the course of this project, editing buttons and other controls that have been set out in the designer, as well as, arguably more crucially, programming the logic and event handlers for each individual button.

Helpfully, as with other IDEs, the code editor in Visual Studio supports a number of facilities that assist the developer in carrying out their task. Examples of these facilities include syntax highlighting, which highlights certain keywords (such as private, void, if, else, and so on), as well as text in speech marks, which are relevant in C#, in order for the developer to understand better what is going on. Additionally, Code Completion using IntelliSense, or Intelligent Code Completion, is an extremely useful tool built into Visual Studio which essentially tries to predict what the developer will write next and automatically fills it, this increases efficiency significantly.

Visual Studio’s code editor also includes collapsing code blocks, which allow for improved navigation, error highlighting and refactoring using background compilation, which compiles code as you are writing it to give live feedback, and a slew of other features such as searching and search and replace which improve quality of life and efficiency many times over.

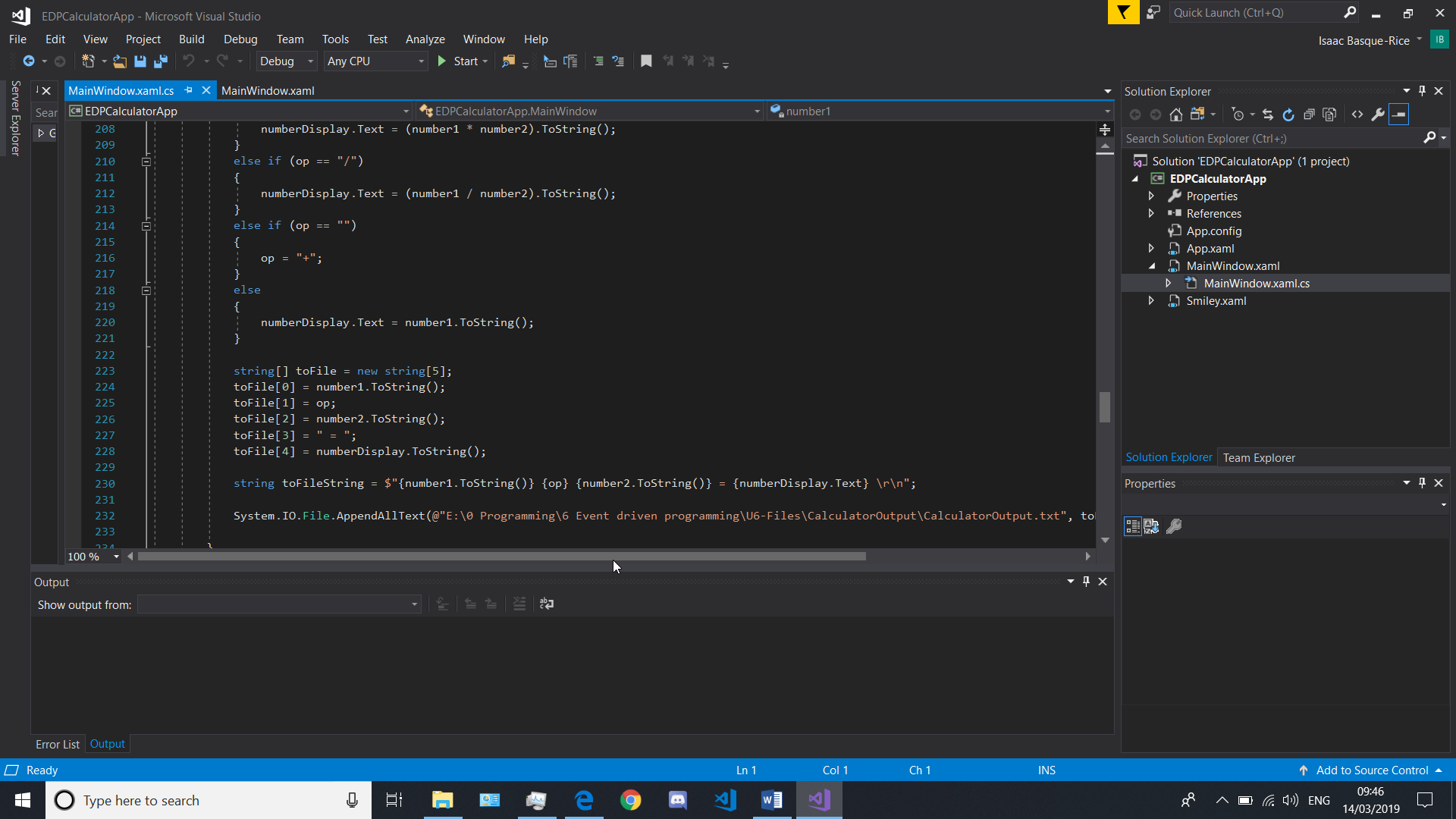


Figure 6, The code editor in Visual Studio displaying C#

The Debugger is yet another key feature of the IDE, in layman’s terms, it builds and runs the application the developer has built so they can make changes and remove bugs (de-bug) the app. This is no doubt an extremely useful function for obvious reasons, but an in-built feature of the VS debugger is the ability to add breakpoints, points at which a program is paused in order for further analysis to take place. This system has a fair few specifics that can be incredibly helpful mid-development as no doubt will be demonstrated later.

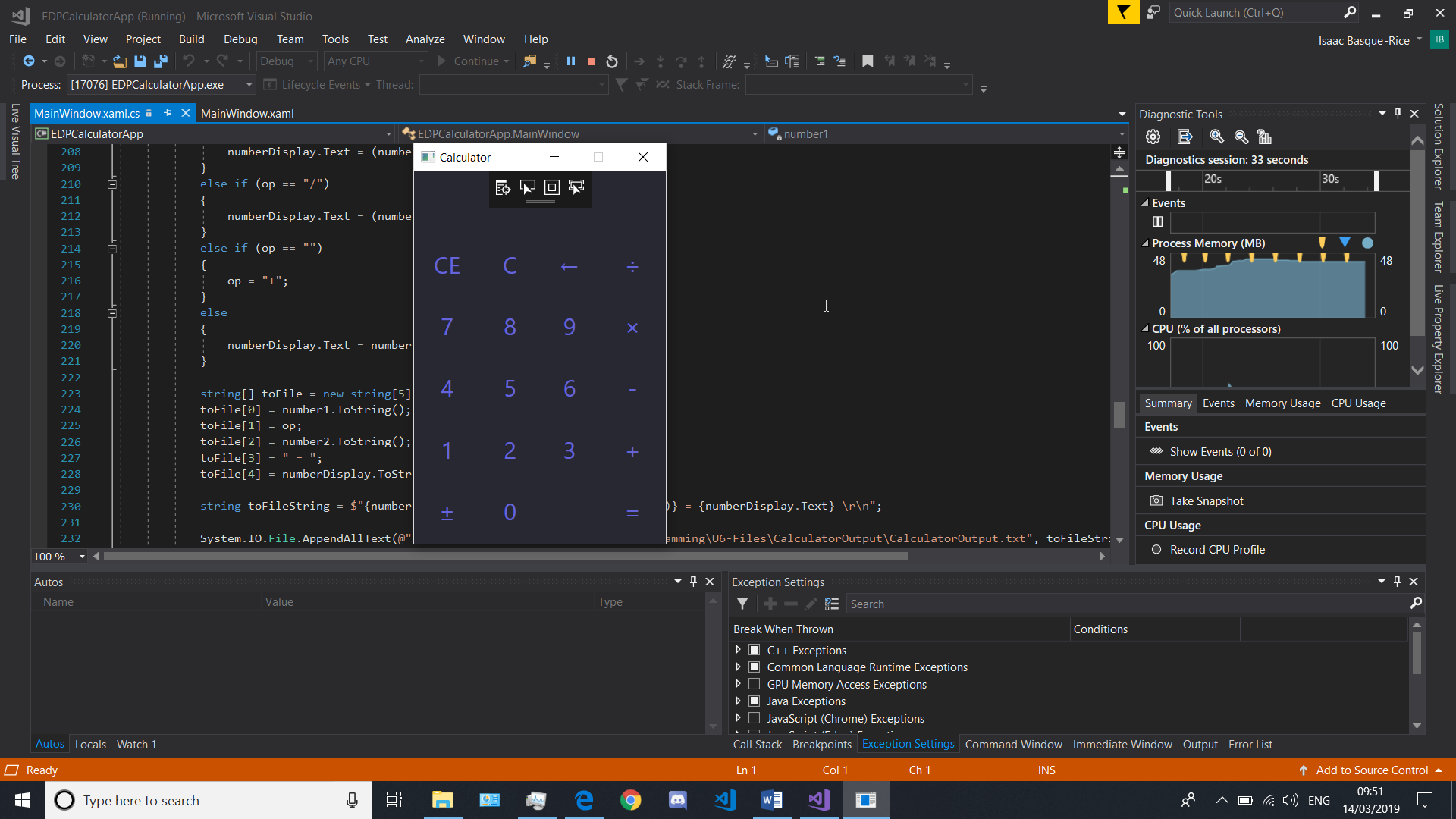


Figure 7, The IDE as it appears when it is debugging, note two things: The app funning in the middle of the screen, and the diagnostics window, which pops up when the developer debugs and allows the developer to understand what is going on "under the hood"

Finally, we come to the designer, specifically, in this instance, the WPF designer. This is a tool that allows the developer to implement features in a what-you-see-is-what-you-get style manner by supporting the drag and drop metaphor. Simplified even further than that it simply allows the developer to, instead of writing code for each individual button, get a button from a toolbox full of pre-defined controls, drag them into the window and drop them and resize them in a manner that the developer wishes, additionally, the developer can double click on the control to generate an empty event handler, which is beyond helpful in the everyday development context.

Helpfully, this designer then allows for multiple ways of editing the controls, primarily through two methods, the properties window and the ability to edit XAML behind, both of which offer helpful methods of development. However, Properties takes a slightly more user-friendly, if slightly developer unfriendly way of doing things, helpful in that it gives the developer categories in boxes in which they can type parameters, but unfriendly in that it disregards formatting in the XAML code. Ultimately, a roughly equal use of both designer and code editor is seen as healthy during development.

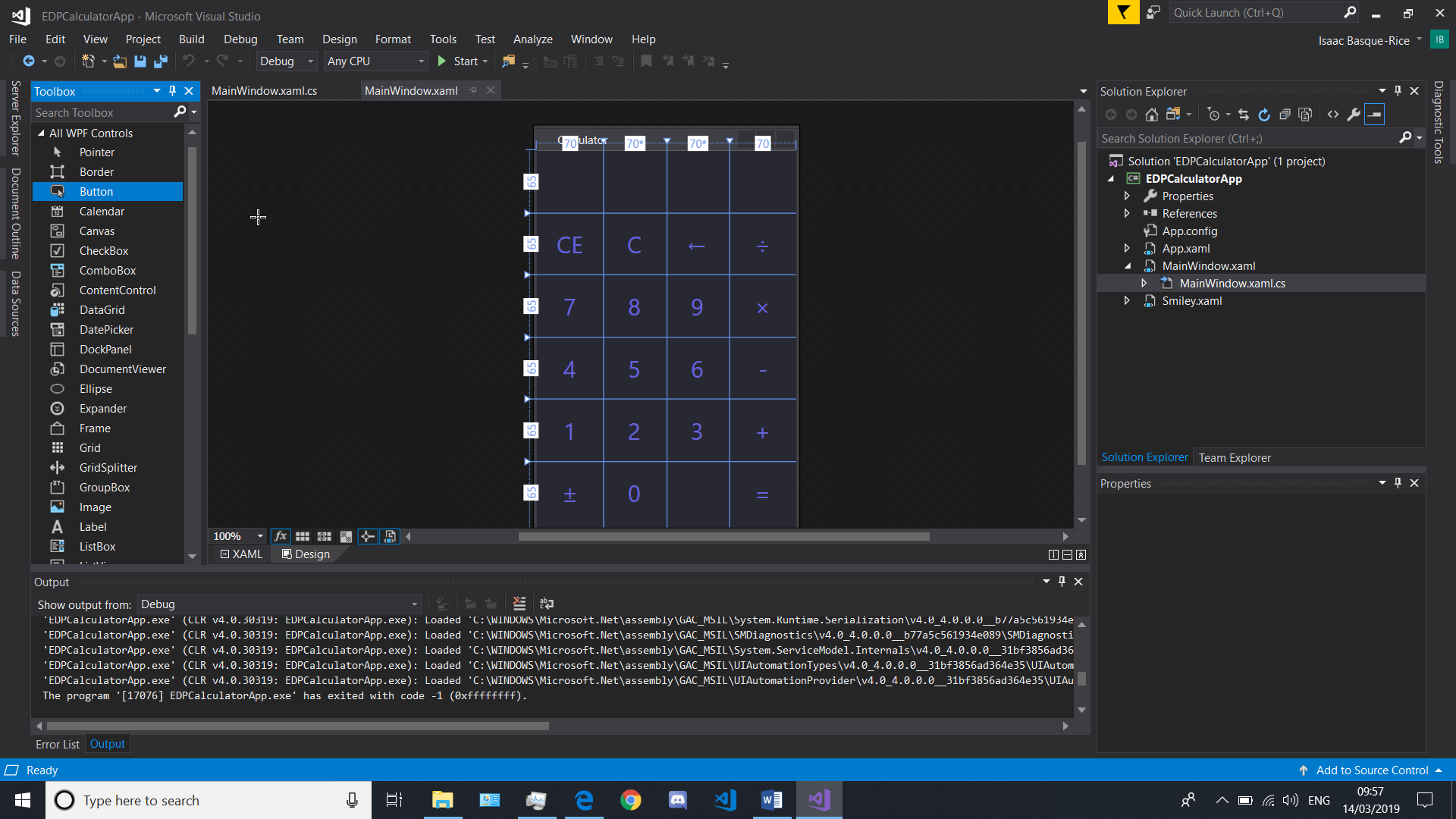


Figure 8, the designer with the toolbox in the left-hand side

The IDE itself, also helpfully, allows for the developer to adjust and customise the environment to the taste of the developer in question. For example, windows can be docked in a multitude of different positions around the IDE.

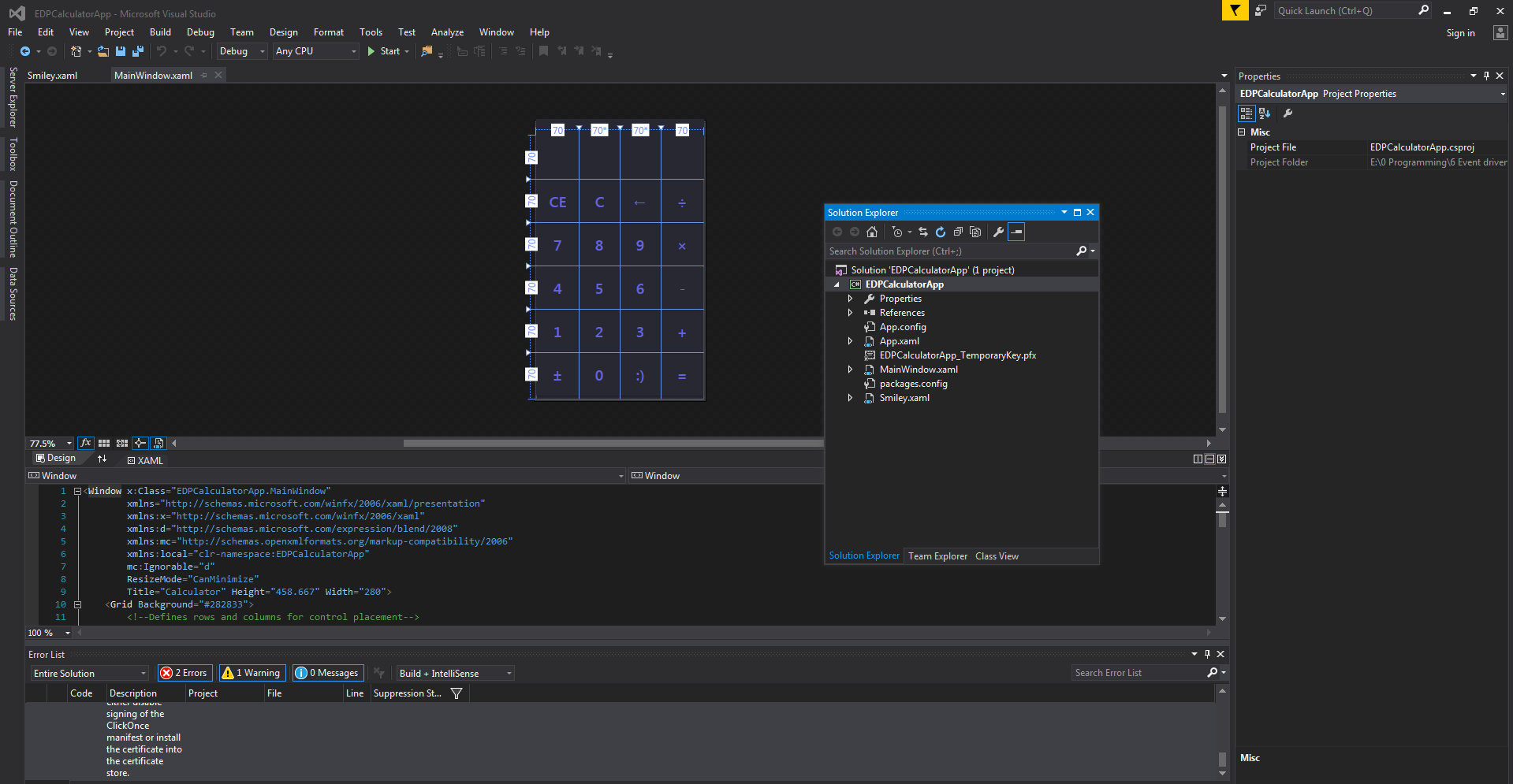


Figure 9, here you can see the Solution Explorer window (fully functionally) in the middle of the screen, the window can also be snapped to different positions (bottom, centre, side, etc.) through the IDE's facilities

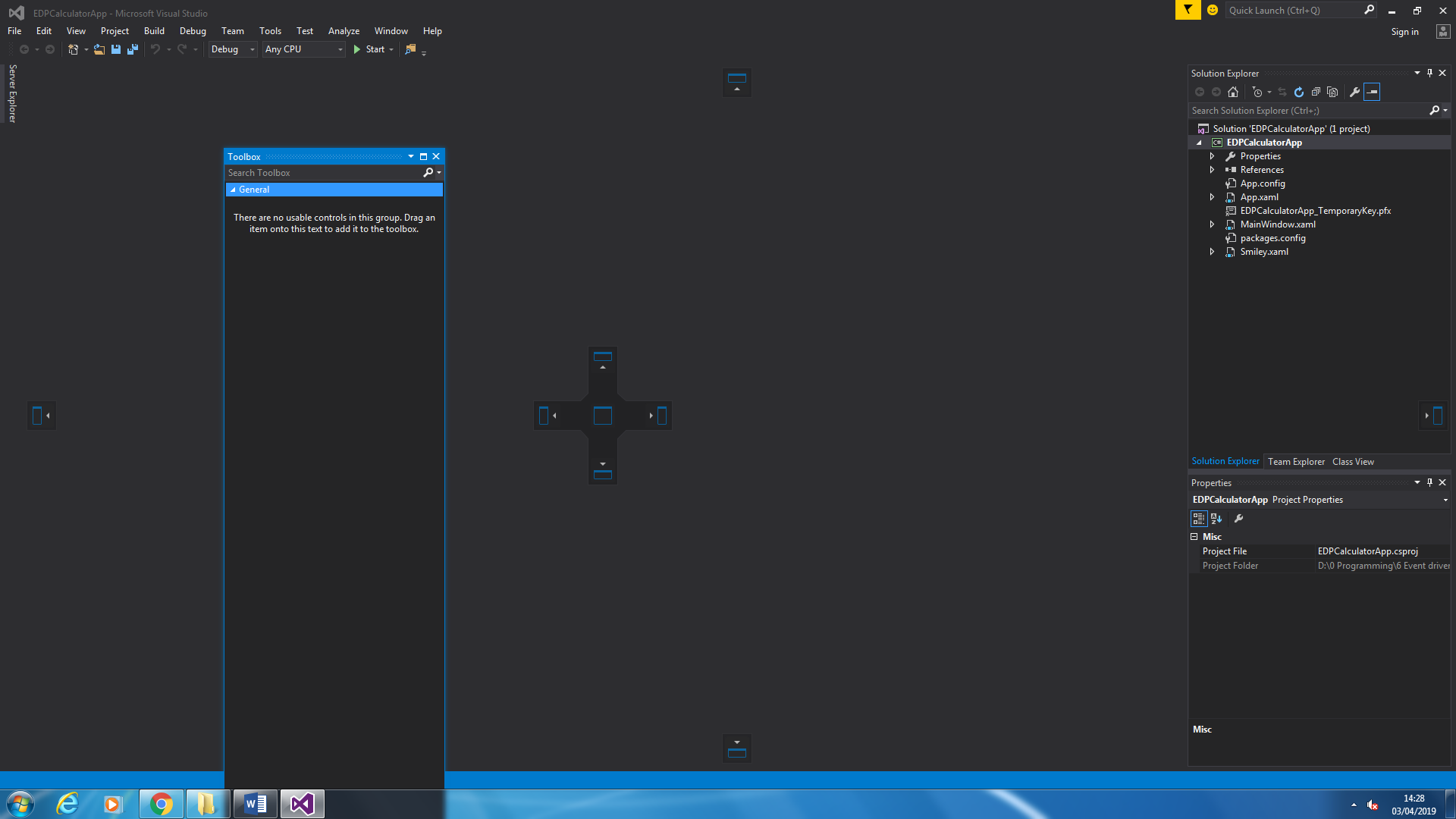


Figure 10, the docking icons in visual studio when dragging the toolbox window

#### Workspaces

The Workspace is a file that allows the developer to gather the relevant source code files into a single collected unit in order to develop a solution. The Visual Studio workspace encapsulates the entirety of the development environment and is very helpful in instances where a project is quite large or requires some maintenance. In short, the workspace is everywhere where a developer can work free of external input for any length of time until their task is completed.

### Creating a New Project

In order to fully demonstrate the power of the IDE, I decided it would be best for a demo app to be created from scratch. Let’s make it a simple app called “demo” which will open a window that says “this is a demo” when you press a button. It will also open a window with an image in it in order to demonstrate the addition of files to a project.

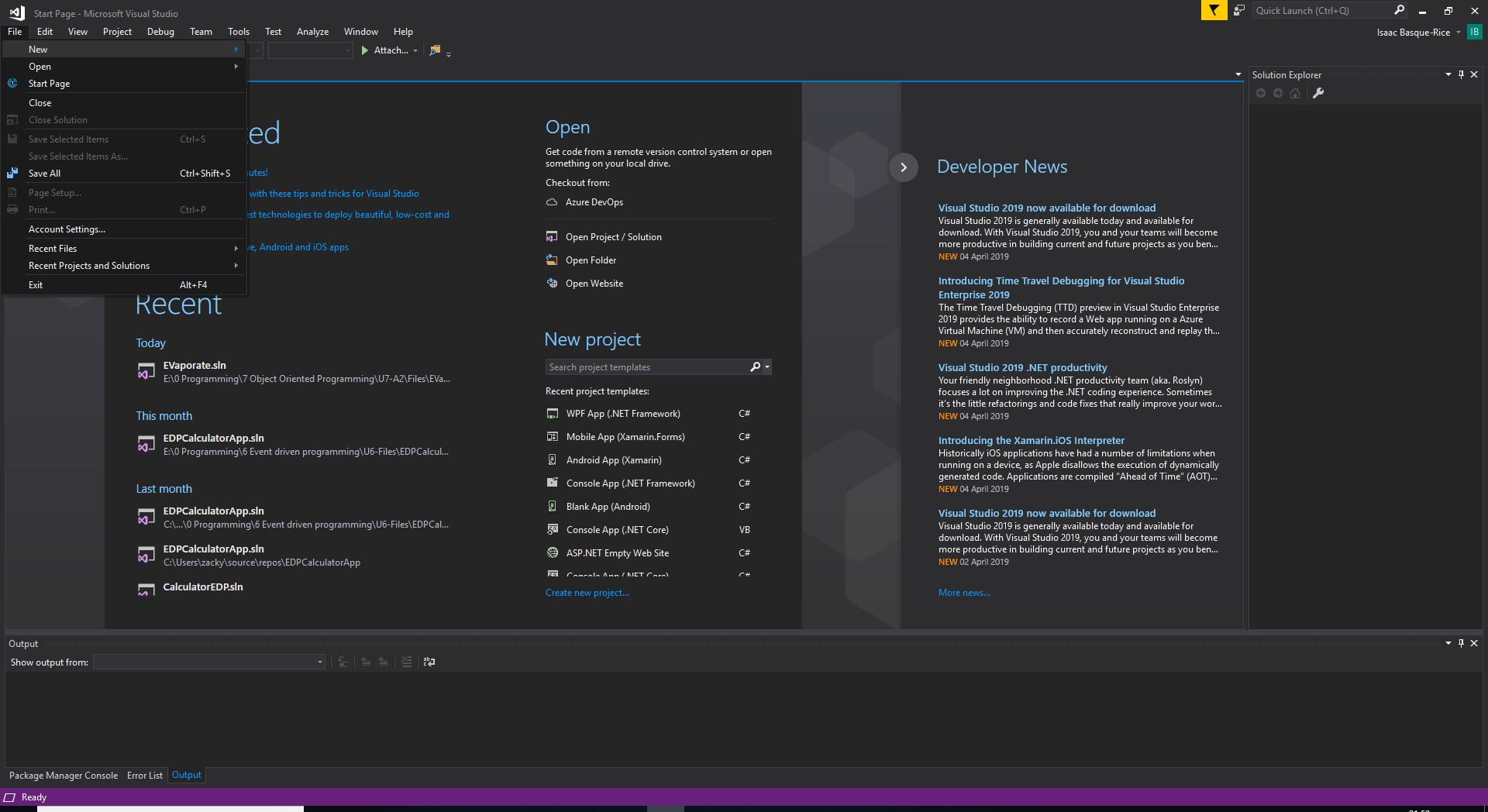


Figure 11, the create new project option



Figure 12, the New Project window, here you can see I've selected what kind of app I want it to be, it's name, where it's saved, and the framework it's targeting. I then pressed OK

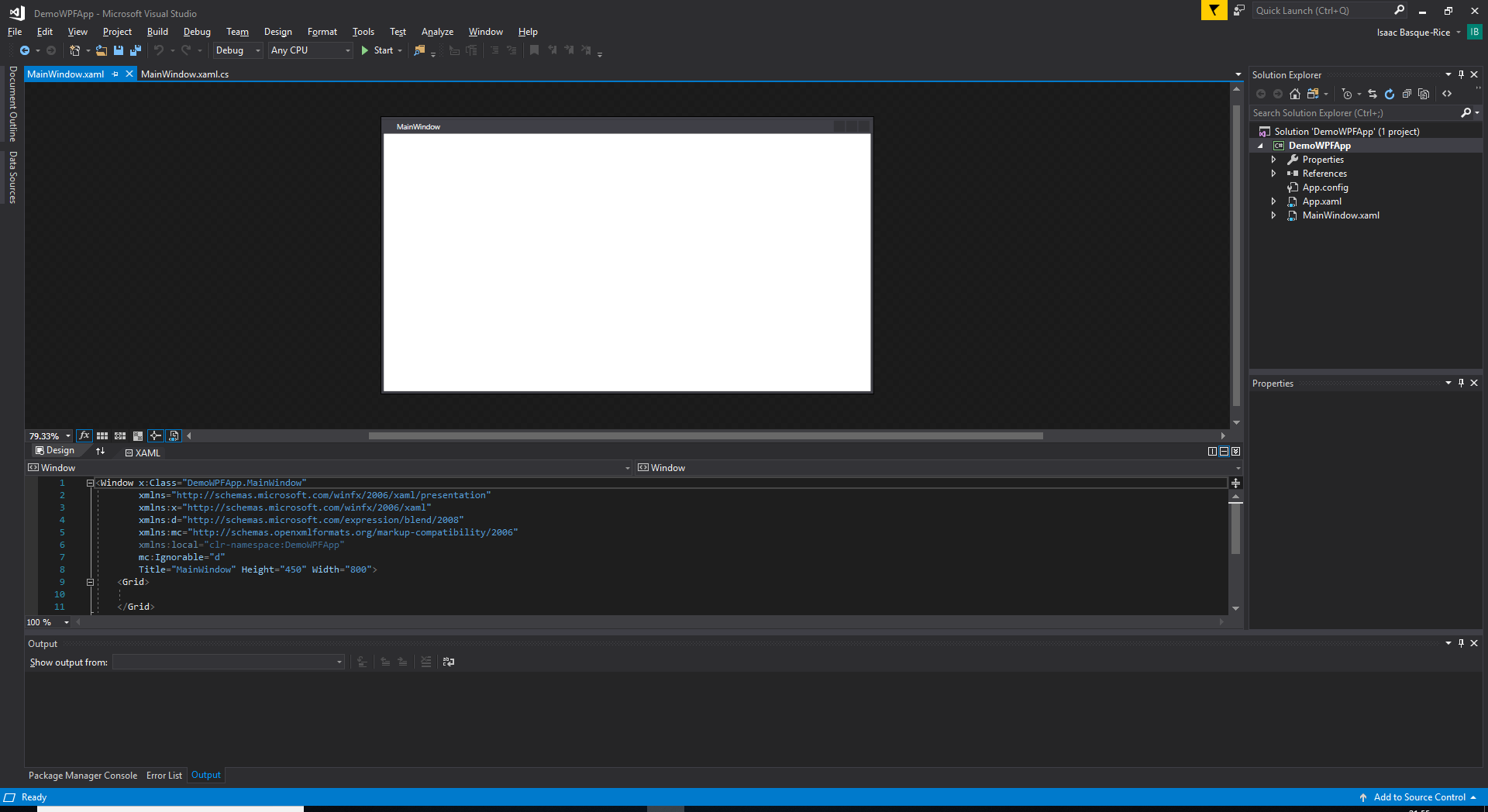


Figure 13, and just like that, the demo app solution file has been created and is ready to be edited

### Adding files to a project

Adding files to projects in Visual Studio is easy, all one needs to do is go to their solution in the solution explorer (the name of the solution with a white icon with C# in the middle and a green outline), right click it, and then go to Add > New Item (or press Ctrl+Shift+A), and then they will be granted with the below window, once the OK button is pressed, the file will be added to the solution

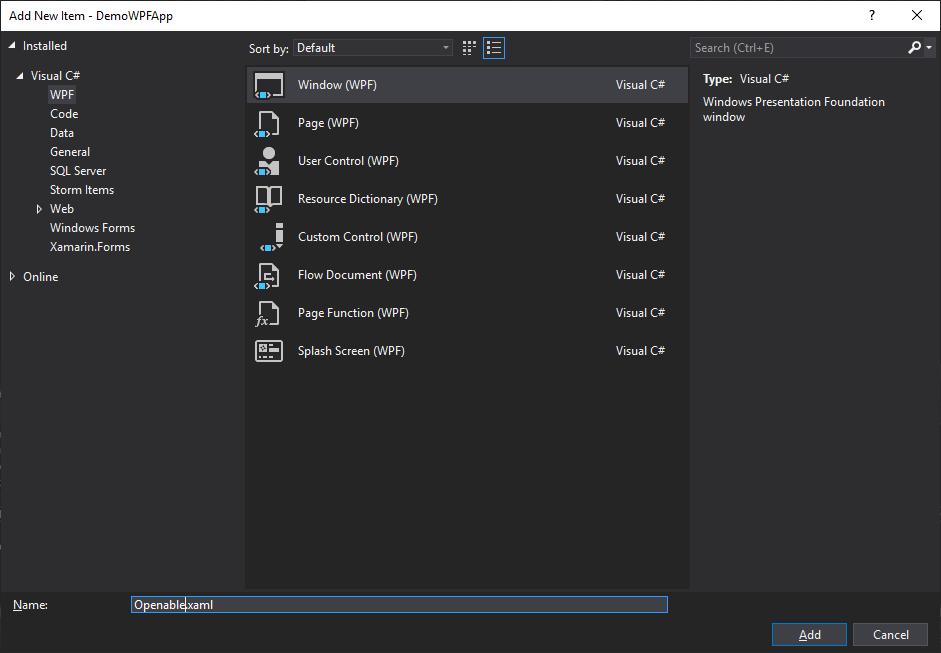


Figure 14, here I have added a new item, the Openable window, to the solution, this will be the window that is opened when I press the button

Below you will find the Solution Explorer after the addition of two files, the Openable.xaml window I added above, and puppy.jpg, which is a JPEG image of a puppy for display purposes only. This image was added to the solution through a simple drag-and-drop method from the Windows File Explorer.

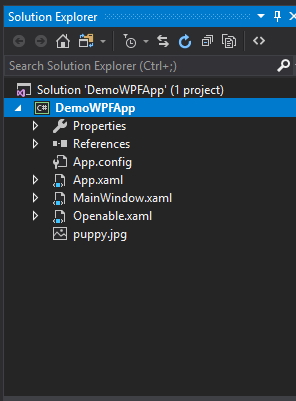


Figure 15, the Solution Explorer window after adding two files, the window and the jpg

### Resources

A resource is an object that can be reused in different places in an application. An example of a resource I have used in a solution, not necessarily in this unit, however, is in Evaporate, wherein I made use of the MaterialDesignInXAML pack, and, as such, made sure the buttons were using the “{StaticResource MaterialDesignRaisedButton}” style. In addition, the Puppy.png image is a resource, as is the other files in the solution.

### Closing a project

Closing a project can be done by pressing the red X button in the top right-hand corner of the solution, if something in the solution is not saved then a window will pop up asking whether you wish to save the solution, to which you can reply Yes, No, or Cancel to abort closing the solution altogether. One can also close the solution through navigating toFile Close Solution, this has the added benefit of not closing the Visual Studio program altogether when closing the solution. In addition to this, the opening of a new solution will almost always close the previous one to save memory

### Moving a project safely

Moving a project in Visual Studio is not always the clearest and easiest thing to do, as such, the following is an instruction as to how to accomplish this.

Here I have “AnotherDemo”, let’s say instead of removing it entirely I want to move it somewhere else, to another solution perhaps. To do this I would right click “AnotherDemo”, click “Remove”, this does NOT delete the folder or any files within, it just removes it from the solution. Then you go into the folder where the project is located, you move it to another solution or wherever you wish it to go, and then right click the solution in the solution you want the project in, go Add > existing Project, find the .csproj file for the project, add that, and then you’ve successfully moved a project in Visual Studio!

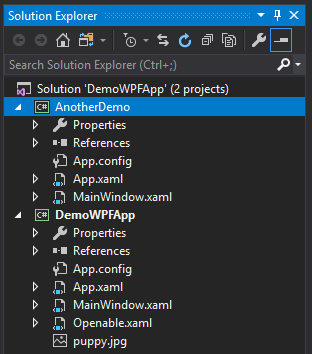


Figure 16, the Solution Explorer with two projects in, DemoWPFApp, and AnotherDemo

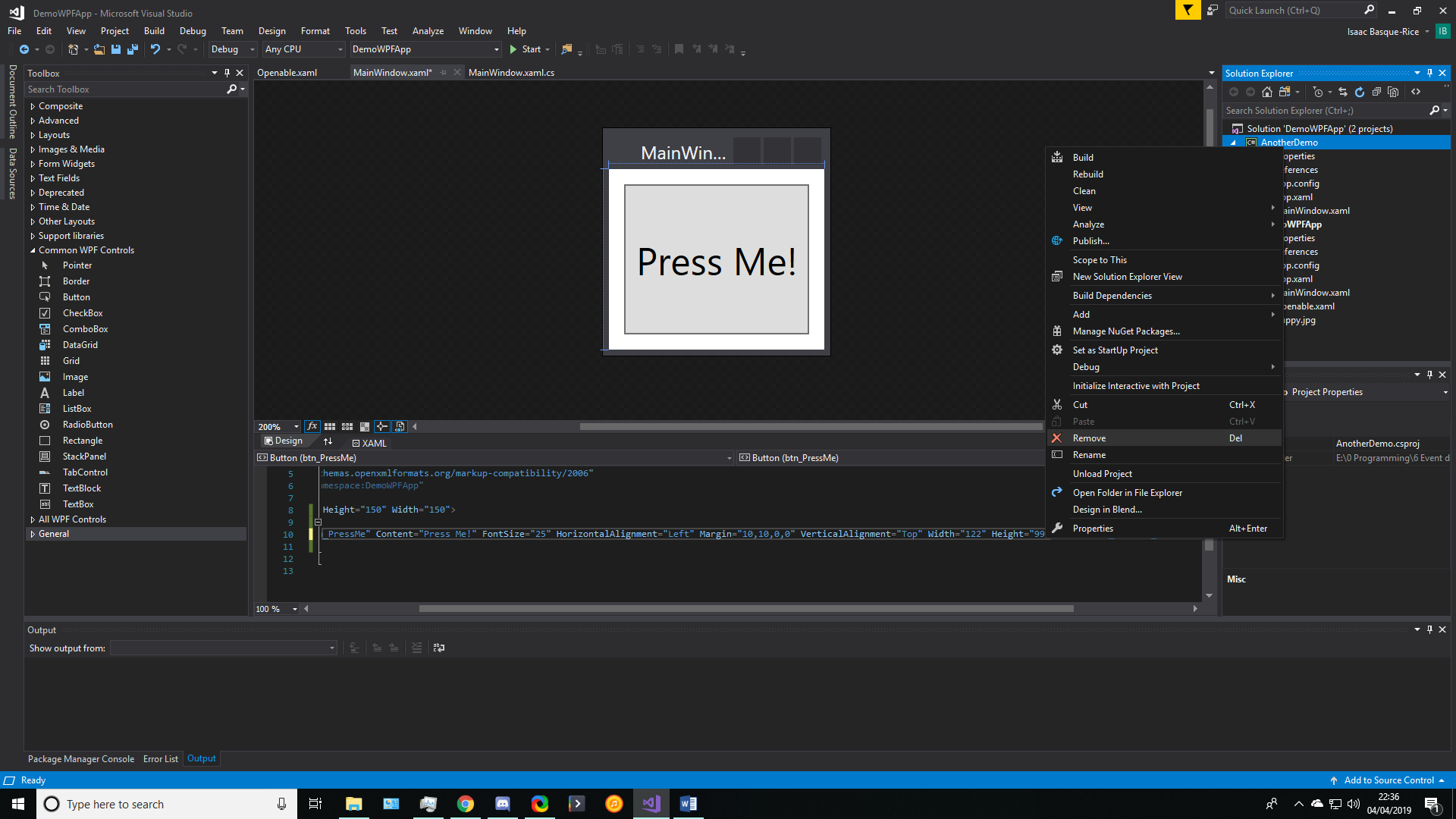


Figure 17, "Remove" AnotherDemo



Figure 18, AnotherDemo in the source folder

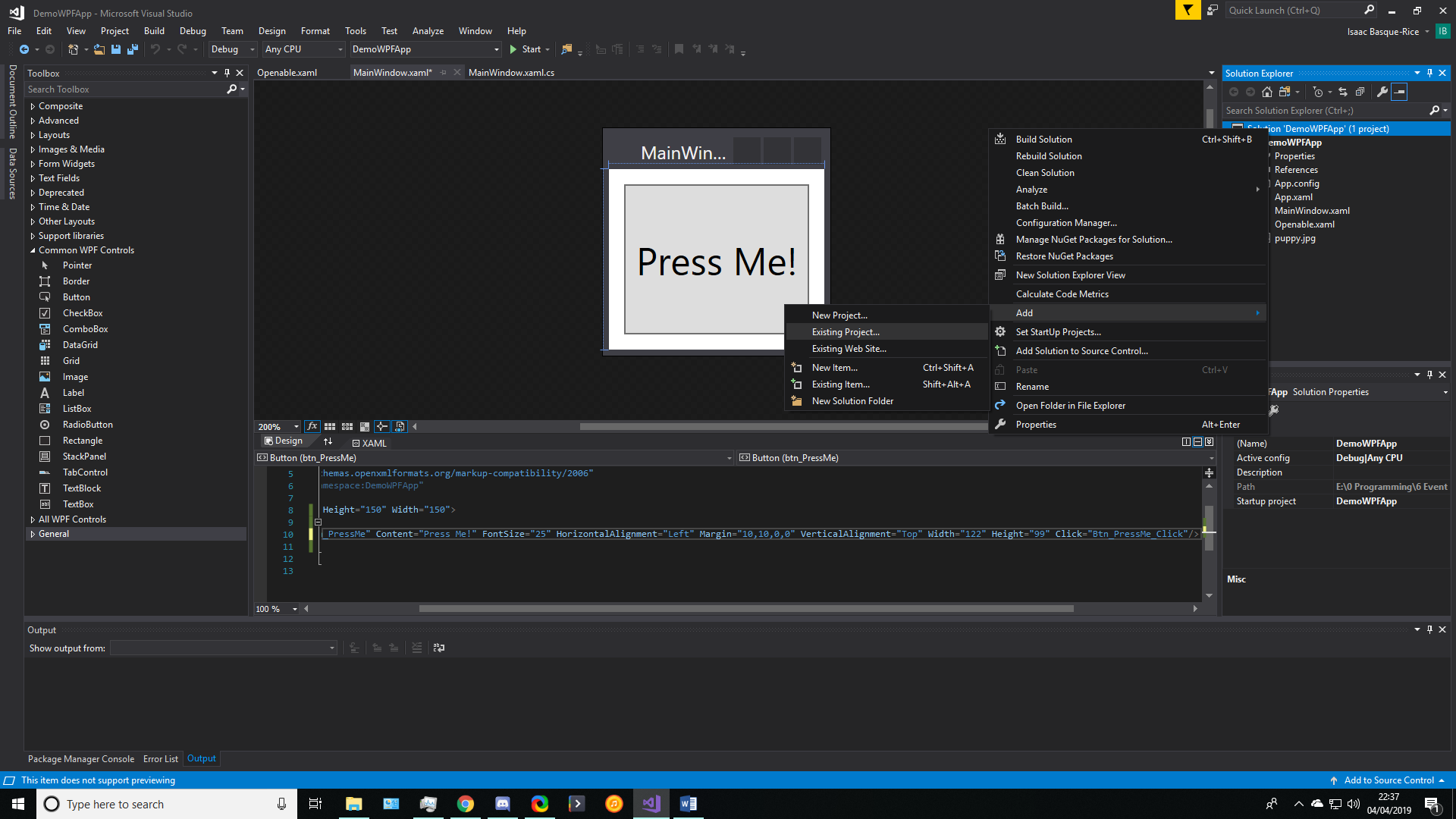


Figure 19, Add > Existing project

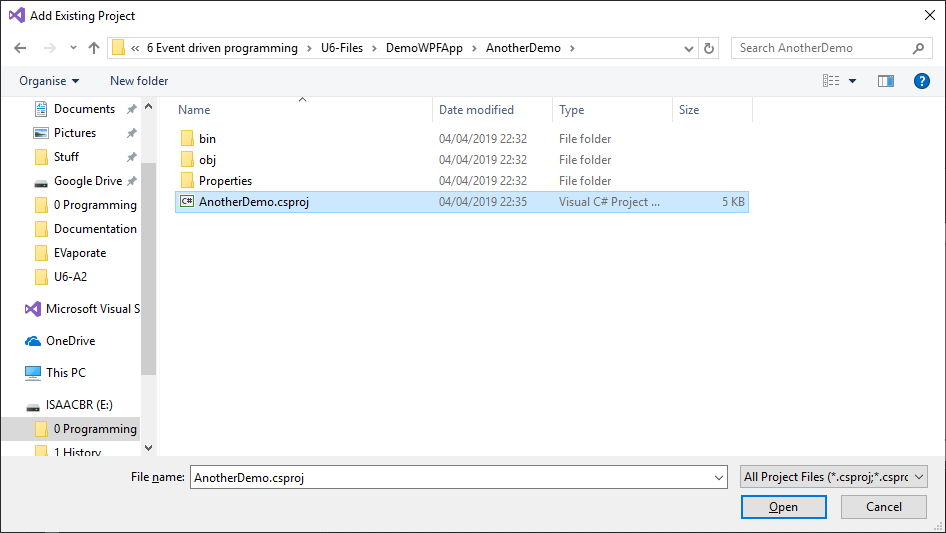


Figure 20, the .csproj file you need to open to re-add a project to a solution

### File operations

We have already gone through creating/adding files into a solution, as well as, through the nature of the thing, reading and writing to files, however there has been some oversight in this section, namely, renaming and opening files in Visual Studio. To open a file, we simply click it once to open it in a temporary window wherein once you click away the window is gone (denoted by the tab opening on the right side and it appearing purple) and a double click for a more permanent opening (on the left, blue). To rename a file we right click the file we wish to rename and click “rename” in the window that shows up, like so:

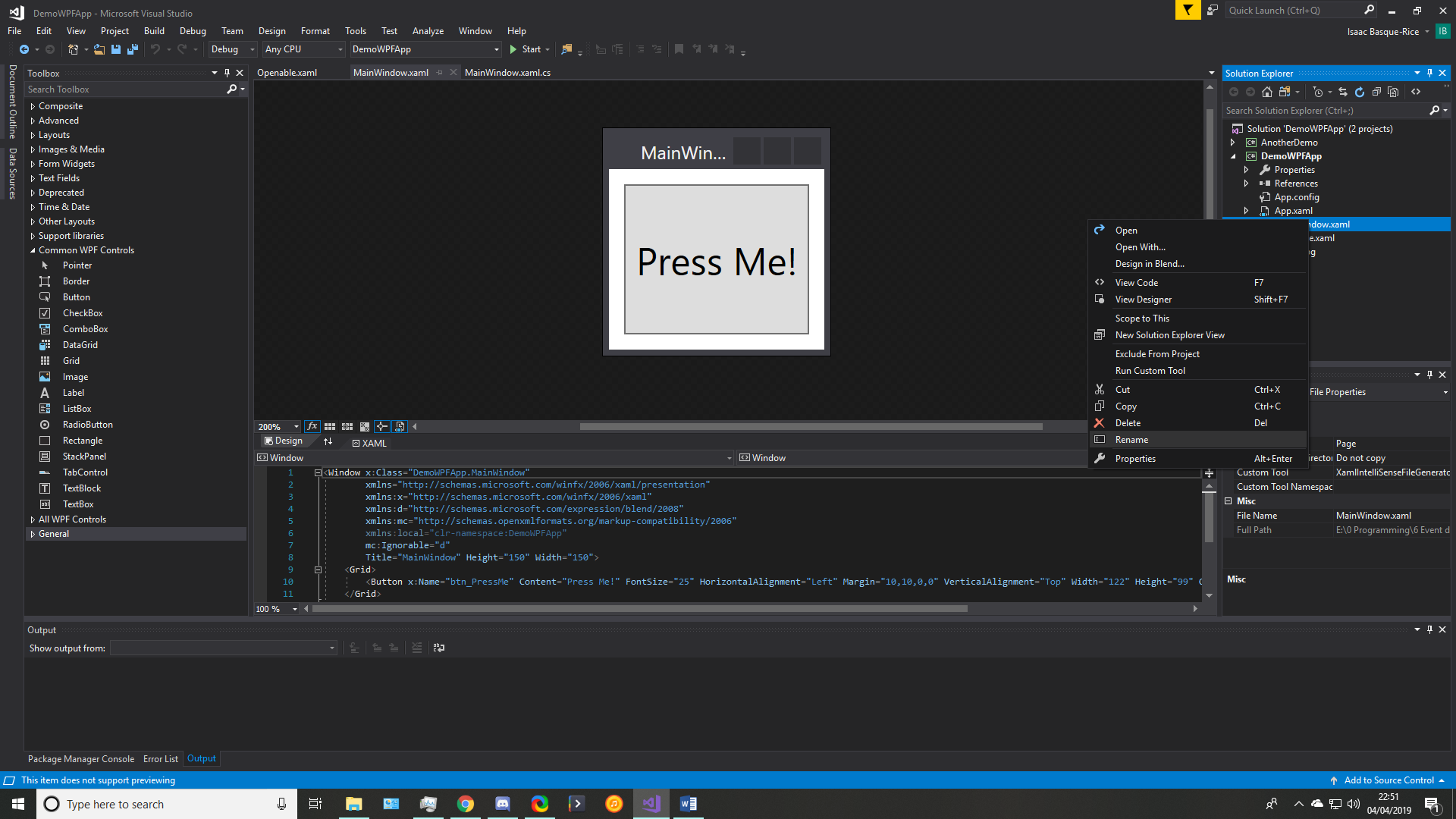


Figure 21, options for changing the file

Other things we can do from the above window include viewing the code and designer views, cutting, deleting, copying the file, and viewing its properties.

#### Load

To load a solution, you navigate to File > Open > Project/Solution (or press Ctrl+Shift+O), and then specify which solution you want to open using a pop-up version of the file explorer, as so:

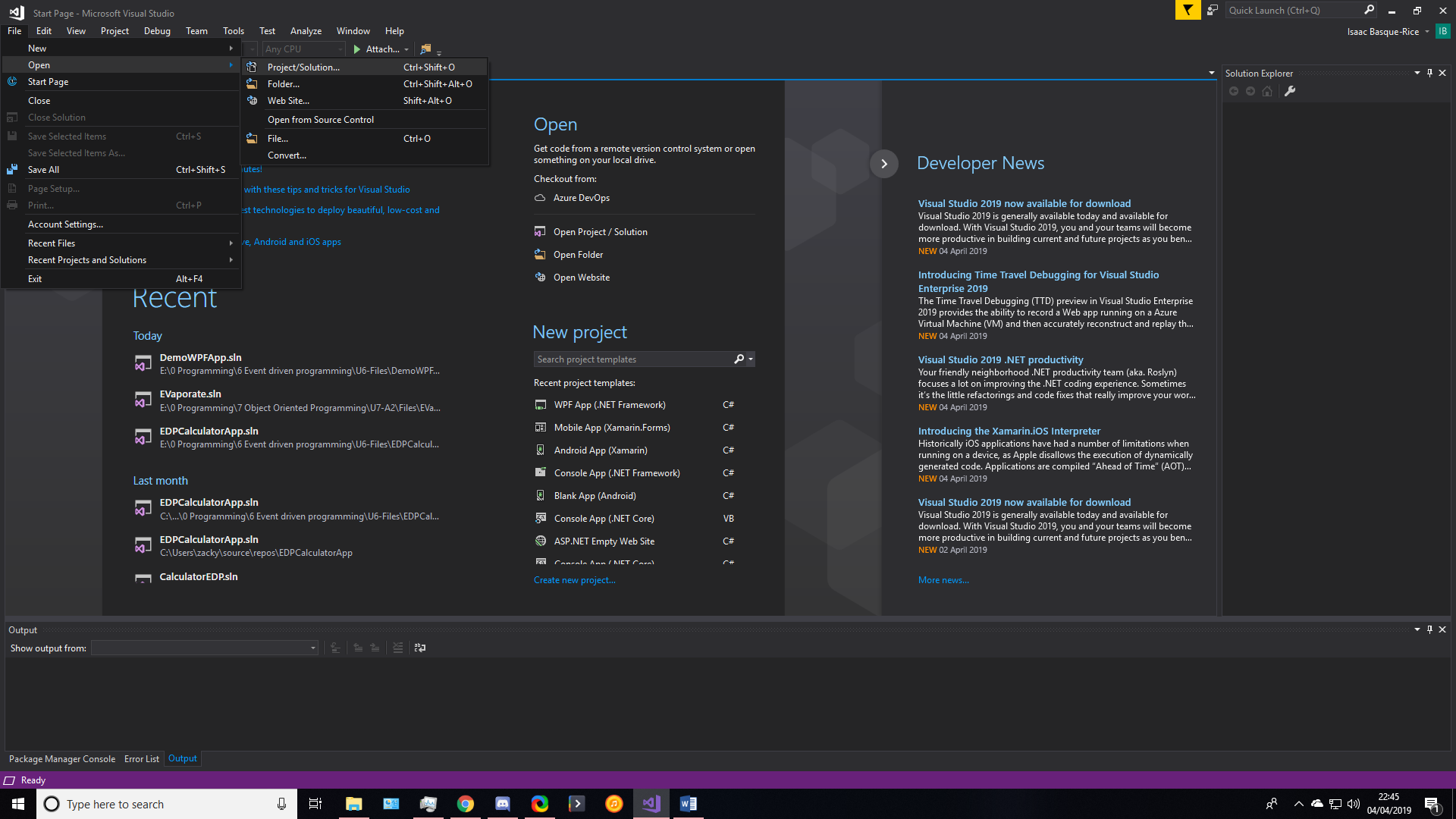


Figure 22, File > Open > Project/Solution

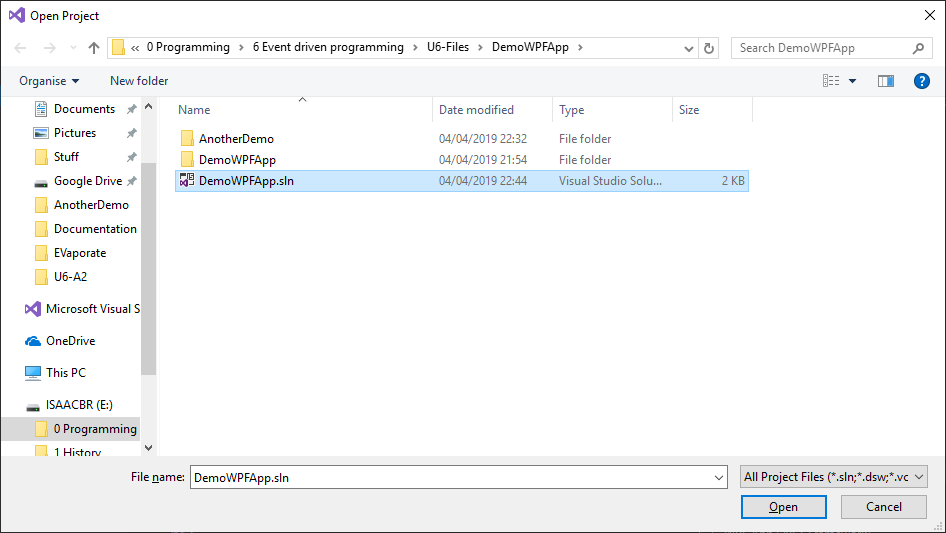


Figure 23, the Pop Up version of File Explorer searching specifically for .sln files (or other project files more generally)

#### Save

On closing a development task without saving the work you have done, you are greeted with the below dialog box, which prompts you to save the work you have done, so you may go back to it later and continue working on it.

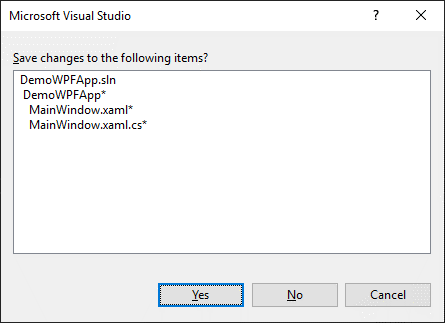


Figure 24, the "save" dialog box

In addition to this, there are number of other ways to save the project, Ctrl+S is the most helpful way in my opinion, but File > Save is a perfectly valid method, also. In visual studio, an unsaved window is usually denoted with the “\*” symbol next to the name of the window in the tab, as is below, this is in order for the developer to know what requires saving where.



Figure 25, an unsaved MainWindow window

### Compilation/Building

Finally, we come to the compilation and building aspect of the program, to just build a program (without running it) we visit Build > Build Solution on the top ribbon bar (or press Ctrl+Shift+B), this shows if there will be any errors and whatnot in the solution. To fully run the solution, we can press F5 on the keyboard or the green arrow at the top:



Figure 26, the Start button

This builds and runs the solution as it has been developed and allows a developer to debug the solution as they go (through breakpoints and whatnot)

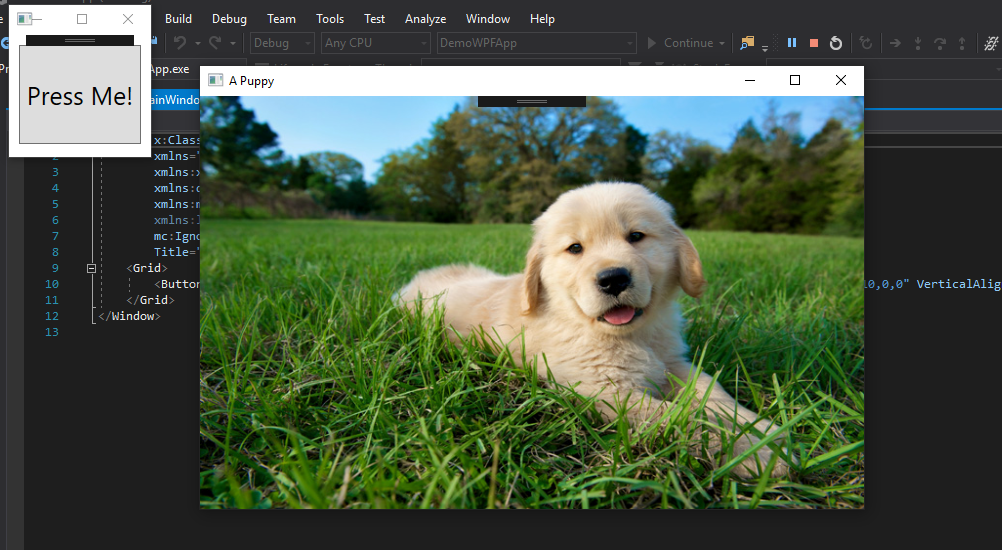


Figure 27, the program being debugged

### Customisation

As mentioned elsewhere, the layout of the IDE can be customised to the developers’ hearts content, the portability of windows within the context of the development environment means that the developer can customise their layout to what will work best and most efficiently for them.

Below we see an example of my personal customisation method, when developing a WPF application my preference is to have the toolbox taking up the majority of the left side of the screen for easy dragging and dropping onto the main WYSIWYG window, below that taking up almost the entiety of the lower fifth of the screen we have a tabbed window with the possibility of displaying the error list, the output, and during debugging, the locals and autos windows and much more. This is because a lot of information needs to be conveyed at once in these windows and they are all of high importance. Next, on the right of the screen, we have two windows, the explorer window, allowing me to open files, and the properties window below, allowing me to change the properties of elements in the application. Finally we have the centre, which is taken up roughly equally with the XAML code editor and the WYSIWYG editor for the application, it is helpful to have both open simultaneously due to the fact that some things may be more easily edited by one method than another, the one-on-top-of-the-other approach I have taken is personal preference.

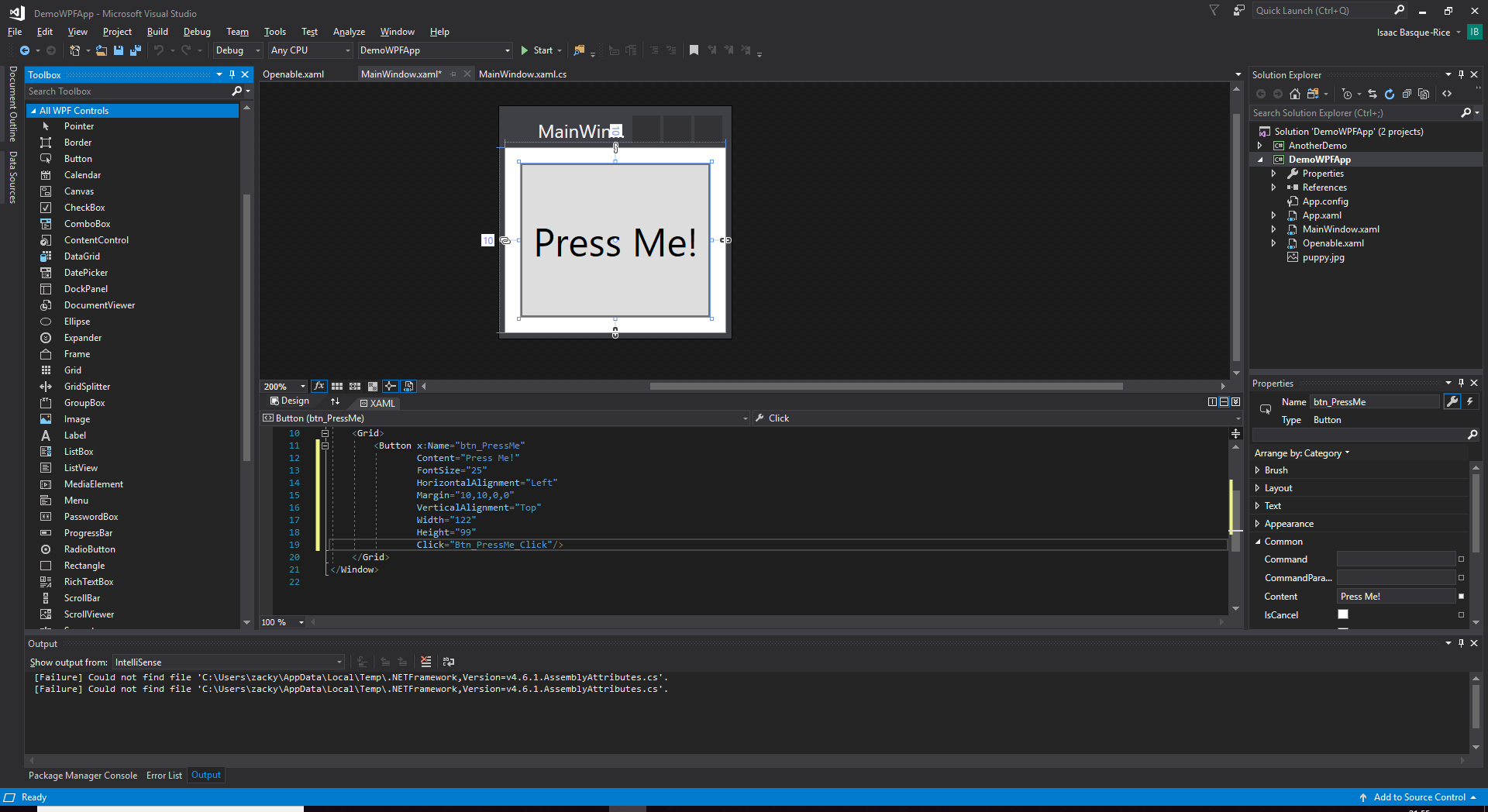


Figure , the customisation layout I have with my instance of the Visual Studio IDE

### Restoring defaults

Restoring Visual Studio to its default settings is a simple task, visit “tools > import and export settings”, and then follow the instructions in the Import and Export Settings Wizard, as seen below:

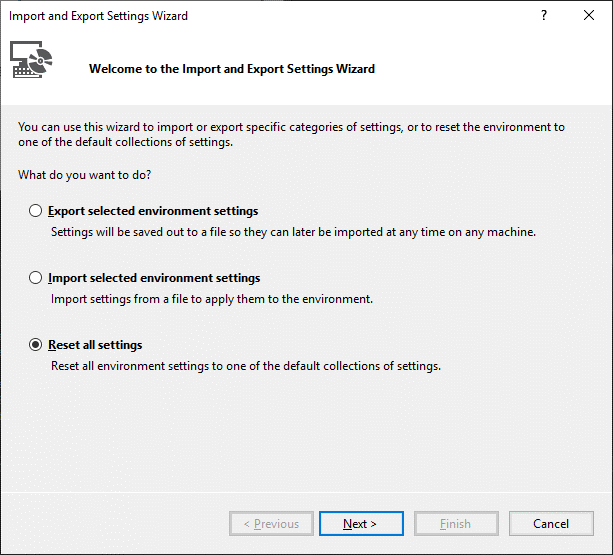
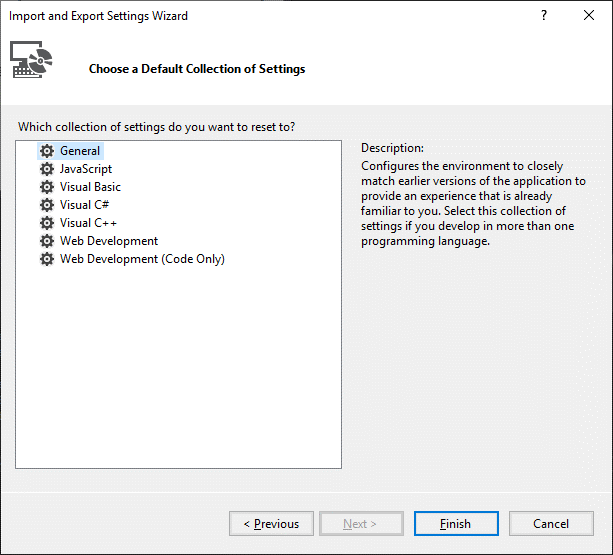


Figure , the first page, click the bottom option to reset defaults



Figure



### Project types

The choice between different types of project in Visual Studio is plentiful, one can choose between any number of application types such as console, windows desktop (WPF), Windows Universal (UWP), Cross-Platform (using Xamarin), and console applications, which all serve different purposes from desktop applications to mobile to extremely simple ones that run out of the console. Below you see the window where one can choose their application type, it is very important that you choose what project type you wish for carefully as failing to choose the correct one can result in major issues for the project.

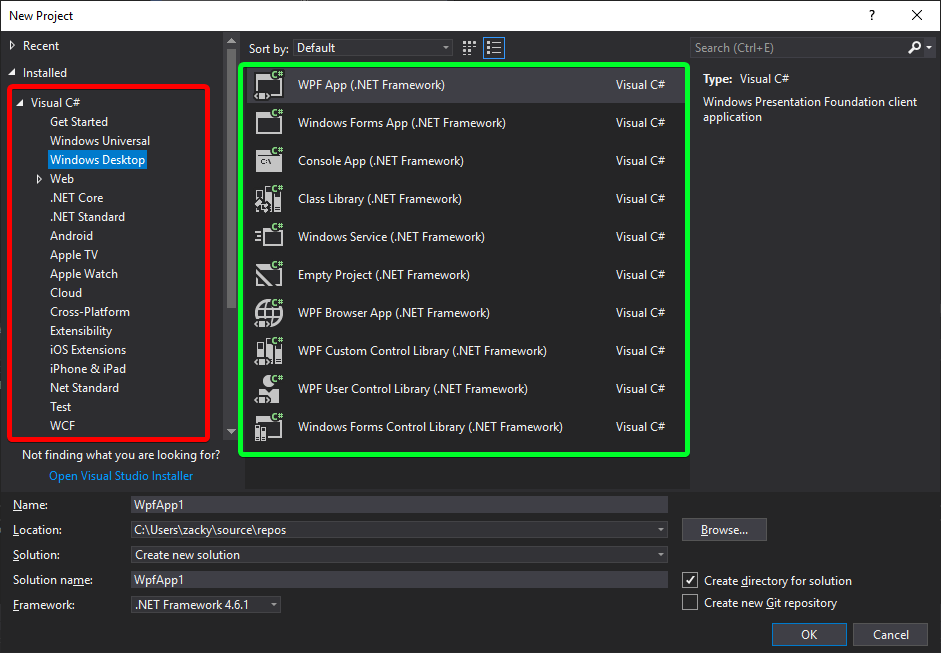


Figure , the project creation window, in red you see a selection of general overarching kinds of application and in green you see more specific types

### Coding conventions and standards

This criterion is fulfilled in the below section, entitled “[Using appropriate features and functions to carry out EDP using correct coding standards and conventions](#_Using_appropriate_features)”, and goes into some detail about how I upheld the correct standards and conventions in the wider application.

### Executables

Forming an executable from an application in WPF is an extremely simple task, and is laid out in the section of this document entitled “[Deploying an EDP application as a professional installation](#_Deploying_an_EDP)”

### Cleaning intermediary files

Cleaning files removes the build artefacts from files from the previous build, which are by-products of said build. In particular it cleans the bin and obj files, which is where any temporary files are stored. The Bin, or binary folder holds binary files which are the actual executable code for the application, and the obj, or object files, are compiled binary files that haven’t been linked but will be in order to create the final executable. These files can be prone to mistakes and corruption so cleaning these files can be crucial.

## Using appropriate features and functions to carry out EDP using correct coding standards and conventions

In this section I will be discussing how I implemented these tools in the development of my calculator application. As mentioned previously, the use of the designer on the front end was fairly liberal, with few exceptions, the majority of the front-end code was built using the designer and as such does adhere to correct XAML coding standards and conventions by virtue of it being automatically generated by the IDE itself.

I built the front-end of the application by first dividing it up into 4 columns and 6 rows, as shown below:

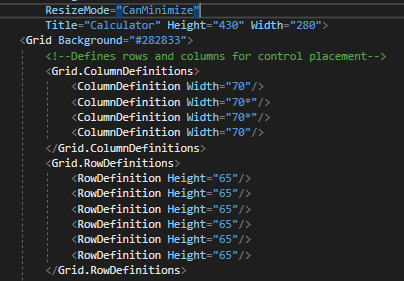


Figure 32, row and column definitions as well as the line that means the calculator cannot be resized and the definition of the size of the calculator

Due to the fact I used this method, however, I decided it would be best for me to individually edit each button to locate them within the grid structure, as such you will see that each button has a "Grid.Row” and “Grid.Column” attribute that is developer specified in XAML, this, however, is the only thing I had to do in the code editor with regards to the front end, everything else was specified in the Properties menu. Below is an example of the buttons constructed in this way, the number group. The other buttons, such as the operator and clear groups, are constructed in much the same way, specifying name, row/column, font size, back and foreground colour, border thickness set to 0 to make it appear cleaner, and then the onclick event handler connected to the back end, with the content of the button coming after between the opening and closing tag, this is best practice.



Figure 33, the Number Buttons

It must be noted, incidentally, that in order to generate each OnClick event handler in the solution, I made use of the facility in VS which allowed me to double-click on the relevant button itself in the designer and generate an empty event handler for me to program.

Below here is the properties menu of the 0 button, note the formatting specification within this menu, specifically with regards to front and background colours and text properties:

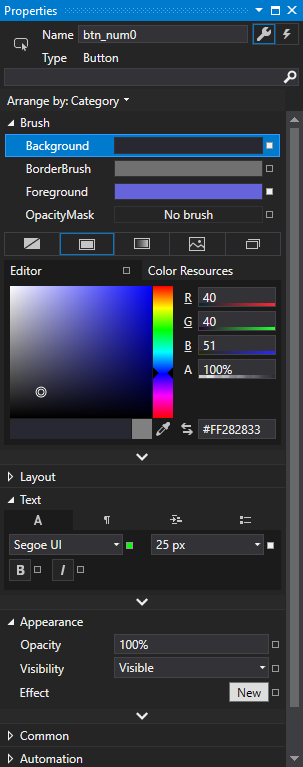


Figure 34, the expanded properties menu of the 0-number button, also note that the name of the button was specified at the top

With regards to correct coding standards and conventions as far as C# is concerned, the document I was primarily referencing here was Lance Hunt’s 22-page guideline document “C# Coding Standards for .NET”, published in March 2007 (found at this address: <http://se.inf.ethz.ch/old/teaching/ss2007/251-0290-00/project/CSharpCodingStandards.pdf>). Whilst it is, at the time of writing, twelve years old, it contains basic guidelines and naming conventions that still hold up to this day, additionally, after cross-checking it with the official Microsoft documentation and other sources I can confirm the validity of the document, I am using this as it is seemingly the most comprehensive.

Firstly, hunt mentions comment conventions, specifically how a C# developer must only make use of “//” to declare comments (as opposed to “/\* … \*/”) and they should never “flowerbox”, or surround the comment with unnecessary characters for aesthetic purposes, as the image below demonstrates, I have stuck to this convention.

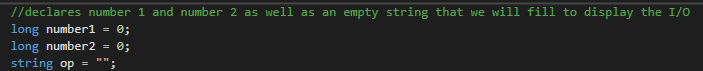


Figure 35, a comment using correct conventions for C#

The naming conventions in C#, that is, how to name variables, classes, project files, and so on, follow a strict guideline. Fundamentally, spaces are not allowed in any name in almost every programming language, to deal with this and to differentiate words from each other, programmers make use of “cases” such as PascalCasing and camelCasing (formatted like that), I have stuck to this closely, however this is mostly due to the fact that names were frequently generated by the IDE itself. An example of this is “private void Btn\_opEquals\_Click(object sender, RoutedEventArgs e)”, which (as denoted by the capital B in “Btn\_”) is PascalCased, despite the fact that the original button (x:Name="btn\_opEquals") is camelCased.

I did not use Hungarian notation (ex: iNumber1 or strOp to denote an int called number1 or a string called Op), instead opting to camel case the relevant variables, the abbreviations in names I used (specifically btn\_) is widely accepted and had been advised by my teacher for the first project I worked on, and all of my variables and other named things were descriptive and concise.

With regards to code formatting, as one can see from the code snippet below, curly brackets always occupied their own lines, tab length is always equivalent to 4 spaces, indentation is used where necessary, and, naturally, each statement occupies its own line.

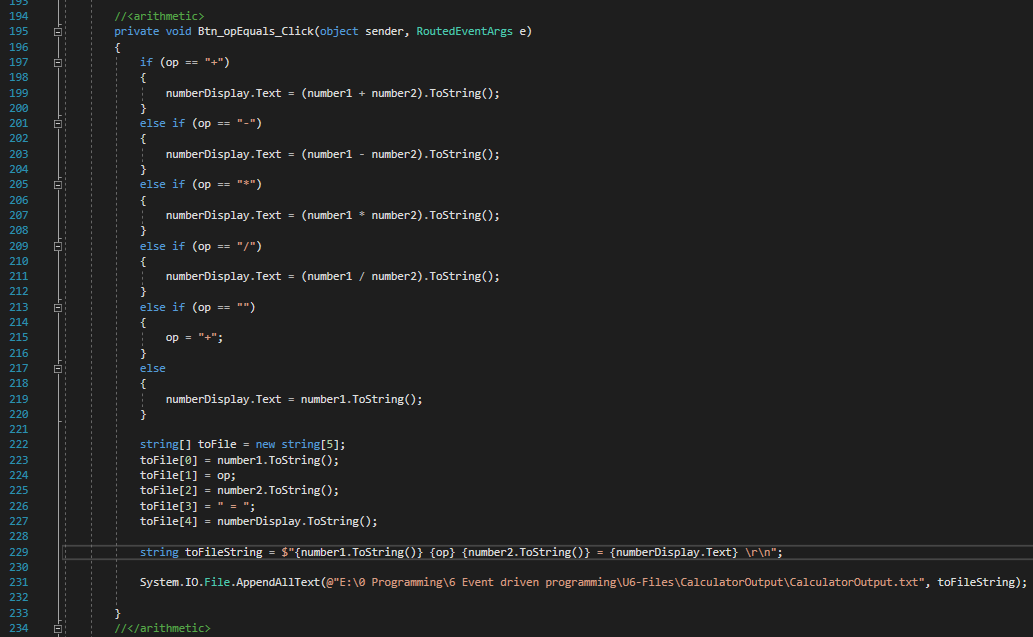


Figure 36, my example of correct formatting and best practice with regards to standards and conventions

Errors and issues in the process of compiling and/or building the solution are commonplace, the two main issues are errors in code (usually syntactic errors that prevents the compiler from understanding how to build the program), and exceptions, which are issues In code when syntax is valid, for example when a program points to a non-existent location or has an incorrect file path. In the case of errors, the program tends to not even start to build, however exceptions take a different approach, and launch the application until the exception point is hit (perhaps pressing a specific button and whatnot).

An example of a compilation/build problem I encountered was the following:

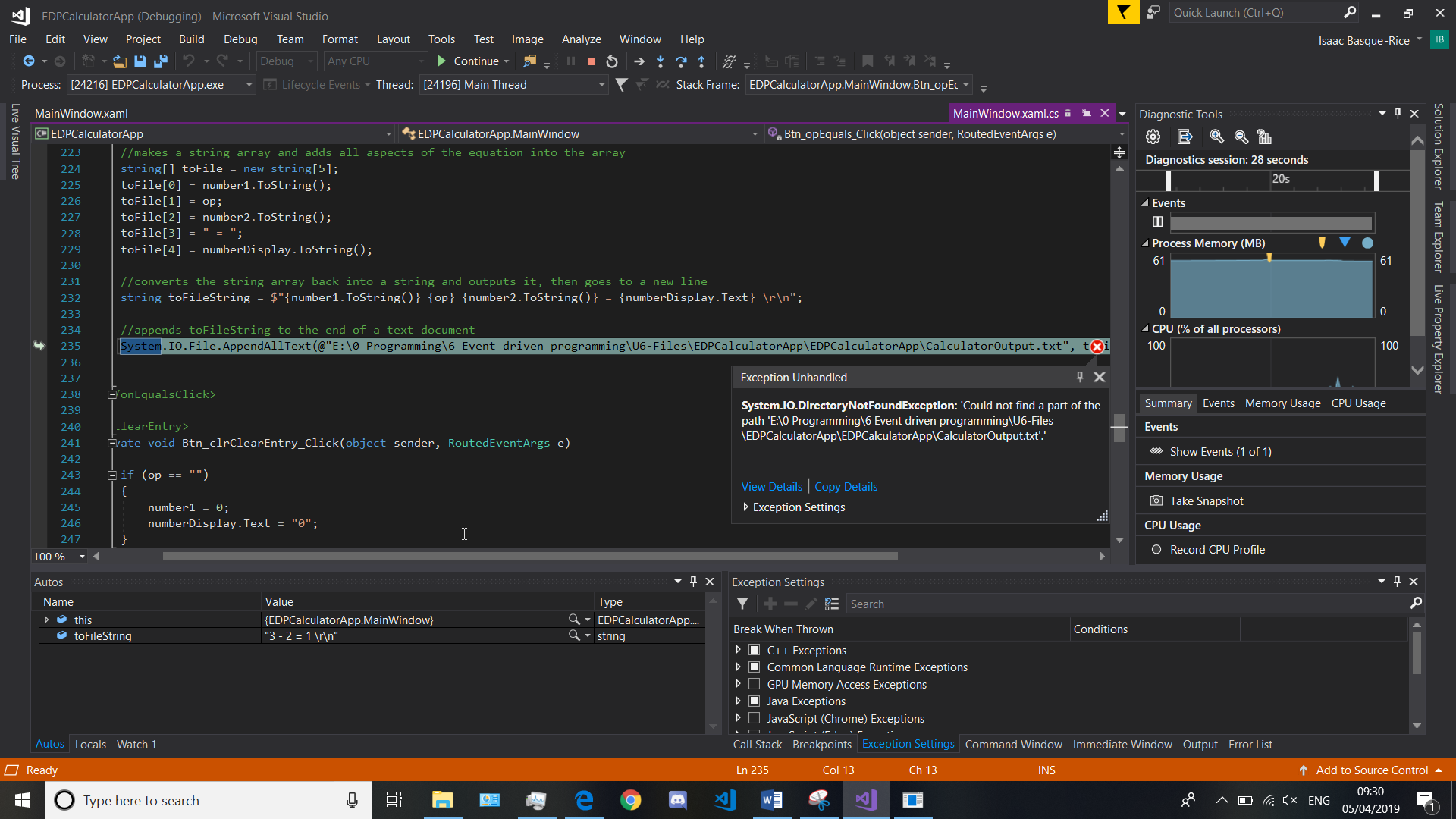


Figure 37, An exception thrown in the output path

“System.IO.DirectoryNotFoundException

Message=Could not find a part of the path 'E:\0 Programming\6 Event driven programming\U6-Files\EDPCalculatorApp\EDPCalculatorApp\CalculatorOutput.txt'.”

This exception is thrown when a directory, or file location, is not found in the context of the file system, this could be due to a mistake in the file path. After looking for errors in the file path through the explorer I discovered that due to the fact it is an absolute file path (it describes the file location fully, from the drive root level down) a change in drive name, as happens between devices, (in this case the file path described an E:\ drive when the correct drive was actually D:\), can seriously mess with the program and throw an exception. To remedy this, I have set the output file to a relative file path (replacing the full file path with just “CalculatorOutput.txt”) so it can change depending on the context.

Warnings in the context of an IDE are notices that notify the developer of something that is syntactically valid, and will be compiled into the application, but otherwise affects either the readability, functionality, or speed at which the program is run, and as such is recommended that you change.

An example of a warning in my solution was the instancing of a variable within a method that was never used, this threw a warning due to the fact that a position in memory was reserved for a variable but never used, thus slowing the program unnecessarily, below you will see the dialog for the error:

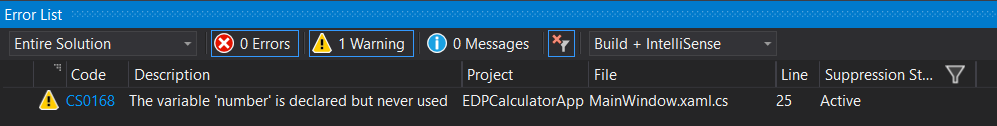


Figure 38, the error window displaying warning CS0168

Errors are similar in presentation to warnings however primarily concern syntactic inaccuracies and incorrectness generally. Programs will not compile if errors are present and as such they must be dealt with as a matter of urgency. Below is an example of an error I made in the program to demonstrate the error feature.

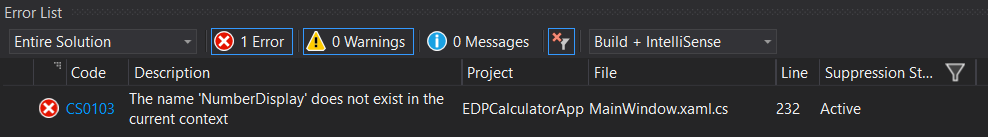


Figure 39, An example of error CS0103 wherein the name "NumberDisplay" is non-existent in the context of the program, this is because this name is formatted differently (numberDisplay, camelCased) it throws an error

During debugging, there are a range of tools that I, as a developer, can use in order to have a full understanding of the application. The first example of a debugging tool is the most well-known, the breakpoint tool, below you will find a screenshot of the locals window after a breakpoint has been hit, this displays the name of the variables, the value, and the type, which is the kind of variable it is (int, string, etc.)

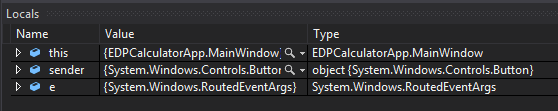


Figure 40, the local’s window early in the execution of the program (after first input, a button press)

Stepping in the context of the program is the process of running a single command at a time. Specifically, stepping over the program allows for the further execution of a program whilst also in a breakpoint context, pressing F10 allows stepping over to occur and allows or the developer to view the changing values in the locals window over time as it is navigating from one method to another.

Stepping into a method allows the developer to understand the specifics of a method and execute a program one line at a time, for example, if one were to place a breakpoint in my solution on an on click event handler for the number 5 button, stepping into it would produce the following two locals windows, the first being a Boolean checking whether the value of the operator is true (filled) or false (empty), if it is false, which it is, it moves on to edit number1, after stepping through this it eventually fills the value of the variable with the number 5 and then steps out of that, ready for the addition of an operator.

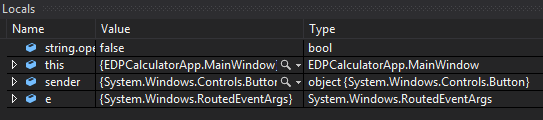


Figure 41, the locals window displaying a false Boolean value

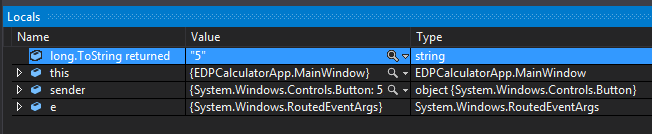


Figure 42, the locals window displaying the value "5" as a string, this is the value that is displayed on the output window.

Next, we have stepping over, which is fairly self-explanatory, stepping over a method skips the breakpoint debugging to the next method, allowing for navigation through the program to take place at a much faster pace than it otherwise would.

Finally we have stepping out, this is a process of reversing the step in procedure within a method, essentially, it moves you outside of the context of the method you have stepped into in order for you to move on with debugging the program

During the debugging process, specifically during the operation of a breakpoint, it is possible to open a so-called “watch window”, and add variables to it. This window, as can be assumed, allows the developer to watch the values and types of a number of different variables during normal running of the program. The use of this is to see, in a simpler fashion, what is being taken in to the application and how it functions during the running process.

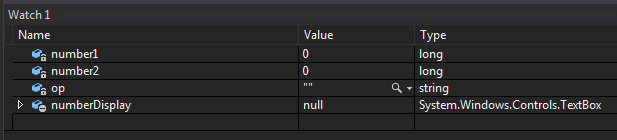


Figure 43, the watch window with all empty values at the start of the program

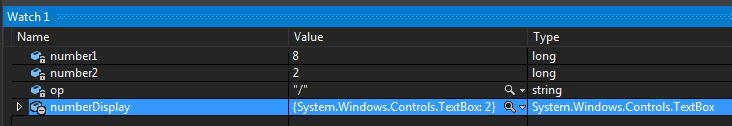


Figure 44, the watch window after a sum (8/2) has been performed, note the values of each variable have changed

Finally we have the output and immediate windows, which are also useful in the debug process. Firstly, the Output window shows the developer multiple things through the use of a drop down menu, it can show the build logs (which document the process of building the application from pressing start to having it open), to the Debug window, which goes into more detail about the debugging process in particular and shows logs relating to the loading of a multitude of different packages in order to ensure the correct building and loading of the application. Below are two examples of the outputs available through the outputs window, the first being from build, and the second from debug

* Build

1>------ Build started: Project: EDPCalculatorApp, Configuration: Debug Any CPU ------

1>C:\Program Files (x86)\MSBuild\14.0\bin\Microsoft.Common.CurrentVersion.targets(2884,5): warning MSB3327: Unable to find code signing certificate in the current user’s Windows certificate store. To correct this, either disable signing of the ClickOnce manifest or install the certificate into the certificate store.

1>C:\Program Files (x86)\MSBuild\14.0\bin\Microsoft.Common.CurrentVersion.targets(2884,5): error MSB3326: Cannot import the following key file: . The key file may be password protected. To correct this, try to import the certificate again or import the certificate manually into the current user’s personal certificate store.

1>C:\Program Files (x86)\MSBuild\14.0\bin\Microsoft.Common.CurrentVersion.targets(2884,5): error MSB3321: Importing key file "EDPCalculatorApp\_TemporaryKey.pfx" was canceled.

========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========

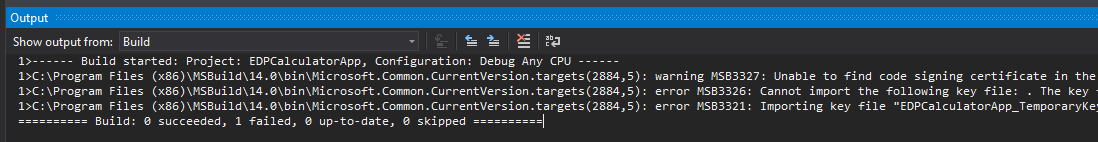


Figure 45, Output > Build

* Debug (note, this is less than half of the true debug window, this is due to the fact that it contains an inordinate amount of information that is not relevant in that quantity

'EDPCalculatorApp.vshost.exe' (CLR v4.0.30319: EDPCalculatorApp.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC\_32\mscorlib\v4.0\_4.0.0.0\_\_b77a5c561934e089\mscorlib.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.

'EDPCalculatorApp.vshost.exe' (CLR v4.0.30319: EDPCalculatorApp.vshost.exe): Loaded 'C:\Windows\assembly\GAC\_MSIL\Microsoft.VisualStudio.HostingProcess.Utilities\14.0.0.0\_\_b03f5f7f11d50a3a\Microsoft.VisualStudio.HostingProcess.Utilities.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.

'EDPCalculatorApp.vshost.exe' (CLR v4.0.30319: EDPCalculatorApp.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Windows.Forms\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.Windows.Forms.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.

'EDPCalculatorApp.vshost.exe' (CLR v4.0.30319: EDPCalculatorApp.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.

'EDPCalculatorApp.vshost.exe' (CLR v4.0.30319: EDPCalculatorApp.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Drawing\v4.0\_4.0.0.0\_\_b03f5f7f11d50a3a\System.Drawing.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.

'EDPCalculatorApp.vshost.exe' (CLR v4.0.30319: EDPCalculatorApp.vshost.exe): Loaded 'C:\Windows\assembly\GAC\_MSIL\Microsoft.VisualStudio.HostingProcess.Utilities.Sync\14.0.0.0\_\_b03f5f7f11d50a3a\Microsoft.VisualStudio.HostingProcess.Utilities.Sync.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.

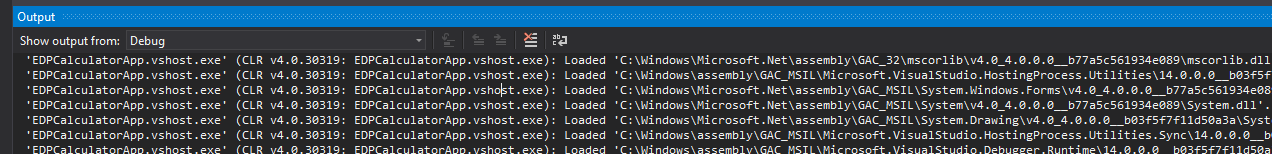


Figure 46, Output > Debug

Next, and finally, we have the Immediate window, this window allows a developer to execute more code during a build by entering it a line at a time, this is done for a multitude of reasons, including testing problematic code, changing the value of a variable mid execution, changing properties during execution, and to view the debugging output.

## Using appropriate debugging tools to identify and resolve programming faults

Several issues arose during development, an example of which was the following:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Severity | Code | Description | Project | File | Line | Suppression State |
| Error | CS0029 | Cannot implicitly convert type 'long' to 'string' | EDPCalculatorApp | E:\0 Programming\6 Event driven programming\U6-Files\EDPCalculatorApp\EDPCalculatorApp\MainWindow.xaml.cs | 32 | Active |

The facility in Visual Studio to highlight errors is extremely useful in most, if not all circumstances, the additional ability to instantly search for error codes within the IDE and for it to launch within the developer’s default browser serves as a huge boon to me, the developer. I discovered an extremely easy fix to this, appending the ToString() method to the end of the offending line using dot notation fixed all of my issues up nicely.

However, some issues are not so easily fixed or caught with the error list, sometimes perfectly valid code can result in some unwanted effects, an example of this is the current issue, where when I type in 25 / 5 the output is given as 125. To combat this issue, I placed a breakpoint on the line which should deal with division arithmetic, the line contained in the “else if (op == "/")” statement, line 212, I then repeated the same sum. In the Autos window after stepping in to the code during the break period I discovered that it was taking the two numbers and operating on them as if they were multiplying instead of dividing, I then looked at my code and noticed that I had accidentally swapped around the \* and / operators in the code, this was rectified immediately and the program functioned as intended.

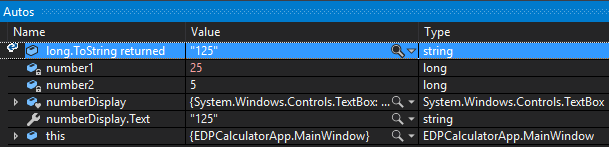


Figure 47, the Autos window, this is what led me to realise what was happening in my program

## Documenting an event driven programming solution

### Technical/Internal Documentation

In this section I will be describing how I developed the application from a technical standpoint so, if necessary, another developer can pick up development of this task where I have left off.

This app, as one can imagine, is fairly simple. The calculator itself occupies two files, MainWindow.xaml and MainWindow.xaml.cs, the former dealing with the front end, appearance and whatnot, and the latter dealing with the logic, as can be inferred from the .cs extension.

The addition of further controls to the front end is almost certainly unnecessary, however if it does become necessary, the following is the exact format by which controls must be written:

<Control x:Name="ctl\_catName" Grid.Row="x" Grid.Column="y" FontSize="25" Background="#282833" Foreground="#6564DB" BorderThickness="0" Click="Ctl\_catName\_Click">Content</Control>

Figure 48, an example of a control

An explanation, the formatting exists within the first tag, beginning with “Control”, this will let the compiler know whether it is a button, text field, text box, radio button, and so on. Next, we have the name, the formatting I have gone with is “ctl\_catName with ctl\_ specifying the control (such as btn\_ for button or tb\_ for text box), cat specifying what category it’s in (such as num for numbers or op for operators), and Name specifying the individual control, i.e. what exactly it does.

After this everything is essentially par for the course, Grid.Row and .Column specify where the button is located in relation to the pre-defined grid I laid out at the beginning of the project, formatting must always stay the same to stay in line with my design philosophy for this app (font size 25, text being a sort of light purple, background being dark grey, and so on). Finally, the click event handler is auto-generated by the IDE itself when you double click on the control so this is of little to no concern.

Next, and arguably more importantly, we come to the logic of the application, and here is where I will be explaining how I built it in a fair amount of detail.

The core of the app revolves around three variables, that is two numbers and a string. The numbers are not declared using the standard “int” data type, but rather the “long” one, this is because int can only display 32-bit numbers, that is any numbers between 231-1 and -231+1, or all signed 32-bit integers. This number (±2,147,483,647) is okay for most day-to-day calculations, however the “long” data type allows for all signed 64-bit integers (±9,223,372,036,854,775,807), which is orders of magnitude bigger with little to no difference in difficulty of development or memory usage. As such, I decided to play it safe and implement the “long” data type for both required variables

All three of these variables are declared as empty upon the initialisation of the app, like so:

long number1 = 0;

long number2 = 0;

string op = "";

Figure 49, the three core variables to my program as they appear upon initialisation of the program

Beyond this, it is simply a case of allowing the user to fill these variables with the relevant data, to achieve this as far as the numbers are concerned, I wrote the following few lines of code for each number button:

private void Btn\_numX\_Click(object sender, RoutedEventArgs e)

{

if (op == "")

{

number1 = (number1 \* 10) + X;

numberDisplay.Text = number1.ToString();

}

else

{

number2 = (number2 \* 10) + X;

numberDisplay.Text = number2.ToString();

}

}

Figure 50, a template for an event handler for number buttons, the Xs in this instance represent whatever number it is

This is an event handler for each of the number buttons in the solution. It first checks to see if the op variable is empty, if it is, this means we are in the first part of the equation and as such are working with number1, if it’s not empty, we are working with number2. The handler then takes the number that is already being held in the variable, multiplies it by 10, and then adds the value of the number button to it (denoted by X above). Let’s say we have the number 3 in number1, because we haven’t hit an operator button yet, but we have hit, for example, the number 5 on the keypad, we take 3, multiply it by 10 to get 30, and then add 5, to get 35, or the desired result. We then set the value of number1 to now be 35, then convert number1 to a string and display that string in the number display.

Now we come to the operators, or “ops”, as I have referred to them in code. The code below is four separate event handlers, one for each operator, that sets the value of “op” to be the value of the operator in question, and then sets the display value to 0, mirroring the value of number2 by this point in the equation. Note, additionally, the comments at the start and end of this block, this is to denote the beginning and end of a specific block of code that performs a specific action, in this case, operator logic. This comment format is present in each of the code blocks in the logic, from numbers to operators to clearing buttons.

//<operatorLogic>

private void Btn\_opAdd\_Click(object sender, RoutedEventArgs e)

{

op = "+";

numberDisplay.Text = "0";

}

private void Btn\_opSubtract\_Click(object sender, RoutedEventArgs e)

{

op = "-";

numberDisplay.Text = "0";

}

private void Btn\_opMultiply\_Click(object sender, RoutedEventArgs e)

{

op = "\*";

numberDisplay.Text = "0";

}

private void Btn\_opDivide\_Click(object sender, RoutedEventArgs e)

{

op = "/";

numberDisplay.Text = "0";

}

//</operatorLogic>

Figure 51, code for operators

Next we come to the onEqualsClick section, which is the logic for what happens when, as might be obvious, the equals button is clicked. This is where the actual operations occur, the adding, subtracting, and so forth, additionally, as you will see, I was also tasked with the addition of data structures to my solution, so as a result I convert each element in the operation to a string array and output it to a text document located within my USB, this is as a method of remembering what previous operations have been done.

//<onEqualsClick>

private void Btn\_opEquals\_Click(object sender, RoutedEventArgs e)

{

//the part that actually performs the operation

if (op == "+")

{

numberDisplay.Text = (number1 + number2).ToString();

}

else if (op == "-")

{

numberDisplay.Text = (number1 - number2).ToString();

}

else if (op == "\*")

{

numberDisplay.Text = (number1 \* number2).ToString();

}

else if (op == "/")

{

numberDisplay.Text = (number1 / number2).ToString();

}

else if (op == "")

{

op = "+";

}

else

{

numberDisplay.Text = number1.ToString();

}

//makes a string array and adds all aspects of the equation into the array

string[] toFile = new string[5];

toFile[0] = number1.ToString();

toFile[1] = op;

toFile[2] = number2.ToString();

toFile[3] = " = ";

toFile[4] = numberDisplay.ToString();

//converts the string array back into a string and outputs it, then goes to a new line

string toFileString = $"{number1.ToString()} {op} {number2.ToString()} = {numberDisplay.Text} \r\n";

//appends toFileString to the end of a text document

System.IO.File.AppendAllText(@"E:\0 Programming\6 Event driven programming\U6-Files\CalculatorOutput\CalculatorOutput.txt", toFileString);

}

//</onEqualsClick>

Figure 52, the OnEqualsClick section, which describes what happens on clicking the equals button

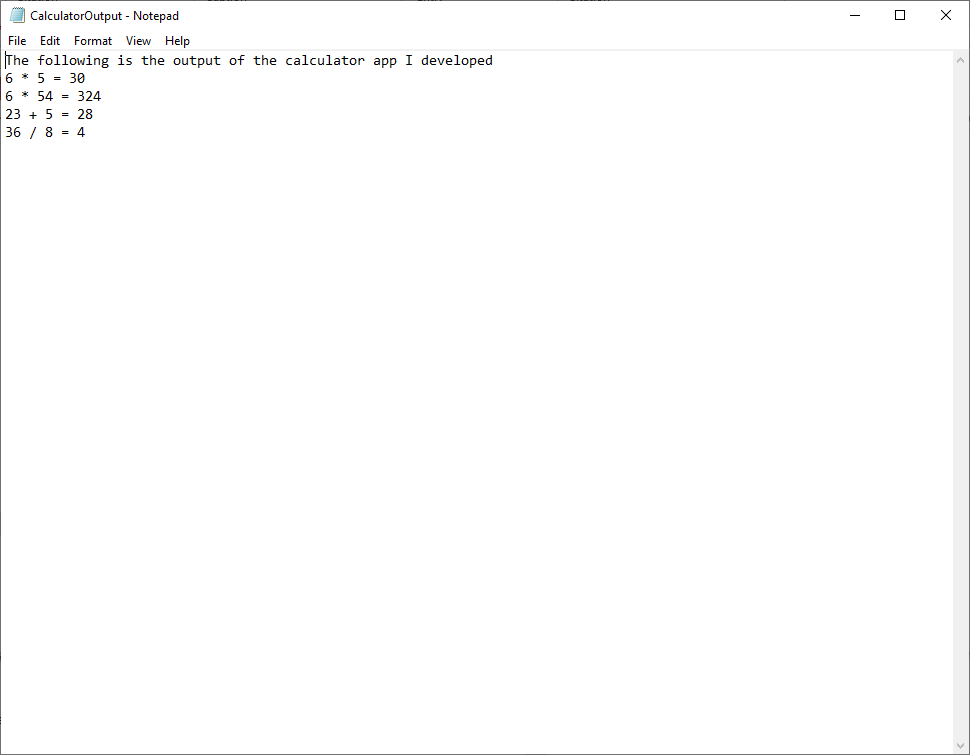


Figure 53, a screenshot of a portion of the output text document after a few operations were performed

Next, we have the negative button, this one is fairly simple, it takes whatever number is relevant (1 or 2) using the same method of checking the value of op as used in the number buttons, and then multiplies that number by -1, turning 56 into -56, for example. Code is below:

//<negativeNumber>

private void Btn\_symPositiveNegative\_Click(object sender, RoutedEventArgs e)

{

if (op == "")

{

number1 \*= -1;

numberDisplay.Text = number1.ToString();

}

else

{

number2 \*= -1;

numberDisplay.Text = number2.ToString();

}

}

//</negativeNumber>

Figure 54, the code for the negative button

Finally, we come to the three clear buttons, ClearAll, ClearEntry, and Backspace, we will be tackling these three events in that order. Firstly, ClearAll, the simplest of the three, merely sets the value of number1 and number2 to 0 and the value of op to nothing, this reverts the calculator to the same position it would have been at launch, the code for this is below:

//<clearAll>

private void Btn\_clrClearAll\_Click(object sender, RoutedEventArgs e)

{

number1 = 0;

number2 = 0;

op = "";

numberDisplay.Text = "0";

}

//</clearAll>

Figure 55, Clear All

Next, we have Clear Entry, which uses the same op value check as mentioned previously and clears the value of number1 or number2 depending on this, note the fact that the value of op is not cleared, this is something that could be improved. Code below:

//<clearEntry>

private void Btn\_clrClearEntry\_Click(object sender, RoutedEventArgs e)

{

if (op == "")

{

number1 = 0;

numberDisplay.Text = "0";

}

else

{

number2 = 0;

numberDisplay.Text = "0";

}

}

//</clearEntry>

Figure 56, clear specific numbers, does not clear op

Finally, we come to backspace. This feature is not perfect, and does not work in instances where decimals can exist. The method by which backspace is achieved is an op value check, as with previous entries, but it then divides one of the variables by 10. In a system where decimals are allowed, this would not achieve the desired effect, and so if it were to be implemented in future a workaround would be required. Code below:

//TODO: find a way of implementing this without having to /10, as this would not work with decimal implementation

//<backspace>

private void Btn\_clrBack\_Click(object sender, RoutedEventArgs e)

{

if (op == "")

{

number1 = (number1 / 10);

numberDisplay.Text = number1.ToString();

}

else

{

number2 = (number2 / 10);

numberDisplay.Text = number2.ToString();

}

}

//</backspace>

Figure 57, Backspace

Next, we come to the miscellaneous category, these are, of course, Tooltips, Identifiers, common formats, Code layout, Comments, and so on and so forth. Below I have set out in what way I have used all of these, and code examples of this.

Firstly, tooltips, and in order to assist the user in some places where their prior knowledge may be lacking, I have taken the liberty of making the addition to tooltips on certain controls, these controls are “Btn\_symPositiveNegative”, “Btn\_symSmiley”, “btn\_clrClearEntry”, and “btn\_clrClearAll”. The tooltips in these instances serve as clarification of some things that may be non-standard or one may not have come across, for example, the XAML for “btn\_clrClearEntry” is as follows (note: the tooltip will be highlighted):

<Button x:Name="btn\_clrClearEntry" Content="CE" Grid.Row="1" Grid.Column="0" FontSize="25" Background="#282833" Foreground="#6564DB" BorderThickness="0" Click="Btn\_clrClearEntry\_Click" Height="Auto">

<Button.ToolTip>

<ToolTip>

Clear Entry (Just this number)

</ToolTip>

</Button.ToolTip>

</Button>

Figure 58, XAML code for btn\_clrClearEntry, note the tooltip, which has been added to avoid any confusion

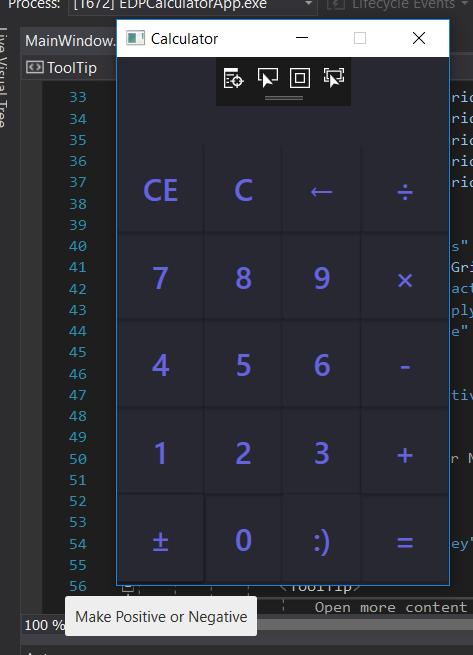


Figure 59, demonstration of the tooltip on the positive or negative button

Next, Identifiers when working within the context of any solution is important. Identifiers serve as a method of allowing us, the developers, to understand at a glance exactly what each control is without having to look at the design view.

For example, the control “btn\_opEquals” is comprised of two parts, “btn\_op”, which lets the developer know what the control is (a button) and what category it’s under (op for operators) and “Equals”, which lets the developer know the exact function.

Moving on from Identifiers but remaining in the same realm of code formatting, common formats of naming conventions (the three main ones being camelCase, PascalCase, and Hungarian Notation) are extremely important to adhere to in order to convey information well and accurately. According to the C# coding standards and conventions on naming rules (<https://github.com/ktaranov/naming-convention/blob/master/C%23%20Coding%20Standards%20and%20Naming%20Conventions.md>) PascalCase and camelCase are to be used in distinct and separate ways, to denote different objects (classes, constructors, local variables, and so on), however Hungarian Notation is not to be used at all, this is due to the fact that it may cause confusion, as well as the fact that it’s essentially useless, as it calls for the type of the variable to be declared at the start of the variable name, which is pointless.

The layout of my code (indentation, appearance, and so on) is fairly consistent, there is not much need really for an extensive amount of indentation in the XAML end of things, with the most being approximately three layers of indentation for tooltips, this is because not many elements have any related lines that can be indented to denote a parent/child sort of relationship.

Additionally, each control occupies a single line of code, this was in an effort to save space and present the code in an understated manner as humanly possible, helpfully, I have ensured that the control attributes (name, row, column, font size, background colour, and so on) are always in the same order, so a future developer will know where they are going to change specific aspects of a control.

### User Documentation

This part of the document is concerned with showing the user how to use the app, for the most part it is extremely simple, however there are some points that must not be glossed over.

Below you will see the calculator app, the app has four primary features and multiple auxiliary features. Fundamentally, the app is designed to take two input numbers, an operator, and output another, third number. The four primary features, or operations, are add, subtract, multiply, and divide, as denoted by the symbols +, -, ×, and ÷, respectively. The = symbol will produce the result of the operation. The CE button clears the current number but leaves the other untouched, the C button clears everything, the backspace button removes the most recent digit, and the ± button makes the number a negative if it’s a positive already and vice versa.

Some notes about the functionality of the calculator, it does not have support for decimals yet, however given the chance to improve upon it this is the first thing I would add. Additionally, located within the files for the app is a text document called CalculatorOutput.txt, this document allows you to go back and look at previous operations you have performed.



Figure 60, the app itself

## Deploying an EDP application as a professional installation

In this part of the document I will be walking through the steps of publication of my EDP app.

After right-clicking the project and clicking the “publish” option, I was greeted with the following windows:

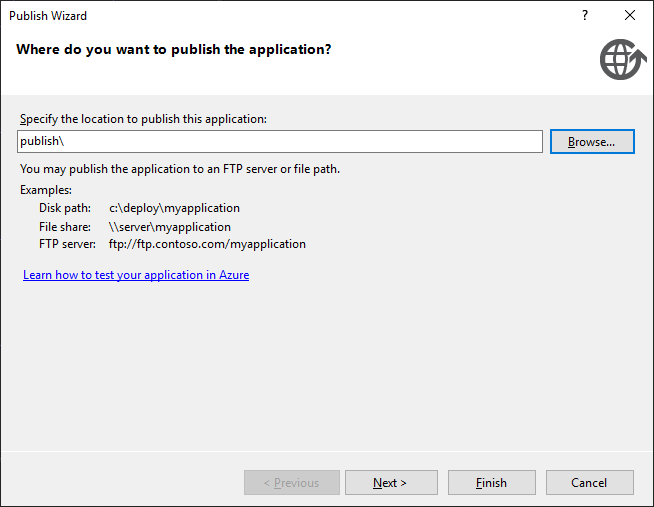


Figure 61, the publish wizard’s first page, the location the user specifies to publish into



Figure 62, the method by which the users will install the application, I have picked CD/DVD-ROM as I do not have access to the other two options at this time

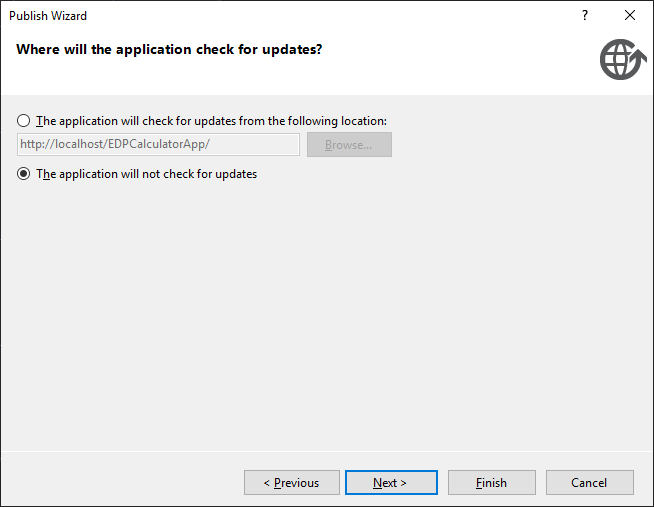


Figure 63, the third window, allowing the option to check for updates to the software

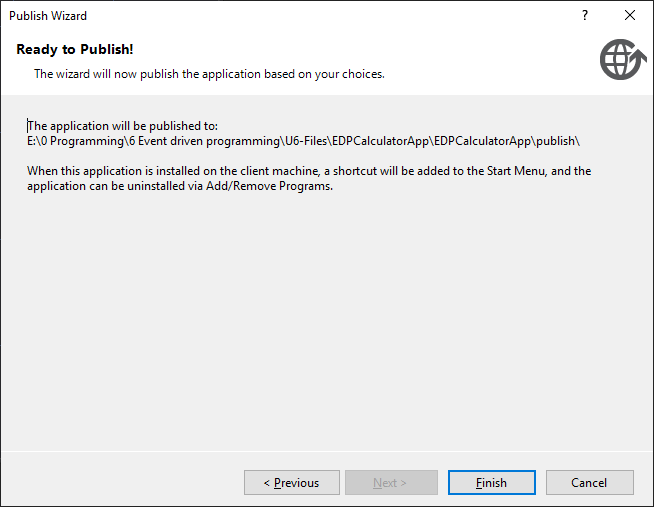


Figure 64, the final window, a review of the options the user has selected and where it will be published

After the wizard, the project began to build into an EXE file, a personal information exchange temporary key was generated in my solution explorer, as can be seen below, and a publish folder was generated and opened in the source files for my application, this folder contained an app files folder, an application manifest file, and a setup file, which, once clicked, takes the user into an installation wizard for my application.

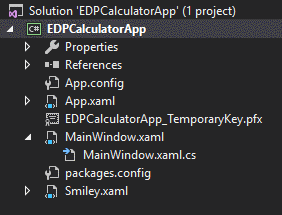


Figure 65, the solution explorer with the pfx key



Figure 66, the file explorer with the publish location open, note the files I have previously mentioned

## Reviewing an EDP solution against the client specification

The client specification, as I understand it, is made up of three primary tasks, each containing several sub-tasks, what follows is my review of my own work against these tasks and sub-tasks.

1. Design an app
   1. Make sure it is a calculator with basic functionality, this can be seen throughout the entirety of the document
   2. Ensure there is documentation, self-evident
   3. Justify a database structure (implementation not required), seen on page 2
2. Develop the designed app
   1. Use correct coding standards and conventions, seen throughout the document but mentioned specifically seen in pages 11-15
3. Deploy the app
   1. Gauge feedback, below
   2. Suggest changes, also below
   3. Review against original design proposal
   4. Present final solution to audience

It is my belief that this document fulfils these requirements as laid out in the spec, the core requirements are essentially filled out, however as a calculator it leaves a fair amount to be desired. Namely functionality surrounding decimals and order of operation, which will be elaborated upon shortly. The third requirement, deploying, feedback, and maintenance, will be described in further detail below, also.

My concern around order of operation in the application is well founded. The order of operation as accepted in all mathematical areas is Brackets, Indices, Division, Multiplication, Addition, Subtraction, or BIDMAS. His, unfortunately, cannot be present natively in my application, because my calculator can only take two operands at a time. This, along with lack of decimal functionality, brackets, and indices, means that one could argue that basic calculator functionality is not fully present in my application.

## Evaluating the EDP Solution using client feedback to calculate an effective maintenance and update strategy

Feedback for my application follows this line, the client mentioned that, given further updating, they would like to see the addition of several core features, namely, being able to use decimals, multiple operands, brackets, indices, and other mathematical operations.

Following this, my update strategy will very much concern the addition of multiple mathematical features. An issue I have had in hindsight in designing the application is that I was not basing it off any one individual calculator, such as the windows one or hardware calculators. To rectify this, going forward I will be basing my development on the Casio FX-83GTPLUS Scientific Calculator, as can be seen below:

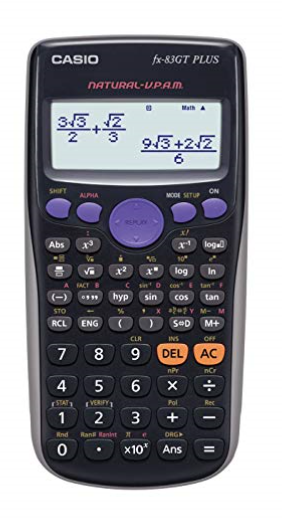


Figure 67, the calculator I will be basing my app on going forward from this point, I am basing it off this one because of my familiarity with it and models like it

As far as strategy for updating goes, I will be working on it over a period of several weeks, ne features will go live every Tuesday, as is standard procedure for development (due to the fact weekends are traditionally time off in development and Mondays are used for final bugfixes). I will most likely be adding features in ascending order of complexity, beginning with an “ans” feature, which returns the earlier answer, and then moving on to decimals, brackets, indices, then fractions, and so on and so forth. This will serve me well as I will be gaining knowledge of how a calculator works fundamentally as time goes on.

Additionally, the client has requested the ability to change themes in the application, going forward I will most likely make further use of the MaterialDesignInXAML library for the construction of a light theme and the addition of a toggle to switch between the two. Further design considerations need to be made with regards to this, but this is the core idea.