Chapter 24 – Working with the IDE

# Objectives

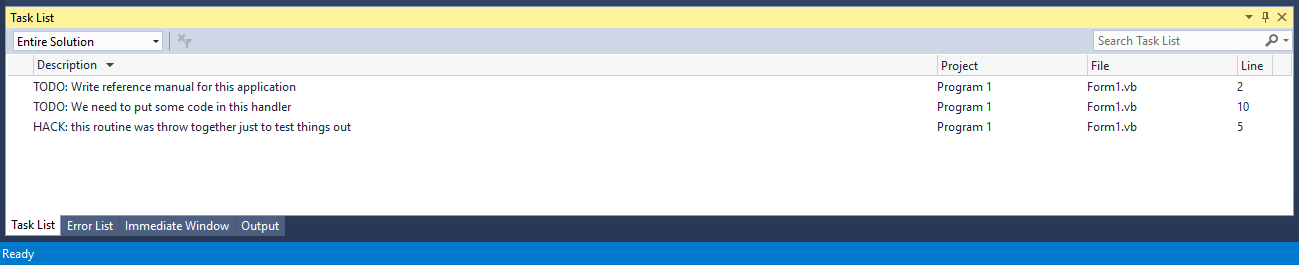
* Task List
* Snippets
* Code Map
* Software Metrics
* Code Analysis Tool
* Profiling
* Unit Testing and Test Driven Development
* Installation Projects

This last chapter will briefly look at some of the other capabilities that are available in the Visual Studio Integrated Development Environment. Simply put this IDE is one of the most powerful and capable on the market. It is the bar that many other development environments strive to hit. When you examine CASE (computer aided software engineering) tools in CIS 424, you’ll see that much of what makes up a CASE tool can be seen in the IDE.

# Task List

Many times, while we are developing code, we need to remind ourselves of things to do or to change. One of the most often used tactics is that we’ll jot the notes down in source code comments and then search our code to find where the items are.

Visual Studio replaces that whole tiresome scheme with a smarter Task List. The Task List can be viewed by clicking on View🡪Other Windows🡪Task List. Once you activate this you’ll notice a new tab in the bottom pane:



We can add tasks that we need to take care of by simply typing in “keywords” in comments inside our source code. To remove items from the task list, we simply remove the “keyword” comment line from the source code, indicating that we’ve taken care of whatever needed to be done.

You may be wondering what the code looked like that had the “keywords” in it to generate the Task List, I showed you above. Well, here it is:

'Chapter 24 - Program 1

Public Class Form1

'TODO: Write reference manual for this application

Private Function MyFunction() As Integer

'HACK: this routine was throw together just to test things out

Return 7

End Function

Private Sub Button1\_Click(sender As Object, e As EventArgs) Handles Button1.Click

'TODO: We need to put some code in this handler

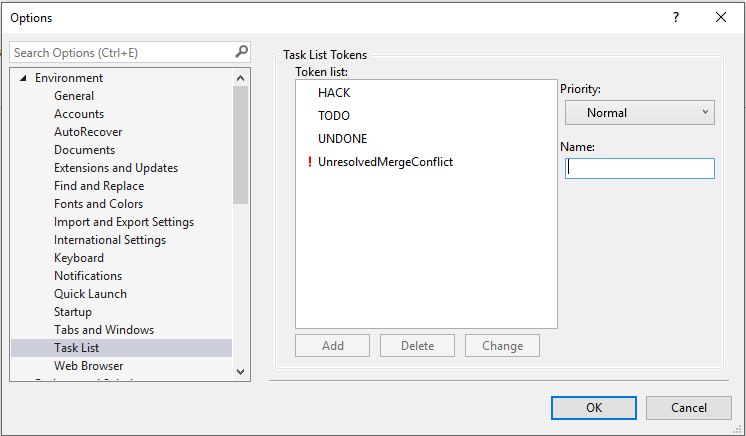
MessageBox.Show("This is the click")

End Sub

End Class

Pretty simple in terms of what the developer must do and now there’s a quick way to jump to where we left off coding or threw things in just to get them to go. To return to the location in the source code where that Task List item was written, I simply double click on the task in the Task List.

If you are interested in creating new “keywords” (they are actually called comment tokens), you can see those values under the Tools🡪Options🡪Environment🡪Task List menu option:



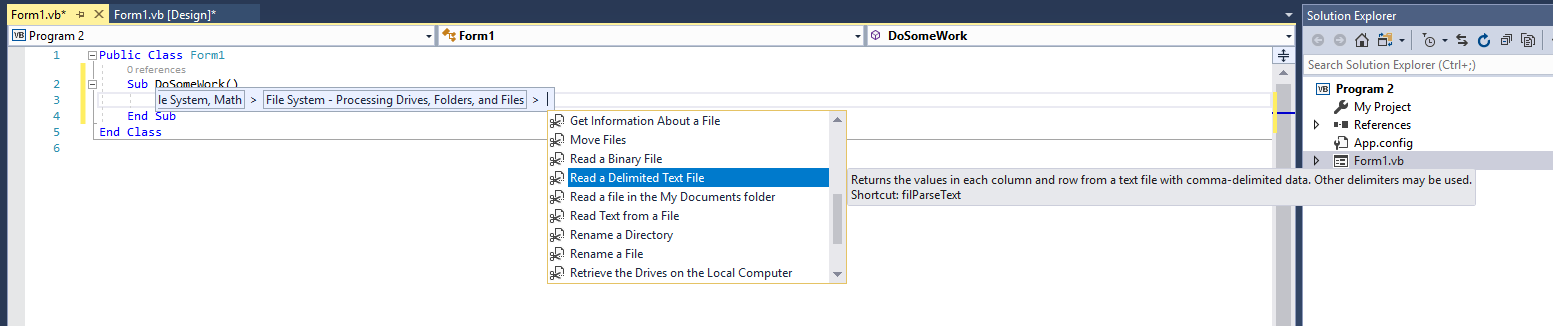
Notice that HACK, TODO, UNDONE and UnresolvedMergeConflicts are predefined for you. You can also assign priority levels. Finally, tasks can be searched, filtered by the project/solution/document level and grouped by file/priority/et cetera. While the Task List isn’t really anything that you couldn’t do with Find, it does make your life a little bit better in a large-scale project.

# Snippets

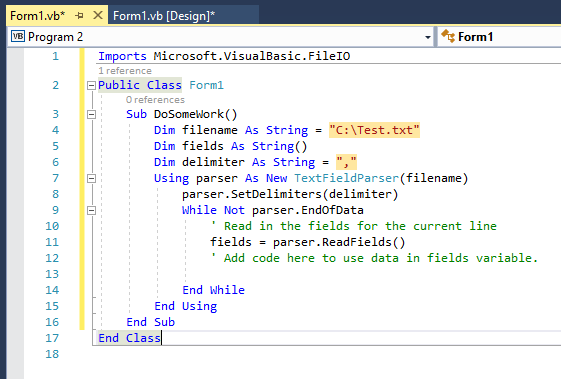
Remember when I told you about the My namespace after we’d pretty much seen everything else? Remember why? I said that I wanted you to learn the proper locations of what My was pointing at – that taking the shortcut doesn’t necessarily help you learn anything. Well, Snippets is in the same vein in my opinion, but they are damn sweet!

Snippets are small chunks of code that you can have inserted into the project you are working on. Snippets are blocks of code that have been marked up with XML so that any replaceable parameters in the code are flagged and so that the snippet can be integrated in with IntelliSense. There are two different ways to access the hierarchical menu-based set of snippets: (A) you can right click in the code editor window and select Snippet🡪Insert Snippet or (B) press CTRL+K, CTRL+X.

When the Snippet window initially shows, you’ll see that everything is neatly categorized and that there are a lot of prebuild common tasks available to use. Here’s what a Snippet selection in process looks like:



You can move back and forth by using the mouse, so if you click on a category that you weren’t really interested in, you can click on the bread crumb list to move back up. After you find a snippet that you want added to your code, simply click on it. VB will then add the code – I selected Read a Delimited Text File and this was the result (Chapter 24 - Program 2):

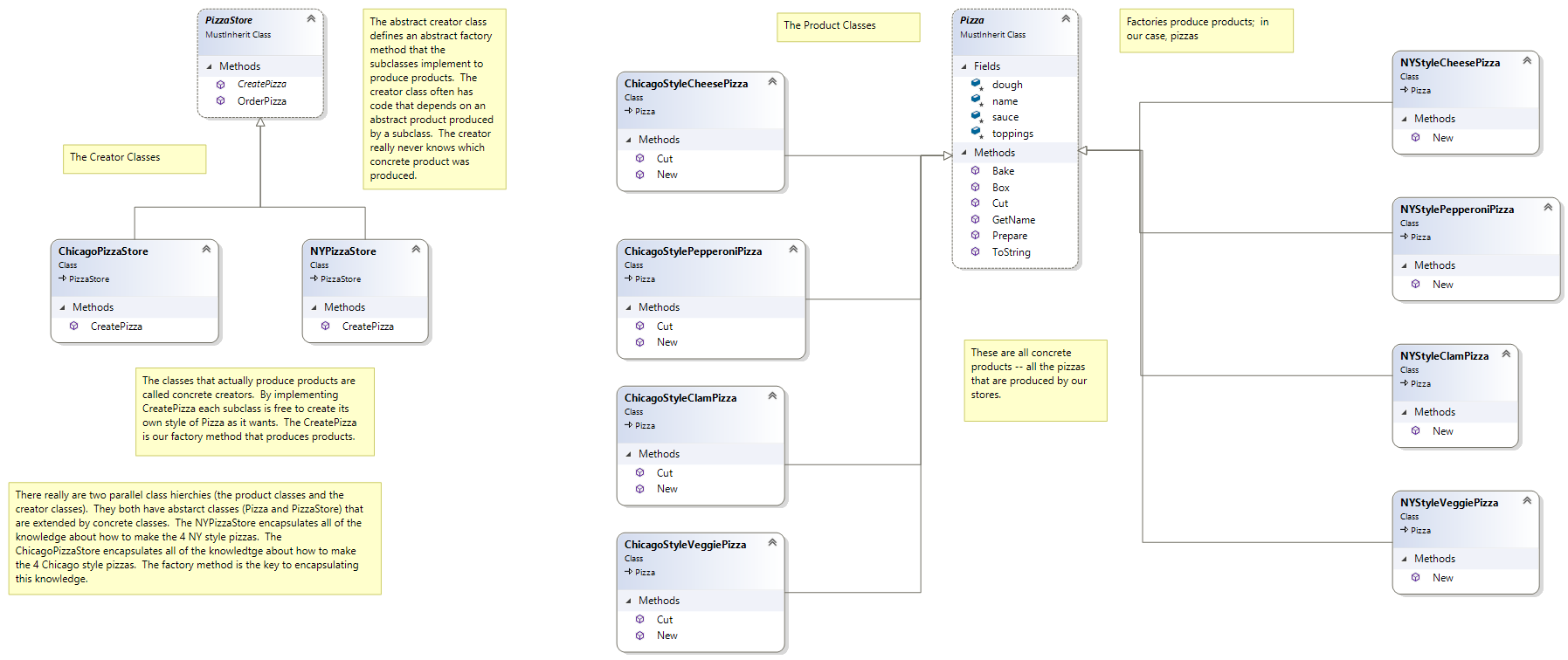


Nice! The whole routine including variables and namespaces is automagically generated. Any parameters that you need to change are highlighted in yellow so that you can adjust them. In this case, the filename and the delimiter between pieces of data should be reviewed and changed as needed.

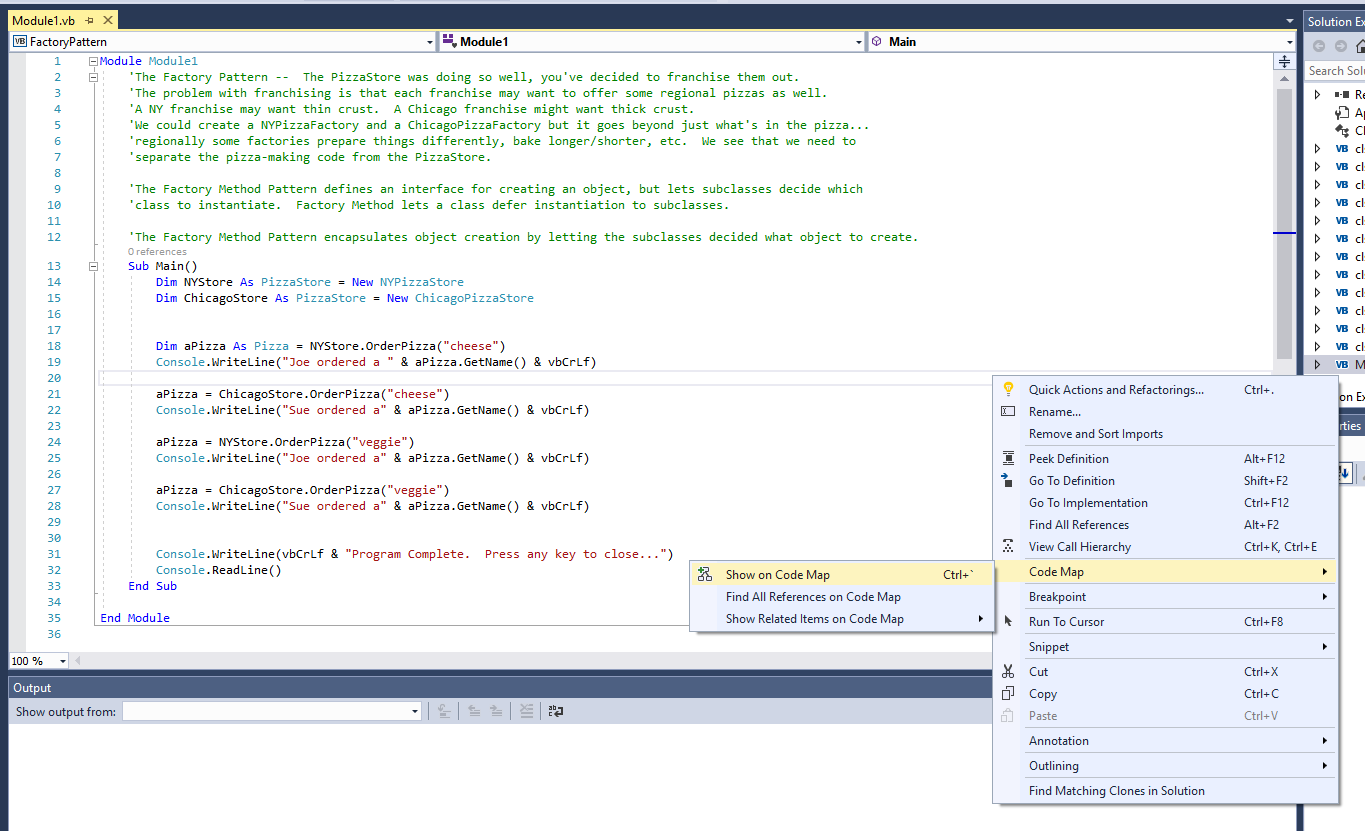
Visual Studio ships with a Code Snippets Manager, but not an Snippet editor to actually create them. Theoretically you could hand edit the XML yourself and then place the file where it needs to go, but that would be a real drag. If you are interested in making snippets, do a bit of Googling and you’ll find some pretty good free ones out there, most notably Bill McCarthy’s Snippet Editor.

# Code Map

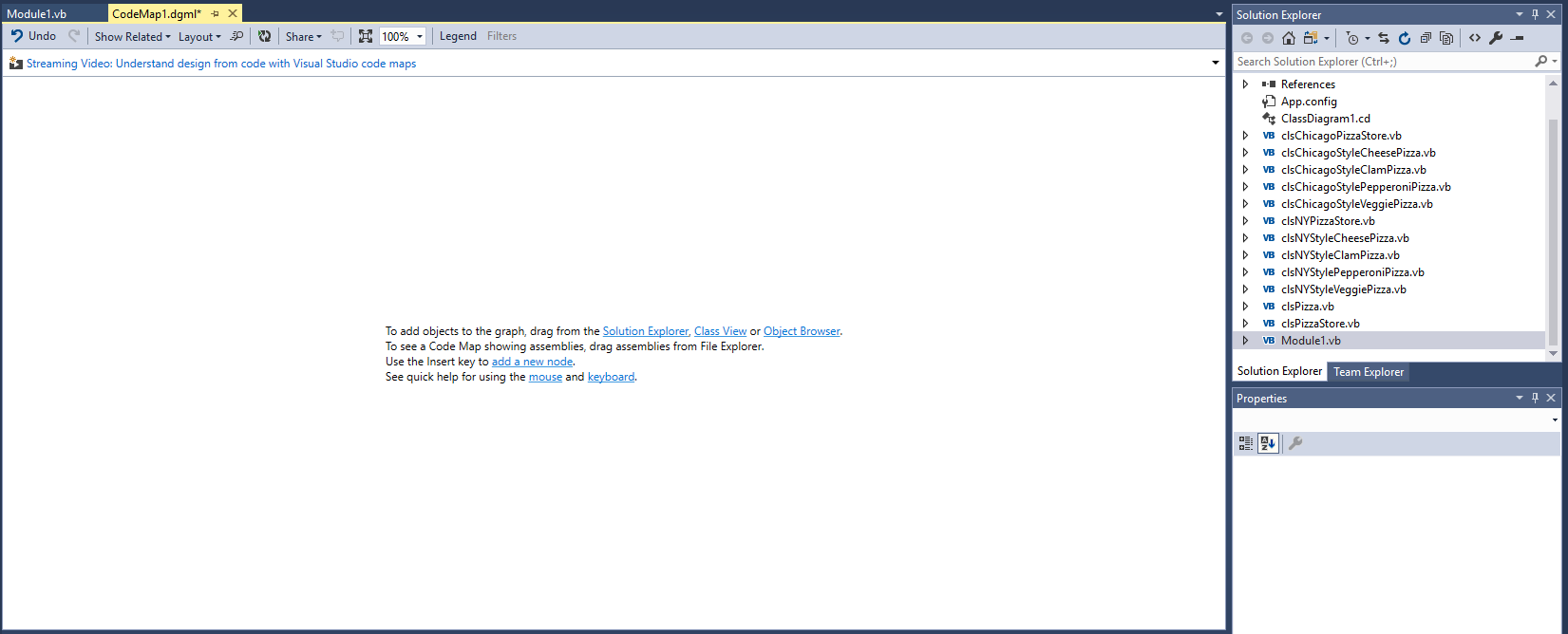
We know that we can create Class Diagrams that show our classes, but there’s not a lot of architectural uncovering that happens there. I’ve opened a Factory design project with a reasonable number of classes and here’s the Class Diagram that I built up to try to show what’s going on:



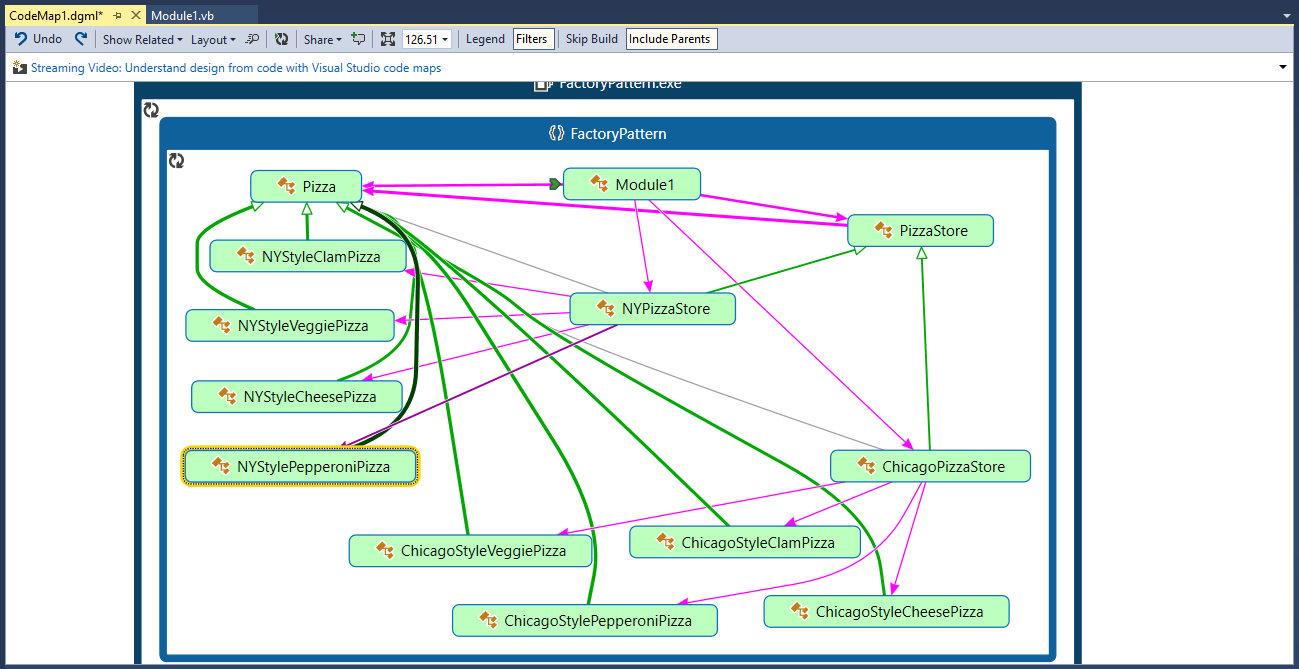
Another tool that we could use to help try to make sense of the architecture is the Code Map. It is a visual diagram of your code showing who calls what. To launch the Code Map, make sure you’re in the editor window of the code that you want to see and then right click, selecting Code Map🡪Show on Code Map from the context menu (Chapter 24 – Program 3):



The next screenshot shows the Code Map window, which is initially empty:

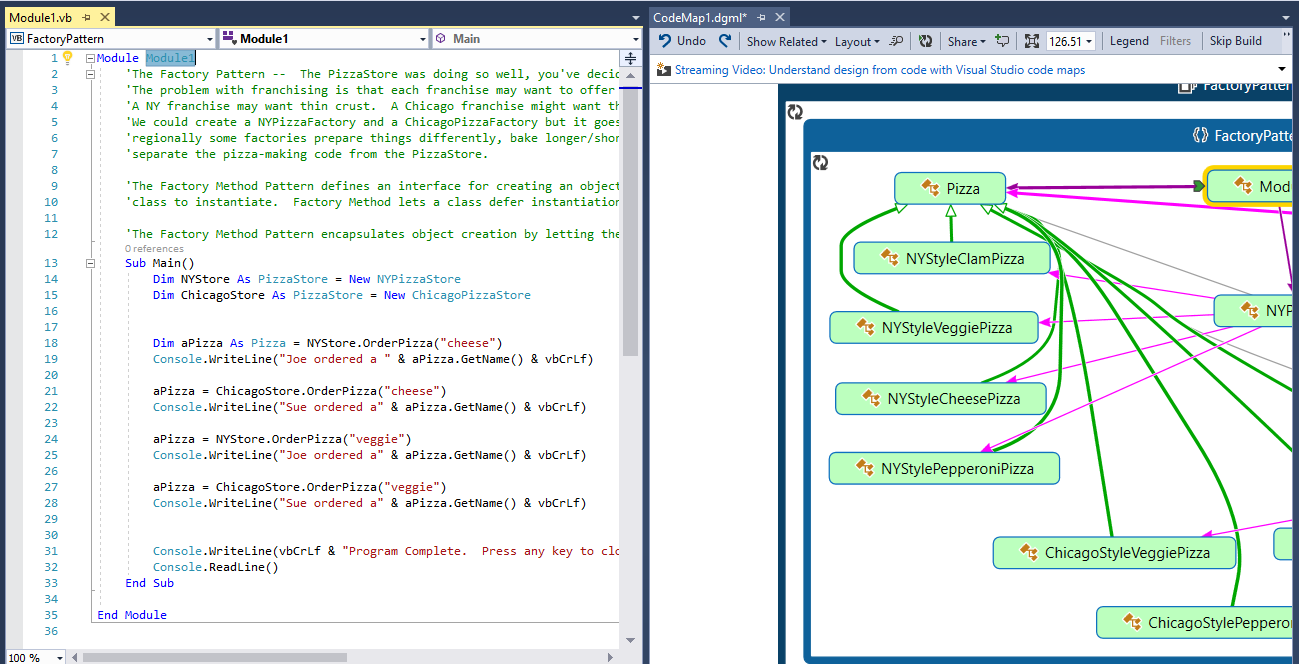


We’re told that we can drag items from the Solution Explorer to the Code Map canvas. Since there’s lots of items over there, I’ll go ahead and drag all my classes and the code module… After a few seconds, I end up with a Code Map looking like this:

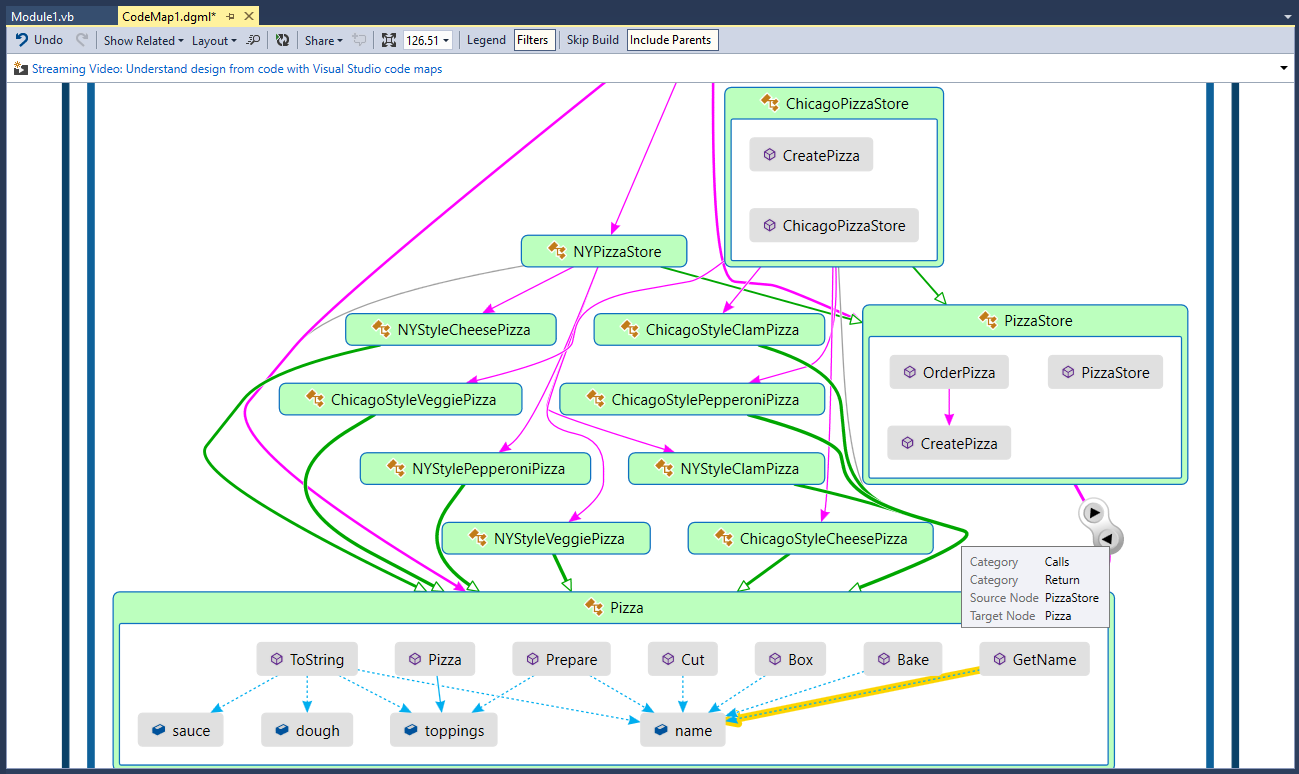


The power in the code map is that you can see who is dependent on whom. We also see architecturally how things are packaged up into namespaces and executables. If double click on any of the entities in the code map, we are taken to the actual source code.

I double clicked on Module1 and the center view pane shows the code map and the source code file:



I can also click the chevron that appears to the left of each entity when I hover over it. Doing so will show the methods calls and uses that are inside of that entity:



If you are trying to explore the architecture of a system that you’ve just inherited, this is a really great way to dynamically do so. Likewise, if you’re not sure of who uses what in a system that you are trying to modify or maintain, you can use the Code Map as a means for discovery. There is a lot more that the Code Map can do, but it gets pretty complex, pretty quick. My recommendation is that you get a good-sized system that you’re comfortable with and look at it using Code Map – that’s the best way you’ll learn what Code Map can do for you.

# Software Metrics

This is another topic that today may not be all that useful to you, but down the road should be. Visual Studio can calculate code metrics on your software to help you understand certain quality aspects of your design. I created a simple Windows Forms application with a single command button click event handler that looks like this:

'Chapter 24 - Program 4

Public Class Form1

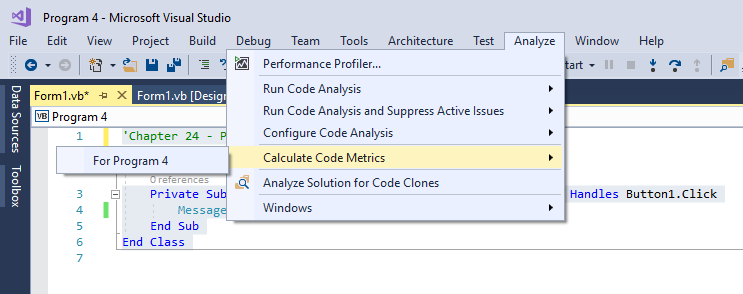
Private Sub Button1\_Click(sender As Object, e As EventArgs) Handles Button1.Click

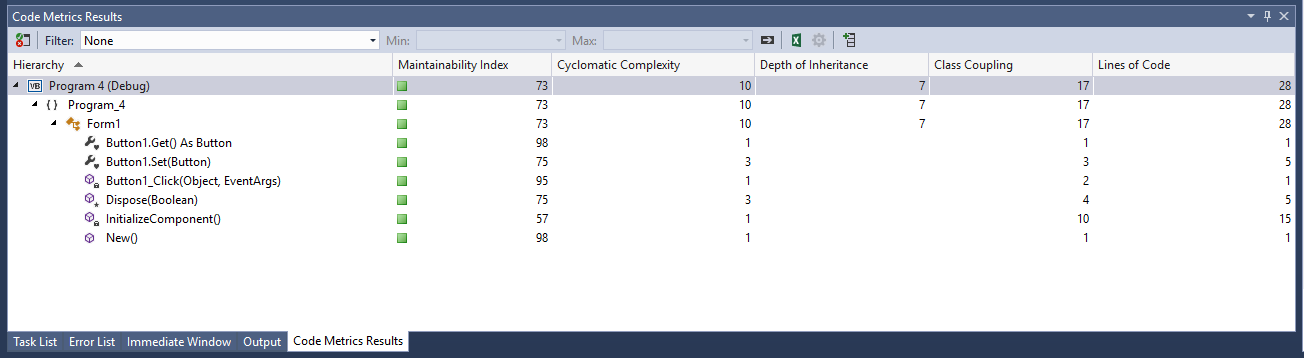
MessageBox.Show("Cool -- you clicked me!")

End Sub

End Class

We can get our code metrics calculated by clicking on the Analyze🡪Calculate Code Metrics menu option:



After a few seconds, the output will appear in a window down near your Immediate window. Here’s the results for our simple little project: 

I expanded the tabs to show that you can drill down into each class and routine. What does each column represent?

* Maintainability Index –this is calculated using a formula with Cyclomatic Complexity, lines of code (LOC) and the Halstead number (again, you’ll learn about these things more in depth in CIS 424 or CS 421) – higher values (up to 100) are better
* Cyclomatic Complexity – this is a measure of how many different execution paths there are in your code. Obviously, the more paths the more difficult it will be to understand and maintain the code. There is no rule of thumb or what’s best with the CC metric; sometimes complex solutions must remain complex. What may be beneficial though is using the number as a baseline marker to know when to refactor code and when to leave it alone. In general, though, we prefer lower values.
* Depth of Inheritance – this is a measure of the number of levels of inheritance present. Lower numbers are again better, but this is dependent on the complexity of the object lattice. This, again, is a rule of thumb count since some Microsoft objects will have depths of 4 or 6, while other third-party components may only have a depth of 1. Does that mean that the third-party component is better? Well only if they don’t break their code on a future release…
* Class Coupling – this is a count of the number of dependencies that an item has on other types except for primitives and built-in .NET types. What does the metric mean? Basically, the more dependencies you have, the harder the code may be to maintain, e.g. what if a data type in a dependency changes or that type isn’t available on some other platform you are trying to port your code to? We prefer lower values on this metric.
* Lines of Code – this is a measure of the number of lines of code in some entity. Again, a rule of thumb – the more code, the harder to maintain. While that’s not necessarily a useful rule at the project or subsystem level, it certainly plays out at the function level.

# Profiling

Many times, we’d like to know where we spend most of our execution time within our code. If we are going to worry about optimizing executing time, that’s where we must start. Again, Visual Studio provides us with a tool that will allow us to profile our code’s execution, e.g. determine how much time is spent in execute and where with regards to the code.

I am going to show you a simple example. I wrote an application that prints the number and the square root of the number, in the range between 0 and 100, for those numbers that are multiples of 5. Yes, I know there are better ways to write it, but that’s the purpose of the profiler…

Here’s the code and the sample execution:

'Chapter 24 - Program 5

Module Module1

Sub Main()

ComputeSqrts()

End Sub

Sub ComputeSqrts()

Dim intLoop As Integer

Dim intValue As Integer

Dim strOutput As String = ""

strOutput = "Select Numbers and Their Square Roots" & vbCrLf

For intLoop = 0 To 100

If intLoop = 5 Or intLoop >= 10 Then

intValue = intLoop

If (intValue Mod 5) = 0 Then

strOutput &= "Number : " & intValue & " Squared : " &

Math.Sqrt(intValue)

strOutput &= vbCrLf

End If

End If

Next

Debug.WriteLine(strOutput)

End Sub

End Module

Output:

Select Numbers and Their Square Roots

Number : 5 Squared : 2.23606797749979

Number : 10 Squared : 3.16227766016838

Number : 15 Squared : 3.87298334620742

Number : 20 Squared : 4.47213595499958

Number : 25 Squared : 5

Number : 30 Squared : 5.47722557505166

Number : 35 Squared : 5.91607978309962

Number : 40 Squared : 6.32455532033676

Number : 45 Squared : 6.70820393249937

Number : 50 Squared : 7.07106781186548

Number : 55 Squared : 7.41619848709566

Number : 60 Squared : 7.74596669241483

Number : 65 Squared : 8.06225774829855

Number : 70 Squared : 8.36660026534076

Number : 75 Squared : 8.66025403784439

Number : 80 Squared : 8.94427190999916

Number : 85 Squared : 9.21954445729289

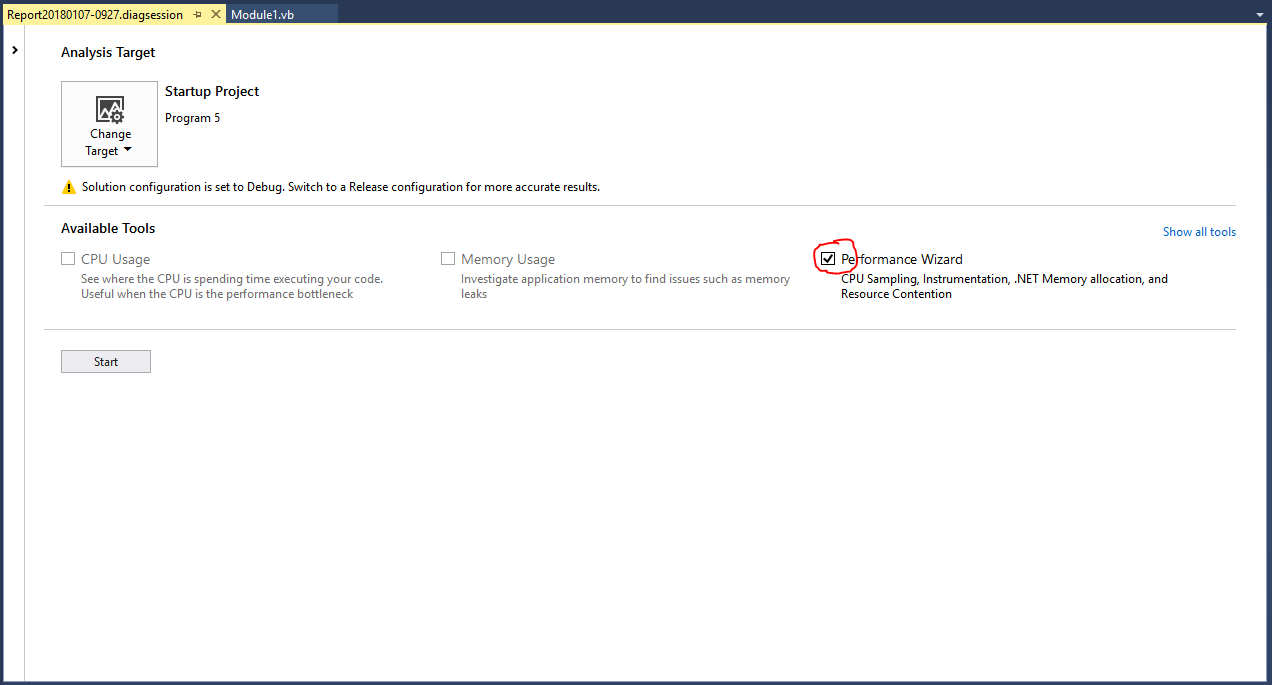
Number : 90 Squared : 9.48683298050514

Number : 95 Squared : 9.74679434480896

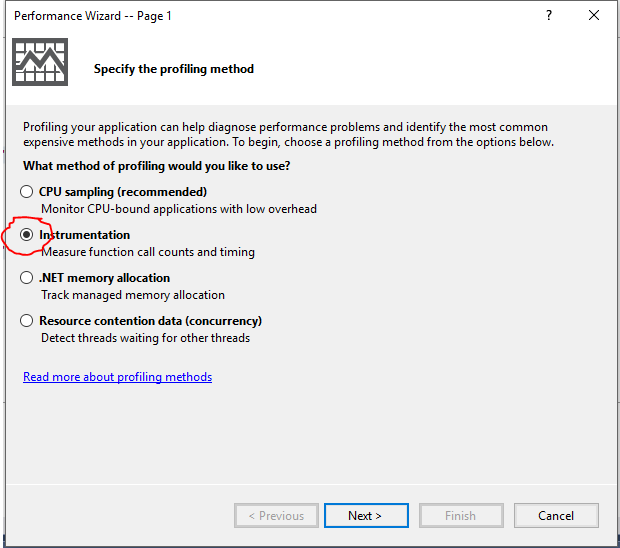
Number : 100 Squared : 10

So, let’s see if we can improve upon this code – we will start the Profiler by going under Analyze🡪Performance Profiler. The Profiler configuration screen appears asking what type of profiling we want to perform. There are lots of things you can do – I am just going to take you through a simple example so that you get a sense of what the Profiler can tell us. If you are really interested in this topic, you need to plan on investing some time reading and researching.

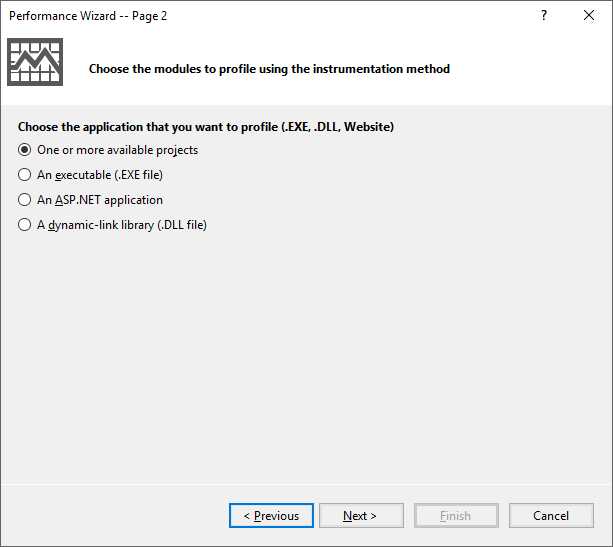
For our purposes, we will select the Performance Wizard by checking it. Next, press the Start button in the middle of the configuration screen:



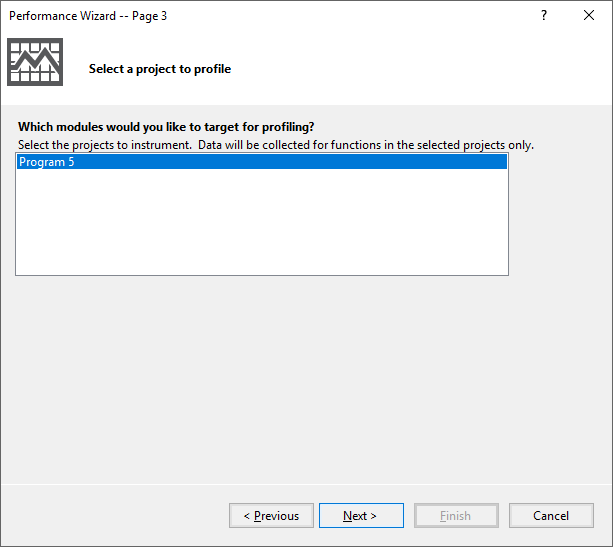
The wizard launches with a few more questions for us. On this screen, select Instrumentation, then press the Next button:



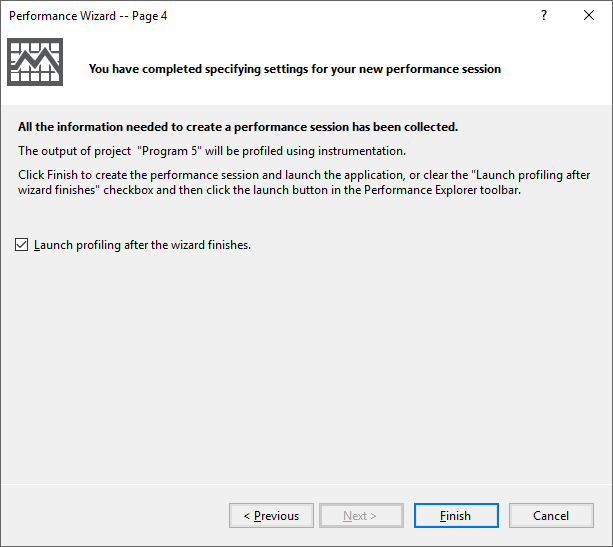
We’re then asked what we want to run the profiler on. Leave things the way they are and press the Next button:



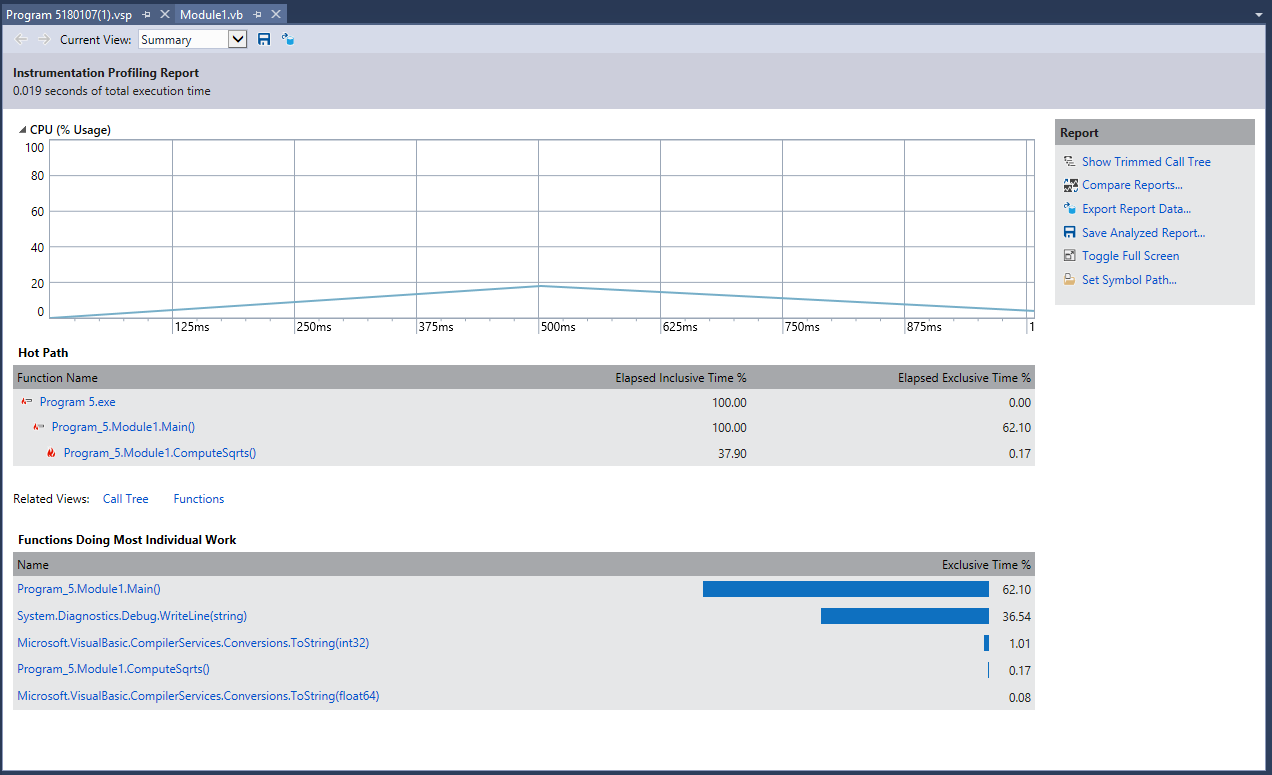
The Wizard now wants to know what modules it should profile. Since we only have on module, leave everything as it is and click the Next button:



Finally, we have reached the end of the wizard. Go ahead and click the Finish button:



After a few seconds, the results of the profiling will appear:



This gives me an overall view of my application and where CPU time is being spent. Again, you’ll need to do some reading and playing around to figure everything out and why it’s important. Now that I’ve got some instrumentation that I can use, I will rewrite my code to make it more efficient and then profile the new version to see what happens. Here’s the improved code:

'Chapter 24 - Program 6

Module Module1

Sub Main()

ComputeSqrts()

End Sub

Sub ComputeSqrts()

'Improvements to the program:

'A) Dropped the extraneous integer variable, using step 5 and no if logic

'B) No string concatentations any more -- calculate and print each line

Dim intLoop As Integer

Debug.WriteLine("Select Numbers And Their Square Roots")

For intLoop = 5 To 100 Step 5

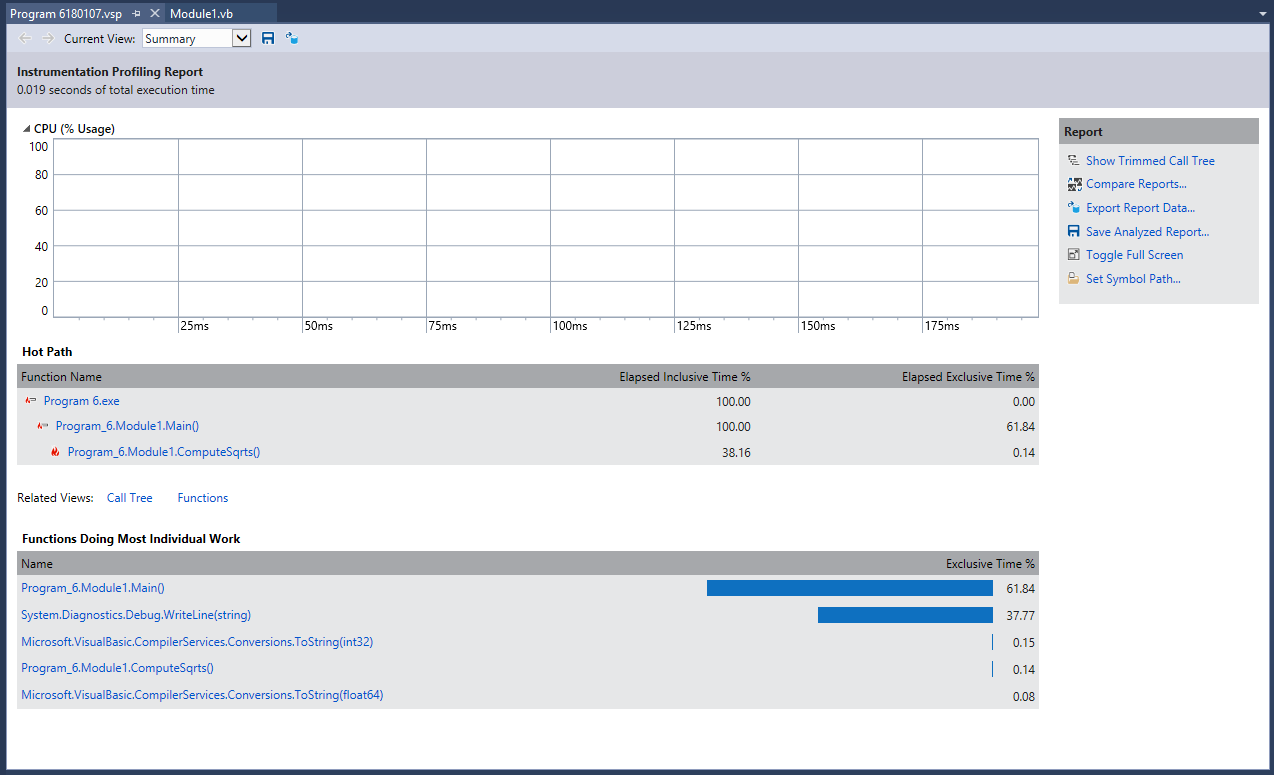
Debug.WriteLine("Number : " & intLoop & " Squared : " & Math.Sqrt(intLoop))

Next

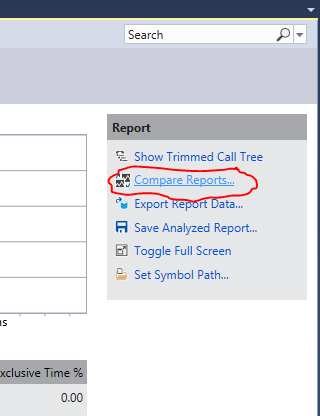
End Sub

End Module

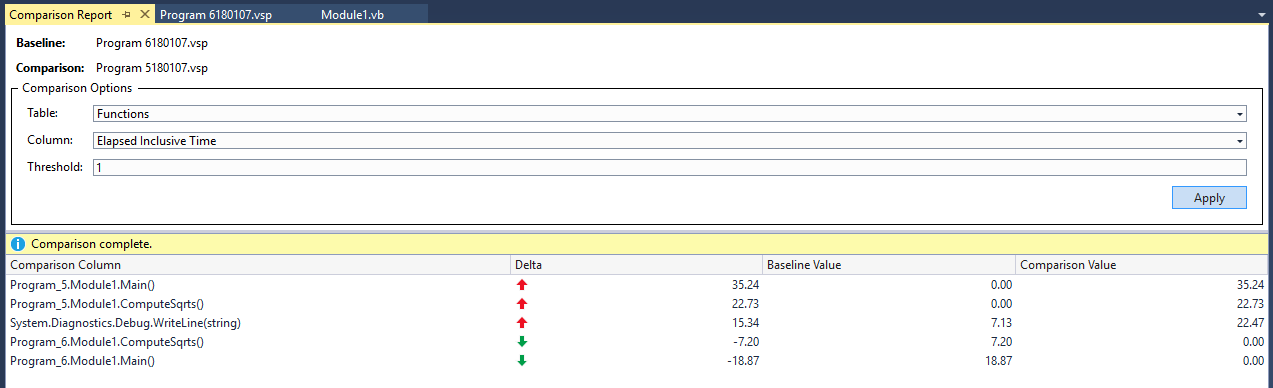
When we execute the program, we get the same output as expected. Let’s try profiling this program following the same steps as above…



One thing that I immediately see, is that there is no CPU build-up in this new version of the program. Compare the profiler’s CPU graphs and you can see this on barely makes a blip. The ToString() conversion value is also way smaller in the new version. Fortunately, we don’t have to do the heavy lifting in trying to figure out which program was better. The profiler can compare two reports:



I asked the profiler to compare the results between my new version of the algorithm (Program 6) and the original version (Program 5). Here’s the comparison on Elapsed Inclusive Time, which is the total time spent in program execution:



In looking at the overall time spent, I have reduced the total execution time of Program 6 over Program 5 by a bit less than 20% (18.87%). The changes to the ComputeSqrts() function accounted for a 7% improvement by itself. This really shouldn’t be a surprise because we know that string concatenation is expensive and we were performing a lot more of it in the original program version than the new version. Also, we came up with better looping logic so we were able to improve upon the loop with the if statement version that the original program had. Nice!

Profiling is a really good way to compare two different algorithms to see which performs better. I could write both Sub DoWorkStyleA and Sub DoWorkStyleB and run my program where the first execution is using Style A and the second Style B and then compare which is better and leave that in the final implementation.

# Unit Testing and Test-Driven Development

When do you test the software you’re building? If I had to guess, I would say most likely after you’ve written a bunch of code. You probably find it boring and tedious, just like documentation – both are attitudes that will result in much poorer quality code than you can produce. Well what if I told you there was a better way: write your tests as you write your code. Sound cool to you? It will help you make sure that you perform adequate testing, that you don’t forget or miss things and it will allow you to retest things if you should happen to change your code. Now do I have your attention?

Test-Driven Development (TDD) is a concept that gets its roots from the late 1990s with the advent of the Extreme Programming (XP) development style. Many companies, even those that don’t embrace agile development, use TDD today. Test-driven development relies on writing test cases early, sometimes even before you have real code (your senior capstone classes will talk more about this). We will constantly be testing through the development cycle to try to locate and remove bugs as early as possible. One thing is certain, waiting until the whole program is written before thinking about testing is a serious recipe for disaster.

## The Assert Class

Many programming languages have the concept of an assert statement built into them. You can simply say, this condition should be true and if it’s not, then we need to error out. The beautiful thing about how asserts work in most languages, is that we can turn them on or off without having to modify code or comment a whole bunch of things out. Think back to your testing in CS 116: you probably put in a whole bunch of printing statements to say, “Program made it to here”, and then either had to comment those lines out or completely remove them before the final executable was produced – lots of tedious, unnecessary work! It’s even more of a drag when you’ve taken all of your instrumentation out and then you need to put it back in later on – turning it on and off would be so much better!

Here are the interesting Assert class methods we want to consider:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| IsTrue | Assumes that some passed in value is true, if not the assertion fails. |
| IsFalse | Assumes that some passed in value is false, if not the assertion fails. |
| AreEqual | Assumes that two values are equal, if not the assertion fails. |
| AreNotEqual | Assumes that two values are not equal, if they are the assertion fails. |
| AreSame | Assumes that two objects are the same, if not the assertion fails. |
| AreNotSame | Assumes that two objects are not the same, if they are the assertion fails. |
| IsNull | Assumes that the return value is Nothing, else the assertion fails. |
| IsInstanceOfType | Assumes that the return value is of a particular type, else the assertion fails. |
| IsNotInstanceOfType | Assumes that the return value is not of a particular type, else the assertion fails. |
| Fail | Immediately fails the current test. |

There is also a StringAssert class with the following methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| StartsWith | Assumes the return value starts with a particular substring, else the assertion fails. |
| EndsWith | Assumes the return value ends with a particular substring, else the assertion fails. |
| Contains | Assumes that the return value contains some substring, else the assertion fails. |
| Matches | Assumes that the tested value matches a given regular expression, else the assertion fails. |
| DoesNotMatch | Assumes that the test value does not match a given regular expression, else the assertion fails. |

Finally, Visual Studio also provides a CollectionAssert class with the following methods:

|  |  |
| --- | --- |
| *Method* | *Property* |
| AreEqual/AreNotEqual | Tests whether two collections contain (or does not contain) the same values in the same order {1, 2} <> {2, 1}. |
| Contains/DoesNotContain | Tests for the presence (or absence) of a particular value within the collection. |
| AllItemsAreNotNull | Assumes that all the items in the collection are not Nothing. |
| AllItemsAreUnique | Assumes there are no duplications in the collection. |
| AreEquivalent/AreNotEquivalent | Tests whether two collections contain the same values {1, 2} = {2, 1}. |
| IsSubsetOf/IsNotSubsetOf | Tests one collection to see if it contains (or doesn’t contain) the items of another collection |
| AllItemsAreOfInstancesOfType | Assumes that the collection only includes items of the same specified type. |

Now that we know what the major classes are, let’s put them to work by writing some unit tests! Assume that we are writing a simple average function in a class. Begin a new Class Library (.NET Framework) project called MyMath.

Here’s the code for the class:

Public Class MyMath

Public Function Ave(ByVal ParamArray arrsngValues() As Single) As Single

Dim sngAnswer As Single

Dim intLoop As Integer

Dim sngSum As Single

For intLoop = 0 To arrsngValues.Count - 1

sngSum += arrsngValues(intLoop)

Next

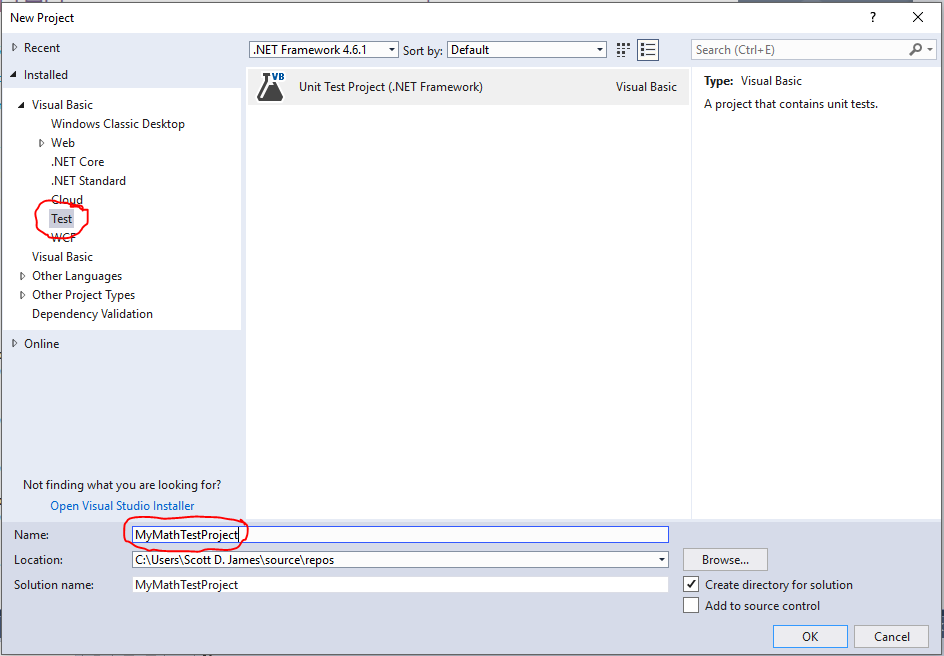
sngAnswer = sngSum / arrsngValues.Count

Return sngAnswer

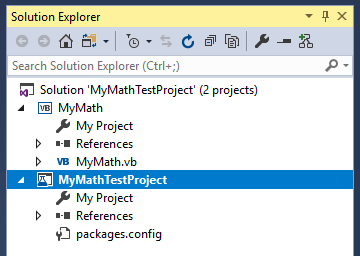
End Function

End Class

You have to agree that’s a pretty simple class with a pretty simple method in it. It’s the perfect time to start writing tests for it. OK, compile (menu option Build🡪Build) your project to a .DLL file and then save the class project. We will now start a new project that is of the type Unit Test Project (.NET Framework). This will be found under the Test section of the Visual Basic projects. Make sure you give the project a better name than the default. I called mine MyMathTestProject:

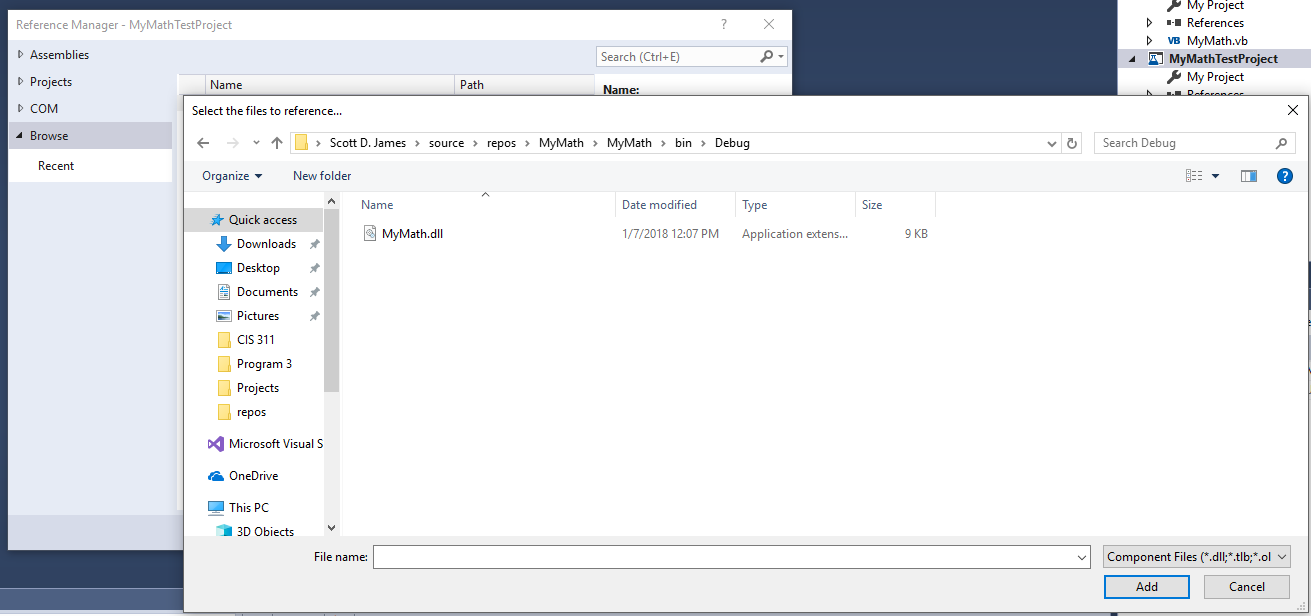


Once you click OK, you’ll see that you get a UnitTest1.vb class created. Next, we need to add the project containing the code that we want to write tests for. I went under File🡪Add🡪Existing Project and browsed until I found the MyMath class project and added it. I also deleted the UnitTest1.class so that I can show you how to add the various test unit types that are available, so go ahead and do that. Once you have things set up properly, your Solution Explorer ought to look like this:



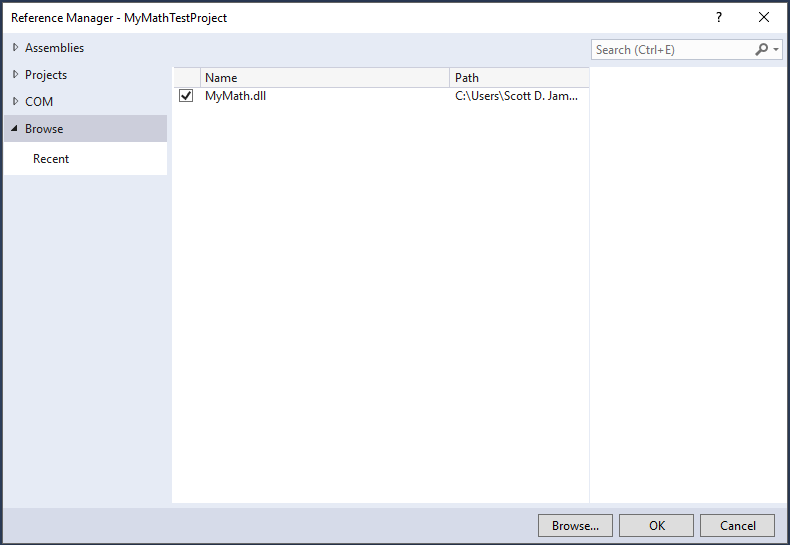
Notice both the MyMath Class Project and the MyMathTestProject Test Project are in this single solution.

We will also need to add a reference to our MyMath object to the unit test library (you must have already compiled this to a DLL as noted above). Make sure that you have selected the MyMathTestProject in the Solution Explorer (it should have the Test Project highlighted as in the screenshot above). Then add a reference to the MyMath object by selecting Browse from the Reference Manager dialog. Browse until you locate you MyMath.DLL file:



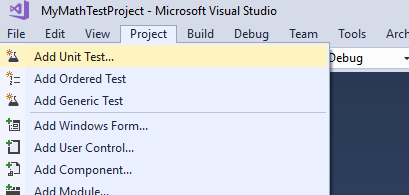
Once you’ve found the DLL file, go ahead and press the Add button.

The Reference Manager should update showing the library checked:



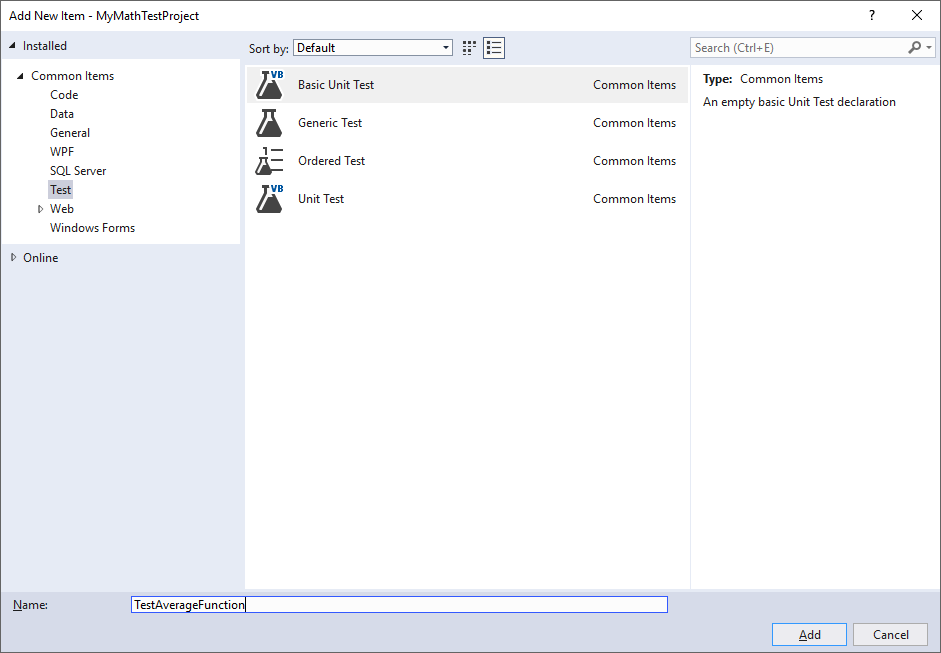
Go ahead and press the OK button to add the reference to the Test Project.

Now let’s see what kinds of testing templates are available for us to use. There are two ways that we can get to the testing templates. If we simply click on the Project menu option, notice that there are different kinds of tests listed right there. The second way would be to use the old Project🡪Add New Item menu option and browse to the Test tab, which would show us the same test choices. VB knows we’re in a test project, so it’s trying to make things simpler by allowing us to just use the Project menu:



It’s interesting to know that we can build different test types. We will be relegating our discussion to just unit tests, but there are also a lot of other testing types you can use: UI testing, generics testing, load testing, web performance testing and so on. When you installed Visual Basic, we didn’t add in any extra testing facilities, so our simple class library tests are limited to the three choices we saw in the screenshot above.

Just to change things up, from the Project🡪Add New Item🡪Test menu, I am going to select Basic Unit Test and I am going to call it TestAverageFunction:



Once the class is added, we see the resulting code in the editor window:

Imports System.Text

Imports Microsoft.VisualStudio.TestTools.UnitTesting

<TestClass()> Public Class TestAverageFunction

<TestMethod()> Public Sub TestMethod1()

End Sub

End Class

You’ll notice there is a skeleton TestMethod1 already written – this shows you the general syntax of what a Test Unit Method will look like. You see that there are a couple of <Test*xxxx*()> decorators in use, so pay attention to those because it’s how VS understands where the testing pieces are – after all the tests and the class are both made up of code. Somehow VS needs to know what’s a test and what’s supposed to be tested... We can keep creating copies of the TestMethod1 skeleton and just give it a new name for each new test we want to write.

Now let’s build our actual test cases – what the average should be when there are just positive numbers, when there are just zeroes, when there are just negative numbers, what happens when we try to average Nothing and what happens when we get overflow and underflow:

Imports System.Text

Imports Microsoft.VisualStudio.TestTools.UnitTesting

<TestClass()> Public Class TestAverageFunction

'We need to add an instance of our class -- this is why

'we had to add the reference to the MyMathTestProject

Dim myMathObject As New MyMath.MyMath

'Now for each test method, we list the method to test,

'the expected result, and what to display should it not work right

<TestMethod()> Public Sub AverageOfNonzeroPositiveValues()

Assert.AreEqual(myMathObject.Ave(1.0, 2.0, 3.0, 4.0, 5.0), CSng(3.0),

"Average of 1 through 5 was not equal to 3.0")

End Sub

<TestMethod()> Public Sub AverageOfZeroValues()

Assert.AreEqual(myMathObject.Ave(0.0, 0.0, 0.0), CSng(0.0),

"Average of 0s array was not 0.0")

End Sub

<TestMethod()> Public Sub AverageOfNonzeroNegativeValues()

Assert.AreEqual(myMathObject.Ave(-1.0, -2.0, -3.0, -4.0, -5.0), CSng(3.0),

"Average of -1 through -5 was not equal to -3.0")

End Sub

<TestMethod()> Public Sub AverageOfBothPositiveAndNegativeValues()

Assert.AreEqual(myMathObject.Ave(-2, -1, 0, 1, 2), CSng(0.0),

"Average of -2 through 2 was not equal to 0.0")

End Sub

<TestMethod()> Public Sub AverageOfNothingShouldBeNaN()

Assert.AreEqual(myMathObject.Ave(), Single.NaN,

"Average of empty array was not NaN")

End Sub

<TestMethod()> Public Sub AverageOfOverflowShouldBeInfinity()

Assert.AreEqual(myMathObject.Ave(Single.MaxValue, Single.MaxValue),

Single.PositiveInfinity,

"Average of MaxValues was not +Infinity")

End Sub

<TestMethod()> Public Sub AverageOfUnderflowShouldBeInfinity()

Assert.AreEqual(myMathObject.Ave(Single.MinValue, Single.MinValue),

Single.NegativeInfinity,

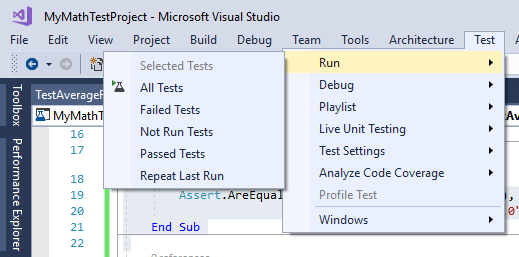
"Average of MaxValues was not -Infinity")

End Sub

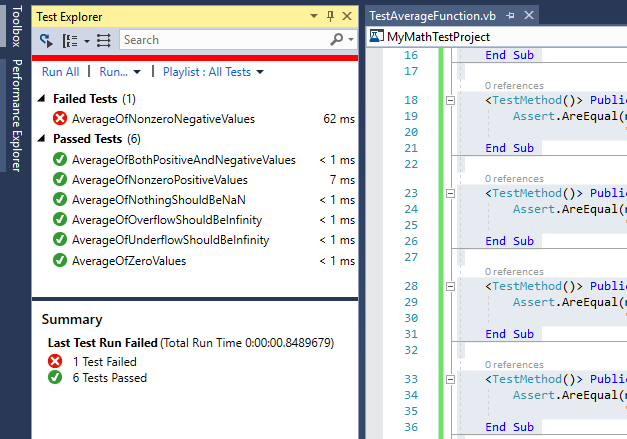
End Class

Notice that we just write test case after test case. Everything in this TestAverageFunction class specifically is there to test out the functionality of the Average method that we wrote in our MyMath class.

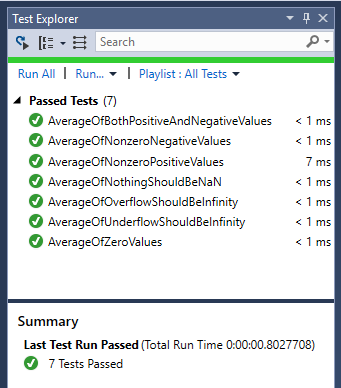
Now that we have our test code, we can run our tests. Don’t click the normal Green play button since that won’t do anything other than give you an error. Instead go under the Test🡪Run🡪All Tests menu option to start the testing process up:



You can see in the Test Explorer that appears to the left of your source code, that 6 of our tests ran fine and 1 failed:

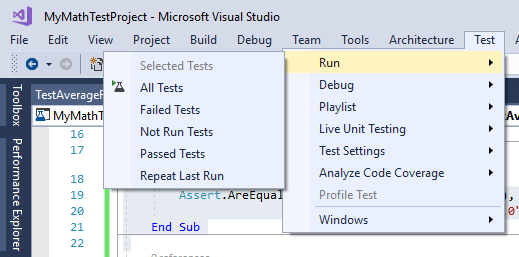


If we carefully examine the code in the failed test routine, we realize that we made a mistake in the value which the answer should have been: we stated 3.0, when it should have been -3.0. Make the change to the test and try it again…this time everything passes!

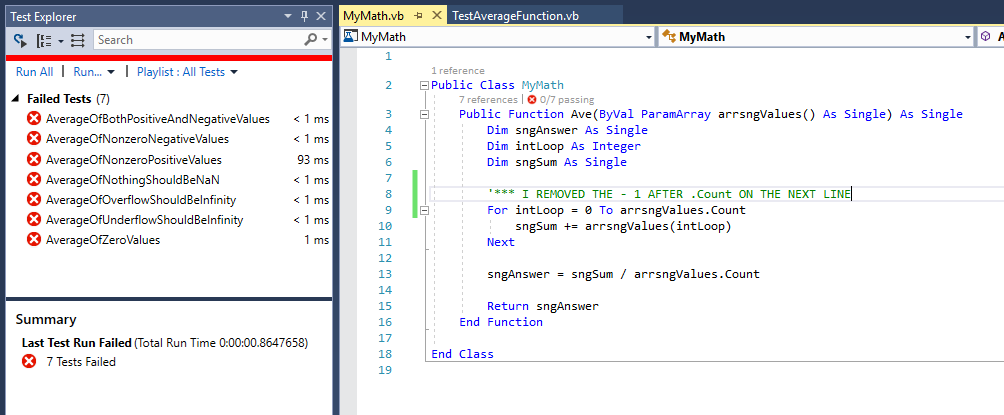


That’s the beauty of TDD, you can test all the time. In fact, if you strictly follow the methodology, you write tests before you write code, e.g. they fail and then you write the code you need to get those tests to pass. If you haven’t thought about doing this before, it’s a pretty interesting exercise and you’ll be amazed at how many mistakes you catch that you used to let through prior to TDD.

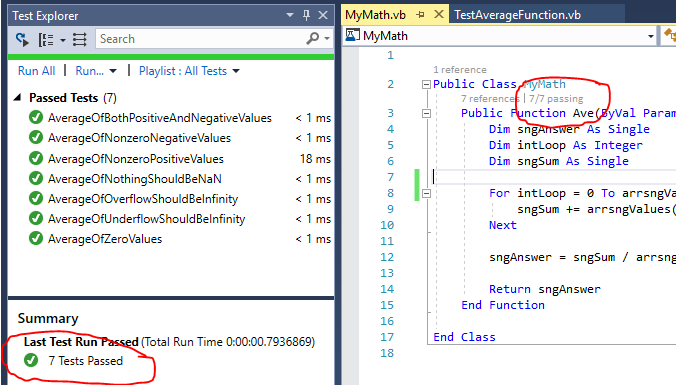
So, while we are currently running all tests, every time, since it doesn’t take long, if you write a bunch of unit tests that could change. You may have also noticed that we can run certain groups of tests: all, those that failed, those that passed and so on. We even can set debug points in our tests and correct those if we have some complex testing we’ve set up. Let’s look at that Test menu again:



Bottom line on testing: when we’re done here, we have a repeatable set of code that will beat on our class to try to ensure that it works right, whenever we ask it to. The tests don’t get lost, they don’t get thrown away, they are a part of the project. Here’s the awesome part – if I break my code, e.g. go into the class file and mess it up, even with a small, insignificant change as in the code on the right:



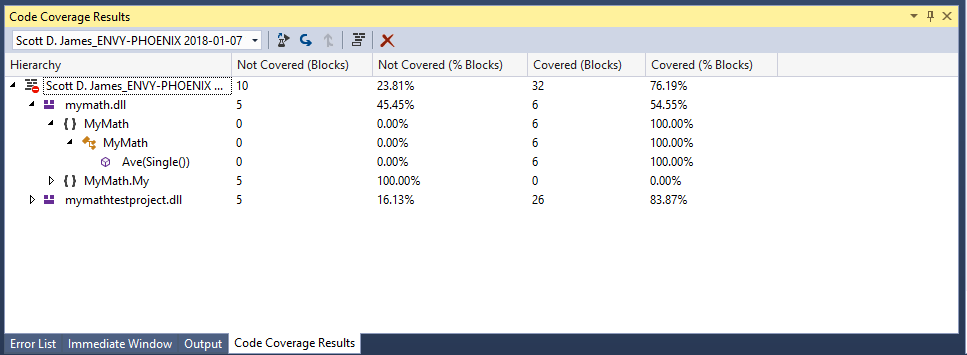
My tests all broke when I run them! I know that I’ve done something wrong in the class and I need to find and fix it. Once I’ve repaired the mistake and rerun the tests, I get success:



How much time have you spent after the fact, going back in and trying to figure out how some change you made affected things? Code that used to work now doesn’t. So, how much time? Too much, I’ll bet! You should be able to see the benefits of having unit testing being handled by the development environment.

This is only an introduction to the testing facilities that are available in Visual Studio. There is Live Testing, which continually runs tests for you as you recompile and change code. VS also contains Coded UI Tests that will record the interactions with a GUI and then replay them. Think about how repetitive testing a GUI would be: click button A, type in value B, check on box C, press button D, scroll listbox E to item 7, and then press button F; as a result textbox G should contain the value… How many times do you want to repeat that process? Yeah, one is the right answer. That’s what Coded UI Tests can do for you. Finally, there is a next level item known as the Microsoft Test Manager that’s integrated into Visual Studio/Team Foundation Server that takes testing to the group level rather than being limited to an individual on a single machine. In summary, there’s lots to learn about testing in Visual Studio!

One last test related discussion is the amount of code that’s covered by our testing efforts. If we click on Test🡪Analyze Code Coverage🡪All Tests, after a few seconds we will receive a result window in the same pane as the Immediate window. If we drill down a bit into the MyMath routine, we see the following:



In my case, my tests covered 100% of the average function which is really what I was after. It may not always be possible to test all 100% of your code 100% of the time due to deadlines in getting deliverables. By properly balancing the responsibility between development teams and QA teams we can do a very effective job however!