C# Code Contracts Succinctly

by Dirk Strauss

Syncfusion | Technology Resource Portal

C# Code Contracts Succinctly

By
Dirk Strauss

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Dirk Strauss is a Microsoft .NET MVP with over 13 years of programming experience. He uses his love for code and technology in general as inspiration to learn and share as much as he can. He has extensive experience in ERP systems (specifically SYSPRO), with warehousing and inventory being the areas that interest him most.

He currently works for Evolution Software (the best company in the universe), which is situated in Cape Town, South Africa. Learning from such skilled individuals who are passionate about technology is the perfect environment where he can live out his passion and creativity.

He is married to Adele, the most stunningly wonderful woman, who gave him two beautiful children, Irénéé and Tristan. They are his rock upon which the troubles of this world flounder and break.

He is always busy after hours tinkering with something or other, being interested in too many things when it comes to technology. Blogging and writing are creational outlets for him; you can find him on Twitter at @DirkStrauss, and at his blog, Dirkstrauss.com.

“There is a different breed of person that can be found in the lonely hours of the night, faces illuminated by the glow of a computer screen, vamping on technology.” ~ Dirk Strauss

Chapter 1 Getting Started

What are Code Contracts?

Many developers might have an idea what Code Contracts are, but they don't have an understanding of the benefits they provide or how to implement them. I have to mention, though, that I've found the attitude toward Code Contracts and their use in development is somewhat divided. There is a group of staunch supporters on the one hand, and on the opposite end of that spectrum, there is a group of opponents. In the middle, however, is a broad group of developers who are unfamiliar with Code Contracts and their uses. It is here that I suspect many of you will find yourself. After working with Code Contracts for a while, I would expect that many of you will naturally migrate to one end of this love-hate spectrum.

Created by RiSE (Research in Software Engineering) at Microsoft, Code Contracts have been around for a number of years. As a matter of fact, the release notes on the RiSE website start at release 1.1.20215 on February 23, 2009. Code Contracts have gone through various revisions throughout the years, and have now found a new home on GitHub after being open-sourced by Microsoft.

Before we continue, let's define what Code Contracts are exactly. The goal of Code Contracts is to provide a language-agnostic way to convey code assumptions in your .NET applications. Let us use the analogy of an actual contract between yourself and a third-party (for example, a bank). It is an agreement between the two of you to ensure both parties act in a specific manner. This is at the heart of Code Contracts. We are assuming certain logic within code, and we define contracts within that code in order to maintain those assumptions. These assumptions can take the form of preconditions, postconditions, and state invariants. If at any time those contractual conditions are broken, we will know about it. This is especially valuable when working in teams.

Download and installation

For the rest of this book, I will be referring to Visual Studio Enterprise 2015 (.NET Framework 4.6) and Code Contracts version 1.9.10714.2. Code Contracts will also work with Visual Studio 2015 Professional, but unfortunately, at the time of this writing, Code Contracts do not work with the free Visual Studio 2015 Community edition. Getting started with Code Contracts in Visual Studio is very easy. From the **Tools** menu in Visual Studio, click **Extensions and Updates**.

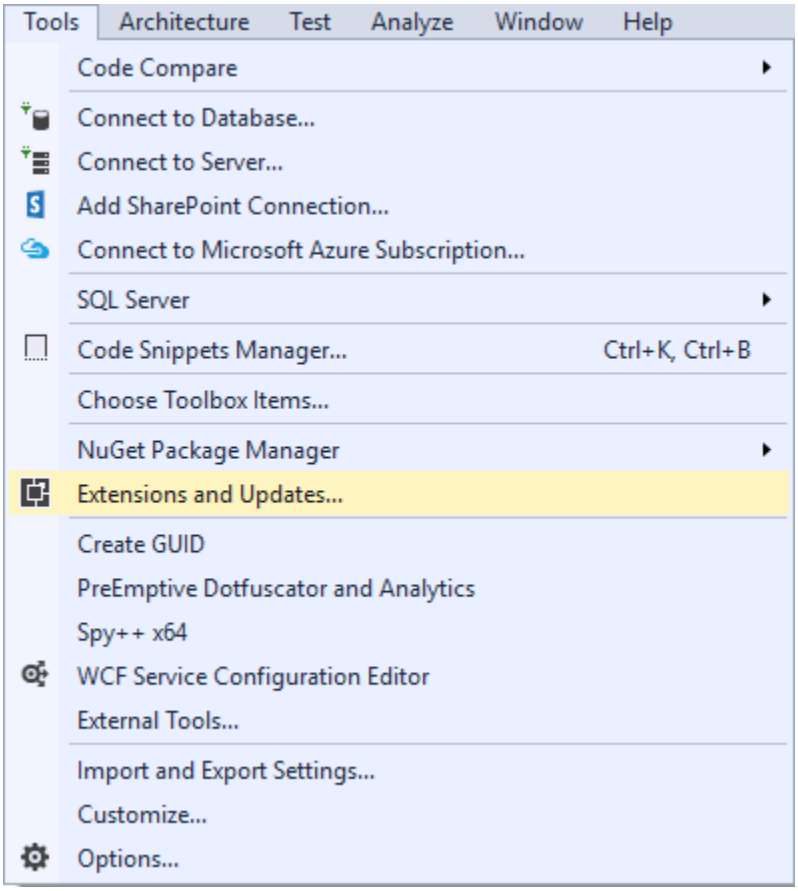

Figure 1: Extensions and Updates Menu Item

This will open the **Extensions and Updates** window (Figure 2), from which you can search the Visual Studio Gallery. Incidentally, you are also able to include Code Contracts in your project via NuGet (more on that later).

In the search box to the right of the screen, enter **Code Contracts** and be sure to have the **Online** section selected in the tree view to the left. When your search results are returned, Code Contracts should be one of the top results. From here it is easy to install Code Contracts. Simply click the **Download** button, which will download an .msi installer that you can run to install Code Contracts.

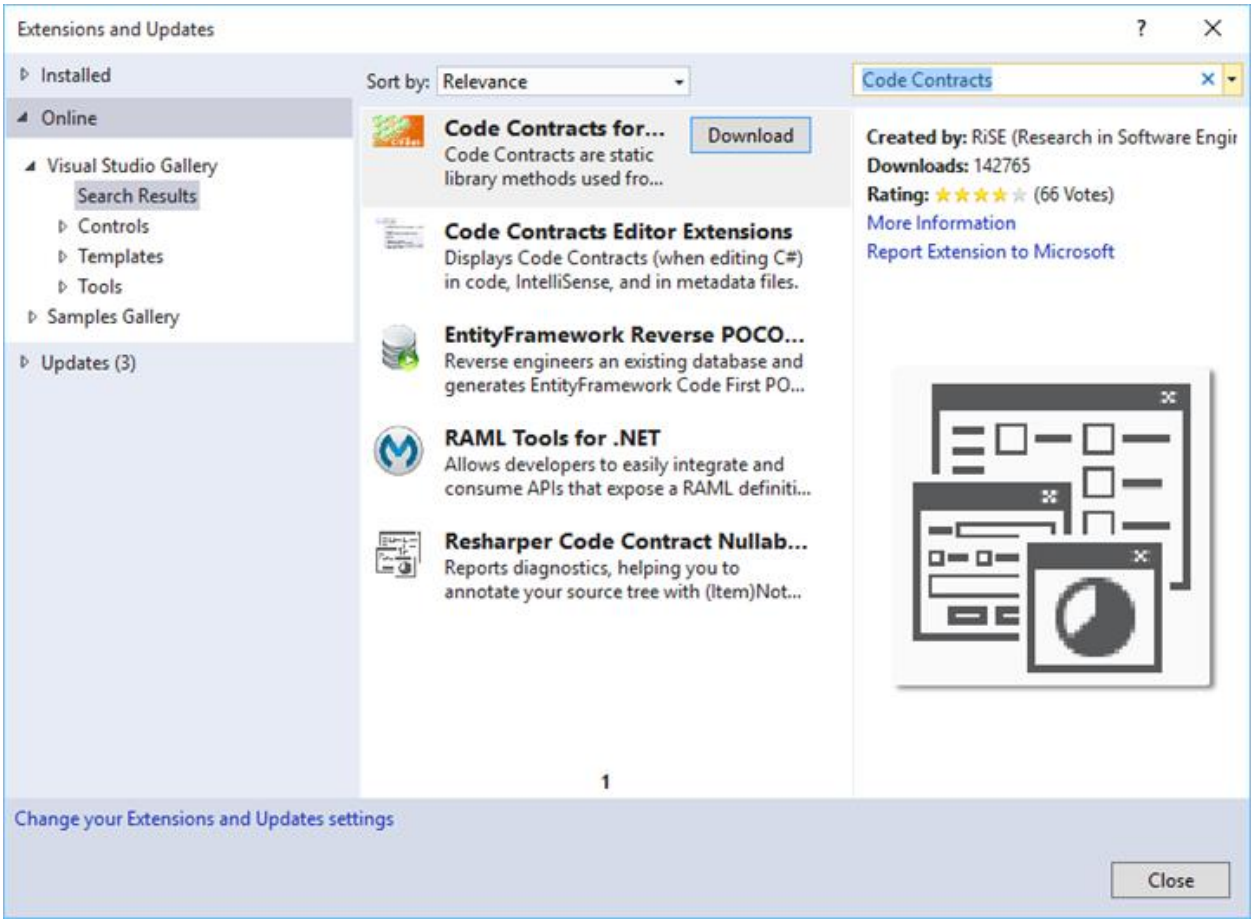

Figure 2: Extensions and Updates

NuGet

The NuGet package manager can also be used to download and include Code Contracts in your solution. You can use the NuGet Package Manager (Figure 3) to search for Code Contracts, or you can use the Package Manager Console from the Tools > NuGet Package Manager option to run the following command to install Code Contracts.

```
PM> Install-Package CodeContracts
```

You will also find a portable version (for use with Windows Mobile applications) of the Code Contracts package on NuGet. It doesn't really matter which method you use to install Code Contracts; it's just a matter which method you prefer.

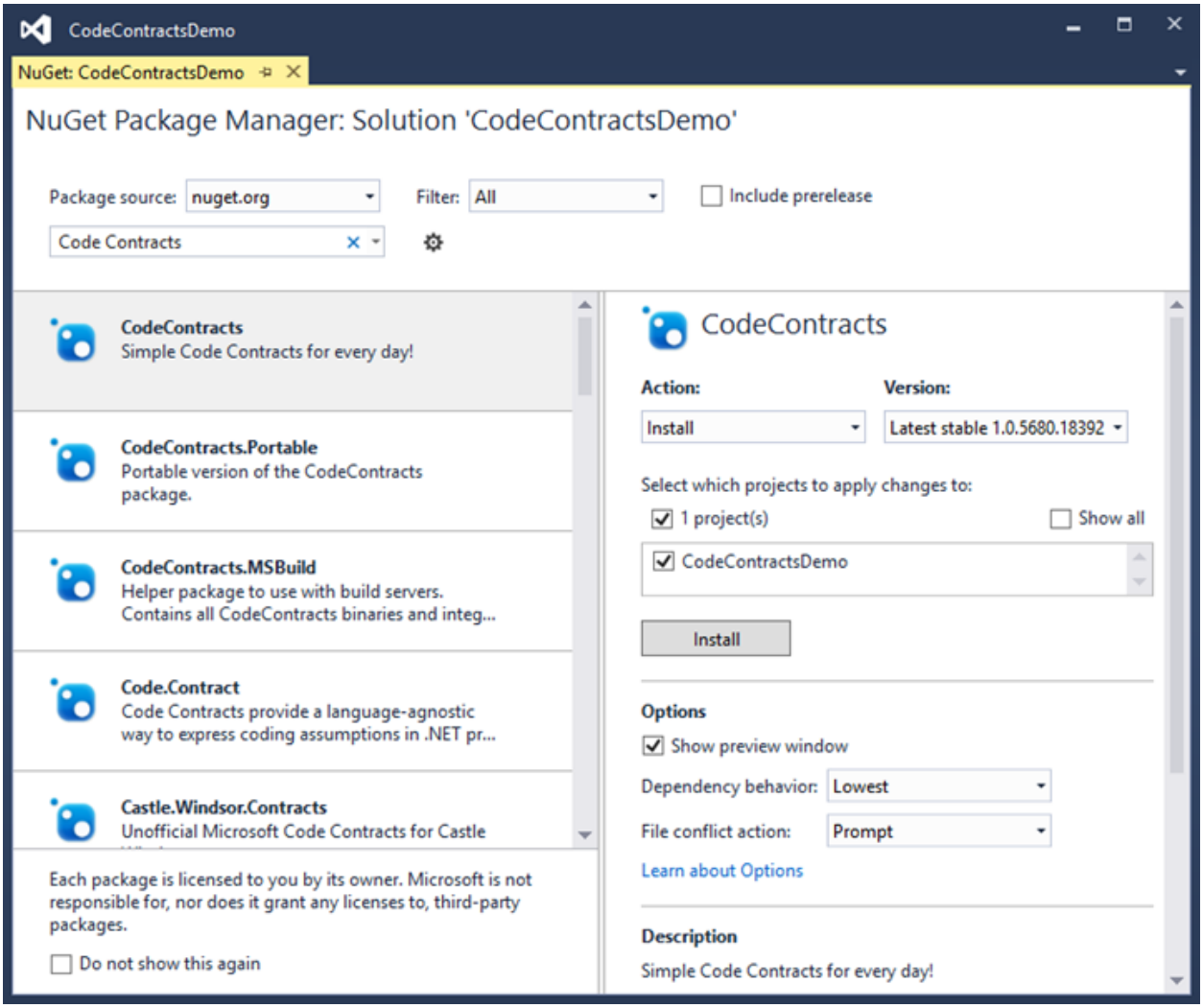

Figure 3: NuGet Package Manager

Visual Studio integration

You will need to restart Visual Studio for the Code Contracts item in the Properties page to become visible. The Code Contracts integration can be found by right-clicking on your project and selecting **Properties** from the context menu. You can also find the Code Contracts integration by selecting **Project** from the Visual Studio toolbar (be sure that you have selected your project in the Solution Explorer), and clicking the ***ProjectName* Properties** menu item (Figure 4).

I have called my Visual Studio project **CodeContractsDemo**, so the item in the **Projects** menu in Visual Studio will be called **CodeContractsDemo Properties**. The familiar Property page opens for your project, and at the bottom of the list on the left of the Property page you will see a new property pane called **Code Contracts**.

Clicking on this will display a smorgasbord of settings for you to select. Don't let the number of settings intimidate you; we will go through these in the next section. Only know that many of the settings are defaulted, and you can leave them that way should you wish to.

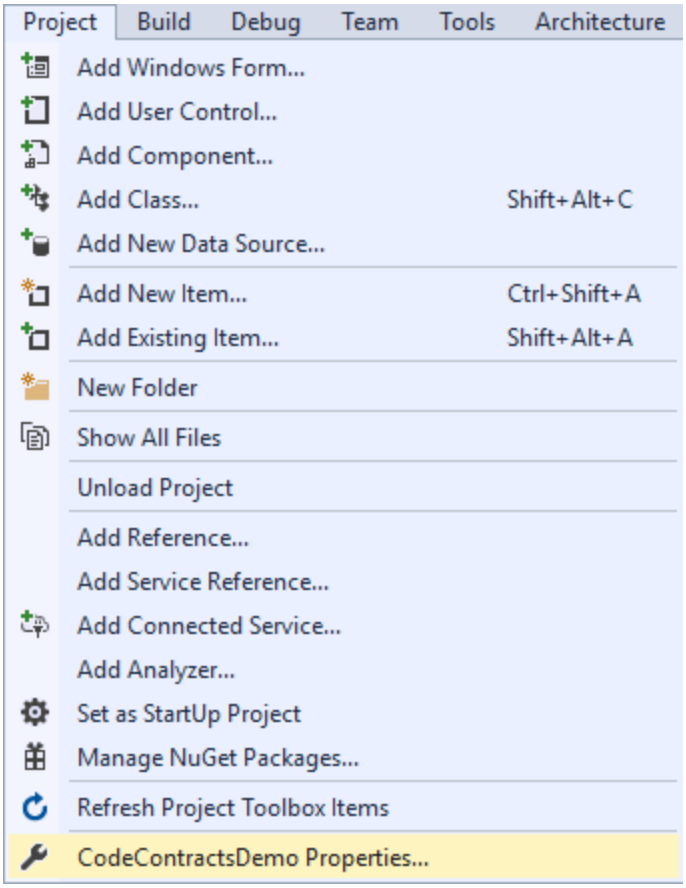

Figure 4: Project Properties

Code Contracts property page

The Code Contracts properties will expose many options and settings to you initially. The properties under Code Contracts are separated into three groups: Runtime Checking, Static Checking, and Contract Reference Assembly.

Enable Code Contracts by selecting either **Perform Runtime Contract Checking** or **Perform Static Contract Checking**.

At minimum, you need to ensure that one of these options is selected in order to enable Code Contracts in your project. Without selecting either of these options, Code Contracts will not be enabled in your project, and you will not be able to benefit from the value they add to your system.

Runtime Contract Checking, as the name suggests, will work its magic during run-time. Static Contract Checking, however, is a different animal altogether. It lets Code Contracts analyze

your code while you're typing code or building your project. This is where Code Contracts become interesting and add the most value for me.

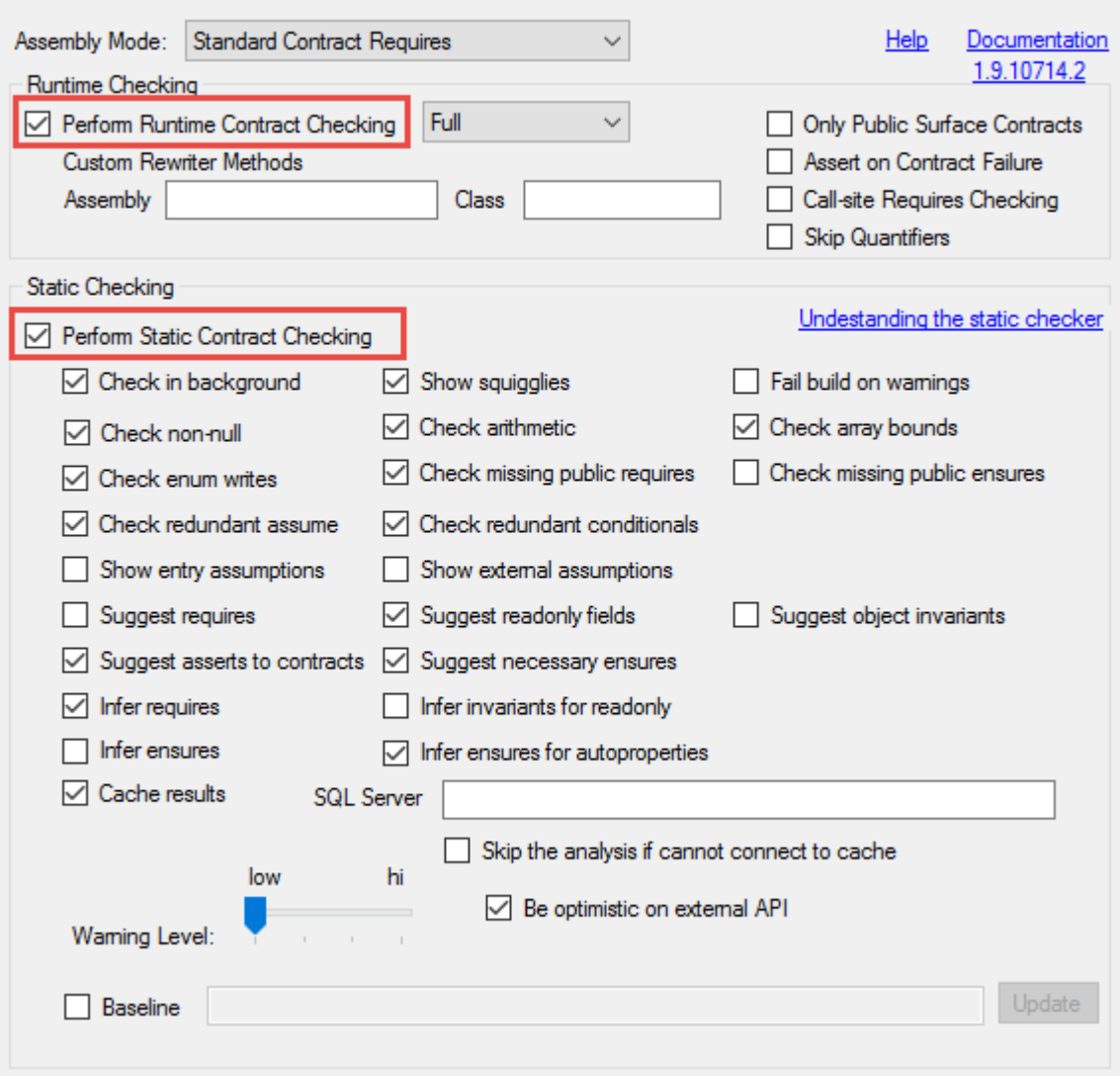

Figure 5: Code Contracts Property Page

Static checking

If you enable static checking, it is advisable to keep the **Check in background** option selected in the Properties page. This is because analysis can take some time to perform. Another advisable option is to select the parallel build option in Visual Studio. From the **Tools** menu in Visual Studio, select **Options**. In the tree-view on the left of the Options screen (Figure 6), expand the **Projects and Solutions** node. You will notice a node called **Build and Run**. Select that, and you will see the option to enable parallel builds.

Any value greater than **1** will enable the parallel builds option in Visual Studio. The default value on my machine is **8**. Entering the value **1** in this text box will effectively switch off parallel builds in Visual Studio. By enabling parallel builds in Visual Studio, you enable Code Contracts to

analyze several projects in parallel. According to RiSE, the Visual Studio extension is thread-safe. There are many other options available in the Code Contracts property page, but you can now start to use Code Contracts without having to configure these settings any further.

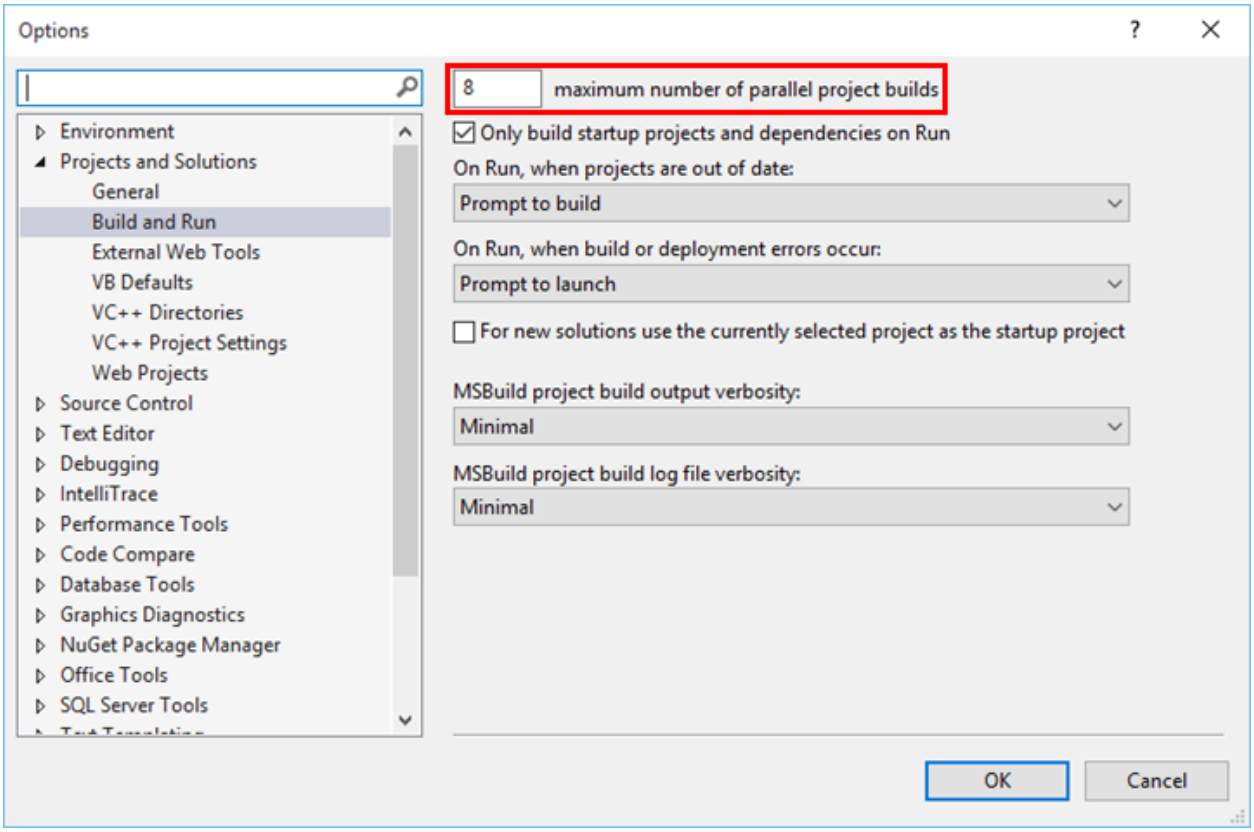

Figure 6: Visual Studio Parallel Builds Option

Chapter 2 Using Code Contracts

Note: The code samples in this book are available at bitbucket.org/syncfusiontech/c-code-contracts-succinctly.

A real-world example

There are a lot of examples illustrating the use of Code Contracts. A lot of them seem to use primitive non-real-world examples to explain the use of Code Contracts. While I understand the benefit of this, I feel that using a real-world example is preferable. So what would classify as a suitable example? Well, a couple of years ago I was working on a system that was being implemented to replace an existing ERP system. Both the old and new ERP systems used numerical serial numbers, but the integration we developed for the new ERP had to use only the new serial numbers.

The old ERP system was being retained for historical purposes, so the integration had to ensure that it only used the serial numbers that would be valid for the new ERP system. We decided to start all serial numbers above one hundred million. This meant that the first serial number created would be 100,000,001. Any serial number lower than this (while valid) should not be allowed into the new ERP system.

In Visual Studio 2015, start by adding the following **`using`** statement to your class.

```
using System.Diagnostics.Contracts;
```

Code Listing 1: Required Using Statement

If you start writing the code for the Code Contracts without the **`using`** statement, Visual Studio 2015 will prompt you to add it via the new light bulb productivity feature, as shown in Figure 7.

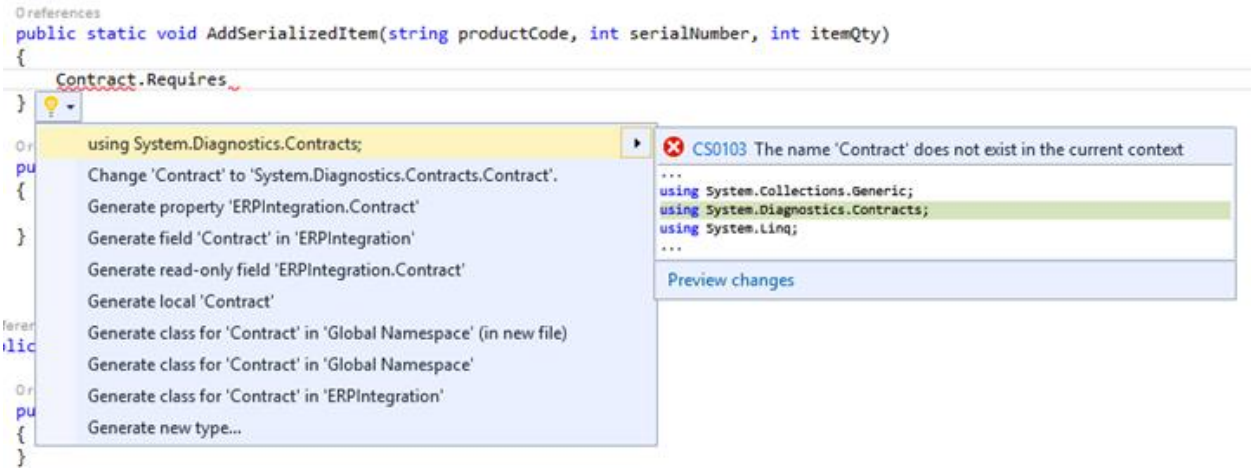

Figure 7: Visual Studio 2015 Light Bulb Feature

You will notice that Visual Studio 2015 suggests a few corrections. In Figure 7, you see that the **`using`** statement is the first suggestion, and is the fix we need to apply in this instance.

Visual Studio 2015 light bulbs

This productivity feature is new in Visual Studio 2015. Light bulbs appear automatically in the Visual Studio editor and provide error-fixing and refactoring suggestions to the developer on the current line of code being typed. Light bulbs can also manually be invoked by pressing **Ctrl+Period** on a line of code to see a list of potential fixes.

The Code Contract precondition

Consider the following code listing, which checks the value of the serial number being passed to the method. This method is used to add a serialized stock item to the inventory table of the new ERP system.

```
public static class ERPIntegration
{
    public static void AddSerializedItem(string productCode, int
serialNumber, int qty)
    {
        Contract.Requires<SerialNumberException>
            (serialNumber >= 100000001, "Invalid Serial number");
    }

}

public class SerialNumberException : Exception
{
    public SerialNumberException()
    {
    }

    public SerialNumberException(string message)
        : base(message)
    {
    }

    public SerialNumberException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```

Code Listing 2: Code Contract Precondition

The **Contract.Requires** statement denotes a precondition. Preconditions are the first statements in the method body. In the previous code listing, the precondition checks the value of the serial number passed to the method and determines if it is a valid serial number.

Now let us take a closer look at the actual Code Contract. The Code Contract is put together as follows: **Contract.Requires<TException>(bool condition, string errorMessage) where TException : Exception**. The Boolean condition is validated, and if it fails, the Code Contract will throw an exception with the message provided. Note that I have added a custom exception to the contract called **SerialNumberException**. It is derived from the **Exception** class, but you can add any applicable exception here to suit your requirements.

The format of the Code Contract in the preceding code listing is but one implementation. The following list illustrates the valid syntax for the Code Contract preconditions:

- **Contract.Requires(bool condition)**
- **Contract.Requires(bool condition, string errorMessage)**
- **Contract.Requires<TException>(bool condition)**
- **Contract.Requires<TException>(bool condition, string errorMessage)**

Personally, I prefer the combination of a specified exception class and the user-defined error message thrown if the condition expressed in the Code Contract fails. This provides the developer with a very powerful mechanism to ensure that the values passed to the method are valid. It also frees the developer from having to defensively code for every eventuality that might break the system. Knowing that the method under contract will always answer to the predetermined rules you specified gives you more peace of mind as a developer.

The Code Contract precondition in action

To illustrate the failure of the precondition defined in the Code Contract, I hard-coded an invalid serial number in the method call. Being a console application, the call to the method is wrapped in a **try**/**catch** statement that outputs the error to the console. Consider the following code listing.

```
static void Main(string[] args)
{
    try
    {
        ERPIntegration.AddSerializedItem("BC32WL", 70012, 1);
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 3: The Code Contract Precondition in Action

When the console application is run, the exception is thrown and displayed in the console window. When using a Code Contract that has a custom exception, such as the one shown in Code Listing 2, you must set the Code Contracts Assembly Mode property to **Standard Contract Requires**.

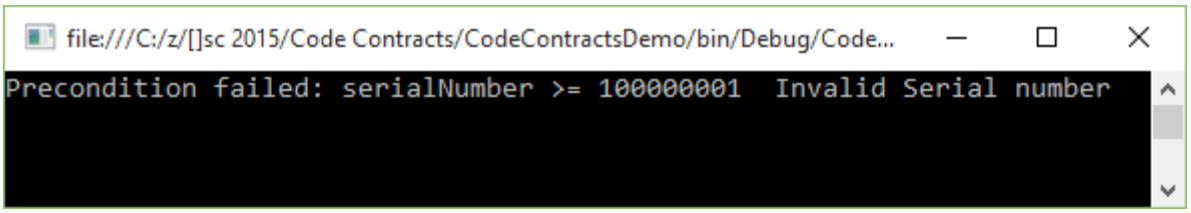

Figure 8: Precondition Failed

This preceding example is a rather simple way to illustrate the power of Code Contracts. In a production system, however, one would obviously not have hard-coded values in the method call. The possibility that these values would come from a database or from user input is very high. As we all know, all input (user or data store) is something beyond our control, and we should treat it as such. Code Contracts mitigate the negative effects of bad data.

While the preceding example is functional, I want more from my Code Contract. I do not want to run my application and then realize that there are sections of code that cause my contracts to fail. Personally, I would like to see any failures at the moment I build my project. This project is small enough, so I can be a bit cavalier with the Code Contract properties as specified in the Project Properties window.

Fail build on warnings

Return to the Code Contracts property page, and if you haven't already done so, enable **Perform Static Contract Checking**. Next, you need to clear the **Check in background** option. Then you will see that the **Fail build on warnings** option is enabled. Select this option.

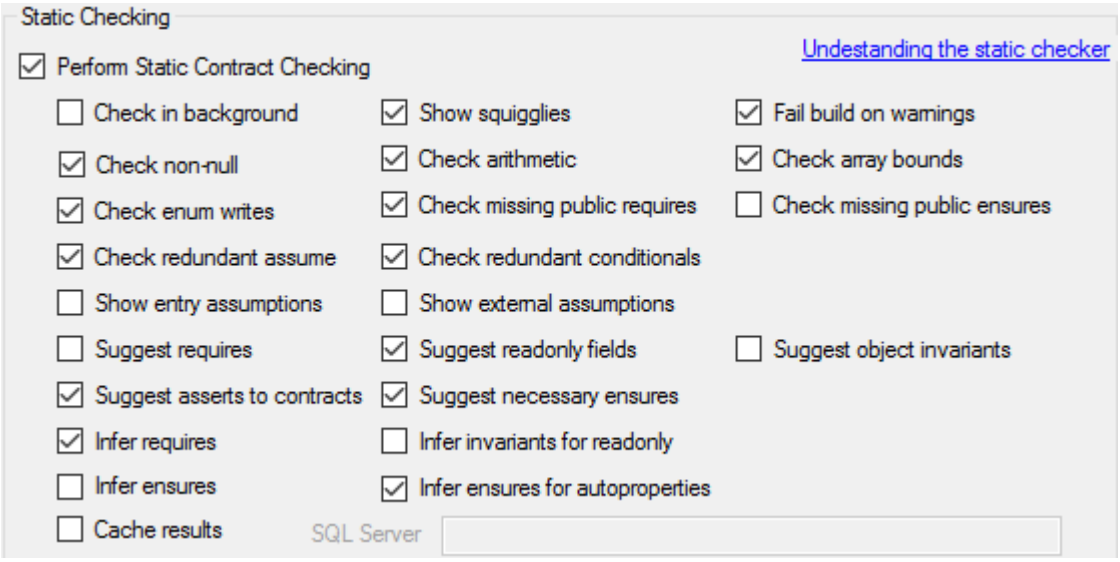

Figure 9: Enable Fail Build on Warnings

Go ahead and build your project. Static Contract Checking kicks in immediately, and because your Code Contract does not pass, your build will fail, as shown in Figure 10. There is, however, a problem. Your **Error List** is empty even though your build failed. Because we only have a single Code Contract, we know where to look. This, however, isn't feasible when we have several code contracts.

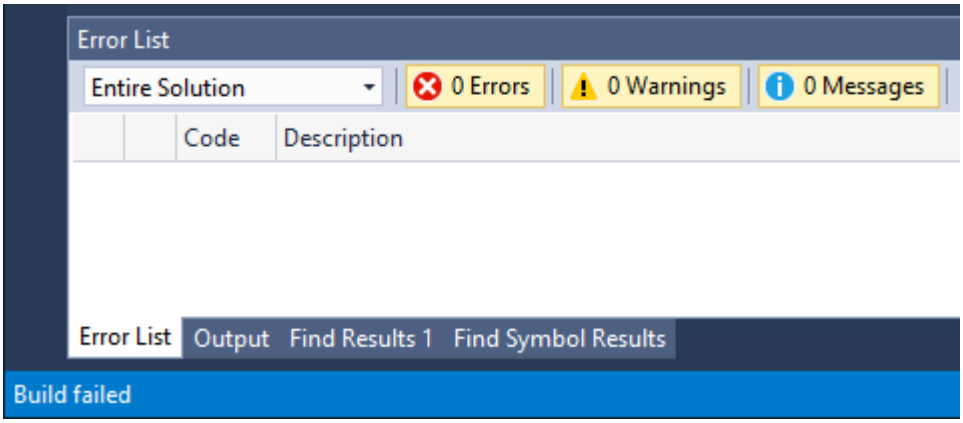

Figure 10: Visual Studio 2015 Build Failed

So what went wrong? To find the issue causing the build failure, head over to the **Output** window. If you don't see the Output window, press **Ctrl+W, O** to bring it into view. You will then see the results of the build and the Code Contract that failed.

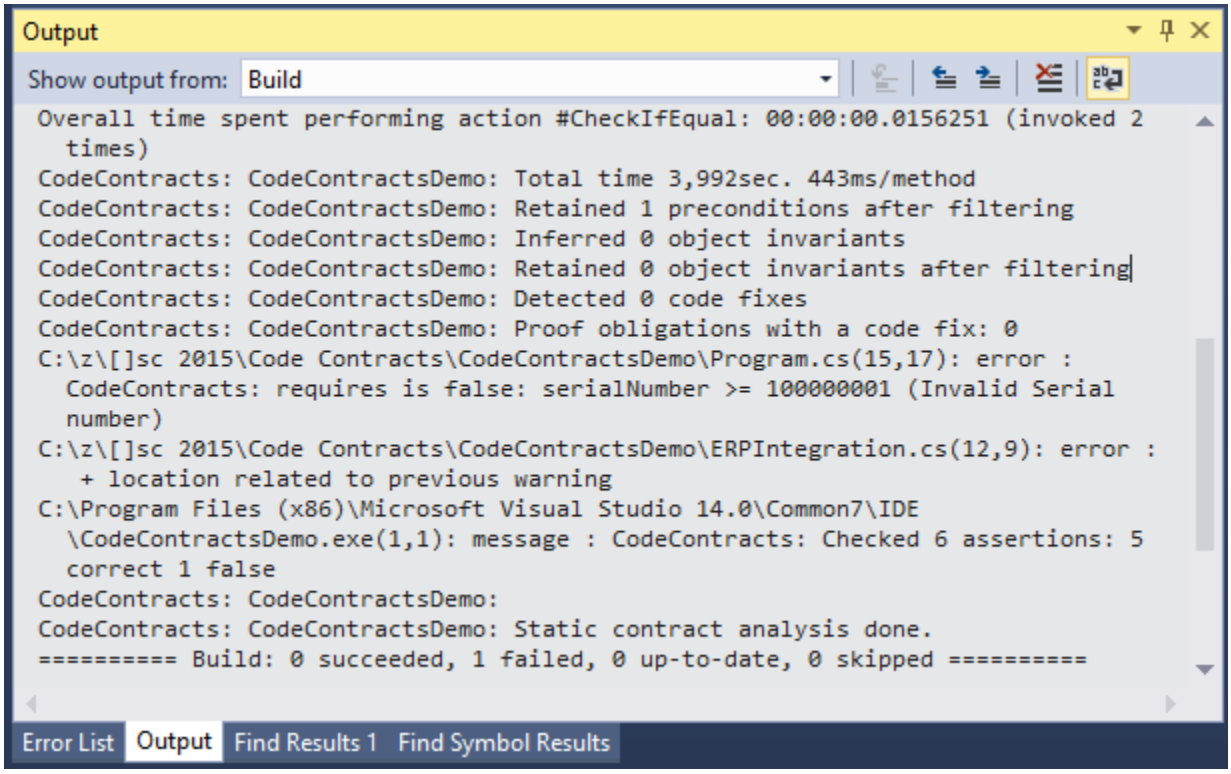

Figure 11: Visual Studio Output Window

With the current version 1.9.10714.2 of Code Contracts, the build errors not being output to the **Error List** has been identified as a bug in the release for Visual Studio 2015. This bug has been fixed and will be included in the next release of Code Contracts. For more information on this issue relating to Code Contracts for Visual Studio 2015, have a look at pull request 166 and issue 137 on GitHub. If you want to view all the open issues regarding Code Contracts, have a look at the Issues on the Code Contracts GitHub page.

Code Contracts is the result of a handful of dedicated individuals who continually work to improve the current version. It's up to users like us to report issues we come across, and in doing so, build a healthy community of influencers and a more stable product. Lastly, since it's open source, you can contribute to Code Contracts and be a part of this incredible Visual Studio extension.

The Code Contract postcondition

The Code Contract postcondition is the method under contract's way of guaranteeing to the calling code that it will return a specific result if all the preconditions are met. Let us go back to the analogy of a bank and a loan for which a contract is drawn up. The bank ensures that the loan will have a fixed interest rate, but only if the repayments are made on time and for the amount due. The payments made by you can be equated to the preconditions of the Code Contract. The fact that the bank ensures a fixed interest rate can be equated to the postcondition.

It is therefore no surprise that the syntax of Code Contract postconditions use the **`Ensures`** method. Another interesting point to take note of is that while the postcondition validates the result of the method under contract, it must appear immediately after the preconditions in the method. Consider the following code sample:

```
public static Warehouse AddSerializedItem(string productCode, int
serialNumber, int qty)
{
    Contract.Requires<SerialNumberException>
        (serialNumber >= 100000001, "Invalid Serial number");
    Contract.Ensures(Contract.Result<Warehouse>() != null);

    ProductCode = productCode;
    SerialNumber = serialNumber;
    Quantity = qty;

    return CreateItem();
}
```

Code Listing 4: Contract Postcondition

As you can see from the previous code listing, I have expanded our method slightly. I have added the postcondition that tells the calling code that the return value of this method will be of type **`Warehouse`** and that it will not be null. The calling code therefore does not need to check if

the object being returned to it is valid or not. The contract specifies that it will return a **Warehouse** object to it.

```
Contract.Ensures(Contract.Result<Warehouse>() != null);
```

Code Listing 5: The Code Contract Ensures a Result

In the real-world example, this basically means that the created stock item will be issued to a specific warehouse based on a certain condition (the product code). The product code identifies the product as a fast mover, raw material, finished good, etc., and has to be issued to the correct warehouse upon creation. Our **AddSerializedItem** method under contract tells the calling code that it ensures the result of this warehouse issue will be stored in the **Warehouse** object. If anything goes wrong, the product code will be issued to a default warehouse. Users of the ERP system can inspect items stored in the default warehouse and manually issue the product codes to the correct warehouse at a later stage.

Figure 12: Code Contract Postcondition Result

I have included the **CreateItem()** method's code in the following code listing. While this code sample is shown merely to explain a concept, the logic is sound. The **switch** statement will have a fixed set of cases it examines, as shown in Code Listing 5. It would not, however, have hard-coded values for the **Warehouse** object it returns. These would be read from a database or other object as the code interacts with the ERP to create the product code entry and issue it to a warehouse. What the code logic does ensure is that the method will always return a **Warehouse** object, and that object will always be a valid warehouse in the system.

```
private static Warehouse CreateItem()
{
    // Add Stocked Item code goes here
    Warehouse IssuedToWarehouse = new Warehouse();
    switch (ProductCode.Substring(0,1))
    {
        case "A":
            IssuedToWarehouse.Code = "FM";
```

```
            IssuedToWarehouse.Name = "Fast movers";
            IssuedToWarehouse.Bin = "A";
            IssuedToWarehouse.BinReorderLevel = 10000;
            IssuedToWarehouse.LastStockTake = Convert.ToDateTime("2015-
09-01");
            break;
        case "B":
            IssuedToWarehouse.Code = "FG";
            IssuedToWarehouse.Name = "Finished Goods";
            IssuedToWarehouse.Bin = "B";
            IssuedToWarehouse.BinReorderLevel = 500;
            IssuedToWarehouse.LastStockTake = Convert.ToDateTime("2015-
09-04");
            break;
        case "C":
            IssuedToWarehouse.Code = "RM";
            IssuedToWarehouse.Name = "Raw Materials";
            IssuedToWarehouse.Bin = "AD";
            IssuedToWarehouse.BinReorderLevel = 7500;
            IssuedToWarehouse.LastStockTake = Convert.ToDateTime("2015-
09-02");
            break;
        default:
            IssuedToWarehouse.Code = "GS";
            IssuedToWarehouse.Name = "General Stock";
            IssuedToWarehouse.Bin = "SS";
            IssuedToWarehouse.BinReorderLevel = 5000;
            IssuedToWarehouse.LastStockTake = Convert.ToDateTime("2015-
09-09");
            break;
    }
    return IssuedToWarehouse;
}
```

Code Listing 6: Warehouse Issue Logic

You can see how postconditions can be used in Code Contracts to create highly robust code. Think of a developer working in a distributed team. Being able to work from anywhere in the world has many advantages and disadvantages (depending on your point of view). The absence of personal, one-to-one communication can be construed as a distinct disadvantage. Not being able to sit in a boardroom where developers can flesh out a problem together remains a challenge. Your brain can subconsciously infer feelings, meanings, and viewpoints based on the multitude of other signals (such as body language, eye movement, or breathing) that we tend to give out.

Communication is more than just the act of speaking. Therefore, as developers, we need to become sharper and heighten our effectiveness in other areas. A great place to start is at the code. With Code Contracts, we are able to defensively preempt certain conditions that may occur. Just because the specification doesn't explicitly mention that a null condition would break the integration, that doesn't mean we don't need to defend against it.

The preceding example shows how developers can bullet-proof code against certain issues that logically could adversely affect the integration.

The Code Contract invariant

Code Contracts allow for the verification of a class' internal state. It does this via the use of Code Contract invariants. As the name suggests, an invariant is something that can never change. It will always be as it is specified in the class under contract.

We now know that our **`AddSerializedItem`** method must be supplied with a valid serial number. Valid serial numbers fall within a specific range.

```
Contract.Requires<SerialNumberException>
        (serialNumber >= 100000001, "Invalid Serial number");
```

Code Listing 7: Contract Requires Condition

We also know that the method under contract guarantees the calling code that a non-null **`Warehouse`** object will be returned when the valid serial number precondition is met.

```
Contract.Ensures(Contract.Result<Warehouse>() != null);
```

Code Listing 8: Contract Ensures Condition

Let us now assume that additional logic has to be added that needs to check the validity of the production date. This date can be in the future as well, so this needs to be supplied by an external data store, user entry, or lookup.

This is quite easy, and accomplished by adding a new private method to the class that has the **`[ContractInvariantMethod]`** attribute applied to it. If we had to add this check on the **`Warehouse`** class, we would need to add the following code to it.

```
[ContractInvariantMethod]
private void Invariants()
{
    Contract.Invariant(this.ProductionYear >= 0);
    Contract.Invariant(this.ProductionMonth >= 0);
    Contract.Invariant(this.ProductionMonth <= 12);
    Contract.Invariant(this.ProductionDay >= 0);
    Contract.Invariant(this.ProductionDay <= 30);
}
```

Code Listing 9: Contract Invariant Method

This tells the Code Contracts that the following properties for the production date have to fall within the following ranges. None of them can be zero, the number of months can't be greater

than **12**, and the days can have a maximum value of **30** (assume we're working with 30-day months).

Usually, you can call the contract invariant method anything you like, but many prefer to call it **ObjectInvariant**. Another point to note regarding the preceding code listing is that the method must have a **void** return type and be scoped as **private** or **protected**.

Code Contract invariant methods allow us to specify the state of a class that is not allowed to change. The code is short and easy to understand and implement.

Other Code Contract methods

Code Contracts also contain various other methods for use in your code. Let us have a look at these examples in the following sections.

Contract Assert and Assume

Some of you might be wondering what the difference is between **Debug.Assert** and **Contract.Assert** when used in your code. **Debug.Assert** is only executed when your code is compiled in **Debug** mode. The **Contract.Assert** method, however, is executed when you debug your code in **Debug** or **Release** mode.

The **Assert** method in Code Contracts can also easily be confused with the **Contract.Requires** method. As we saw in previous code listings, the **Requires** method is a precondition and must always be called at the start of a specific method. This is because **Contract.Requires** contains information regarding the method it resides in. **Contract.Assert** and **Contract.Assume**, on the other hand, are specific to a certain bit of code at some point in time within the method under contract.

So when do we use which method? With **Assert**, the static checker runs and will try to prove the assertion at that specific line of code. **Assume** will let the static checker simply assume that whatever the check is it needs to prove, is true. So why have both? Consider the next code listing.

```
public void CompleteBinPreparation(int quantityRequired)
{
    QuantityRequired = quantityRequired;
    int available = BinQtyAvailable();
    Contract.Assert(QuantityRequired <= available, "Quantity required
exceeds available bin quantity");
}

public int BinQtyAvailable()
{
    MaxBinQuantity = 75;
    CurrentBinQuantity = 50;
```

```
    int QtyAvailable = MaxBinQuantity - CurrentBinQuantity;
    return QtyAvailable;
}
```

Code Listing 10: Assert Static Check

The code in Code Listing 10 checks to see if the **`Bin`** containing the parts has enough space to contain the required quantity. With the **`Assert`** method, we are letting the static checker inspect the value of the **`QuantityRequired`** variable. If we had to pass through a value of **77** for **`QuantityRequired`**, we would see the static checker emit a warning and fail the build (remember, we turned on **Fail build on warnings** in the Code Contracts property page).

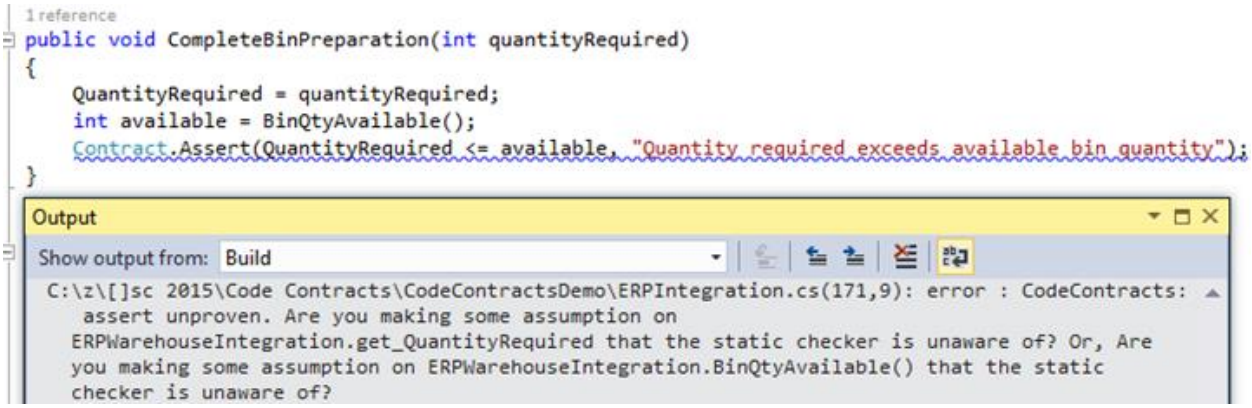

Figure 13: Assert Failed

If we had to modify the code in Code Listing 10 to contain an **`Assume`**, the output would be quite different indeed. Consider the code in the following listing.

```
public void CompleteBinPreparation(int quantityRequired)
{
    QuantityRequired = quantityRequired;
    int available = BinQtyAvailable();
    Contract.Assume(QuantityRequired <= available, "Quantity required
exceeds available bin quantity");
}

public int BinQtyAvailable()
{
    MaxBinQuantity = 75;
    CurrentBinQuantity = 50;
    int QtyAvailable = MaxBinQuantity - CurrentBinQuantity;
    return QtyAvailable;
}
```

Code Listing 11: Assume Static Check

```
1 reference
public void CompleteBinPreparation(int quantityRequired)
{
    QuantityRequired = quantityRequired;
    int available = BinQtyAvailable();
    Contract.Assume(QuantityRequired <= available, "Quantity required exceeds available bin quantity");
}
```

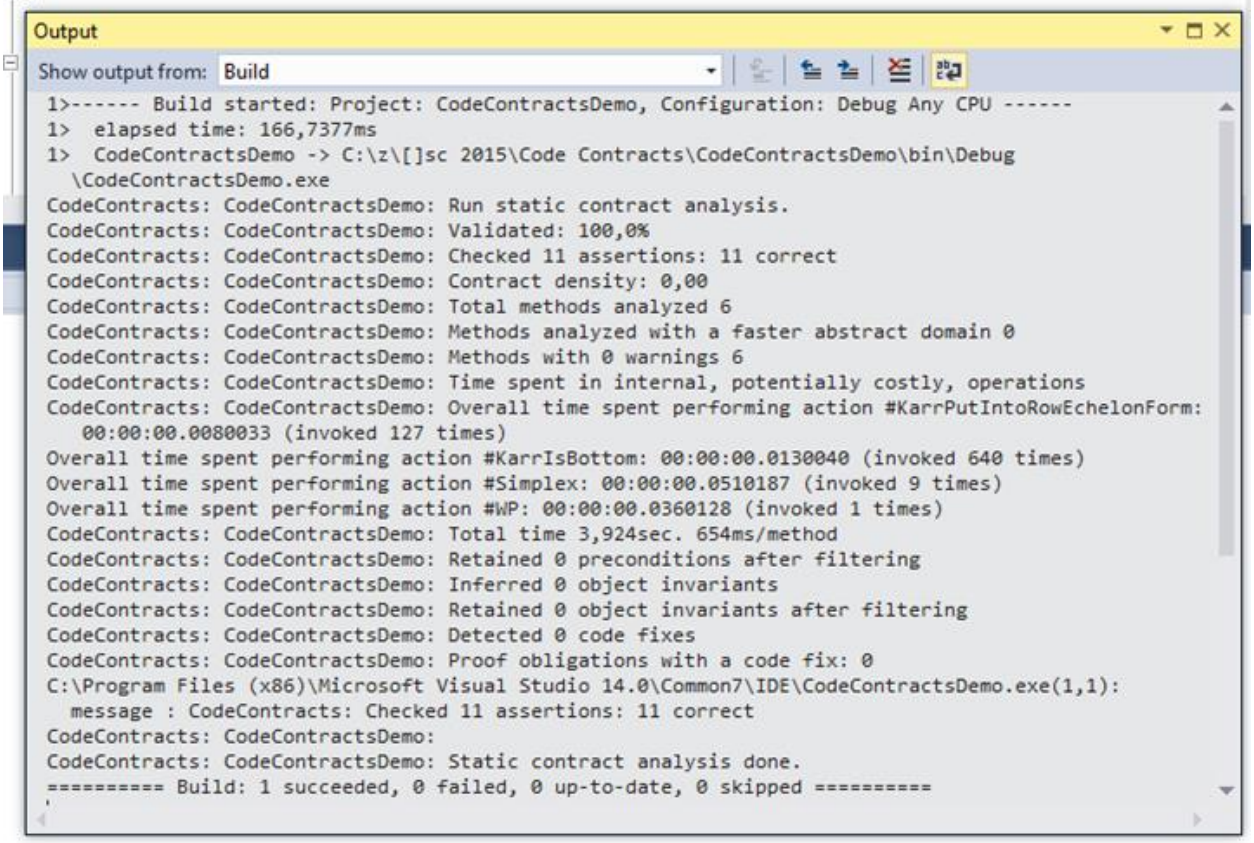

Figure 14: Assume Passed

When we use **`Contract.Assume`**, we are telling the static checker that it needs to assume that the condition under contract is true. Why would we want to do this? Well, the previous code in Code Listing 11 was calling a method we had control over. We could logically add a **`Contract.Ensures`** to the **`BinQtyAvailable`** method to guarantee that it will conform to the Code Contract. Consider for a minute, however, that we are calling a method in another external library. We do not have control over the code contained in that DLL and the developers didn't implement Code Contracts. They do, however, guarantee that the value returned will always take the required quantity into account and return a bin with a sufficient quantity available. We can therefore tell the static checker to assume that this contract condition passes.

Note that the previous examples regarding **`Assert`** and **`Assume`** only apply to the static checker. Because we have **Perform Runtime Contract Checking** on and set to **Full**, the **`Contract.Assume`** will still fail during run-time if the condition being checked fails (the external library returns an invalid value for the available bin quantity).

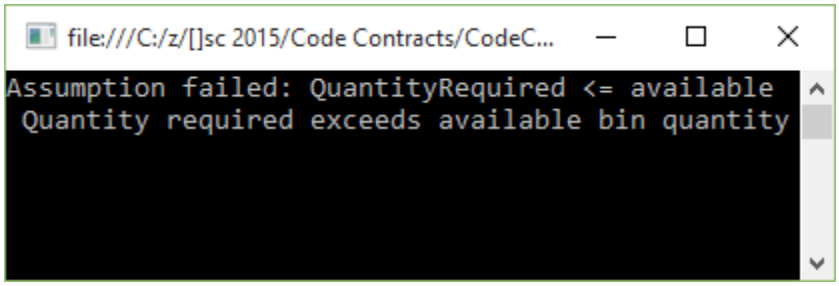

Figure 15: Assume Failed at Runtime

`Assert` and **`Assume`** make for a very powerful combination when dealing with external code and making sure that what you are expecting is returned and valid in the way you require.

Lastly, you can change the **Warning Level** in the Code Contracts property page. If you are expecting to see certain warnings in your Output window but are not, be sure to change the **Warning Level** to high.

Contract.ForAll

Before I continue to explain the **`Contract.ForAll`** logic, I have to point out that it currently fails to validate statically in Visual Studio 2015. Issue 177 has been logged on GitHub for this problem, and there is a workaround provided in the thread. To read more, go to github.com/Microsoft/CodeContracts/issues/177.

For now, I will need to disable static checking to illustrate the use of the **`Contract.ForAll`** method. To disable static checking, right-click your project in Visual Studio's Solution Explorer and select **Properties**. The Code Contracts property page will open. Click on the **Code Contracts** tab.

This is the same page we accessed previously. Under the **Static Checking** group, clear the **Perform Static Contract Checking** option.

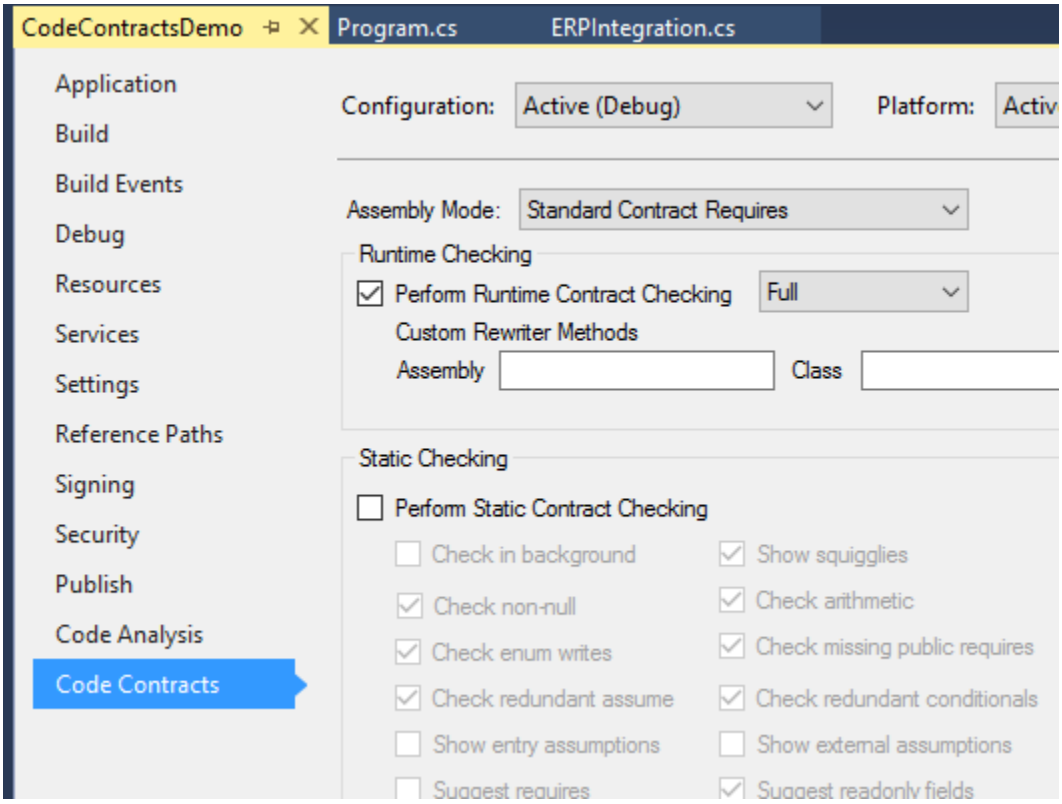

Figure 16: Disable Static Contract Checking

Once this is done, you should be able to have the Code Contract perform the required action and validate the condition you specified.

With that out of the way, let's modify our **`CompleteBinPreparation`** method. The business rules require that before a bin can be processed, the quantity needs to be greater than 5. Let's build this contract by requiring that the **`binQuantities`** array never be null. As explained previously, this is easily done by adding the **`Contract.Requires`** method.

Next, we need to add **`Contract.Assert`** to include the **`Contract.ForAll`** method. This will then use a lambda expression to check that all the quantities are greater than 5. The implementation is shown in the following code listing.

```
public void CompleteBinPreparation(int[] binQuantities)
{
    Contract.Requires(binQuantities != null);
    Contract.Assert(Contract.ForAll(binQuantities, x => x > 5),
        "Some Bins contain invalid quantities");

    // Process bin quantities
    BinCount = binQuantities.Length;
}
```

Code Listing 12: Contract.ForAll Example

If our method under contract passes the validation, it will simply return the bin count in the array. If, for example, the array contains invalid bin quantities, an exception will be displayed.

In our calling code, we will now add an array that contains a few invalid bin quantities. We can see that the quantities **4** and **3** are less than **5**, and therefore invalid.

```
static void Main(string[] args)
{
    try
    {
        int[] iBins = { 4, 3, 61, 51, 88, 55 };
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.CompleteBinPreparation(iBins);

        Console.Write("Bin Count: " + oWhi.BinCount);
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 13: Calling Contract.ForAll Method

If we run the code, the console application will throw the exception and display the error message to the user.

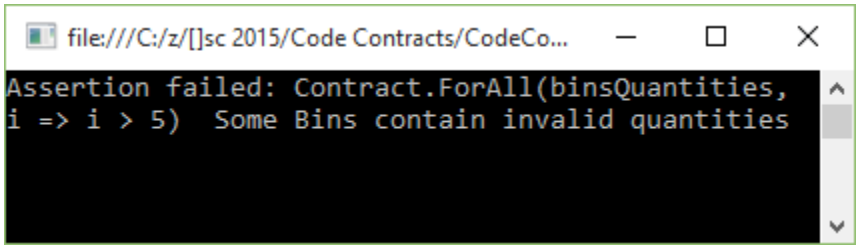

Figure 17: Invalid Bin Quantities

Let us now modify our calling code to contain only valid bin quantities in the **iBins** array. You will see that I have changed the invalid bin quantities of **4** and **3** to **32** and **19**, respectively.

If you run the application a second time, it will cause the Code Contract to pass validation and display the bin count in the console application.

```
static void Main(string[] args)
{
    try
    {
```

```
        int[] iBins = { 32, 19, 61, 51, 88, 55 };
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.CompleteBinPreparation(iBins);

        Console.Write("Bin Count: " + oWhi.BinCount);
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 14: Modified Calling Code

The console application now displays the count of valid bins, which means the Code Contract passed validation.

Figure 18: Valid Bin Quantities

As you can see, the **`Contract.ForAll`** method provides a fantastic way to check arrays of values for entries that could cause problems further down the path of execution if they contravene a business rule. The **`Contract.ForAll`** method can also be used with **`List`** collections, or any **`IEnumerable`** collection.

Contract.Exists

For a moment, let's assume a bin number that needs to be processed goes into a batch of processed bins for 30 days. The system does not know when a bin enters a process phase, and therefore needs to check each bin number before it's processed to ensure that it doesn't already exist in the process queue. Code Contracts provide a nice solution here, too.

Create a method called **`ProcessBin`** and add the **`Contract.Requires`** method to check that the parameter passed to the method is not null. Then add the **`Contract.Assert`** method to include the **`Contract.Exists`** method. This checks to see if the processed queue **`ProcessedBins`** contains the bin we want to process. The following code listing illustrates this implementation.

```
public void ProcessBin(string bin)
```

```
{
    Contract.Requires(bin != null);
    Contract.Assert(!Contract.Exists(ProcessedBins(),
        x => string.Compare(x, bin, true) == 0),
        "Bin " + bin + " already processed");

    // Process bin and add to ProcessedBins collection
}

private List<string> ProcessedBins()
{
    List<string> oBinsProcessed = new List<string>();
    oBinsProcessed.Add("A12");
    oBinsProcessed.Add("CD25");
    oBinsProcessed.Add("ZX4R");
    oBinsProcessed.Add("A11");

    return oBinsProcessed;
}
```

Code Listing 15: Contract.Exists Implementation

In the calling code, we will just use a hard-coded value for the bin number to check. Call the method under contract and pass it a bin that is already in the process queue.

```
static void Main(string[] args)
{
    try
    {
        string BinToProcess = "ZX4R";
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.ProcessBin(BinToProcess);

        Console.Write("Bin " + BinToProcess + " processed");
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 16: Calling Contract.Exists Method

The Code Contract validates if the bin exists in the process queue and displays the output to the user in the console application.

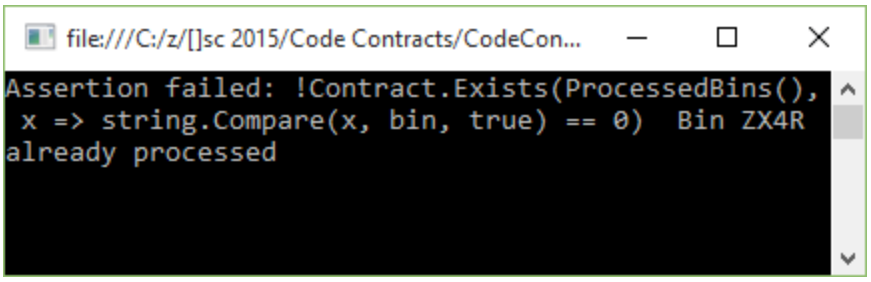

Figure 19: Bins Exist in Process Queue

Going back to our calling code, let us change the bin number to one that doesn't exist in the process queue.

```
static void Main(string[] args)
{
    try
    {
        string BinToProcess = "SSX4R";
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.ProcessBin(BinToProcess);

        Console.Write("Bin " + BinToProcess + " processed");
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 17: Modified Calling Code

If we run the application a second time, the bin will be processed and a confirmation message will be displayed to the user in the console application.

Figure 20: Bin Does Not Exist in Process Queue

`Contract.Exists` allows developers to easily verify the existence of an item based on certain business rules in the system you design.

Contract.OldValue<>

A couple of years ago, I worked on a project for a steel manufacturer. They had a very specific scrap process to follow when off-cuts were produced by cutting steel plates into specific sizes. Another project I worked on had a specific scrap workflow to process. Scrap is something many companies take very seriously, and they need to manage the process carefully.

For the next example of Code Contracts, I will use the scrap process to illustrate the use of the **`Contract.OldValue<>`** method. I have to mention, though, that the **`Contract.OldValue<>`** method can only be used in the conditional expression for the **`Ensures`** contract.

Let us assume that the steel manufacturer needs to minimize the amount of off-cuts produced by the cutting process. To do this, they use a calculation that takes the volume of steel to be cut, and calculate the amount of steel that can be used based on the cutting factor. A perfect cut means that if the volume of steel is 10 m^3, then the resulting cut of steel will also be equal to 10 m^3. All of the steel has therefore been consumed and resulted in zero off-cuts.

An imperfect cut would result in less than the original volume of steel, and this would mean that the user will need to change the cutting factor until they can ensure a near-perfect cut. Without going into more detailed specifics regarding thresholds and limits, let us assume that if anything other than a perfect cut is returned, the cutting process is not approved.

Consider the following code listing.

```
public void CutSteelNoScrap(int volumeSteel, int factor)
{
    Contract.Ensures(volumeSteel != 0, "The volume of steel can't be
zero");
    Contract.Ensures(Contract.OldValue<int>(volumeSteel)
        == CutSteel(volumeSteel, factor) + volumeSteel,
    "The factor used will result in scrap. Please modify the cutting
factor.");
    // Process steel to cut
}

private int CutSteel(int volumeToCut, int factor)
{
    return volumeToCut % factor;
}
```

Code Listing 18: Contract.OldValue<> Implementation

As you can see, I have simply used a modulus operator to simulate the existence of off-cuts based on the incorrect factor. The modulus operator simply returns the remainder of a division and is denoted by the **`%`** operator. Therefore, the following is true:

`19 % 5 = 4`

`12 % 4 = 0`

It's crude, but it's effective in illustrating the point I need to make. In the calling code, we call the **CutSteelNoScrap** method to perform the calculation and tell us if the factor used is incorrect and generates scrap.

```
static void Main(string[] args)
{
    try
    {
        int steelVolume = 4;
        int cutFactor = 3;
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.CutSteelNoScrap(steelVolume, cutFactor);

        Console.Write("Steel fully consumed by cutting process");
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 19: Calling Code

The **Contract.OldValue** method will inspect the value of the variable **volumeSteel** and see if the new value returned by the **CutSteel** method is zero. If it is zero, the sum of **volumeSteel** and **CutSteel** will equal zero. We therefore know that the factor resulted in no off-cuts.

If, however, the sum of **volumeSteel** and **CutSteel** is not equal to **volumeSteel**, then the factor resulted in off-cuts being generated by the cutting process. Running the console application with a steel volume value of 4 m^3 and a factor of 3 results in off-cuts.

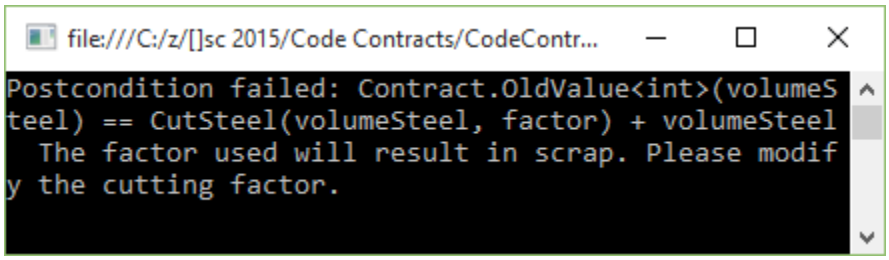

Figure 21: Contract.OldValue<> Fails

Go ahead and modify the previous code listing so that the factor is changed to a value that will not produce a remainder when used in the modulus calculation. Change it to a value of **2** and run the console application again.

```
static void Main(string[] args)
```

```
{
    try
    {
        int steelVolume = 4;
        int cutFactor = 2;
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.CutSteelNoScrap(steelVolume, cutFactor);

        Console.Write("Steel fully consumed by cutting process");
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 20: Modified Calling Code

The console application now returns a perfect cut for the given factor and steel volume.

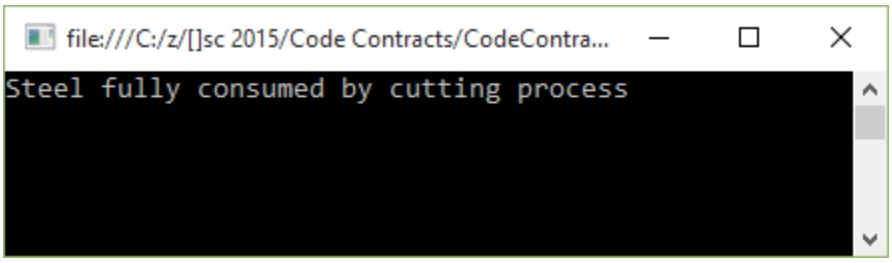

Figure 22: Contract.OldValue<> Passes

While the preceding code listing is an extremely simple way to illustrate the use of the **`Contract.OldValue<>`** method, it can be expanded to provide much more benefit in validating that methods conform to certain business rules.

Contract.Result<>

It is prudent to note that the **`Contract.Result<>`** method cannot be used in a method that has a **`void`** return type. To illustrate the use of this contract, we can reuse the previous code listings and modify them slightly.

```
public int ProductionVolumePerBin(int binVolume, int factor)
{
    Contract.Ensures(Contract.Result<int>() == binVolume,
        "The factor used will result in scrap. Please modify the cutting
factor.");
```

```
    int remainder = CutSteel(binVolume, factor);
    return binVolume - remainder;
}

private int CutSteel(int volumeToCut, int factor)
{
    return volumeToCut % factor;
}
```

Code Listing 21: Contract.Result<>

We can see that the preceding code listing tells the calling method that the method under contract will result in the cut volume always equaling the bin volume. This means that all the steel has been cut perfectly and no off-cuts were made by using the specific factor.

The calling code also doesn't differ much either.

```
static void Main(string[] args)
{
    try
    {
        int binVolume = 4;
        int cutFactor = 2;
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.ProductionVolumePerBin(binVolume, cutFactor);

        Console.Write("Steel fully consumed by cutting process");
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 22: Calling Code

If we ran the preceding code with the factor of **2** and the volume of **4**, we would get a perfect cut.

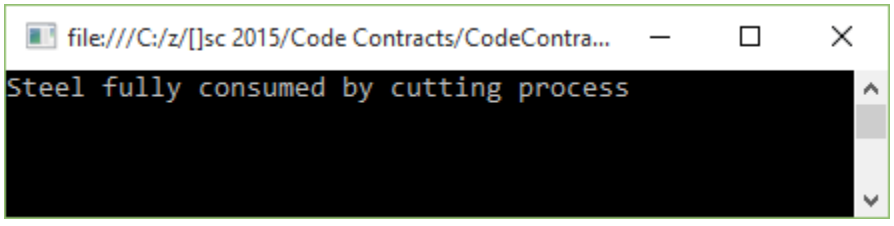

Figure 23: Contract Results in Perfect Cut

If, however, we had to modify the calling code again to a factor that would not satisfy the **`Contract.Result<>`** condition, the following would be output to the console application.

```
static void Main(string[] args)
{
    try
    {
        int binVolume = 9;
        int cutFactor = 2;
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.ProductionVolumePerBin(binVolume, cutFactor);

        Console.Write("Steel fully consumed by cutting process");
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 23: Modified Calling Code

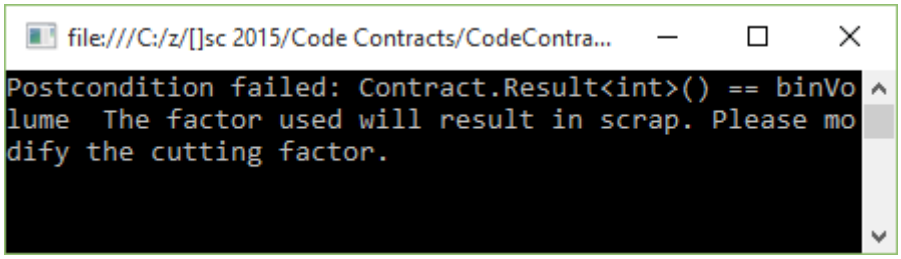

Figure 24: Contract.Result<> Failed

Code Contracts allow us to define exactly what needs to be validated in order for a specific method to pass the contract conditions. Being able to add several contract methods in a single method makes your code more robust and bug-free.

Contract.ValueAtReturn<>

Sometimes you might need to use **`out`** parameters in your methods. Code Contracts can easily be applied here, too. Let us use a slightly modified example of **`Contract.Result<>`** to illustrate this concept.

Keeping with the steel manufacturing code demo, assume that we need to ensure that all bins are filled to capacity given a specific volume of steel. Our **`out`** parameter is a bin over count. If this value is greater than zero, it means that the steel volume exceeds the maximum volume that the bins can hold. To achieve this logic, I will use the modulus operator again.

```
public void EnsureAllBinsFilled(out int binOverCount, int binVol, int
steelVol)
{
    Contract.Ensures(Contract.ValueAtReturn<int>(out binOverCount) == 0,
        "The steel volume exceeds the bin volume");

    binOverCount = steelVol % binVol;
}
```

Code Listing 24: Contract.ValueAtReturn

Our method under contract needs to specify that the **out** parameter **binOverCount** never be greater than zero. To achieve this, we need to use the **Contract.Ensures** method along with the **Contract.ValueAtReturn<>** method.

In Code Listing 24, you will notice that the **Contract.ValueAtReturn<>** references the **out** parameter **binOverCount** and specifies that it must always equal zero.

To implement the method, refer to the following code sample.

```
static void Main(string[] args)
{
    try
    {
        int steelVolume = 10;
        int binVolume = 3;
        int binWastedSpace = 0; // This must always equal zero
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.EnsureAllBinsFilled(out binWastedSpace, binVolume,
steelVolume);

        Console.Write("All bins filled");
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 25: Calling Code

The value of **binWastedSpace** will need to remain equal to zero for the method under contract to pass validation. If we ran this code, we would see that the values provided for **steelVolume** and **binVolume** cause the application to throw an exception.

The exception notifies the user that the steel volume exceeds the maximum volume of the bins provided for the operation.

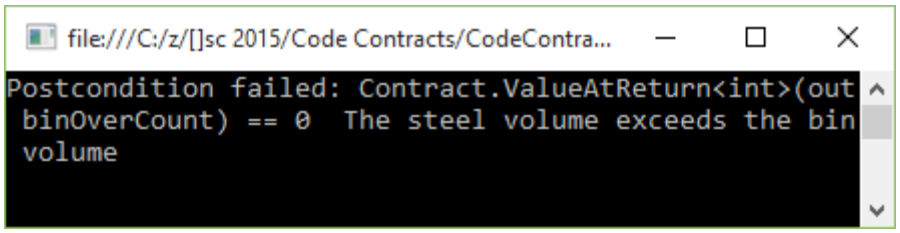

Figure 25: Contract.ValueAtReturn<> Failed

If we had to modify the calling code to provide valid values for the parameters **steelVolume** and **binVolume**, our application would pass validation.

```
static void Main(string[] args)
{
    try
    {
        int steelVolume = 10;
        int binVolume = 2;
        int binWastedSpace = 0; // This must always equal zero
        ERPWarehouseIntegration oWhi = new ERPWarehouseIntegration();
        oWhi.EnsureAllBinsFilled(out binWastedSpace, binVolume,
steelVolume);

        Console.Write("All bins filled");
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.Write(ex.Message);
        Console.ReadLine();
    }
}
```

Code Listing 26: Modified Calling Code

All I have done is change the value of the bin volume to **2**. Running the application results in a successful validation of the bin logic.

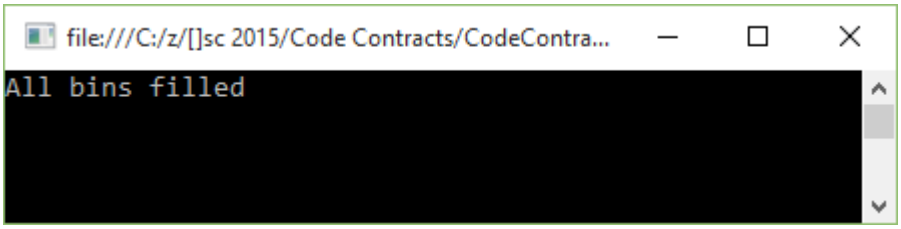

Figure 26: Contract.ValueAtReturn<> Succeeded

Without having to write extensive code logic to validate incoming and return values, we have provided a lot of validation code in the form of Code Contracts.

Chapter 3 Some Useful Tips

Using code snippets

You are likely already familiar with code snippets in Visual Studio; Code Contracts also provide this feature. When installing Code Contracts, the snippets are added. If you are using C#, you simply type the shortcut and press the Tab key twice.

In Visual Basic, the use of code snippets is slightly different. Type the shortcut and press the Tab key. There are subtle differences between the shortcut keys in C# and Visual Basic. The **`Contract.Requires`** shortcut, for example, is invoked in C# by typing **`cr`**, but in Visual Basic, it is invoked by typing **`creq`**.

C# Code Snippets

The following code listings illustrate the C# code snippet shortcuts. The shortcut used is the first line that is commented out followed by the code generated by the snippet.

```
//cr
Contract.Requires(false);
```

Code Listing 27: Contract.Requires Snippet

```
//ce
Contract.Ensures();
```

Code Listing 28: Contract.Ensures Snippet

```
//ci
Contract.Invariant(false);
```

Code Listing 29: Contract.Invariant Snippet

```
//crr
Contract.Result<int>()
```

Code Listing 30: Contract.Result Snippet

```
//co
Contract.OldValue(x)
```

Code Listing 31: Contract.OldValue Snippet

```
//cim
[ContractInvariantMethod]
```

```
[System.Diagnostics.CodeAnalysis.SuppressMessage(
    "Microsoft.Performance", "CA1822:MarkMembersAsStatic",
    Justification = "Required for code contracts.")]
private void ObjectInvariant()
{
    Contract.Invariant(false);
}
```

Code Listing 32: ContractInvariantMethod Snippet

```
//crn
Contract.Requires(arg != null);
```

Code Listing 33: Contract.Requires Not Null Snippet

```
//cen
Contract.Ensures(Contract.Result<string>() != null);
```

Code Listing 34: Contract.Ensures Contract.Result Snippet

```
//crsn
Contract.Requires(!String.IsNullOrEmpty(arg));
```

Code Listing 35: Contract.Requires String with Value Snippet

```
//cesn
Contract.Ensures(!String.IsNullOrEmpty(Contract.Result<string>()));
```

Code Listing 36: Contract.Ensures Contract.Result with String Value Snippet

```
//cca
Contract.Assert(false);
```

Code Listing 37: Contract.Assert Snippet

```
//cam
Contract.Assume(false);
```

Code Listing 38: Contract.Assume Snippet

```
//cre
Contract.Requires<ArgumentException>(false);
```

Code Listing 39: Contract.Requires with ArgumentException Snippet

```
//cren
Contract.Requires<ArgumentNullException>(arg != null, "arg");
```

Code Listing 40: Contract Requires Argument Not Null Snippet

```
//cresn
Contract.Requires<ArgumentException>(!String.IsNullOrEmpty(arg));
```

Code Listing 41: Contract.Requires String Parameter with Value

```
//cintf
#region IFoo contract binding
[ContractClass(typeof(IFooContract))]
public partial interface IFoo
{

}

[ContractClassFor(typeof(IFoo))]
abstract class IFooContract : IFoo
{
}
#endregion
```

Code Listing 42: Interface Template and Associated Contract Class Snippet

Extending code snippets

This next section is not, strictly speaking, exclusive to Code Contracts, but I would be remiss if I didn't mention the ability you have to extend your code snippets. I have always found that extending code snippets is a bit of a hassle. Luckily for developers, there are some really generous community members out there that create excellent extensions for Visual Studio. Snippet Designer is one such extension.

Before you can use it, you will need to install Snippet Designer from the **Extensions and Updates** under the **Tools** menu.

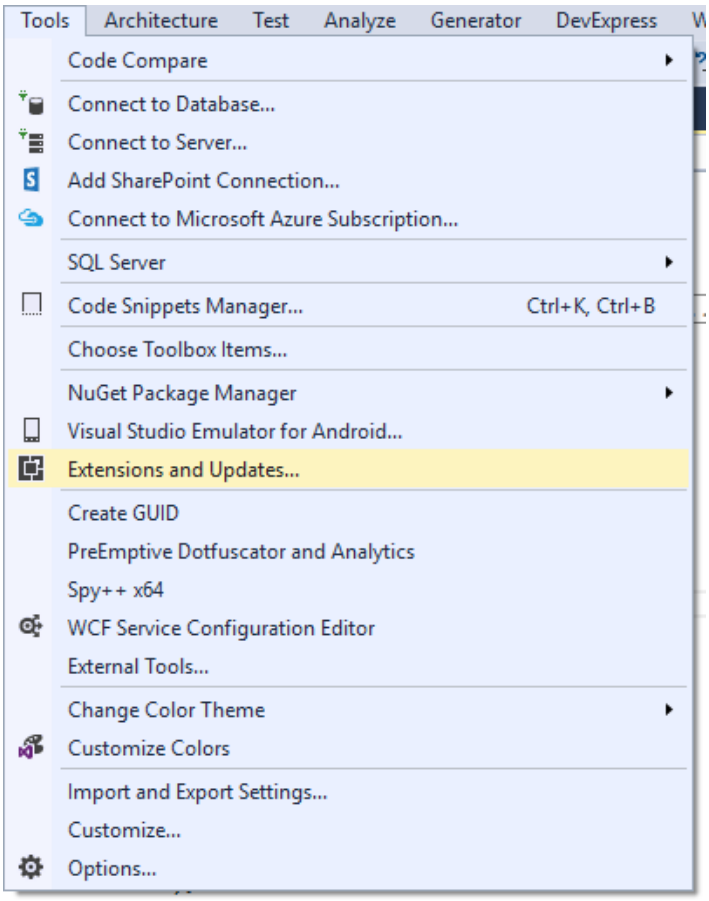

Figure 27: Visual Studio Extensions and Updates Menu Item

Once you have installed Snippet Designer, it will appear under your installed extensions.

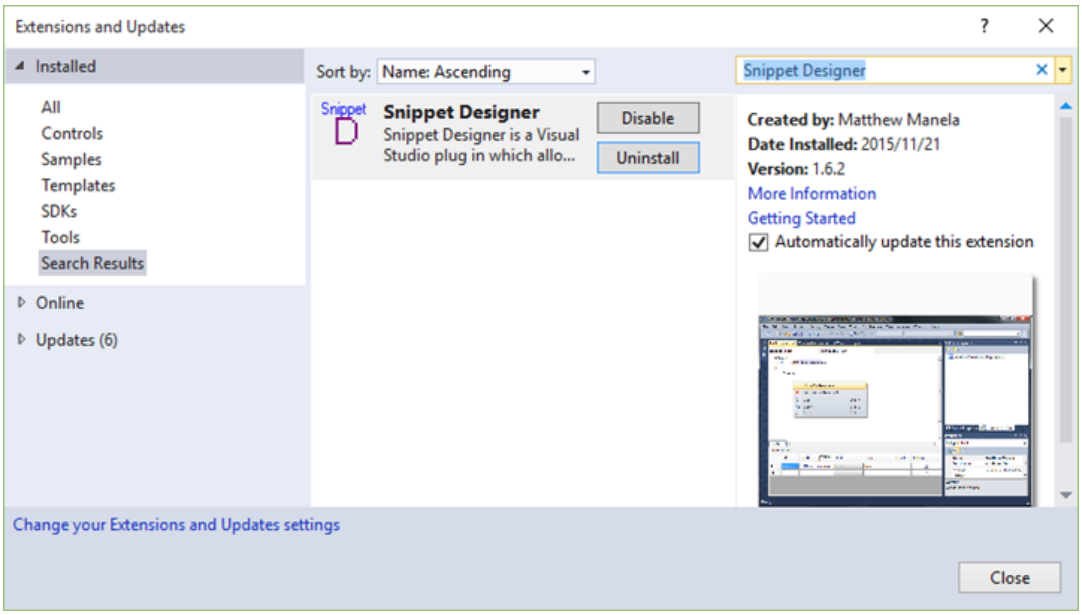

Figure 28: Snippet Designer Extension

At its most basic, Snippet Designer gives you the ability to generate code snippets on the fly. Regarding Code Contracts, there are a set few that I always want to include within my methods. I want to be able to chain them together under a single code snippet, and for this, Snippet Designer is very well suited.

Highlight the Code Contracts you want to create a snippet of, open the context menu, and click **Export as Snippet**.

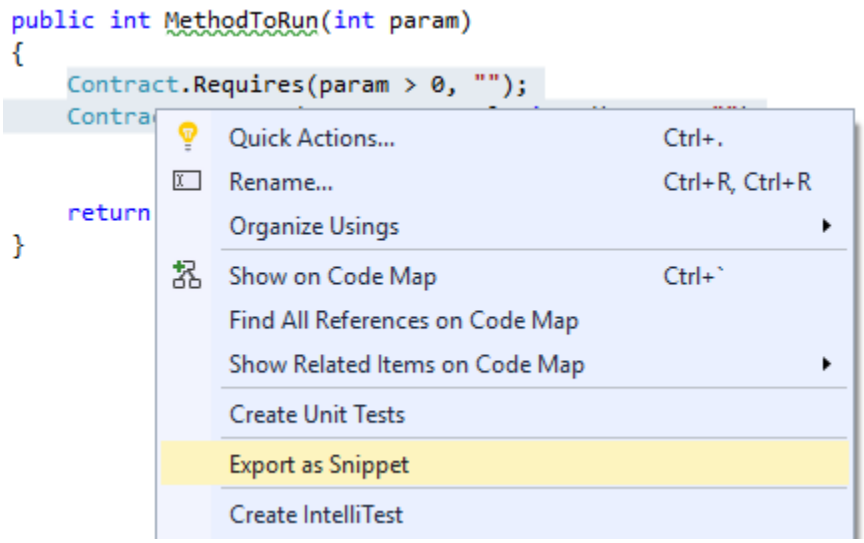

Figure 29: Export as Snippet

The Snippet Designer window is now displayed in a new tab within the Visual Studio 2015 IDE.

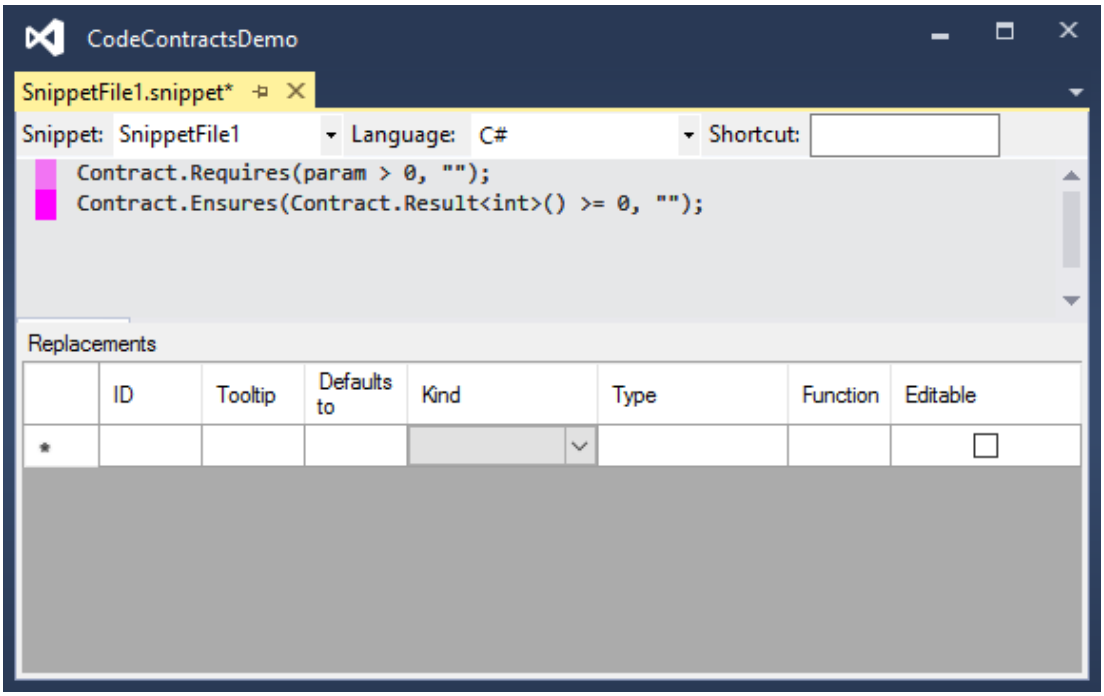

Figure 30: Snippet Designer Editor

Please note that the pink highlight is not a feature of Snippet Designer; it's another great extension called Heat Margin. This, however, is beyond the scope of this book.

In the Snippet Designer window, you can see the code you highlighted in the editor window.

Select the section of code that you want replaced at the time of generation. Then, right-click and select **Make Replacement** from the context menu.

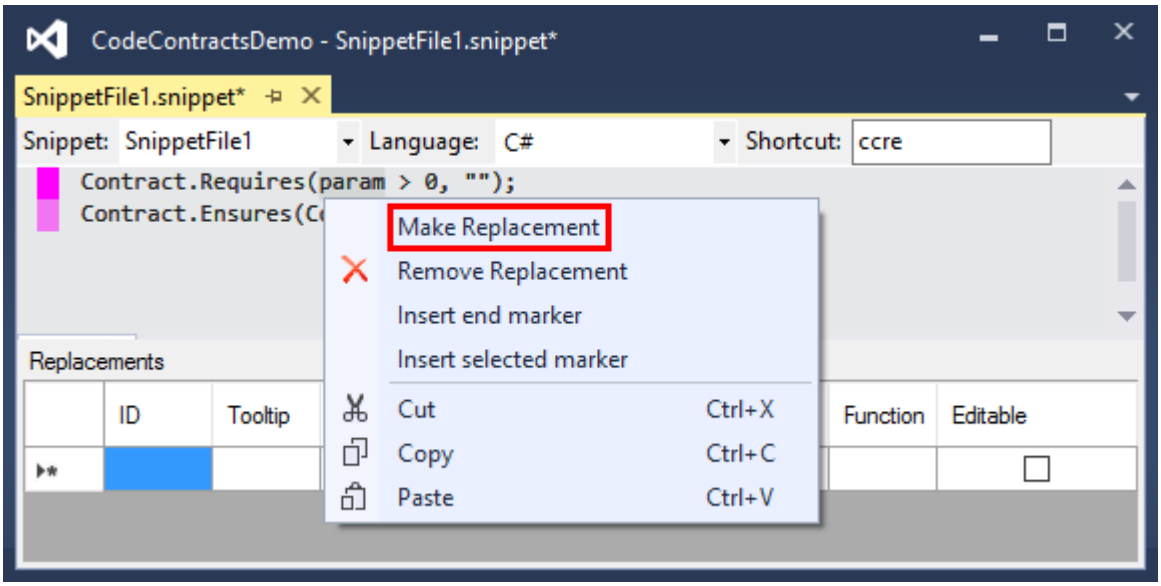

Figure 31: Make Replacement

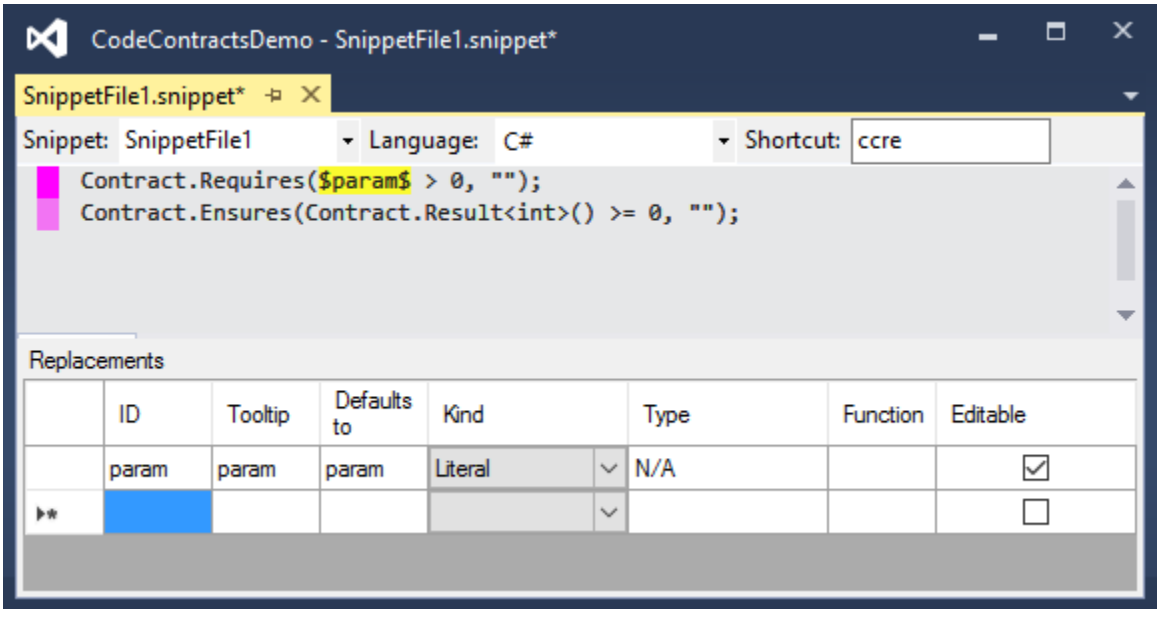

Figure 32: Completed New Snippet

You will see that the parameter is replaced by a placeholder and the properties of that placeholder are defined in the Replacements grid. All that is left to do is give the shortcut a name (I called mine **ccre** for “Code Contract Requires Ensures”) and save your code snippet.

In the code editor of Visual Studio, type the **`ccre`** shortcut and press the Tab key twice. My code snippet is inserted for me with the parameter identified as a replacement highlighted so that I can edit it immediately.

```
0 references
public int MethodToRun(int parameter)
{
    Contract.Requires(param > 0, "");
    Contract.Ensures(Contract.Result<int>() >= 0, "");


    return parameter;
}
```

Figure 33: Code Snippet Inserted

This is an extremely easy tool to use, and I have not even begun to explore the possibilities it provides developers. My focus here is simply to illustrate that code snippets can be extremely useful to developers and can be extended to suit your specific needs during development.

Code Contract documentation generation

One thing that I am sure most (if not all) developers dislike is creating documentation. It has become somewhat of the elephant in the room in many project meetings. Everybody knows that documentation is essential, and I am sure that there are a lot of developers out there that do create concise and rich documentation. This has obviously led to many tools and extensions that aim to make this process easier.

It therefore makes sense that Code Contracts can also integrate documentation of the included contracts. To enable the documentation generation, you need to open the **Build** tab of your project properties.

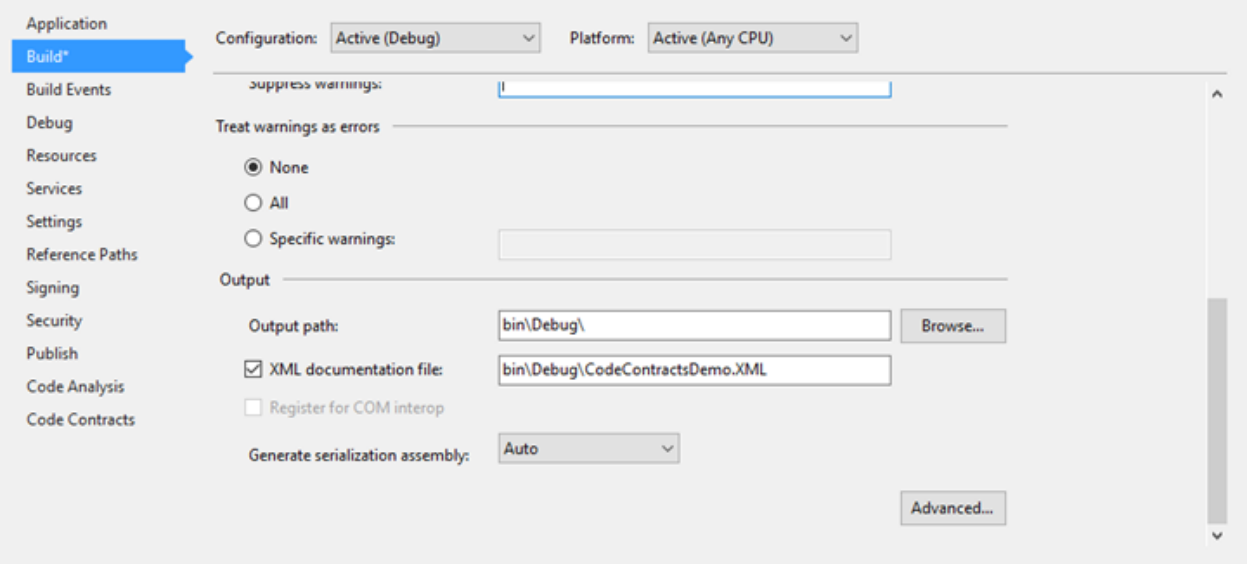

Figure 34: Enable XML Documentation File

Here, you need to select the **XML documentation file** option. If you don't enable this option, the documentation XML file will not be created. Here you can also specify an output path and a name for the documentation file.

The next option you need to modify is in the **Code Contracts** tab. If you scroll to the bottom of the Code Contracts tab, you will see that the **Contract Reference Assembly** value is not specified. Change this to **Build**.

Lastly, you must ensure that the **Emit contracts into XML doc file** option is selected. This will make the Code Contract comments part of the generated documentation file.

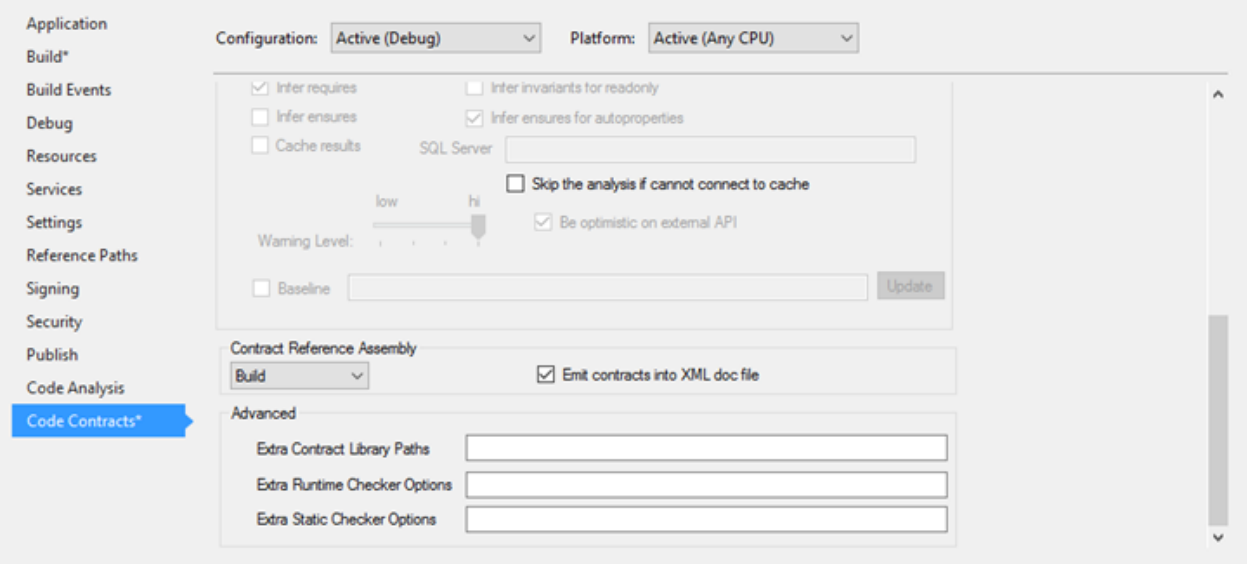

Figure 35: Enable Code Contracts Documentation

You then need to ensure that you have added good comments to your code. It's a habit you need to get into as soon as you create a new method or line of code where the logic isn't obvious. On the flip side, having too many comments in your code is also not a good idea. So when should you comment, and when should you not? A good rule of thumb is to only comment code when the logic behind the code isn't obvious.

In the following code listings, the logic is obvious in the first one (Code Listing 43), so commenting is not actually needed. In Code Listing 44, however, the use of the modulus operator should be explained in a comment because the reason for doing this isn't obvious.

```
//Bad comment: Calculate the available quantity
int QtyAvailable = MaxBinQuantity - CurrentBinQuantity;
```

Code Listing 43: Unnecessary Code Comment

```
//Good comment: Use modulus to determine if the factor produces any scrap
return volumeToCut % factor;
```

Code Listing 44: Comment Clearly Explains Logic

A well-commented method might look as follows.

```
/// <summary>
/// Calculate any remainder after the modulus operation between volume
and factor
/// </summary>
/// <param name="volumeToCut"></param>
/// <param name="factor"></param>
/// <returns>Remainder after cutting</returns>
private int CutSteel(int volumeToCut, int factor)
{
    // Use modulus to determine if the factor produces any scrap
    return volumeToCut % factor;
}
```

Code Listing 45: Well-Commented Method

After adding relevant comments, you need to build your solution. You need to be aware of any warnings you might receive during your build when the documentation file is being generated. These will usually be related to missing XML comments in your source code. It's a good idea to go ahead and fix these by adding clear and relevant code comments to your methods and properties.

After the build has completed, your XML document will be found in the path you specified when you configured your build settings. In this example, the XML document generated includes the Code Contract descriptions.

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>CodeContractsDemo</name>
  </assembly>
  <members>
    <member name="T:ERPWarehouseIntegration">
      <summary>
            ERP Warehouse Integration class to manage the cutting of steel
volume and available bin quantities
            </summary>
    </member>
    <member name="P:ERPWarehouseIntegration.MaxBinQuantity">
      <summary>
            The maximum bin quantity for bins
            </summary>
    </member>
    <member name="P:ERPWarehouseIntegration.CurrentBinQuantity">
      <summary>
            The current bin quantity available
            </summary>
    </member>
    <member
name="M:ERPWarehouseIntegration.ProductionVolumePerBin(System.Int32,System.
Int32)">
      <summary>
            Calculate the production volume of steel per bin
            </summary>
      <param name="binVolume" />
      <param name="factor" />
      <returns>Bin Volume less Remainder</returns>
      <ensures description="The factor used will result in scrap. Please
modify the cutting factor." csharp="result == binVolume" vb="result =
binVolume">result == binVolume</ensures>
    </member>
    <member
name="M:ERPWarehouseIntegration.CutSteel(System.Int32,System.Int32)">
      <summary>
            Calculate any remainder after the modulus operation between
volume and factor
            </summary>
      <param name="volumeToCut" />
      <param name="factor" />
      <returns>Remainder after cutting</returns>
    </member>
    <member name="M:ERPWarehouseIntegration.BinQtyAvailable">
      <summary>
            Ensure that a non-negative value is returned for available bin
quantity
```

```
            </summary>
      <returns>Available bin quantity</returns>
      <ensures csharp="result &gt;= 0" vb="result &gt;= 0">result &gt;=
0</ensures>
    </member>
    <member
name="M:ERPWarehouseIntegration.EnsureAllBinsFilled(System.Int32@,System.In
t32,System.Int32)">
      <summary>
            Ensure that all bins are filled and that the steel volume does
not exceed the maximum bin volume
            </summary>
      <param name="binOverCount" />
      <param name="binVol" />
      <param name="steelVol" />
      <ensures description="The steel volume exceeds the bin volume"
csharp="binOverCount == 0" vb="binOverCount = 0">binOverCount ==
0</ensures>
    </member>
  </members>
</doc>
```

Code Listing 46: Generated XML Document File

We have easily generated concise XML documentation for our code without any heavy lifting. If you comment your code regularly, your generated documentation will be up-to-date and a true reflection of the state of your code.

Creating user-friendly documents

The generated XML file is great, but it needs another step to create well-formatted, human-readable, HTML-type documentation. For this task I will use Sandcastle Help File Builder, which you can download from GitHub at github.com/EWSoftware/SHFB. During the guided installation, make a point of reading through each screen's instructions and notes, as these contain important information you need to be aware of.

I would suggest reading more tutorials on using Sandcastle Help File Builder. The tool has so much to offer that one needs to really get under the hood of this excellent tool. For the purposes of this book, however, I will not go into any further detail other than showing you how to create a basic .chm help file.

Before you use Sandcastle Help File Builder, you have to ensure that your project has a namespace. Otherwise, you will see the following error displayed on the help file build.

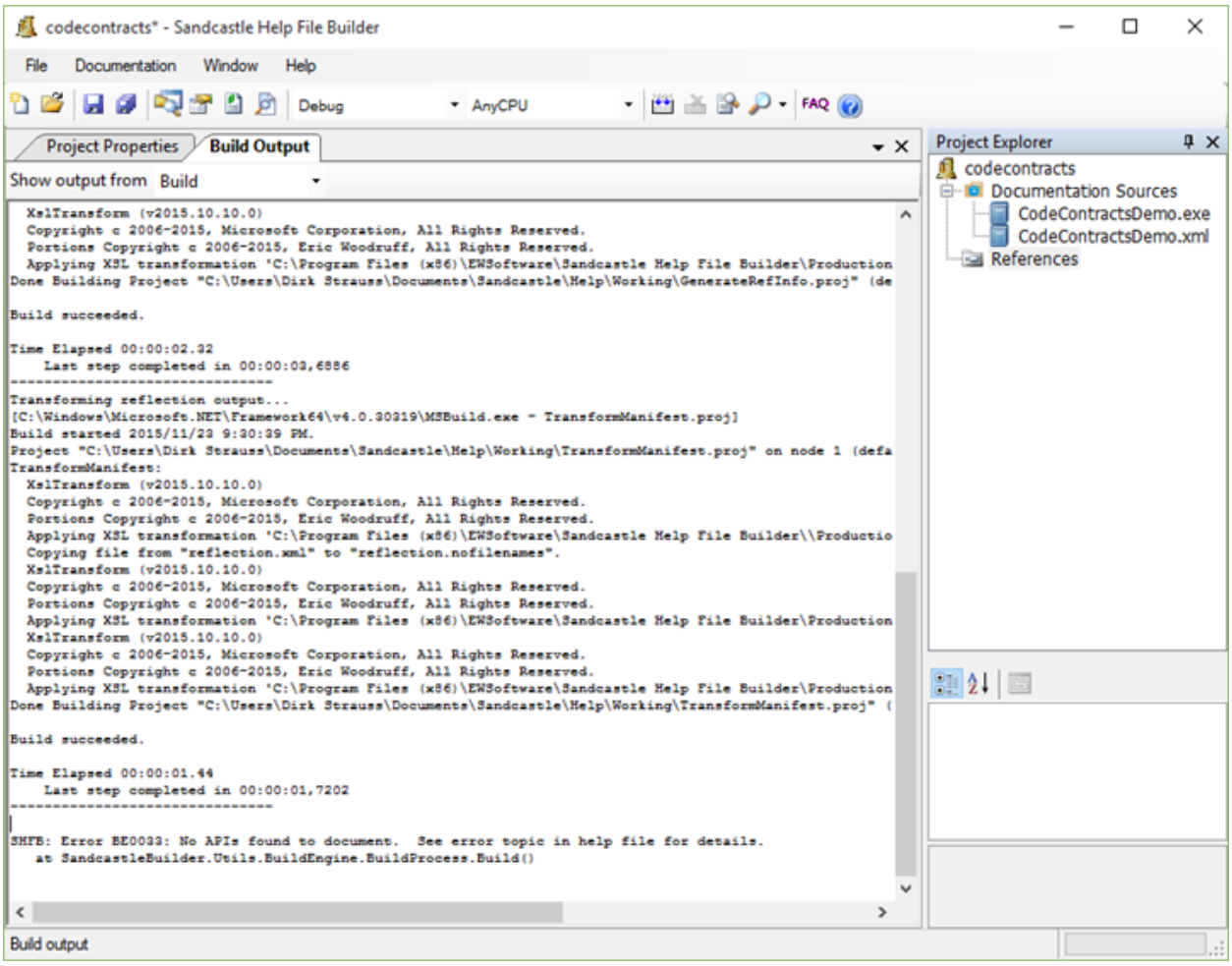

Figure 36: Build Output Error

```
SHFB: Error BE0033: No APIs found to document. See error topic in
help file for details.
   at SandcastleBuilder.Utils.BuildEngine.BuildProcess.Build()
```

Code Listing 47: Sandcastle Help File Builder Error

I simply added a namespace to my class, as illustrated in Code Listing 48.

```
namespace CodeContractsDemoProject
{
    /// <summary>
    /// ERP Warehouse Integration class to manage the cutting of steel
volume and available bin quantities
    /// </summary>
    public class ERPWarehouseIntegration
    {
```

Code Listing 48: Namespace Added to Class

Once you have done that, open **Sandcastle Help File Builder** if it's not already open. In the **Project Explorer**, right-click the **Documentation Sources** node and add the .dll or .exe where your project built to (usually your bin folder). I opted to have my XML document output to my bin folder.

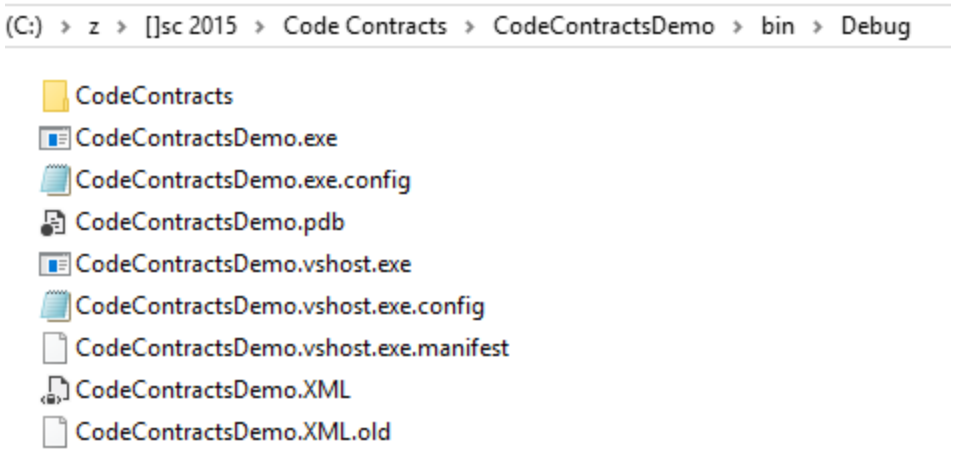

Figure 37: Project Bin Folder

Once I have selected the .exe file, the XML documentation file is also automatically picked up and added as a documentation source by Sandcastle Help File Builder. The added files will be displayed in the **Project Explorer** under the **Documentation Sources** node.

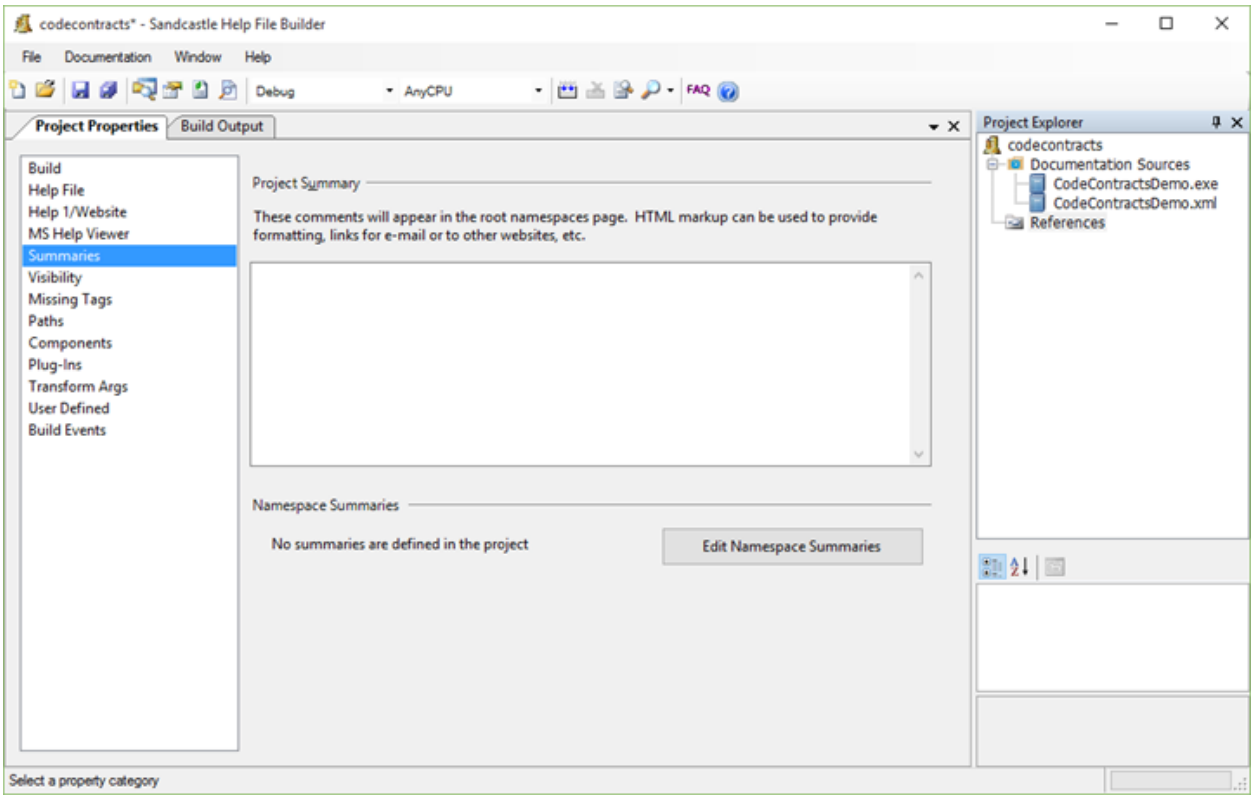

Figure 38: Documentation Sources and Summary

There are various other sections in the **Project Properties** tab that you can modify and set. One that's important to remember to configure is the **Summaries** section. You need to ensure that you have added a namespace summary comment in the **Summaries** page of the **Project Properties** tab. Click the **Edit Namespace Summaries** button to open the **Namespace Summaries** window.

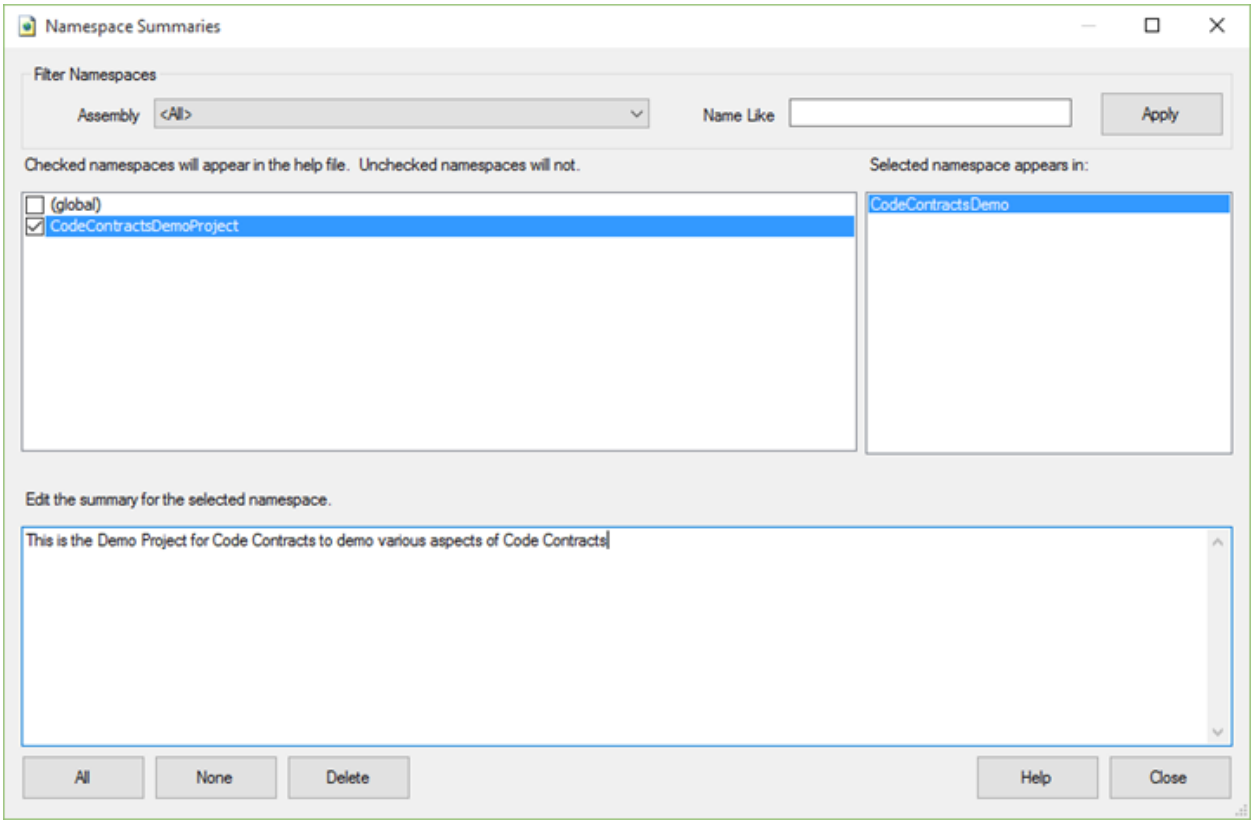

Figure 39: Namespace Summaries Window

Select the namespace you want to add a summary for and enter the summary description in the **Edit the summary for the selected namespace** text box. When you are done, click the **Close** button and save your project.

You are now ready to build your Sandcastle Help File Builder project, which will produce the help files as specified by you.

Types of help file formats supported:

- HTML Help 1 (chm)
- MS Help Viewer (mshc)
- Open XML (docx)
- Markdown (md)
- Website (HTML/ASP.NET)

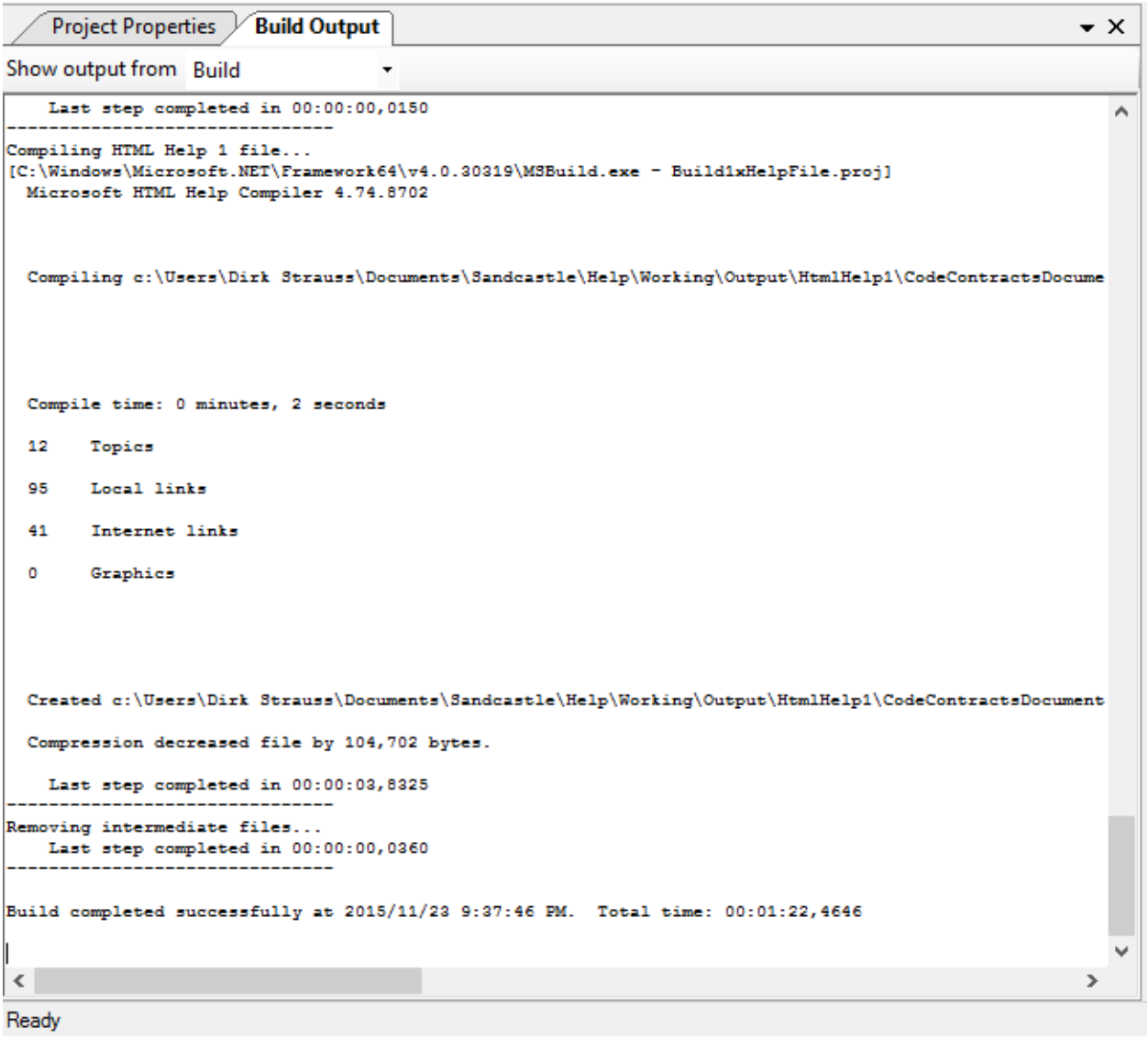

Figure 40: Sandcastle Help File Builder Successful Build

When the build has completed, navigate to the output directory for the generated help file. Mine defaulted to the path **Documents\Sandcastle\Help**.

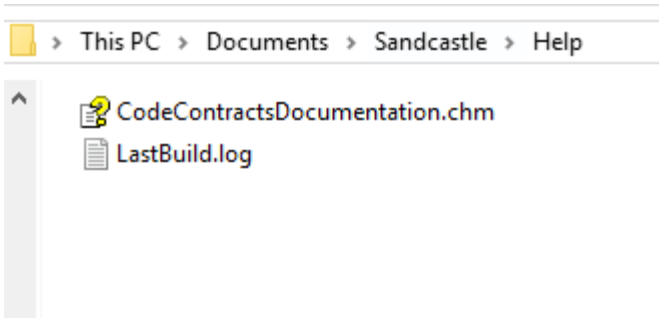

Figure 41: Help File Output Path

Opening the generated help file, you will see that the compilation took the namespace summary added in the **Summaries** section earlier and added it as the summary text for the generated help file.

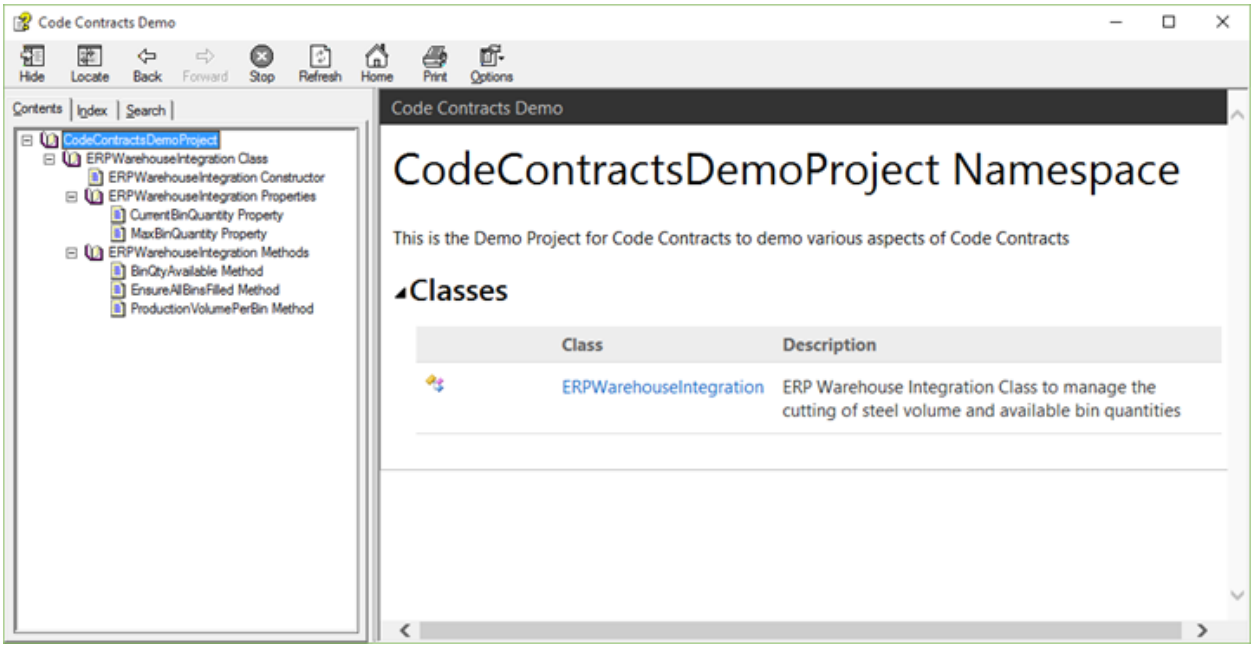

Figure 42: Generated Help File

Creating documentation from your code contract comments is really easy once you know how to do it. With superb tools such as **Sandcastle Help File Builder** and the tight integration Code Contracts offer with the output of comments into your generated XML file, developers have everything they need to create rich documentation for their projects.

Abstract classes and interfaces

Creating Code Contracts for abstract classes and interfaces needs to be approached slightly differently, as you might expect. This is because abstract classes and interfaces cannot contain method bodies. Many developers still wonder what the difference is between abstract classes and interfaces. One of the best explanations I have heard to distinguish when to use an abstract class and when to use an interface is the following.

Abstract Class

If you have many classes that can be grouped together and described by a single noun, you are most likely dealing with an abstract class. The abstract class is then named as this noun. Another very important thing to take into consideration is that these inherited classes share some sort of functionality, and that you would never create an instance of the noun (abstract class).

Think about the following example, where you have an abstract class of type **Human**. You will never instantiate just a **Human**, but rather a kind of **Human**, such as a **Female** or **Male**. Therefore, **Male** and **Female** both inherit from the abstract class **Human**. The abstract class will then implement a method **void Sleep()**, which all humans must do (shared functionality).

To implement Code Contracts on abstract classes, you need to create a separate **ContractClass** class and associate the contract class with the abstract class via the use of attributes.

```
using System.Diagnostics.Contracts;

/// <summary>
/// Human Abstract Class
/// </summary>
[ContractClass(typeof(HumanContract))]
public abstract class Human
{
    public abstract void Run(int distance);
    public abstract void Sleep(int hours);
}

/// <summary>
/// Human Contract Class
/// </summary>
[ContractClassFor(typeof(Human))]
public abstract class HumanContract : Human
{
    public override void Sleep(int hours)
    {
        Contract.Requires(hours >= 8,
            "You need more than 8 hours of sleep each night.");
    }
}
```

Code Listing 49: Abstract Class and Contract Class

The abstract class **Human** uses the attribute **[ContractClass(typeof(HumanContract))]** and the contract class **HumanContract** uses the attribute **[ContractClassFor(typeof(Human))]**.

The abstract class **Human** basically specifies that all humans will sleep and run. In the contract for the **Sleep()** method (defined in **HumanContract**), we are defining that all humans must sleep for a minimum of eight hours every night.

Let's create a **Male** class that inherits from our **Human** abstract class.

```
public class Male : Human
{
    public override void Run(int distance)
```

```
    {
        Console.Write("The distance run was " + distance);
        Console.ReadLine();
    }

    public override void Sleep(int hours)
    {
        Console.Write("The hours slept were " + hours);
        Console.ReadLine();
    }
}
```

Code Listing 50: Male Class Inheriting From Human

From the **Male** class we can see that it contains no Code Contracts at all. All it does is inherit from the **Human** abstract class.

```
namespace CodeContractsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Male oMan = new Male();
                oMan.Sleep(5);
            }
            catch (Exception ex)
            {
                Console.Write(ex.Message);
                Console.ReadLine();
            }
        }
    }
}
```

Code Listing 51: Instantiation of Male Class with Five Hours of Sleep

If we had to instantiate the **Male** class and call the **Sleep** method with five hours, our Code Contract would kick in and tell us that humans need eight hours of sleep each night.

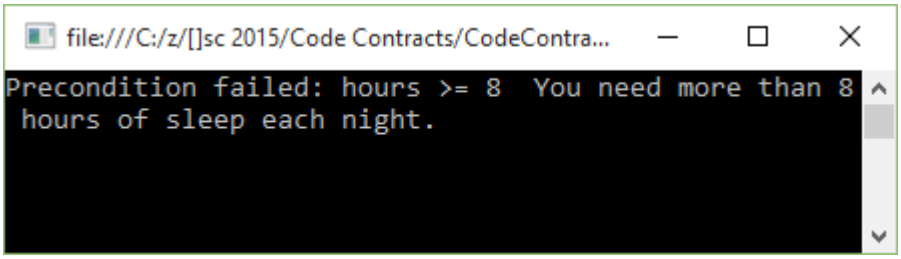

Figure 43: Male Class Sleep Method Contract Violated

Modifying our calling code to specify a nice Saturday morning snooze of nine hours, paints a different picture.

```
namespace CodeContractsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Male oMan = new Male();
                oMan.Sleep(9);
            }
            catch (Exception ex)
            {
                Console.Write(ex.Message);
                Console.ReadLine();
            }
        }
    }
}
```

Code Listing 52: Instantiation of Male Class with Nine Hours of Sleep

The Code Contract is validated and the **`Sleep()`** method passes validation.

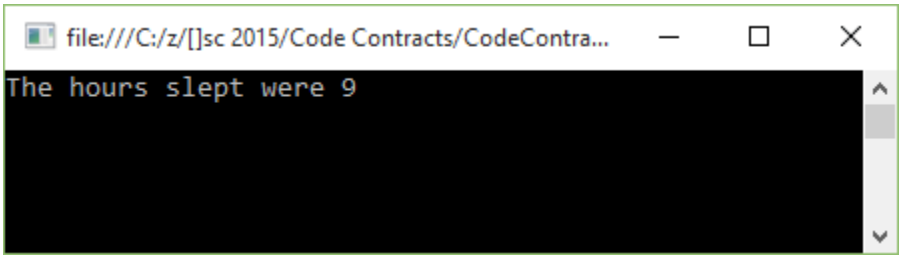

Figure 44: Male Class Sleep Method Contract Passed

Code Contracts and abstract classes work together beautifully to create a truly stable and robust code base for a team of developers.

Interfaces

Let's use the previous example of **Male** and **Female** classes. Fast forward 20 years from now, and we discover aliens on Mars. They are surprisingly similar to us, making them human, but they are neither male nor female. We now have a new type, **Alien**, which inherits from the abstract class **Human**.

Now think of interfaces as verbs. Which verb can be applied to my classes in general? All humans need to learn new things and can be taught, so now we can create an interface called **ITeachable**. We will implement the **ITeachable** interface on the **Human** abstract class, because all humans are teachable. The aliens, however, are slightly more advanced than **Male** or **Female**, and can rearrange their molecules into different structures, making aliens able to shape-shift. Therefore, we can create an interface called **IShapeShiftable**, which only applies to **Alien**.

Creating the interface is much like the abstract class. It also needs to contain a contract class, and the interface and contract class both need to point to each other via their attributes.

```
using System.Diagnostics.Contracts;

[ContractClass(typeof(IShapeShiftableContract))]
public interface IShapeShiftable
{
    void Man(int shapeDuration);
    void Woman(int shapeDuration);
    void InanimateObject(int shapeDuration);
}

[ContractClassFor(typeof(IShapeShiftable))]
abstract class IShapeShiftableContract : IShapeShiftable
{
    void IShapeShiftable.InanimateObject(int shapeDuration)
    {
        Contract.Requires(shapeDuration <= 12);
    }

    void IShapeShiftable.Man(int shapeDuration)
    {
        Contract.Requires(shapeDuration <= 4);
    }

    void IShapeShiftable.Woman(int shapeDuration)
    {
        Contract.Requires(shapeDuration <= 4);
    }
}
```

Code Listing 53: Interface Implementing Code Contract Class

We can now create contracts for the interface via the contract class **`IShapeShiftableContract`**, exactly like we did earlier for abstract classes.

Abstract classes versus interface

To conclude the examples of abstract classes and interfaces, it all boils down to where you want your implementation. If you want to share implementation among all derived classes, you will be creating an abstract class. If you need your implementation specific to a single class or several classes, but not all classes, use an interface.

Code Contracts easily cater to both.

Method purity

A book on Code Contracts would not be complete without mentioning purity. When referring to Code Contracts, what exactly does method purity mean? Well, Code Contracts have an attribute called **`[Pure]`** that you can decorate methods with. This is basically an expression of the quality of the method, and that the method can't change the state of any objects seen by callers. In other words, pure methods are only allowed to change objects created after the method has been entered.

Code Contracts require all methods called inside a contract to be pure. The reasons for this are simply:

- To avoid side effects in methods used in preconditions and postconditions.
- To make the task of the static checker easier because it can assume object states will be the same after the method is called.
- To improve design by specifically clarifying your intent that the method will not alter the state of objects after the method call has ended.

To illustrate the effect that a non-pure method has on the static checker, consider the following code listing.

```
public class DemoPurity
{
    /// <summary>
    /// Property for cutting factor
    /// </summary>
    public int CutFactor { get; private set; }

    /// <summary>
    /// Public Constructor
    /// </summary>
    /// <param name="cutFactor"></param>
    public DemoPurity(int cutFactor)
    {
        CutFactor = cutFactor;
```

```
    }

    /// <summary>
    /// Calculate the volume cut
    /// </summary>
    /// <param name="volumeSteel"></param>
    /// <param name="factorModifier"></param>
    /// <returns></returns>
    public int VolumeCut(int volumeSteel, int factorModifier)
    {
        Contract.Requires(CalculatedCutFactor(factorModifier) >= 0);

        return volumeSteel / (CutFactor * factorModifier);
    }

    /// <summary>
    /// This is not a pure method
    /// </summary>
    /// <param name="factorModifier"></param>
    /// <returns></returns>
    public int CalculatedCutFactor(int factorModifier)
    {
        CutFactor = CutFactor * factorModifier;
        return CutFactor;
    }
}
```

Code Listing 54: Method Failing Purity

The property **CutFactor** is specified in the constructor for **DemoPurity**. The method **VolumeCut** includes a required contract on the **CalculatedCutFactor** method. As you can see, the **CalculatedCutFactor** method is definitely not pure because it modifies the **CutFactor** property. It is also sloppy code. The static checker will fail on the build because the method is not pure.

```
1>------ Rebuild All started: Project: CodeContractsDemo, Configuration:
Debug Any CPU ------
1>C:\z\[]sc 2015\Code
Contracts\CodeContractsDemo\ERPIntegration.cs(197,9): warning CC1036:
Detected call to method 'DemoPurity.CalculatedCutFactor(System.Int32)'
without [Pure] in contracts of method
'DemoPurity.VolumeCut(System.Int32,System.Int32)'.
CodeContracts: CodeContractsDemo: Run static contract analysis.
C:\z\[]sc 2015\Code Contracts\CodeContractsDemo\ERPIntegration.cs(197,9):
warning CC1036: CodeContracts: Detected call to method
'DemoPurity.CalculatedCutFactor(System.Int32)' without [Pure] in
contracts of method 'DemoPurity.VolumeCut(System.Int32,System.Int32)'.
```

```
14.0\Common7\IDE\CodeContractsDemo.exe(1,1): message : CodeContracts:
Checked 15 assertions: 13 correct (2 masked)
CodeContracts: CodeContractsDemo:
CodeContracts: CodeContractsDemo: Static contract analysis done.
========== Rebuild All: 0 succeeded, 1 failed, 0 skipped ==========
```

Code Listing 55: Shortened Output Results

I have removed all the extra lines not relevant to this discussion on purity from the output text. You can see that the static checker is not happy with the **`CalculatedCutFactor`** not being pure. We can obviously go ahead and just add the **`[Pure]`** attribute to the **`CalculatedCutFactor`** method, but that would not be good programming practice. If we did this, it would result in a successful build because we have basically told the static checker that our method **`CalculatedCutFacor`** is pure. If we therefore modified our code, as shown in Code Listing 56, the static checker would assume that the method is pure.

```
public class DemoPurity
{
    /// <summary>
    /// Property for cutting factor
    /// </summary>
    public int CutFactor { get; private set; }

    /// <summary>
    /// Public Constructor
    /// </summary>
    /// <param name="cutFactor"></param>
    public DemoPurity(int cutFactor)
    {
        CutFactor = cutFactor;
    }

    /// <summary>
    /// Calculate the volume cut
    /// </summary>
    /// <param name="volumeSteel"></param>
    /// <param name="factorModifier"></param>
    /// <returns></returns>
    public int VolumeCut(int volumeSteel, int factorModifier)
    {
        Contract.Requires(CalculatedCutFactor(factorModifier) >= 0);

        return volumeSteel / (CutFactor * factorModifier);
    }

    /// <summary>
    /// This is still not a pure method
    /// </summary>
```

```
    /// <param name="factorModifier"></param>
    /// <returns></returns>
    [Pure]
    public int CalculatedCutFactor(int factorModifier)
    {
        CutFactor = CutFactor * factorModifier;
        return CutFactor;
    }
}
```

Code Listing 56: Add Pure Attribute to Method

If we had to build our project, the resulting output would indicate a successful build.

```
1>------ Rebuild All started: Project: CodeContractsDemo, Configuration:
Debug Any CPU ------
1>  elapsed time: 886,9984ms
1>  elapsed time: 177,9426ms
1>  elapsed time: 1120,0034ms
CodeContracts: CodeContractsDemo: Run static contract analysis.
1>  CodeContractsDemo -> C:\z\[]sc 2015\Code
Contracts\CodeContractsDemo\bin\Debug\CodeContractsDemo.exe
CodeContracts: CodeContractsDemo: Validated: 100,0%
CodeContracts: CodeContractsDemo: Checked 15 assertions: 13 correct (2
masked)
CodeContracts: CodeContractsDemo: Contract density: 0,98
CodeContracts: CodeContractsDemo: Total methods analyzed 6
CodeContracts: CodeContractsDemo: Methods analyzed with a faster abstract
domain 0
CodeContracts: CodeContractsDemo: Methods with 0 warnings 5
CodeContracts: CodeContractsDemo: Time spent in internal, potentially
costly, operations
CodeContracts: CodeContractsDemo: Overall time spent performing action
#KarrPutIntoRowEchelonForm: 00:00:00.0140007 (invoked 784 times)
Overall time spent performing action #KarrIsBottom: 00:00:00.0120029
(invoked 1515 times)
Overall time spent performing action #WP: 00:00:00.2000032 (invoked 5
times)
Overall time spent performing action #CheckIfEqual: 00:00:00.0080005
(invoked 29 times)
Overall time spent performing action #ArraysAssignInParallel:
00:00:00.1430061 (invoked 1 times)
Overall time spent performing action #Simplex: 00:00:00.0810009 (invoked
19 times)
CodeContracts: CodeContractsDemo: Total time 7,821sec. 1303ms/method
CodeContracts: CodeContractsDemo: Generated 2 callee assume(s)
CodeContracts: CodeContractsDemo: Retained 0 preconditions after
filtering
```

```
CodeContracts: CodeContractsDemo: Inferred 0 object invariants
CodeContracts: CodeContractsDemo: Retained 0 object invariants after
filtering
CodeContracts: CodeContractsDemo: Discovered 3 postconditions to suggest
CodeContracts: CodeContractsDemo: Retained 1 postconditions after
filtering
CodeContracts: CodeContractsDemo: Detected 1 code fixes
CodeContracts: CodeContractsDemo: Proof obligations with a code fix: 3
C:\Program Files (x86)\Microsoft Visual Studio
14.0\Common7\IDE\CodeContractsDemo.exe(1,1): message : CodeContracts:
Checked 15 assertions: 13 correct (2 masked)
CodeContracts: CodeContractsDemo:
CodeContracts: CodeContractsDemo: Static contract analysis done.
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

Code Listing 57: Successful Build

The problem here is that simply adding the **[Pure]** attribute to the **CalculatedCutFactor** method does not make it pure. This is the point I am trying to make with this section on method purity. We need to ensure that methods called inside a contract are pure because they are so by design, not because we have decorated them with the **[Pure]** attribute. Let's have a look at our **CalculatedCutFactor** method again and modify it slightly so that we are not violating the rules specified for method purity.

```
public class DemoPurity
{
    /// <summary>
    /// Property for cutting factor
    /// </summary>
    public int CutFactor { get; private set; }

    /// <summary>
    /// Public Constructor
    /// </summary>
    /// <param name="cutFactor"></param>
    public DemoPurity(int cutFactor)
    {
        CutFactor = cutFactor;
    }

    /// <summary>
    /// Calculate the volume cut
    /// </summary>
    /// <param name="volumeSteel"></param>
    /// <param name="factorModifier"></param>
    /// <returns></returns>
    public int VolumeCut(int volumeSteel, int factorModifier)
    {
```

```
        Contract.Requires(CalculatedCutFactor(factorModifier) >= 0);

        return volumeSteel / (CutFactor * factorModifier);
    }

    /// <summary>
    /// This is a pure method
    /// </summary>
    /// <param name="factorModifier"></param>
    /// <returns></returns>
    [Pure]
    public int CalculatedCutFactor(int factorModifier)
    {
        return CutFactor * factorModifier;
    }
}
```

Code Listing 58: Pure Method Marked as Pure

The **CalculatedCutFactor** method is now a pure method because it does not change the values of **CutFactor** or **factorModifier**. It can now suitably be decorated with the **[Pure]** attribute. Our **Contract.Requires** in the **VolumeCut** method can now use this pure method to check that the result of **CalculatedCutFactor** will not be zero, because this would result in a divide by zero exception on the return.

Method purity is a really good practice to follow, not only when using Code Contracts, but in every method you write.

Contract abbreviator methods

I am sure that those of you who use Code Contracts on a regular basis have run into the issue of repeating similar sets of Code Contracts across multiple methods. Code Contracts have a solution for this, and they are called **abbreviator methods**. The concept is really simple. You can see that the code in Code Listing 59 contains repeated sets of Code Contracts.

```
public class AbbreviatorDemo
{
    /// <summary>
    /// The factor for the cutting volume
    /// </summary>
    public int Factor { get; private set; }
    /// <summary>
    /// The maximum volume a bin can contain
    /// </summary>
    public int MaxVolume { get; private set; }

    /// <summary>
```

```
    /// Fill the bin with the volume of steel
    /// </summary>
    /// <param name="steelVolume"></param>
    public void FillBin(int steelVolume)
    {
        Contract.Requires(steelVolume > 0);
        Contract.Ensures(steelVolume <= this.MaxVolume);
    }

    /// <summary>
    /// Empty the bin of all steel contained
    /// </summary>
    /// <param name="steelVolume"></param>
    /// <returns></returns>
    public bool PurgeBin(int steelVolume)
    {
        Contract.Requires(steelVolume > 0);
        Contract.Ensures(steelVolume <= this.MaxVolume);
        Contract.Ensures(Contract.Result<bool>() == true);

        // Purge bin and return successful result
        return true;
    }

    /// <summary>
    /// Perform a partial bin fill
    /// </summary>
    /// <param name="steelVolume"></param>
    /// <returns></returns>
    public bool FillBinPartially(int steelVolume)
    {
        Contract.Requires(steelVolume > 0);
        Contract.Ensures(steelVolume < this.MaxVolume);
        Contract.Ensures(Contract.Result<bool>() == true);

        return true;
    }
}
```

Code Listing 59: Repeated Contracts

We can now make use of **`Abbreviator`** methods to simplify the code and reference them in multiple methods. Using **`Abbreviator`** methods, the code can be refactored as follows.

```
public class AbbreviatorDemo
{
    /// <summary>
    /// The factor for the cutting volume
```

```
    /// </summary>
    public int Factor { get; private set; }
    /// <summary>
    /// The maximum volume a bin can contain
    /// </summary>
    public int MaxVolume { get; private set; }

    /// <summary>
    /// Fill the bin with the volume of steel
    /// </summary>
    /// <param name="steelVolume"></param>
    public void FillBin(int steelVolume)
    {
        ValidSteelAndMaxVolume(steelVolume);
    }

    /// <summary>
    /// Empty the bin of all steel contained
    /// </summary>
    /// <param name="steelVolume"></param>
    /// <returns></returns>
    public bool PurgeBin(int steelVolume)
    {
        ValidSteelAndMaxVolume(steelVolume);
        EnsurePositiveResult();

        // Purge bin and return successful result
        return true;
    }

    /// <summary>
    /// Perform a partial bin fill
    /// </summary>
    /// <param name="steelVolume"></param>
    /// <returns></returns>
    public bool FillBinPartially(int steelVolume)
    {
        ValidSteelAndMaxVolume(steelVolume);
        EnsurePositiveResult();

        return true;
    }

    /// <summary>
    /// Abbreviator method for steel and max volume
    /// </summary>
    /// <param name="steelVolume"></param>
    [ContractAbbreviator]
    private void ValidSteelAndMaxVolume(int steelVolume)
```

```
    {
        Contract.Requires(steelVolume > 0);
        Contract.Ensures(steelVolume <= this.MaxVolume);
    }

    /// <summary>
    ///  Abbreviator method for successful result
    /// </summary>
    [ContractAbbreviator]
    private void EnsurePositiveResult()
    {
        Contract.Ensures(Contract.Result<bool>() == true);
    }
}
```

Code Listing 60: Using Abbreviator Methods

I have added two new methods, **`ValidSteelAndMaxVolume`** and **`EnsurePositiveResult`**, and added the **`[ContractAbbreviator]`** attribute to them. This enables me to cut down on the repeated sets of Code Contracts in my methods. It makes for easier reading and clearer functionality when looking at the methods under contract. Another point to keep in mind is that **`Abbreviator`** methods can contain calls to other **`Abbreviator`** methods if needed.

Chapter 4 Testing Code Contracts

Pex evolves into IntelliTest

Many of you might be familiar with Pex, and it probably will not come as a surprise that Pex has evolved into IntelliTest. IntelliTest is an integrated feature in Visual Studio Enterprise 2015. Code Contracts and IntelliTest integrate well, and it is really easy to get started with IntelliTest.

Getting started: Create IntelliTest

As mentioned previously, IntelliTest is integrated into Visual Studio Enterprise 2015. If you want IntelliTest integration in other versions of Visual Studio 2015, you can make your voice heard at the Visual Studio 2015 UserVoice site. The folks over at Microsoft in the Visual Studio team do keep a careful eye on this site—your vote will never be cast in vain.

Getting started with IntelliTest is really simple. Let us revisit one of the methods we worked with in a previous chapter.

```
/// <summary>
/// Calculate the production volume of steel per bin
/// </summary>
/// <param name="binVolume"></param>
/// <param name="factor"></param>
/// <returns>Bin Volume less Remainder</returns>
public int ProductionVolumePerBin(int binVolume, int factor)
{
    Contract.Ensures(Contract.Result<int>() == binVolume,
        "The factor used will result in scrap. Please modify the cutting
factor.");

    int remainder = CutSteel(binVolume, factor);
    return binVolume - remainder;
}

/// <summary>
/// Calculate any remainder after the modulus operation between volume
and factor
/// </summary>
/// <param name="volumeToCut"></param>
/// <param name="factor"></param>
/// <returns>Remainder after cutting</returns>
private int CutSteel(int volumeToCut, int factor)
{
    // Use modulus to determine if the factor produces any scrap
```

```
    return volumeToCut % factor;
}
```

Code Listing 61: Create IntelliTest for method

As before, the preceding code listing tells the calling method that the method under contract will result in the cut volume always equaling the bin volume. This means that all the steel has been cut perfectly and no off-cuts resulted by using the specific factor.

To generate a new IntelliTest for the **`ProductionVolumePerBin()`** method, right-click on the method and select **Create IntelliTest** from the context menu.

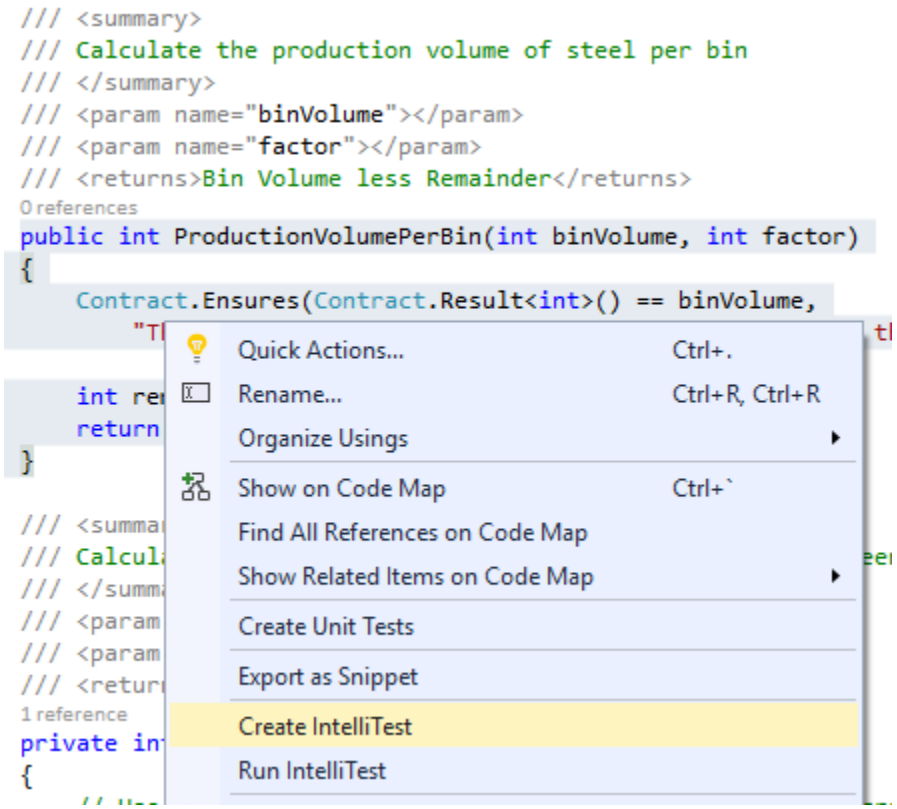

Figure 45: Create IntelliTest

Visual Studio will now display the Create IntelliTest window where you can configure additional settings for your generated IntelliTest.

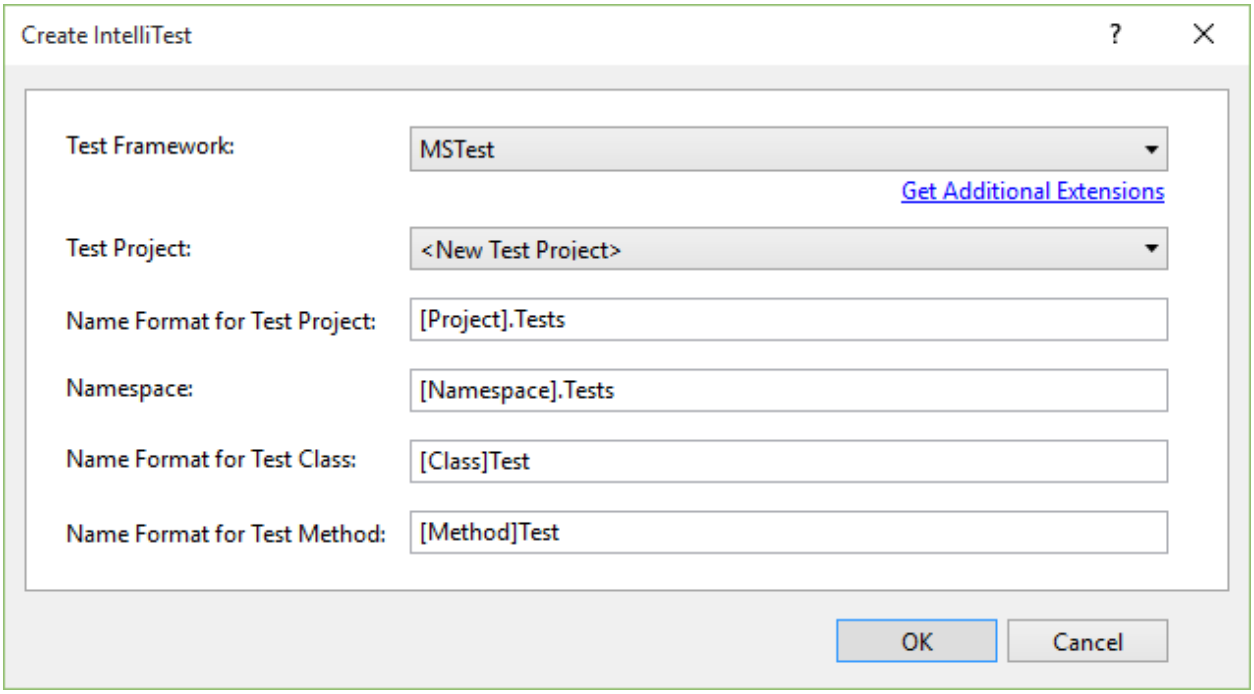

Figure 46: Create IntelliTest Settings

If this is your first time creating an IntelliTest, you will see that **MSTest** is the only option listed under **Test Framework**. You can, however, install third-party unit test frameworks if desired (more on this later). When you are done, click **OK**.

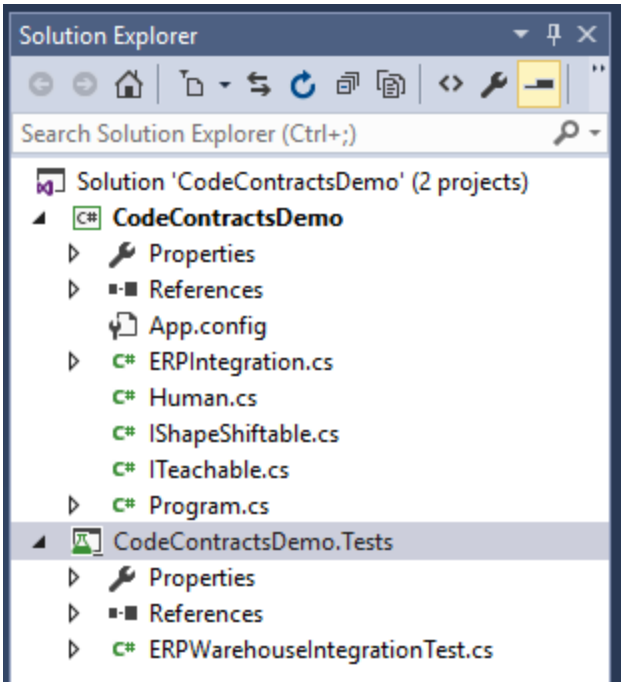

Figure 47: Test Project Added to Solution

Visual Studio will now create your test project for you. When it has completed the process, the new project will be visible in the **Solution Explorer** window.

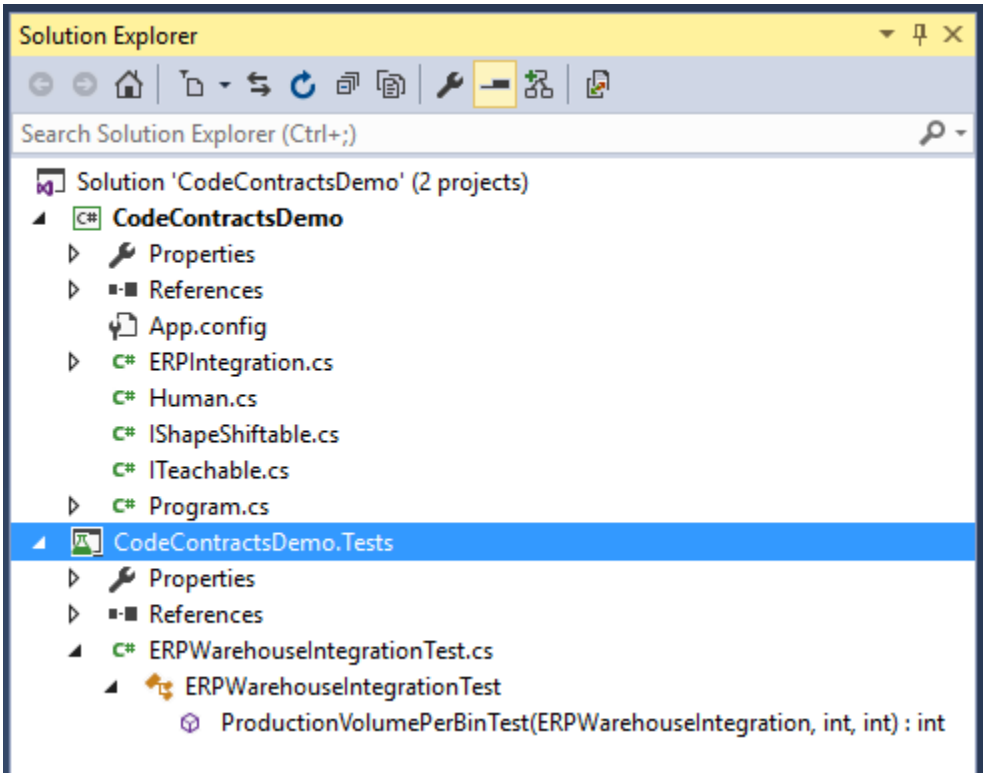

Figure 48: IntelliTest Created

Expanding the **ERPWarehouseIntegrationTest.cs** file, you will see that a test called **ProductionVolumePerBinTest** has been created.

```
// <copyright file="ERPWarehouseIntegrationTest.cs">Copyright ©
2015</copyright>
using System;
using Microsoft.Pex.Framework;
using Microsoft.Pex.Framework.Validation;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CodeContractsDemoProject.Tests
{
    /// <summary>This class contains parameterized unit tests for
ERPWarehouseIntegration</summary>
    [PexClass(typeof(ERPWarehouseIntegration))]

[PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
    [PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException),
AcceptExceptionSubtypes = true)]
    [TestClass]
    public partial class ERPWarehouseIntegrationTest
```

```
    {
        /// <summary>Test stub for ProductionVolumePerBin(Int32,
Int32)</summary>
        [PexMethod]
        public int ProductionVolumePerBinTest(
            [PexAssumeUnderTest]ERPWarehouseIntegration target,
            int binVolume,
            int factor
        )
        {
            int result = target.ProductionVolumePerBin(binVolume, factor);
            return result;
            // TODO: add assertions to method
ERPWarehouseIntegrationTest.ProductionVolumePerBinTest(ERPWarehouseIntegrat
ion, Int32, Int32)
        }
    }
}
```

Code Listing 62: ProductionVolumePerBinTest Created

All it took to generate this test was a few clicks. The integration of IntelliTest allows developers to easily create tests for critical logic in their code.

Run IntelliTest

After we have created the first IntelliTest, we need to do something with it. The logical option is to run the test and see what the results are of the test on the method. To do this, right-click the method and click **Run IntelliTest** in the context menu.

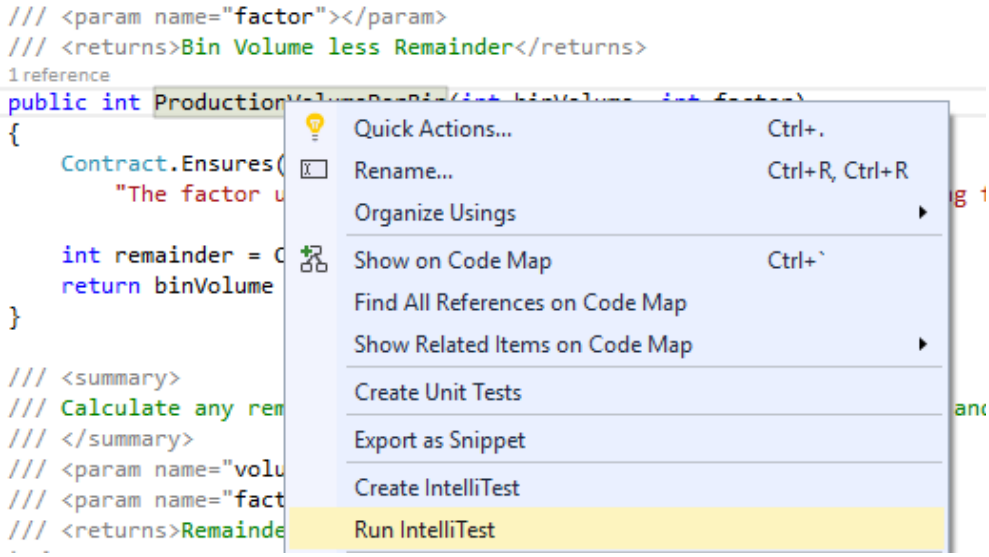

Figure 49: Run IntelliTest

Visual Studio will proceed to build your project and then launch the **IntelliTest Exploration Results** window. This window displays all the possible method parameters that would result in the maximum coverage of code for the method being tested. You can now get an overview of how the method fared under test.

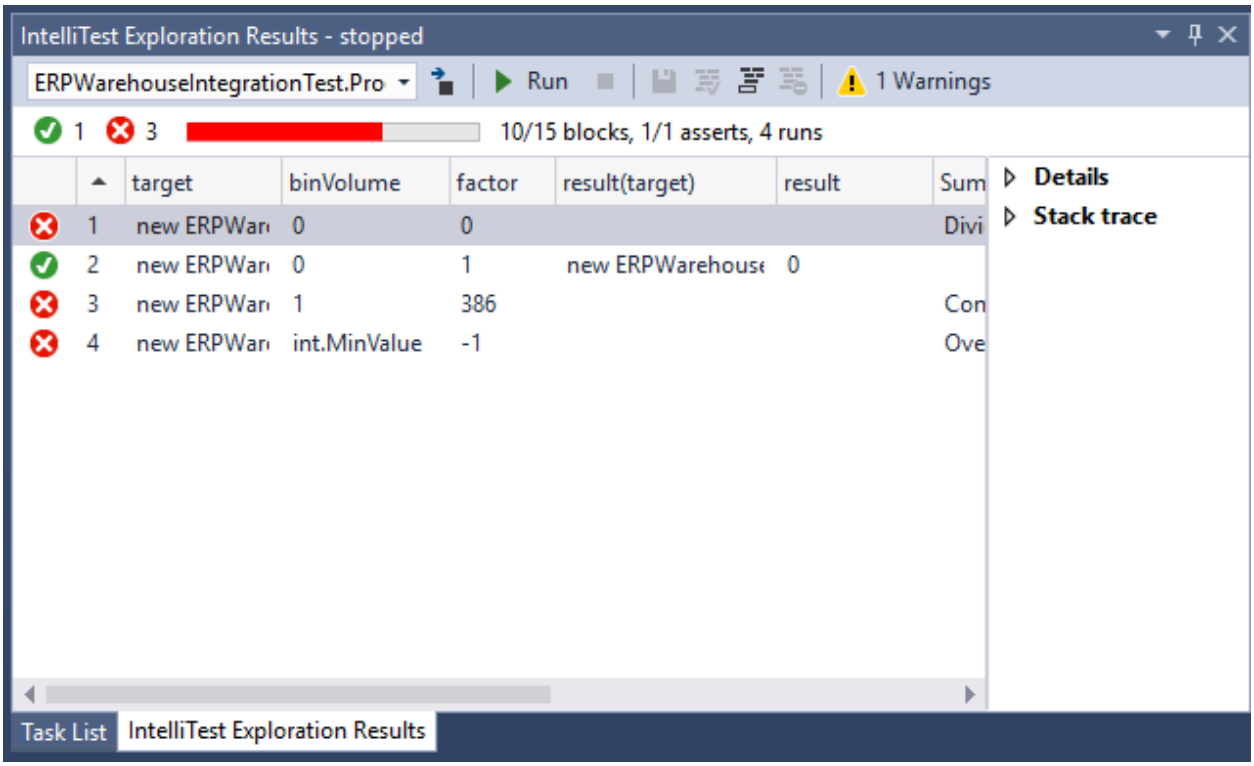

Figure 50: IntelliTest Exploration Results

Immediately we can see that three tests failed and one passed. To the right of the **IntelliTest Exploration Results** windows, the **Details** and **Stack trace** are displayed in collapsible nodes. Let's have a closer look at the test results.

Test 2: Passed

The **`ProductionVolumePerBin()`** method must always result in a cut volume equal to the bin volume. From the variables used in Test 2, we can see that it passed because the bin volume was **`0`** and the result of the method was also **`0`**.

```
[TestMethod]
[PexGeneratedBy(typeof(ERPWarehouseIntegrationTest))]
public void ProductionVolumePerBinTest33()
{
    ERPWarehouseIntegration eRPWarehouseIntegration;
    int i;
    eRPWarehouseIntegration = new ERPWarehouseIntegration();
    i = this.ProductionVolumePerBinTest(eRPWarehouseIntegration, 0, 1);
```

```
    Assert.AreEqual<int>(0, i);
    Assert.IsNotNull((object)eRPWarehouseIntegration);
    Assert.AreEqual<int>(0, eRPWarehouseIntegration.MaxBinQuantity);
    Assert.AreEqual<int>(0, eRPWarehouseIntegration.CurrentBinQuantity);
}
```

Code Listing 63: Generated Code in Details

Looking at the **Details** node, we can see the generated code for the unit test.

Test 1: Failed, DivideByZeroException

Looking at Test 1, we can see that it failed because it tried to divide by zero.

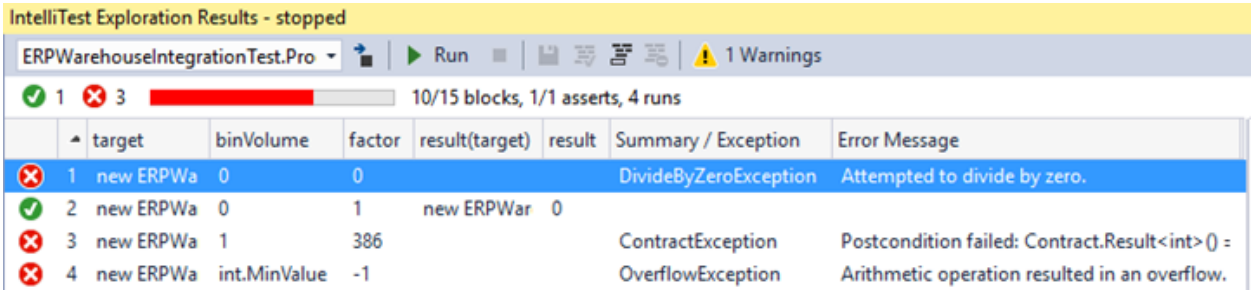

Figure 51: Divide by Zero Exception

This means that we forgot to take care of a factor value of zero. If you look at the **`CutSteel()`** method you will see that we are dealing with a modulus and not a divisor. So how can we have a divide by zero exception? Well the rule of thumb is that if the second operand of a **`/`** or a **`%`** is zero, we will have a divide by zero exception.

Test 3: Failed, ContractException

Test 3 failed because the Code Contract ensures a result that the method isn't adhering to. Remember, the **`ProductionVolumePerBin()`** method ensures that the bin volume is returned to the calling code, which signifies that we have produced a perfect cut with zero scrap. The test failed, and in a production environment this situation will happen. Our Code Contract ensures that the result returned by the method will always equal the bin volume. This means that we need to deal with the eventuality of an imperfect cut.

Test 4: Failed, OverflowException

The last failed test is rather interesting. We can see that the test was passed **`int.MinValue`** for **`binVolume`**. According to the MSDN documentation, **`int.MinValue`** represents the smallest possible value for **`Int32`**. It is also a constant with a value of **`-2,147,483,648`**. The factor has a value of **`-1`** and the modulus results in an **OverflowException**. This means that our code in the **`CutSteel()`** method will effectively be **`int.MinValue % -1;`** and this fails our test.

The reason that this happens is because the C# language specification implements it as such. In **7.8.3 Remainder operator** of the **C# Language Specification** it states: “If the left operand is the smallest **`int`** or **`long`** value and the right operand is **`-1`**, a **`System.OverflowException`** is thrown.”

Note: If you would like to dig a little deeper, you can view this thread on StackOverflow for a very good explanation on why this is implemented this way in C#.

Fixing test failures

We can see from the previous IntelliTests that there are some loopholes in our **`ProductionVolumePerBin()`** method. These need fixing, and the most obvious one to fix is the divide by zero exception. Let’s add a **`Contract.Requires()`** to our method to only allow factor values greater than **`1`**.

```
/// <summary>
/// Calculate the production volume of steel per bin
/// </summary>
/// <param name="binVolume"></param>
/// <param name="factor"></param>
/// <returns>Bin Volume less Remainder</returns>
public int ProductionVolumePerBin(int binVolume, int factor)
{
    Contract.Requires(factor > 1,
        "The supplied cutting factor must be more than the value 1.");
    Contract.Ensures(Contract.Result<int>() == binVolume,
       "The factor used will result in scrap. Please modify the cutting
factor.");

    int remainder = CutSteel(binVolume, factor);
    return binVolume - remainder;
}
```

Code Listing 64: Modified Code to Ensure Valid Integers

After adding the **`Contract.Requires()`** precondition, the method will only allow valid cutting factor values. Run the IntelliTest again by right-clicking on the **`ProductionVolumePerBin()`** method and selecting **Run IntelliTest** from the context menu.

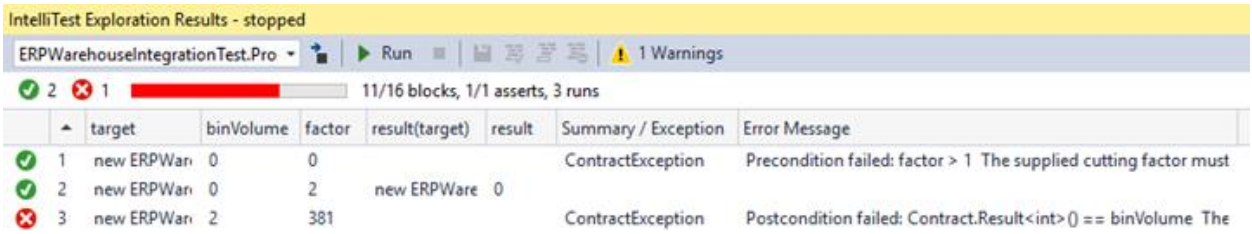

Figure 52: IntelliTest Results after Valid Integer Change

The results of the test are quite different. Our contract preconditions are working correctly and limiting the erroneous values from being passed to our method. However, we can see that the IntelliTest passed a **`binVolume`** value significantly smaller than the **`factor`** value. Let us work on this issue first by requiring the **`binVolume`** value to never be smaller than the **`factor`** value.

```
/// <summary>
/// Calculate the production volume of steel per bin
/// </summary>
/// <param name="binVolume"></param>
/// <param name="factor"></param>
/// <returns>Bin Volume less Remainder</returns>
public int ProductionVolumePerBin(int binVolume, int factor)
{
    Contract.Requires(factor > 1,
        "The supplied cutting factor must be more than the value 1.");
    Contract.Requires(binVolume > factor,
        "The cutting factor cannot be greater than the bin volume");
    Contract.Ensures(Contract.Result<int>() == binVolume,
        "The factor used will result in scrap. Please modify the cutting
factor.");

    int remainder = CutSteel(binVolume, factor);
    return binVolume - remainder;
}
```

Code Listing 65: Modified Code to Ensure Valid Cutting Factor

To achieve this, we need to add another **`Contract.Requires()`** pre-condition that will require that the **`binVolume`** value is always greater than the **`factor`** value. Run the IntelliTest again from the context menu.

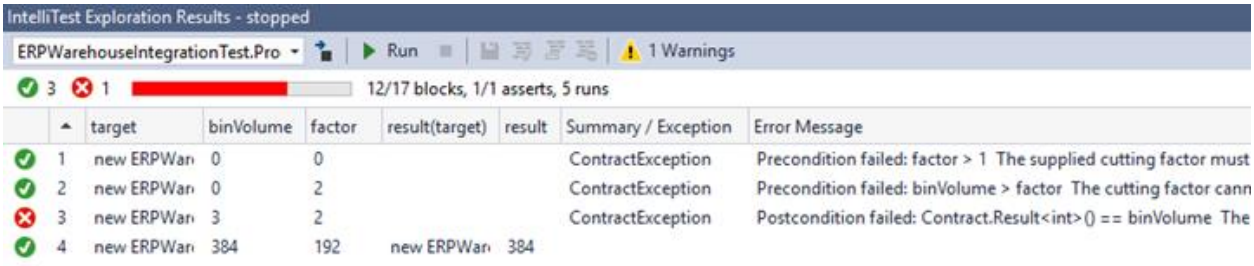

Figure 53: IntelliTest Results after Valid Cutting Factor Change

The results returned from this test tell us that the only issue we are experiencing with the **`ProductionVolumePerBin()`** method is that it still fails on the value being returned. Our method guarantees the calling code that it will return a perfect cut every time, and it is failing this contract. We might want to consider adding a bit more intelligence to this method by letting our code suggest a valid cutting factor to the user if the supplied one is not valid.

```
/// <summary>
/// The new valid cutting factor calculated by ProductionVolumePerBin
/// </summary>
public int CalculatedCuttingFactor { get; private set; } = 0;



/// <summary>
/// Calculate the production volume of steel per bin
/// </summary>
/// <param name="binVolume"></param>
/// <param name="factor"></param>
/// <returns>Bin Volume less Remainder</returns>
public int ProductionVolumePerBin(int binVolume, int factor)
{
    Contract.Requires(IsEven(binVolume),
        "Invalid bin volume entered");
    Contract.Requires(factor > 1,
        "The supplied cutting factor must be more than the value 1.");
    Contract.Requires(binVolume > factor,
        "The cutting factor cannot be greater than the bin volume");
    Contract.Ensures(Contract.Result<int>() == binVolume,
        "The factor used will result in scrap. Please modify the cutting
factor.");

    int remainder = CutSteel(binVolume, factor);
    while ((binVolume - remainder) != binVolume)
    {
        CalculatedCuttingFactor = CalculateNewCutFactor(binVolume);
        remainder = CutSteel(binVolume, CalculatedCuttingFactor);
    }

    return binVolume - remainder;
}

/// <summary>
/// Calculate any remainder after the modulus operation between volume
and factor
/// </summary>
/// <param name="volumeToCut"></param>
/// <param name="factor"></param>
/// <returns>Remainder after cutting</returns>
private int CutSteel(int volumeToCut, int factor)
{
    // Use modulus to determine if the factor produces any scrap
    return volumeToCut % factor;
}

/// <summary>
```

```
/// Calculate a new cutting factor
/// r.Next(1, 7); returns a random number between 1 and 6
/// </summary>
/// <param name="binVol">Upper range value of random (bin volume +
1)</param>
/// <returns>
/// A new cutting factor greater than 1 and equal to the bin volume
/// </returns>
private int CalculateNewCutFactor(int binVol)
{
    Random r = new Random();
    return r.Next(2, binVol + 1);
}

/// <summary>
/// Ensure that the passed volume is even
/// </summary>
/// <param name="volume">The volume to verify</param>
/// <returns>boolean</returns>
public bool IsEven(int volume)
{
    return volume % 2 == 0;
}
```

Code Listing 66: Intelligent ProductionVolumePerBin Method

As you can see from the previous modified code listing, I have done a few things. Business rules state that the bin volume will always be an even number. I have therefore added a **`Contract.Requires()`** pre-condition to ensure that only even integers are passed to the **`ProductionVolumePerBin()`** method.

Another addition to our code is a **`CalculatedCuttingFactor`** property that will hold the newly calculated cutting factor if the supplied factor is invalid. For this I have included a new method called **`CalculateNewCutFactor`** that will try alternate values for the cutting factor in order to produce the perfect cut.

The **`ProductionVolumePerBin()`** method will determine if the cutting factor is valid. If not, it will run the **`while`** loop until a valid cutting factor is returned and a perfect cut is achieved. In a production environment, however, you might want to consider using a sanity loop counter variable to create an exit condition or throw an exception when some maximum number of iterations is reached. There is still a lot of fine-tuning that can be done to the **`ProductionVolumePerBin()`** method, which I will not go into here as I simply want to illustrate a concept.

After the code is modified, run the IntelliTest again.

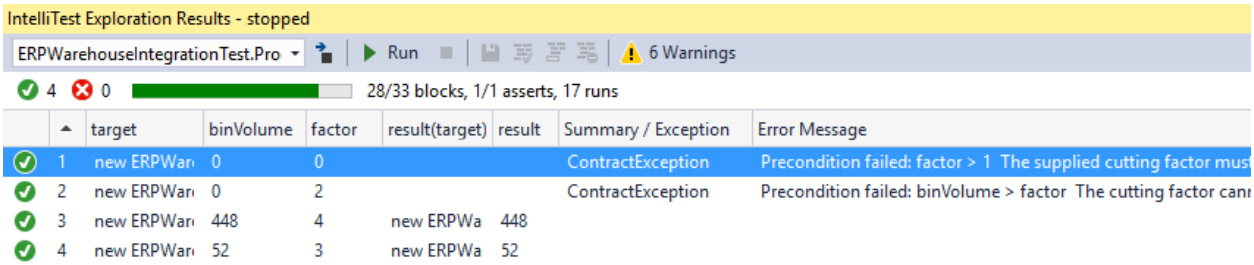

Figure 54: All IntelliTest Tests Passed

From the test result we can see that the **`ProductionVolumePerBin()`** method has held up to the requirements imposed by our Code Contracts. It has also generated six warnings, which I'll discuss shortly.

The calling code can now implement the **`ProductionVolumePerBin()`** method without needing to cater for invalid values being returned. It knows that the method will return a perfect cut every time. The only check that needs to be done is to see whether a new factor has been suggested or if the supplied factor is valid.

```
int binVol = 20;
int factor = 3;
CodeContractsDemoProject.ERPWarehouseIntegration oWhi =
    new CodeContractsDemoProject.ERPWarehouseIntegration();
int result = oWhi.ProductionVolumePerBin(binVol, factor);
if (oWhi.CalculatedCuttingFactor != factor &&
oWhi.CalculatedCuttingFactor != 0)
{
    Console.Write($"The supplied cutting factor of {factor} resulted in "
        + "an imperfect cut. The system suggests using the following "
        + $"cutting factor: {oWhi.CalculatedCuttingFactor}");
}
else
    Console.Write($"The cutting factor of {factor} resulted in 0 scrap");

Console.ReadLine();
```

Code Listing 67: Code Calling ProductionVolumePerBin

You will notice that I am using **string interpolation** in the **`Console.Write`** each time. This is one of the new features in C# 6. The preceding code only needs to check the **`CalculatedCuttingFactor`** property to see if the cutting factor has changed. It knows that under contract, the **`ProductionVolumePerBin()`** method will always result in a perfect cut.

We can further improve the preceding code, but the concept I wanted to illustrate is clear. Code Contracts lend themselves very well to tests created with IntelliTest in Visual Studio Enterprise 2015. You can combine the power of both technologies to create highly robust code and highly enforced business rules to make your applications perform well in a production environment.

IntelliTest warnings

A discussion of IntelliTest would not be complete without discussing the Warnings output screen.

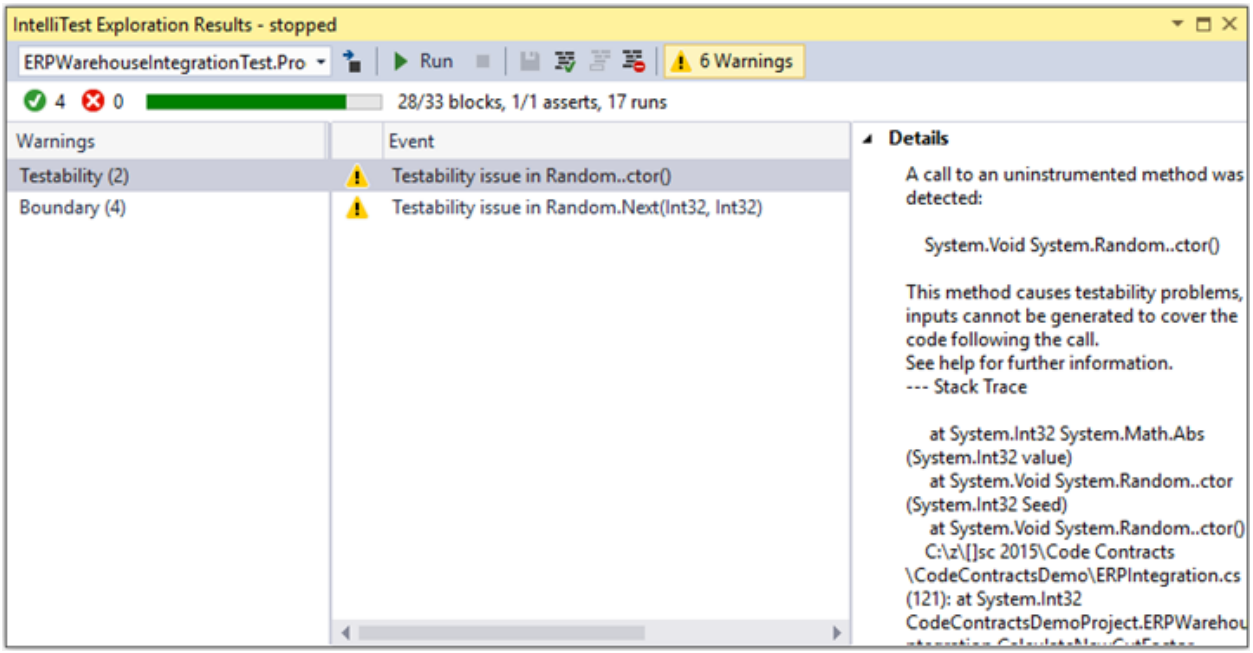

Figure 55: IntelliTest Warnings

You will notice that the **IntelliTest Exploration Results** screen generated six warnings. These are mostly normal warnings that might require your attention. To be on the safe side, review these for any obvious issues. While this is somewhat beyond the scope of this book, I will briefly mention two warning types seen in **Testability** and **Boundary**.

Testability

The warning we see is that an uninstrumented method was detected in our code. This is specific to the **`Random()`** method used in the **`CalculateNewCutFactor()`** method. This simply means that IntelliTest cannot dig down into all the paths in my code in order to generate the required outputs it needs in order to test.

Boundary

IntelliTest imposes certain limits on paths it executes in order to prevent it from getting stuck in the event that the application goes into an infinite loop. These limits can be modified by clicking the **Fix** icon on the menu bar.

Code coverage

Ideally you would want to see 100 percent code coverage (33/33 blocks). Our tests only covered 28/33 blocks. Further reading on IntelliTests would allow you to understand how to ensure good code coverage and which warnings can be suppressed safely.

Installing third-party frameworks

As mentioned previously, you are able to get additional extensions for the test framework to use when creating IntelliTests. To do this, you can use the Visual Studio Extension Manager or go to the Visual Studio Gallery on the MSDN website.

Here's how to install third-party frameworks from Visual Studio Extensions:

1. Navigate to **Tools** and select **Extensions and Updates**.
2. Expand **Online** > **Visual Studio Gallery** > **Tools**, and select **Testing**.
3. Browse the results and select the framework you require.
4. Click **Download**.

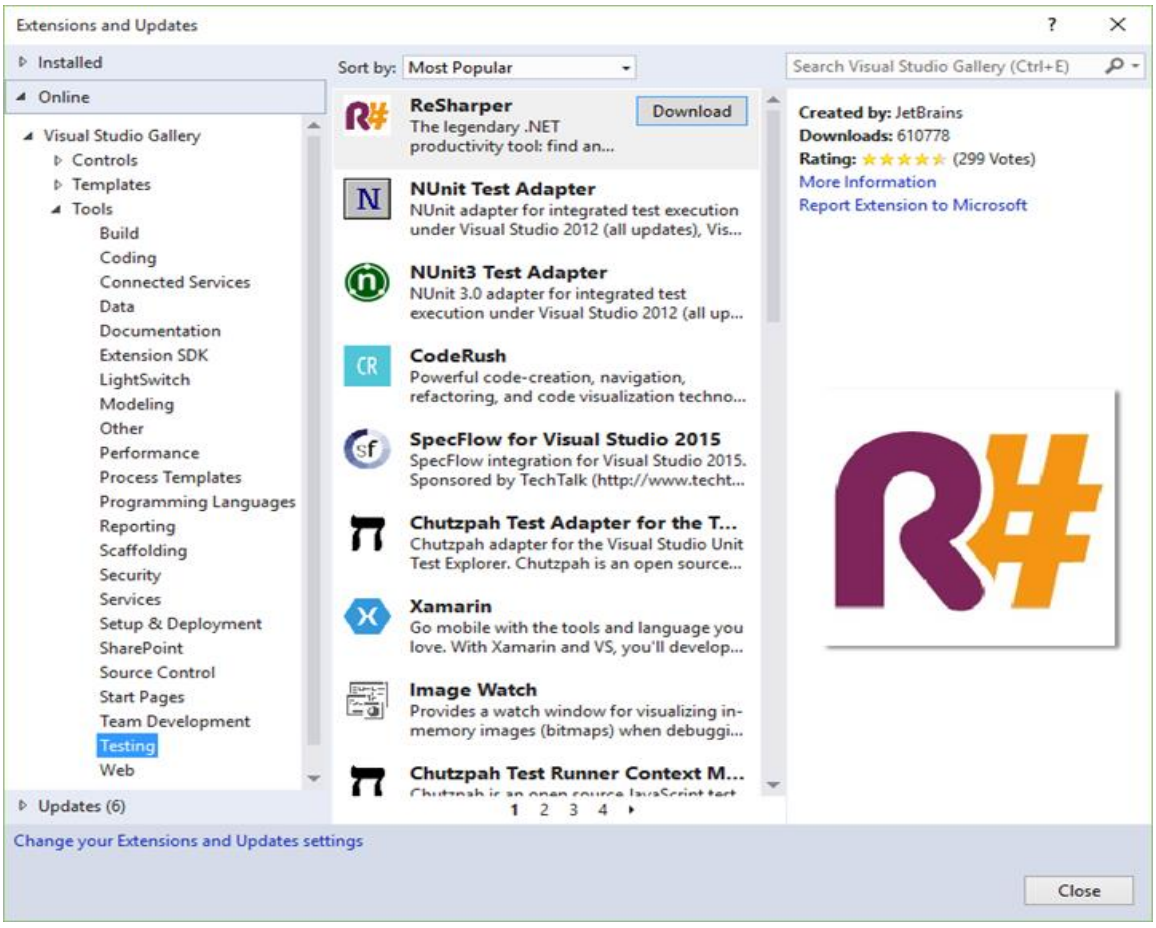

Figure 56: Installing Third-Party Frameworks from Visual Studio

Installing third-party frameworks from the Visual Studio Gallery:

1. Go to the Visual Studio Gallery.
2. Enter the framework name in the **Find** text box.
3. Select the framework from the search results to download.

If you are not sure which framework you want or don't know the name of the framework you are looking for, you can browse a list of frameworks on the Visual Studio Gallery:

1. Go to the Visual Studio Gallery.
2. Click the **Browse** link.
3. Under **Categories**, expand **Tools** and choose **Testing**.
4. Choose the framework you want and download the tool.

Chapter 5 Code Contracts Editor Extensions

Making Code Contracts more useful

So far we have seen a lot of what Code Contracts can do. Here's a tip on how to make the contracts you create more useful to developers using your classes and methods. Using Code Contracts Editor Extensions will allow you to see what the method you're calling into requires with regard to the contracts defined in that method. It will do this without requiring you to drill down into the method and see what Code Contracts it implements.

```
.ProductionVolumePerBin(binVol, factor);
```

int CodeContractsDemoProject.ERPWarehouseIntegration.ProductionVolumePerBin(int binVolume, int factor)
Calculate the production volume of steel per bin

Figure 57: ProductionVolumePerBin Quick Info

The method we used in Chapter 4 displays the comments you provided in the XML comments for that method in the Quick Info window when the pointer hovers over the method. This is expected behavior, but I have no idea from that Quick Info window what contracts the method implements. Code Contracts Editor Extensions changes this. To install it, go to the **Tools** menu in Visual Studio and click **Extensions and Updates**.

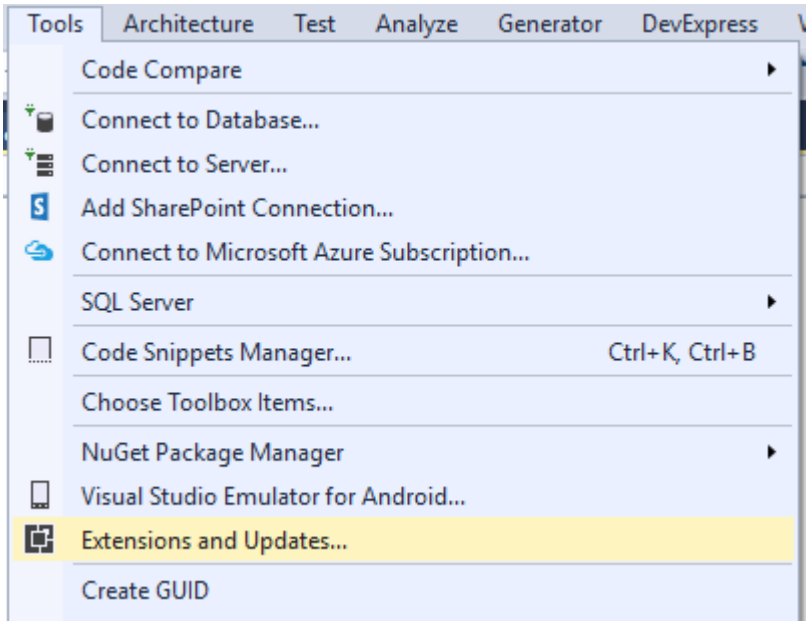

Figure 58: Extensions and Updates

From the **Extensions and Updates** window, select the **Online** tab and search for **Code Contracts Editor Extensions**. The results returned should be fairly limited. From here, click **Download** to download and install the extension.

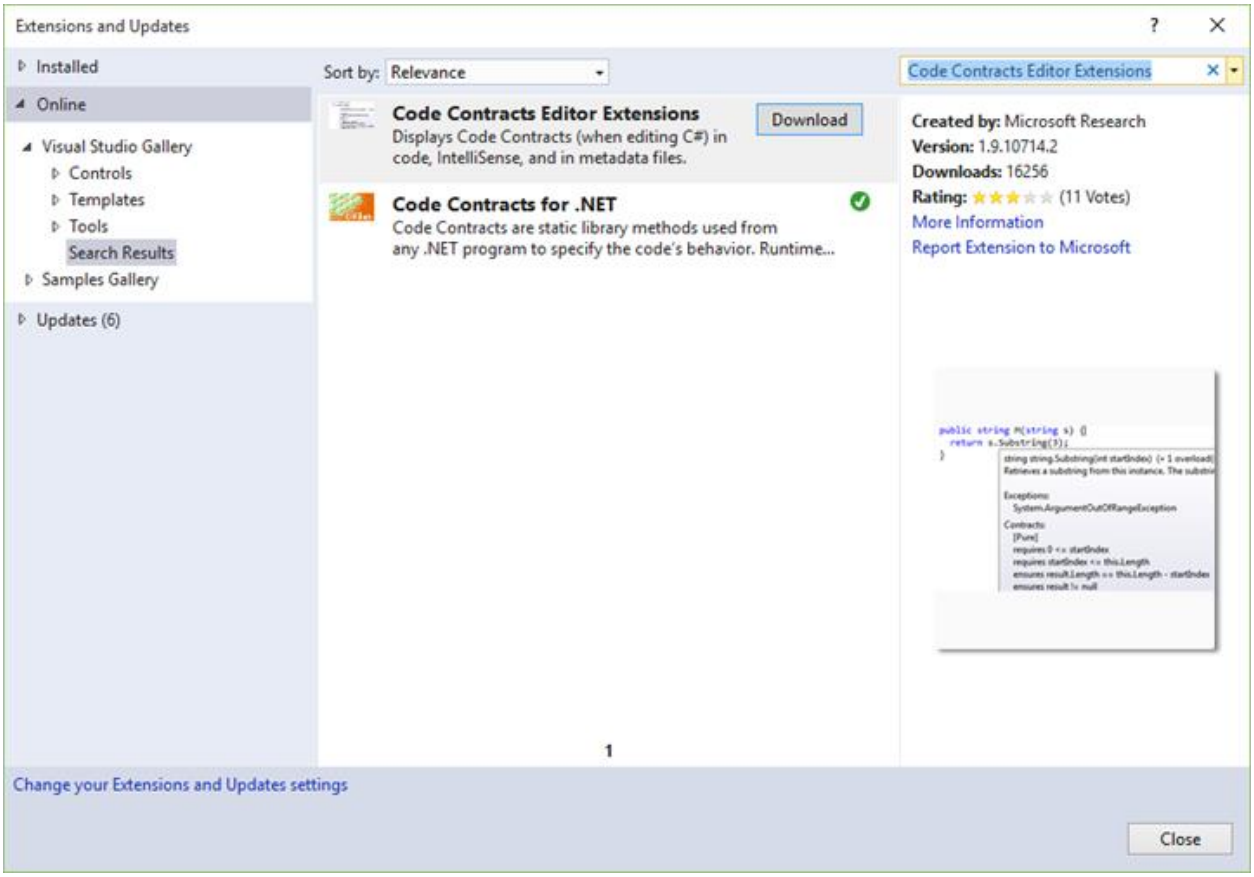

Figure 59: Install Code Contracts Editor Extensions

Once Code Contracts Editor Extensions have been installed, you will need to restart Visual Studio. The beauty of this extension is that you now get a peek inside the methods you create, which will display the contracts for that method.

Before this will work though, you need to ensure that the **Contract Reference Assembly** is set to **Build** in the Code Contracts settings.

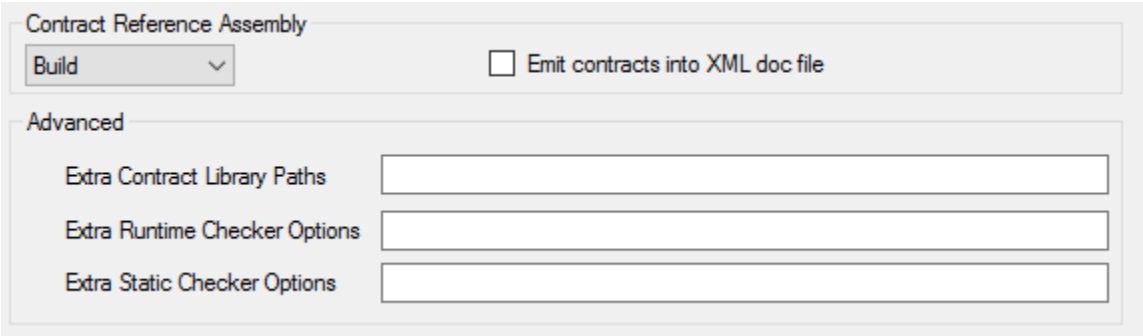

Figure 60: Contract Reference Assembly

Now if you hover over the **`ProductionVolumePerBin()`** method, you will see that the Quick Info window is rich with information regarding the contracts it implements.

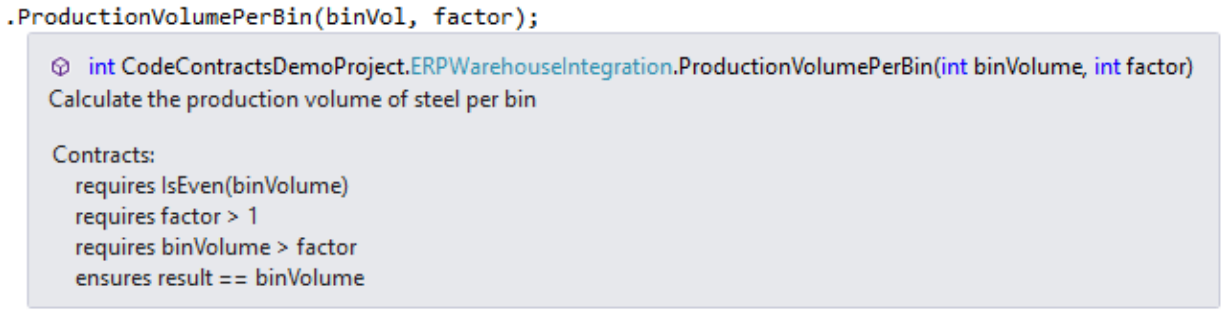

Figure 61: Method Tooltip Enhanced with Contracts

This allows me to be able to use the method without having to see inside the method, and pass it valid parameters that will validate successfully. I now have much more information regarding the method I'm calling.

Another gem when using Code Contracts Editor Extensions is the ability to see which Code Contracts a base class implements. Have another look at our **`CalculateNewCutFactor()`** method.

```
/// <summary>
/// Calculate a new cutting factor
/// r.Next(1, 7); returns a random number between 1 and 6
/// </summary>
/// <param name="binVol">Upper range value of random (bin volume +
1)</param>
/// <returns>
/// A new cutting factor greater than 1 and equal to the bin volume
/// </returns>
private int CalculateNewCutFactor(int binVol)
{
    Random r = new Random();
    return r.Next(2, binVol + 1);
}
```

Code Listing 68: Random() Method

Hovering over the **`Next()`** method, we can peer into the contracts it requires. The Code Contracts Editor Extensions do this by mining the base classes and displaying the contracts implemented.

```
private int CalculateNewCutFactor(int binVol, int factor)
{
    Random r = new Random();
    return r.Next(2, binVol + 1);
}
```

```
int Random.Next(int minValue, int maxValue) (+ 2 overloads)

Contracts:
  requires minValue <= maxValue
  ensures result >= minValue
  ensures result < maxValue || maxValue == minValue && result == minValue
```

Figure 62: Random Next Method Contracts

We can see that one of the Code Contracts implemented by the **`Next()`** method is that the **`minValue`** must be less than or equal to the **`maxValue`**. For our requirements, I want to ensure that the **`minValue`** is always greater than or equal to **`2`**. Using the information we were able to glean regarding the Code Contracts implemented by the **`Next()`** method, we can now go ahead and create our own **`GetRandom()`** method that will conform to our Code Contract requirements.

```
/// <summary>
/// Calculate a new cutting factor
/// r.Next(1, 7); returns a random number between 1 and 6
/// </summary>
/// <param name="binVol">Upper range value of random (bin volume +
1)</param>
/// <returns>
/// A new cutting factor greater than 1 and equal to the bin volume
/// </returns>
private int CalculateNewCutFactor(int binVol)
{
    return GetRandom(2, binVol + 1);
}

/// <summary>
/// Get a random number
/// </summary>
/// <param name="minValue">Value not less than 2</param>
/// <param name="maxValue">Upper range value of the random number to
generate</param>
/// <returns>A random integer</returns>
static int GetRandom(int minValue, int maxValue)
{
    Contract.Requires(minValue >= 2,
        "minValue cannot be less than 2");
    Random r = new Random();
    return r.Next(minValue, maxValue);
}
```

Code Listing 69: Custom GetRandom() Method

Using Code Contracts and Code Contracts Editor Extensions allows us to write more robust code and fine-tune our code to easily conform to the required business rules.

Tip: You need to keep in mind that the `Random` class isn't a true random number generator. When you call `Next()`, for example, the `Random` class uses some internal state to return a number that appears to be random. It then changes its internal state so that the next time you call `Next()`, it returns another apparently random number. Generating true random numbers is beyond the scope of this book; if you require true randomness, you will need to do a bit more research.

Chapter 6 Conclusion

So far, we have looked at setting up Code Contracts and using them to validate logical correctness by using the various contract methods. We looked at how Code Snippets make life easier for developers who use Code Contracts extensively. We also saw how to integrate Code Contracts into your documentation output and how to generate user-friendly documentation using Sandcastle Help File Builder.

Moving on to more advanced topics and tips, we looked at how to integrate Code Contracts with abstract classes and interfaces, and we saw that Code Contracts lend themselves very well to testing your methods with IntelliTest. We explored how another tool called Code Contracts Editor Extensions expose base classes and your custom Code Contracts to calling code right there in the Quick Info window.

From here, I would suggest checking out the Code Contracts GitHub page and starting to integrate Code Contracts into your projects. The more you use Code Contracts, the easier it will become to create well documented, robust code.

Chapter 7 Tools and Resources

Code Contracts on GitHub

To have a look at the source code for Code Contracts, contribute some code, or familiarize yourself with certain aspects on Code Contracts, visit the project on GitHub. There is also a forum and FAQ that could be a great help.

Code Contracts at Microsoft Research

For more on the background of Code Contracts, go to the Microsoft Research website.

Code Contracts User Manual

For more background knowledge and to solidify certain concepts, I suggest you read through the Code Contracts User Manual.

Hottest Code Contract answers on Stack Overflow

Stack Overflow is one of the best sources on the web for answers and solutions to your Code Contract questions.

Code Contracts Editor Extensions

See the Visual Studio Gallery for the Code Contracts Editor Extensions.

Sandcastle Help File Builder

To generate great documentation from your XML comments, grab a free copy of Sandcastle Help File Builder on GitHub.