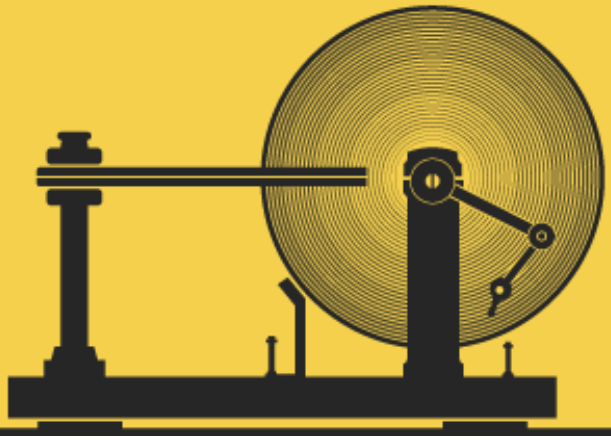


ReSharper Succinctly

by Peter Shaw

Syncfusion | Technology Resource Portal

ReSharper Succinctly

By
Peter Shaw

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Igal Tabachnik
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Morgan Cartier Weston, 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

As an early adopter of IT back in the late 1970's to early 1980's, I started out with a humble little 1k Sinclair ZX81 home computer.

Within a very short space of time this small 1k machine became a 16k Tandy TRS-80, followed by an Acorn Electron and eventually, after going through many other different machines, a 4MB arm powered Acorn A5000.

After leaving school and getting involved with DOS-based PC's, I went on to train in many different disciplines in the computer networking and communications industry.

After returning to University in the mid 1990's and gaining a BSc in Computing for Industry, I now run my own Consulting Business in northeast England called Digital Solutions Computer Software Ltd. I advise clients at both a hardware and software level in many different IT disciplines, covering a wide range of domain-specific knowledge from mobile communications and networks right through to geographic information systems, banking and finance, and web design.

With over 30 years of experience in the IT industry with many differing and varied platforms and operating systems, I have a lot of knowledge to share.

You can often find me hanging around in the Lidnug .NET users group on LinkedIn that I help run, at Stack-Overflow (and its GIS-specific board) and on Twitter as @shawty_ds, and now also on Pluralsight, where my various videos are available.

I hope you enjoy the book, and get something from it.

Please remember to thank Syncfusion (@Syncfusion on Twitter) for making this book (and others in the range) possible, allowing people like me to share our knowledge with the .NET community at large. The *Succinctly* series is a brilliant idea for busy programmers.

Introduction

Welcome to ReSharper Succinctly, an e-book that will hopefully change the way you work, making you into a far more productive developer than you ever thought possible.

You are about to take a tour of what is probably one of the most well-known productivity add-ons for Visual Studio ever produced for .NET developers.

Over the course of this book, you'll learn exactly what ReSharper is, how to get started with it, and more importantly, what it can do for you as a busy developer.

One thing to note though—throughout this book we'll refer to ReSharper as R#; this is actually a more common nomenclature used by those that actually install it. So when you see me mention R#, don't worry—I'm not referring to some arcane new programming language.

Your first experience with R#

If your first experience with R# is anything like mine, then I can tell you now, it will frustrate you.

I remember when I first started to use it. It would routinely trip me up while trying to do its thing. I'd often lose my temper with it for rewriting my code, but please do persevere with it. You'll soon realize that some of the rewrites it performs for you are actually really useful, and as for creating well-rounded, well-structured, elegant code, it can often make optimizations that will actually result in the compiler making better decisions when compiling your projects.

I'm not going to pretend, however—it will annoy you over the first few months, especially if you're the type of developer that likes to have complete control over what you do.

You'll also see an increase in "squiggles," those nice little red wobbly lines that Visual Studio puts in your source code when you do something it objects to. However, you'll quickly come to realize that these "squiggles" come in many different colors and for many different reasons.

Learning the different colors is important, because not every one represents an infraction that the compiler will object to. In fact, many of them relate to things that R# can do to improve your code for you, but in many cases it won't actually make the change until you ask it to.

Before we move on, let's talk versions. For this book, I'm using the following:

- Visual Studio 2013 Ultimate, Update 2
- R# 8.1 Build 8.1.23.546, but in order to demo the install I'll be switching to v8.2

Ready to start your journey? Great, then let's begin.

Chapter 1 Getting Started

What exactly is ReSharper?

ReSharper is a productivity extension designed to be installed under most versions of Visual Studio (Microsoft's flagship programming environment). Now in its 8th release, R# has been used by programmers for the best part of the last seven to nine years (over five years by your humble author), and shows no signs of stopping.

However, once you install it you'll find that it's much more than just a simple extension. It contains an absolutely staggering number of tools under the hood.

There are tools to reformat your code, tools to make sure you don't break your builds, and tools to help you find your way around complex large solutions. There are options to allow a senior or lead developer to enforce an enterprise-wide standard that all developers must adhere to, meaning better cohesion between team players on a development team.

Finally, it will (if configured to do so) improve your code for you as you go, as well as providing hundreds of keyboard shortcuts that mean you can keep your fingers where they need to be.

R# has become so popular that Visual Studio itself in its latest builds has actually got some of R#'s features built into it, and many of the other popular tools now also emulate the same key bindings for similar things.

However, there's no substitute for R# once you get to know your way around, and believe me, I've tried the others—even given them a real fair, long-term trial—but just could not gel with them the way I do with R#

R# doesn't stop there.

The developer group JetBrains has also thought long and hard about possible edge cases found while using this tool, and realized that this is a development tool used by developers. It may sound obvious, but many companies actually forget to think like this, and simply just make "Tools for Devs" without ever stopping to consider anything that might occur outside of the tool.

As a result, JetBrains has made R# extensible and even produced an SDK available for free to install via NuGet.

This SDK allows you to create your own plug-ins for R#, so if there is a scenario or a tool that you need that’s not already provided, it's child's play to add your own extensions. There is still a partial download available from the JetBrains website, but this no longer contains anything other than project templates and MSBuild targets, allowing you to get a jumpstart on building your own extensions. The meat of the SDK and its DLLs are all now only distributed via NuGet.

All told, R# could be the ultimate tool for any serious developer using Visual Studio—as you'll see as we go further into this book, the list of things that R# can do to help you seems endless.

Installing ReSharper

Before you can get started using R#, you first need to install it.

If you want to test it before buying, you can [download a 30-day trial version](#) from the JetBrains web site. Currently, it looks like this:

Figure 1: JetBrains website

Click the pink button on the right to get the latest version (currently 8.2).

Go ahead and download the latest build, and make yourself a fresh cup of coffee while it arrives on your PC.

Running the installer

Before we actually launch the installer, make sure that you have no instances of Visual Studio running. R# works with Visual Studio 2005 upwards, so if you want the install to run smoothly, make sure you quit all versions that you'll want this to install in.

In reality, I've installed it before with Visual Studio running, but I had to quit and reload before anything took effect, so it's better to just terminate any running instances while you run the installer.

Note: By the time this book is released, v9 will likely be mainstream, so there will be a new installer. Details on this are in the last chapter of this book.

Once you click on the downloaded file, a dialog box will ask if you want to run the file. Click **Yes**, and after a minute or two you should be greeted with the following:

Figure 2: First installer screen

As you can see, this copy has picked up all three versions of Visual Studio that I have installed. Make sure your read the license agreement, and click the check box to verify that you have read it. Next, click either **Install** or **Advanced** to continue.

Install will take you through the select sensible defaults route, and do most of the work for you. The advanced route will give you the following:

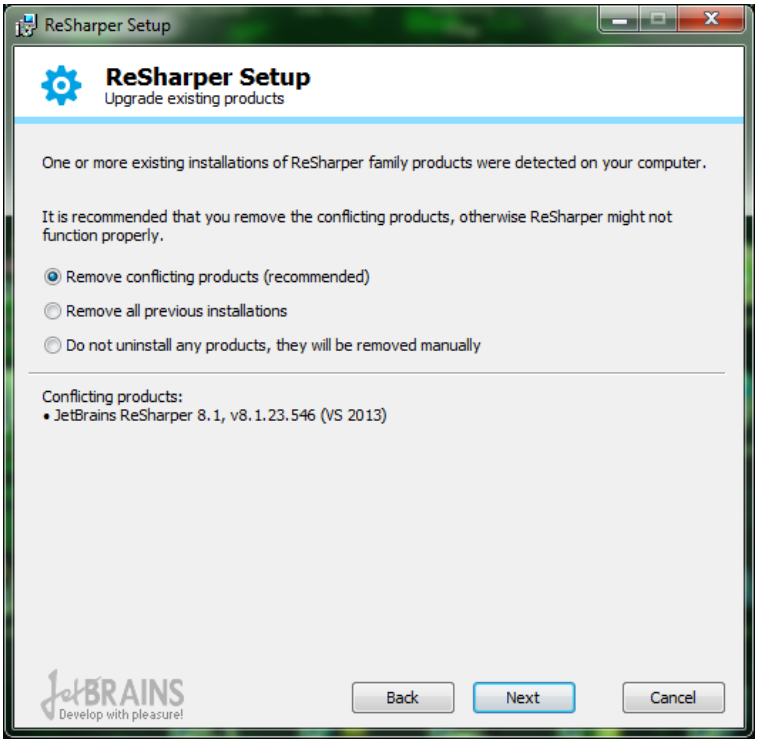


Figure 3: Initial installer screen when an advanced install is chosen

In my case it's picked up the 8.1 version, which it says it wants to remove. We'll allow it to do so (I'll reinstall 8.1 later as I don't have a full license for 8.2); you'll most definitely want to do this if you're performing an upgrade.

Make your selection and click **Next**, and you should then see the following:

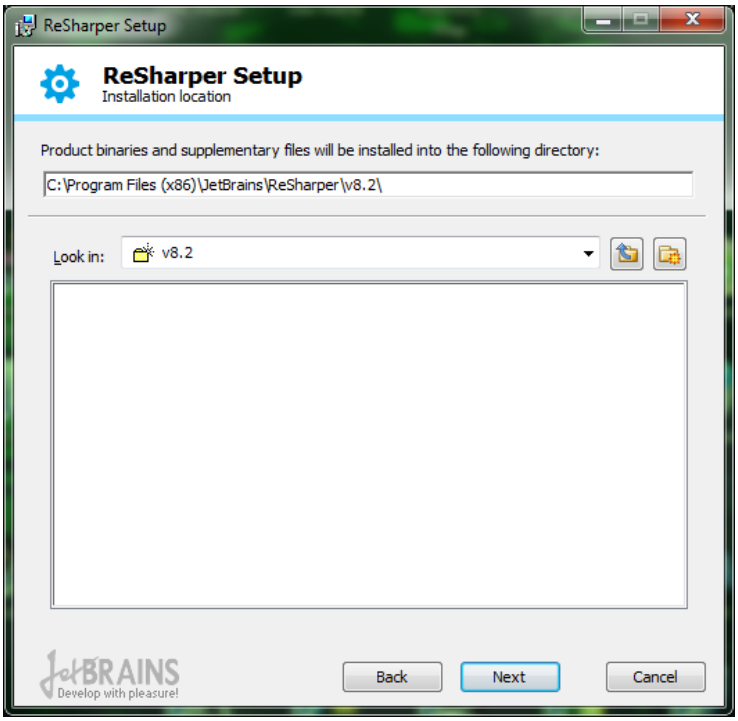


Figure 4: Installer screen requesting save location

This is just the usual directory setup dialog box; the default is usually the best choice.

Click **Next**, then **Install,** and everything should continue in an automatic fashion. You'll see this dialog while it completes the installation:

Figure 5: Installer progress screen

Once the install is complete, you should see the following window:

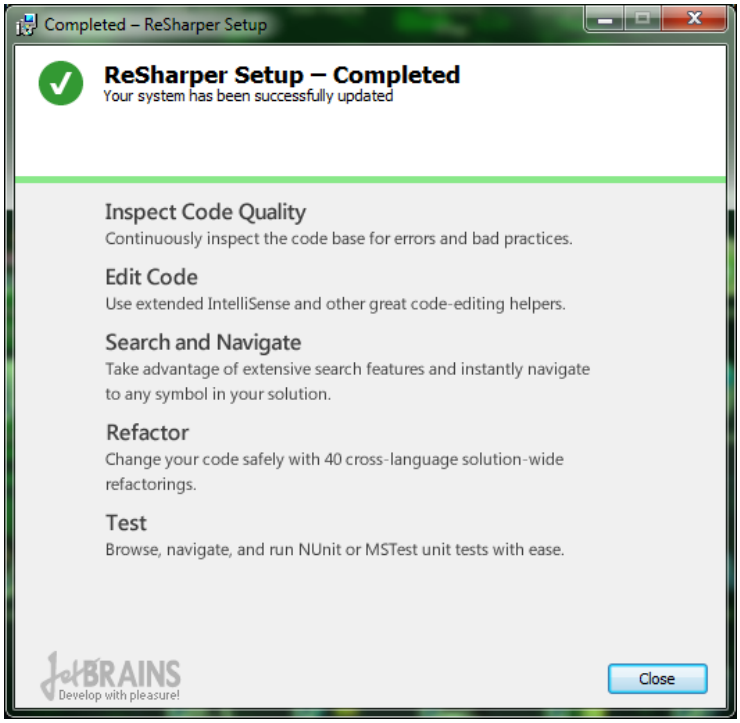


Figure 6: Installer complete screen

Click **Close**, and at this point, we're ready to fire up Visual Studio and start exploring.

R# first look

The first time you fire up Visual Studio after installing R#, things might take a little longer than normal to start up. Don't worry, R# can take a little bit of time to configure things on the first run. You may also be asked to choose a keyboard scheme; just select the set of keyboard shortcuts you'd like to use and everything will continue as normal.

You'll know R# is ready to run when you see the R# menu in your Visual Studio Menu bar:

Figure 7: Visual Studio menu bar showing R# installed

Let's have a look at what's on the menu.

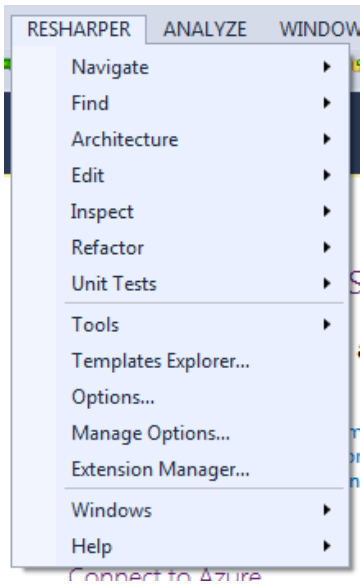


Figure 8: Main R# Visual Studio menu

These are the main core tools available to the R# user. However, as you'll see if you start to hover over the options, most of them are not enabled.

One thing you will see when you first use ReSharper is the product registration dialog box:

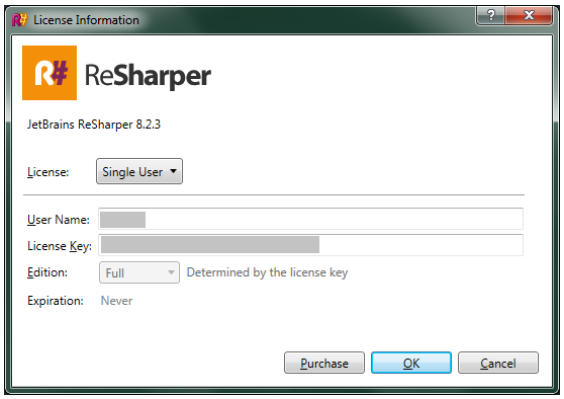


Figure 9: The ReSharper Registration dialog

Registration is a required step, and you won't be able to use the product without doing so. If you're not ready to buy the product just yet, you can apply for a 30-day free trial from the JetBrains website.

The copy I'm using for this book is already registered, as you can see in the previous figure. Once you have a registration, trial or otherwise, you'll need to enter the details provided into this dialog box to continue.

Because R# is a programming tool, you need to have an active project loaded for majority of the menu items to be enabled.

If we go to **RESHARPER** > **Help** > **About JetBrains ReSharper**:

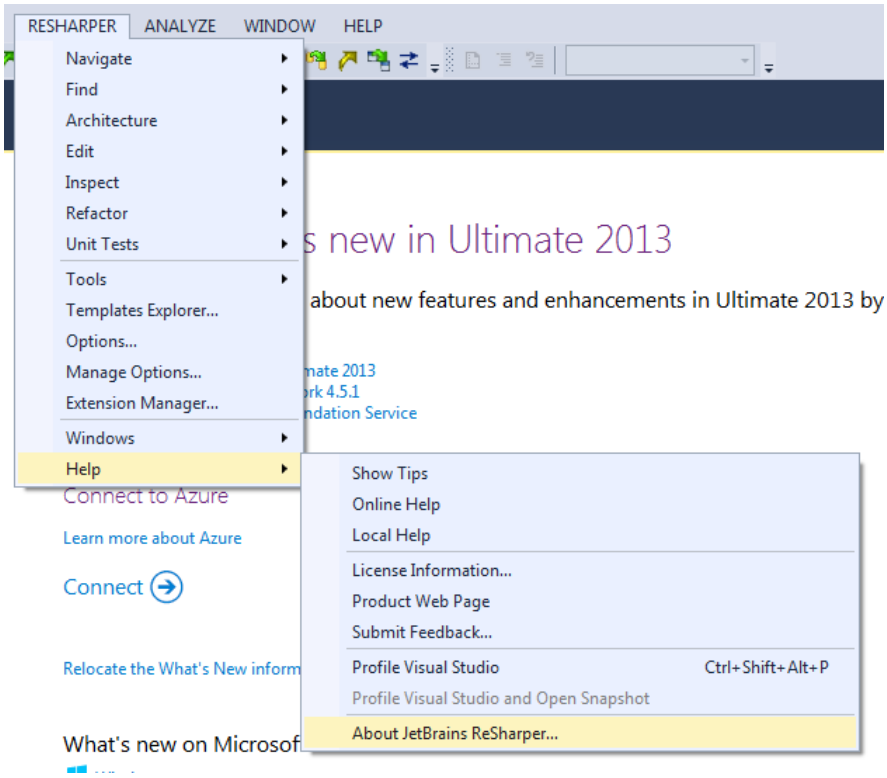


Figure 10: R# Help menu to get the About window

We should at least be able to get the About information up:

Figure 11: R# About window

R# options

Follow the menu path **RESHARPER** > **Options**, and you should see the following:

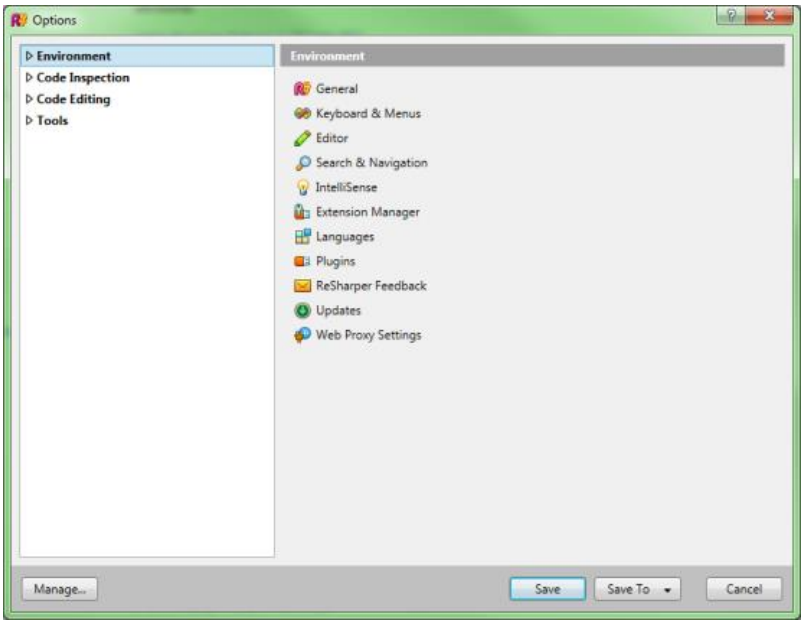


Figure 12: R# initial options window

This might be somewhat larger with the options trees expanded. There are a LOT of options—in fact, if you have the smaller default dialog, I can guarantee that you are going to want to make the dialog box a lot larger. When I'm working with the available options, I can often have the box almost as large as my screen resolution!

If you click on the main tree nodes without expanding them, you should see that you get the underlying options list in the right-hand pane. In Figure 12, you can see I'm ready to go with the environment options. Click the small triangle to the left of the text and expand the Environment options tree. You should see the following:

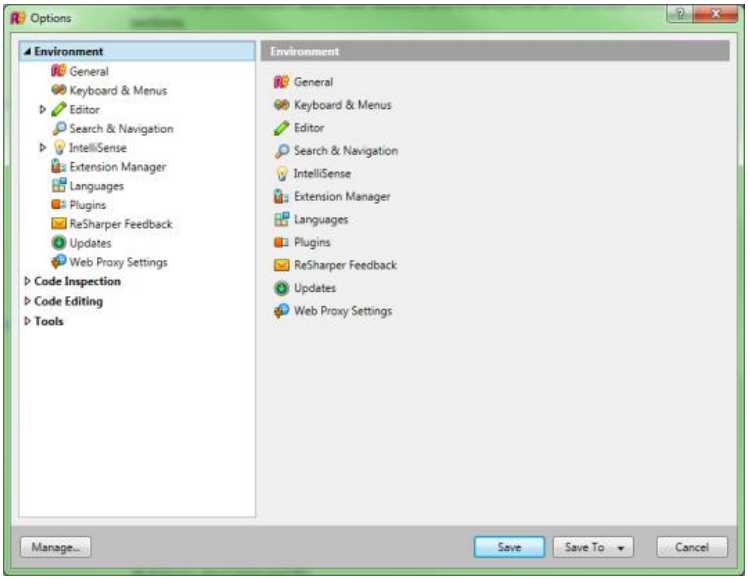


Figure 13: Options dialog showing environment options expanded

Click on **General** and the right panel will change. You'll most likely want to expand the box size at this point.

The General options, followed by the Keyboard and Editor options, are the ones you'll most likely want to change in the first instance; the rest we'll cover as needed throughout the rest of this book.

Starting with the Environment options, you should have something like the following:

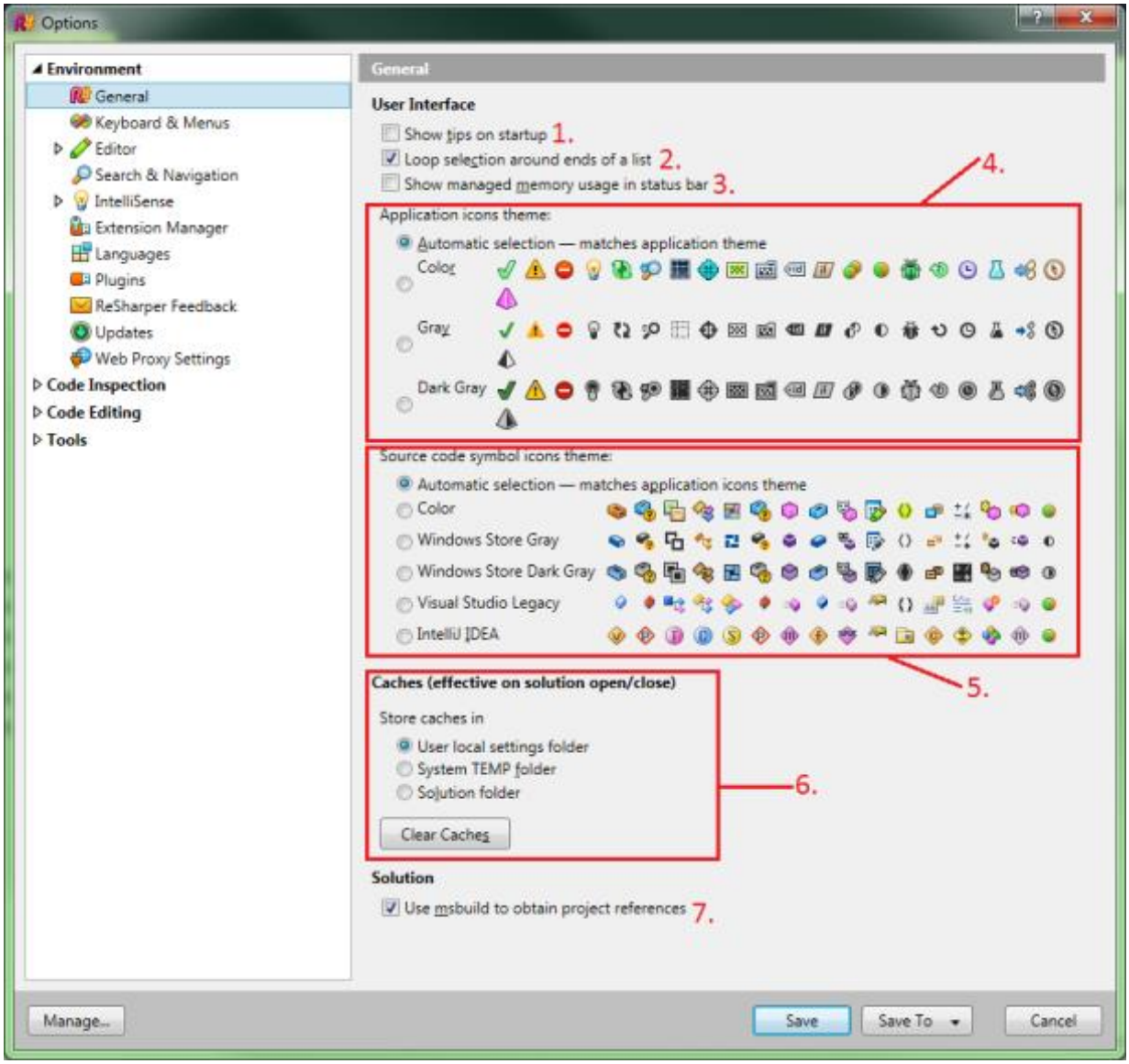


Figure 14: R# General options

1. **Show tips on startup**: This simply enables the R# tips dialog box when you first run VS. It's worth having this on for a while when you first start using the plug-in; you'll learn a lot of things that R# can do. The tips dialog box is also quite smart—it won't show you things you already use!
2. **Loop selection**: This might be a little bit of a mystery at first—it was to me. This option actually affects the pop-up lists (similar to VS's built-in IntelliSense pop-up lists). When you use a pop-up list and navigate to the end of it, having this option set will make your selection cursor jump back to the top; having it unselected will make the selection cursor stay where it is.
3. **Show managed memory**: This will show the memory consumption not just of R#, but of all the other add-ons you may have installed and running, and that are used by Visual Studio itself. This will be shown in the lower-right side of the Visual Studio status bar.
4. **Application icons**: These options allow you to choose the icon set you'd like R# on the whole to use. I generally just leave it on automatic, allowing R# to choose the most

appropriate icons. If you're used to another similar tool that uses a different icon set, you can force R# to use icons similar to that.

5. **Source code icons**: These options are similar to the general application icons, but instead of being app-specific, they are used in the source code display of your projects. You'll see them appear in test screens and down the left margin of your code window when R# is showing you something.
6. **Cache options**: While R# is working, it saves a lot of temporary data. Things such as internal code maps, and lists of options, warnings, and other things. You can choose where R# saves these caches. Usually you'll want to use system folders, but you decide you want to store them in the project folder, then you'll almost certainly want to make sure you configure things so the temp folders are not committed to your code repositories. The Clear Cache button allows you to expunge any temp data that's accumulated, should you need to.
7. **Use msbuild**: Selecting this option makes R# use the Microsoft Build engine (and project files) to figure out any references that are linked to the project. The opposite (not selected) is to have R# interrogate the various .NET assemblies in your project and work out the references map that way.

Moving on to the Keyboard & Menus options, if you click on the entry in the tree to the left, you should see something like the following:

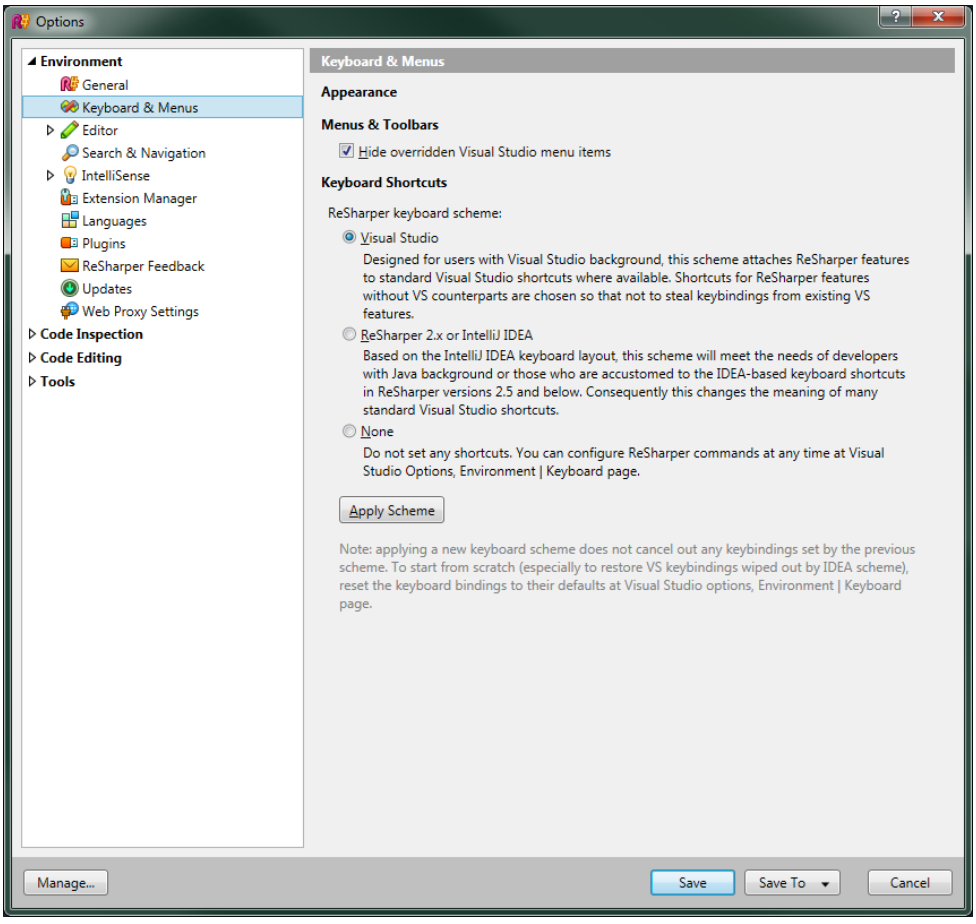


Figure 15: R# Keyboard & Menus options

The keyboard and menu options are pretty straightforward. The first option hides items that are already present in Visual Studio, which have been overridden or superseded by functionality available in R#. You can uncheck this so that everything is visible all the time. Visual Studio can get a bit crowded if you do this though, so it's highly recommended to leave it selected.

The options available for keyboard shortcuts are the same decisions you may have made when you ran the plug-in for the first time. R# remaps a lot of the existing Visual Studio keyboard keys, so that they activate the enhanced functionality available in the plug-in.

There are by default three sets of predefined maps (as well as the ability to create your own custom bindings). The choices in this dialog box are the same ones that you may have already been asked the first time you fired up R#.

If you choose the **Visual Studio** option, then your key mappings will mirror those already found in Visual Studio. **ReSharper 2.x or IntelliJ** will set your keyboard shortcuts up to be the same as another JetBrains product, IntelliJ IDEA. The last option of **None** won't set any keyboard mappings, and you'll have to use the custom keyboard options elsewhere to set all your own bindings up from scratch.

I've found the most productive and useful way is to select the Visual Studio mappings, then customize only the few that I need that are different.

Finally, we'll take a quick look at the Editor options.

You'll note that the Editor options menu has two sub-menus; as with the previous option sets, you'll need to click the small triangle to expand those options to see the tree. Your options dialog box should look similar to Figure 16.

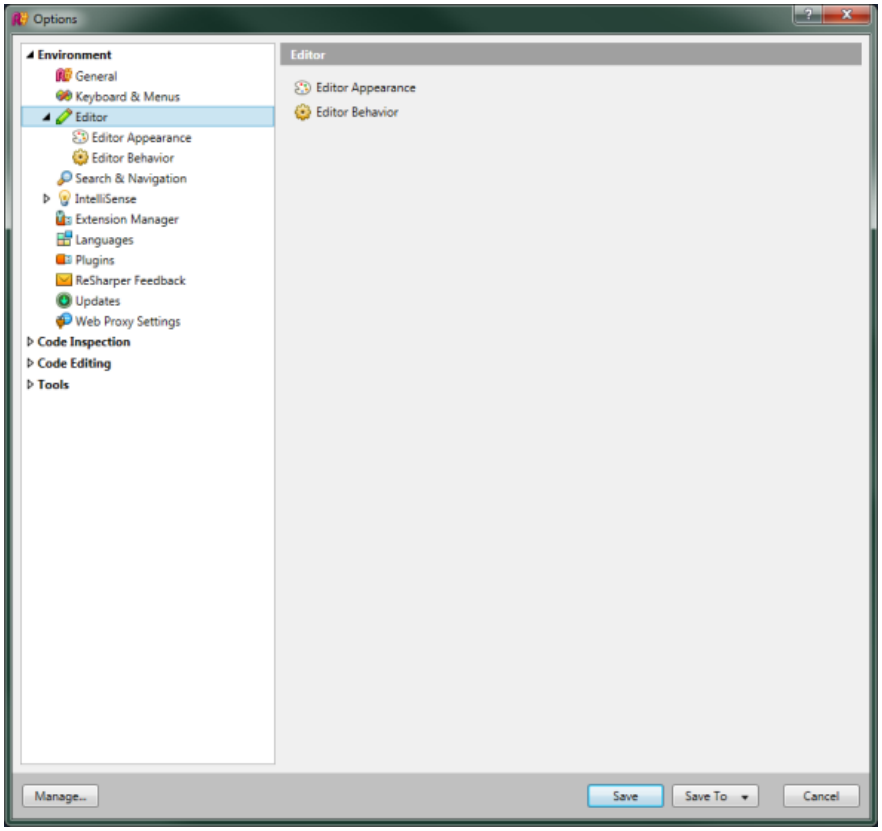


Figure 16: Initial Editor options

If you click on the **Editor appearance** options, you'll get a number of initial settings that affect how R# shows up in the main VS code editing window. This dialog box should look something like this:

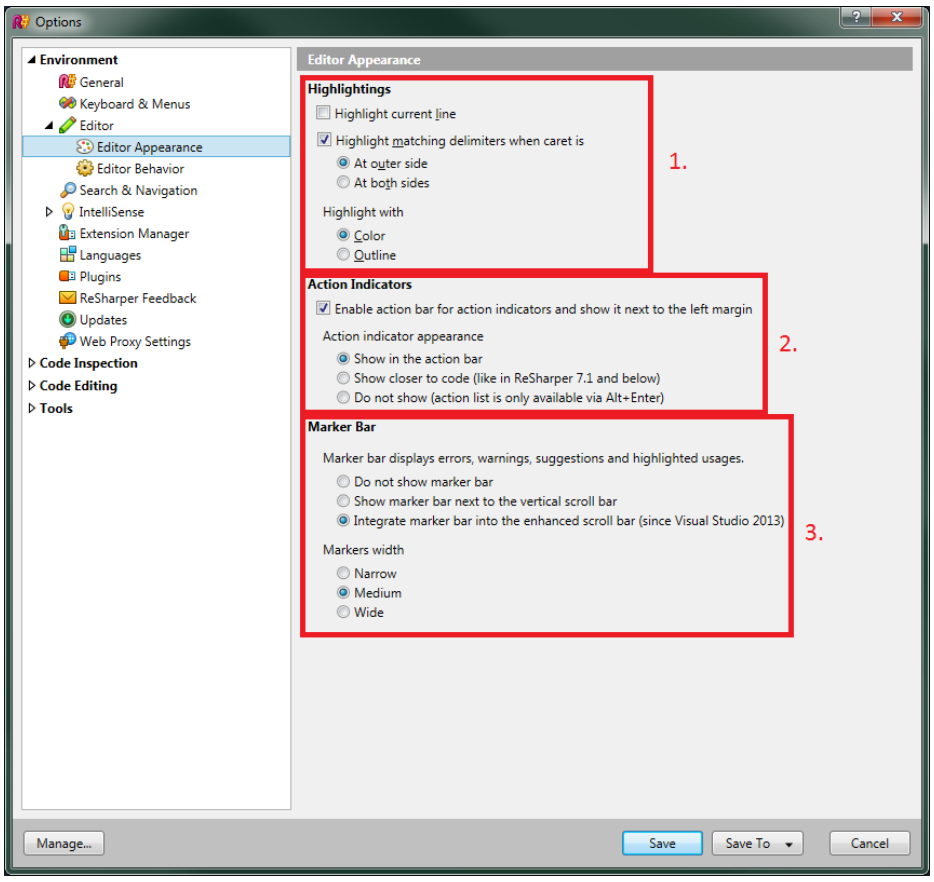


Figure 17: R# Editor Appearance options

The first set of options (section 1 in Figure 17) deal with helping you see where you are in your code.

- **Highlight current line** simply highlights the line you're currently working on, making it easy to see where you are when scrolling up and down at speed in your code.
- **Highlight matching delimiters** deals with things like parenthesis and quotation marks. If you move your cursor immediately to the left or right of one of these marks, then the closing mark will also be highlighted, enabling you to easily see where the closing section of the enclosed block is. The sub options **At outer side** and **At both sides** allow you to specify what's highlighted.
- **Highlight with** allows you to choose what to highlight with. **Color** makes the text in the highlighted block change to a defined color, whereas **Outline** is the more traditional form of a plain-colored border around the selection.

The second group deals with the R# action bar, which we'll meet soon. The action bar is R#'s way of showing you the points in your source code that it believes needs your attention. The action bar is positioned to the left of your Editor window (next to line numbers and breakpoints) and will display small icons, allowing you to click and select menus affecting the code at your current location.

- The **Enable action bar** option turns the bar on and off. Since most of what you'll use in R# is based on what you can see in this bar, it's usually recommended to leave this enabled.
- **Action bar appearance** allows you to choose how and where R# uses the bar to notify you of code issues. “Show in action bar” from v8 onwards, puts the light-bulb or hammer icon in the action bar to the left of the code, “Show closer to code” was the original legacy (pre-v8) behavior where the icon was shown over the code where the inspection is marked, potentially obscuring the code below. Finally, “Do not show” means don't show action bar icons at all for potential code issues.

The last of the three options are only recommended for those that have been using R# for a considerable amount of time and are used to knowing what R# will highlight and when, or for users who don't wish to be notified as they work, and instead choose to only use the “Code cleaning” tools from the menu prior to saving (or similar tools).

The third group of options deal with the bar on the right side of your Editor window known as the marker bar. Anyone who's seen a screen shot or video showing Visual Studio with a lot of colored lines in the right-hand margin has most likely already seen this in action. The marker bar is where R# places colored dashes that when hovered over will tell you what R# believes is a possible problem, error or, improvement at that location in your source code.

Clicking the dash mark will allow you to navigate to that place in the document, giving you instant access to options in the left column action bar, or via any pop-ups activated using **Alt**+**Enter** (more on this in the next section).

Similar to the action bar, you have three radio options that allow you to display the marker bar integrated with the Visual Studio 2013 marker bar, or as a separate bar alongside the existing one, or not at all. You'll almost always want to keep this enabled, as it's single-handedly the best way to get an overview of the current state of the source file you’re working with. I'll come back to the marker bar in the next chapter when we start looking at how R# integrates into the Visual Studio editor.

The last option in section 3 allows you to set the width of the bar to **Narrow**, **Medium**, or **Wide**.

The Editor Behavior options as seen in Figure 18 allow you to change the way R# formats and changes your source for you when typing.

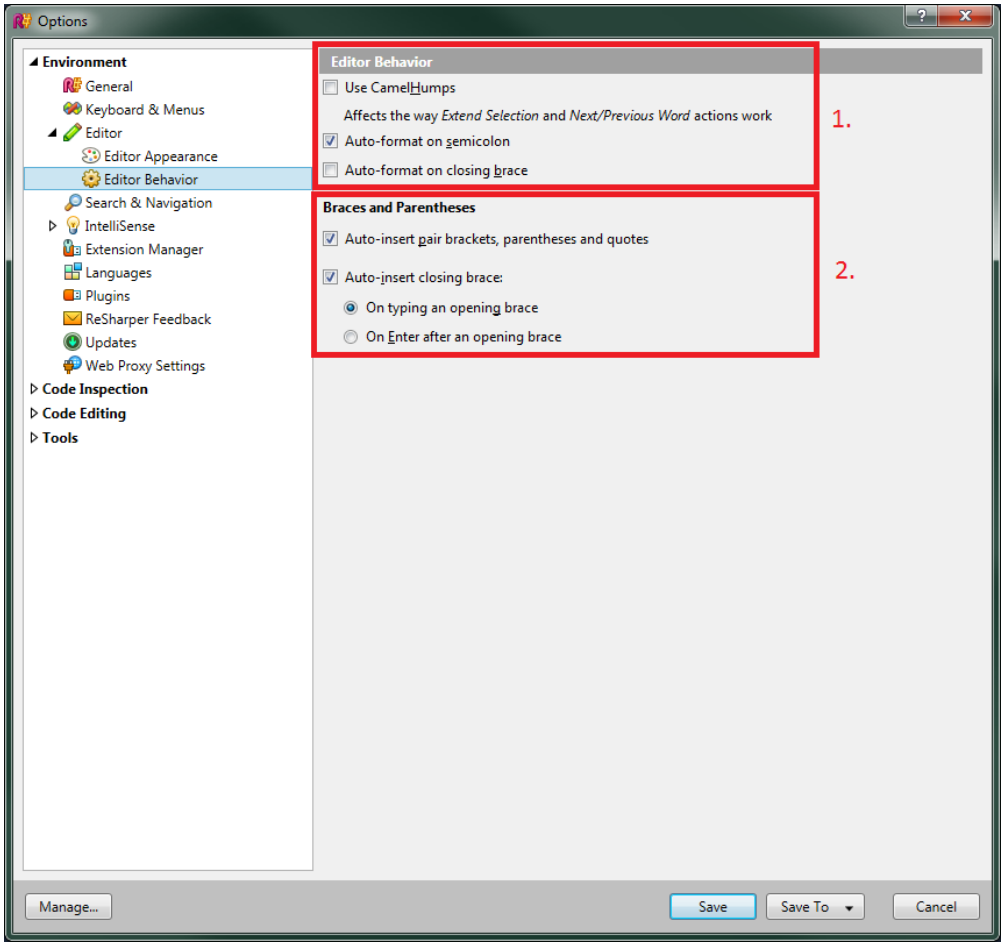


Figure 18: R# Editor Behavior options

Section 1 deals with basic editor behavior.

Use CamelHumps, when selected, means that certain keyboard actions (such as tab to expand templates) will be affected by the setting in differing ways. This also has a big effect on the way ReSharper's selection and navigation tools work; shortcuts, such as **Ctrl**+**Left/Right** and extend selection (which you'll see more of later) will use this setting to determine where to split identifiers into individual words when selecting parts of the text.

For those who are new to this, CamelHumps is the term given to text that is typed with the very first letter in lower case, then the rest of the first letters of each word in the sentence capitalized, with no spaces between the words, for example:

```
thisIsAnExampleVariableName
```

This is a common typing notation used in the Java and JavaScript worlds, and is often used for local variables in C# under Visual Studio. Some developers have their own typing and naming styles, so ReSharper gives you the option of turning this off if you wish to.

The remaining two in the first group, **Auto-format on semicolon** and **Auto-format on closing brace**, refer to how R# deals with your source code formatting when reaching the end of the line, statement, or current block.

Throughout the R# plug-in, there are various options to allow R# to format your code for you, in a style that ensures you'll always have a consistent look and feel to the way your code is formatted. If these rules were applied constantly as you typed your code, it would be very difficult for you to type correctly, especially if your code is changing around you.

The format on semicolons and braces means that R# will wait until one of these situations is encountered before it tries to apply any formatting rules that you may have set in your R# options. Waiting until a semicolon is reached generally means that you've reached the end of the line that you're currently typing, whereas waiting for a brace often means waiting until you finish an enclosure, such as using the default property syntax when creating objects containing data.

We'll leave our exploration of the R# options alone for now—I could write an entire book just exploring the options as they stand. We'll cover other options as we need to in further chapters of this book. In the meantime, if you want to explore the rest on your own I do encourage you to do so.

I've been using R# for almost six years now, and even I still discover things I never knew it could do. It truly is a massive plug-in with a staggering number of tools and options hidden behind the scenes, many of which you may never even discover.

The main help page for all the R# options can be found here.

ReSharper in the editor

Finishing off our first look at R#, we'll see the changes it makes to your general code-editing windows.

As has already been mentioned, R# adds two bars to the editor, an action bar on the left, and a marker bar on the right. Those of you who are using Visual Studio 2013 will already have gotten used to the marker bar, as VS2013 now has one of its own. Those who are still using an older Visual Studio version won't yet have been introduced to the concept.

The default editor view in the current version of R# should look something like Figure 19:

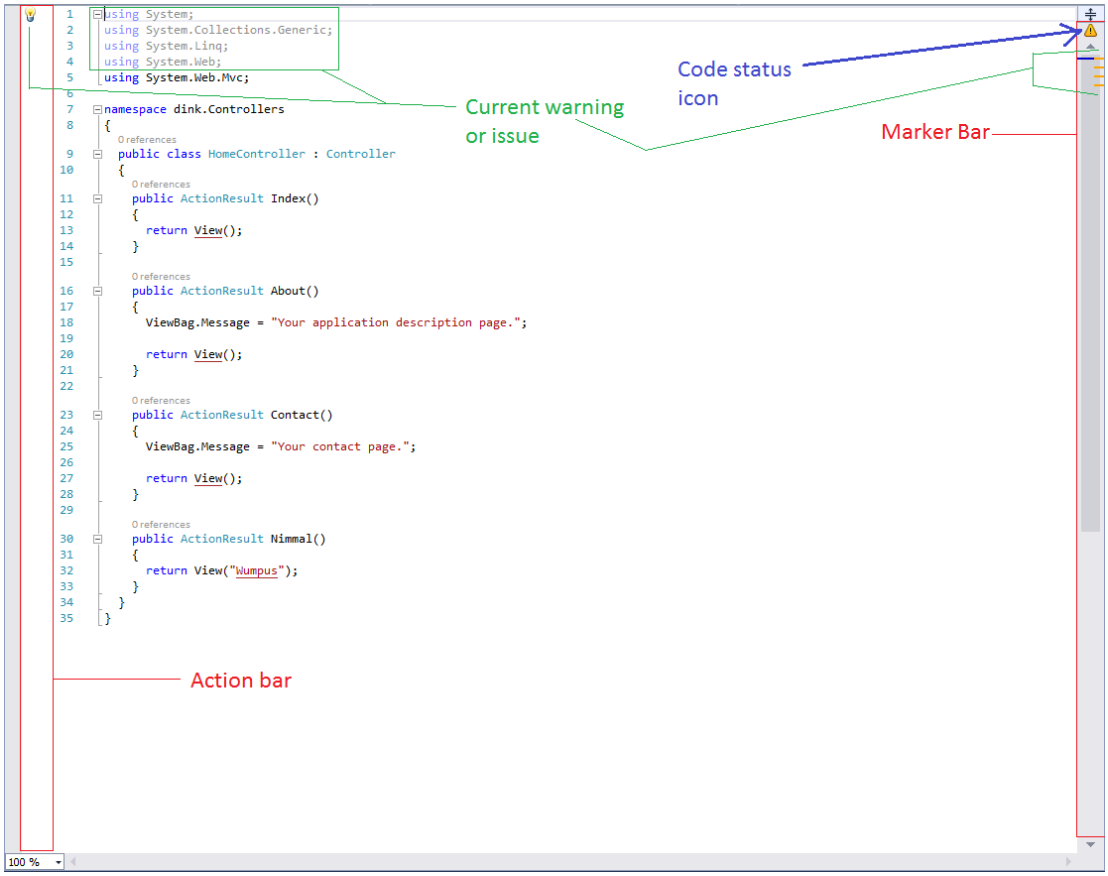


Figure 19: General code editor layout

As you can see in Figure 19, we have our action bar on the left, and the marker bar on the right.

In this particular bit of code, you can see we have a light bulb icon showing inside the action bar, because we have a possible warning situation where our cursor currently is. The warning is an “Unused imports” warning, and R# colors the unused imports in a different color than those that are used, so you can see immediately which are not used.

R# also goes one step further: not only does it effectively shade the code, but it provides the light bulb icon, which means you can press the main R# keyboard combination of **Alt**+**Enter** to remove the unused imports. You can also click on the light bulb and use the pop-up menu to select the action you wish to take.

You'll also notice that the marker bar has four yellow bars near the top of the document, one for each unused import line, with the Code Status icon showing as a yellow triangle.

The items mentioned here are the general changes you'll see in your code windows. Depending on your settings and the severity of warnings and other issues, you may also see different colored squiggles under various parts of your source code.

These squiggles work in exactly the same way as the native VS ones, except as with the marker bar, they take on different colors for different statuses. Like in Visual Studio, red generally indicates an error, green indicates an improvement, yellow indicates a warning, and blue indicates a recommended practice.

We'll cover the different colors in more detail later. For now, other things you may notice are various underlined items that were not previously underlined.

For example, Figure 19 shows an ASP.NET MVC controller from a quick-and-dirty hack I put together to help demonstrate things in this book. You'll notice that the view functions and view names are underlined.

What gets underlined depends very much on what type of file you're viewing and what the project type is. In CSS files for example, you'll get underlines matching the color of a color tag underlined, and using the R# key board shortcuts on them will allow you to pick the color from a color chooser built right into your editor.

Again, as we progress through the book, we'll cover these and others in more detail. To wrap this chapter up however, there's one final thing we need to look at.

Keyboard shortcuts and ReSharper's master key

I've already mentioned it a few times, but now it's time to give it a little more of the limelight. If there is one key press combination that you will use every day that you use R**#**, it's **Alt**+**Enter**. This one combination is R#'s master shortcut key, and whenever or wherever you are in your source file, if there is an action that can be performed, then **Alt**+**Enter** will make this happen.

You can also press **Alt**+**Enter** twice in rapid succession (known as a double-tap among some of us) to action and select the default action. Looking back at Figure 19 for example, if you were to position your cursor on a shaded unused import line, and press **Alt**+**Enter** twice rapidly, you'd fix and remove all four warnings in one go.

There is, however, a warning to be noted here.

Be very careful, especially when you first start using R#. There have been many times over the past six years where I've had to be ready with Ctrl+Z to undo some action, simply because I've been too quick to react to something R# has pointed out to me, and realized later that I should have been paying more attention.

However, the fact remains that 90 percent of all of R#'s immense functionality is accessed from the various keyboard menus and shortcuts. R# is designed to keep your hands where your developer's brain needs them, and that's on your keyboard.

Without its many keyboard-accessible tools and shortcuts, R# would be nowhere near as powerful as it is, and it certainly wouldn't help you make the productivity gains that are possible. The best bit of advice I can give you is to learn the shortcuts—learn them well, and learn them early. If you do, you'll be amazed what you start achieving in a very short space of time.

Chapter 2 ReSharper as a Programmer's Aid

As you've already seen, R# integrates perfectly fine into your general developer's workflow, but as of yet we've not actually mentioned much of exactly what it does or indeed can do in this workflow.

Looking back at Figure 19 in the previous chapter, if I were to position my cursor on one of the shaded import lines, then press **Alt**+**Enter** and expand the menu, I should see the following:

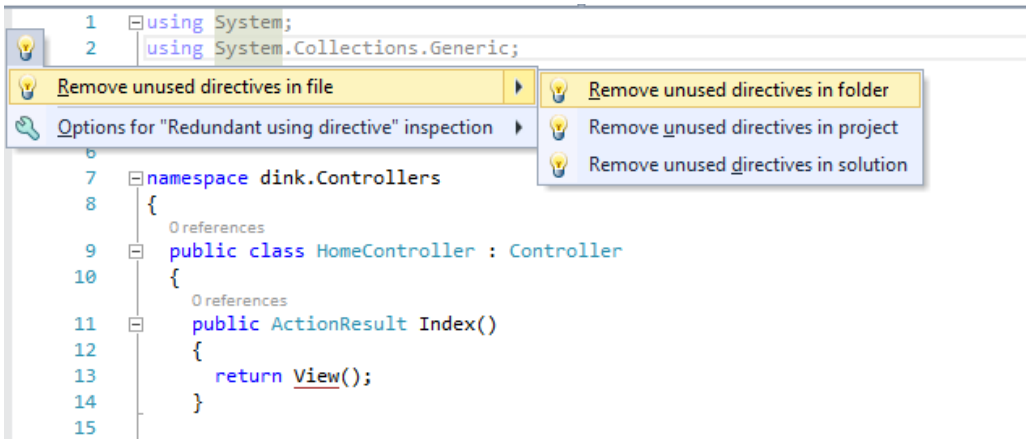


Figure 20: Expanded unused import action menu

Just selecting the first menu option on the first menu would remove only that unused import line, but as you can clearly see, there is also a submenu that allows you to make this change in all files in the same folder, in the entire project or even your whole solution. In this case we're using the removal action. Many actions are only applicable to the immediate line they are placed on, and only a select number of actions have the ability to affect the entire solution. The entire process is known as *Fix in Scope* in the ReSharper documentation.

There is a list of available actions for these Fix in Scope menus on the JetBrains website.

(Note: The list in our example is for version 8; once version 9 is released, there may be more.)

If you click **Alt**+**Enter** followed by a couple of arrow key presses, you can potentially rid a solution with hundreds of files of every unused import warning in about three key presses.

You can also configure how R# responds to unused import warnings. If we expand the second option in the same manner, we should see something like the following:

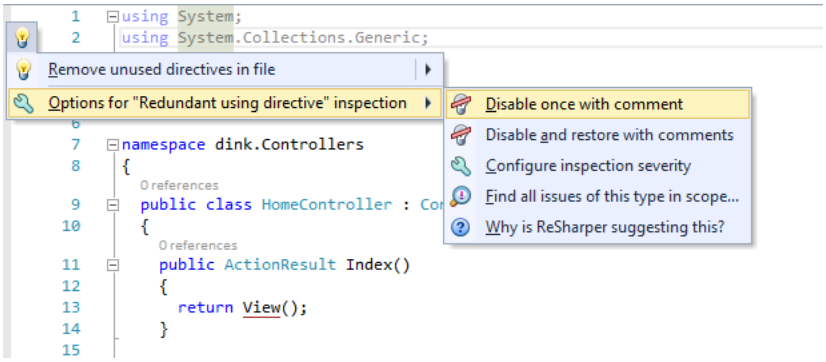


Figure 21: Unused import actions menu inspection options

If you select **Disable once with comment**, the line in question will get wrapped with a specially formatted comment that R# recognizes. This comment will prevent R# from flagging the line as an issue to address, but this will apply only to the file in which the comment is placed—it will not affect inspection of the same issue in other parts of the solution.

The **Disable and restore with comments** option allows you to enable and disable the notification, if you just want to disable it temporarily, for example. This saves you having to delete the lines in the first option manually).

If you select **Configure inspection severity**, you should see the following dialog box:

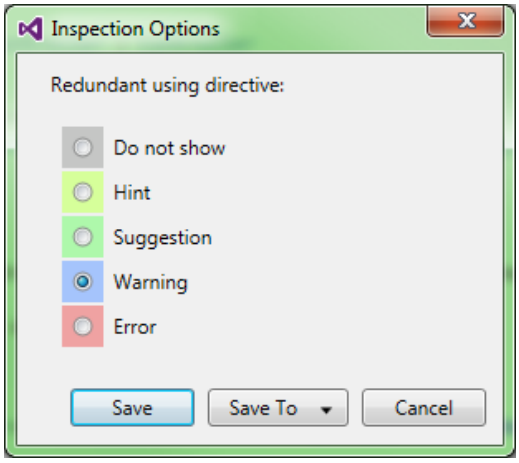


Figure 22: Inspection severity options

This dialog box allows you to permanently set the global options as to how R# notifies you for the particular case being looked at. For instance, if you were happy to have unused imports left in your source code, then you might select **Do not show;** you will not be notified again about this particular case. If you mark something as an **Error**, then you'll get the red squiggly underline that's commonly seen when other errors are flagged in your source code.

Hints and **Suggestions** are the green underlines for possible improvements in your code, but these won't make any difference in operation if you choose to ignore them. Finally, **Warning** shows the issue as a warning in the ReSharper errors list.

How you choose to mark an issue is entirely up to personal preference. If you're a team lead however, this is the first step you may want to take, to force your team to adopt a given set of standards (more on this in just a moment).

The last two options on the menu are **Find all issues of this type in scope** and **Why is R# suggesting this?** The first allows you to do a regular search operation just as if you'd pressed **Ctrl + H**, but instead of searching for a phrase or pattern, you only search for the issue that you're currently inspecting.

Why does R# give you this option if it already gives you the ability to solve them all with a single key press? Well as I've already pointed out, there may be cases where you know you're going to be importing a library, for example, but for now you just want the import statement left there. In this case you might wish to find some of the issues, wrap them in comments, then go back and run the option to correct any other warning of that type.

By doing things this way, you can search out those that you want to treat separately with great ease, and then you a global fix on the rest. Remember, R#'s sole purpose is to make you more productive, and even though this feels like something minor, it could potentially (like many other R# options) save you more manual work that you realize.

The final option, **Why is R# suggesting this?**, is just a helpful aid to the developer designed to aid in the decision process. In our example it's only an unused import. However, it could be some other issue that the developer in question doesn't have a great deal of experience with.

If this option is selected, your default browser should open and take you to an explanation page on the JetBrains website describing the ramifications of the selected issue that looks something like this:

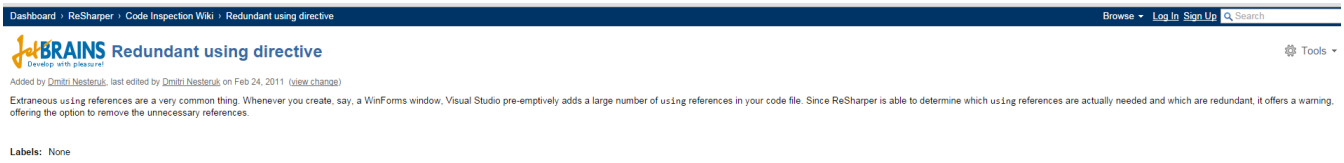


Figure 23: Issue description on the JetBrains website

For our unused import issue, there's really not much to say, but other inspections have much more inside information available, and as you start to use R# more you may be surprised by some of the tips you pick up.

Revisiting the inspection severity options

If you recall when I showed you the inspection severity options (Figure 22), I mentioned that as a team lead this might be your first port of call in setting up some inspection options for your team's projects.

Well, if you're like me, when you first saw this, you might have thought "I like this idea, but it's going to take a lot of work to set it up." I initially thought this because I figured I'd need to find the options for every inspection I was interested in, then go through this single dialog box and make a decision on each one.

You'll be pleased to learn that you don't need to do this at all. If you open the R# options as we did previously, then go to **Code Inspection** > **Inspection Severity**, you'll find that you have every inspection that R# knows about right at your fingertips.

You can expand the trees of issues for a given file type in the right-hand pane, and configure how that issue responds to Visual Studio in exactly the same way you would have from the individual dialog box.

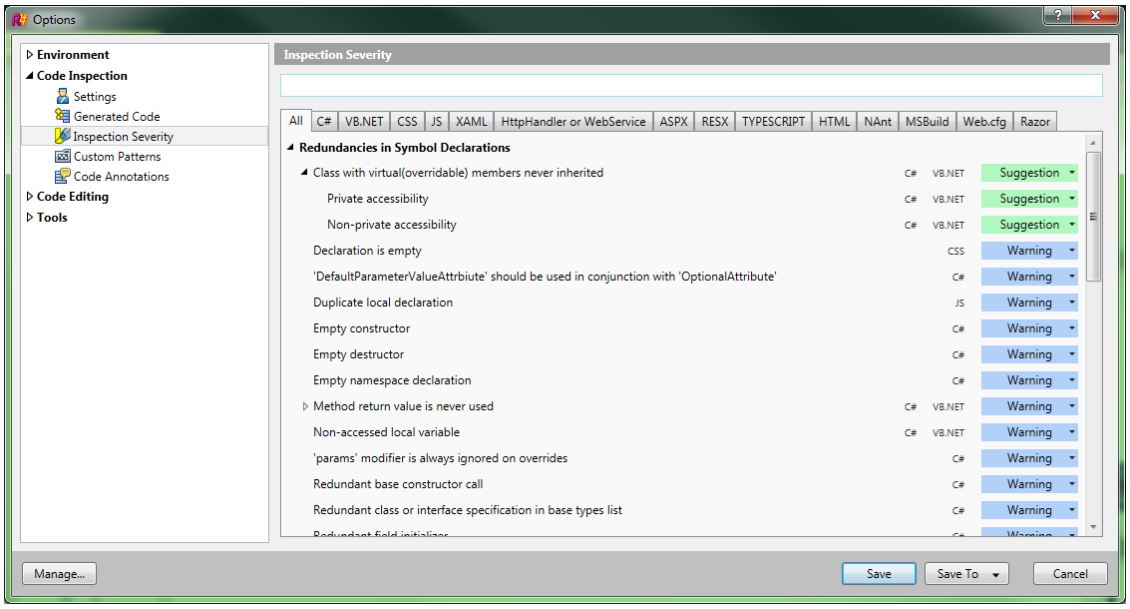


Figure 24: R# Inspection severity options

If you look closely at Figure 24, you'll also notice that there are two Save buttons on the bottom of the options dialog box. The first is just a regular Save button, and saves all your chosen options as the default options for your installed copy of R#. The Save To button, however, has more options.

If you click **Save To**, you should see the following three options:

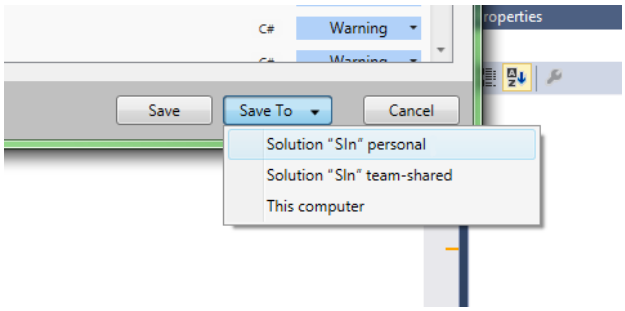


Figure 25: Save To options for the R# options dialog box

Selecting the first option saves the options directly into your active project. This allows you to have different settings for different projects, and if R# finds these settings when you load a project, it will use what it finds to override the current settings for that option.

The second option saves the file in the solutions directory alongside the existing solution files. The intention here is that this file is saved in your project's source repository so that all team members automatically have the same settings.

The last option, **This computer**, saves the file in the default place that ReSharper would normally save its settings; these settings can then be exported using **Manage Options** (found on the main ReSharper menu). Typically, exporting options allows you to spend time setting up collections of settings and file them away to reuse them in the future—simply load them when starting a new project.

The options for saving to different stores are the very thing that allow a project lead to enforce a given set of standards, giving them the option to maintain control over the generated code. Remember too, that when you save these option sets, it's not just code severity settings you're saving; you're saving any options changed from the defaults in ReSharper.

Which inspections can ReSharper do for me?

To close this chapter, I'm going to show you just a few of the different code inspections you're likely to see regularly. Producing and describing an entire list of every single thing R# will look out for is way beyond the scope of this book—there are so many things that R# can watch for, I could easily fill the rest of this book, and quite possibly another two!

You've already seen one of the most common ones, the unused import; this is an easy one to understand. If you do a lot of ASP.NET MVC programming, another one you'll likely see often is the missing view inspection:

```
0 references
public class HomeController : Controller
{
    0 references
    public ActionResult Index()
    {
        return View();
    }

    0 references
    public ActionResult About()
    {
        ViewBag.Message = "Your application description page.";
```

Figure 26: ASP.NET MVC Controller with a missing Razor view

In this case you can see that rather than underlining it with a squiggly line, it actually displays the keyword in red.

Note: In Figure 26, you can see a line that reads "0 references". This line is not a part of ReSharper, but part of a Visual Studio 2013 Ultimate feature, called CodeLens.

It all still works the same way though. If you position your cursor on the word and press the default R# key (**Alt+Enter**), you should get the following menu:

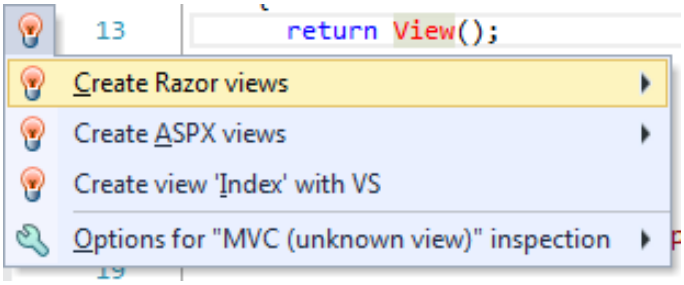


Figure 27: Inspections menu for a missing MVC razor view

If you expand on the various options using the arrow keys, you'll see that you get straightforward options to create partials, create a view with a layout, and all the normal options you're used to.

You can also select **Create index with VS** to get the normal Visual Studio dialog box up, and you have the same options for notification as you saw previously.

You'll also see that in this case the light bulb in the menu is red. This is showing you that this is a serious error, and if left unattended, it may cause your application to break when deployed.

If you go ahead and fix the issue by creating a view, then come back to your code, you might notice that it now looks slightly different:

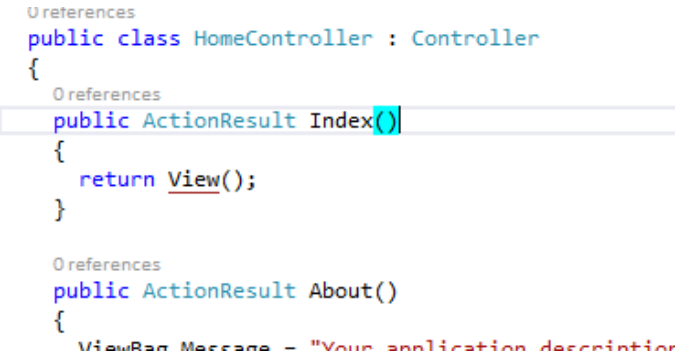


Figure 28: MVC Controller code after fixing view issue

You'll now see that you have a solid red line under it now. Don't worry though, this is not a bad thing; what you're looking at now is ReSharper's view navigation feature, which allows you to quickly navigate directly to a controller's view with minimal effort.

One other thing that R# does for you is to provide extra navigation features. If you position your cursor on the **View** keyword, but this time instead of pressing Alt+Enter, press **F12** (or hold **Ctrl** and click with your mouse), you should get the following menu:

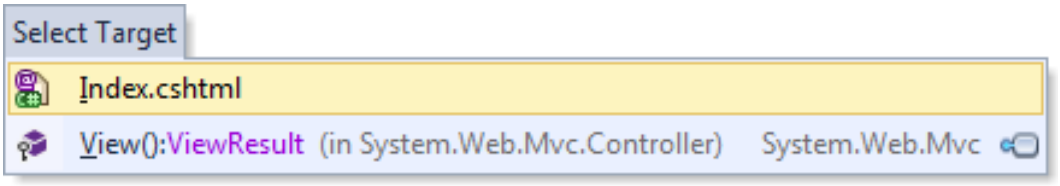


Figure 29: R# F12 menu on an MVC View keyword

If you then press **Enter** on the first of those options, you'll be taken directly to your Razor view code. If you select the second, then you'll end up in the object browser right on the appropriate view result assembly in the .NET Framework.

This solid underline can also be found in other places. For example if we take a look at our Razor views, specifically where we generate links, we may see something like this:

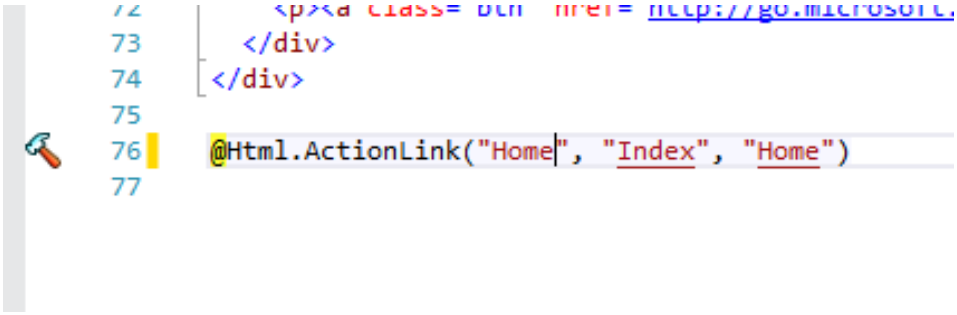


Figure 30: R# solid underlines in a razor view

You'll see immediately when you move over this, that you have a menu icon in the left-hand column. As before, the left-hand icon tells you that you have an Alt+Enter menu available; however, if you also position your cursor on one of the underlined options and press **F12**, you'll also be taken straight to the controller in question that this link would refer to.

Where you see solid underlines, you'll likely have operational choices too. These choices range from navigation aids to procedures that you might commonly want to do, such as splitting a long string into two separate strings. The key thing here is that if it's solid and not squiggly, then it's not likely to be an error; if it's a squiggly line, then depending on its color it will be either a notification, hint, warning, or an error.

To give you an example of a green squiggly, the following image shows a snippet of code from a default account controller produced using the ASP.NET MVC 5 web application default template:

```
10  {
11      [Authorize]
        3 references
12      public class AccountController : Controller
13      {
            0 references
14          public AccountController()
15              : this(new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext())))
16          {
17          }
18
            1 reference
19          public AccountController(UserManager<ApplicationUser> userManager)
20          {
21              UserManager = userManager;
22          }
23
            16 references
24          public UserManager<ApplicationUser> UserManager { get; private set; }
25
26          //
27          // GET: /Account/Login
```

Figure 31: Snippet of code from the ASP.NET MVC account controller

You'll see that the `public` keyword in the two cases shown is underlined in green; this would mean that you're looking at either a hint or a suggested improvement.

If you position your cursor on it and press the default R# key (Alt+Enter), then you should see the following menu:

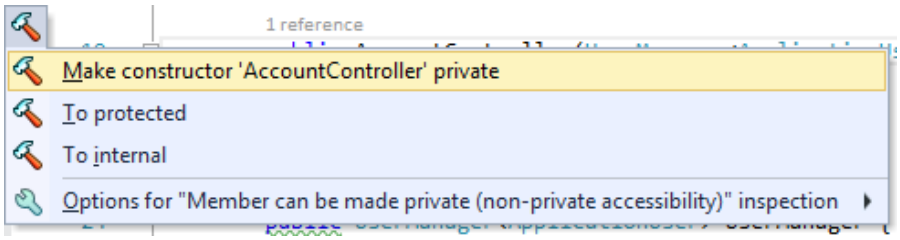


Figure 32: R# suggestion menu for the public keyword

If you actually hover over the option prior to opening this menu, you'll see that R# tells you it is possible to make the constructor private, and so the menu reflects this with appropriate choices for actions you might want to take.

In this case R# has analyzed your project source code, and realized that the constructor is never called from any external code. This means that you can seal that method away, or even remove it all together. Inspections like this help make your object-orientated code much better by removing public hook points that may never get used, or identifying code that may be surplus to requirements, leading in the end to a much simpler code base.

R# doesn't only make you more productive, but after a while you'll also find that you start to spot these good practices yourself and actively remove them—or even not add them in the first place. In essence R# actually trains you to be a better developer.

And there we go—that's the 100-foot overview of R#. Going forward from here, we'll start drilling down into the various menus available from the Visual Studio toolbar and take you on a tour of the specific R# tools that are at your disposal.

Chapter 3 Navigation Tools

In this chapter we'll be taking a more in-depth look at the tools provided on the ReSharper Navigate menu.

If you expand the main Navigate menu by going to **RESHARPER** > **Navigate,** you should see the following:

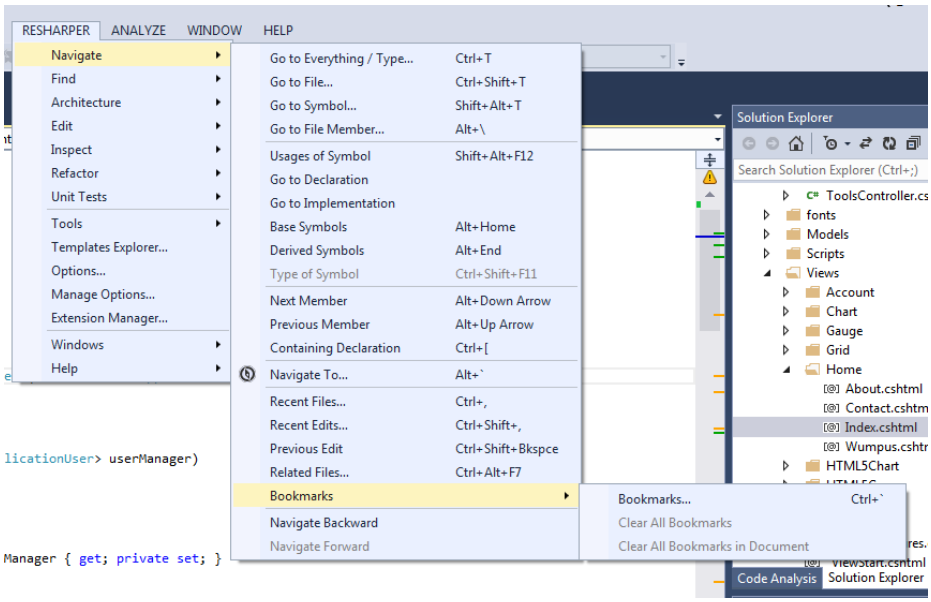


Figure 33: R# Navigation tools menu

As its name suggests, the navigate menu and its options are there to help you get around your solution in the quickest possible way.

You've already seen some of the navigation features in the last chapter, when we used F12 to move between our views and controllers. F12 however, just like it does in standard Visual Studio, will still take you to the definition of a method, variable, or class.

Give it a try: place your cursor over a variable or method in your application and press **F12**; you'll see it behaves just the same. However, you should also try it with something for which you don't have the source code available. If you do this, you should find that rather than doing nothing as most versions of VS do, you should be dropped straight into the Object Browser on the object or interface that you are examining.

Note: I am aware that in some versions of Visual Studio you do get better navigation features. For example, if you try this on an MVC Attribute in Professional or higher VS2013, then you'll actually get a reconstructed source class, which VS builds for you on the fly. Many of these advanced features of VS are only available in the higher-end

products, so for the purposes of this book, the fact that many versions don't do anything unless you have the source available stands true.

F12 is just the tip of the iceberg when it comes to R#. I've loaded a more complex solution into my copy of VS2013 for the rest of this chapter; if you really want to explore the navigation commands fully, I'd strongly advise you to do the same. The project I've loaded contains nine separate projects, eight class libraries, and an ASP.NET MVC front end. If you don't have a solution with quite a few moving parts, you might find that many of the menu options remain disabled. Even with the solution I have loaded, I still get dimmed options on some of the menus.

Go To Everything

The first four options on the navigate menu are all derivatives of "Go to…", and as developers these are a set of go-to's you'll be pleased to see.

The first one, **Go-to Everything/Type**, is kind of the master go-to. Position it over a type, method, enum, or class, and then press **Ctrl+T** or navigate the menus from the VS main menu bar. You should see one of R#'s now famous pop-up menus appear:

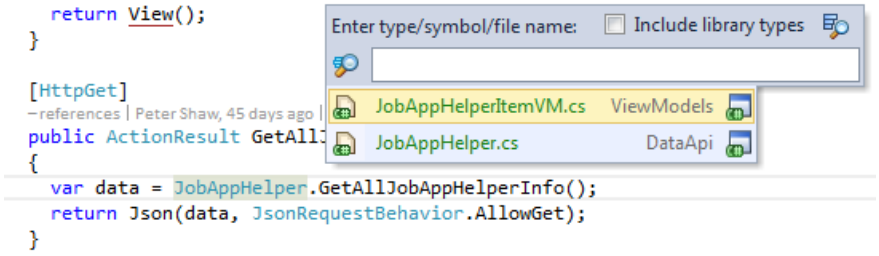


Figure 34: Context-sensitive Go-to everything/type pop-up menu

In this example, I positioned my cursor onto the `JobAppHelper` call in my MVC action. I then pressed **Ctrl+T** and ReSharper immediately opened up a menu that shows me two things.

The first is a list of my two most recently used items, in my case my two JobApp helper files.

The second is the search box; anything I type in here will be filtered to a list of options containing the phrase I typed. If I type "JobApp" here, then all my classes, objects, and models containing JobApp in their name would be listed.

Using Ctrl+T essentially gives you another of R#'s specialized search tools. Just as we saw in the previous chapter when looking for code issues to correct, the Ctrl+T key press gives you the option to search for type and member names, all while maintaining a certain level of wildcard searching (just in case you're not quite sure how to spell what you're looking for). Because it's context-aware, it will also look at its surroundings, and suggest things you may be looking for based on your location in the code. If your target is in the suggestions list, a simple click on it will take you straight there.

The second option in the group, **Go to File**, and again the familiar R# menu pops up, this time allowing you to search for file names in your project:

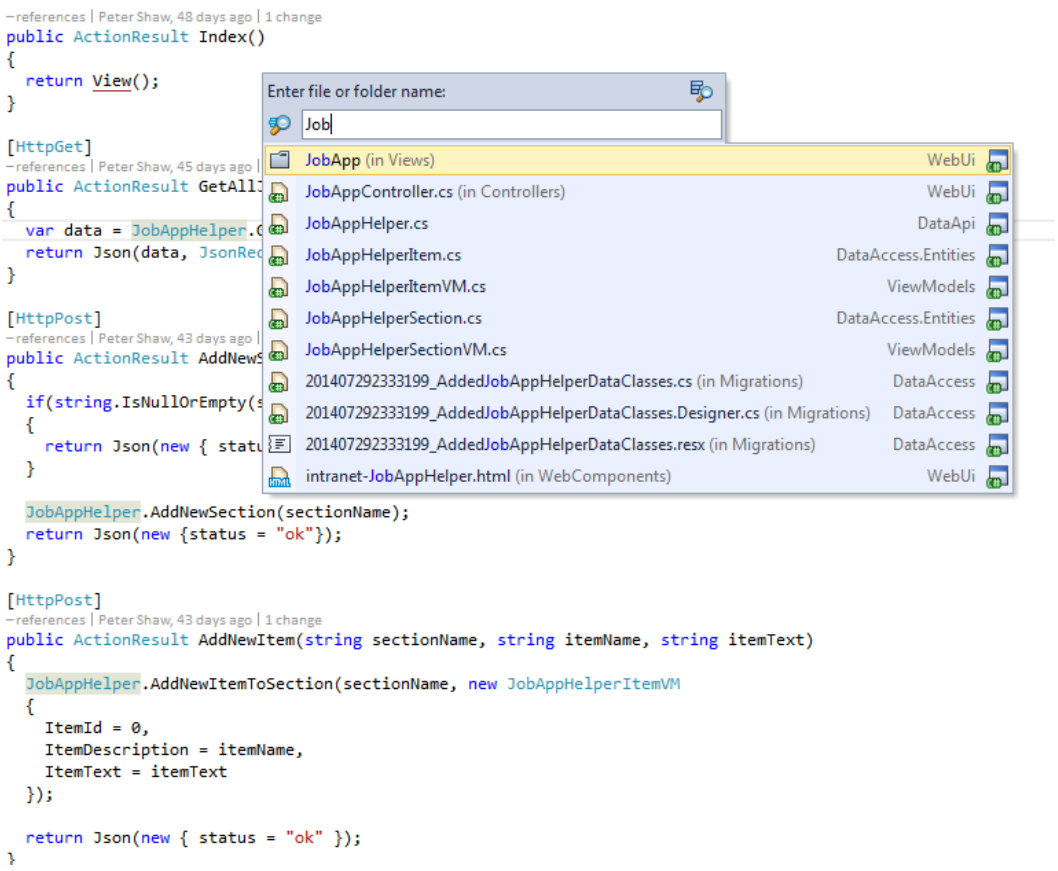


Figure 35: R# pop-up menu for Go-to File

In Figure 35, you can see that I've started to type “job,” and R# has presented me with a list of possible file matches. If I select any of the matches, that file is then immediately opened within the VS editor. Just as with other areas in Visual Studio, you can also use abbreviated searching. This means, for example, you can enter “JAC” (or “jac” in lowercase), which form the initial letters of `JobAppController`, and anything matching will be displayed.

The third option, **Go to Symbol**, opens up yet another pop up on the press of Shift+Alt+T. This one allows you to search for symbol usages.

For example, if you have a variable (private or otherwise) in a file in your solution called `MyJobCount`, and you use the Go to Symbol navigation option, the location where this variable is defined will be displayed in the search results, allowing you to go directly to that location.

I've found this most handy when, for example, I remember writing a bit of similar code, but can't remember the full method or class name. With Go to Symbol, all you need to remember is something in the method or class definition and R# will do the rest.

The final go-to is the **Go to file member** command. This command simply allows you to navigate to methods and other definitions within the currently opened file.

If I open this on one of my controllers for example, I get the following:

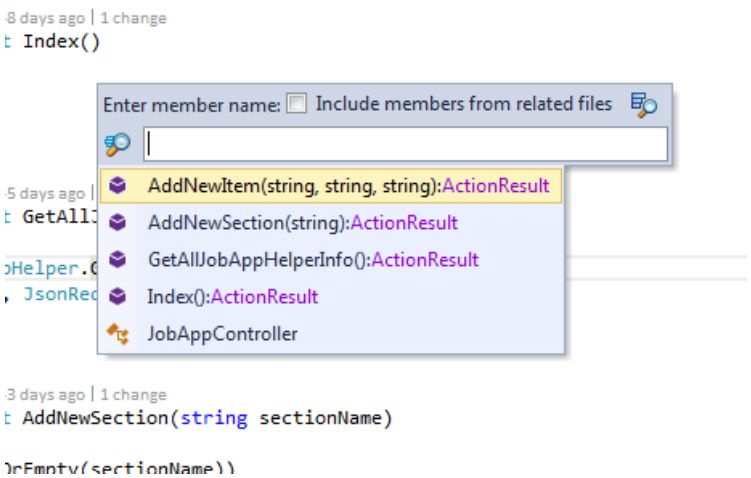


Figure 36: Navigate to Go to File Member pop-up menu

In my case the controller is a very short one, but imagine if you were dealing with a controller from a large e-commerce site—you can easily and efficiently jump from action to action using Alt+\ and a quick wildcard.

The next group of commands on the menu are all related to our F12 scenario.

Usages of Symbol (Shift+Alt+F12) will list places where the symbol currently under the cursor have been used. The following example shows this command being executed on an Entity Framework DbSet declaration:

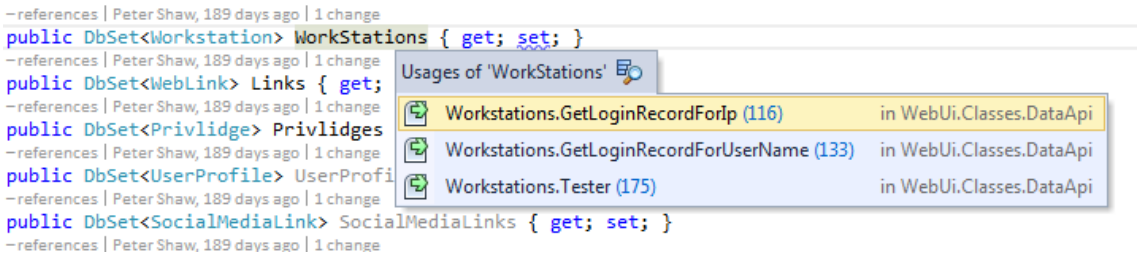


Figure 37: Usages of Symbol pop-up menu

You can clearly see the usages of that data table with the exact line numbers and file name where they are used, and as with all the other menus, clicking on one will jump to that location.

Go to Declaration and **Go to Implementation** are both single-shot commands; if you position your cursor onto a type definition, the former will take you directly to the file and location where that type is defined. The Implementation derivative is similar, but that will take you directly to the implementation of a method within a type.

The **Base Symbols** command will list any symbols on a derived type that are in a base class.

For example, if I press Alt+Home on my Entity Framework data context class, I get the following:

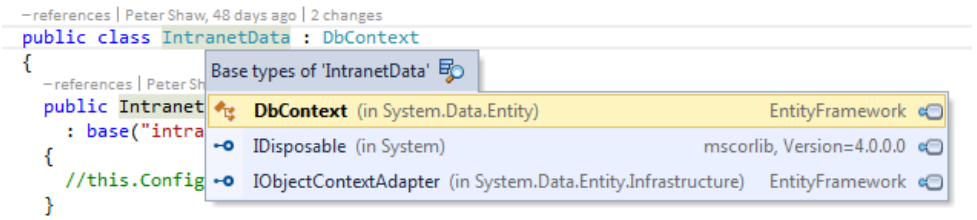


Figure 38: Base Symbols pop-up menu

As you can see in Figure 38, this allows me to navigate the types and classes that my data context is built on. If I select **DbContext** and the source code is not available, one of two things will happen. The first time you attempt to access an object, you'll be asked if you would like ReSharper to attempt to either obtain the source code, or decompile it, or you'll be asked if you wish to use the object explorer as shown in Figure 39. If you've previously made this choice, then you'll automatically be given that choice the next time.

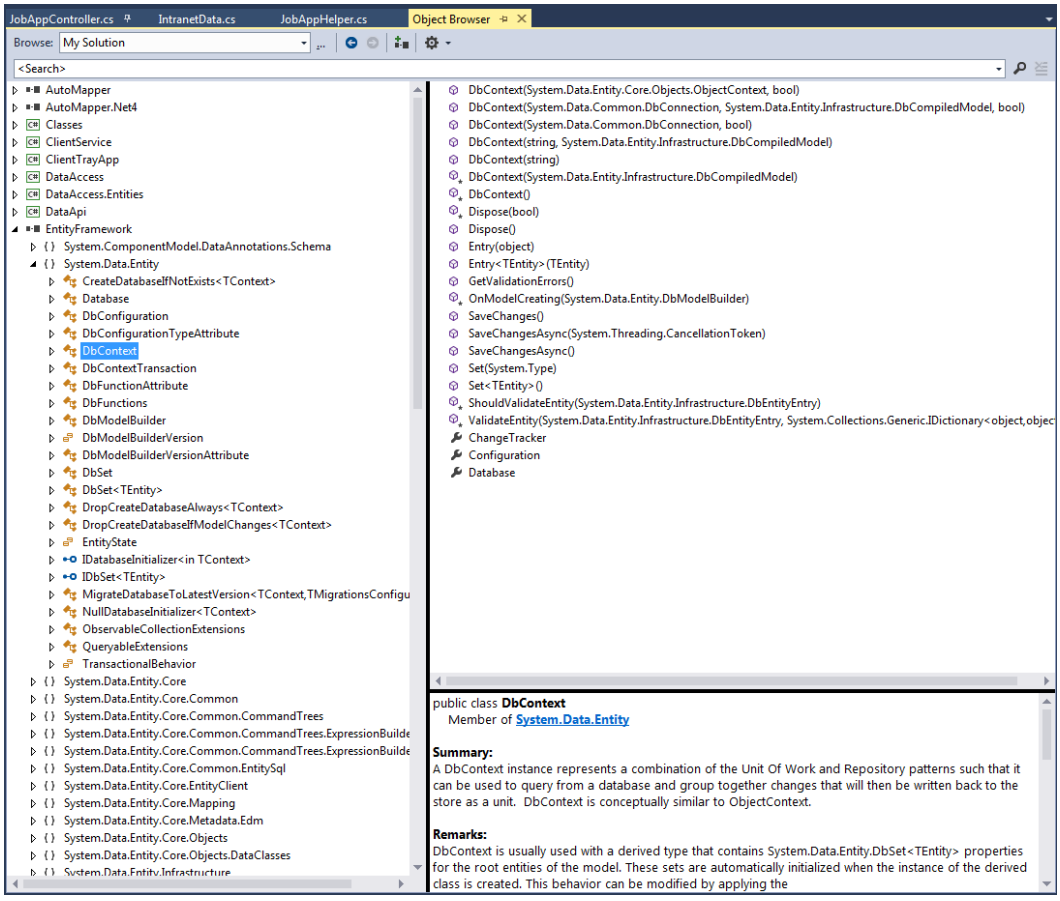


Figure 39: Object browser opened by the menu selection in Figure 38

Whichever choice you get (which by default will be the source code decompiler), you can easily change the default options in the appropriate section of the ReSharper Options dialog box; since the decompiler is provided by default with all versions of ReSharper, you will always have the ability to decompile assemblies that you're using. Object Explorer will often be the last on that list of options.

The opposite of Base Symbols is Derived Symbols; if you position the cursor on the **DbContext** in the previous example and press **Alt+End**, you'll be moved slightly to the left and positioned on the declaration for the data context.

If you had more than one data context in your solution, then pressing Alt+Enter over that symbol would have popped up a menu listing them both. Derived symbols will list any class or type that derives from the symbol or interface under your cursor and allow you to rapidly navigate to them.

The last one in the group, **Type of symbol** (Ctrl+Shift+F11), navigates to the type of the symbol underneath the cursor.

For example, in Figure 34, you saw that we could navigate and search for methods and classes in the type `JobAppHelper`. Furthermore, you should be able to see in the image that we assign the result of the call to a variable called `data`, which is defined using the C# `var` keyword.

If later on in the code you were working with the `data` variable, but were unsure as to which class it came from, positioning your cursor on the `data` part of the name and pressing Ctrl+Shift+F11 would take you immediately to the definition for the `JobAppHelper` class.

Beyond Go-To

Moving on to the rest of the Navigation menu:

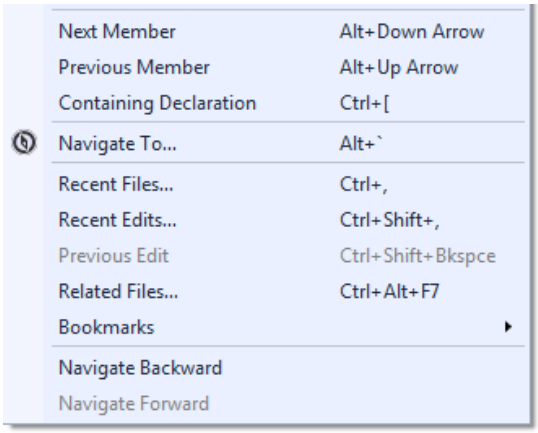


Figure 40: Remaining options on the navigation menu

Next Member and **Previous Member** will step you up and down in your source file, automatically positioning you on Variable, Property, and Method declarations. It's very much like using Alt+PageUp and Alt+PageDown to move from inspection to inspection, except you can immediately jump around the main blocks of your code without having to scroll.

Containing Declaration jumps you to the container of the current member you're positioned on. For example, if you have a class called `Customer`, and within that class you have a method called `FindAddress`, positioning your cursor on `FindAddress` and pressing Ctrl+[will position you on the class definition given that the class is the parent to the method, which is very convenient for navigating quickly to the top of the class.

Recent Files when invoked should produce a pop-up menu that looks similar to the following:

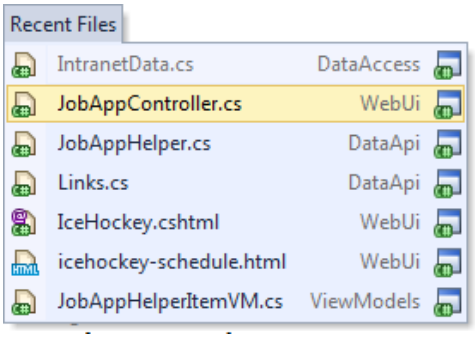


Figure 41: Recent Files navigation menu

This pop-up menu allows you to go straight to any of the files you most recently worked with.

The **Recent Edits** option is very similar, but instead of a list of files, it shows you your most recent edits across all files in the current solution:

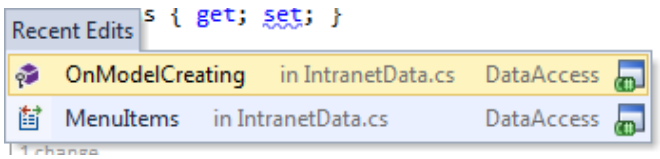


Figure 42: Recent Edits navigation menu

Ctrl+Shift+Backspace will take you back to the position of your most recent edit, allowing you to quickly backtrack and undo changes if you need to.

The **Related Files** option is mostly useful when dealing with web and Razor views, but can also navigate between WinForms code and its associated designer file, or between XAML code and any bound view models. In most cases where a file linkage exists in Visual Studio, ReSharper can quickly navigate between instances using this command.

If we browse to a Razor view in our project, then press Ctrl+Alt+F7, we'll get the following pop-up menu:

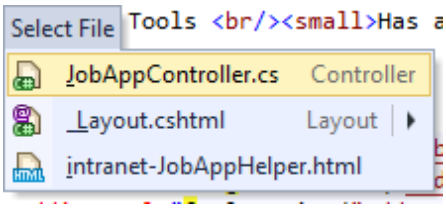


Figure 43: Related Files pop-up menu

Because this is a Razor view, R# has correctly identified that the file has an accompanying controller, and that it also has a master layout page. I'm also using PolymerJS in this application, so R# is also telling me that there's an HTML page associated with it, and also offers it as a selection on the menu. Selecting any of them using the arrow keys and Enter, or by clicking with the mouse, will immediately place you in that file ready for editing.

If you position the selection bar on the entry for the layout and press the right arrow key, you'll also see any files that are related to the layout file, which by virtue of the relationship to layout are also dependencies of the current view. In my solution, it looks like this:

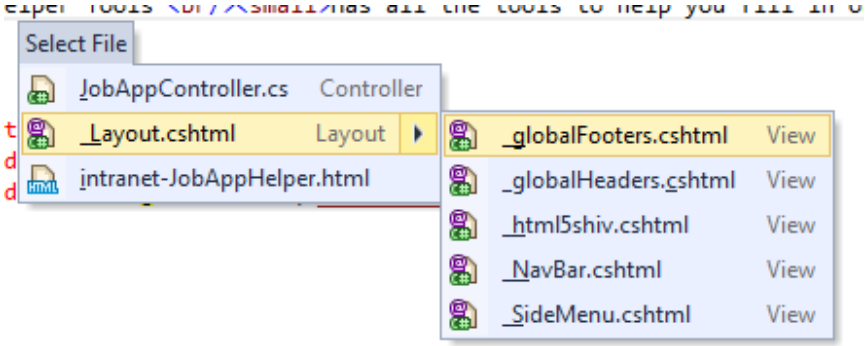


Figure 44: Sub-menu showing dependencies of the Razor layout file

You can easily see that my layout file uses many other partials, all of which I can easily bring to the front of my editing experience in little more than a couple of key presses.

The last three, **Bookmarks** and **Navigate Forward/Backward**, allow you to create bookmarked points in your code, and jump back and forth easily.

Navigate Backward and Navigate Forward simply look at your history, then attempt to work out the best type of navigation for you. If, for example, you just performed an edit and executed Navigate Backward, R# will choose to navigate to your last edit. If you were recently moving forward through your file looking at members in your class, then moving forward will repeat the move to next member command.

Like many of R#'s tools, the backward and forward commands are quite generic and context sensitive; this can be a source of frustration at times, as it can occasionally pick the wrong type of navigation. Once you learn how to tame them, you'll find everything gets much easier.

Bookmarks

One of the most useful options under the navigation tools that R# offers has to be the bookmarks facility.

Just as you would bookmark pages on the internet, or pages in that massive .NET book that's on the desk beside you, ReSharper allows you to do the same with ease.

If you expand the Bookmarks submenu, you should have a pop-up menu like the one shown in Figure 45:

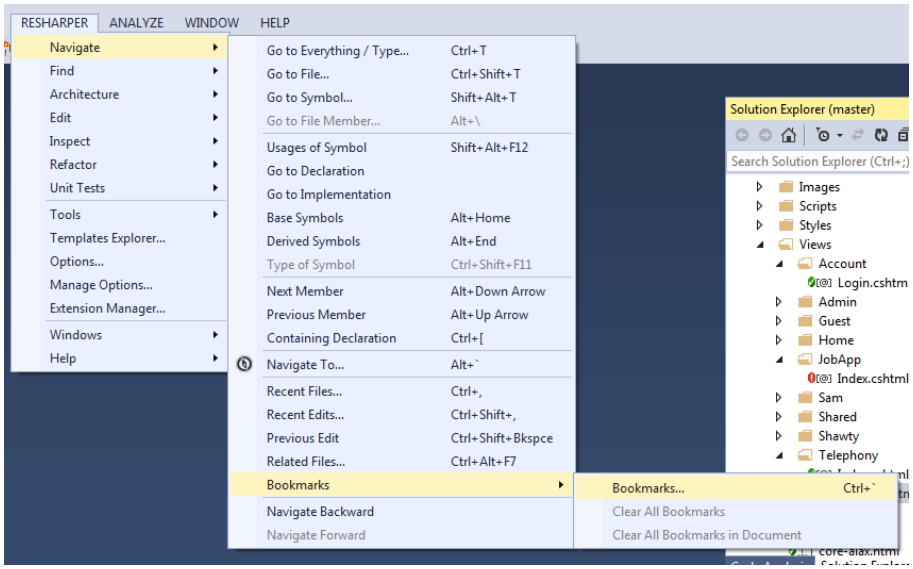


Figure 45: R# Bookmarks menu

Initially (unless you've defined some bookmarks already without realizing it), only the main bookmarks option will be available. Clicking on this should open the bookmarks pop-up menu, and should look something like the following:

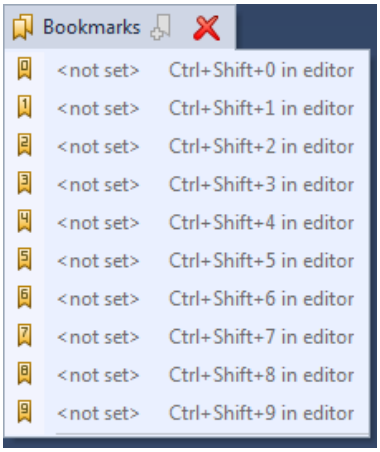


Figure 46: R# Bookmarks pop-up menu

You can define up to nine bookmarks by moving your cursor to the position you wish to bookmark, then pressing **Ctrl+Shift+*<0 to 9>*.** This will then place a small bookmark symbol in the left-hand column next to your source code as shown in Figure 47:

Figure 47: Code snippet showing bookmark

If you then reopen the bookmark pop-up menu, you should see the bookmark you defined appear:

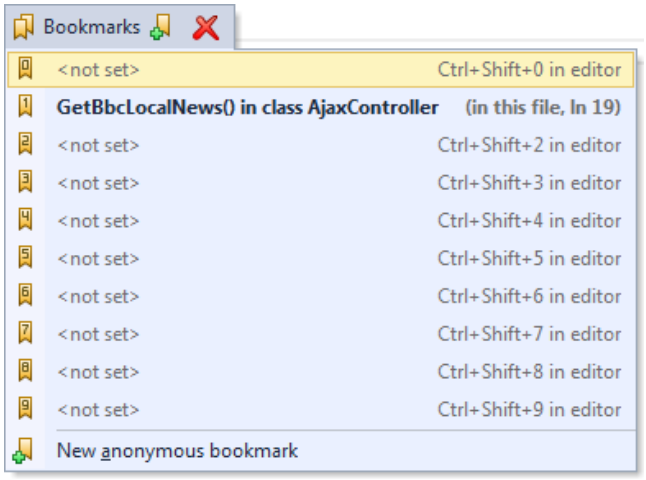


Figure 48: Bookmarks pop-up menu showing the newly defined bookmark

One thing you need to be aware of—when you create a bookmark, if you have the R# tool icon in the left column at the same time, you won't be able to see your new bookmark as you define it. Once you move your cursor away from the line and thus hide the tool or menu icon, you'll be able to see the bookmark. This caused me a fair bit of frustration and I kept pressing Ctrl+Shift+*Number* (thus toggling the bookmark on and off), believing that it wasn't working.

Moving to a bookmarked line and pressing the Ctrl+Shift combination will remove the mark, and moving to a different line and pressing the combination of an existing bookmark will move the bookmark to that position.

Finally, if you want to navigate to the bookmarks you have created, simply press Ctrl+*<0-9>*, where the digit is the number of the bookmark you wish to navigate to, or press Ctrl+ and use the pop-up menu to select the bookmark you'd like to navigate to.

And there we have it—those are R#'s tools for getting around your project code.

As you can see, a lot of thought has gone into exactly how each option behaves, and while there is some overlap, you can see that using the keyboard shortcuts is much faster than having to navigate to menus all the time.

The key here is practice. The more you use these key combinations in your work flow, the more natural they'll become. Use them alongside the regular Visual Studio keys, and before long you'll wonder how you did without them.

Before we move on, one random and useful key shortcut that should always be committed to memory is Shift+Alt+L (Locate in Solution Explorer), which will highlight the file you're currently editing in the Solution Explorer.

Chapter 4 Find and Editing Tools

You could be forgiven for wondering why R# has an entire section devoted to finding things—after all, most of the navigation tools we covered in the last chapter are designed for that, right?

The answer here is actually a yes and a no answer. Why? Well the navigation tools are designed to get you around quickly and efficiently. In a way, they do have to search for things, but that search is often very general and not specifically targeted to a particular point in time.

Yes, you can use the navigation tools to jump from method to method, but what if you don't know the method name or place to start?

That's where R#'s Find tools come into play.

Say you're jumping around between tabs that you have open, and you can't remember which tab a particular function is in, but you remember it starts with "abc." Most developers working in Visual Studio will instinctively press Ctrl+F to bring up the find.

R#'s Find tools work in a similar way; they are designed expressly to find something that you're not entirely sure of.

Finding Usages

In a complex project, you might have many dependent modules and layers, especially if you follow n-tier style architecture. Presentation layers might use business layers, which in turn might use data layers, and everything may be all cross-cut with security layers—and that's before we start thinking about things like attributes in MVC projects and NuGet dependencies across the board.

If you have a data library for example, you might want to know which parts of your solution it's used in. For this task you can use the Find Usages option.

Position your cursor on a method anywhere in your project. I've picked a class in my intranet application that uses Yahoo Query Language (YQL) to get the news local to my area. I want to see which parts of my project use the local news feed, so I right-click on the method declaration, and select **Find Usages** from my context menu:

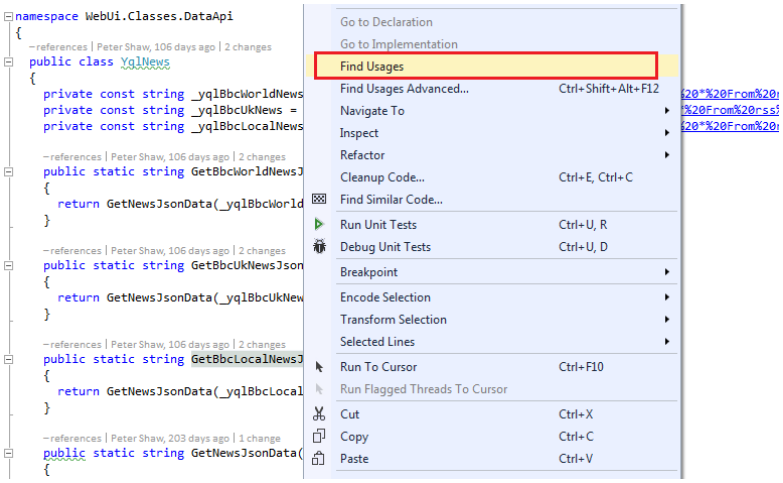


Figure 49: Selecting Find Usages from the Context menu

You can also use the main R# menu in Visual Studio by clicking on **ReSharper** > **Find** > **Find Usages**. Find Usages by default doesn't have a keyboard shortcut setup, but like everything else in R#, this is easily changed in the keyboard options.

You'll also note that there's a Find Usages Advanced menu option too; if you click on this one, you'll get a different dialog box altogether:

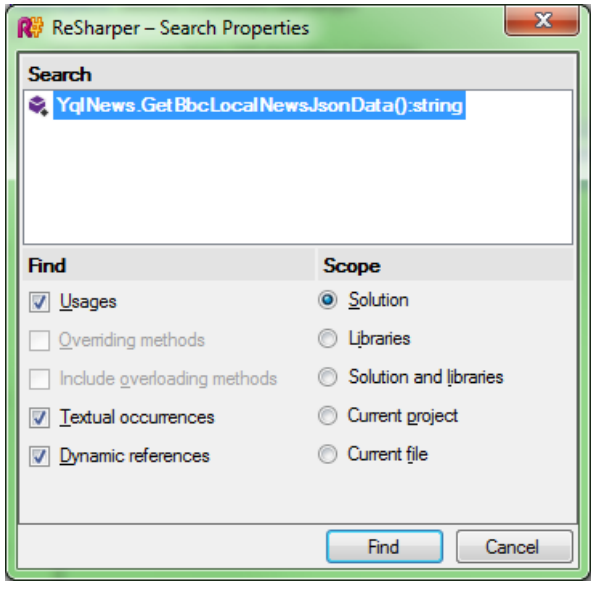


Figure 50: R# advanced find usages dialog box

By default, the Find Usages tool only searches for code-based usages at a solution level. The Advanced dialog box allows you to be more precise with your searches, and allows you to perform tasks such as searching in comments and looking for dynamic uses.

You can narrow the scope down too, so that rather than searching the entire solution, you can search only the given project. You can also look for overloads of your methods, which is especially useful when using a lot of abstract classes.

Moving on down the menu brings us to **Find code dependent on module**. This option is one for the NuGet fans among you. If you expand the references section of any of your projects, and right-click on any of the references in the tree, you'll see that this option also appears on the Context menu, as Figure 51 shows:

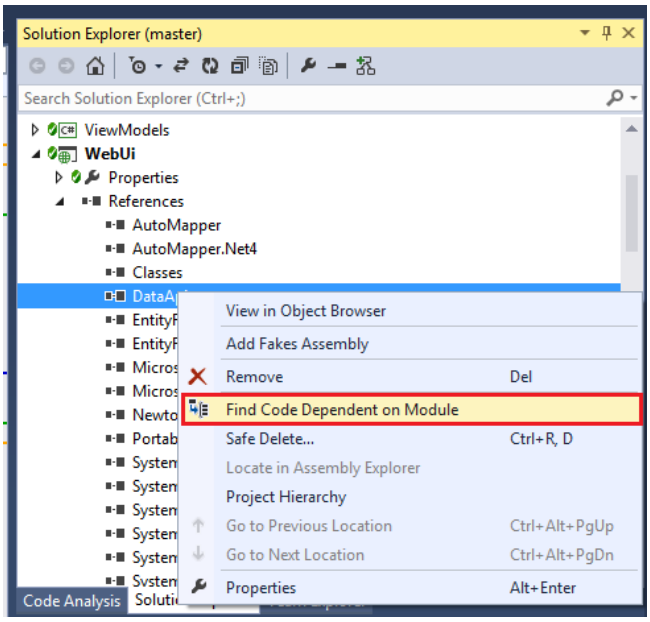


Figure 51: Find Code Dependent on module, direct from the Solution Explorer

When you select this menu option, R# will find all the code in your current scope that references the selected module in your references.

In Figure 51, I've right-clicked on my Data Access layer, so once I click the left arrow button to select the option, I'll immediately get a list of all the places in the current project where that library is in any way referenced:

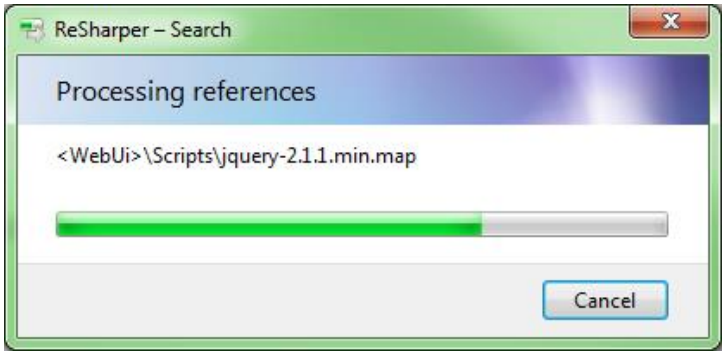


Figure 52: R# searching for references to the current module

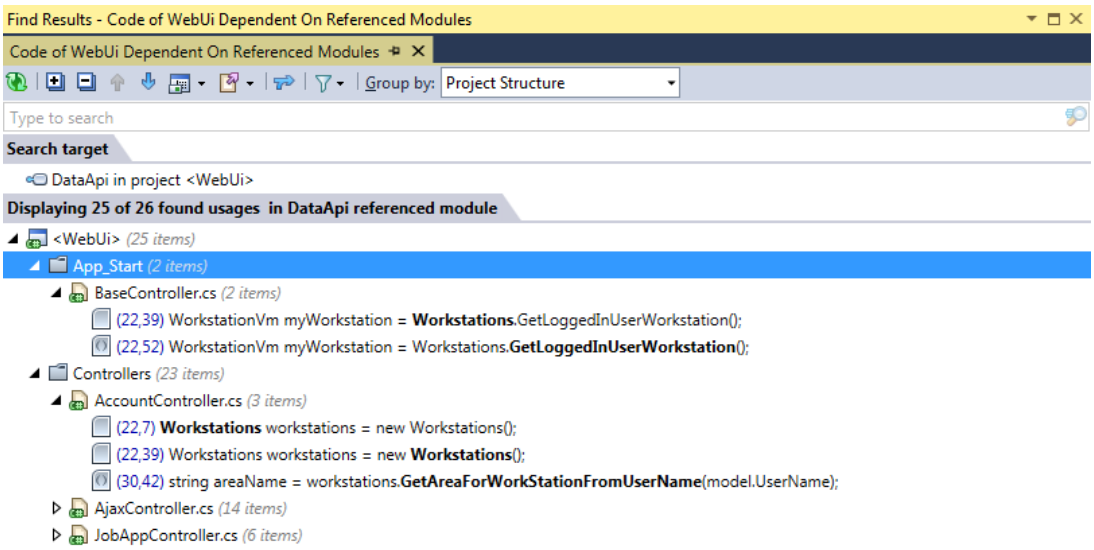


Figure 53: The results of the search that took place in Figure 52

Once the Find Results window appears, you can dock it into the Visual Studio environment just like any other panel, and double-clicking on any of the entries will take you directly to that location in your source code.

Multiple searches will also open multiple search result tabs in the same panel, so you can easily search on a number of different packages, then keep the lists open for rapid access.

The next option, **Find Symbols External to Scope**, works best at a project level. When used correctly this option will search for and list any external references to the selected file, module, or project. If we try this on the WebUI in my intranet project, you'll get a list of anything that the project depends on (and knows about) to function correctly.

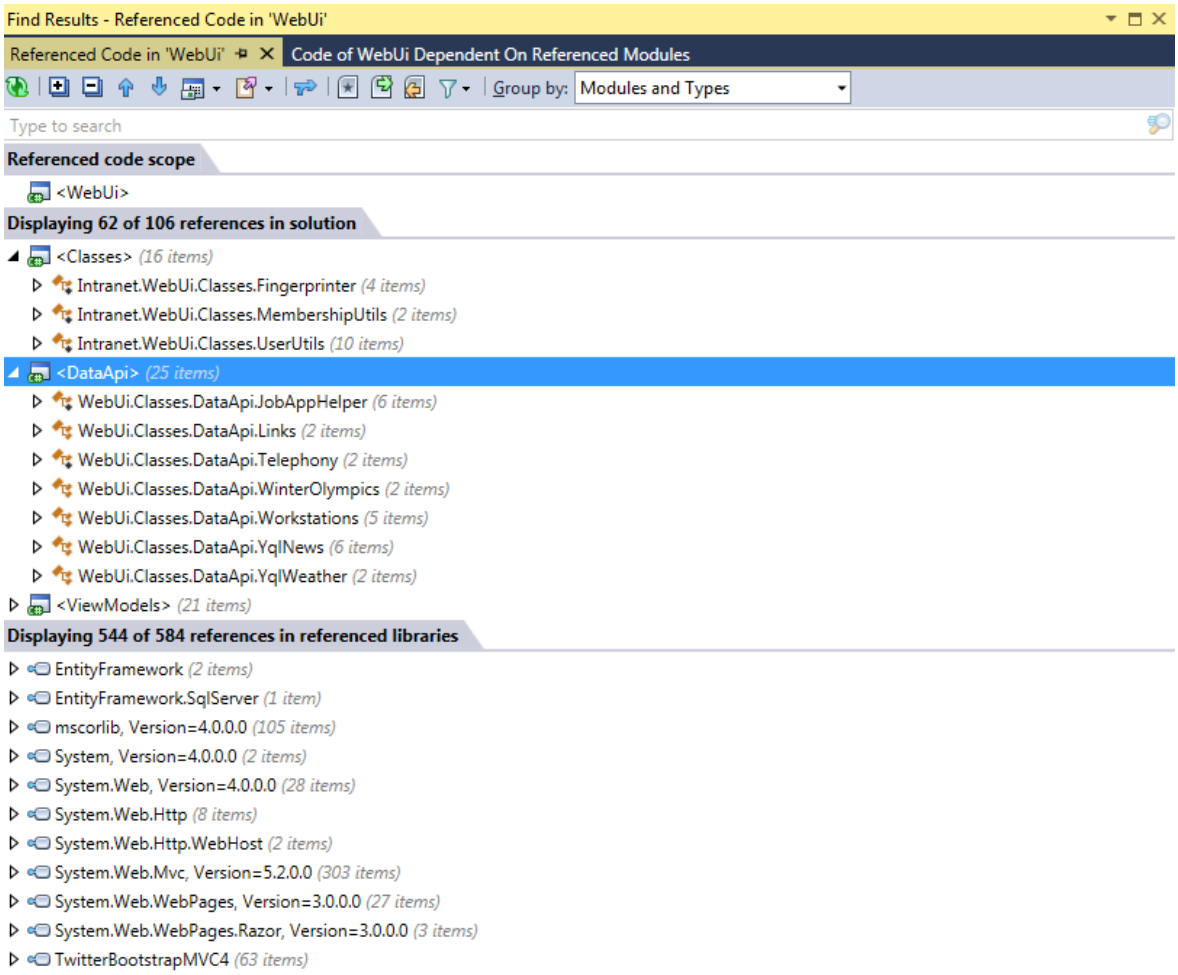


Figure 54: Results for Find Symbols External to Scope on the WebUI project

As you can see, I can instantly build up a picture of exactly which references are used and which are not, allowing me to easily remove redundant links and promote smaller, quicker builds.

The **Optimize References** option is designed to do exactly that, so that we don't have to do it manually. Right-click on the References branch of a project in your Solution Explorer, and select **Optimize References**. After a short while, you'll get a report showing you exactly which references are and are not used, how those that are used fit into the big picture, and which references R# believes can be removed. The result should look something like Figure 55:

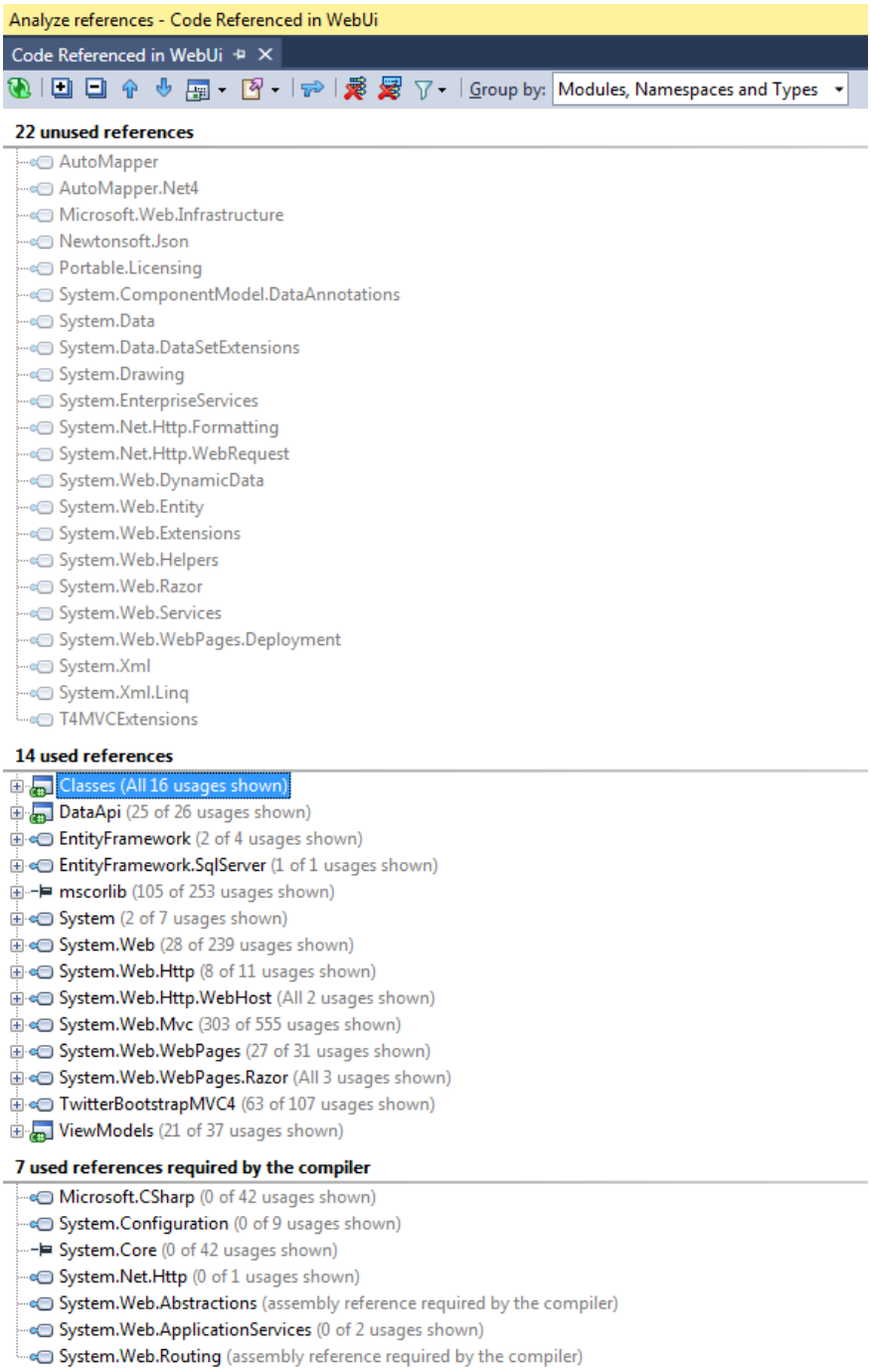


Figure 55: R# optimize references report

You can then use the various tools along the top of the report to organize and act upon R#'s recommendations.

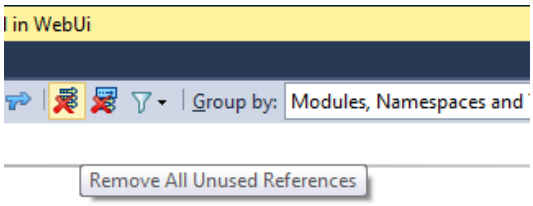


Figure 56: Organize References toolbar

Remember, though—this is not guaranteed to get everything 100 percent correct. Some libraries use some very clever tricks to get themselves where they need to be, and occasionally R# cannot detect their usages correctly.

To be safe, ALWAYS save your project and back it up before you make this kind of large-scale change. That way if anything does go wrong, you can easily go immediately back to where you started.

Pattern Searching (the Holy Grail of “Find all the things”)

The final search tool has to be one of the most powerful search tools I've ever seen in any code editor or IDE. I generally don't do meme culture, but in this case it's absolutely justified:

Figure 57: FIND ALL THE THINGS meme

Pattern searching goes above and beyond searching for simple strings, and instead introduces you to a whole new concept of being able to visually create complex search scenarios that are geared towards examining code in ever-increasing, specialist ways.

You can also use the R# options dialog box to permanently save collections of often-used search patterns, which can be shared with other members of your team, as mentioned previously.

Structural patterns can also go one step further. You can define them, then set them up to eliminate common code smells. You'll see more on this later on, but for now, know that R# constantly keeps an eye on your code as you type. As it does this, it watches out for common patterns that developers use and suggests improvements.

For example, R# might watch for “For Each” loop usage where the entire loop can simply be replaced with a single LINQ style select statement.

Using structural patterns, you can easily set up your own solution-wide code smells detections and then decide how you want them to be handled. A practical example will help demonstrate what I mean. Figure 58 shows a snippet of code from a project recently converted from Java source code:

```
        22 references
74      public override string ToString()
75      {
76        StringBuilder str = new StringBuilder();
77        
78        int f_number = (fnumh << 8) + fnuml;
79
80        str.Append(string.Format("channelBaseAddress: {0}\n", channelBaseAddress));
81        str.Append(string.Format("f_number: {0}, block: {1}\n", f_number, block));
82        str.Append(string.Format("cnt: {0}, feedback: {1}\n", cnt, fb));
83        str.Append(string.Format("op1:\n{0}", op1.ToString()));
84        str.Append(string.Format("op2:\n{0}", op2.ToString()));
85
86        return str.ToString();
87      }
88    }
89  }
90
```

Figure 58: Snippet of C# code converted from Java

A common pattern used by Java programmers is to use the `Format` function with the Java implementation of the `StringBuilder` class to produce formatted strings. This converts quite easily to C#, because all we have to do is prefix the `Format` calls with `string`, and everything is good.

C#'s string builder, however, has an `AppendFormat` method, which takes exactly the same parameters as `string.Format`, but gives much neater source code.

Let's use R#'s structural patterns to detect, notify, and help change this code smell.

Click **ReSharper** > **Find** > **Search with Pattern** to get the pattern definition dialog box open:

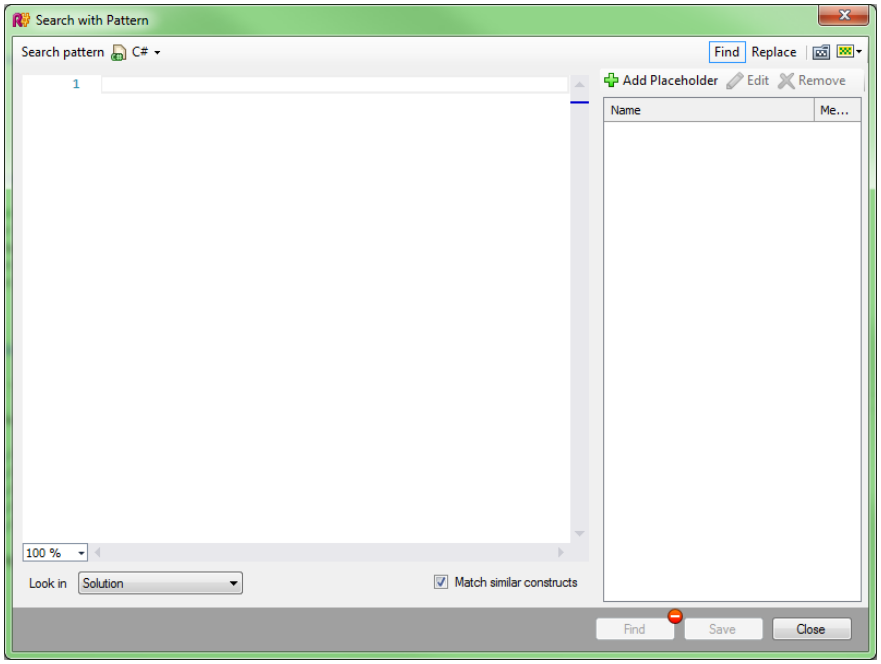


Figure 59: R# Structural search pattern dialog box

If you look at the converted code, we want to search for an expression that is of the `StringBuilder` type, followed by a period and the keyword `Append`. This keyword is then followed by the phrase `string.Format()`, which contains two arguments—a format string and a variable number of parameters. In your Search pattern dialog box, enter the following structured search string:

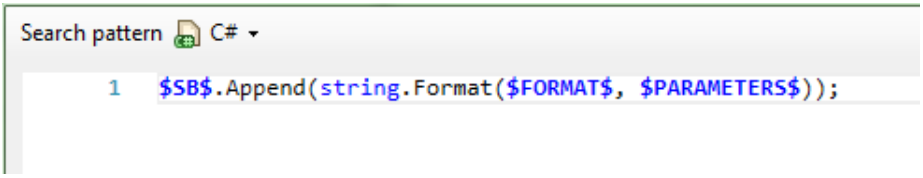


Figure 60: Structural search pattern to find converted Java code smell

The `$SB$` part at the beginning represents the string builder expression, the `$FORMAT$` part represents the format string, and the `$PARAMETERS$` section represents the remaining parameters in the `string.Format` call. If these are initially red instead of blue, you need to define them and their types.

In R# 8.2 however (compared to earlier versions), I was pleased to find that R# automatically figured out what I was doing and set the parameters up for me. Even if you have this capability, you still need to tweak the parameter settings, however.

To the right of the structured pattern dialog box you should see a section that looks similar to the following:

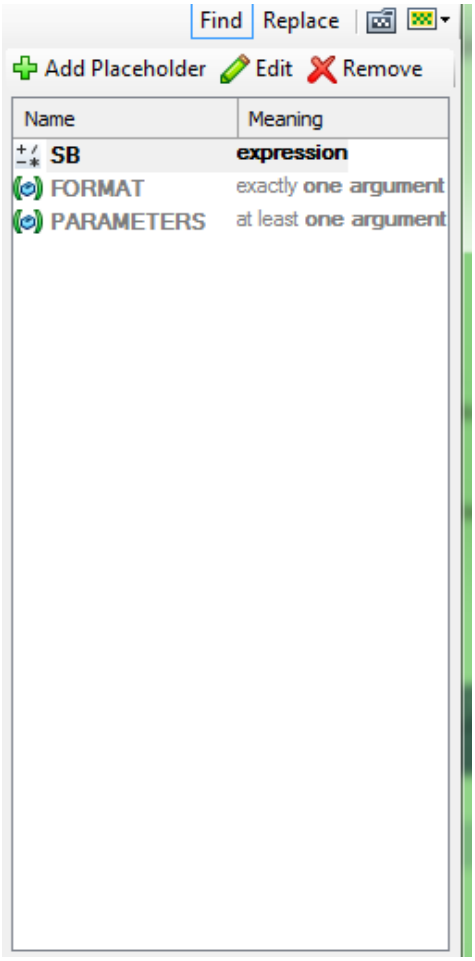


Figure 61: Structured Search parameters

If you don't already have anything added, use **Add Placeholder** instead of **Edit** in the next set of actions.

Click on (or add) **SB**, then click **Edit**. You should see the following:

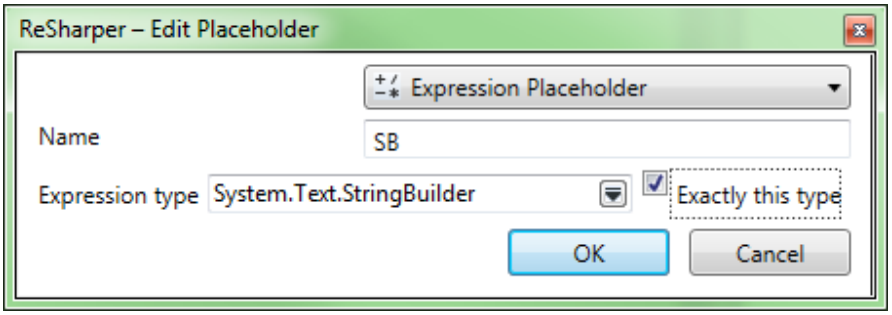


Figure 62: Expression editor for the SB parameter

Ensure that **Expression Placeholder** is selected, the Name is set to **SB**, Expression type is set to **System.Text.StringBuilder**, and that the box beside **Exactly this type** is checked. Then click **OK**.

This will ensure that R# only searches for expressions that are of type string builder and followed by Append, while honoring the other criteria also.

As you did with the SB parameter, click on **FORMAT**, then click **Edit** and bring up the properties for that token:

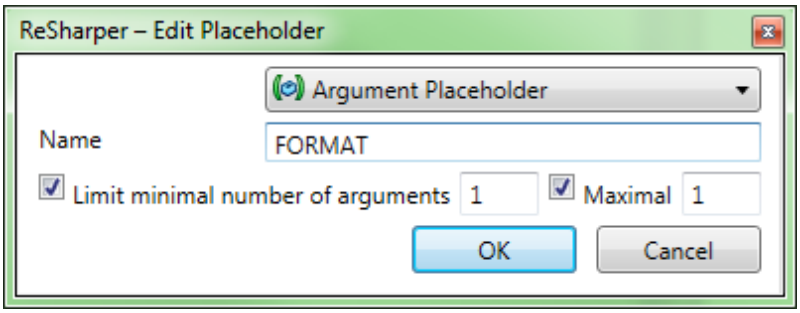


Figure 63: Expression editor for the FORMAT parameter

For FORMAT we want one occurrence of a string, and one only; we also want to make use of the matched text soon, so we want this to be an argument placeholder. Make sure the settings for FORMAT match those in the dialog box in Figure 63, then click **OK**.

Finally, do the same for the PARAMETERS token; this one will be used to grab anything that's left in the `string.Format` call. Because `string.Format` takes a variable number of parameters, we need to ensure that we grab them all.

Set the PARAMETERS options up as follows:

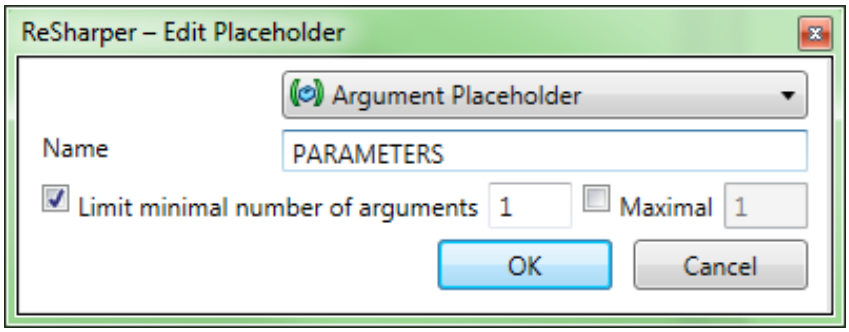


Figure 64: Expression editor for the PARAMETERS parameter

You'll see here that we want to ensure there's at least one parameter, but we don't really care how many more there are beyond that, so we specify a minimum of one, and no maximum.

If you set **Look in** (near the bottom) to **Current File** and click the **Find** button, you should end up with something similar to the following (assuming you're using similar code):

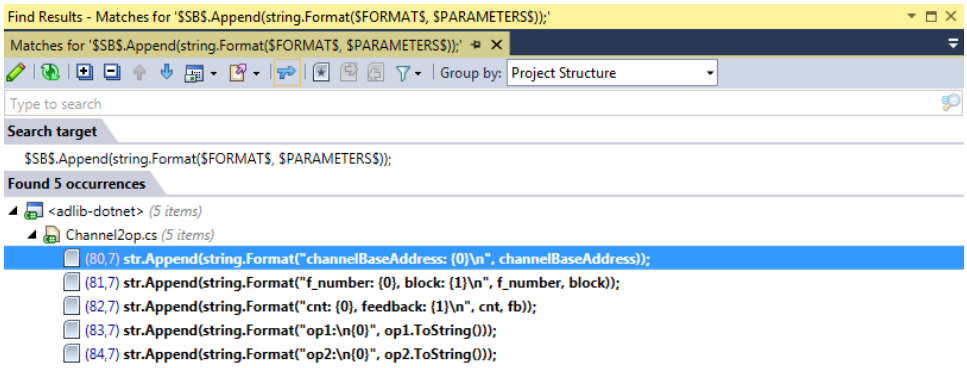


Figure 65: Results from the string builder structured pattern search

I only set it to Current File because I know this is the only file that has this pattern, but I could have easily set the search to be solution-wide, and then gone through and found every occurrence of this pattern in my solution.

For now, just close the results, then from the R# menu, reopen the **Search with Pattern** dialog box.

If you look toward the top-right corner of the dialog box, you'll see two buttons: Find and Replace. If you click **Replace**, your dialog box should change to look like the following:

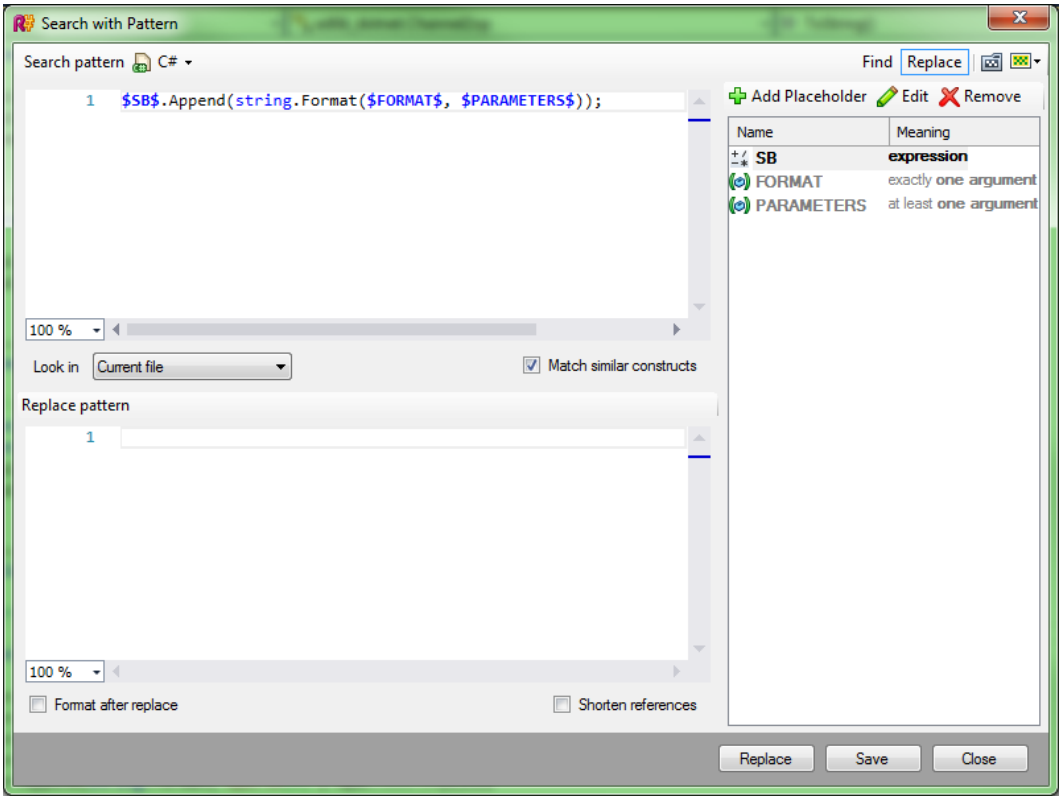


Figure 66: Structured search dialog box with replace options enabled

What you can do now is to enter a replacement string in the lower box, which can use all the parameters you specified in the search. Enter the replacement pattern into the lower half as follows:

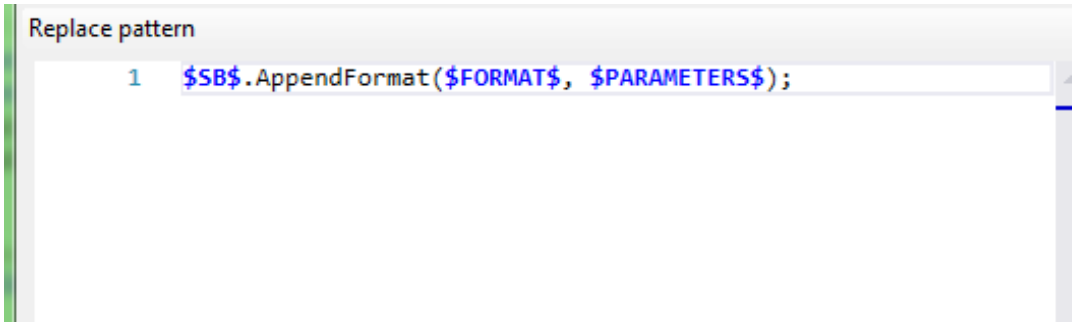


Figure 67: Replacement pattern for our code smell example

If everything is ok, you should see that your Find button near the bottom of the dialog box has changed to Replace. Go ahead and click it. You'll get the normal selection box to decide which ones you want to replace, and with a couple more clicks you should find that you quickly eradicate those nasty code smells and make C# a happy place to be once again.

Impressed? We still have one more trick up our sleeve—press **Ctrl+Z** to undo that last change and bring our code smells back.

Once we have our original `string.Format` lines reinstated, head to **ReSharper** > **Options** > **Code Inspection** > **Custom Patterns**. You should end up with something like this:

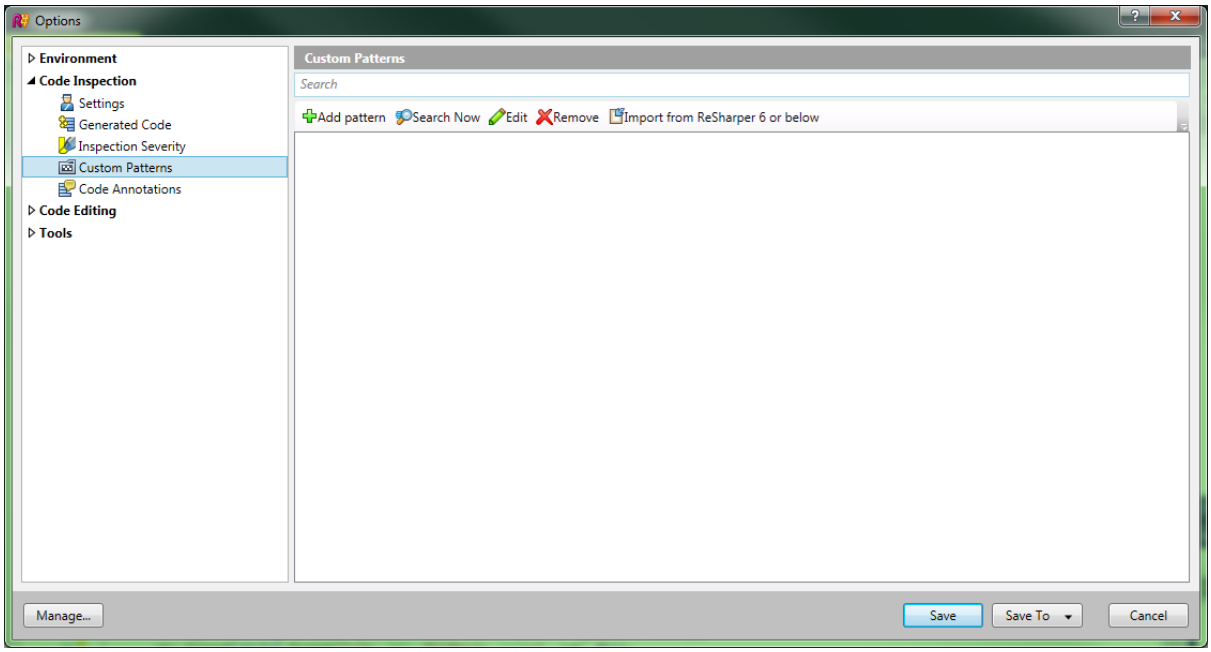


Figure 68: R# Options dialog box for structured patterns

If you click **Add Pattern**, you should see your original pattern still in its editing box show up, just like the following image:

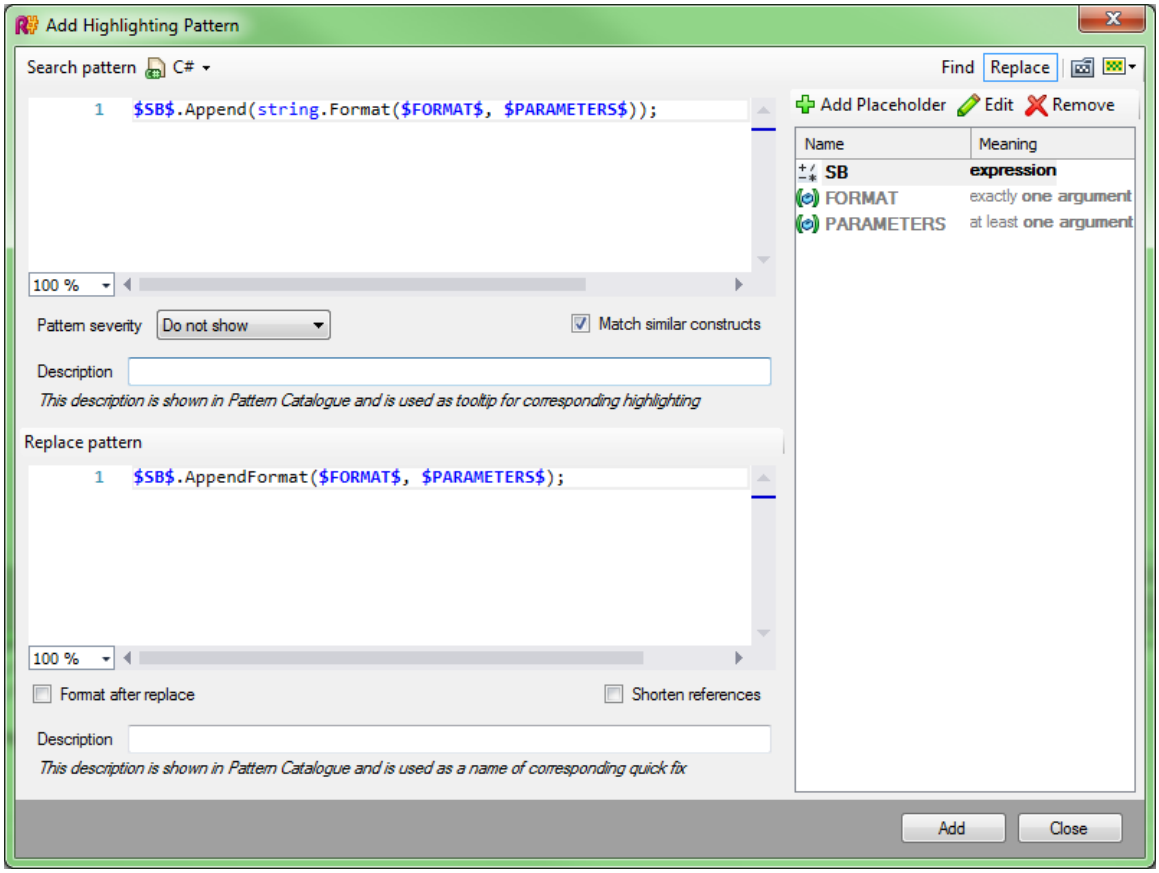


Figure 69: R# pattern editing box called from Options

This time however, you should see there are now two description boxes, one under each of the editing inputs. These descriptions are used by R# in the code editor to highlight and point out code smells that match the “append” one automatically.

The first one is used by the tooltip; when you hover over a highlighted pattern in your editor, the second one shows up as the name in the Alt+Enter quick-fix menu. For now, add **Usage can be changed to AppendFormat** in the first description, and **Change to use AppendFormat** in the second. You'll also see that you have a Pattern Severity dropdown menu. You can change this to any of the warning and error conditions we discussed earlier in the book. If you set it as an error, then you can actually prevent your project or solution from being built if it contains any matches to this pattern. For now let's change the severity to **Show as Warning**, and then click **Add**.

If you now click **Save** to save to the default settings location, then look at your editor, you should see you have yellow warning marks in the right-hand margin and colored underlines with tooltips under each of the lines.

Figure 70: Visual Studio editor showing R#s reaction to the append structured pattern

Press **Alt+Enter** or click on the Quick Fix menu, and you'll see the replacement description now appears as an option:

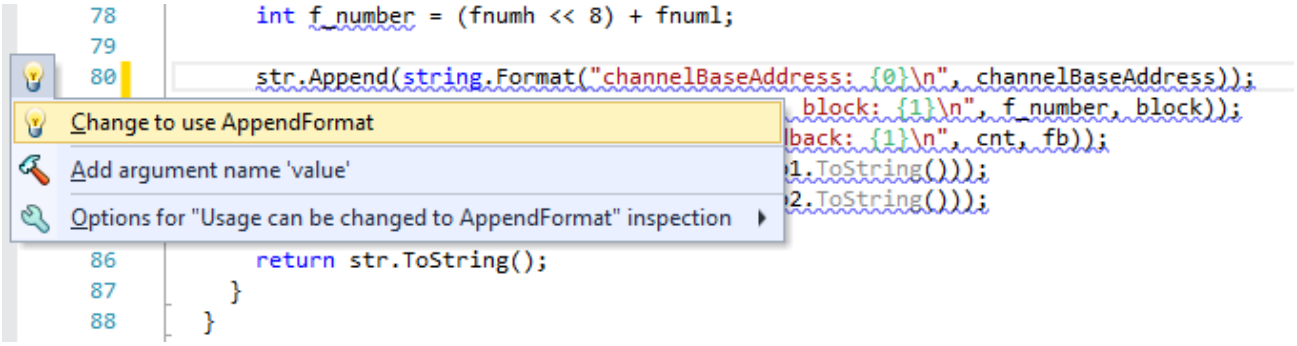


Figure 71: R# Quick Fix menu showing the fix option for the append code smell

As expected, selecting the menu option, or double tapping Alt+Enter will quickly fix the issue just as you would for any other R# Alt+Enter action.

I'm sure you can think of a lot of creative uses for this. I have structured patterns set up in mine to quickly switch between data access technologies for example; this is done by looking for "SQLite*" style patterns and replacing them with "PSql*" which when activated, allow me to quickly replace connectors, string builders, table adaptors, and all sorts of other things easily and quickly. Remember too that these patterns are saved to “team-shared” settings as described previously, which means that when you set them up, and share them, your entire team then has access to them.

The last three options on the Find menu all have to do with marking and navigating to those found areas in your code. **Highlight usages in file**, where appropriate, will close any search dialog box you might have open and place a highlight on any match to any search you just performed.

These highlights can and will span multiple files if that’s what you asked for. Solution-wide, project-wide, or even just local to a file, every match that matches the criteria will be highlighted.

To help with this file spanning, you'll find the two final options presented in the Find menu are **Go To Previous Location** and **Go To Next Location**, which do exactly what they say on the tin, and allow you to jump back and forth through your search results with a simple key press.

Editing Tools

In the last section of this chapter, we'll turn our attention to the editing tools that R# provides for the developer.

As with the previous sections, we'll start off with an expanded image of the main menu:

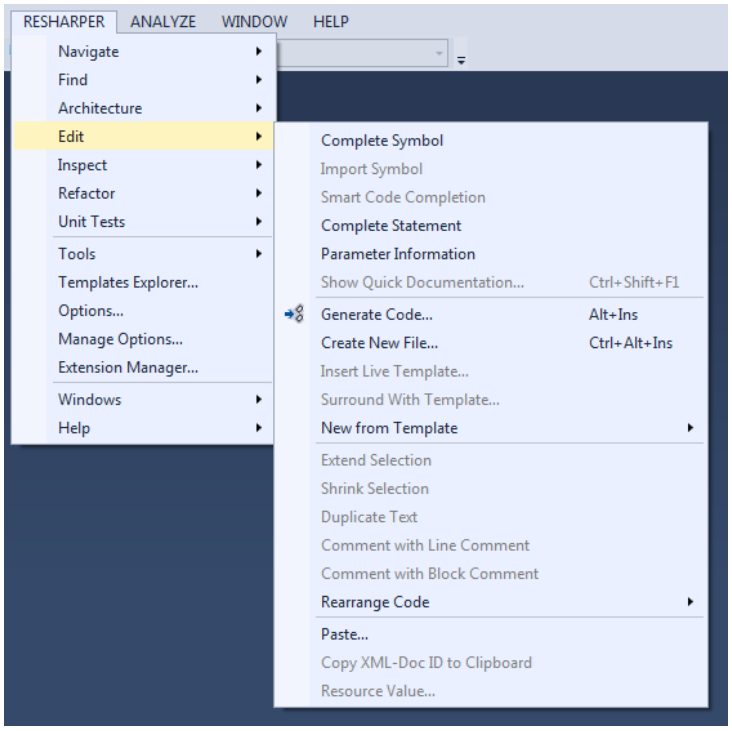


Figure 72: R# Edit menu

Many of the options you see here are dedicated to completing the statements you're currently typing, or the snippets and variable names you may be currently working on.

Complete Symbol, for example, is functionally identical to the normal Visual Studio IntelliSense that most VS users see every day.

Import Symbol, on the other hand, is like IntelliSense on steroids. This option will not only look for matches within your local namespace, class, or other local object, but it will look at the entire landscape of your current project or solution. Then it will present a list much like the normal IntelliSense completion list, with the added option of listing namespaces, assemblies, and anything else that is also reachable from your current source code position.

If, for example, I have AutoMapper installed as a NuGet package in my project, but have not yet added a `using` or other reference to the library. I could start typing one of AutoMapper's symbols and import symbol will correctly identify that; the mentioned reference could be reached by including a `using` clause to AutoMapper.

Not only will R# then complete the statement for me, but it will also ensure that any `using` statements or further references required to make things work are also added into my current source file.

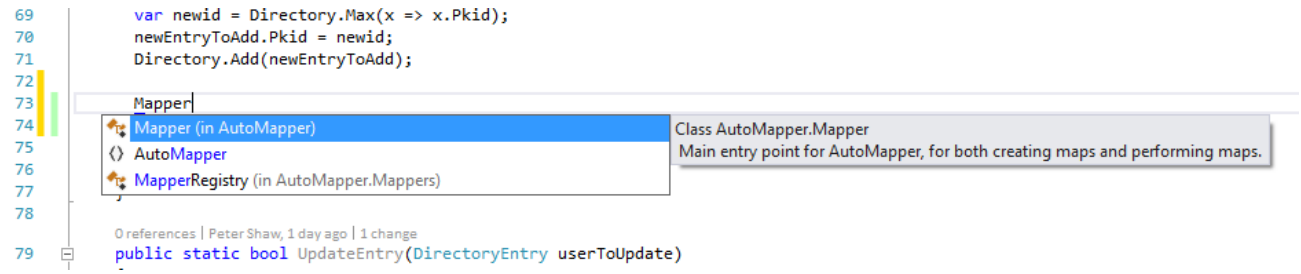


Figure 73: Import symbol completion pop-up menu showing AutoMapper before it was registered

The next menu option is **Smart Code Completion**, which, like the previous two, is designed to complete the statement or part thereof that you're currently working on. What makes Smart Code Completion different to the others however, is its treatment of variables and types.

Smart Code Completion ONLY displays candidates that are either of the same type or a compatible type, or can be confined to the same interface, object, or abstract type as the entry you're typing against. This means that smart completion is mostly geared towards completion of function parameters, or already declared and known result types.

It reduces the chance that you'll add code errors to your project by simply not giving you the choice of something that it knows will not work. While it's still not foolproof (you can assign an integer to a double, for example), it does prevent many of the mistypes that you often see in code produced by junior and trainee developers.

If you're working to a specific interface, then it helps by only listing class types that conform to that interface, rather than listing objects that are spelled similarly or in the same type family, which can often shorten the list of choices you have and make a menu easier to navigate.

Complete Statement is an amalgamation of all of the above menu options, and just serves to wrap everything up under one menu. The advantage here is that you can perform all three types of completion in the same action. In my case I don't have shortcut keys set up on the other three options, only on the final one. When my custom shortcut key is pressed, I get to select the best option from the menu at the appropriate time.

The last two options in the top group are mainly to help you as a developer from a documentation point of view. When your cursor is placed inside an area of your source code that takes parameters, activating the **Parameter Information** option will display the parameters information tip as follows:

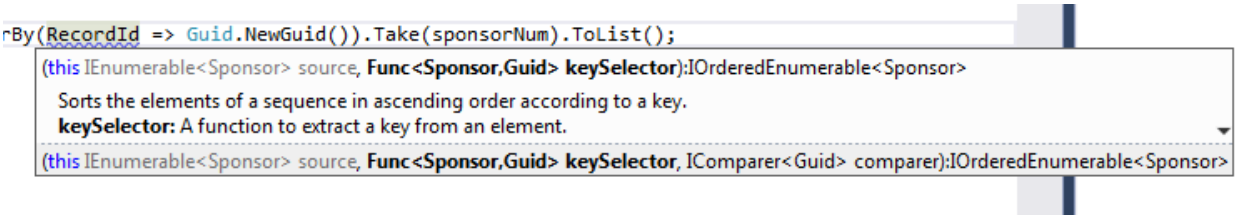


Figure 74: R# parameter information tip

The parameters tip will show you the parameters from any code in your solution, or from any framework and third-party libraries you might have loaded. It does this by periodically examining the landscape of your application, and caching what it finds available in the metadata of the loaded assemblies.

Show Quick Documentation is also a similar development aid, in that it attempts to give you a miniature help view of the function or method call you're currently positioned over:

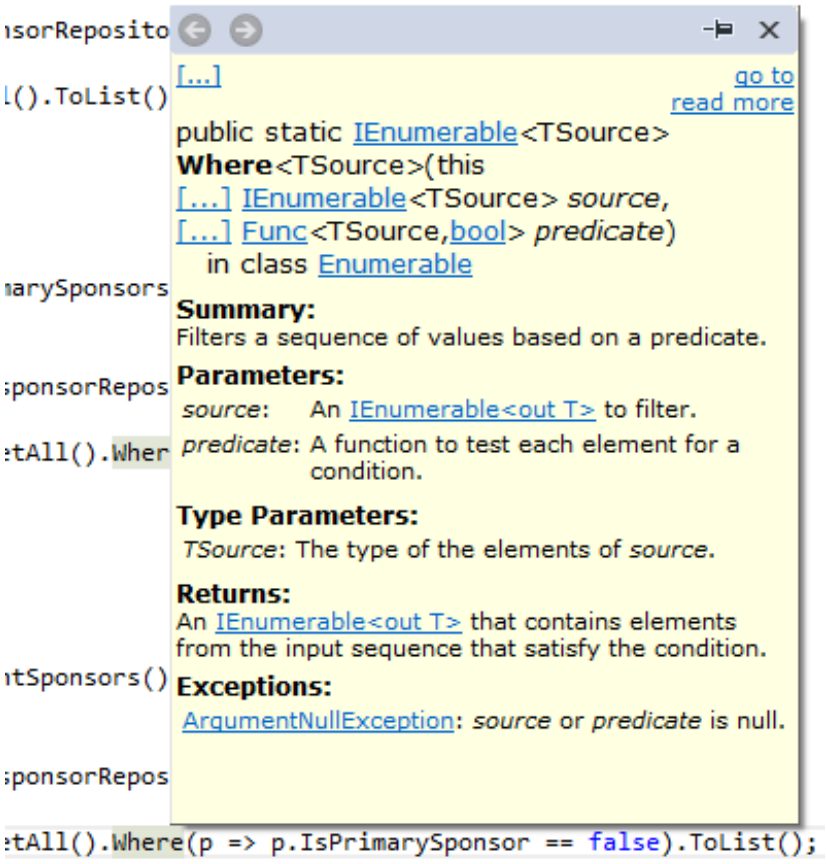


Figure 75: R# quick help pop-up menu

Just like the parameter tip, this will also attempt to automate documentation for your own code and projects in your solution. If you use the XML commenting system built into Visual Studio, and mark up your methods correctly, quick help will build on-the-fly doc tips like those above, and display the required information for developers writing software against your own in solution toolkits. If the call is against .NET framework or other third-party assemblies, then it will do its best to obtain either symbolic information or metadata where it can.

In the case of the .NET base libraries, R# has a lot of these tool tips built into its internal product, ready for use. The default shortcut key for this is Ctrl+Shift+F1.

Chapter 5 Code Generation

There's one thing that R# does, and does amazingly well: it generates code. From snippets to templates, from wraps to conversions, R#'s tools were among the first things added to the product, and continue today to be one of its outstanding features.

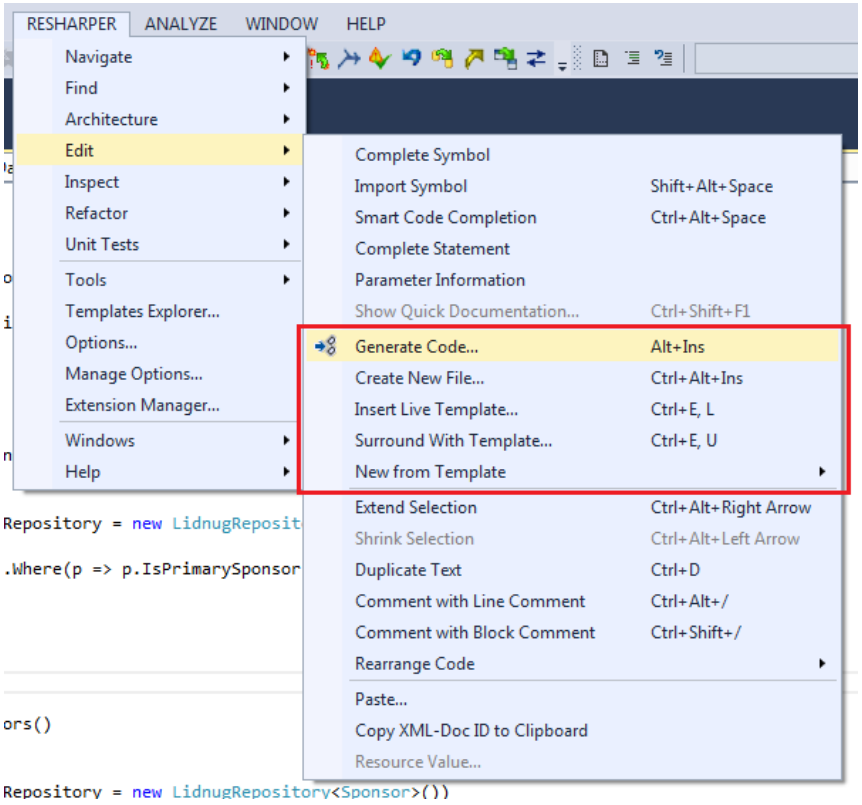


Figure 76: R# code generation features on the Edit menu

In the second group of options in our Edit menu, you'll see just a few of the options R# provides for helping you in your daily task of writing the application code for the products you and your team build.

So what exactly can R# generate?

Let's start with the **Generate Code** option (Alt+Ins). This option is geared specifically to one task that developers do very often: create classes for objects.

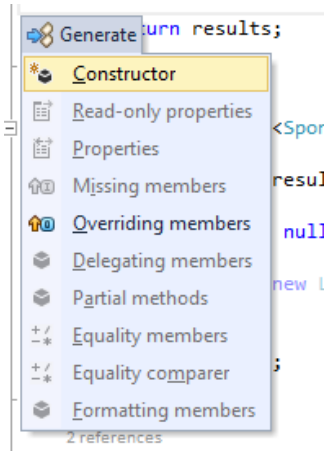


Figure 77: R# Generate Code menu

If you place your cursor anywhere in a source file, then activate the Generate Code pop-up menu, you'll see something that looks like Figure 77. However, we can get a better feel for what each option does if we create an empty class, then go through the options in turn. Create an empty class in your code, something like the following:

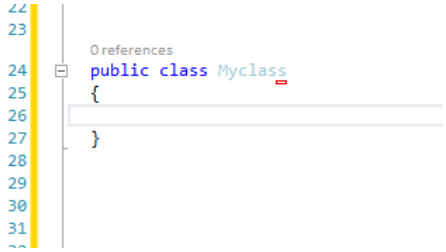


Figure 78: Empty class created to demo the create code functions

Now position your cursor inside the empty class and press **Alt+Ins** (or use the menu) and pop open **Generate Code** menu once again. You should now see that there are a few more options available on the pop-up than there were last time.

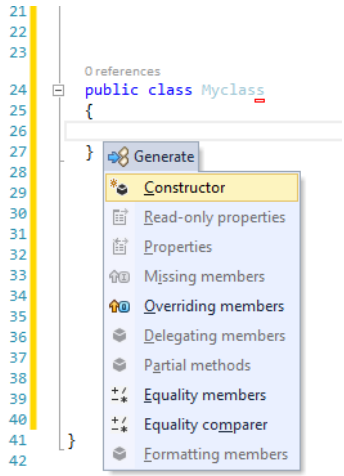


Figure 79: Generate Code pop-up menu in an empty class definition

The four options available are tasks that R# considers important to the creation of an empty class. If you select **Constructor**, you should get a default constructor:

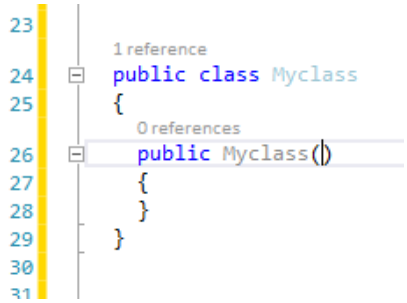


Figure 80: Empty constructor created by Generate Code

Your cursor will be automatically positioned, so you can immediately start to type your parameter list. This may not be a groundbreaking feature to most people, but when you consider that you just did that in about two key strokes, long-term usage really builds up a picture of how many minutes, even hours, it can save you.

Looking back at Figure 79, you'll see **Overriding members**, **Equality members**, and **Equality comparer** listed as options. Anything in your base class that can be overridden, or set up to make an `.Equals` class-specific call can be created and injected with these three calls.

If you select **Overriding members** for example, you'll get a pop-up menu that looks like the following:

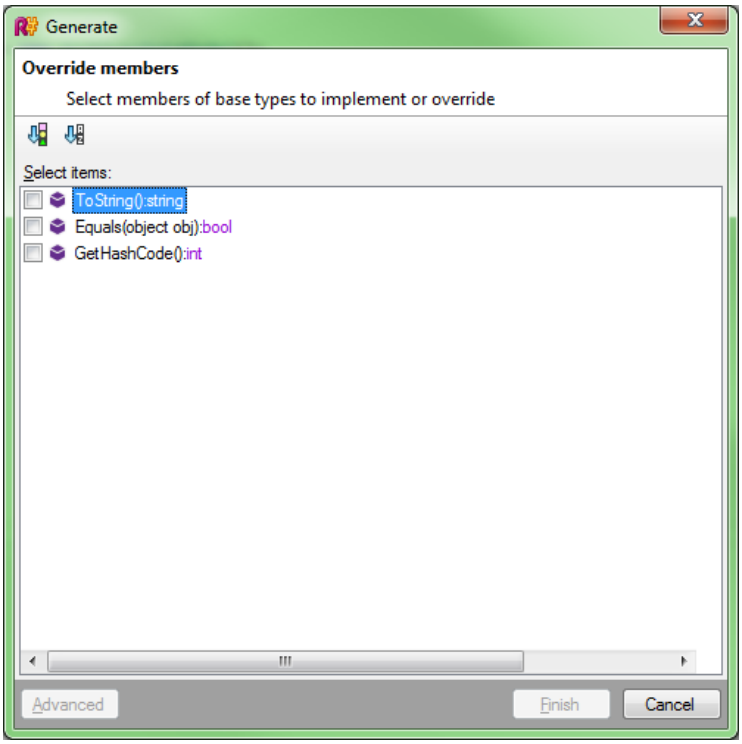


Figure 81: Override members dialog box

Anything on the class that can be overridden will be listed here. In this case, I've got just a simple class, so just the general .NET base calls are available. If I select all three and click **Finish**, I get the following code:

Figure 82: Code generated by R#'s Overriding Members option

Note: You might have noticed by now that in some of the screenshots, the color of some of the code is faded. This is another of R#'s helpful code diagnosis tools that helps you see how good your code coverage is. In Figure 82, all three of the overrides are faded, simply because they haven't yet been customized, and so offer no value to the code base. We'll cover this more in later sections.

As you can see, you now have a `ToString`, `Equals`, and `GetHashCode` ready to fill out with our own custom code. Once we add our code into these, then .NET will call those methods when and where they're needed.

You might have noticed that I've gone down the code generation menu from top to bottom, as I have done with most of the other menus in this book. This might not show ReSharper's code generation functionality in the best light, simply because I've added code into my class that ends up creating redundant code. In a real project, you'd likely want to use one or the other based on your project's needs. I've done things the way I have to keep things simple for the beginner user of ReSharper. As you gain experience with the product, it's worth remembering that many of the tools do duplicate functionality, and on occasion blindly activating things without first making an educated decision can sometimes create more work than it saves.

As a final demonstration, place your cursor back inside your class definition, and activate the **Generate Code** option once more.

This time, select **Equality Members**, and you should see that because of the previous Overridden Members option, it asks you to confirm if it should change the Equals stub, which is already generated:

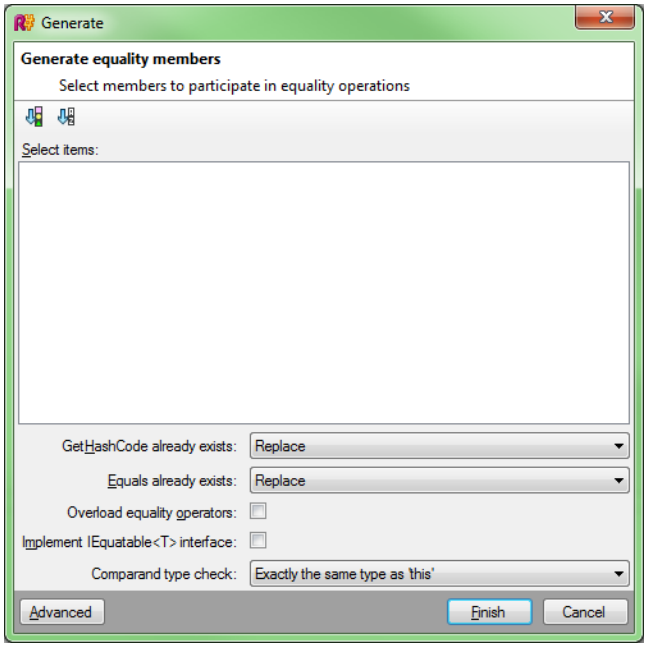
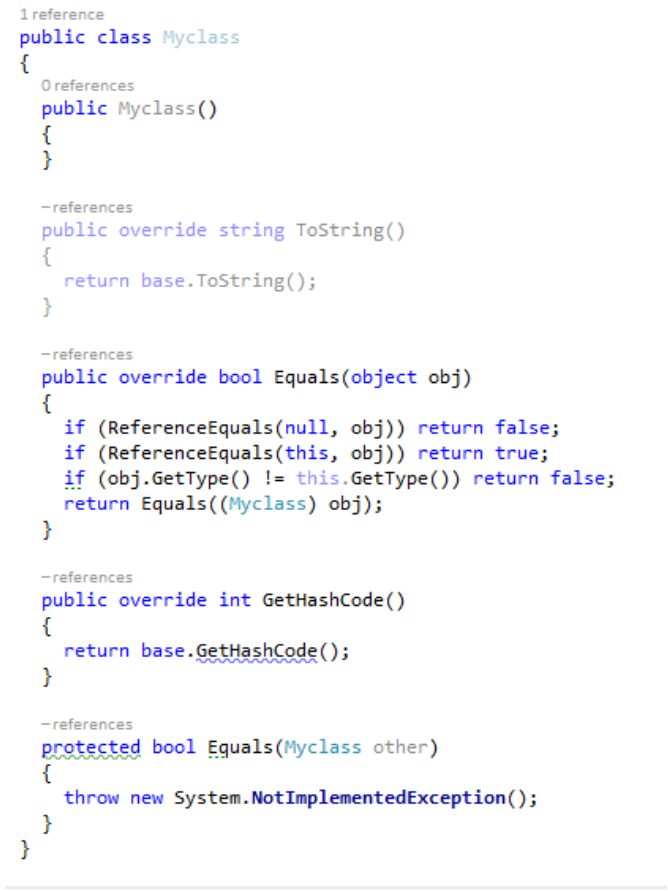


Figure 83: Confirmation that you are about to replace an existing overridden method

In this case I chose to replace them, since I hadn't put any custom code in the method. If I had, then I could have skipped the implementation, or gotten something called "Side by Side." This would have injected the code in a manner that would not have caused any problems with my existing implementation while allowing me to see the code that would be generated.

If we now look at the code once more, you'll see that some slightly more boilerplate code has been created for you to use:

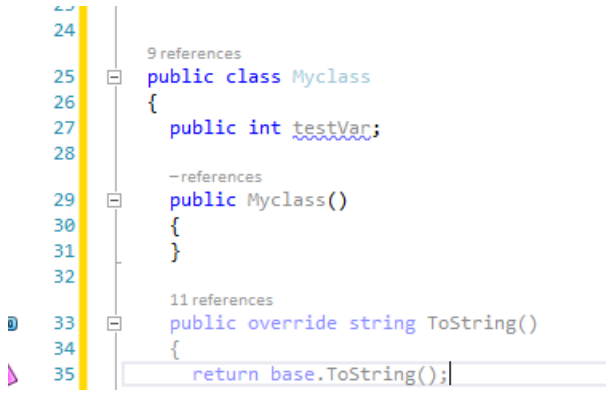


```
1 reference
public class Myclass
{
  0 references
  public Myclass()
  {
  }

  0 references
  public override string ToString()
  {
    return base.ToString();
  }

  0 references
  public override bool Equals(object obj)
  {
    if (ReferenceEquals(null, obj)) return false;
    if (ReferenceEquals(this, obj)) return true;
    if (obj.GetType() != this.GetType()) return false;
    return Equals((Myclass) obj);
  }

  0 references
  public override int GetHashCode()
  {
    return base.GetHashCode();
  }

  0 references
  protected bool Equals(Myclass other)
  {
    throw new System.NotImplementedException();
  }
}
```

Figure 84: Empty class code after running Equality Members

Implementing the **Equality Comparer** option stubs out the code needed to make use of the IEqualityComparer interface. This further allows you to create and support dynamic comparison interfaces in your code, giving you the ability to delegate the comparison to an external class while still retaining the ability to control the result.

Like many of R# other options, the rest of the menu is context-specific. First add a property to your class as follows:



```
24
   9 references
25 public class Myclass
26 {
27   public int testVar;
28
     0 references
29   public Myclass()
30   {
31   }
32
     11 references
33   public override string ToString()
34   {
35     return base.ToString();
```

Figure 85: Dummy class with a property added

If you then position your cursor on the property and press **Alt+Ins**, four more options become available:

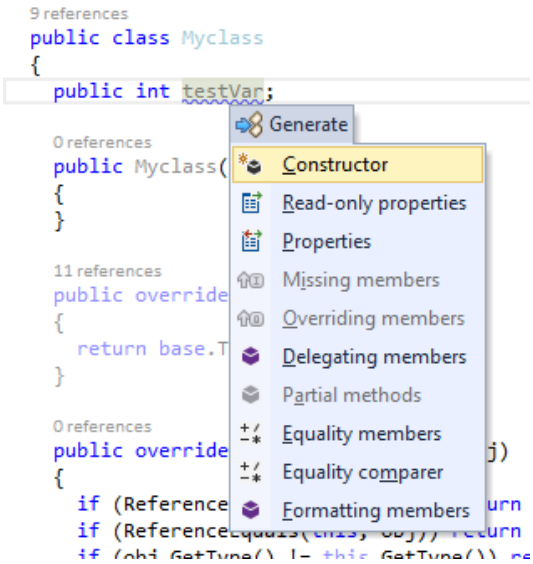


Figure 86: Generate Code menu with property based options available

Choosing the **Read-only properties** option gives you the following dialog box, asking you which properties in your class you would like to generate read-only stubs for:

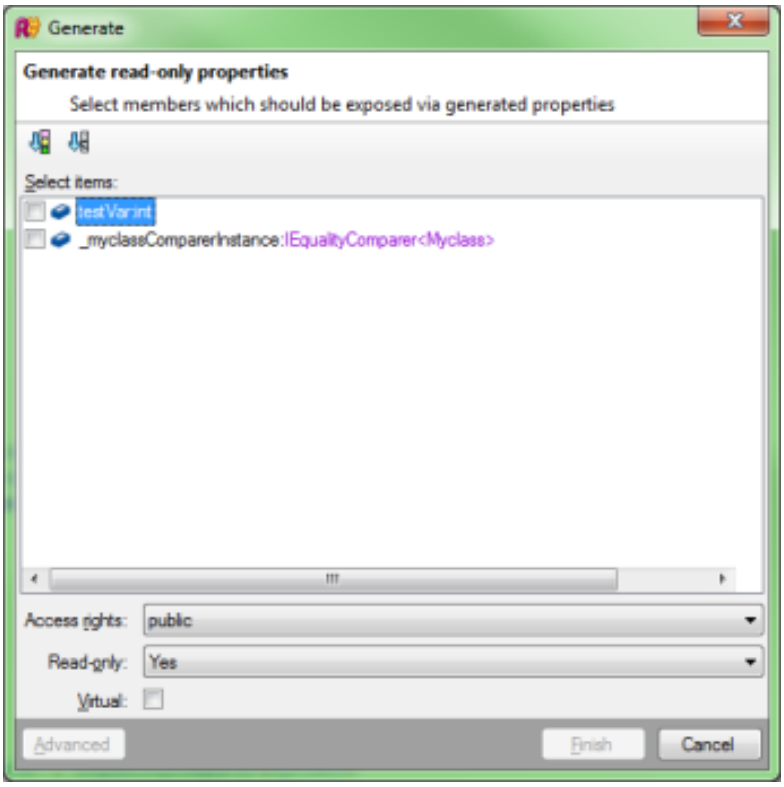


Figure 87: R# read-only properties dialog box

In this example, I selected **testVar** and clicked **Finish**, ending up with the following code:

```
                0 references
25              public class Myclass
26              {
27                  public int testVar;
28
                    0 references
29                  public int TestVar
30                  {
31                      get { return testVar; }
32                  }
33
                    0 references
34                  public Myclass()
35                  {
36                  }
37
```

Figure 88: Class showing testVar read-only property

If I wished to, I could now use Alt+Enter on the green underlined public accessor or the class variable, change the access to protected, internal, or private, and force all reads to the `Var` to come into the class via the read-only accessor.

The **Properties** option does a similar thing, but it creates full read/write property accessors in the process, rather than the read-only ones we've just seen.

The remaining options, **Delegating members**, **Missing members**, and **Partial classes**, are all used to generate code stubs for things like interface contracts, extensions of existing partial classes, and the creation of method stubs used to delegate access to the given property or variable.

Just that one menu option alone has the power to reduce your daily key press count by a factor of about 100, and we haven't even gotten started on code and snippet templates yet.

And that brings us onto the next stop on our tour, **Create New File**.

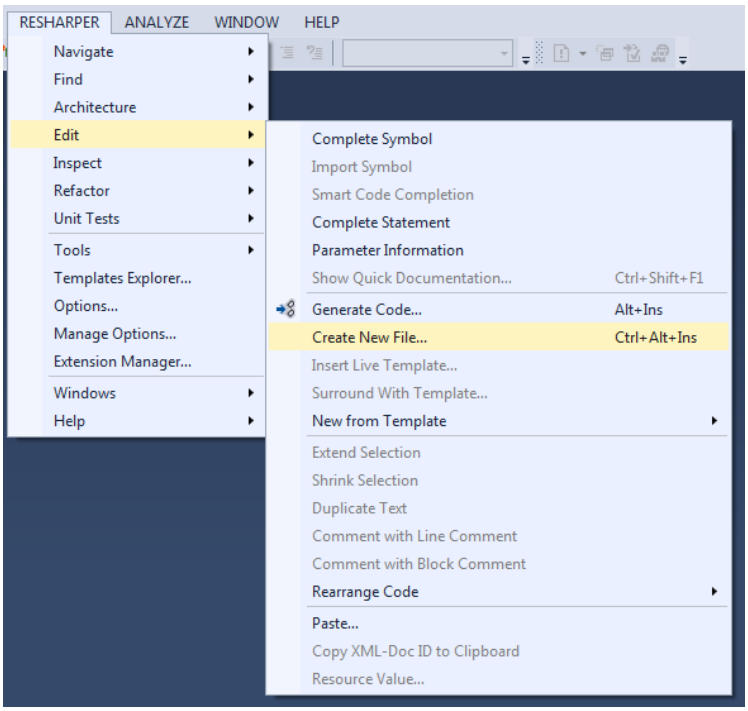


Figure 89: Create New File

Unlike many of these menus, Ctrl+Alt+Ins is not context-sensitive. This means that you can press these keys just about anywhere, from your code editor to a highlighted solution tree root.

So what's the advantage of using this over the standard Visual Studio's "Add new…" dialogs? Well for one, the actual contents of the menu are context-sensitive. If I locate myself in an MVC project and press Ctrl+Alt+Ins, I get this pop-up menu:

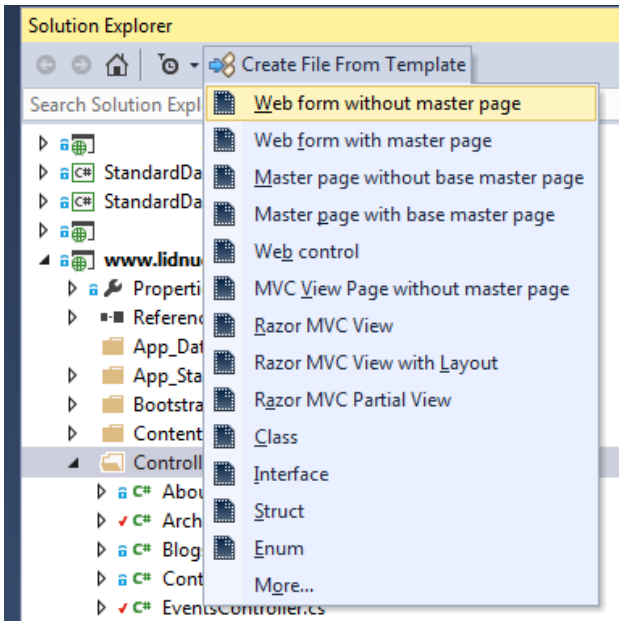


Figure 90: Create File From Template pop-up menu for an MVC project

However, if I locate myself in a standard class library, I only get this:

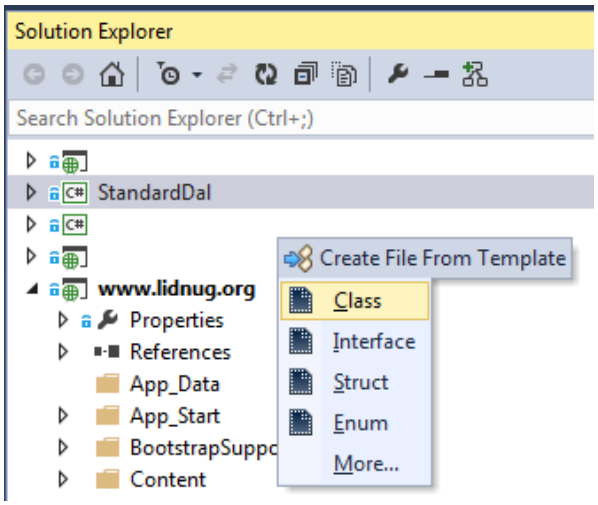


Figure 91: Create File From Template pop-up menu for a class library project

Different project types show different options on the menu, and as you might have already guessed, all of this is customizable in the R# options. If you click on the R# main menu in Visual Studio, in the second section you'll see an option marked **Templates Explorer**.

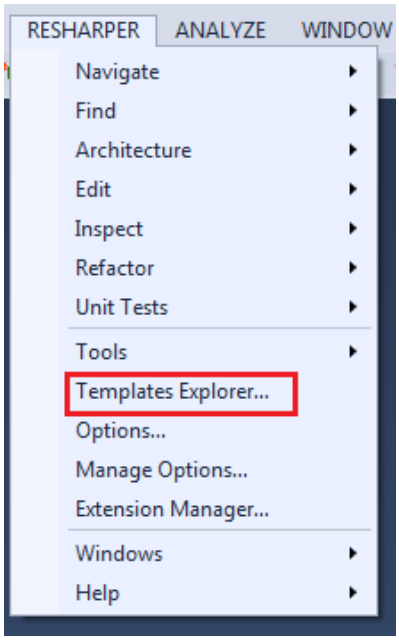


Figure 92: R# Templates Explorer menu

If we click on this option, we get a rather large Visual Studio editor window called the Templates Explorer, with three tabs in it. The tab we're interested in is the **File templates** tab.

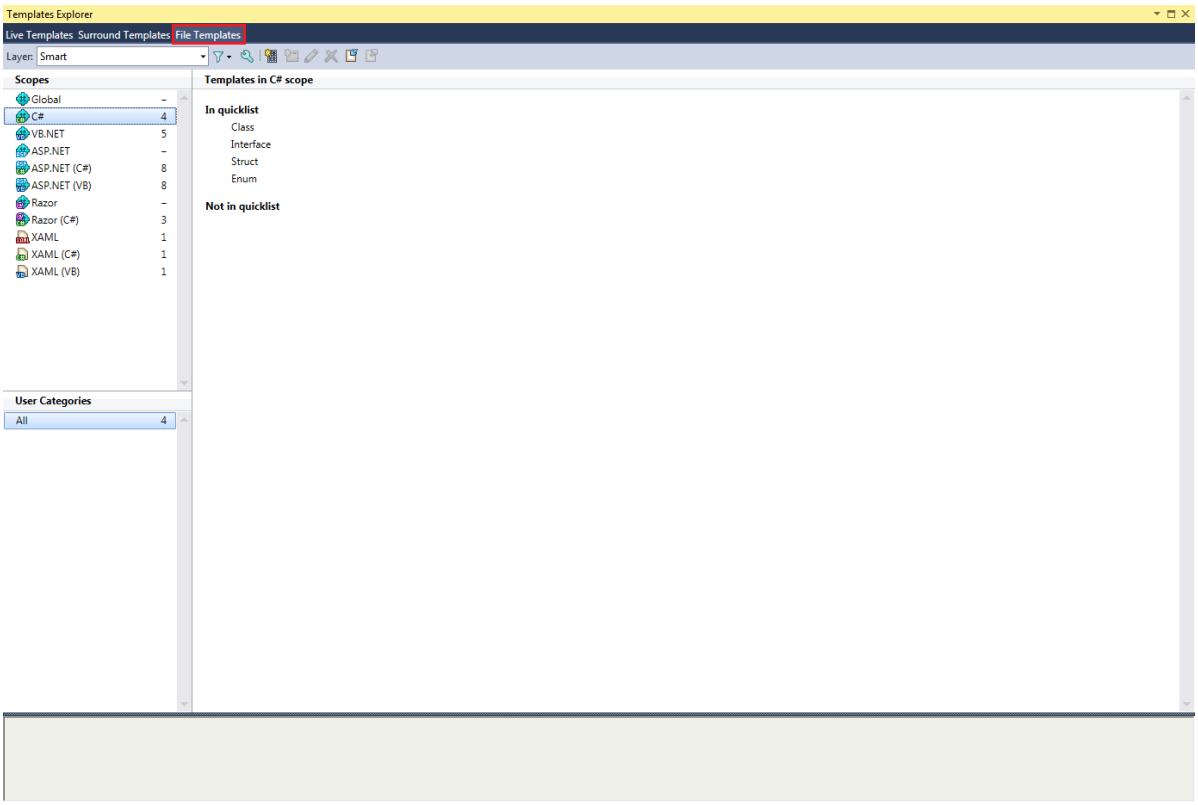


Figure 93: R# File Templates editor

Immediately, you can see in the right-hand pane that we have the **In Quick List** and **Not in Quick List** options, which as you might guess, determine whether or not this entry will appear in our pop-up menu when we press Ctrl+Alt+Ins to insert a file.

If we look at the right-hand side of the dialog box, we can see various scenarios such as Global, C#, and ASP.NET. These scenarios control which project types the different lists work with. If you work your way down the list, you should see in the left-hand pane each of the groups that I demonstrated previously.

I'm going to leave the Templates Explorer, but we'll come back to it very soon. For now, click the **Close** icon on the top right of the templates editor, and let's return to looking at our code-editing capabilities.

If you select a file to insert from the **Create new File** menu, you'll get the usual experience of R# highlighting some areas in the template with a red border, requesting that you fill some things in, like variable names or layout names.

The convenience of this menu is that it's one key press away, and it's quick and uncomplicated. The Visual Studio Add New menu and dialog box often requires you to read the menu options to make sure the item you want is on there, and if it's not, then you need to go into the full dialog and look through the installed templates.

Once again, R# makes an already existing process quicker and easier to activate, and without having to reach for your mouse.

Live templates (accessed by **Insert Live template** from the **Edit** menu, or **Ctrl+E** followed by an **L**) are similar to inserting files. The difference here is that the code snippet is inserted directly into your source code at the position of your cursor. The default menu on a blank line in a razor view file looks like this:

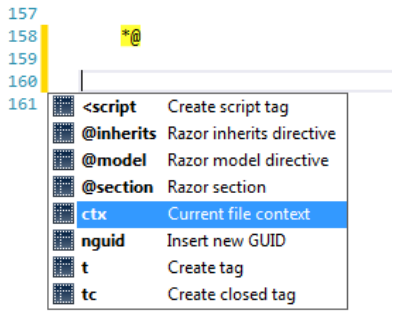


Figure 94: R# insert live template menu

And like the File menu, it will change depending on file type and scenario of intended usage.

The last option in the second group is **Surround with Template** (Ctrl+E followed by U). Like its two predecessors it generates code, but instead of linear code generation, it works by wrapping a selected region.

It is most useful when editing HTML and Razor code, whereby if you select an area of text in your HTML source, you can easily place a new tag around it. For example, if we had a paragraph without P tags, we could select the text and activate **Surround with Template**, choose **tag**, and type in the tag name. Other uses for this tool include adding compiler conditionals such as #if blocks and language constructs such as try/catch and loop structures.

Of course, this doesn't have to be just a tag; it can be parenthesis, curly braces, or anything else that normally would have some kind of opening and closing delimiter.

The best part about this tool is, because this is such a common operation, R# will often add the **Surround with…** quick menu icon to the left margin action bar, meaning that you can very often get directly at the wrap template function with nothing more than a couple of taps on Alt+Enter.

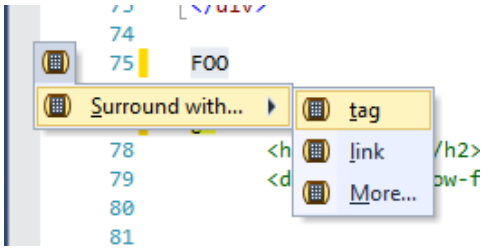


Figure 95: R# Surround with Alt+Enter menu

As you saw briefly a few pages back, ReSharper has a template editor. This editor allows you to customize everything you've just seen, and make your own custom files, wraps, and live templates available for your own use.

There are hundreds of possible combinations, and even more macros to use with them. Unfortunately, there's no room in this book for me to list every one of them, so instead, we'll make a quick useful template for each of the three scenarios, and I'll leave you to explore the rest.

Let's start with a live code template. Open the Template Explorer from the R# main menu, and make sure the Live Templates tag is selected.

We'll make a template to help us create HTML forms using Razor in an ASP.NET MVC project, so click on **Razor** in the right-hand list of the live templates. You should see something like the following:

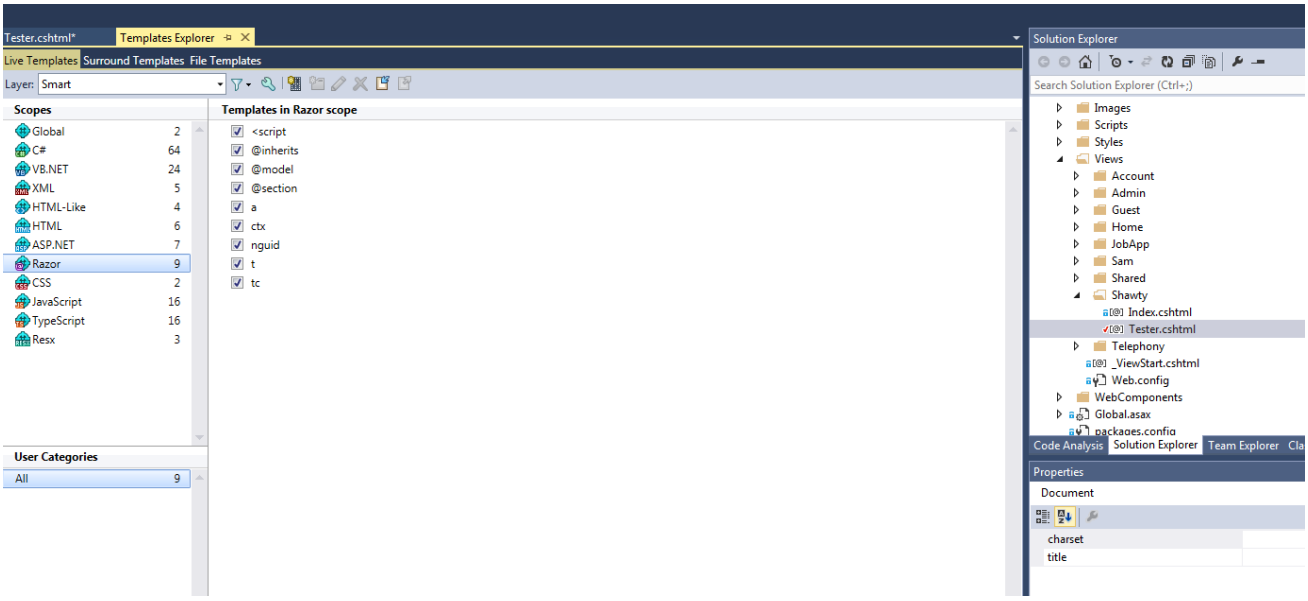


Figure 96: Visual studio showing the live templates editor for Razor

On the toolbar above your templates list, you can see a black and gray checkbox next to the blue bar. Check this box to create a new template; your display should change to look something like the following (bear in mind I have my windows docked):

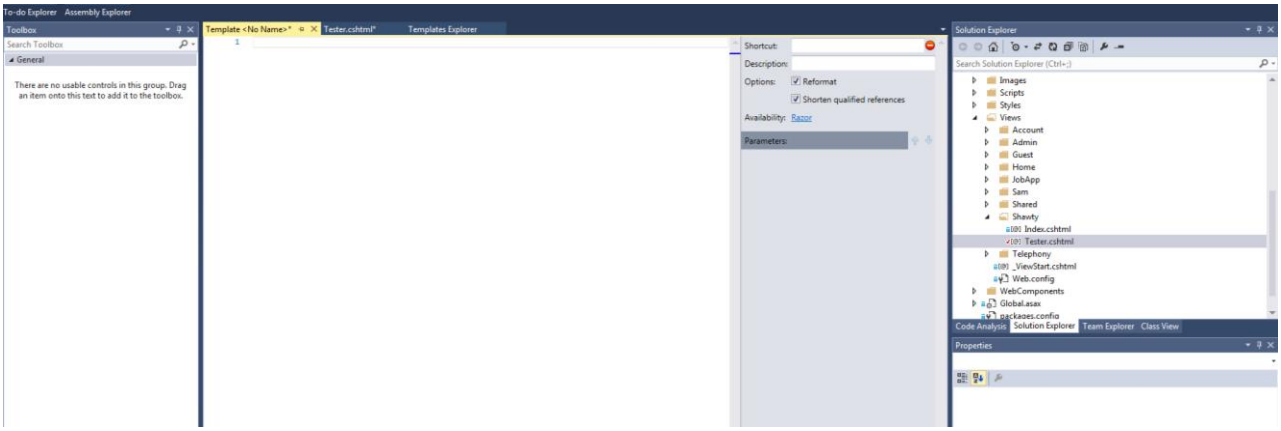


Figure 97: Visual studio showing the New Template editor

In the main section in the center we have a normal editor, which functions just like any other editor in Visual Studio. To the right of this editor, we have the parameters for our template.

- **Shortcut**: this is the text we want R# to recognize before it offers to complete the template for us.
- **Description**: This is the human readable description you'll give to your template. This will show in tool tips when referencing it.
- **Reformat**: With this option, R# will reformat your code to match your current coding styles once the expansion is complete.

- **Shorten Qualified References**: This instructs R# to make any references in your template as short as possible. For example, if you have MyLib.Namespace.Function in your template, and you already have MyLib.Namespace in your `using` list, R# will automatically change the reference to just `Function`.
- **Availability**: This will open a dialog box allowing you to choose exactly where your template will appear, by default this will say "Razor" as we selected razor from our left hand list. You can however go much further than just "Razor," you can select a number of options that allow you to choose exactly how and when your live template will be available.

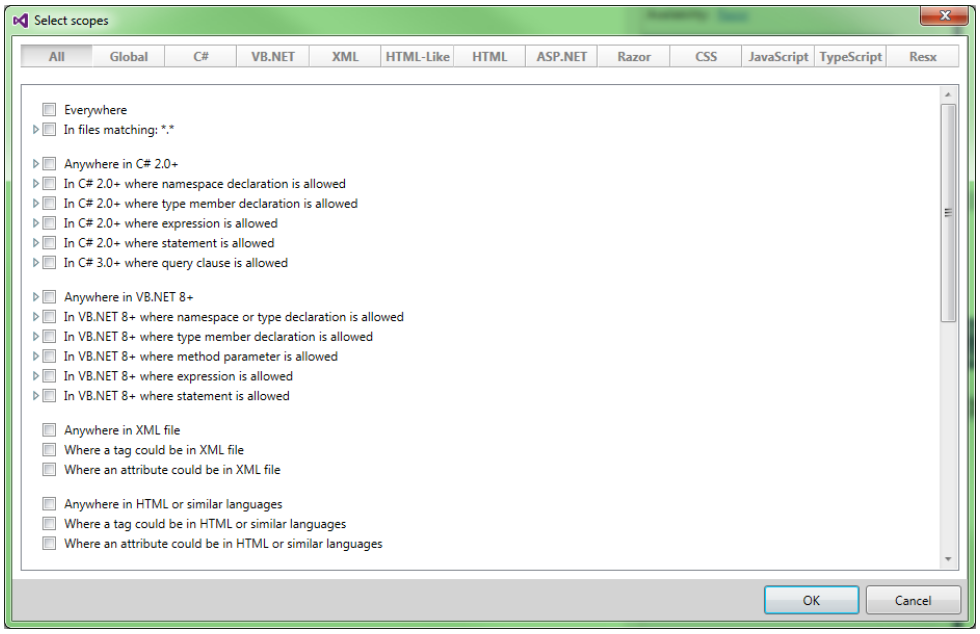


Figure 98: Scope selection dialog box to choose where a template is allowed

Finally, we have our **Parameters** section. As you add expandable parameters to your template, they will appear here, and allow you to pick which of R#'s many macros apply to them.

Parameters are introduced into your template by surrounding the parameter name on both sides with a $ symbol. For example if you wanted the parameter *Name* in your template, you'd specify it using `$Name$`. This would in turn automatically add it to the parameters section in the editor, allowing you to assign a ReSharper macro to it.

Once you declare a parameter, you can use it as many times as you like. Whichever value is expanded into it will be used everywhere in your template that you declare a use of the parameter.

For our live template, we want to expand a typical `using` enclosure used when creating forms in Razor, so we start by putting the following code into our Template editor:

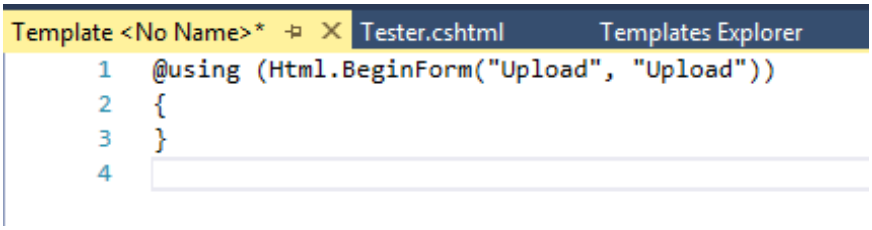


Figure 99: Initial code in our Template Explorer for our Razor live template

In the code in Figure 99, we want to replace the `Upload` strings with attributes allowing the user to fill in the controller and action names, and in the { } section, we need to tell R# where the end is.

There are some special parameter tags that are reserved by R#, you can see one of them below, `$END$`.

To add the required parameters, we change our code to look like this:

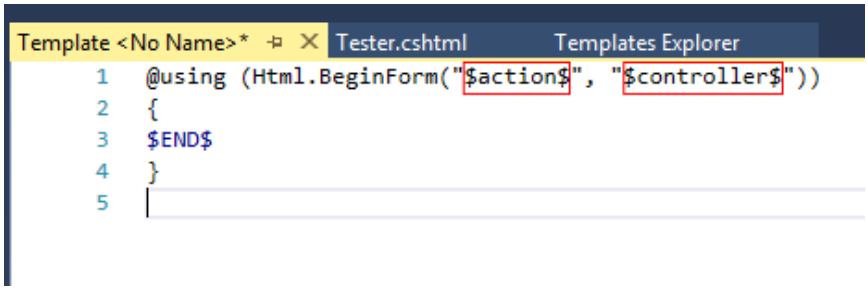


Figure 100: Template code from Figure 99 with the parameters added

One thing to note from Figure 99 is that the `END` template is not right at the end of the template.

This is because `END` marks the point in the template where you would like the cursor to be placed once expansion is complete. In this case, we want the cursor placed so that the end user can continue typing the inner code for the form.

Once you enter the parameters into your live template and fill in the other details, you should now see that the right-hand side in your editor looks like this:

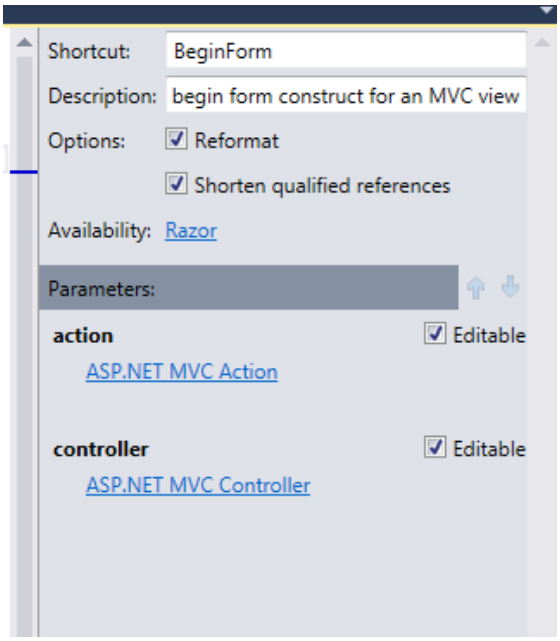


Figure 101: Settings for our new Razor live template

At this point, we can now click on the blue **Choose Macro** links, allowing us to choose an appropriate macro that R# will use when assigning a value to our parameter.

You'll also notice that each parameter has an “editable” tick box; if you select an appropriate macro, you can get R# to fill that in automatically and not allow the end user to actually change anything. In practice though, I've never seen any templates that use this.

Click on the macro entry for **Action**, and you should see the following:

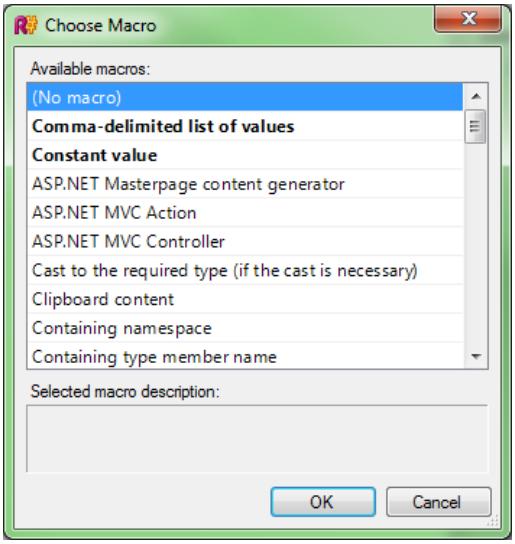


Figure 102: R# Choose Macro dialog box

Choose **ASP.NET MVC Action** for our action parameter, then repeat the same steps for Controller, except this time choose the appropriate controller macro.

Feel free to explore the other macro types available—you'll quickly see that you can do everything from specify a hard-coded list of options, right through to searching every assembly on your machine for possible namespace matches.

If you now click **Save** (or press Ctrl+S) to save the template, you should immediately be able to start using it.

Open or create a Razor view, and type **BeginForm**. If you pause, you should see R# display a tooltip inviting you to complete the template expansion using the Tab key.

If you then press Tab, you should see your template appear, and your cursor be placed allowing you to choose an action and controller for the form.

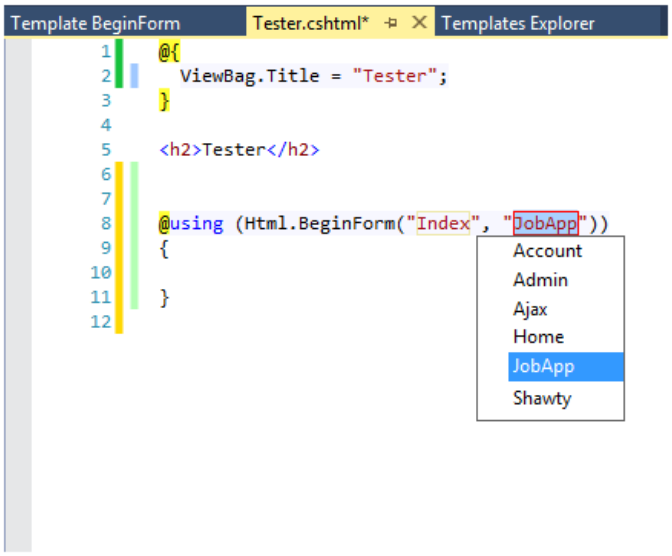


Figure 103: Visual Studio showing our template expansion in progress

A small word of warning: when you're defining your template names (especially with things like Razor) don't be tempted to put an at sign (@) in front of the template name. Being able to type and expand @BeginForm would be great, BUT… the Razor interpreter will try to make sense of it, preventing R# from being able to manage it. The worst part about it is, you'll often not realize right away why your template expansion isn't working. The first time I encountered this, it had me scratching my head for a good few hours.

For the next example we'll create a surround/wrap template that will help us to create anchor tags wrapped in an `<li> </li>` (the type of layout that would typically be used in a navigation menu, for example).

The process for creating a surround template is nearly identical to that of a live code template, except for one crucial difference. Instead of using `$END$` to mark the end of template expansion, you use `$SELECTION$` to mark the bit of code in your document to be wrapped.

Unlike with Live templates, you'll find that in your settings section, all you have is a description. This makes sense, as there is no shortcut code to trigger the process. Instead, the option will be displayed from either the menu or the template chooser, depending on whether you decided to add the template to the quick list or not.

For the surround example, create a new template in the Surround Templates tab of the Template Explorer, then enter the following code and settings:

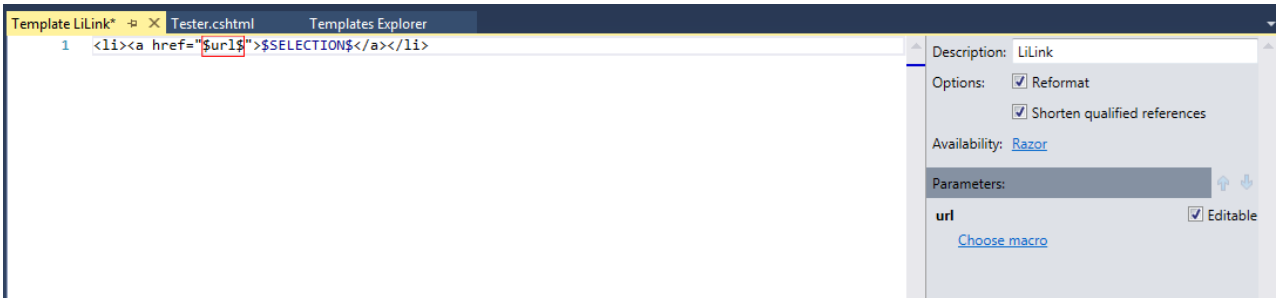


Figure 104: Our example Surround template

As you can see in Figure 104, I've left the URL parameter as **Choose macro**. This is intentional and will allow the end user of the template to type their own choice of content into the place holder.

Once we save our template, we then use the Template Explorer to move the new template entry so that it appears in our quick list.

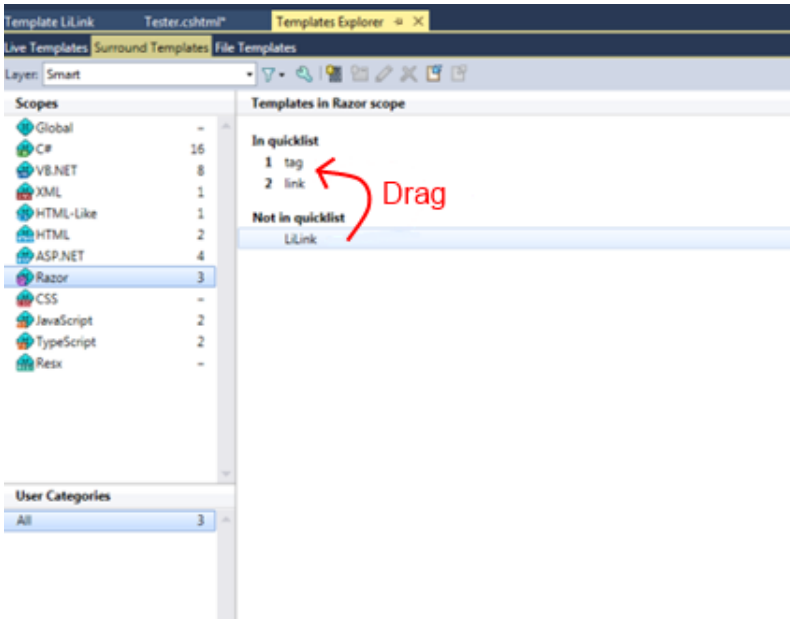


Figure 105: Moving our Surround template to the quick list

If we switch back to a Razor view and try it, we should see the following:

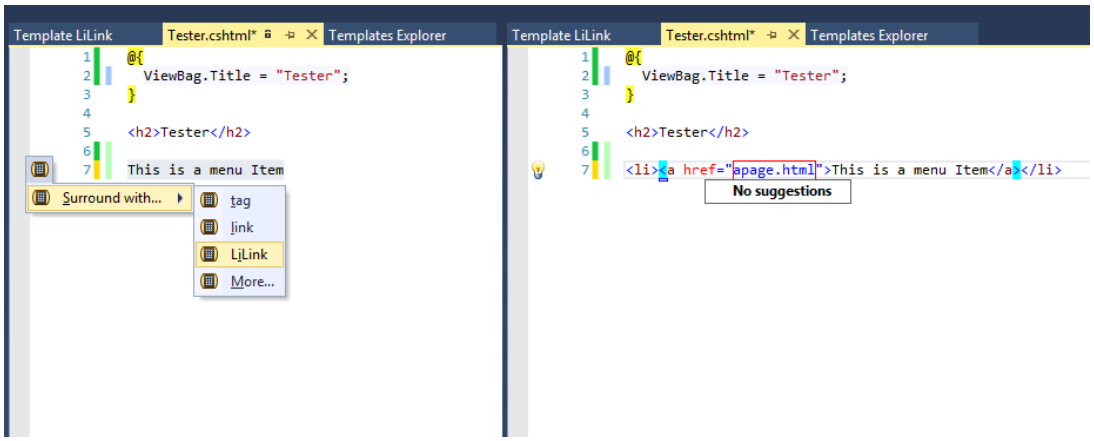


Figure 106: Our new Surround template in action

I'll leave an exploration of creating a new file template to the reader at this point; the process is exactly the same as for the other two.

The last thing to know about live templates are the remaining predefined template variables.

You've already met `END` and `SELECTION`; there are two more in the form of `SELSTART` and `SELEND`. These remaining two variables are used to mark the start and end of a section of template text that you would like to be preselected once the template expansion is complete.

The idea here is that you can place some default text in the template (for example a `Console.Writeline` statement in an exception handler), but you can have it preselected so that as soon as the end-user starts typing anything new, it gets automatically replaced.

The remaining entries on the Edit menu are mostly involved with simple cursor movements.

Extend Selection and **Shrink Selection** deal with intelligently changing the selection size, and will attempt to extend to cover words and variable names, all while taking into account things like underscores, coding style, and white space.

Duplicate text (Ctrl+D) will make a duplicate of the currently selected text after the selection; if the selection covers a carriage return, then the new text will be placed on the next line.

This is useful for quickly duplicating similar lines that have small modifications with relative speed.

Comment with Line Comment and **Comment with Block Comment** again do exactly as the names suggest—they select an item of text and either enclose the entire multi-line selection in a block, or just place the // at the front of the first line to mark that line as a comment.

The Rearrange Code submenu is designed to allow you to move entire blocks of code around in your file, for example, to assist with reordering function calls or property declarations.

Finally, we have the **Paste** option, which opens a clipboard history window containing the most recently used pastes in the project, allowing you to potentially go back to a previous snippet and paste it again. This option is followed by the **Copy XML Doc ID…**, which copies the XML Documentation node ID for the selected code member to the clipboard. The last option, **Copy Resource Value,** will copy the value associated with the selected resource file identifier to the clipboard, ready to be pasted elsewhere.

Chapter 6 Code Inspection Tools

One of ReSharper's main features is its ability to tell you where you can improve your code, and while a lot of this functionality is pretty much automatic, there are a number of things hidden under the hood to help you even further.

Like other tools, most of these are accessible from the main RESHARPER menu under the Inspect submenu.

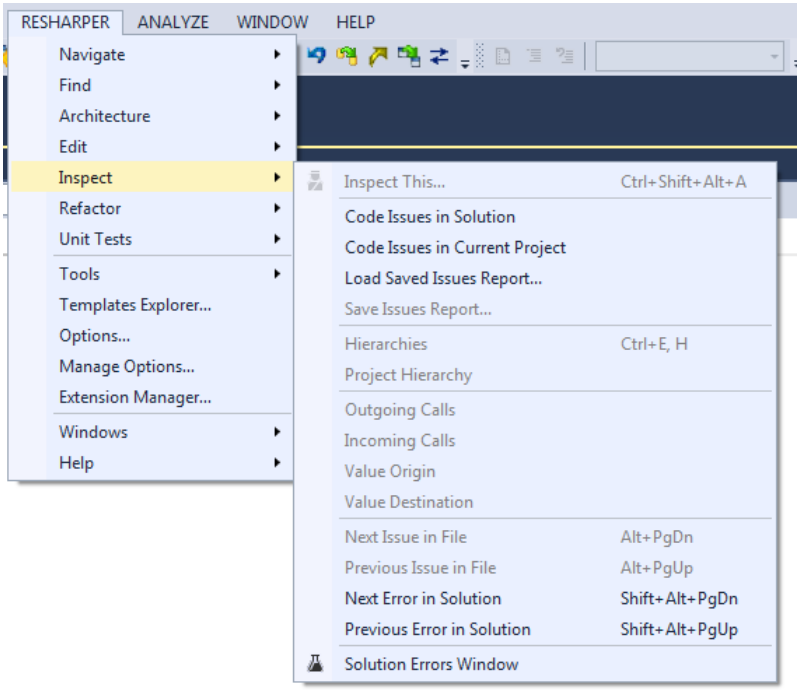


Figure 107: R# Code Inspection menu

Everything here, like the majority of R#'s toolset, is context-sensitive, so the options you have available will depend on where you are in your project.

To kick things off, we'll start by examining the global options, which can be seen in Figure 107.

When you're working with R# as your guide, you'll already be aware that you get different colored squiggles, warning and error ticks, and other helpful indicators.

In many cases, you don't actually need to address those as you see them. Using the **Code Issues in Solution**, **Code Issues in Current Project**, and **Load Saved Issues Report**, you can get the overview of anything that R# thinks needs your attention.

As the name suggests, one option covers the entire loaded solution and all its projects, and the other covers only the project you currently have open in your editor.

Opening a project view in my editor and selecting **Code Issues in Current Project** gives me the following view:

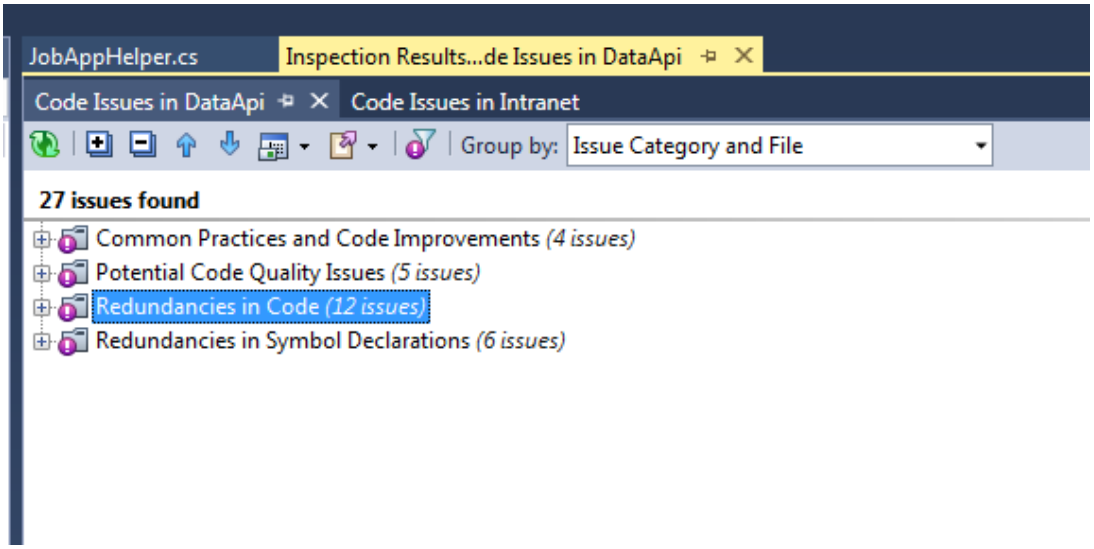


Figure 108: Code issues in Current Project

The window that opens for your project is exactly the same as that for a solution, just likely with fewer entries, depending on how large the solution is.

You can immediately see that by default, issues are grouped into functional areas. This is deliberate so that your thought train remains in similar areas of concentration when working through the items in the tree. As developers, we're all too painfully aware of what it's like to context-switch between vastly different subjects. Dealing with potential code changes, errors, and changes is no different.

Jumping from, say, an advisory that you can change something to a `LINQ` expression, to a bunch of missing symbols and syntax errors, is just as challenging as changing tasks, so to me, the default organization shows great thought from the product designers regarding how developers might use R#.

In Figure 109, you can see that all the issues I have in the project I'm looking at are warnings, and if we expand the redundancies in the code tree for example, we can clearly see what the issue is. If we then double-click on any of those issues, we'll be taken directly to the location in our project where that issue occurs.

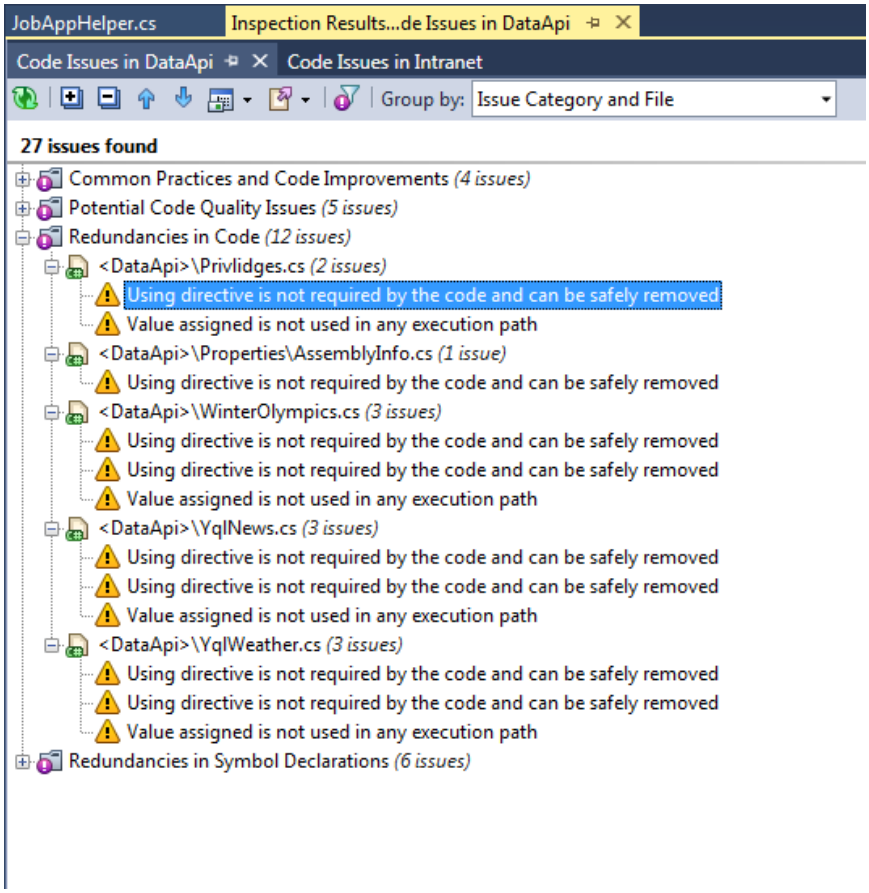


Figure 109: Code issues in project expanded to show warnings

If you think back to previous chapters where we covered finding similar issues, you quickly see that just from one entry in one project, you could potentially eradicate many issues with just a few key presses and mouse clicks.

If you're the tech lead on a project, it gets better. Just above the issues tree you'll see a toolbar that allows you to effortlessly move from issue to issue, collapse and expand all nodes, and more.

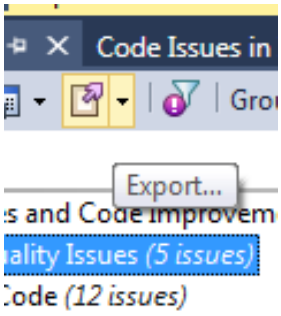


Figure 110: Export button on Issues toolbar

The Export button shown in Figure 110 allows you to export the entire list to a file, which can then later be reloaded using the **Load Saved Issues report** option on the Inspect menu.

With this option, you can isolate sets of issues for specific team members, and then pass those lists to team members in an email, or attach them to a bug tracking system, allowing you to easily delegate tasks and keep track of what's being done.

Inspecting variable value flow

If you open a source file in your project, then move to and optionally highlight an assignment to a variable or property:

```
29      using (IntranetData intranetData = new IntranetData())
30      {
31        var allInfos = intranetData.JobAppHelperSections.Where(p => p.OwnerWorkStationId == workStationId);
32        if(allInfos.Any())
33        {
34          results = Mapper.Map<List<JobAppHelperSectionVM>>(allInfos);
35        }
36      }
37
38      return results;
39    }
40
```

Figure 111: Code snippet showing a variable being used

You can then open the R# **Inspect** menu and choose the **Value Origin** option. R# will open an inspection window showing you the exact source of the selected variable, allowing you to instantly pinpoint where its value was last set.

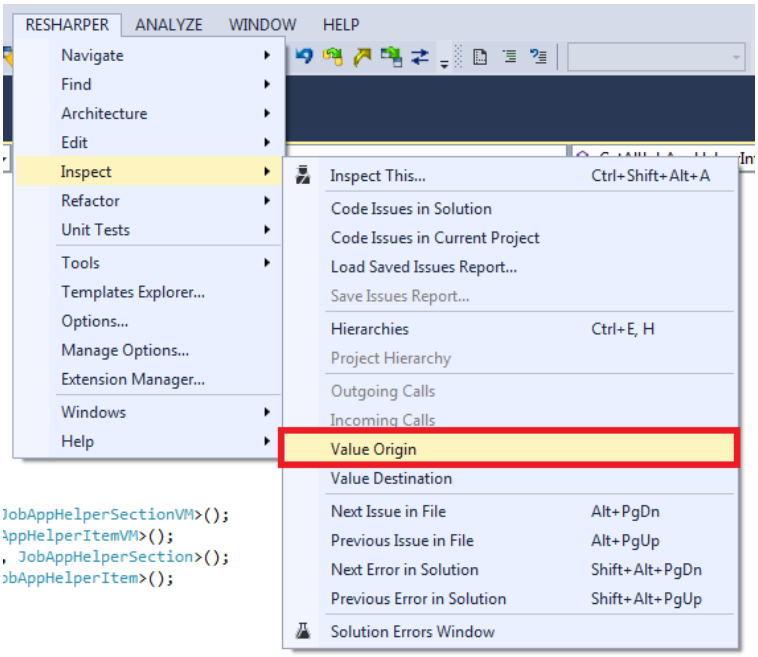


Figure 112: R# Inspect menu showing Value Origin option

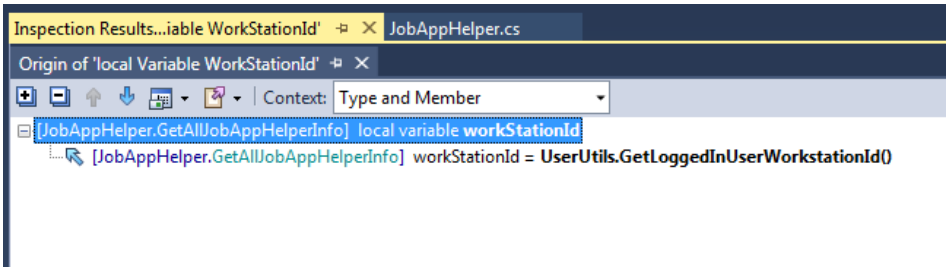


Figure 113: Inspection results showing the origin of a variable

Pinpointing the destination of a variable is just as easy—select **Value Destination** from the same menu, and you can just as easily go the other way. Using these two options allows you to easily track the progress of a value through any piece of code without having to read and follow every line.

As an example, this helps a great amount when following code in an `if-then` construct, as it gives you the option to traverse deep into any of the clauses and then instantly backtrack to get the source value used to make the decision in the first place. If you're rewriting old code, particularly spaghetti code, then this is a real time saver.

In the code in Figure 111, you can see that we have a custom data context called `intranetData`. If you're working with this code, you might want to know the namespace tree where this custom type is defined, possibly even leading right back to its base object. R# also has this covered.

In your code, position your cursor on a line with a custom data type in it, then select **Hierarchies** from the Inspect menu. This will produce a window showing you the type path leading all the way back up the chain to the base object, and everything in between.

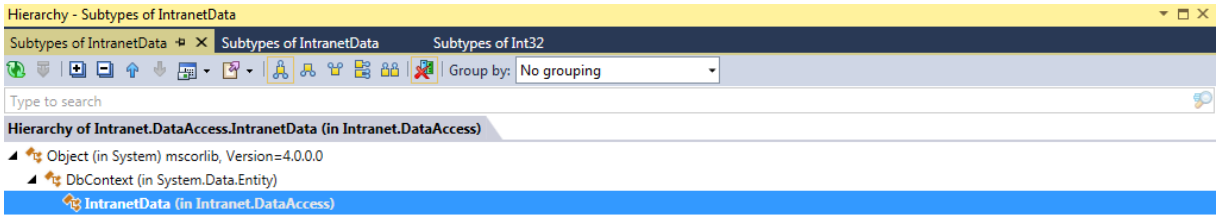


Figure 114: Type Hierarchy inspector

Like any of the other inspection windows, you can double-click on any entry in the tree and be taken to the appropriate location in the appropriate viewer. If the source for a module is available, then a source view will be provided; if not, the assembly viewer or object browser will be used instead.

Inspecting method flows

Variables and properties are not the only things to get the all-star treatment when it comes to inspecting and analyzing your projects. If you open up a source file, and position your cursor on a function call, then use the inspect menu to select **Incoming Calls**, you can easily get a list of each place in your project or solution that a call to that method comes from.

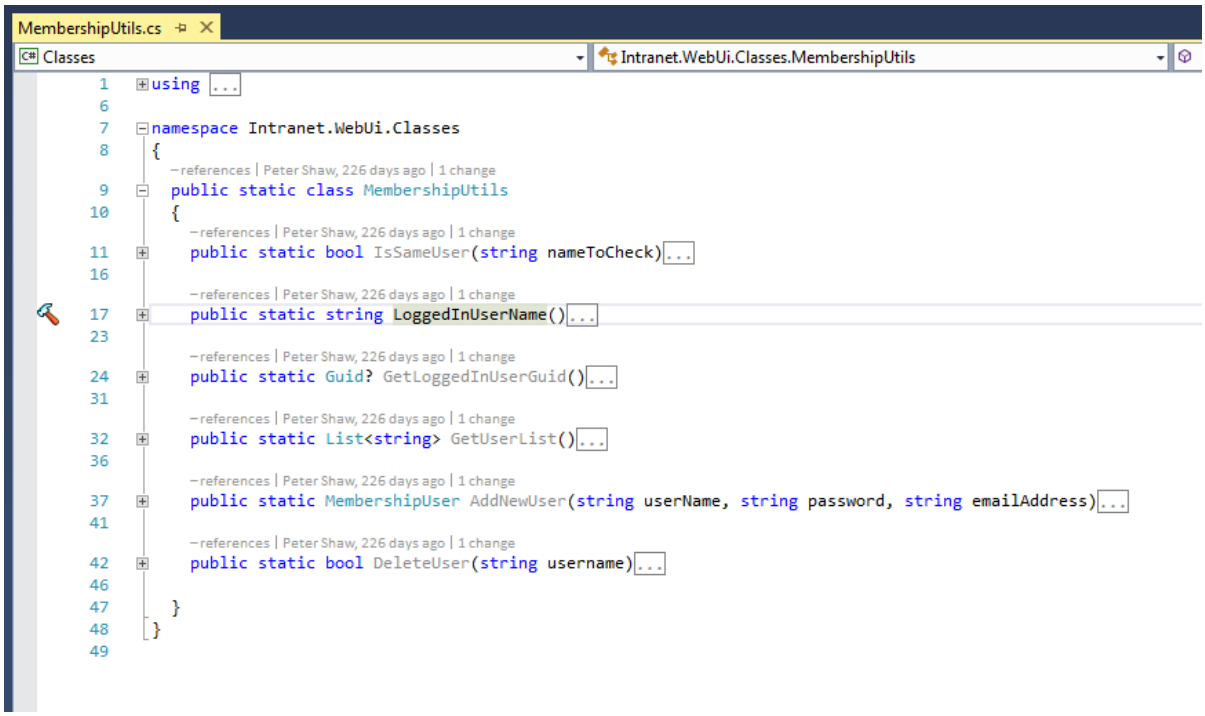


Figure 115: Code sample showing a utility library

In Figure 115 only one usage of `LoggedInUserName` was detected, and this was used in one of the project's Razor views.

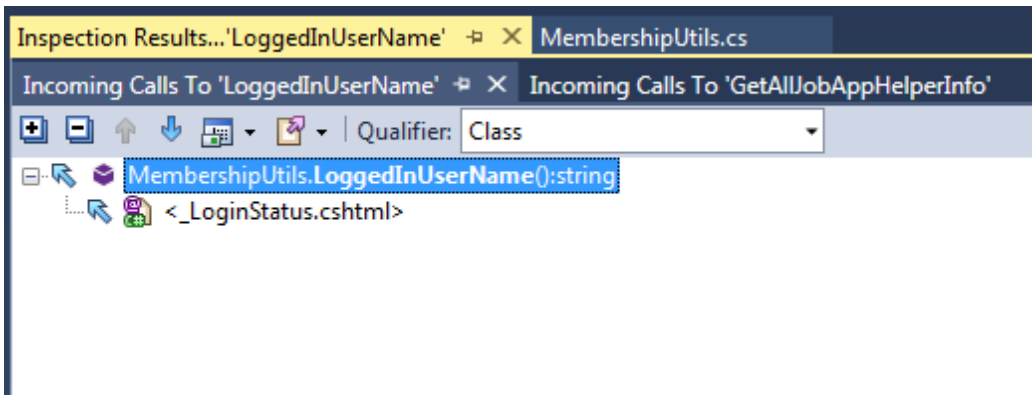


Figure 116: Incoming calls tree for selected method in Figure 115

The inverse is also available—if you select **Outgoing Calls** from the same menu, R# will then show you all the methods used by the selected function or method.

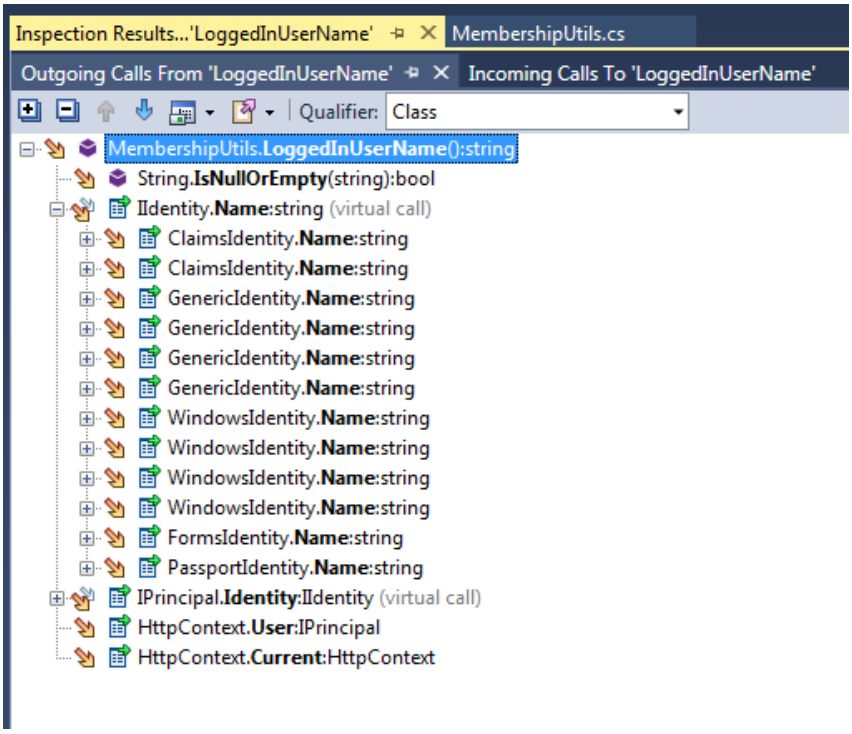


Figure 117: Outgoing calls for the selected method in Figure 115

In a very short space of time, it's possible to inspect a section of code for potential issues, then find exactly what calls, and what is called by, the highlighted code. This ease of navigation and understanding of a complex project makes the task of getting to grips with a new code base easy right from the start.

The remaining options on the inspect menu are all amalgamations of the other issues, or exist to allow single shortcut key-press navigation following the main mantra of R#: allowing the developer to keep his or her hands on the keyboard while coding.

Chapter 7 Code Refactoring Tools

We all know the drill—Red, Green, Refactor—it's the staple diet of agile, test-driven teams all over the world these days.

The problem is, refactoring is a very complex procedure; renaming that critical method in an object, then going and finding every point in the solution where it was used, is a very easy way to use up an entire week.

It comes as no surprise that R# provides some amazingly useful tools to help with this task. In fact, it could be said that the main reason R# came to market in the first place was to service this need, given that the tools in VS at the time where very primitive in comparison.

R#'s refactoring tools are mostly keyboard-driven (more so than most of the functionality we've seen so far); this is primarily because most of the refactoring tools are designed to be used in process while in a typical coding session.

So what exactly is the art of refactoring?

Well that depends on exactly which way you look at it—to some, refactoring is just general code tidying; to others, it's completely restructuring a complex project.

In general, anything that involves renaming methods, objects or properties, or involves moving chunks of code around and generally rearranging things, is classed as refactoring. It’s for this reason that refactoring is probably one of the most volatile tasks a busy developer would have to do.

Imagine for just a moment that you have the following class:

```
public class MyClass
{
  public string returnTwo()
  {
    return "2";
  }
}
```

We can clearly see it returns a string. Now let's suppose we use this as a utility class in lots of places in a huge web project. In fact, every time we want to present the number 2 in the web layer, we call this method, in this class.

As you can likely visualize in your head, this could be a lot of inbound calls.

Now let's suppose we want to move this method into a utility class that exists outside the web layer. This will allow us to reuse the functionality in other places where we need a number 2 as a string.

Typically, without R#'s help, this would involve us copying and pasting the class into a new file, then going through our solution looking for all the references to the class or method and fixing the links up manually so that things compile once again.

With R# we can simply place our cursor on the method or class signature, and with a few key presses, have everything changed in a matter of minutes.

Let's suppose we have the following class in our web project:

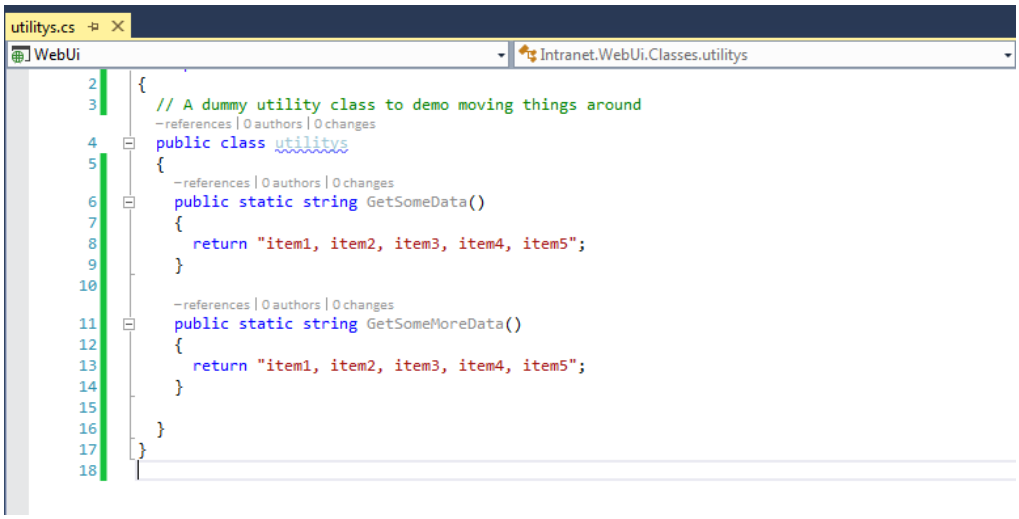


Figure 118: Demo class in our web project to refactor

And let's suppose we have the following usages:

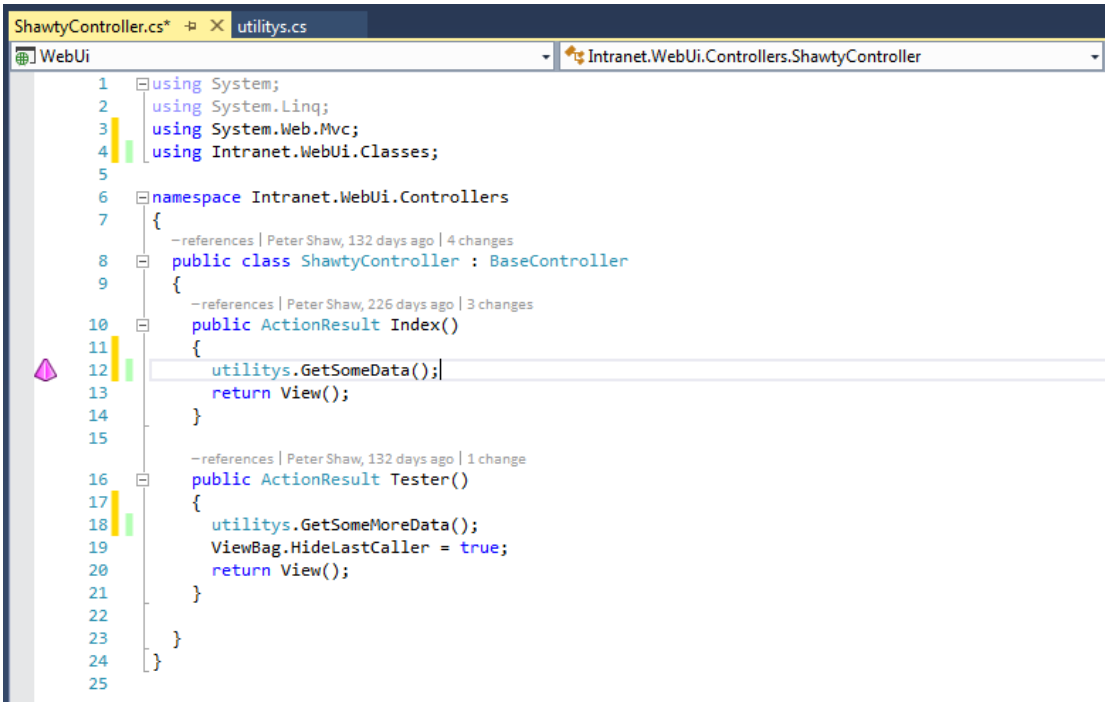


Figure 119: Usages of our demo code in Figure 118

As you can see, we have the correct `using` statement in our usage code and all is happy.

Now let's rename one of our methods.

```
public class utilitys
{
    -references | 0 authors | 0 changes
    public static string GetSomeDataRenamed()
    {
        return "item1, item2, item3, item4, item5";
    }
```

Figure 120: Renamed method in our demo code

You'll immediately see that the code where we use that method starts showing as an error.

```
public ActionResult Index()
{
    utilitys.GetSomeData();
    return View();
}
```

Figure 121: Usage code showing an error after a rename

In this case it's only one instance, but it could easily be 100. If we used R#'s refactor methods to rename this however, we could have done all of those renames all in one go.

Ctrl+Z the renamed method, then position your cursor on the now original name and press **Ctrl+R**, **R** (that's Ctrl+R twice in rapid succession), and you should be greeted with the following:

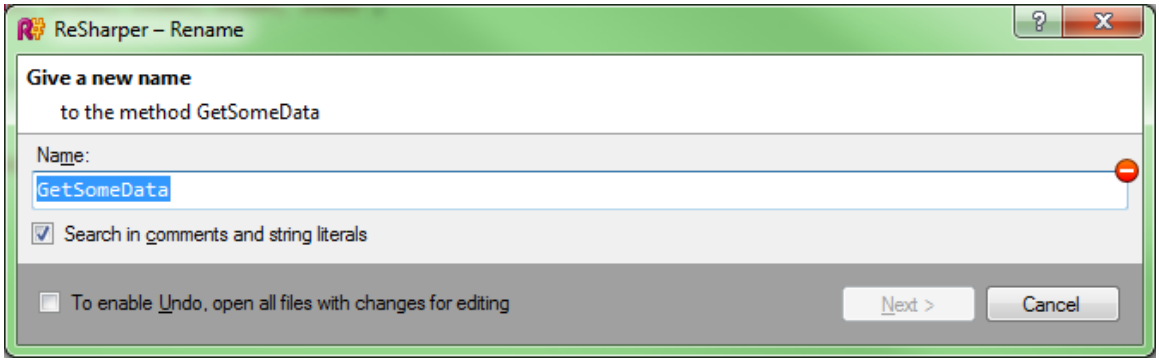


Figure 122: R# reference Rename dialog box

Now type your new name in the name field, then click **Next** and go through the wizard defaults.

As you can see, you've renamed your method in its source file.

```
public static string GetSomeDataRenamed()
{
  return "item1, item2, item3, item4, item5";
}
```

Figure 123: Source method renamed

If you look at your usages, you also see that R# has found and renamed those for you too.

```
–references | Peter Shaw, 226 days ago | 3 changes
public ActionResult Index()
{
  utilitys.GetSomeDataRenamed();
  return View();
}
```

Figure 124: Usage renamed by R#

The time savings alone can add up to hours, not just minutes, and this is just the tip of the iceberg.

Let's once again undo all of the renames we previously performed, then go back to our dummy class. This time, use the "Refactor, Move" key press **Ctrl+R**, **Ctrl+O**. You should get the following dialog box:

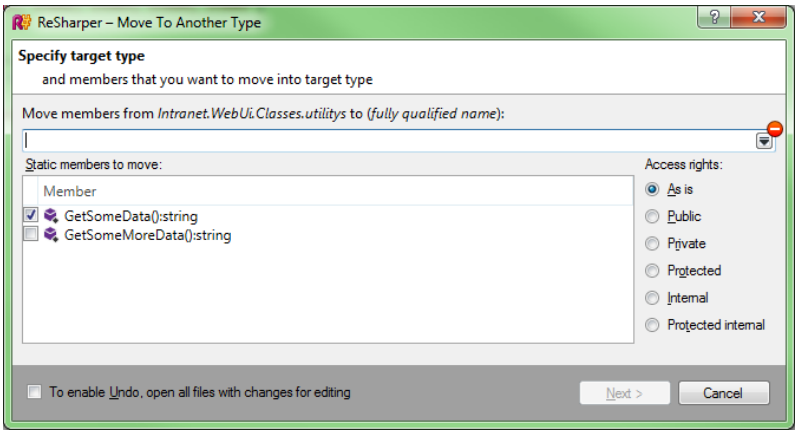


Figure 125: R# Refactor, Move dialog box

In this case, we only have one method selected, but we could easily select all of them if we wanted to. You can also see that we can change the access type when we move, should we wish to.

If you click on the small drop-down arrow at the end of the text input box, you'll get a navigator that allows you to pick where in your solution you wish to move the method.

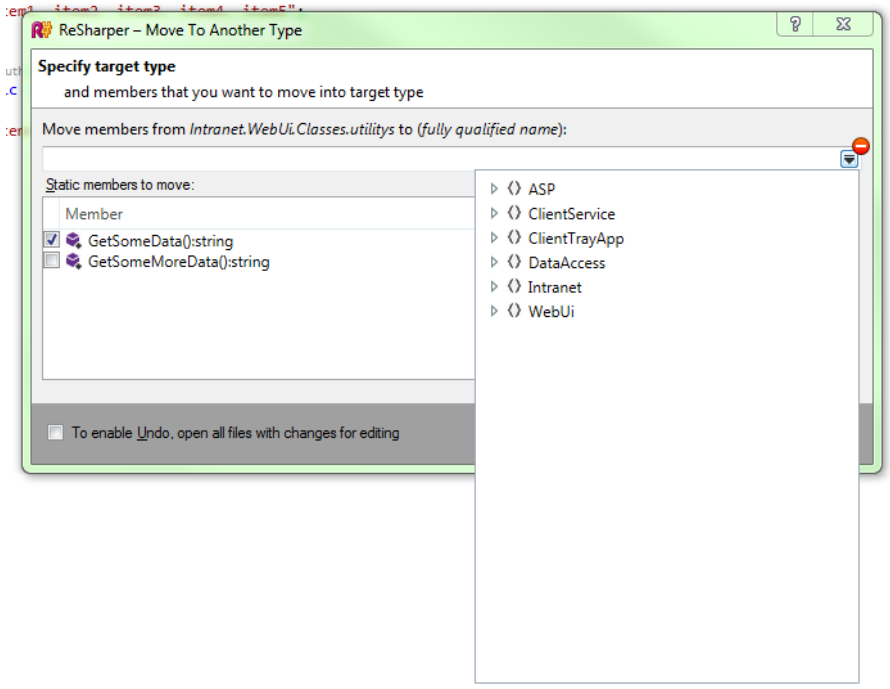


Figure 126: Refactor, Move dialog box showing namespace navigator

If I pick a new place to move the method to, R# will perform any work it needs to do to move the method and fix up any used references.

If we open the file I moved the method into, you'll see it sitting there as expected:

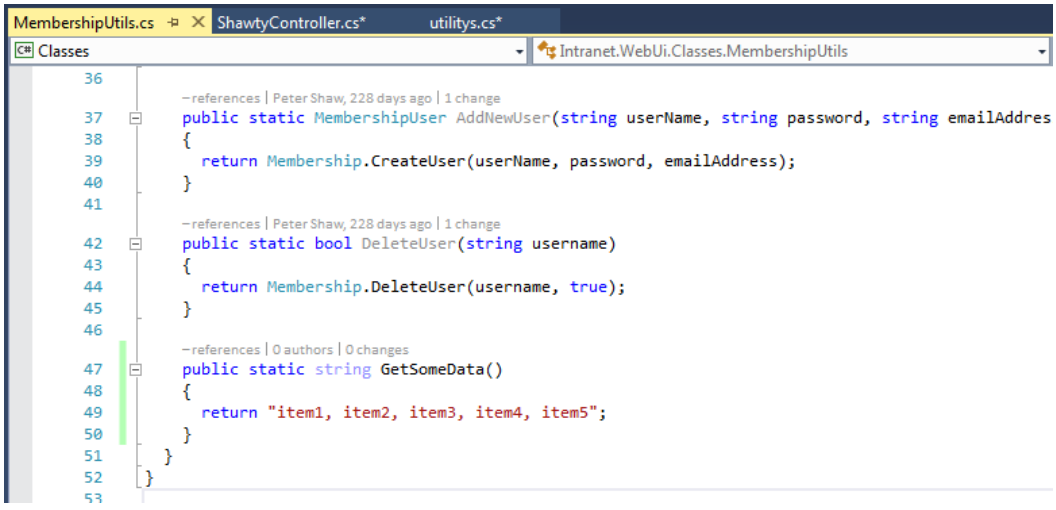


Figure 127: Test method moved to another class

You'll also see that it's been removed from its original file.

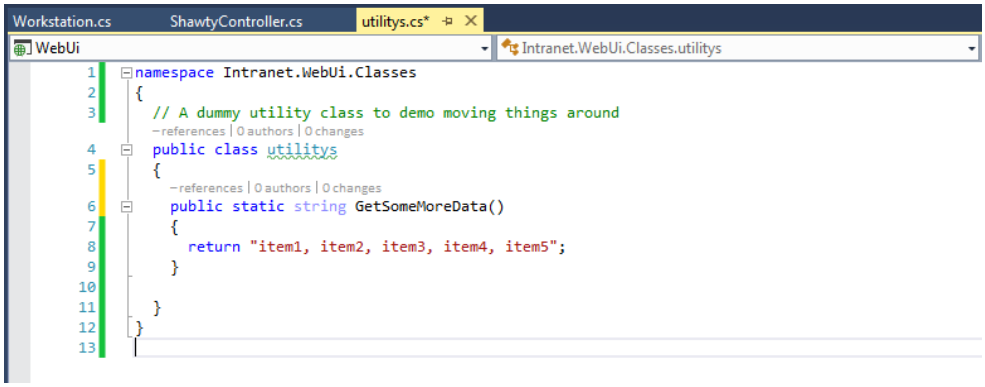


Figure 128: Dummy class with test method removed

And our usage has been fixed up to reference the new location of the method.

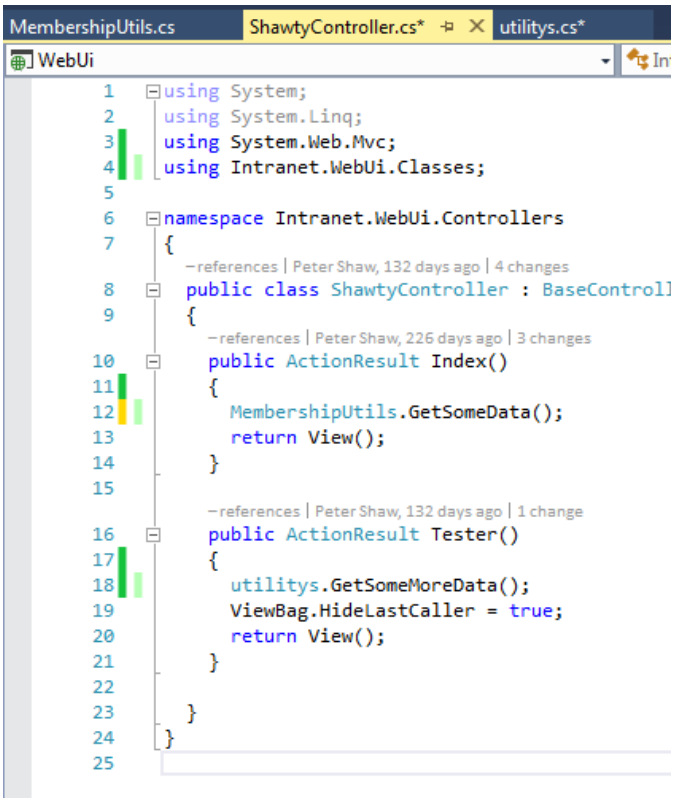


Figure 129: Usages fixed up after a method move

R# has many other similar operations. The main menu from the Visual Studio toolbar looks like this when a method signature is selected:

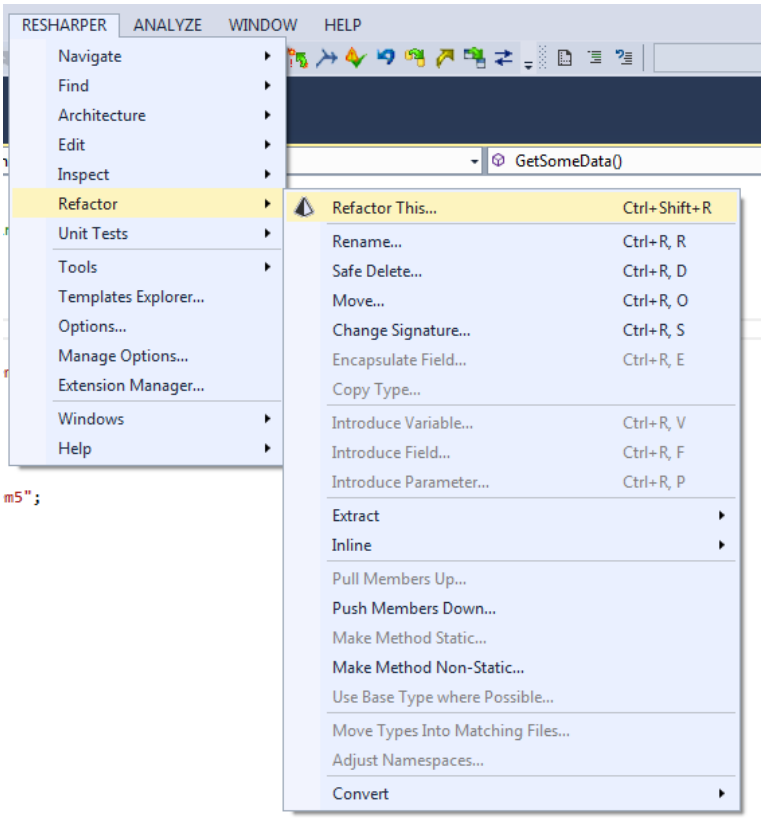


Figure 130: R# main Refactor menu

We've already seen the **Rename** and **Move** options. **Safe Delete** is used to remove a method from your project, and then clean up any remaining usages afterwards.

Change Signature is used to change a method signature, allowing you to swap parameters around, change types, or do things the safe way and construct an override.

Select your method signature and press **Ctrl+R**, **Ctrl+S**, and you should see the Change Signature dialog box:

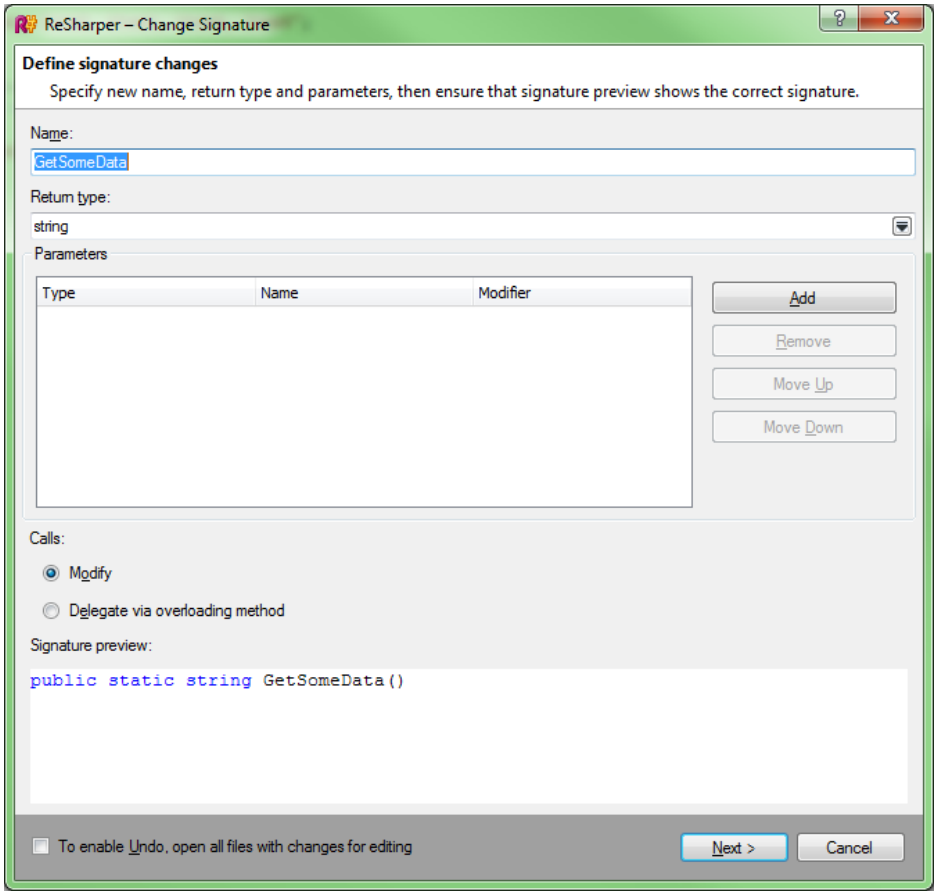


Figure 131: R# Change Signature dialog box

For our demo method, we have no parameters currently on it—let's change that and add some. We'll make this a modification rather than a delegate. Make sure that the **Modify** option is selected, then click **Add**, set the type to **`string`** and the name to **`dataName`** with no modifier, and then click **Next**.

Because we're adding something that wasn't originally there, we'll see a further dialog box:

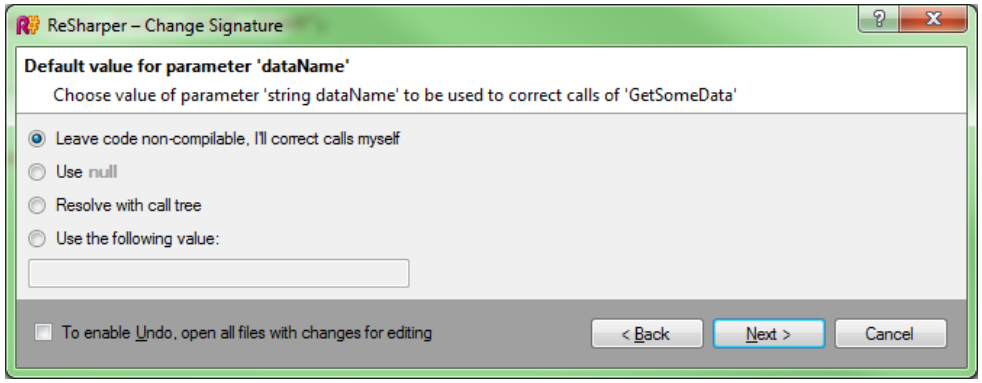


Figure 132: Parameter addition dialog box

As you can see, R# will ask you how you'd like to set the value of the new parameter when it makes changes to the signature at the points of usage.

The default is that you need to change things yourself (you'll get compiler errors when you build). You can add a null value, set a static value, or build a call tree, all designed so you can maintain an error-free compile while altering what you need to.

For this example, I've chosen null, which gives the following results:

```
0 references | 0 authors | 0 changes
public static string GetSomeData(string dataName)
{
  return "item1, item2, item3, item4, item5";
}
```

Figure 133: Our test method with a parameter added

As expected, our usage has also been updated to match:

```
0 references | Peter Shaw, 132 days ago | 4 changes
public class ShawtyController : BaseController
{
  0 references | Peter Shaw, 226 days ago | 3 changes
  public ActionResult Index()
  {
    utilitys.GetSomeData(null);
    return View();
  }
```

Figure 134: Usage updated by parameter addition

If you refactor a method to take a new parameter, there's often a use case that the parameter will be used to initialize or set a value in the class the method is part of. R# can help further after a refactor by helping you “introduce” properties and variables into a containing class.

Position your cursor on the string `dataName` parameter you've just added to your test method, then click on **RESHARPER** > **Refactor** > **Introduce Field** or press **Ctrl+R**, **Ctrl+F**. You should get the Introduce Field dialog box that looks like the following:

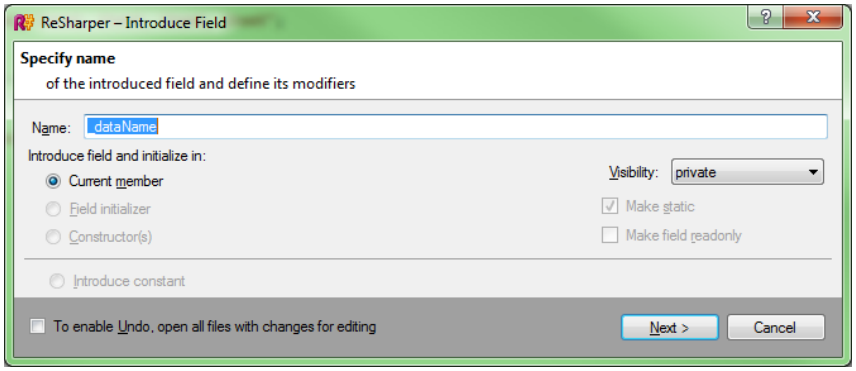


Figure 135: R# introduce field dialog box

From here, you can set the field name, accessibility type, and other settings. Select the options you wish to use, then click **Next**.

You should see that your method source is now changed to look like the following:

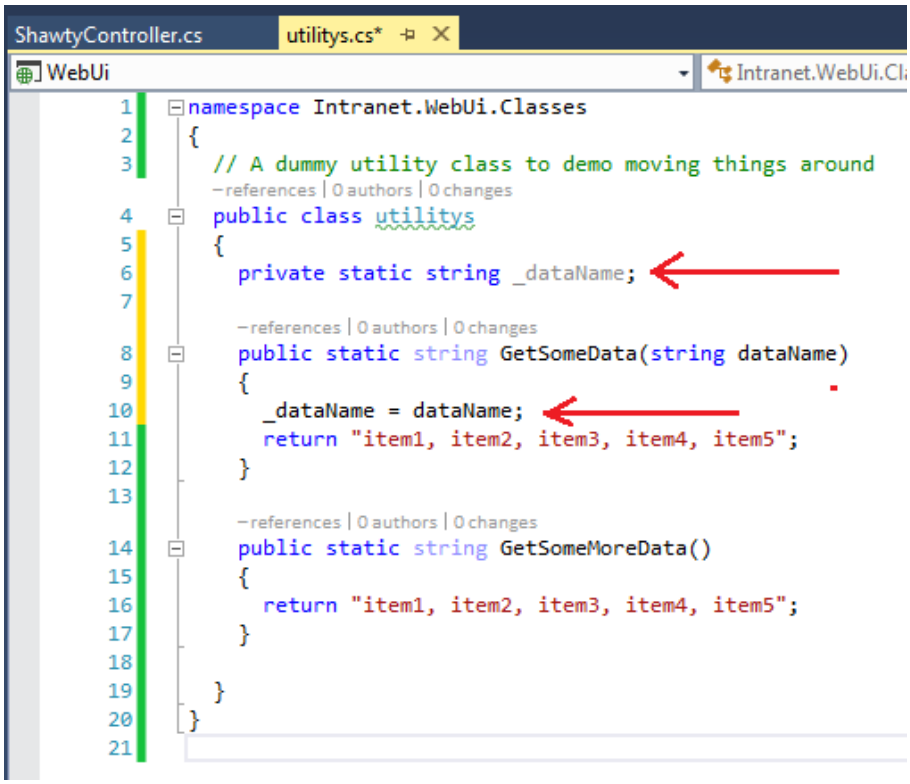


Figure 136: Result of asking R# to introduce a field to our class

It may not seem like much, but you've just saved yourself over 50 key strokes with those couple of mouse clicks. Considering that you can introduce multiple instances very rapidly, you can see how fast you might be able to flesh out a fully decoupled class dependency, ready to consume classes using a Dependency Injection (DI) based pattern.

One of the more useful of R#'s refactoring commands comes into its own when you're copying and pasting methods in from other projects.

Let's imagine we just pasted the code in our dummy class from a different project altogether. We might expect that the namespace is different to where we want to use it:

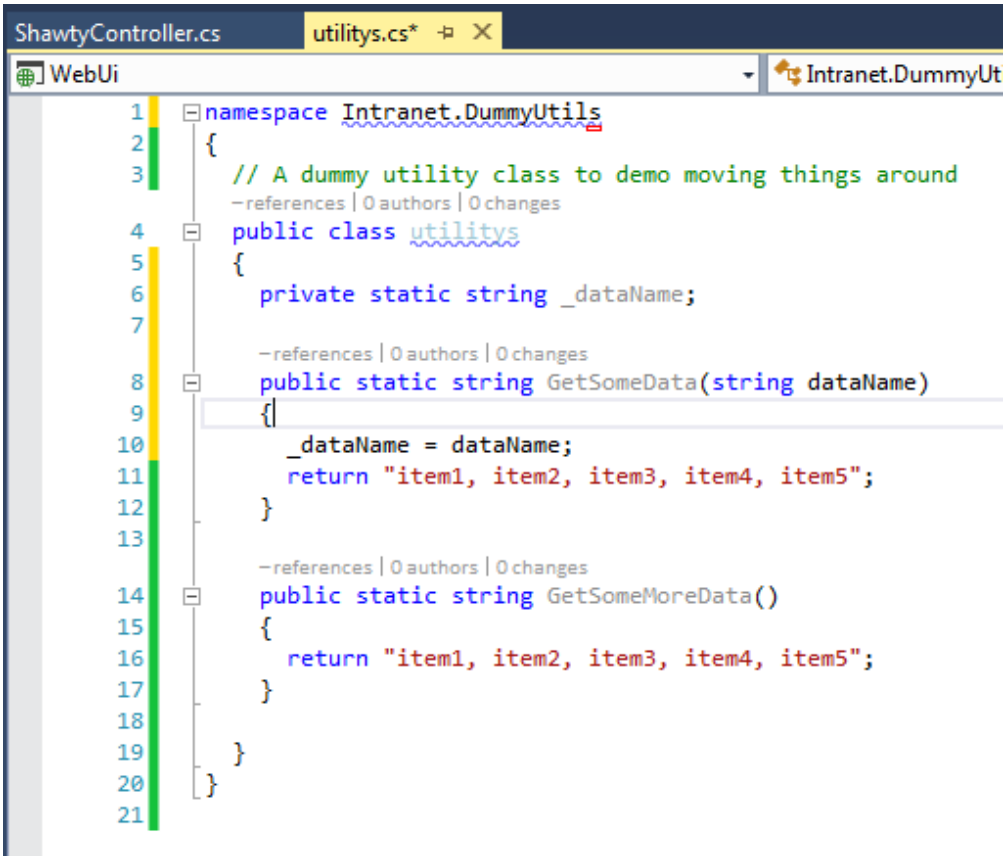


Figure 137: Demo code showing misnamed namespace

As you can see in Figure 137, our class namespace is highlighted with a blue squiggle; this is because R# can see that it's been pasted into a class that lives in `Intranet.WebUI.Classes`, but the actual namespace in the file is `Intranet.DummyUtils`.

At this point I could simply just move my cursor to the namespace line and tap Alt+Enter twice to activate R#'s default fix mechanism. We can also use the extended tools from the various menus.

We can access the namespace fix by right-clicking on the class in Solution Explorer and selecting the **Refactor** submenu.

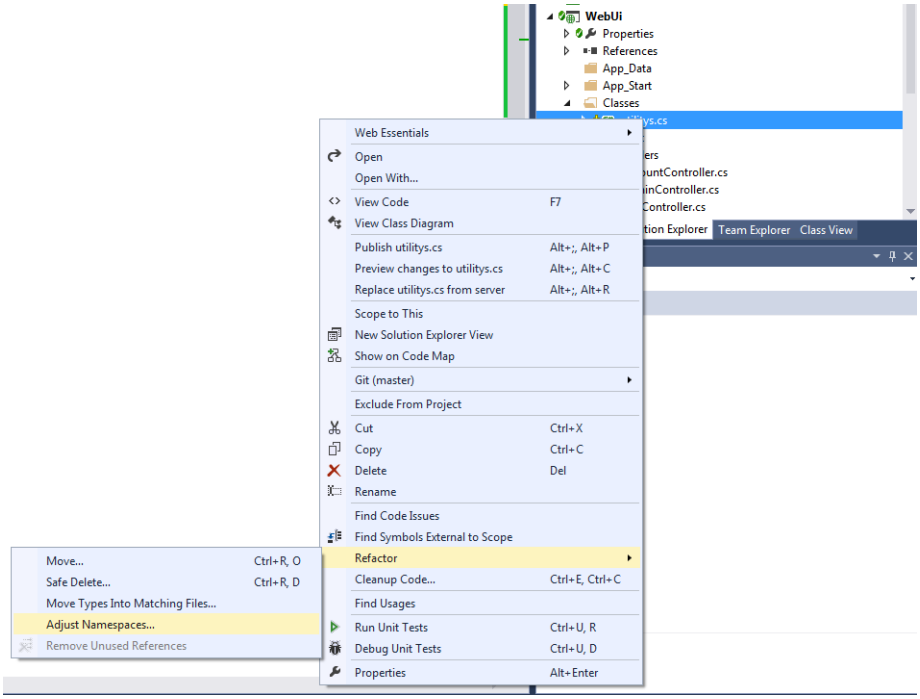


Figure 138: Adjust namespace accessed from Solution Explorer

This will give you a dialog box allowing you to choose which namespaces you want to correct.

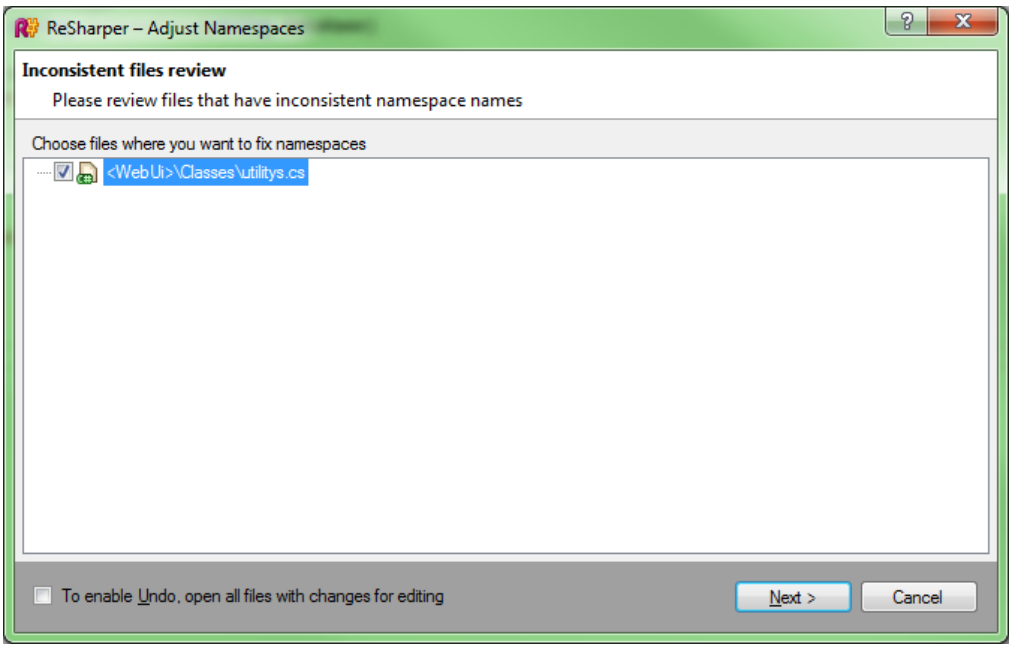


Figure 139: R# namespace choose dialog box

In this case I've only selected one file, but you can also select multiple files (for example, if you just copied some files from a different project), allowing you to correct the namespaces for each of them all in one go. As expected, when you click Next, the namespace is adjusted correctly in each of the files you check in the dialog box.

You'll also notice in the menu above that our old friend **Refactor, Move** is also available from here. When invoked here at a class or file level, it's slightly more powerful than simply using it to move a method, as we did previously.

If you select the **Move** option from here, you'll be greeted with the following dialog box:

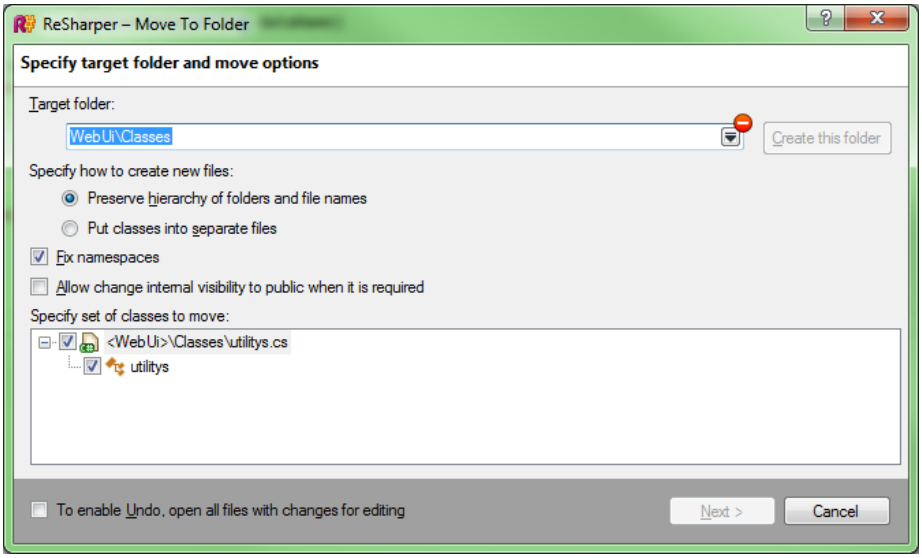


Figure 140: Refactor, Move dialog box when called from the Solution Explorer

As before, you can drop the destination field down to select where in your solution you'd like to move the file to. This time, however, you'll get much more in the navigator:

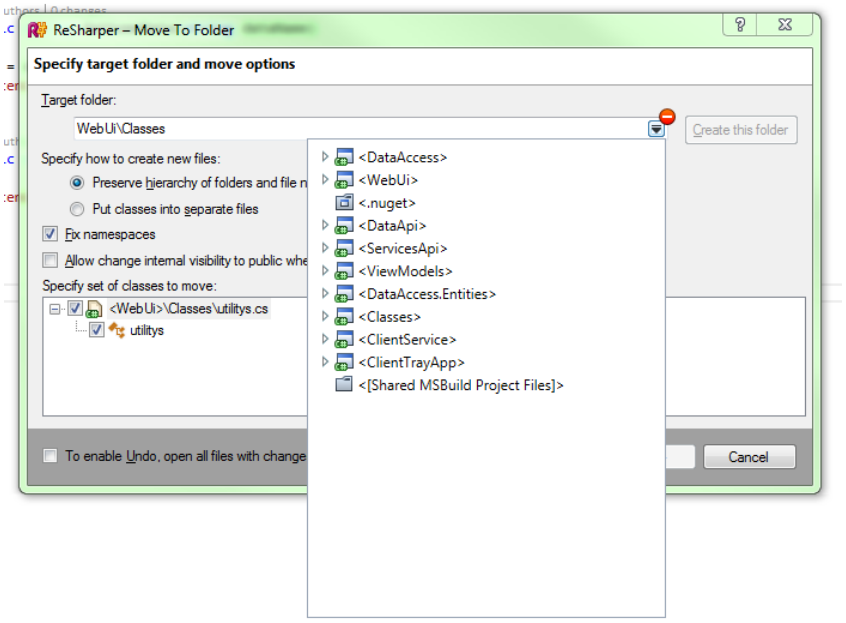


Figure 141: Refactor, Move dialog box with extended navigator

You can see right away that not only do you get to navigate namespaces, but you also get to navigate the full solution hierarchy. Because you're moving an entire file, you get the choice to choose not only its namespace, but its physical location too.

As with the many other refactor options, when you use this, everything in the file will be relocated, usages will be fixed up, namespaces will be adjusted to match the new location, and `using` statements in files that use this class will also be updated to reference the new location.

The main reason purpose of this feature is to break complex code out into simpler units.

If you have a collection of utility libraries that are used in many different places, you may want to put these into a completely separate class library project. The extended move feature helps with this task almost effortlessly, and again, reduces what otherwise might be a half a day's work, into nothing more than a few clicks of the mouse.

There are still more refactoring options

If what we've already covered isn't enough for you, there's still more in the refactoring menu if you dig deep enough. If we follow **RESHARPER** > **Refactor** > **Convert**, we get the following third-level submenu.

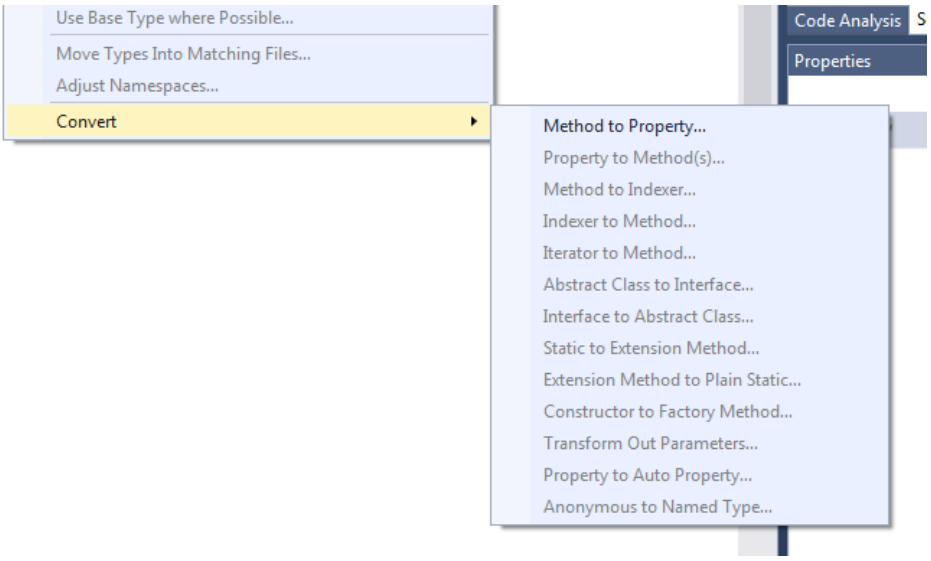


Figure 142: Third-level Refactor, Convert menu

If we position our cursor on a method, we can convert that to a property, as you can see in Figure 142. Others are enabled depending on the member type selected, but you can easily see the extent of the conversions we can do.

In my copy, I don't have shortcut keys assigned to a lot of these, but you can (and you should), allowing you to do ALL of these conversions on the fly, again following the R# mantra of keeping your fingers on the keyboard.

There's still more—if you venture down the **Extract** third-level submenu, you'll find the following:

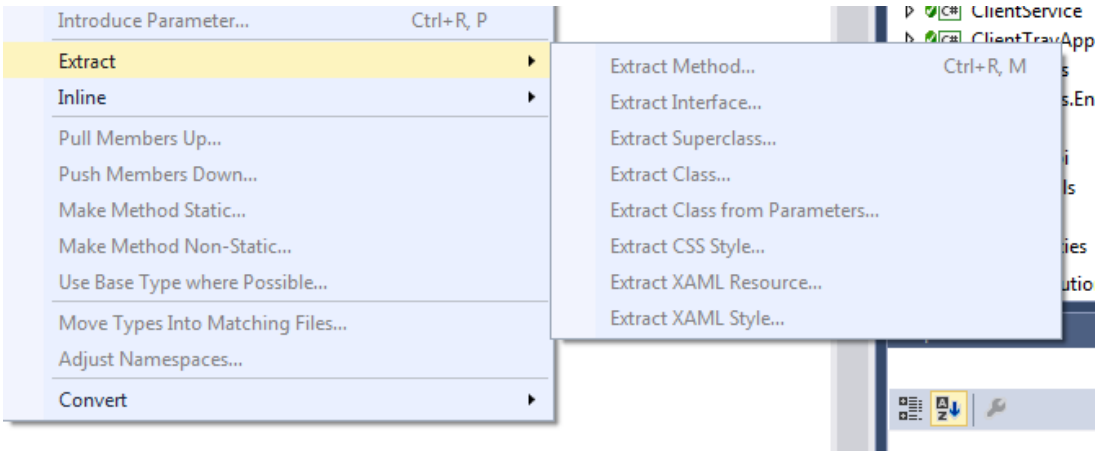


Figure 143: Refactor third-level Extract submenu

The options here are used to extract various entities into separate files and constructs, or to create derived implementations based on the source.

For example, if you're positioned at a class level, you might want to extract an interface, which can then be used to aid with the construction of an interface-driven architecture.

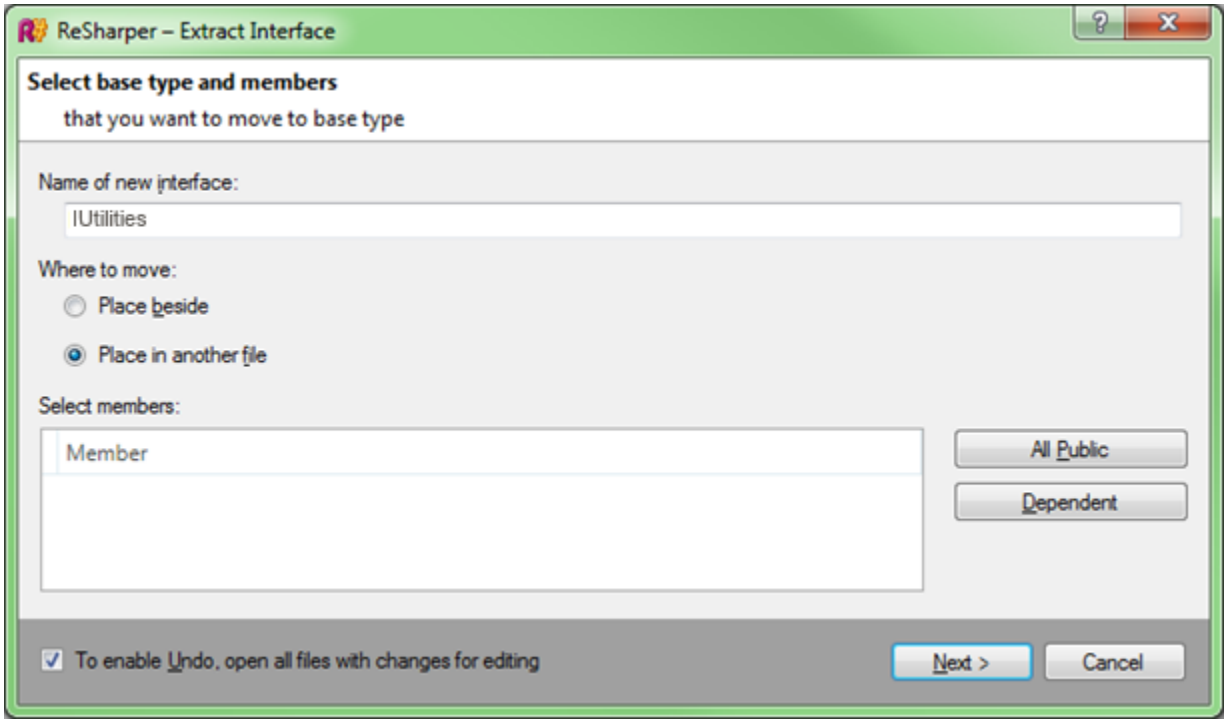


Figure 144: Extract Interface dialog box

The code produced by the choices in Figure 144 is placed in IUtilities.cs and looks like the following:

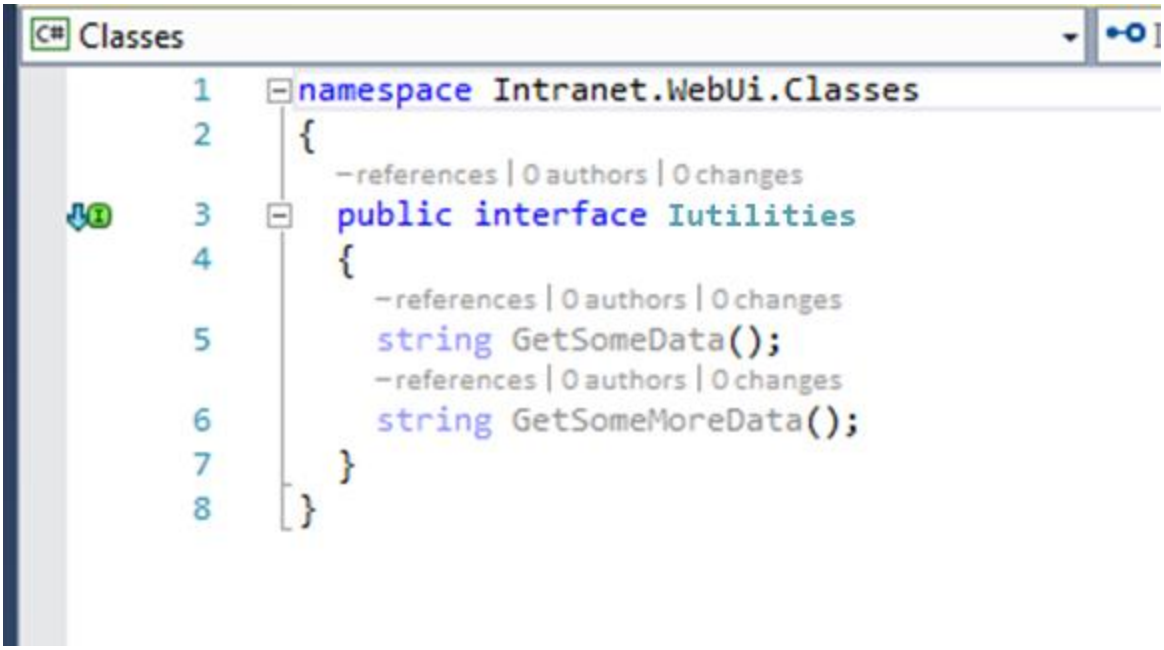


Figure 145: Code produced by extracting an interface off our test class

Your source class will also get updated to use the new interface:

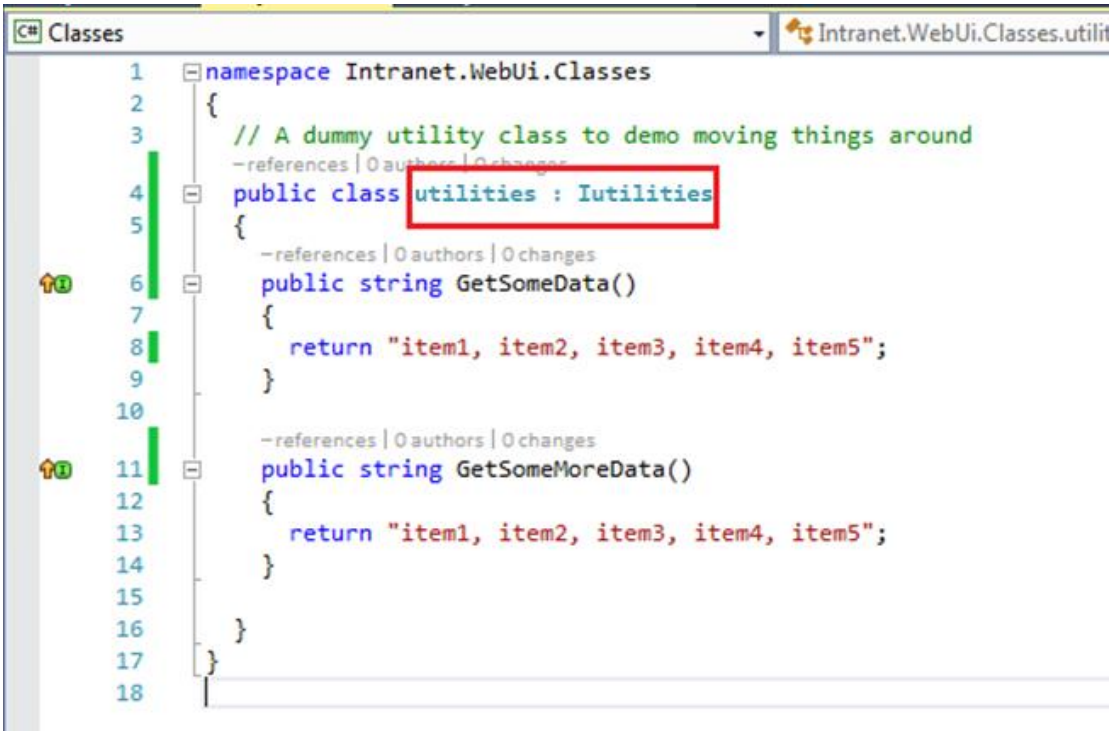


Figure 146: Source file updated to use the extracted interface

There's so much functionality in the refactoring section of ReSharper that you may never even know half of the commands are there. In the process of researching topics for this book, I found functionality that I never knew existed.

All I can recommend is that you explore the menus. Create a dummy project or two, then select and move your cursor to different locations.

The JetBrains website does its best to group things into functional groups so that you can find things based on the job you're looking to do, rather than by ordering in the menus. This goes a long way to helping you explore. However, the best way to learn by far is to just play with the product.

The rest of the refactoring options I'll leave as an exercise to the reader—it's much more fun that way.

Chapter 8 Unit Testing Tools

If you’re a test-driven development (TDD) practitioner, then you'll no doubt know that there's an absolute mass of different tools out there with the sole purpose of making testing easier.

There are more frameworks than you can shake a stick at, and then there are the continuous integration and build tools that will run these tests and perform automated builds for you.

The problem is, most developers actually don't have time to investigate all of these frameworks—all they want to do is to begin writing their tests, then to be able to run them and find out a result.

Unfortunately, what generally happens is that they spend more time learning the subtle nuances of each framework. Then they spend time making sure they can test everything they want to test, and then they spend even more time figuring out how to integrate the test framework into their build environment.

With all these factors in play, TDD nirvana in Visual Studio would be best achieved by just being able to write a test and not worry about how to make it work.

R# brings this option to the table in the form of its Unit Testing tools, and out of the box it supports NUnit and MsTest, along with JavaScript unit test support in the form of QUnit and Jasmine.

On top of that, you can choose to use your default browser or PhantomJS to run your JS test scenarios, and the built-in test runners and session managers for your .NET code.

If you take a look in the **Tools** > **Unit testing options** from the ReSharper **Options** dialog box, you have a very fine-grain amount of control over how the test frameworks run should you need it, but R# is set by default with the best practice defaults, enabling you to immediately just start writing tests.

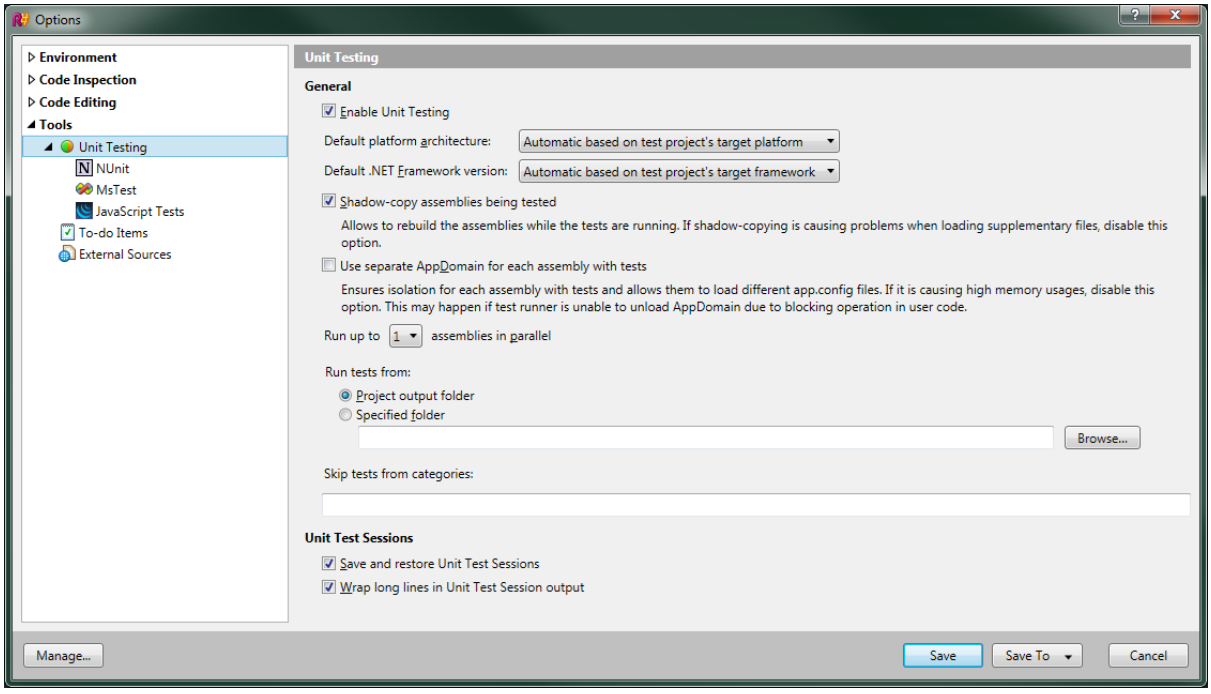


Figure 147: R# general unit test options

Some of the options you have include being able to force tests to run under a specific .NET version or on a specific app domain. You can even relocate your tests so they run from an entirely different location to that of your project, and each of these settings can be saved at project level, so the settings traverse the entire team.

The menu itself is typical of the others we've seen so far, with options being context-sensitive.

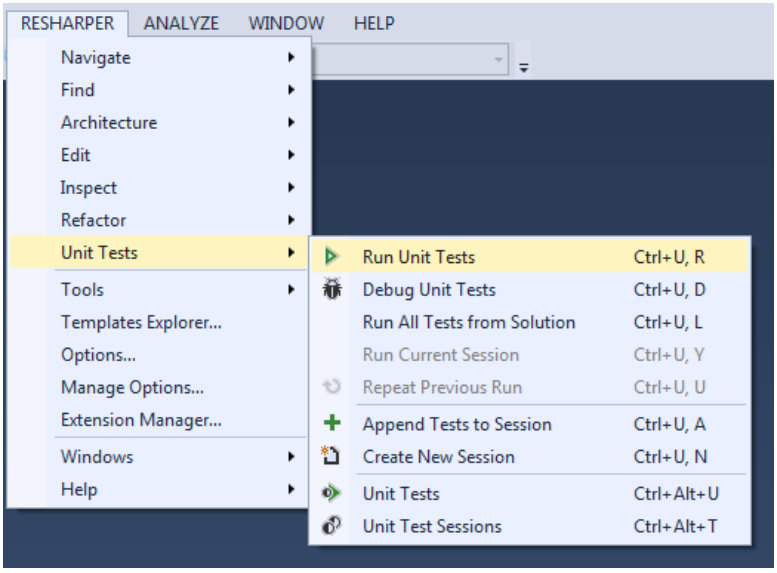


Figure 148: The R# unit tests menu

Personally, I normally only use the bottom two options, **Unit Tests** and **Unit Test Sessions**, the first of which will open the following window:

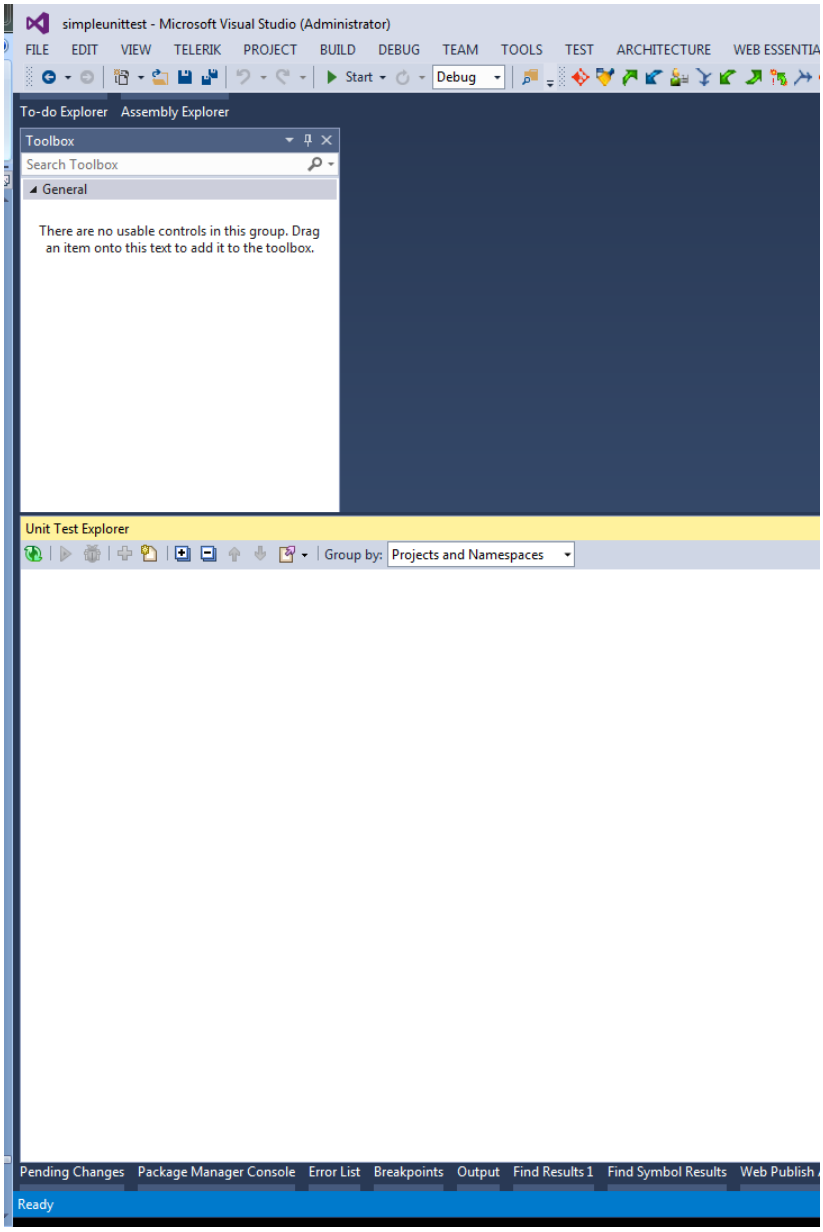


Figure 149: R# Unit Test Explorer

The Unit Test Explorer allows you to explore and manage all the tests you have available from one central location. When you have tests available in your solution, they show up in here grouped by the option you have set in your **Group By** drop-down menu at the top of the window.

The second window, **Unit Test Sessions**, allows you to manage your test runs as individual entities. This is useful because it allows you to do things like comparing current and previous runs side by side to observe the differences.

When selected, the Unit Test Sessions options will open the following window:

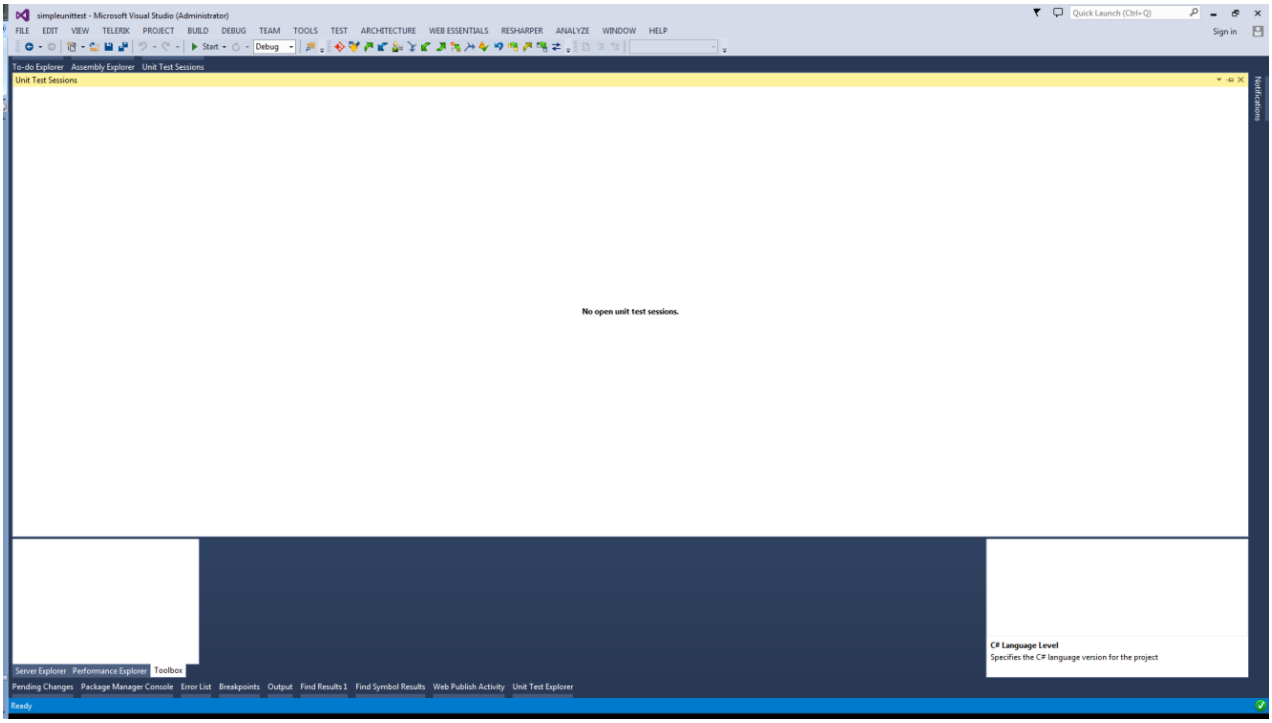


Figure 150: R# Unit Test Sessions window

A sample project

For the purposes of showing the functions available, I've created a very simple Console mode project into which we're going to add a couple of dummy arithmetic classes. We'll then manage the tests for these classes using the various R# options.

To start with, we just have a simple, empty project:

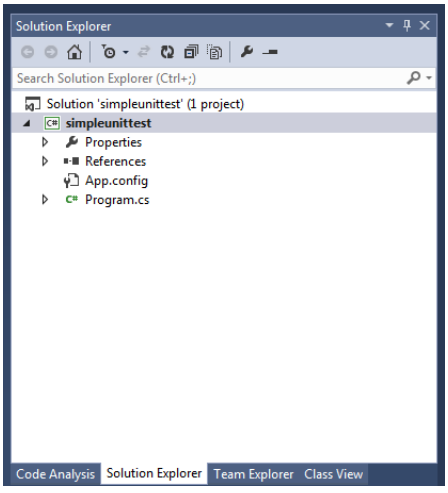


Figure 151: Sample project ready to have tests and classes added

Our specifications tell us that our arithmetic engine needs to provide functionality to add and subtract fixed values and a percentage, and to keep a running total. We should also be able to set an initial level when we create a copy of the class.

To get started, we'll create the following class:

```
namespace simpleunittest
{
  public class Arithmetic
  {
    private decimal _runningTotal;

    public decimal RunningTotal
    {
      get { return _runningTotal; }
    }

    public Arithmetic(decimal runningTotal)
    {
      _runningTotal = runningTotal;
    }

    public decimal AddValue()
    {
      return 0;
    }

    public decimal SubtractValue()
    {
      return 0;
    }

    public decimal AddPercentage()
    {
      return 0;
    }

    public decimal SubtractPercentage()
    {
      return 0;
    }

  }
}
```

In addition to the required functions, each function should also return the new running total as a result. We should also have a constructor that creates a copy of the class with a default value of 200 in the running total, and a public accessor that allows us to read the running total at any time, but not modify it.

First we'll add a regular unit test project to our solution—right-click on your solution entry in Solution Explorer, then add a new project. Select **Unit Test Project** for the project type.

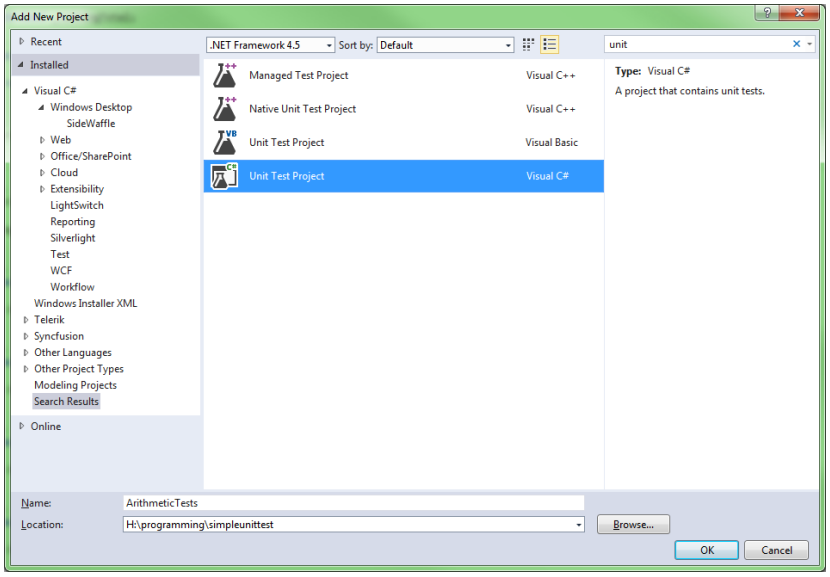


Figure 152: Creating a unit test project

As is usually the case, Visual Studio will create a default test feature. However, if you look closely at the new file in the VS IDE, you'll see it looks a little different than usual:

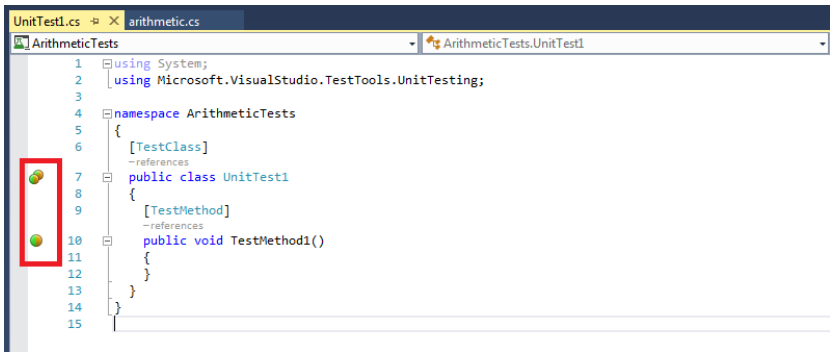


Figure 153: Initial test fixture created by Visual Studio

You'll notice that R# has been working its magic and added its own little icons in the left margin; these markers are R#'s control icons for your tests.

By clicking on an individual icon (such as the lower one highlighted in red in Figure 153), you can run just a single test, directly from within your test project. The upper icon (that looks like multiple icons) allows you to run all tests in a class.

The icon also serves as an Alt+Enter quick menu icon, and if you press the default R# key combo when your cursor is on the method signature, you should get the following context menu:

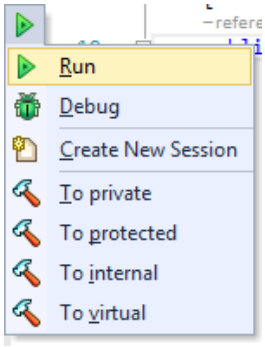


Figure 154: R# Alt+Enter menu for test fixtures

R# brings something to the unit test table that other frameworks do, but it provides it in a more convenient form, and that's the ability to set breakpoints on your tests and then debug them the same way that you use Visual Studio to debug production code.

We'll come back to the debugging side of things in a little while, but for now let's create a test to new up and make sure our default constructors work.

I've created the following two tests for the constructors on our `Arithmetic` class, but as you can see, the class entries are highlighted in red to show that the unit test project can't find them.

This is a good point to use some of the tools seen in previous chapters to get R# to add the appropriate references for our test class.

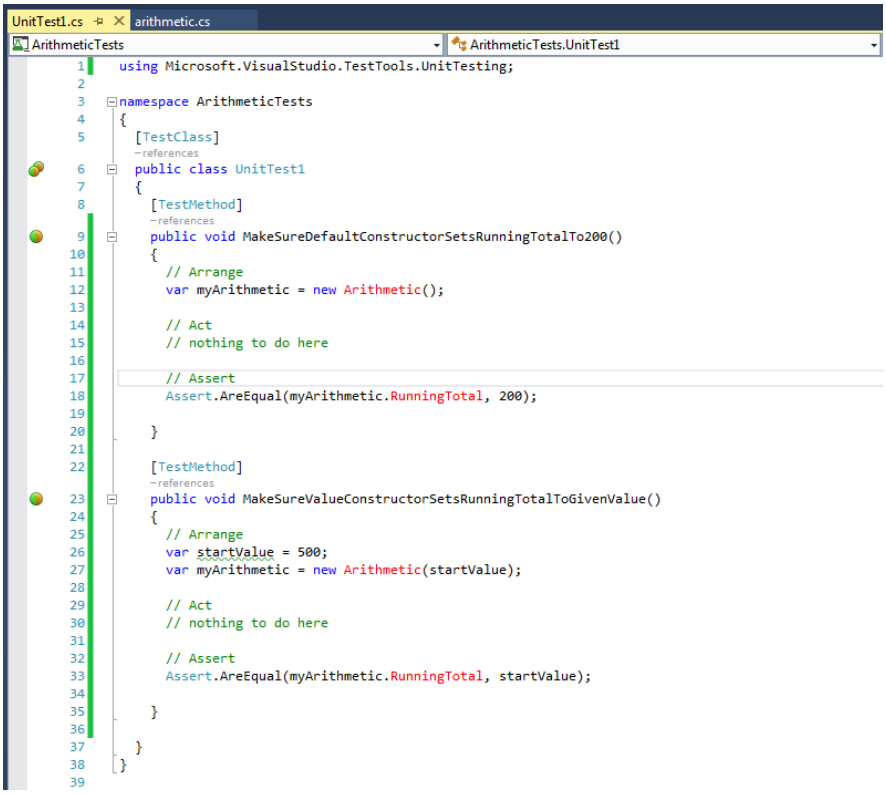


Figure 155: Our first unit tests in the VS editor

If I position my cursor on line 12 in my test fixture, then press the default R# key combo, we get the following pop-up menu:

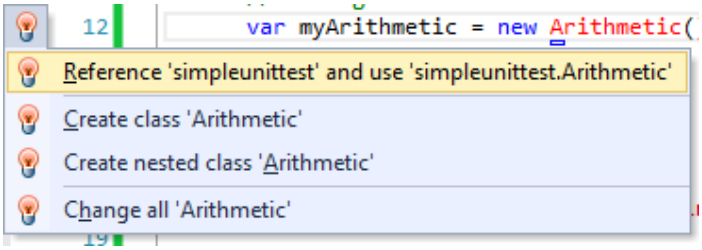


Figure 156: R# pop-up menu in our test fixture code

As shown previously, if you select the first option, R# will add a reference to the project, a `using` statement, and anything else needed to make sure the error is resolved.

Once we resolve these links, we can see our test code should now be ready to run.

UnitTest1.cs | arithmetic.cs

ArithmeticTests | ArithmeticTests.UnitTest1

```
1   using Microsoft.VisualStudio.TestTools.UnitTesting;
2   using simpleunittest;
3
4   namespace ArithmeticTests
5   {
6     [TestClass]
      references
7     public class UnitTest1
8     {
9       [TestMethod]
        references
10      public void MakeSureDefaultConstructorSetsRunningTotalTo200()
11      {
12        // Arrange
13        var myArithmetic = new Arithmetic();
14
15        // Act
16        // nothing to do here
17
18        // Assert
19        Assert.AreEqual(myArithmetic.RunningTotal, 200);
20
21      }
22
23      [TestMethod]
        references
24      public void MakeSureValueConstructorSetsRunningTotalToGivenValue()
25      {
26        // Arrange
27        var startValue = 500;
28        var myArithmetic = new Arithmetic(startValue);
29
30        // Act
31        // nothing to do here
32
33        // Assert
34        Assert.AreEqual(myArithmetic.RunningTotal, startValue);
35
36      }
37
38    }
39  }
40
```

Figure 157: Our test fixture is now ready

If we click on our double icon at the top of our tests, we can run them. Let's go ahead and do that:

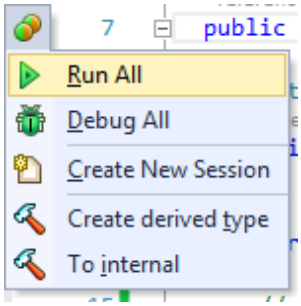


Figure 158: Clicking on the double icon gives us this menu

Select **Run All**, then after a moment or two (depending on the speed of your machine), you should get something like the following:

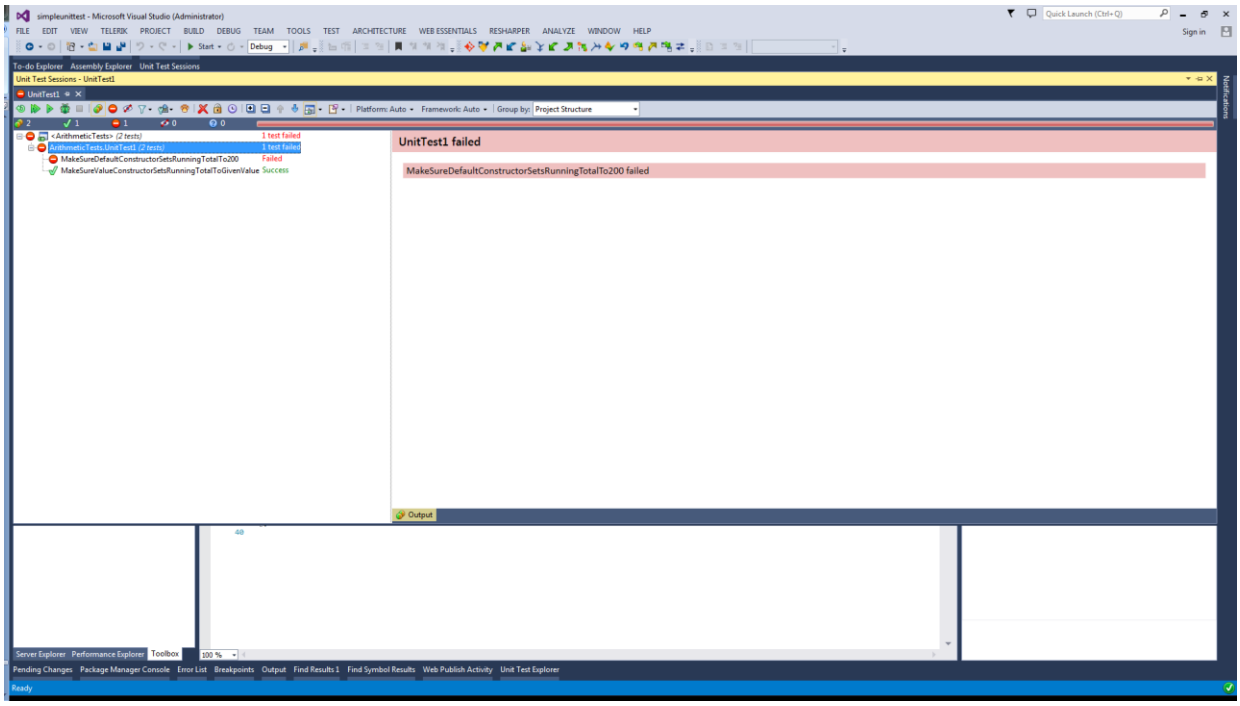


Figure 159: Running a new test session using the R# test runner

As you can see from Figure 159, one of our tests has failed. Let's go take a look at why. If you click on the name of the failed test, the test runner will tell you the exact reason things failed.

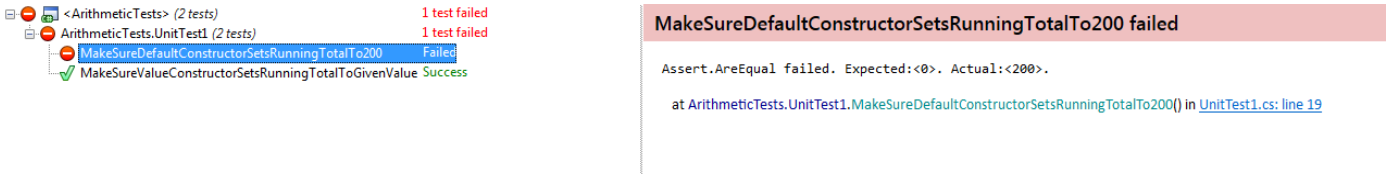


Figure 160: Result from our failed test

You can see from the report in Figure 160 that the result was `0` when it should have been `200`. If we take a look at the default constructor in our class, we'll quickly see why.

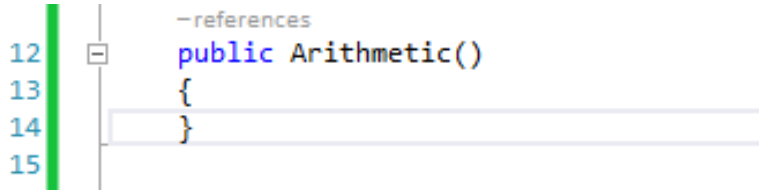


Figure 161: Our default constructor is empty

We don't do any initialization of our running total in the default constructor; if we correct that and add this into the constructor:

```
_runningTotal = 200;
```

Then re-run our tests:

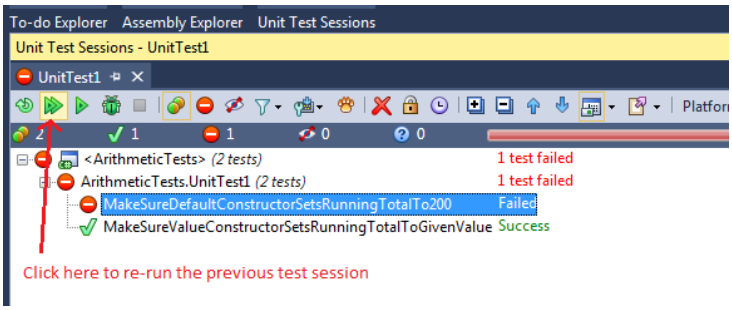


Figure 162: Re-run the previous test session

We should see that everything now passes.

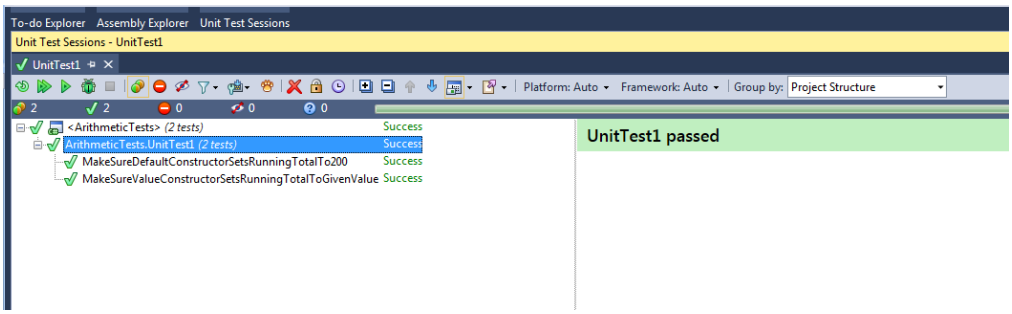


Figure 163: Our tests are now successful.

We can now go back to our test project and add a new test fixture. This time we'll test adding a fixed value of `200` to our running total. We'll create the arithmetic object using the default constructor, so the running total at start should be `200`.

Before we even get to running anything, we can see there's a problem.

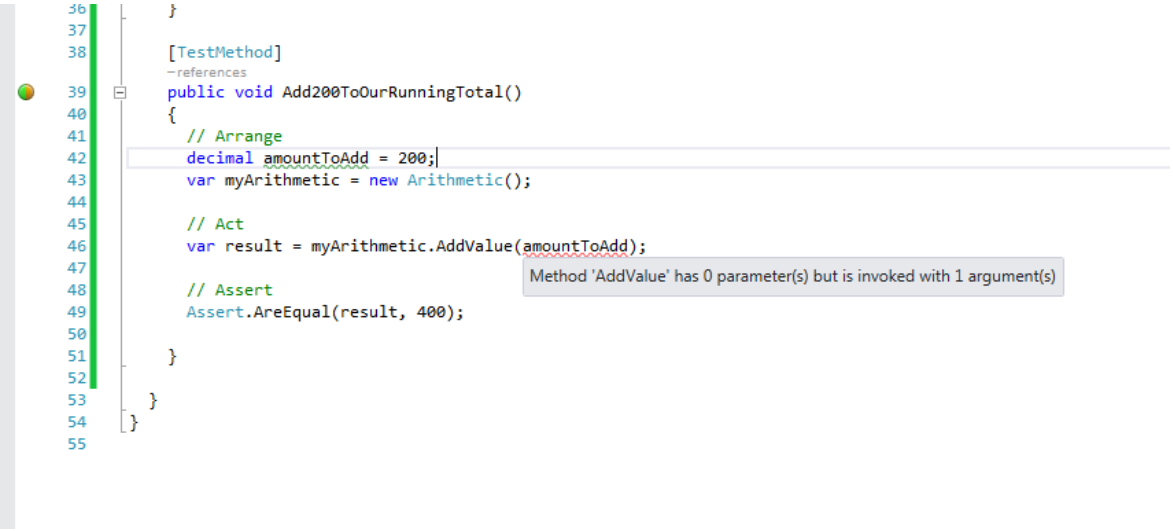


Figure 164: Before we even try to run anything, we can see a problem

Let's fix that first—we won't be able to run our tests if we can't compile things.

Change the `AddValue` method in your arithmetic class so that it looks like the following:

```
public decimal AddValue(decimal amountToAdd)
{
  return 0;
}
```

Our test fixture should now have no errors and be able to run. If you go back to your test sessions window, you should see that R# has automatically added your new test fixture to the runner window.

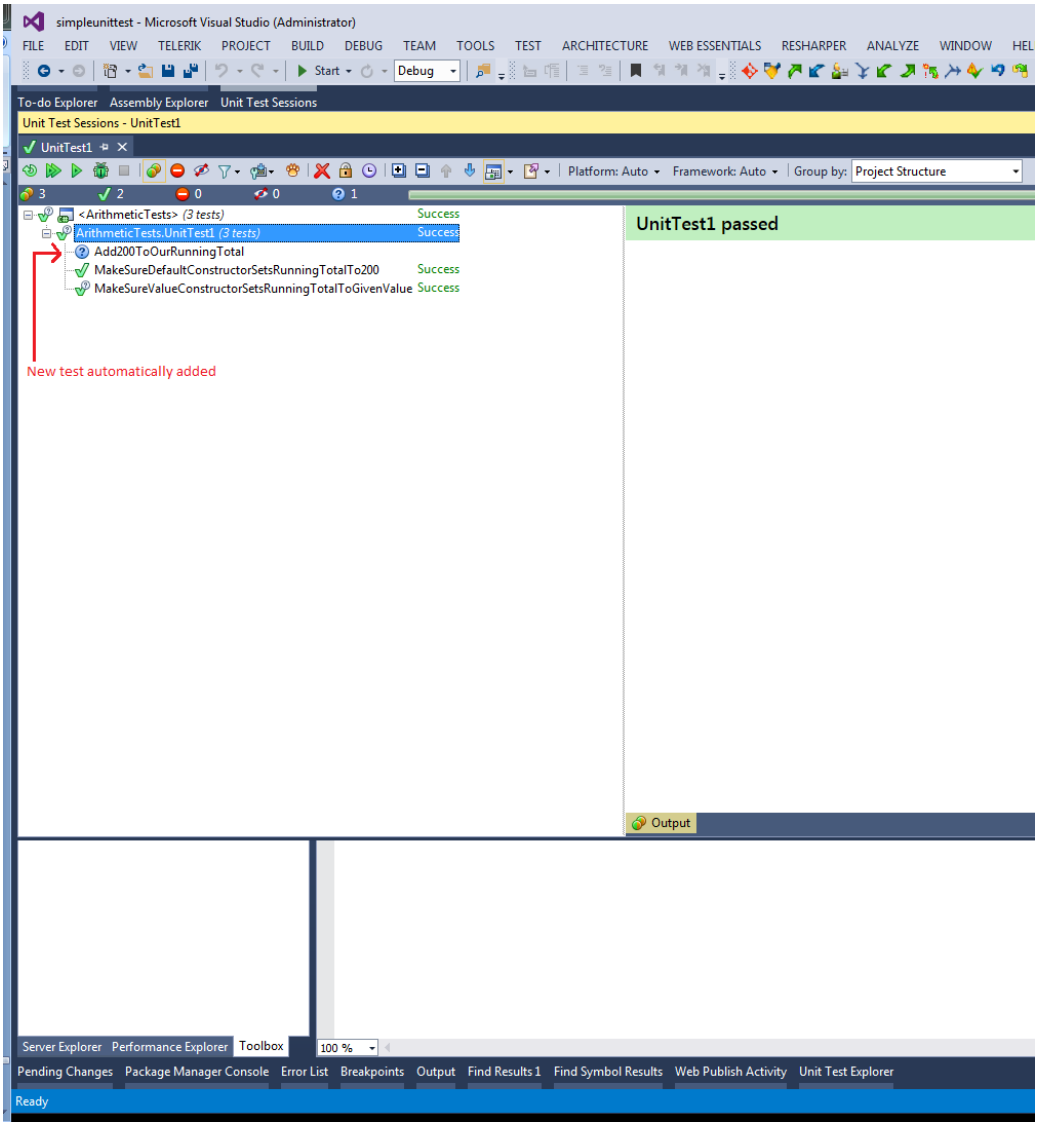
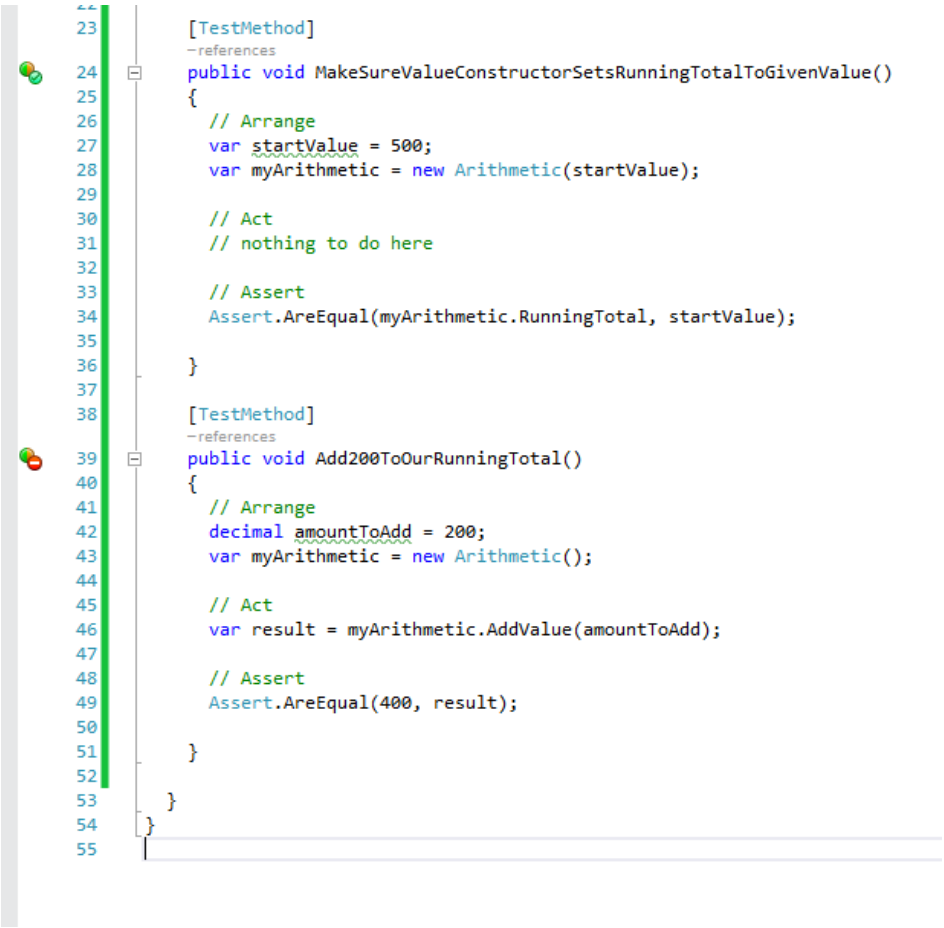


Figure 165: Unit test session with new test automatically added

As before, if you click the double arrow the tests should all be run, with the two previous tests still passing and the new one failing.

Again, you'll see the test failure report in the test runner session window. However, if you look back at your test fixture code, you'll see that the left margin icon has also changed:



```
23      [TestMethod]
        0 references
24      public void MakeSureValueConstructorSetsRunningTotalToGivenValue()
25      {
26        // Arrange
27        var startValue = 500;
28        var myArithmetic = new Arithmetic(startValue);
29
30        // Act
31        // nothing to do here
32
33        // Assert
34        Assert.AreEqual(myArithmetic.RunningTotal, startValue);
35
36      }
37
38      [TestMethod]
        0 references
39      public void Add200ToOurRunningTotal()
40      {
41        // Arrange
42        decimal amountToAdd = 200;
43        var myArithmetic = new Arithmetic();
44
45        // Act
46        var result = myArithmetic.AddValue(amountToAdd);
47
48        // Assert
49        Assert.AreEqual(400, result);
50
51      }
52
53    }
54  }
55
```

Figure 166: Test fixture code after running the tests

The successful tests now have a small green tick next to them, whereas the failing tests have a small red “no entry” sign next to them to indicate that they currently don't pass.

Just this small change to the IDE means you can leave your tests running in the background and go straight back to work, knowing that when the tests finish, the results will be right there in front of you without needing to break off what you’re doing and context-switch to another task.

This instant feedback means you can just continue working through your test coverage with no interruptions and down time. Couple that with the following default keyboard shortcuts:

- Ctrl+U, L—Run all tests in solution.
- Ctrl+U, Y—Run currently open test session.
- Ctrl+U, U—Repeat previous test run.

And just as with everywhere else in R#, you save time and stay more productive, with your hands on the keyboard.

While we've been doing this, the Unit Test Explorer has been quietly keeping track of our test coverage.

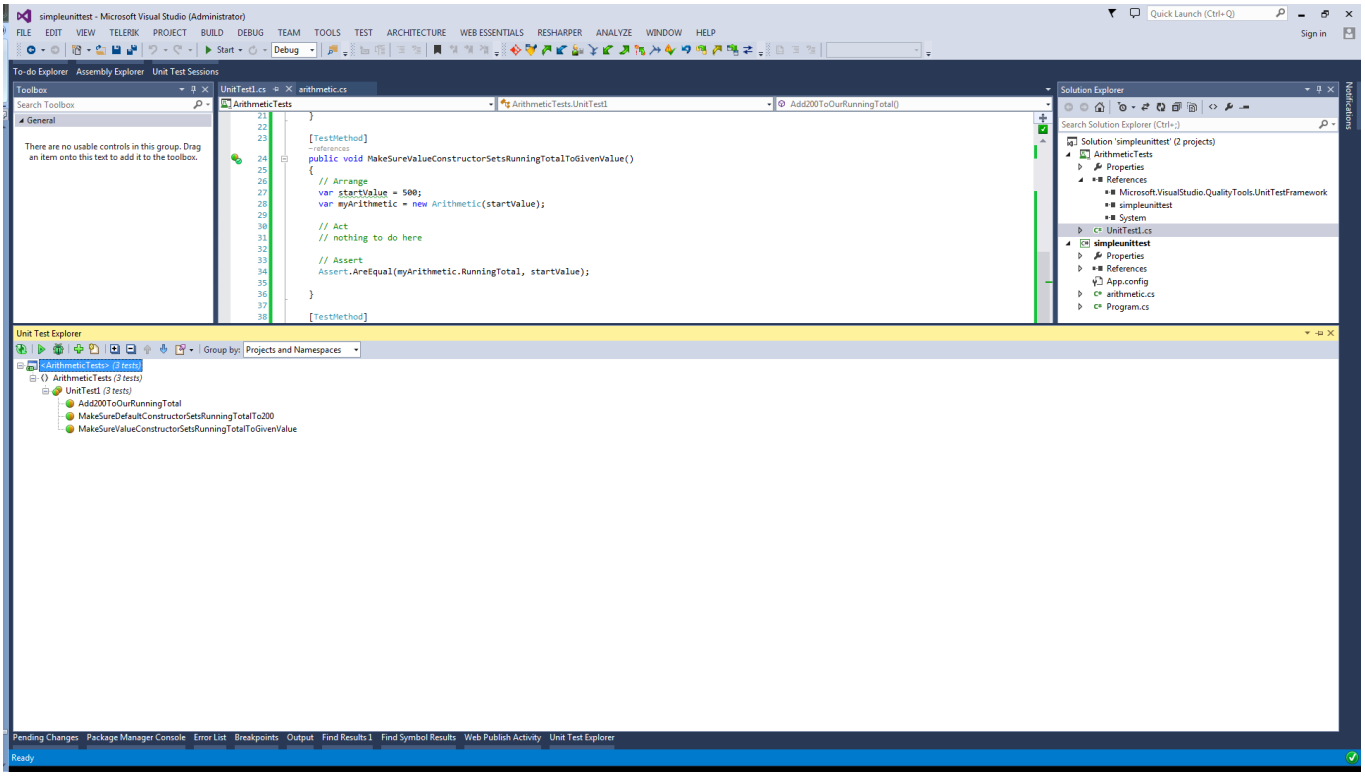


Figure 167: R# Unit Test Explorer

You can run your tests from here, which will open your current session and trigger everything. You can export a list of your tests, or use it to navigate to the tests in your solution by double-clicking on them. Think of the Unit Test Explorer as a file manager for your unit tests, allowing you to organize them and see at a glance what you have.

You can export your tests in text, HTML, and XML formats by clicking the **Export** icon on the Unit Text Explorer toolbar.

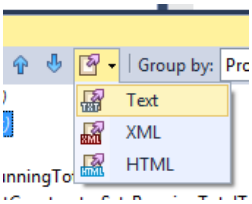


Figure 168: Unit Test Explorer, Export icon

When you choose to export, you get a data preview allowing you to either copy and paste the data, or choose a file to save it to.

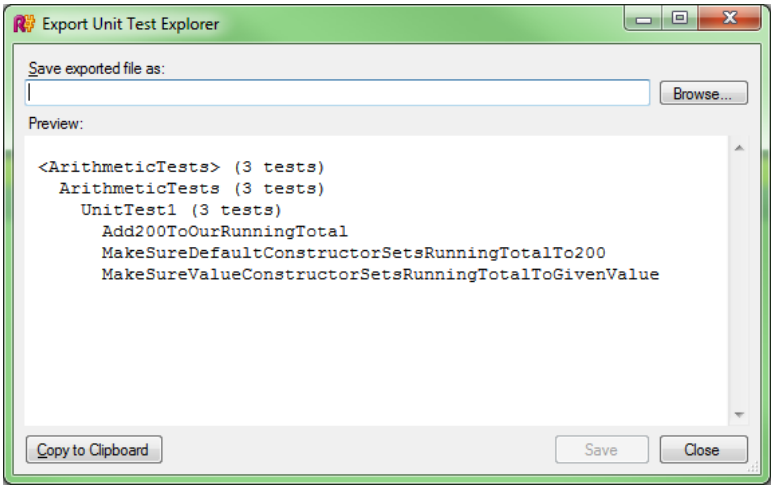


Figure 169: Text export of unit test list

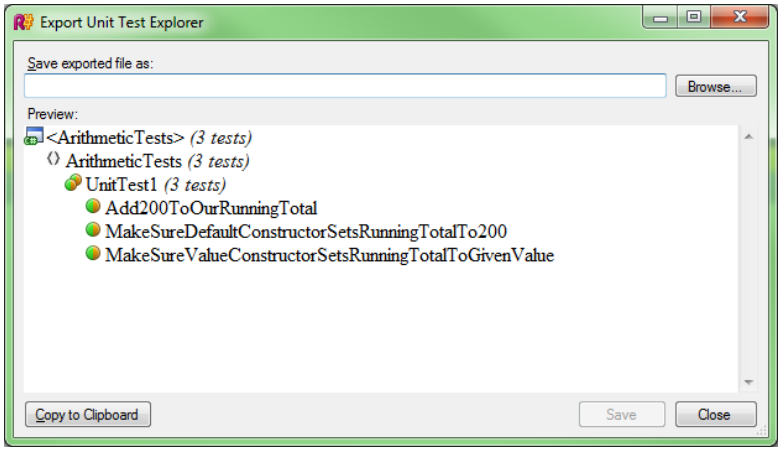


Figure 170: HTML export of unit test list

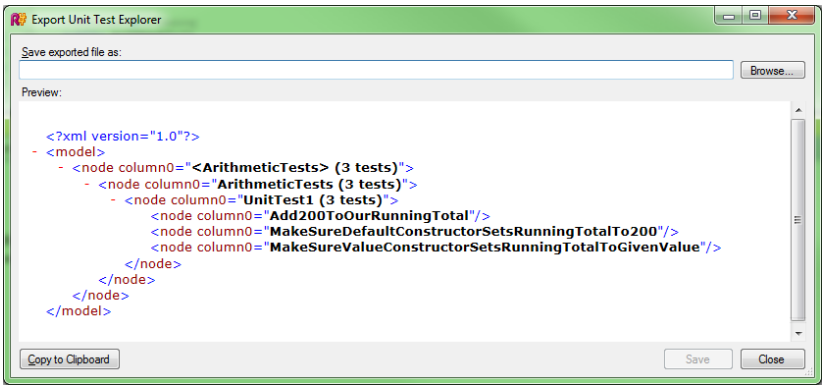


Figure 171: XML export of unit test list

You can also perform the same operation in the test runner session in order to export your current test coverage into another application.

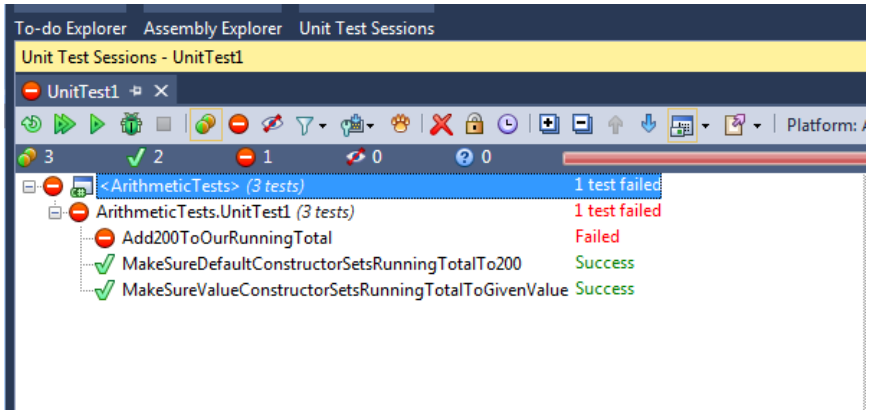


Figure 172: Current unit test run

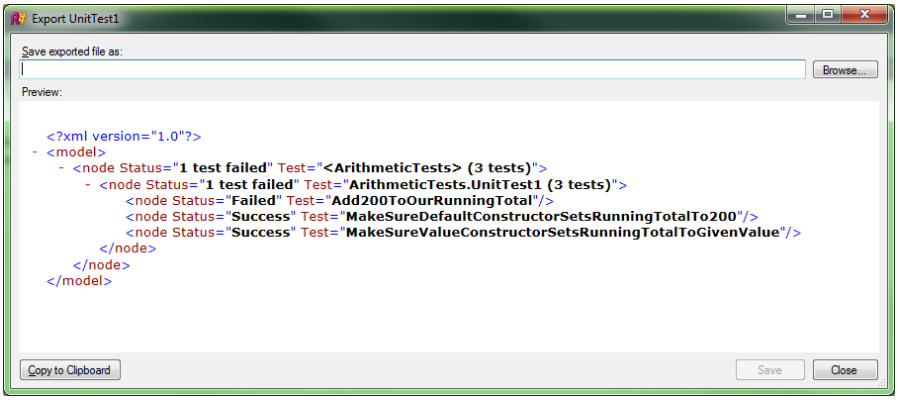


Figure 173: XML export produced by the run in Figure 172

Taking the current state of your project's unit tests, and being able to incorporate that data into your own coverage and management system, means you can easily graph test coverage and progress for project managers and business units, allowing instant dashboards and other tools to keep a near real-time update on a project.

Unit testing with R# as your companion is now easier and faster than it previously was. This is great because it allows you to get your focus back on the code and away from the admin side of things. And don't be fooled—I've only covered creating tests in C# here. R# gives the same level of detail and attention to tests written in JavaScript too.

All tests in a solution, regardless of language, appear in the same windows and have the same left-margin controls for instant running and feedback.

Chapter 9 Architecture Tools

Once you've bent, refactored, and tested your solution to destruction, the last thing you need to do is examine the 1,000-foot view to ensure that your overall architecture looks exactly the way you planned it to.

It comes as no surprise that R# also has that covered.

Left-click on the Solution node in your Solution Explorer, then head to **RESHARPER** > **View Project Dependencies**, and you should get something similar to the following:

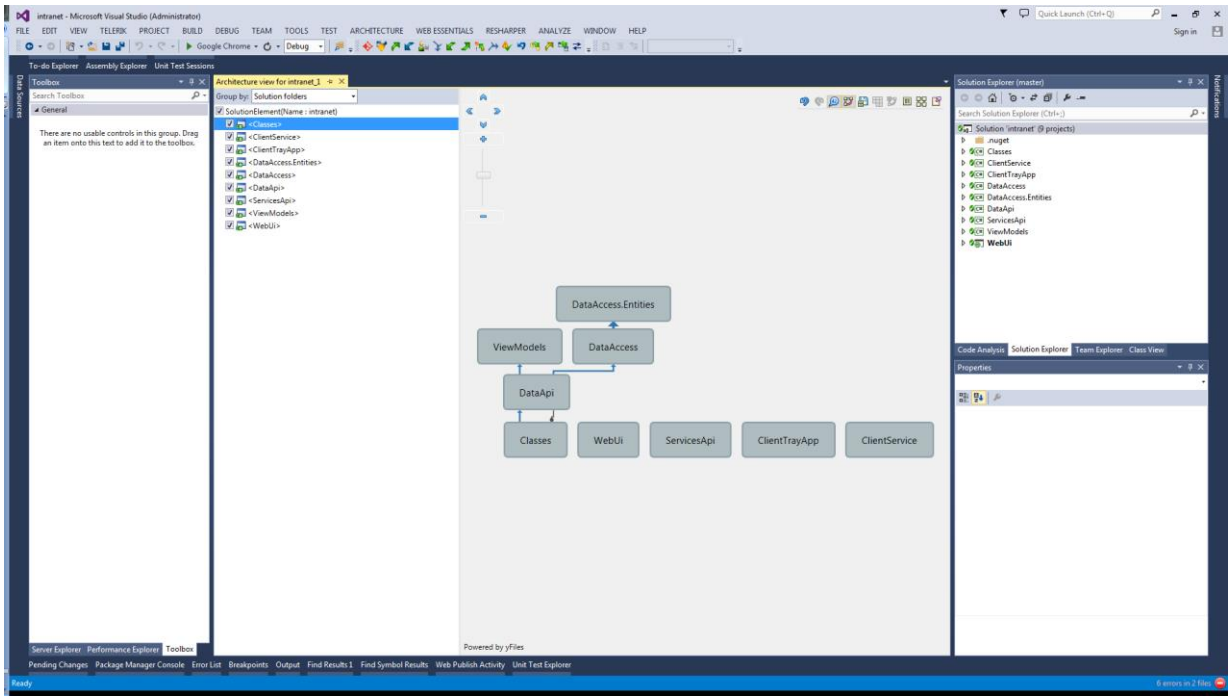


Figure 174: The R# architecture explorer

You can immediately see the big picture, and you instantly have tools to zoom in, move around and export, and manipulate the diagram in different ways.

The first two controls (blue arrows) are the **Undo** and **Redo** controls, allowing you to move back and forth through changes in your diagram.

The third button is the **Coupling Analysis** control. This control turns R#'s deep inspection of the references between modules in the solution on and off. It generally runs in the background, and on large solutions may take several minutes to complete. Allowing it to run (it is on by default) means that R# can build an incredibly detailed picture of the architecture, but at the expense of waiting for that diagram.

The fourth button is the **Show/Hide transitive references** icon. With this on, links between modules are collapsed into one single link, and with it off, independent multiple links between modules can be seen. With a large project, this can make a lot of difference in the readability of your diagram.

The fifth button, with a small disk image on it, allows you to save the currently displayed architecture file to disk—not just to safeguard a copy of the diagram, but also to allow future comparison.

If you go back and look at the architecture menu on the main VS toolbar, you'll see there's a second menu item allowing you to load a file for comparison.

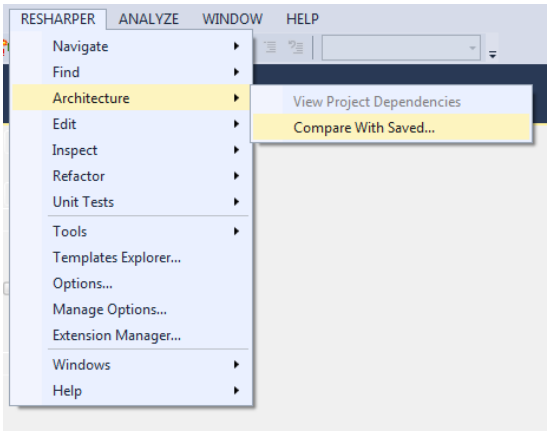


Figure 175: ReSharper Architecture menu showing Compare With Saved

The idea here is that you can take snapshots of your architecture as you develop it, then run comparisons on the project to see what the differences are over a period of time.

You'll see that the sixth button in Figure 174 is what looks like a small grid. When you have a saved file loaded alongside the current architecture diagram, clicking on this icon will ask R# to generate a human-readable description of the differences it finds between the two models.

Continuing on, the next button is the **Show Possible Paths** button. In my screenshot in Figure 175, this option is dimmed due to the fact that the solution I have loaded is simply not big enough, or complex enough, for it to work effectively.

The **Show Possible Paths** function is typically used when you have large, multi-solution projects open, or when you create a dependency graph from many different items at project level, rather than a singular one at solution level. The tool is used to analyze the current architecture and detect possible violations, such as circular references, or links between modules that where unintentionally placed via references to other modules.

The next two buttons are used to collapse and expand the nodes in the graph, and the final button is used to export a copy of the currently displayed graph, out to a PNG file for use elsewhere.

In Figure 174 you can also see that to the right of the diagram is a list of the projects included; unchecking the modules you don't wish to see will remove them from the diagram. In my case, If I uncheck **ClientService** and **ClientTrayApp** (since they are not direct dependencies of the overall project), my diagram reorganizes itself to remove those two modules.

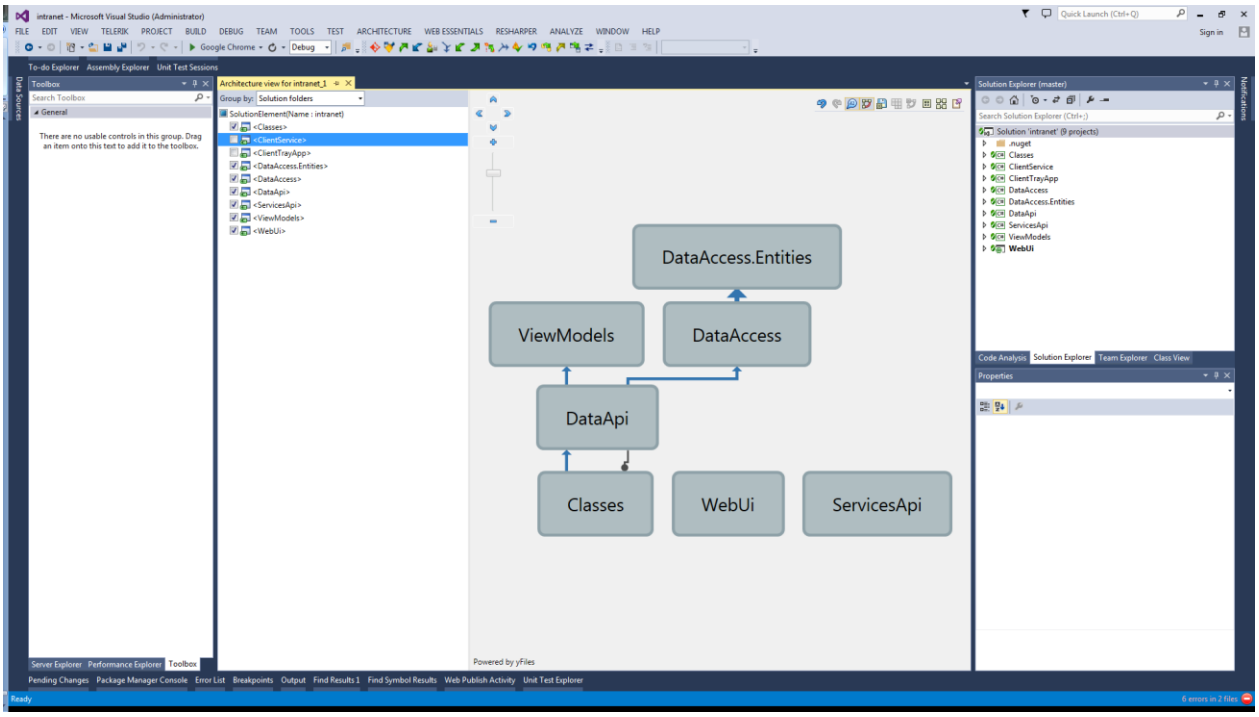


Figure 176: R# Architecture view, reorganized to show the un-ticked references removed

If you hover over a graph node, the rest of the graph is no longer highlighted to give emphasis to the selected node:

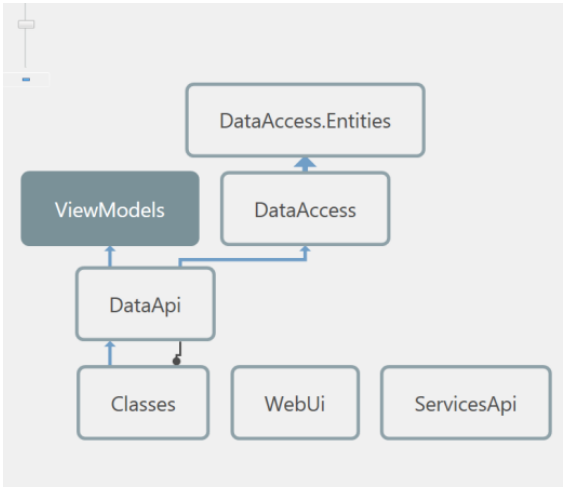


Figure 177: Architecture view with de-emphasized nodes

If you then right-click on a node, you get a pop-up menu giving you instant access to many of the tools we've already explored, allowing you to quickly assess the state of a given node in the project.

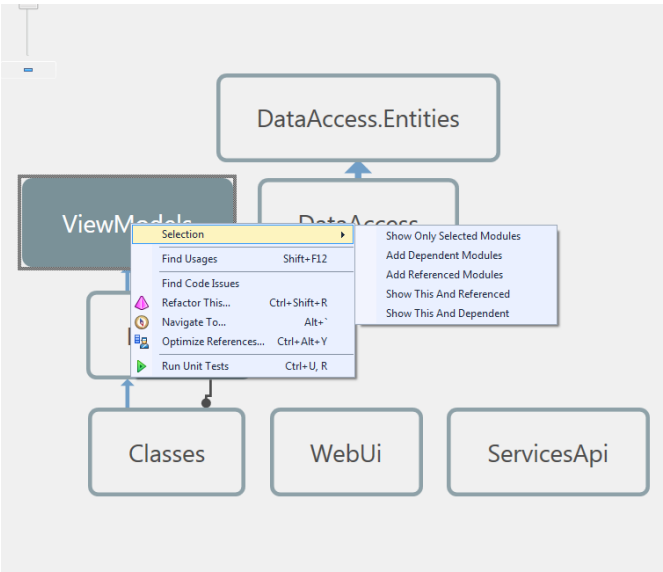


Figure 178: Architecture diagram showing quick access to tools via the pop-up menu

The architecture diagram is your project's road map, allowing the senior and lead developers to see at a glance if things are progressing the way they should be, and allowing quick trouble-shooting and remedial action when needed.

Chapter 10 Extending ReSharper

At this point, you might be wondering just how we could possibly extend something that already has so much functionality in it, but it is possible, and there are great reasons for it.

R# can be extended in one of two primary ways. The first is via its own internal extensions gallery. In much the same way that NuGet has revolutionized Visual Studio development, the R# extensions gallery has also opened the door and transformed R# from a humble plug-in to a first-class extension tools platform.

If you go to **RESHARPER** > **Extension Manager** on the main menu, you should see something like this:

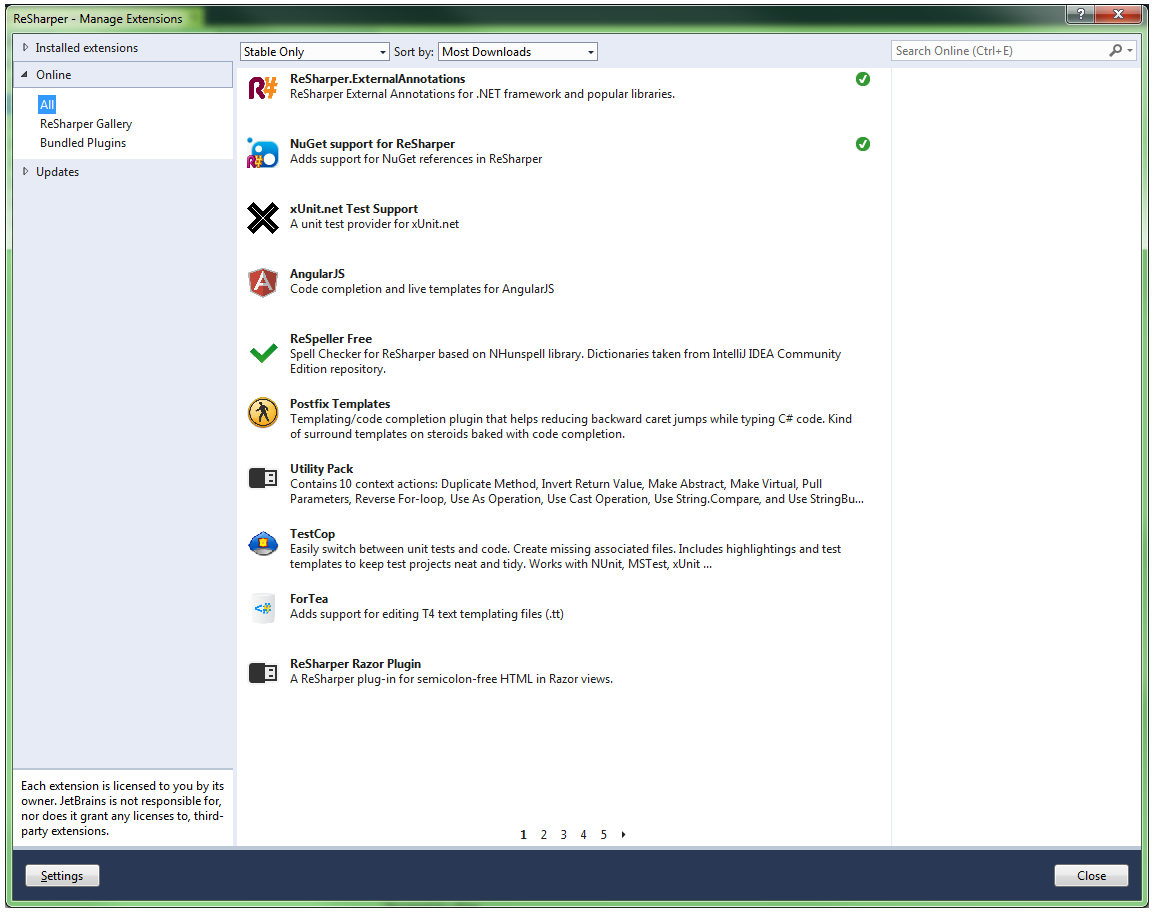


Figure 179: The R# extensions gallery

It's purposely designed with the same look and feel as the NuGet GUI so that developers will instantly know what they are doing, and how to find an extension that will help.

As good as the core R# product is, there will always be extended functionality that end-users will want. The Extension gallery provides a mechanism for third parties to provide that extensibility.

By default, as you can see in Figure 179, we have NuGet support and R# External Annotations loaded. The External Annotations are added extras allowing R# to work with some of the more popular third-party libraries that get used in most .NET projects, as well as some of the extended .NET functionality that often doesn't get the extent of coverage that other parts of the framework do.

The second means of extending R# is through the use of the free SDK that JetBrains provides as a download from their website.

Just below the main download, you'll see Related Downloads; one of these will be the SDK, allowing you to write your own third party add-ons to be used directly within your installation.

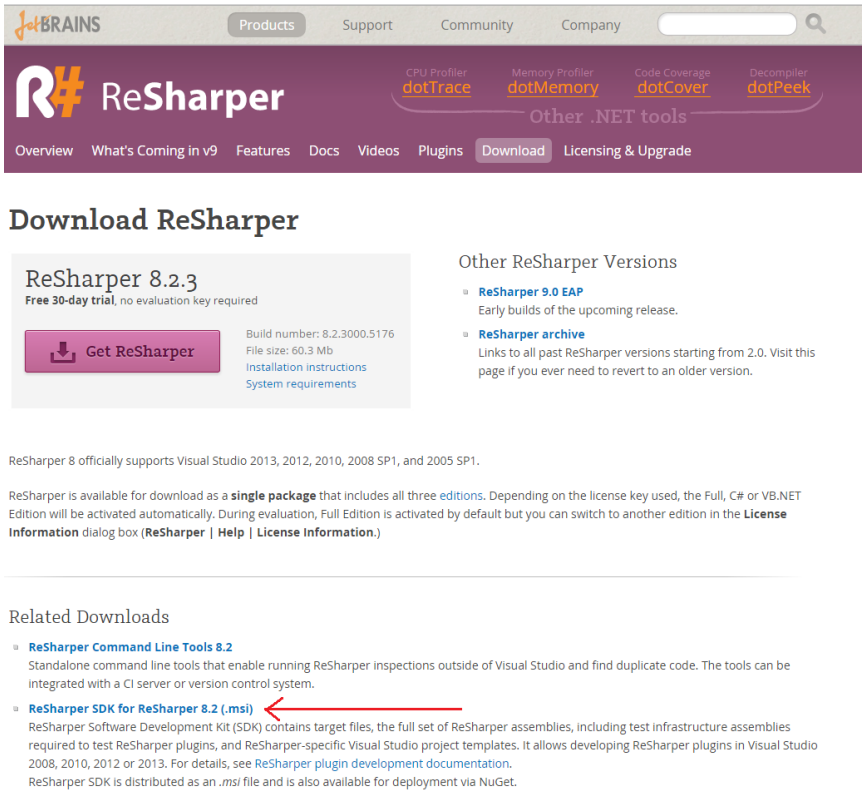


Figure 180: R# downloads page showing the link for the SDK

An in-depth description of how to use the SDK to create your own add-ons is far beyond the scope of this book, but there are some good samples included, and the documentation will easily get you started with the tools. Beyond that, there's a massive community of dedicated R# users out on the internet, many of whom are more than willing to help with questions and advice.

Chapter 11 ReSharper Version 9

Anyone who develops software knows that things don't stand still for anyone, and ReSharper is no exception.

While writing this book, I regularly consulted and acted on feedback from the company who made the product, JetBrains, to make sure that anything I include in this book is technically accurate and reflects ReSharper in the best possible way.

During this time, I've been made aware that Version 9 is now on the horizon, and by the time this book actually reaches you, there is a very good chance that V9 will be the stable released version. I've therefore decided to add an extra chapter to the book covering some of the more important changes to come about in ReSharper 9.

In general, a lot of what I've already documented in the book will continue to work as is, or in a very similar manner to how it does in the current version. The changes documented in this chapter are based on my access to the JetBrains Early Access Program, and while I was originally going to rewrite the appropriate parts in the book to cover the new version, I felt that it was a better strategy to try and keep good coverage of both versions, and to help readers with the transition from V8 to V9.

The new installer

One of the biggest surprises is the new installation process. In the new version, the installer does not just install R#, it now serves as a central platform to install many products by JetBrains. It also allows you to see at a glance on the first screen which development products you have, and where they will be installed.

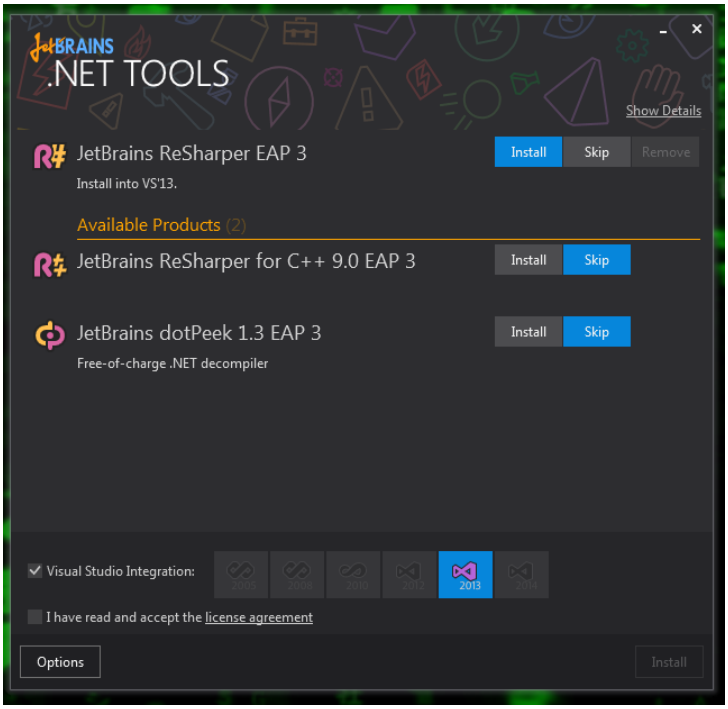


Figure 181: The new R# V9 installer

Once you click the install buttons and icons to make your appropriate choices, you're ready to click the main Install button. Note that R# V9 now includes first class support for C++ as a standalone product, rather than as an integration to the overall system.

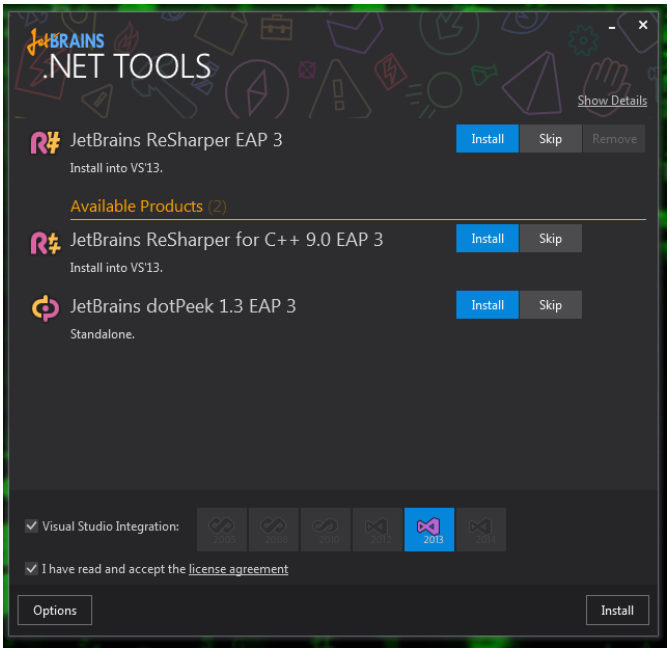


Figure 182: R# V9 installer ready to go

If you have older versions of the products you're installing, everything will be automatically uninstalled for you.

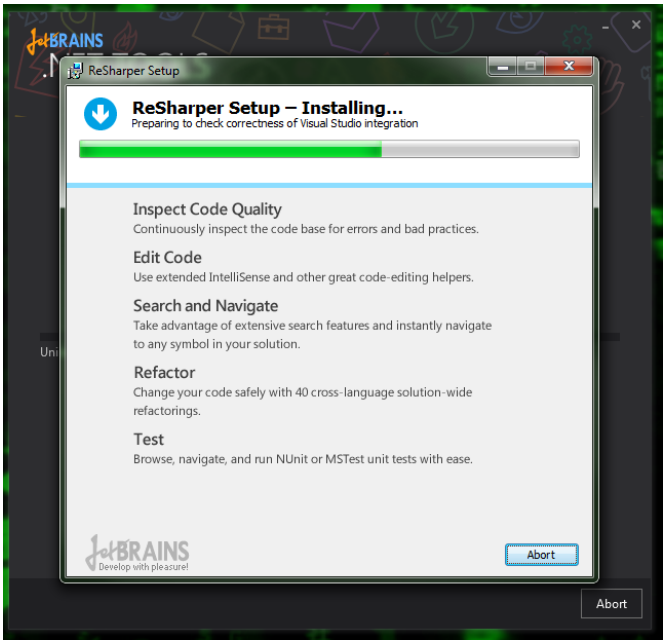


Figure 183: New installer uninstalling old products

The new installer will then work its way through your installation choices.

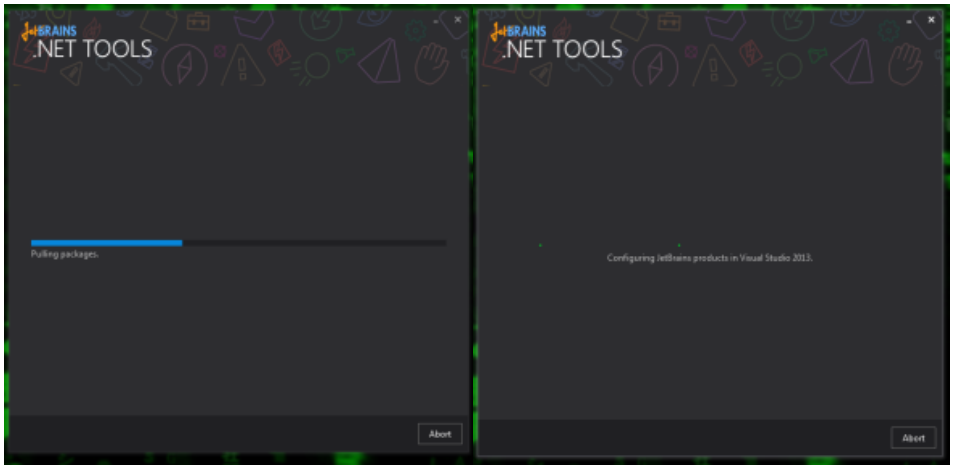


Figure 184: The new installer

One good thing about the new installer is that it will examine your Visual Studio environment while installing, and inform you at the end of any general errors it finds.

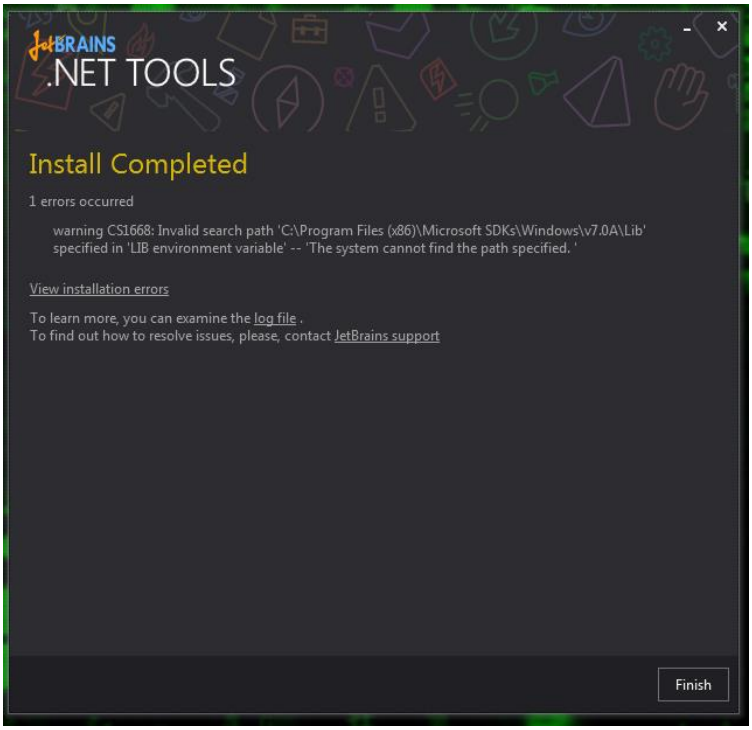


Figure 185: Installer finished, and it found some problems you might be interested in

After you finish installing and re-run Visual Studio for the first time, you may find that you need to re-license your products. You will also find that in addition to needing a new license to move from V8 to V9, you will also have to license individual JetBrains products separately.

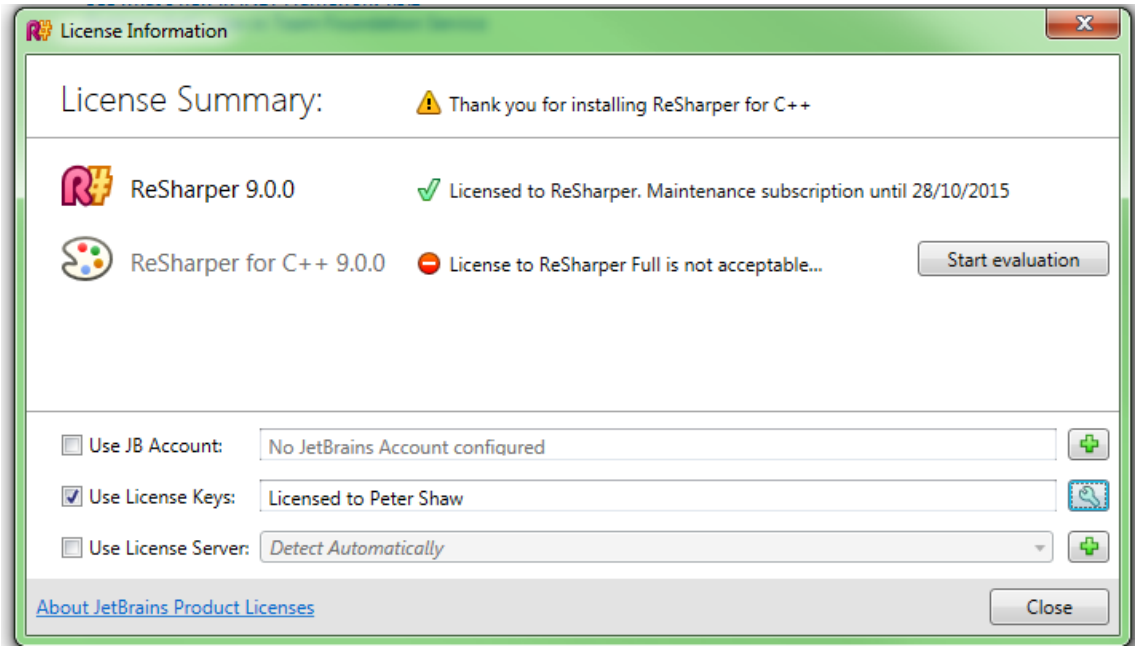


Figure 186: New license options

In my case, I have a standalone license that only covers R#, and not R# for C++.

If R# has any problems running, then any issues, exceptions, or other problems are recorded in the Exceptions Manager.

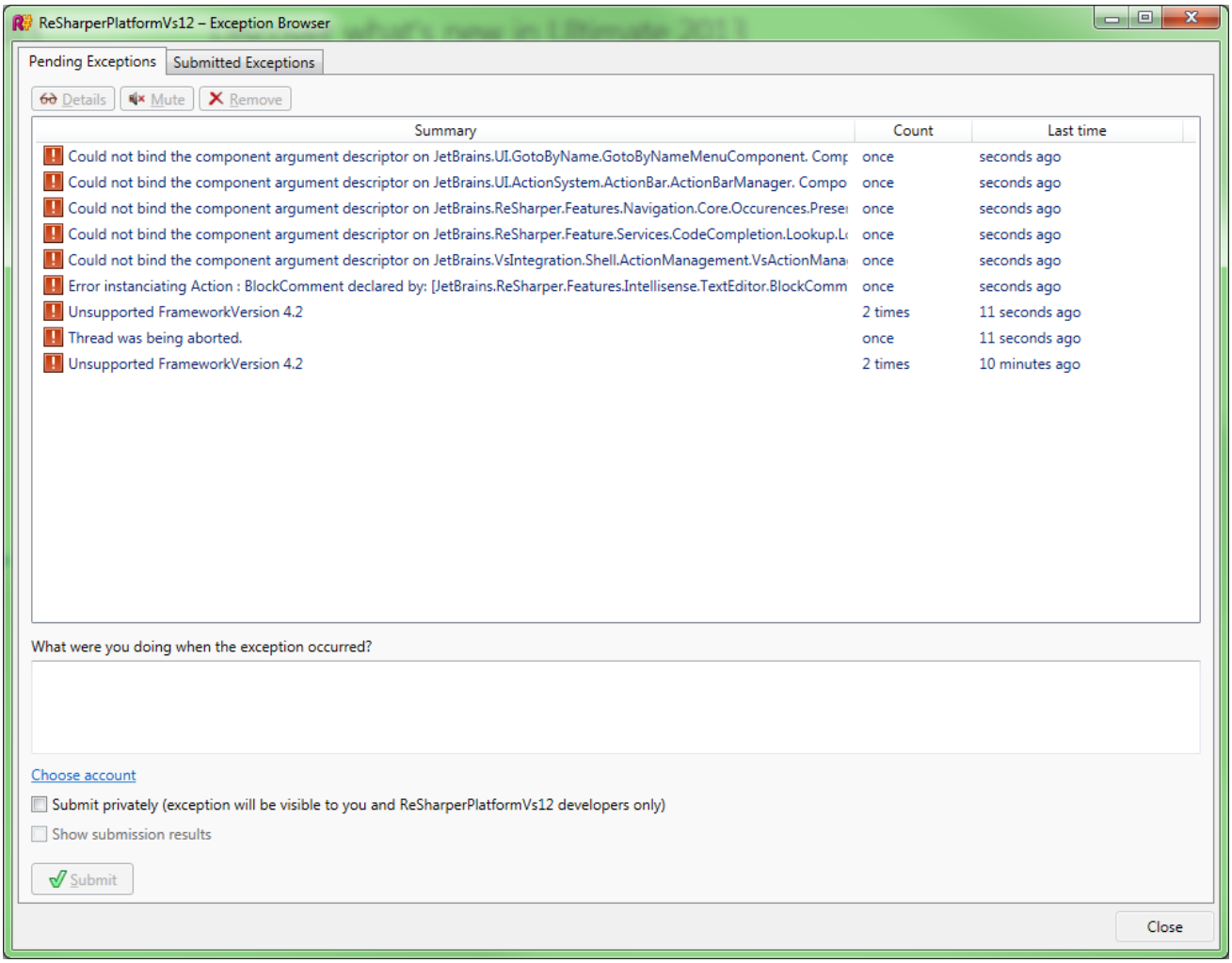


Figure 187: R# v9 Exceptions Manager

If you have the dialog in Figure 187 hidden, you'll also see a notification in the Visual Studio status bar that something is wrong:

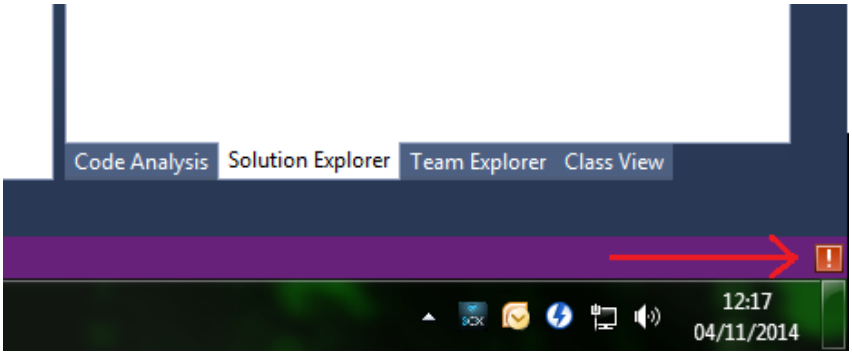


Figure 188: Status warning showing that R# had some issues

You can select entries in the dialog box in Figure 187, then submit them directly (with or without account information) to the JetBrains development team, who will then respond to them to make the product better.

If you're running the EAP versions, submitting errors like this is vitally important, because it helps improve the product prior to general release.

Once your exceptions have been submitted for review, you should get a dialog box like the following to confirm the submission status:

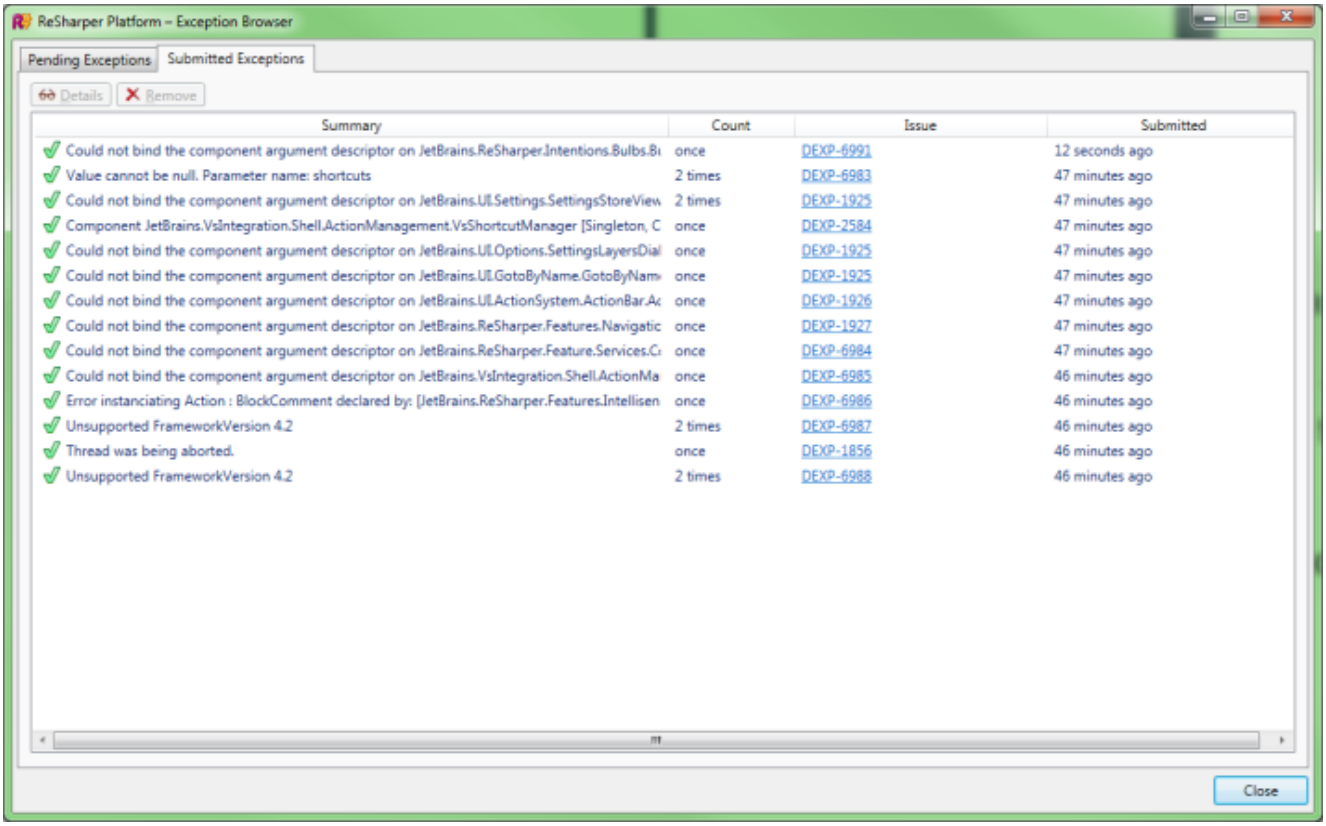


Figure 189: R# Exceptions, submission status

Changes to the R# Options

There's been a whole raft of new additions to the R# options dialog box, including the ability to now search in the options, rather than having to know where something is.

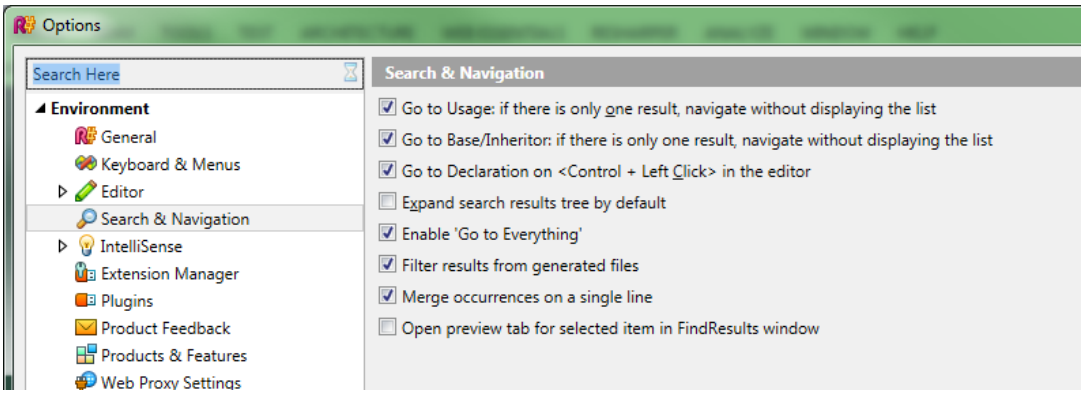


Figure 190: The R# v9 Options dialog box

As you can see in Figure 190, there's now a real-time search box above the options tree on the left-hand side. For demo purposes I've typed in “Search Here,” and you can see that R# has presented me with the most likely options it believes I'm looking for.

There are also new changes to the global options that allow you to enable and disable single specific feature groups.

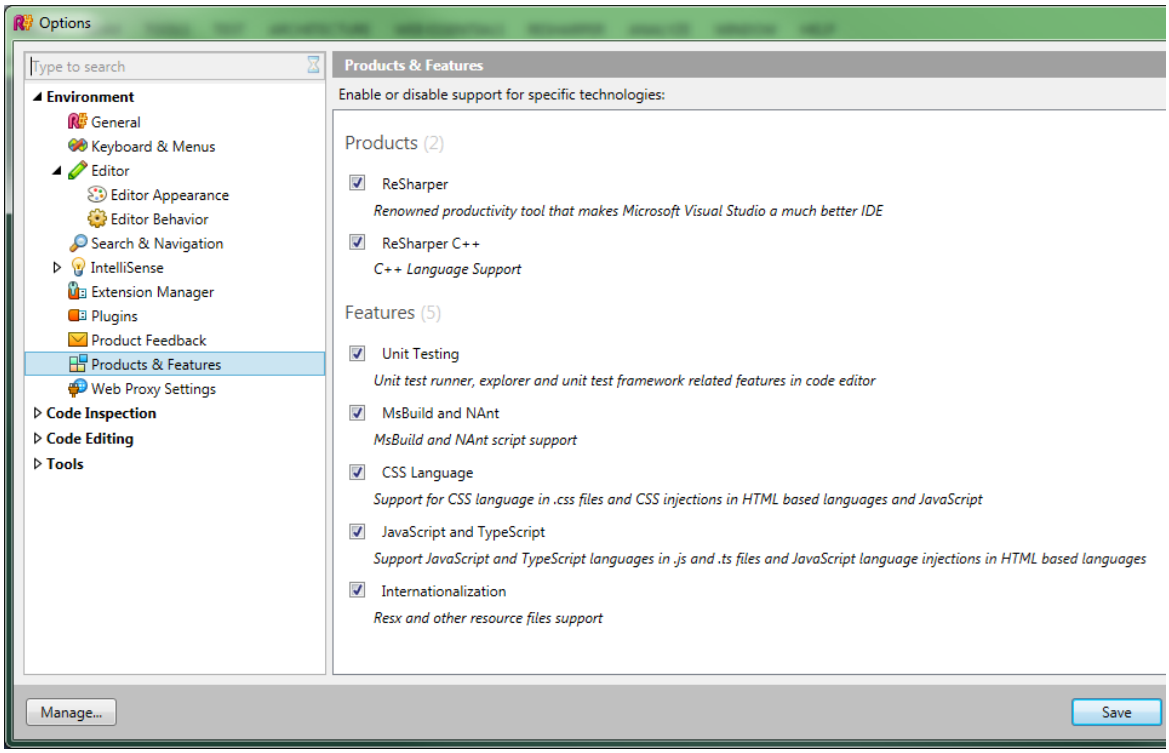


Figure 191: Options for enabling, disabling specific feature groups

In previous versions of R#, if you had an issue with Razor templates for example, you would have to disable R# entirely while you worked on your Razor code, then re-enable it again when finished. With the new modular options, you can disable specific modules, allowing you to better fine-tune your R# experience.

As well as the options mentioned here, there are also a staggering amount of new code formatting options, code style checks, and other new inspections.

New Keyboard Shortcuts

The master Alt+Enter key can now also be used to search for and go directly to an action that R# has available. If you activate the master key on a line that has no current inspection infringements, you'll get the action search pop-up window:

Figure 192: R# v9 Action Search pop-up window

This pop-up window can be used to provide hints and searches on R#'s top-level menus and tools, and R# will actually suggest tools you might want to run on the current line that could help you perform a task that you'd never even thought of.

It's a bit like running tools on an already great bit of code, and finding out that R# has already looked at the way you work, and figured out that you might want to perform similar operations to those already performed elsewhere.

Where possible, you'll also get quick access directly to top-level menus, directly from the Alt+Enter keyboard combination, if choosing that menu would be an applicable option on the current line where your cursor is located.

Other improvements

One new feature that the team is currently working on is called Navigate to Exposing APIs. This new navigation tool will search through any API (local or remote) that your project has access to. This search can be performed for a specific type of object, allowing you the possibility of taking an object under interface control and finding which methods can be used to consume and produce those specific object types.

If you have a need to return a specific type of object, then this new navigation feature can save you hours of analysis in third-party libraries trying to find that all-elusive method that you need to complete a month's worth of work.

The architecture tools have also been massively expanded. The new dependency diagram format will show you at a glance what levels of dependency inner objects have on outer objects:

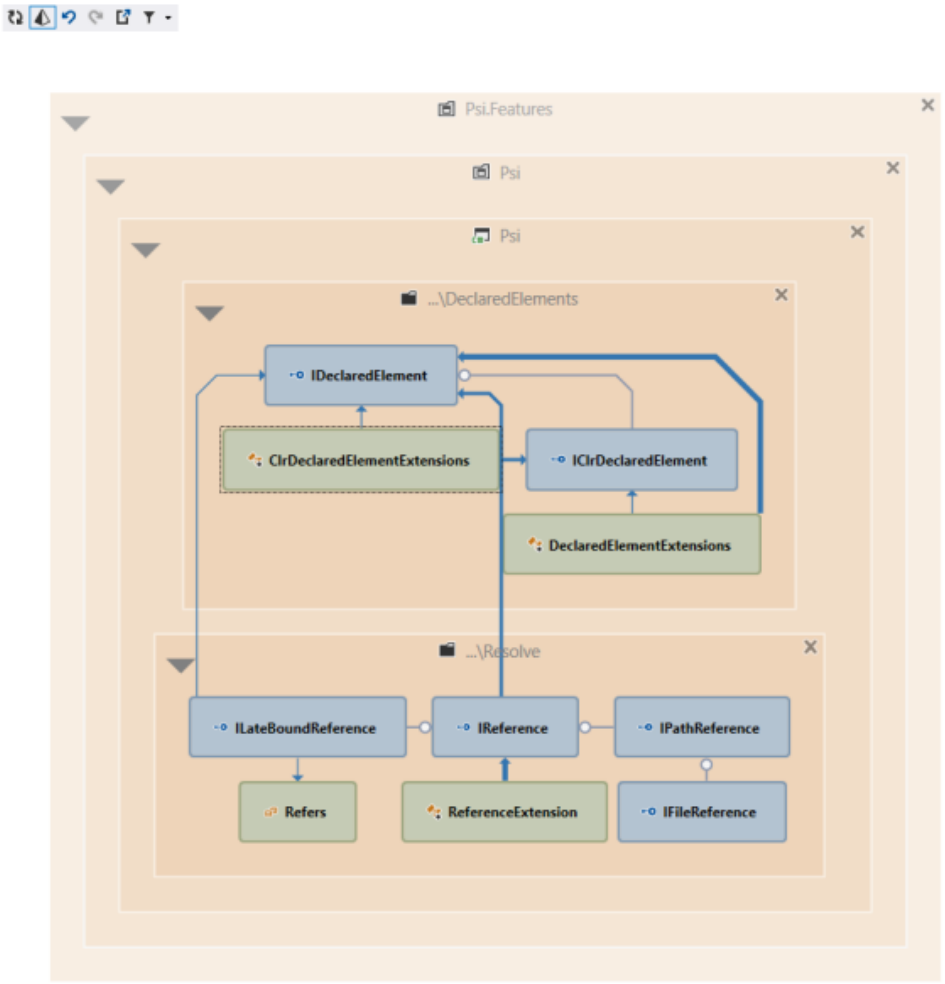


Figure 193: R# v9's new dependency layer architecture diagram

The Navigate menu now has some serious power, allowing you to get from one place in your code to any other place with minimum of effort.

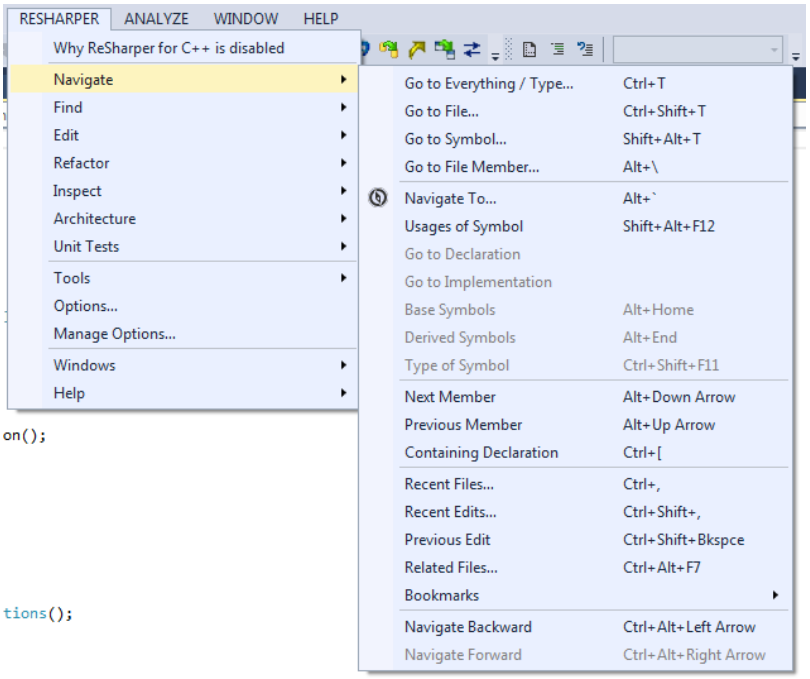


Figure 194: R# v9 Navigate menu

The Find and Replace options now support regular expression-based searches over your entire solution:

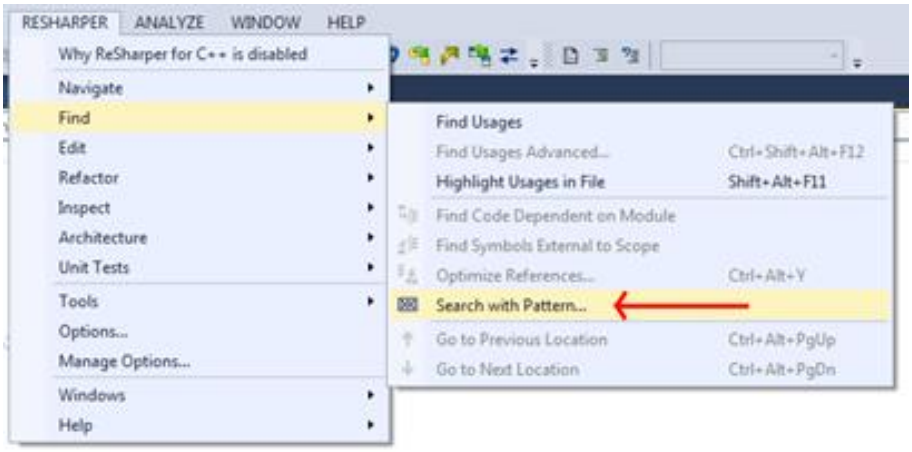


Figure 195: R# v9 now supports Regex pattern-based searches

Some of the features are still in development, and so can't be demonstrated fully in this one chapter, but it's clear that JetBrains is continuing to innovate and improve on the ReSharper product line.

Is This the End?

Not by a long shot. In the course of these pages, I've only demonstrated the common functionality that ReSharper exposes to the day-to-day Visual Studio user.

I'd be lying through my teeth if I said there was nothing more to discover. I've been using R# now for approximately six years, and during the course of this book I could easily say that I've increased my knowledge of the product easily by 50 percent.

There are options and items of functionality that you might never discover, simply because the project you're working on might never need them.

There's an army of passionate supporters out there, who provide feedback every day to JetBrains, or who develop new ideas, new plug-ins and new templates.

There are product evangelists going out into communities speaking to the end-users, finding out what they do and don't like about ReSharper.

ReSharper represents the pinnacle of .NET extension tooling, and is (and always will be) a very hard act to follow—even Microsoft has added functionality to Visual Studio that was first brought into play by the extension.

ReSharper continues to move at an ever-increasing pace of innovation, with its developers actually considering what developers need from the product. It's written by developers, for developers to use.

I hope what you've read here has given you much more appreciation and insight into how R# can help you become a better, more efficient developer, and that you continue working with it.