

Refactoring With Delphi 2005

By Craig Murphy

We all strive for good, clean code, highly polished architectures/designs. However, managers and customers want us to ship products as quickly as possible. Sadly the two often cause conflict and whilst we ship working software, as professional developers we are rarely happy our code, citing poor design, niggles, quick fixes, kludges, rushed coding, etc. as reasons to throw away large parts of the code in favour of a re-write.

Fortunately, refactoring allows us to improve the design of our code, without affecting the way it works. Delphi 2005 offers us IDE-based support for a handful of common, but effective, refactorings. Over the course of this article I will explore Delphi 2005's refactorings for both .NET and Win32 development.

The Origins of Refactoring

The concept of refactoring is not new; its origins can be traced back as far as 1992 at least. Refactoring is something that we have all been practicing for many years, without actually knowing it. Indeed, like gravity (which existed prior to it being named), each and every one of us have been spending time improving the design of existing code, re-testing it, incrementing the version number and shipping new versions to customers. Between 1992 and 1995 the word refactoring entered common usage; in 1999, Martin Fowler pushed refactoring into the mainstream with the publication of his Refactoring book.

Martin Fowler's seminal works [1] and his web-site [2] provide us with a collection of small changes that we can apply to our code, small changes that drive towards improving the design and making the code easier to read, more maintainable thus extending its life. Fowler is correct in stating that his book and his web-site are not definitive collections of refactorings as it is very likely that you will discover "new" ways of improving the design of your existing code.

Why Refactor?

How many of you find yourself looking at a piece of code only to wonder "how does that work?" Under such circumstances, do you find yourself re-writing large parts of the code such that it becomes "your code" and that you have a better understanding of its make up?

Similarly, you may have heard of the phrase "design debt"? Design debt revolves around the notion of a cost/time saving in the present, perhaps in order to meet a release date. The design debt manifests itself through hurried development, or a "get it working" attitude. You may find that you have cut corners to meet the release date, you may have duplicated code, methods that are too long, classes that do too much, reliance on null object checks, "data" classes with zero or very few methods or classes with overly exposed fields (too many public members). Whilst you may have employed some or all of these methods in order to get your product "out the door", clearing the design debt downstream is a much costlier exercise. Ultimately, your code becomes harder to maintain/improve and rapidly loses its value.

During the late 90s, the eXtreme Programming (XP) community coined the phrase "code smells". Code smells manifest themselves in code as items of design debt: any where you have cut corners in order to get some piece of functionality working is often a candidate for a code smell. Code smells are usually instantly recognisable: a handful of smells were mentioned in the last paragraph.

Refactoring provides us with a means of learning more about how and why code works, it lets us identify situations where design improvements are necessary and provides us with the steps required to improve the design without affecting the way in which the code works.

In a nutshell, by practicing refactoring, your code becomes more maintainable and easier to understand.

The Refactoring Process

William Wake [3] makes light work of describing the refactoring process:

1. Start with a working program
2. While smells remain, choose the worst smell
3. Select a refactoring that will address the smell

4. Apply the refactoring

Wake accepts that this will not give you a perfect design, even after many refactorings. But it will leave you in a better place where further design decisions can be made with more confidence.

Knowing when to stop is the trick! Kent Beck [4] provides us with the notion of “simple design”. Simple design is a state that meets the following criteria:

1. Run all the tests
2. Has no duplicated logic. Be wary of hidden duplicates like parallel class hierarchies
3. States every intention that is important to the programmers
4. Has the fewest possible classes and methods

Manual Refactoring vs. Automated Refactoring

Manual refactoring involves us taking a piece of code that is known to work, but is perhaps long-winded, convoluted or difficult to understand, then using code editing techniques including cut’n’paste and search’n’replace to refactor the code into something more elegant than before.

It is fraught with danger as it relies on a human being applying the refactoring. Whilst we might be able to accurately repeat a very small refactoring, it is unlikely that a refactoring that involves making a number of steps across more than a few units will be successfully repeated each and every time. Indeed, any refactorings that you perform should be small and be “un-doable” or reversible.

The dangers of cut’n’paste and search’n’replace should be fairly self-evident, especially when you consider that refactoring should provide repeatable, consistent results. I cannot be the only person who has suffered from a cut’n’paste that went wrong resulting in a bug or problem going undetected simply because the end result compiled. Similarly, search’n’replace often appears to work, but how often have you performed a search’n’replace on a variable name only to notice a run-time TLabel.Caption (or similar) has changed too?

Luckily, with the release of Delphi 2005, we now have IDE-based access to a number of popular refactorings, backed with the knowledge that there is a reliable undo engine supporting us! Delphi 2005’s refactorings are also type-sensitive, whereby the refactoring engine “knows” about the context in which a refactoring is being applied: for example, this allows us to perform intelligent search’n’replace.

Delphi 2005: Refactoring Support

Delphi 2005 offers us the following refactorings: Rename, Extract Method, Declare Variable, Declare Field, Extract Resource String and Find Unit. The refactorings are context-sensitive and will only be available if the correct conditions are met. Access to the refactorings can be achieved in any of three ways: via keyboard shortcuts, via right-clicking and choosing the Refactoring or via the Refactor option on Delphi 2005’s main menu.

I will go through each of these refactorings in turn. The first refactoring is Rename: I will spend a little more time covering it such that you have a good understanding of how the refactoring works and how Delphi 2005 handles the whole process.

Rename

Fowler states that the Rename refactoring should be employed when “the name of a method does not reveal its purpose”,

Borland have extended this definition to include fields, variables and resource strings.

Consider the following method (Listing 1):

```
function toRoman(num: integer): string;
var s : string;

begin
    s:='';

    while num >= 1 do
    begin
```

```

    s := s + 'I';
    num := num - 1;
end;

toRoman := s;
end;

```

Listing 1

Whilst its intent is fairly clear, we might wish to rename the “num” parameter to something more meaningful. Delphi 2005 lets us achieve this with the Rename refactoring. All we have to do is right click the mouse on any one of the references to “num”. In this instance, because “num” is a parameter of the toRoman function, Delphi 2005 classifies this refactoring as Rename Parameter as can be seen in Figure 1.

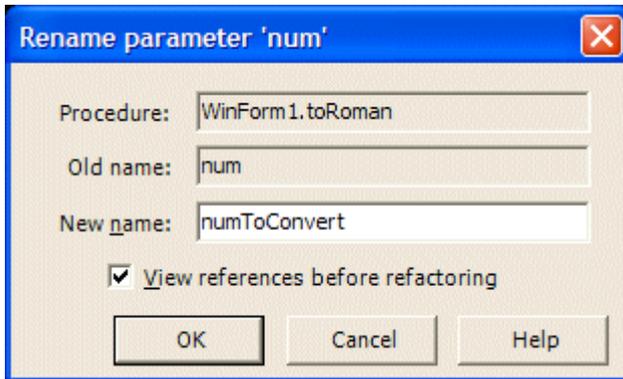


Figure 1: Delphi 2005’s Rename refactoring at work

The Rename refactoring expects us to provide a new name for the parameter/variable that we are renaming. Looking at figure 1 again, you will notice that I have simply renamed num to numToConvert.

By default the “View references before refactoring” checkbox is enabled. This is a sure-fire demonstration of why automated refactoring is better than manual refactoring. Delphi 2005 lets us preview the code that will be affected by the refactoring before we apply any changes. Take a look at figure 2, it presents Delphi 2005’s new “Refactoring” dock window.

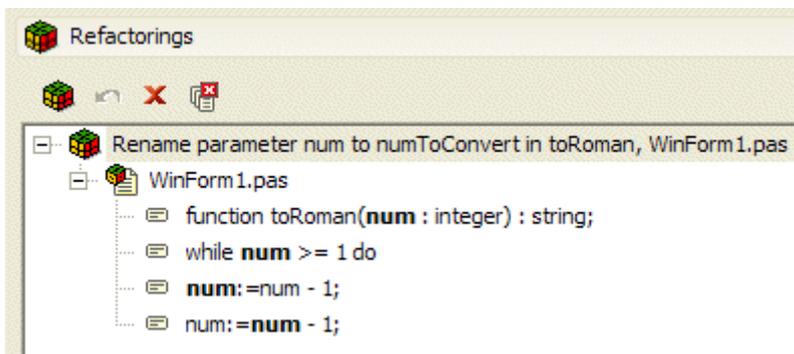


Figure 2: Delphi 2005’s Refactoring “browser”

This is the beauty of automated and tool support for refactoring. Delphi 2005 is able to use its own parse tree to identify the instances (within the code, not objects!) where the parameter/variable “num” is used. We are then able to visually check that these items should be renamed and apply the refactoring by clicking on the “Apply Refactoring” icon (the coloured cube) or by selecting the refactoring or by pressing Control-R.

Listing 2 presents the same piece of code after the Rename refactoring has taken place.

```

function toRoman(numToConvert: integer): string;
var s : string;

begin
    s := '';

    while numToConvert >= 1 do
    begin
        s := s + 'I';
        numToConvert := numToConvert - 1;
    end;
end;

```

```
toRoman := s;  
end;
```

Listing 2

Granted in this case a search'n'replace might have been quicker. However if you were to select a code fragment and apply a selective search'n'replace, you will still have to look at each and every change in turn. This problem exacerbates itself if the item you are renaming is embedded in other parts of the code fragment. For example, suppose the code fragment contained references to 'enumerator', it would be replaced with 'enumToConverterator', which would clearly cause a compile-time error and would require that you spend some time rectifying the problem.

Once a refactoring has been applied, the first thing we should do is make sure that the refactoring has not broken the code: it may still compile and run, but does it still do what it was meant to do? Perhaps not such a worry in this small example, however it is crucial if you are refactoring your mission critical applications! With that in mind, Borland have been kind enough to provide us with an undo facility. If a refactoring breaks your application, i.e. tests fail, it is possible to click on the Undo Refactoring icon or press Control-Z.

The Refactoring dock window appears to offer a history of the refactorings that you may have applied. It also seems to offer selective undo, however in my tests I found that it is more of a case of all or nothing, i.e. all refactorings are undone.

Simple Extract Method

It is very likely that this will be the most popular refactoring and one that you will find yourself using very frequently, especially on legacy code, complicated code, or code you did not write yourself. Delphi 2005's Extract Method refactoring is based on Fowler's Extract Method (110) refactoring. Indeed, my first foray into the world of refactoring relied on Extract Method for a lot of the time, such was the way the original code was written.

It is likely that you will find the Extract Method refactoring particularly useful if you know that your code has similar/identical code fragments replicated in more than one place, i.e. there is duplicated code. Whilst you know that the code works, any changes that you make in the future have to be applied to the duplicated code too.

The problem associated with duplicated code exacerbates itself when the code changes are given to somebody else. You may well remember to apply changes to all duplicated code because you can remember where they are, but a rookie or even an experienced developer might not appreciate that the code they have just changed has to be changed elsewhere too.

If you consider your own experience, how often do you fix a local problem and are loath to touch code that is remote or "one step removed" from the piece of code you have just fixed? Unless you implicitly know that the duplicated code exists, you are naturally reluctant to go looking for it never mind make two or more changes to code that is perhaps spread between different units. If the code under examination was written by somebody else, the reluctance to change "other" code increases considerably.

That is why Delphi 2005's Extract Method can help us. It offers us a means of consolidating duplicated code by extracting the true duplication into a single method, leaving behind a call to the extracted method.

The Extract Method refactoring, like the Rename refactoring, is intelligent. It has the capability of using Delphi 2005's parse tree to examine the code that you wish to extract into a single method.

This first example will consider a simple case whereby the original author has used comments to indicate what the proceeding code is about to achieve. Consider listing 3, it highlights two chunks of code that really should be a method in their own right, never mind the poorly labelled Listbox variables (we will rename these later!)

In a production application, it is very common to see similar comments tied in with another bad smell: long methods. In both cases, comment smell and long method smell, we should consider using Extract Method to make the code more readable and understandable. I know that I am guilty of using comments to indicate intent within an as yet unwritten method body. Invariably, the comments remain and I end up with a long method.

```
procedure TWinForm.PopulateListboxes;  
begin  
    // Populate metallurgy  
    Listbox1.Items.Add('Killed Carbon Steel');  
    Listbox1.Items.Add('Carbon Steel');  
    Listbox1.Items.Add('Stainless Steel');
```

```

// Populate joint type
Listbox2.Items.Add('Butt Welded');
Listbox2.Items.Add('Socket Welded');
end;

```

Listing 3: Bad Smell – comments suggesting method names...

If we select the three lines beginning “Listbox1” then invoke Delphi 2005’s Extract Method refactoring (Control-Shift-M), figure 3 appears.

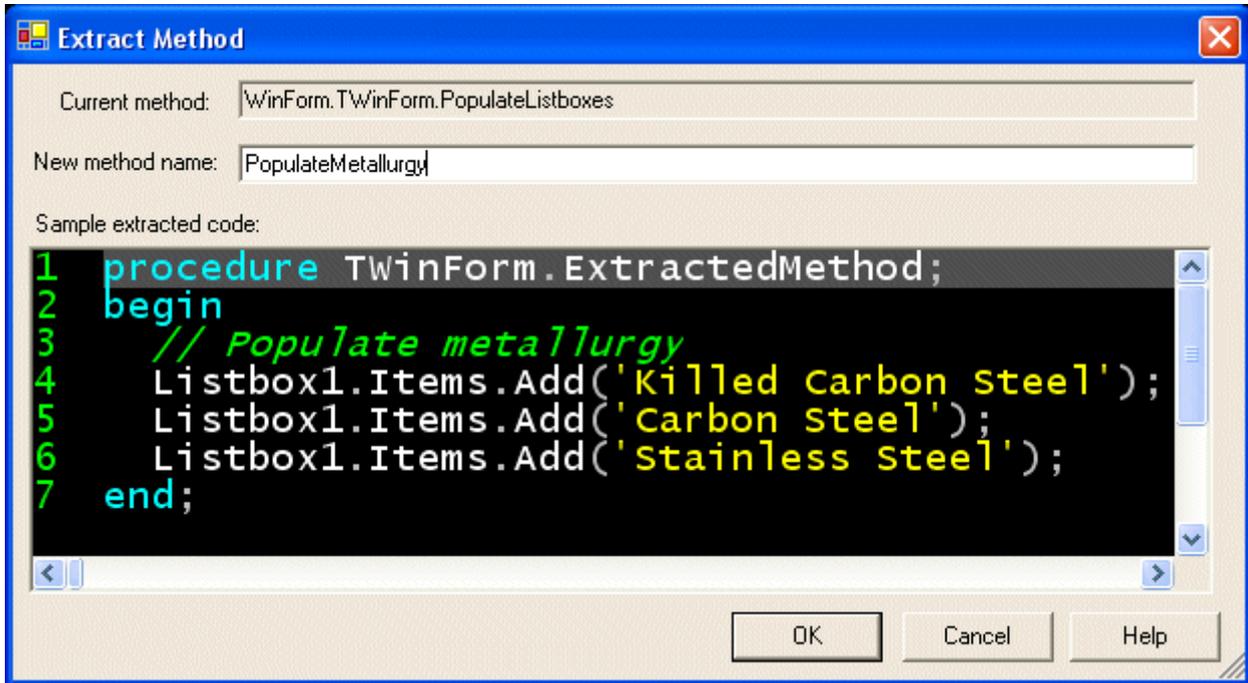


Figure 3: Delphi 2005’s Extract Method at work

Interestingly, the Extract Method refactoring has brought the comment along with it! Not to worry, it saves us removing it from the original method. This refactoring expects us to provide a name for the new method; I have chosen PopulateMetallurgy as that was the original comment. Clicking on OK creates Listing 4 and replaces the extracted code in the original method.

```

procedure TWinForm.PopulateMetallurgy;
begin
  // Populate metallurgy
  Listbox1.Items.Add('Killed Carbon Steel');
  Listbox1.Items.Add('Carbon Steel');
  Listbox1.Items.Add('Stainless Steel');
end;

```

Listing 4: Extract Method at work

After this first refactoring, the PopulateListboxes method now looks like Listing 5.

```

procedure TWinForm.PopulateListboxes;
begin
  PopulateMetallurgy;

  // Populate joint type
  Listbox2.Items.Add('Butt Welded');
  Listbox2.Items.Add('Socket Welded');
end;

```

Listing 5: After a simple Extract Method refactoring

Repeating the process for Listbox2 gives us Listing 7:

```

procedure TWinForm.PopulateListboxes;
begin
  PopulateMetallurgy;
  PopulateJointType;
end;

```

Listing 7: After two simple refactorings, the code is more readable

Complicated Extract Method

The next example will consider the case of duplicated code within the same method. Consider the following piece of code (Listing 8):

```
function toRoman(n: integer): string;
var s : string;
begin
  if (n<=3) then
    // Duplicated
    while (n > 0) do begin
      s := s + 'I';
      n := n - 1;
    end

  else if n = 4 then s := 'IV'
  else if n = 9 then s := 'IX'
  else if n >= 10 then begin
    s:='X';
    n:=n-10;
    // Duplicated
    while (n > 0) do begin
      s := s + 'I';
      n := n - 1;
    end;
  end
  else if n >=5 then begin
    s:='V';
    n:=n-5;
    // Duplicated
    while (n > 0) do begin
      s := s + 'I';
      n := n - 1;
    end;
  end;
end;

toRoman := s;
end;
```

Listing 8: Before the Extract Method refactoring

At the time of writing, November 2004, Delphi 2005 struggles with this particular refactoring. After a little experimentation, it appears that the Extract Method refactoring struggles with code this is part of or close to an IF condition. For example, if we select the very same duplicated code block in listing 6, Extract Method returns Listing 9:

```
procedure DealWith123(var n: Integer);
var s: string;
begin
  // Duplicated
  while (n > 0) do
  begin
    s := s + 'I';
    n := n - 1;
  end;
end;
```

Listing 9: Incorrectly Extracted

Clearly listing 9 will make any automated unit tests that we may (should) have fail. Whilst the parameter n is a var parameter, the variable s is magically local.

Interestingly, if we move the duplicated code fragment above the “if (n<=3) then”, then attempting to extract the duplicated code fragment, gives us the correct result. Listing 10 presents the code I was expecting from my first Extract Method attempt.

```
procedure DealWith123(var n: Integer; var s: string);
begin
  while (n > 0) do
  begin
    s := s + 'I';
    n := n - 1;
  end;
end;
```

Listing 10: Correctly Extracted

Declare Variable

This is a moderately useful refactoring based loosely on Fowler's original "Introduce Explaining Variable (124)" refactoring.

Delphi 2005 is smart enough to use its own parse tree to help identify the target type. For example, if you have introduced a new variable using an assignment, Delphi 2005 will create a new variable of the same type as that on the right hand side of the assignment statement. This is best explained by way of a short example.

Consider the following trivial line of code:

```
for n := 0 to 100 do begin end;
```

If we move the cursor close to the for loop control variable n, then choose the Declare Variable refactoring (Control-Shift-V) we are presented with figure 4.

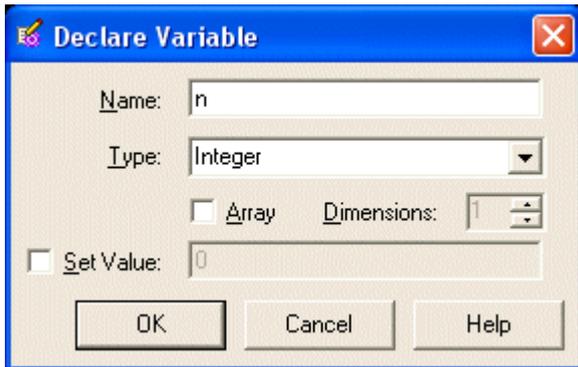


Figure 4: Delphi 2005's Declare Variable refactoring at work

The Declare Variable refactoring recognises that n is of type Integer – it does this by examining the code that appears close to the variable, in this case the "for loop". However, there may be a chance that you do not want to use an Integer, hence the availability of the drop-down menu of types.

Clicking on OK declares the variable n, as shown in Listing 11.

```
var  
  n: Integer;  
begin  
  for n := 0 to 100 do begin end;  
end;
```

Listing 11: Declare Variable

Declare Field

In a similar vain to Declare Variable, Declare Field lets us introduce a field in to the current class.

Consider Listing 12, it presents an undeclared variable/field fFlag.

```
begin  
  for n := 0 to 100 do begin  
    if fFlag then begin end;  
  end;  
end;
```

Listing 12: Before Declare Field

If we move the cursor over fFlag, then choose the Declare Field refactoring (Control-Shift-D) we are presented with figure 5.

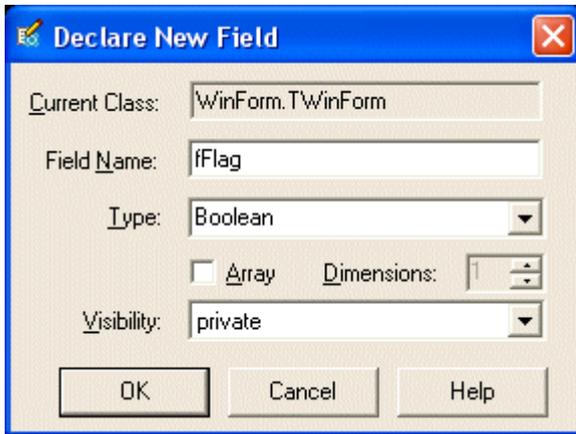


Figure 5: Delphi 2005's Declare Field refactoring at work

Again, based on what the refactoring engine sees around `fFlag` it works out the expected type for `fFlag`, in this case `Boolean`. Now, because we are declaring a new field within the current class, the refactoring engine needs us to help it with the field's visibility, which is `private` by default.

Clicking on `OK` leaves the cursor where it is, but does add the following code fragment to the current class declaration:

```
private
    fFlag: Boolean;
```

This refactoring will save a lot of time. Imagine the time that you spend inside a method, you introduce a new field, you then have to move back to the class declaration, add the field and then return to the method body. Bookmarks can help, as does `Control-Shift-Up/Down`, but it is still nowhere near as fast as this refactoring.

Extract Resource String

From experience, I think we all know that string literals in code are not the best thing. The Extract Resource String refactoring helps remove string literals by creating `resourcestring` equivalents. This refactoring is easy to apply, simply move the cursor over the string literal and press `Control-Shift-L`. Figure 6 will appear.



Figure 6: Delphi 2005's Extract Resource String at work

Clicking on `OK` adds the following code fragment to the current unit's implementation section:

```
resourcestring
    StrKilledCarbonSteel = 'Killed Carbon Steel';
```

If the process is repeated for each of the string literals in Listing 4, Listing 13 is the net result.

```
procedure TWinForm.PopulateMetallurgy;
begin
    // Populate metallurgy
    Listbox1.Items.Add(StrKilledCarbonSteel);
    Listbox1.Items.Add(StrCarbonSteel);
    Listbox1.Items.Add(StrStainlessSteel);
end;
```

Listing 13: Extract Resource String

Being able to rapidly apply the Extract Resource String refactoring is very useful. Prior to this refactoring, we had to go through a cut'n'paste exercise and we had to work out suitable names for the resource strings. Now Delphi 2005 does it for us, we can be a little more productive elsewhere!

Find Unit

According to the Delphi 2005 help, Find Unit is classified as a refactoring. In reality its purpose is to make locating units, particularly units with long names such as `System.Collections.Specialized.StringEnumerator`, easy to find.

Whilst not a major problem in previous versions of Delphi, now that we are looking at Delphi and the .net framework, the sheer profusion of .net namespaces (which equate to Delphi units) does mean we might find ourselves spending a lot of time either typing in fully qualified namespaces or a similar amount of time trying to locate/remember them! Hence Find Unit can be classed as a refactoring of sorts: if you believe that refactoring revolves around the ethos of productivity then yes, Find Unit is a refactoring.

Find Unit can be invoked by pressing Control-Shift-A, or via the Refactoring menu. Figure 7 presents the Find Unit dialog. As you can see, it offers us an incremental search facility, entering "String" narrows the list of available units down to just those containing the word "String". Of course, being Delphi, it also offers us the ability to add the unit to the public Interface section or to the private Implementation section.

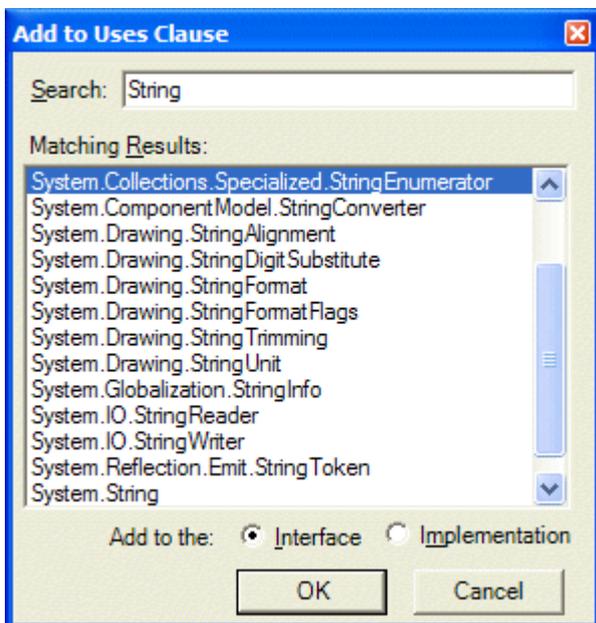


Figure 7: Delphi 2005's Find Unit feature at work

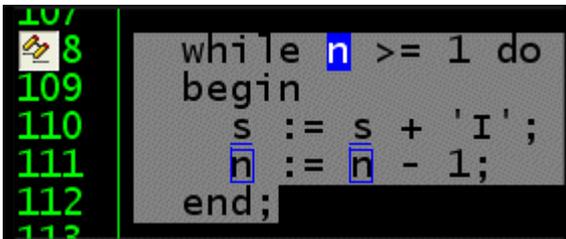
I did have a few issues with the Find Unit feature, notably I received the following error message: "Specified cast is not valid". This problem occurred on two separate machines, but went away on one machine after Delphi 2005 was shutdown and restarted.

Synchronised Editing

In addition, Delphi 2005's Sync Edit feature also allows us to perform some basic refactoring. Synchronised editing is a great little feature that allows us to select a block of code and then rename any poorly named variables within that block. It differs from the primary Rename refactoring in that it only applies to the code fragment that is selected.

Take a look at the code presented in Figure 8. I have selected the while loop, Delphi 2005 has indicated that the SyncEdit feature is available, signified by the presence of an icon in the left-hand gutter.

Clicking on this icon kicks off the SyncEdit process: the IDE then parses the selected code looking for items that you may wish to change en mass, in this case the variables `n` or `s`. By default SyncEdit will assume that you wish to change the first variable it finds, however pressing the TAB key whilst in SyncEdit mode moves the cursor between the possible items.



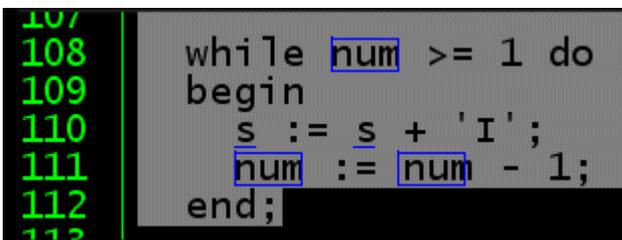
```
107  
108 while n >= 1 do  
109 begin  
110     s := s + 'I';  
111     n := n - 1;  
112 end;  
113
```

Figure 8: Sync Edit before synchronised editing has taken place

To rename the variable `n` to `num` in three places cost the time it takes to select the block of code (5 lines), the mouse click on the SyncEdit icon (or Ctrl-Shift-J) followed by a single cursor right and the letters 'um'. Granted not a massive time saving in this trivial example, however consider a larger example spread over more than 5 lines.

Equally, the fact that the IDE is performing the change for us is a good thing: we could have spent time changing a variable manually, only to find we had missed one, leaving the compiler to tell us. All these little mistakes add up and make us less productive. I know that using the compiler to identify "broken code" is a good idea and that compiler time today is cheap, however using the IDE ensures that we get it right first time!

Enough justification, after the fewest possible keystrokes, we end up with Figure 9.



```
107  
108 while num >= 1 do  
109 begin  
110     s := s + 'I';  
111     num := num - 1;  
112 end;  
113
```

Figure 9: Sync Edit after synchronised editing has taken place

Win32 Refactoring Support

If you wish to remain with a Win32 version of Delphi, such as 5, 6, or 7, you can still enjoy the benefits of refactoring by purchasing third party software such as ModelMaker's Code Explorer or Jacob Thurman's Castalia 2/3.

Ensuring Success

After a piece of code has been refactored, we have to be sure that the change has not affected the operation of the code fragment or the application as a whole. In order to ensure refactoring success, we need to re-test the code fragment and the application. Of course, manually testing code is boring and tedious as my test-driven development article confirms [5].

The solution to this problem is to maintain a suite of tests that can be run before refactoring and then again after refactoring - if the tests fail after the refactoring, the refactoring can be reversed and a re-test performed.

It is for this reason that unit testing, test-driven development plays a major part in the refactoring process. Prior to any refactoring, we should ensure that we have a suite of repeatable tests that can be run through an automated testing framework. Delphi 2005 comes supplied with NUnit [6], a testing framework for use with .net applications. If you are working with Win32 applications, DUnit [7] provide a similar testing framework.

Odd Bits...

On page 10 of the Delphi 2005 Overview PDF you will find a bold statement:

"At each stage, the refactorings ensure that the code works the same before the program was changed."

Whilst arguable, I would not agree that it is in refactorings that ensure the code works before and after refactoring. I strongly believe that it is the test cases that ensure any refactoring has not broken the code.

And on page 13 you will find:

"refactoring...also ensure that what the code developers create is correct."

Defining "program correctness" is a very emotive subject and opens up all kinds of arguments/discussions involving mathematics and formal notations. Unfortunately, if the pre-refactored code does not do what you expect (one definition of incorrect at least), then refactoring cannot magically "correct" your code.

What Do I Tell My Manager?

Hurried development, code smells, etc. mean you cut corners to get the job done. So you did not get it right first time. Not a problem, you made the release date, the product shipped. Certainly in my experience, after a product has shipped, the customer either likes what they see, and comes back asking for more, or development of the next version starts. Either way, the design debt that you incurred is still there to be paid off...and like all debts, interest is accruing, paying off the debt will cost us more time and cost than the original debt.

So how do you explain to you manager that you want to spend a period of time refactoring what is already out there and working?

Each year, Mercedes (please replace with the car your manager drives!), release a new version of their cars. Year on year, the physical appearance of the car does not change very much – this is because of the time it takes to change the metal pressing machines that make car parts. However, the interior and the trim are improved year on year. Small bugs in the electrics are ironed out; known faults within the engine compartment are dealt with, etc. Whilst the external characteristics of the car remain unchanged, the interior, the engine, the trim, etc. are refactored until they are economically perfect.

It is for this reason that the first release of a car typically suffers from lower sales figures: folks wait until the second release of the car knowing fine well that the manufacturer will have improved the existing design, removed bugs, etc. thus making the second and subsequent releases that much more appealing.

Like software development schedules, car manufacturers aim to hit marketing windows too, they have market demographics, "best time to sell", etc. in mind. They too cut corners [sic] in order to meet their deadlines.

If refactoring is good enough for the car makers, who sell your manager a new car ever 1-2 years, it should be good enough for software development professionals.

Summary

Code is an asset. If you let your code deteriorate it will lose its value. Do you want to re-invent the wheel each and every time you write an application? Refactoring allows us to inject life into our code; we can enhance the value of code by making it generic and re-usable. In turn, our next application might take us a little less time to develop...and that has to be a good thing.

Refactoring is all about a number of things. Productivity, increased understanding, greater reliability, increased customer satisfaction (fewer bugs, agile delivery/flexibility)

This Article's Resources

[1] Refactoring: Improving the Design of Existing Code, Martin Fowler, Addison-Wesley, 1999, ISBN 0-201-48567-2

[2] <http://www.refactoring.com>

[3] Refactoring Workbook, William Wake, Addison-Wesley, 2004, ISBN 0-321-10929-5

[4] Extreme Programming Explained, Kent Bent, Addison-Wesley, ISBN 201-61641-6

[5] Test-Driven Development Using Delphi 8, Craig Murphy, The Delphi Magazine, Issue 104 (April 2004)

[6] NUnit: <http://www.nunit.org>

[7] DUnit: <http://dunit.sourceforge.net/>

Craig is an author, developer, speaker, project manager, Certified ScrumMaster and Microsoft Most Valuable Professional (XML Web Services). He specialises in all things XML, particularly SOAP and XSLT. Craig is evangelical about .NET, C#, Test-Driven Development, Extreme Programming, agile methods and Scrum. He can be reached via e-mail at: bug@craigmurphy.com, or via his web site: <http://www.craigmurphy.com> (where you can also find the source code and PowerPoint files for all of Craig's articles, reviews and presentations).