Refactoring

Extracted from Code Better in Delphi for ADUG 2022

begin

begin 1

What is this “refactoring” you speak of? 1

A Malodourous Source 2

Specific Refactorings 2

Re-ordering Statements 2

Rename Refactoring 2

Automated Refactorings 4

Extract Method 4

Extract Method Object 6

Split With 7

Introduce Explaining Variable 9

Summary 10

Further Learning 11

Contact 11

end. 11

What is this “refactoring” you speak of?

The number 12 can be expressed by the statement 3×4=12. That is to say, 12 has the factors of 3 and 4. However, we could express 12 differently, such as 6×2=12, so we could take 3×4 and refactor it to 6×2, and we haven’t changed the fact that our statement expresses 12. There’s little complexity difference between 3×4 and 6×2. However, there are additional ways of expressing 12 - you can imagine something with powers of 4, logarithms and cubed roots. This would be a complicated way of expressing 12, which might be better expressed by something simpler (by, umm, I dunno, maybe 12).

Similarly, code can be expressed in different ways. Some might be more complicated than others, sometimes faster, more efficient, or easier to understand. Refactoring is the process of changing code from one form to another - without its behaviour changing. Notice I stated “the process of changing”, not the “result of changing”. This is a subtle yet important distinction. Like your maths teacher requiring you to “show your working”, you need to follow

a process to change code from one form to another to ensure that the behaviour has not changed and bugs are not introduced. If you don't, you're not refactoring - you're just changing code.

There is an extensive library of scripts that you can follow to transform your code. These scripts are called refactorings. Many of these were written down by Martin Fowler in his book Refactoring - Improving the Design of Existing Code (now in its second edition).

I would like to point out that when I'm working on legacy code, I will spend some time just randomly refactoring code to help me understand how it works, revert all my changes, and begin cleaning up the code in earnest.

A Malodourous Source

There is a concept in refactoring called "code smells", which are certain patterns and structures in your code that might indicate areas that can be improved by refactoring. It's important to note that these smells are not bugs. The code can work perfectly but might be difficult to maintain. Each smell has a name such as "too many parameters" where a method might have, well, I think the name speaks for itself. Or "feature envy", where a class uses the methods of another class excessively. Each of these smells will have some possible refactorings that can be performed to deodourise your code.

Specific Refactorings

Here are some basic refactorings that might be useful. Some of these are specific to Delphi, others will work with any language. These are some of the most common refactorings, and you can get a long way with knowing only these. But if you want to become an expert in refactoring, you will need to do more studying from the Further Learning section.

Re-ordering Statements

This is not a traditional refactoring, but often you will have statements (or groups of statements) within a method that are independent. That is, they can be executed in any order, giving you the freedom to move them around. This will allow you to group common concepts, suggesting further refactoring (particularly the Extract Method Refactoring below). Sometimes just reording statements can make code much easier to read but be very careful as some statements may have side effects that others depend upon.

Rename Refactoring

This is one of the simplest refactorings, changing the name of something - and we've already seen how the names of things can significantly affect both the readability and maintainability of code. There are many things you can change the name of: variables, methods, procedures,

types, units and so on. One of the simplest things we can rename is a local variable, which we are going to do here.

```
var
  CustomerIndex : integer;
begin
  for CustomerIndex := 0 to Customers.Count-1 do
    Customers[CustomerIndex].ProduceInvoice;
end;
```

Here, we want to rename the variable `CustomerIndex` to something more manageable, perhaps `Index` or `i`. The reason `i` is a common choice is that in mathematical notation (with `j` and `k`), it is used as a summation variable, which then transferred to early texts on computation and has been a tradition in programming ever since. It's also convenient that it's the first letter of index and iterator (which might be why it was originally chosen). This is a long-winded way of explaining why this single letter variable `i`, and subsequently `j` and `k`, are fine in this context.

To perform the refactoring, we just rename the variable and then try to compile. Compilation will fail, showing you where you need to rename its various instances. This technique of compiling and fixing errors is called "leaning on the compiler", as you let the compiler find all the instances of what you need to rename. You would probably manually rename all the instances that you can see before compiling.

You can also do "search and replace" by highlighting the code you want, pressing *Ctrl+R*, searching for the old name, and replacing it with the new name. Remember to select **whole words only** and unselect **match case**.

Yet another alternative is to highlight the code and use the synchronised edit function, either by clicking the icon shown on the left of the image below or using the keyboard shortcut *Ctrl+Shift+J*.

This will allow you to replace every instance of the variable name in one go visually. Although, you need to remember that neither of these replacement techniques knows anything about context. If you have a local variable called `caption` you want to rename, and you also highlight `form1.caption`, it will also rename that instance of `caption`.

Scoping is probably your biggest problem here. For instance, if you are changing a string variable called `caption` in a method on a form, then if you miss an occurrence of the `caption` variable, the `caption` property on the form will be used - and you won't notice a problem until something goes wrong at runtime. The reverse is true as well. You could inadvertently rename the caption property of a form, rather than a local variable. If you are using `with` statements, then "all bets are off" as determining scope can become very confusing.

The easiest way to avoid these issues is to use the automatic refactoring tools built into Delphi. These can save you a significant amount of manual effort.

Automated Refactorings

If you click on the variable's name anywhere in the code, you can either **Right-click|Refactoring|Rename** (*Ctrl+Shift+E)*, or go to the main menu and select **Refactoring|Rename**. You can then just specify whatever name you like - hopefully, something with some meaning - and the IDE will rename the variable for you. This is quite handy for renaming local variables, but it is much more useful when renaming something used outside the unit.

For instance, if you want to rename `DataModule1` to `dmImages`, an operation that might have far-reaching effects within your application. It might be referenced many times throughout your code and perhaps in your DFM files. The compiler will happily warn you about places in your code that you need to update, but it might not be until runtime that you get an error about a bad reference in one of your DFM files (say, a button referencing an image list on the data module). However, suppose you use the rename automated refactoring (the shortcut is *Ctrl+Shift+E* - remember - know thy shortcuts). In that case, all the properties in the DFM files will also be updated for you, saving you much frustration (and perhaps also your users if you don't have any automated testing).

Extract Method

After Rename, Extract Method is the most common refactoring that I use and is a fantastic way of reducing the complexity of code. If you take a large procedure and split it in two, the total complexity from the two procedures is likely less than from the single large procedure. This is because complexity rises faster than linearly with line count. The human brain can only handle so many concepts simultaneously, and if you go beyond this, you may run into trouble.

Extract Method can be fantastic for reducing duplicated code. Often you will see an identical block of code repeated twice or more. This is a great candidate for Extract Method, as each of these identical blocks can be replaced by a single method call. These code blocks might be in the same function or unit or distributed throughout an entire project (or projects).

It can also reduce the number of lines in a method by grouping lines of code that have a similar concept together into their own method. Provided that this method has a name that describes this concept, it will make the original method easier to understand.

This time I'm going to forgo the explanation of the manual Extract Method process and go straight to the automated one, but first, here is some sample code - taken from refactoring.com and converted to Delphi.

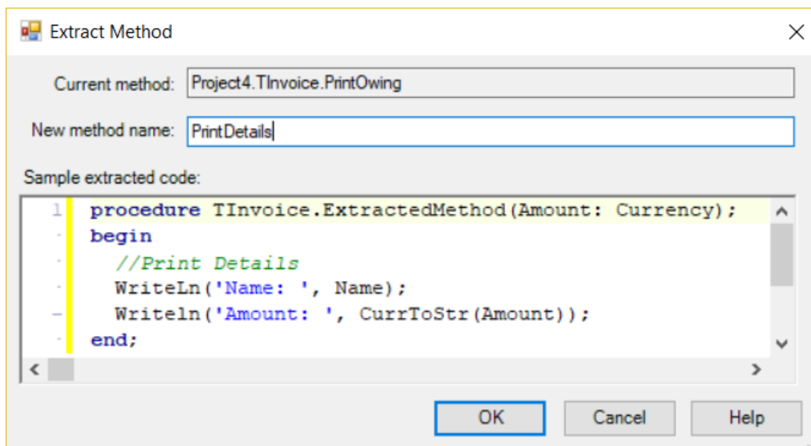
```
procedure TInvoice.PrintOwing(Amount: Currency);
```

```
begin
  PrintBanner;

  //Print Details
  WriteLn('Name: ', Name);
  Writeln('Amount: ', CurrToStr(Amount));
end;
```

Becomes

```
procedure TInvoice.PrintOwing(Amount: Currency);
begin
  PrintBanner;
  PrintDetails(Amount);
end;

procedure TInvoice.PrintDetails(Amount: Currency);
begin
  WriteLn('Name: ', Name);
  WriteLn('Amount: ', CurrToStr(Amount));
end;
```

Highlight the two `WriteLn` statements, then either **Right-click|Refactoring|Extract Method**, or do the equivalent from the menu bar. Or do what I did and use the shortcut *Ctrl+Shift+M*.



This will then show you the new method, including any parameters (in this case, `Amount`). The tool is also smart enough to add local variables to the extracted method and remove local variables from the source method. The extracted method will always be a procedure (never a function), so you have to refactor it yourself if you want it to be a function.

It can be quite instructive just to see what an extracted method looks like by highlighting some code and bringing up the Extract Method window. If there are not many parameters, it might be a suitable candidate. This is, of course, assuming that the code belongs together and you can find a suitable name.

When refactoring some legacy code, if I see a comment relating to a block of code, I look to see if that code can be extracted with the method's name based on the comment, which is what happened in the above example. The new method name became `PrintDetails`, making the "Print Details" comment redundant and therefore removable.

One deficiency with the implementation of this in Delphi is that the refactoring will not hunt out any duplicate code and replace that with the extracted method, leaving you to do this manually and possibly make mistakes.

The Automated Extract Method tool in Delphi has been quite unstable in some versions of Delphi. I've had many troubles with XE7 (missing parameters or just not working), but it got better in later versions. Delphi 10.4 and 11 have problems when you use inline variables. There is also an Extract Method in the Model Maker Code Explorer (MMX) IDE Plugin.

Extract Method Object

In this refactoring, we convert a procedure or function into an object. This can be very handy when you have some horribly huge procedure that you want to break down into smaller parts, but you want to keep those parts together. The basic operation looks like this:

```
procedure ComplicatedMethod(P1, P2: integer);
var
  Variable1 : string;
  Variable2 : string;
begin
  { Code }
end;
```

Becomes:

```
type
  TExtractedComplicatedMethod = class
  private
    Variable1, Variable2 : string;
    P1, P2: integer;
  public
    constructor Create(P1, P2: integer);
    procedure Invoke;
  end;

constructor TExtractedComplicatedMethod.Create(P1, P2: integer);
begin
  inherited Create;
  Self.P1 := P1;
  Self.P2 := P2;
```

```
end;

procedure TExtractedComplicatedMethod.Invoke;
begin
  { Code }
end;

procedure ComplicatedMethod(P1, P2 : integer);
var
  Extracted: TExtractedComplicatedMethod;
begin
  Extracted := TExtractedComplicatedMethod.Create(P1, P2);
  try
    Extracted.Invoke;
  finally
    Extracted.Free;
  end;
end;
```

The `ComplicatedMethod` body gets copied to the `Invoke` method on our new class, with the class's variables and parameters becoming fields. You might think that we've just created a bunch more code that we need to maintain, and you would be correct. However, we can now begin several extract method refactorings within the `Invoke` method, and those methods become `private` methods of our class. It also makes it easier to inject any global state, either to the constructor, `Invoke` or as properties/setters on the class. After all this, it might be possible to write tests for some or all of our new class.

There are many variations on this, including passing parameters to `Invoke`, having `Invoke` be a class method (no need to instantiate the class), keeping the local variables with `Invoke`, etc. It's just going to depend on what suits the situation.

I remember doing this on a particularly horrible procedure of over one thousand lines, much of which seemed to be indented halfway across the screen. Previously, I had spent hours trying to understand what it did, with minimal success. By breaking down the code, I could begin to understand it. I noticed that a sizable chunk of code could never be reached and so I could remove it.

Split With

This is a refactoring that is specific to Delphi. It's a mechanism to reduce the impact of a `with` statement. Ideally, you want to remove every `with` statement from your code, but because it is so easy to go wrong when removing them (particularly when they're nested), you probably only want to remove them where you are modifying code. Sometimes I want to extract a method from within a `with` statement. However, the automated extract method refactoring

doesn't work from within a `with` statement. So the code below shows a refactoring where we split a `with` statement in half.

```
{ ... }
with X do
begin
  StatementsThatRelyOnX; { lots of complex code }
  StatementsThatDontRelyOnX;
  OtherStatementsThatRelyOnX;
  StatementsThatRelyOnX; { more complex code }
end;
{ ... }
```

Becomes

```
{ ... }
with X do
begin
  StatementsThatRelyOnX;
  StatementsThatDontRelyOnX;
  OtherStatementsThatRelyOnX;
end;
with X do
begin
  StatementsThatRelyOnX;
end;
{ ... }
```

So far, we've achieved absolutely nothing beyond adding a few lines of code in the middle. However, you can then further adjust the code to something like:

```
{ ... }
with X do
begin
  StatementsThatRelyOnX;
end;
StatementsThatDontRelyOnX;
X.OtherStatementsThatRelyOnX;
with X do
begin
  StatementsThatRelyOnX;
end;
{ ... }
```

And because the middle two statements (or more in an actual refactoring) are no longer tied to the `with` statement, we could extract them into a method using the automated Extract Method refactoring. I also feel that two smaller `with` blocks are easier to deal with than one enormous block, as you now have four edges to shift code from instead of just two. It is possible to extract a `with` statement provided the whole `with` statement is within the extraction.

Introduce Explaining Variable

Also, sometimes called Extract Variable. If you have some complicated statement (often conditional logic in an `if` statement). You take some of that logic and assign it to a variable with a name that explains that logic. This is better explained by an example. This one is taken from refactoring.com.

```
Result := Order.quantity * Order.ItemPrice -
  Max(0, order.quantity - 500) * Order.ItemPrice * 0.05 +
  Min(Order.quantity * Order.ItemPrice * 0.1, 100);
```

Becomes

```
BasePrice = Order.Quantity * Order.ItemPrice;
QuantityDiscount = Max(0, Order.Quantity - 500) * Order.ItemPrice * 0.05;
Shipping = Min(BasePrice * 0.1, 100);
Result := BasePrice - QuantityDiscount + Shipping;
```

As you can see, we have broken up a single statement into four. Rather than being an incomprehensible calculation, it has become four simple calculations that are relatively easy to follow. The first three statements of this refactoring comprise parts of the original calculation assigned to an 'Explaining Variables'. The code achieves the same calculation, except now you can understand how its parts relate together.

Let's look at an example I used in a CodeRage talk

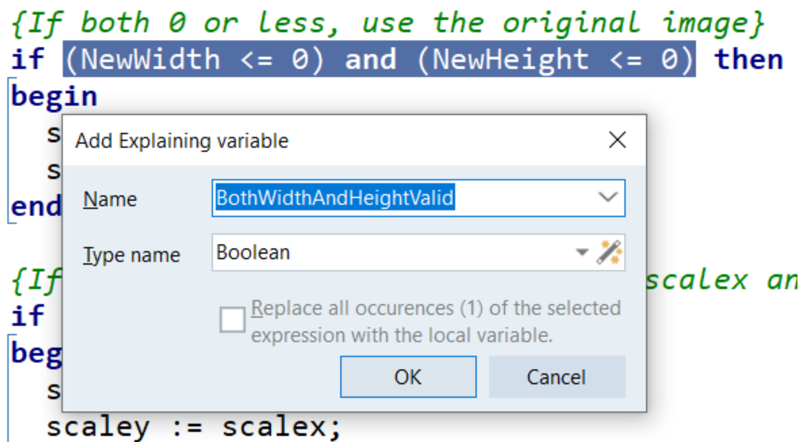
```
procedure TSomeObject.ProcessCommand(Command: string; Priority: integer);
begin
  if ((LowerCase(Command) = 'open') or (LowerCase(Command) = 'unlock')
    or (LowerCase(Command) = 'unfasten')) and ((Priority = 0) or (Priority > 5))
then
  begin //if the Command string is an open command or the Priority is not set or
high
  { ... Some Code ... }
  end;
end;
```

If we look at the comment (which is always a great place to start when refactoring), we see that there are two separate conditions in this `if` statement. If we apply our Introduce Explaining Variable refactoring, we will get:

```
procedure TSomeObject.ProcessCommand(Command: string; Priority: integer);
var
  IsOpenCommand : boolean;
  PriorityIsHighOrNotSet : boolean;
begin
  IsOpenCommand := MatchText(Command, ['open', 'unlock', 'unfasten']);
  PriorityIsHighOrNotSet := (Priority = 0) or (Priority > 5);
  if IsOpenCommand and PriorityIsHighOrNotSet then
  begin
  { ... Some Code ... }
  end;
end;
```

You should also notice that I've simplified the `IsOpenCommand` logic from what it was initially (do this in a second step). The `MatchText` routine is in the `System.StrUtils` unit, a good knowledge of the RTL can not only reduce the amount of code you write but make your code easier to understand and extend at the same time.

You can use the automated refactoring in the IDE (**Refactoring|Introduce Variable...**) or if you have Model Maker Code Explorer (MMX) installed you can use its command (**MMX|Add|Add Explaining Var...** *Shift+Alt+L*). Both require you to highlight the code you want to introduce a variable for and invoke the command.



Shown is Add Explaining Variable in MMX. Notice that it is capable of replacing multiple occurrences (as is Introduce Variable in the IDE).

Summary

I would have liked to include more refactorings but had to stop somewhere as this is an introduction to refactoring, not an entire book on the topic. The refactorings I have included are fairly straightforward, and you should be able to take advantage of them immediately.

Further Learning

The book you want to read is Refactoring: Improving the Design of Existing Code by Martin Fowler. This is the bible on refactoring and is now in its second edition. The examples in the first edition are in Java and Javascript for the second, but I didn't find a problem translating the concepts to Delphi. The book itself is pretty accessible and straightforward to follow. I've read it cover to cover twice and looked up refactorings many times. Originally written in 1999, but still completely relevant over 20 years later. Also, Working Effectively with Legacy Code by Michael Feathers is a great resource.

Contact

Email: alister@learndelphi.tv
YouTube: https://www.youtube.com/user/codegearguru
Web: https://LearnDelphi.tv
LinkedIn: https://linkedin.com/in/alisterchristie/
Twitter: https://twitter.com/AlisterChristie
Facebook: https://www.facebook.com/LearnDelphitv/
Instagram: https://www.instagram.com/christiealister/

I look forward to hearing from you as you continue your Delphi journey.

```
end.
```