This paper was originally presented at the Southwest Fox conference
in Gilbert, Arizona in October, 2014. http://www.swfox.net

Refactoring VFP Apps

Rick Borup
Information Technology Associates, LLC
701 Devonshire Dr, Suite 127
Champaign, IL 61820
Voice: (217) 359-0918
Email: rborup@ita-software.com

Do your applications suffer from Stale Software Syndrome? When you crack open a section of existing code for some simple maintenance work, are you reminded that you've frequently said "I really need to clean this up some day"? Does a simple enhancement request end up taking hours instead of minutes because you never did clean up that section of code? I'd wager these are common occurrences for most developers, and the longer these situations linger the worse they can get. In this session we'll explore how and why code gets stale, and how refactoring can be an effective solution.

Introduction

Code gets stale for many reasons. Developers make changes in a hurry, new features get tacked on without much regard for the original design, or the original design may not have been very good to begin with. Programming languages and development environments evolve and improve, but apps developed in an earlier version often remain stuck there.

Over time, stale code becomes increasingly cumbersome and difficult to work with. Changes take longer than they should, and new features are delayed because the structure of the code doesn't readily accommodate them.

These kinds of accumulated issues are sometimes referred to as *technical debt*. When technical debt grows large enough, developers end up spending more of their time working around it – in essence, paying "interest" on the debt – instead of doing productive work.

When that happens, it's time to refactor.

What refactoring is

Refactoring is the process of improving the internal design of software without changing its external behavior.

Internal design refers to the structure of the software. In object-oriented software, this means the way the classes are constructed and combined as well as the way code is written within class methods. Refactoring is the process of improving the way the software is structured, so in that sense refactoring is another word for restructuring.

External behavior means the behavior that can be observed when the software is run. One of the key tenets of refactoring is that no change be allowed to alter the output produced by the software for any given set of inputs. A *legal* refactoring is one that follows this rule, although not all legal refactorings lead to real improvement.

The most popular and frequently cited book on refactoring is probably Martin Fowler's *Refactoring: Improving the Design of Existing Code* [1], which I'll refer to throughout this paper simply as *Refactoring*.

Purpose of refactoring

In his Ph. D. thesis at the University of Illinois, William Opdyke characterized refactoring as a way to make software easier to understand, change, and reuse [2].

The classes in an object-oriented software application typically vary in purpose from general to specific. The more general a component, the more likely it can be reused "as is" or modified slightly to fulfill a similar requirement. Before a developer can modify or reuse a class, he or she must first understand it. Once understood, the class becomes easier to modify or reuse.

Fowler points out that without refactoring, the internal structure of software applications tends to decay over time. This is caused primarily by changes that are made, often under the pressure of deadlines, without full regard to their future consequences. The original design loses cohesion, the structural integrity of the software is compromised, and future changes become more difficult. When this happens, refactoring can help by making the code more amenable to changes and additions.

In a nutshell, the primary purpose of refactoring is to facilitate the process of change [1]. Conversely, the need to make changes often provides the motivation to begin refactoring.

What refactoring isn't

It's important to recognize that refactoring is *not* about improving performance, fixing bugs, or adding features.

Improved performance may be a happy by-product of refactoring, although in some cases a bit of performance may actually be sacrificed in return for a better internal structure. While not intended to fix bugs, refactoring can often reveal the root cause of a bug and thereby make the bug easier to fix. And although refactoring itself does not, by definition, add new features, it is frequently a useful precursor in situations where the existing structure of the code does not readily support the addition of new features.

Reasons to refactor

The sub-title of this paper is "If it ain't broke, fix it!". This is meant to provoke you to ask "Why would I want to fix something that isn't broken?" In other words, if a piece of software is meeting its requirements and performing satisfactorily, why expend any effort to change it? What's the potential benefit? Is changing it worth the risk of breaking something that's currently working? And, given that refactoring is strictly internal and the end user isn't going to notice any difference anyway, who's going to pay for all that work?

Those are all legitimate questions, and it's perfectly valid to take the opposing viewpoint and say "If it ain't broke, *don't* fix it". I'll come back to that point of view later, when discussing some of the risks of refactoring, but for now let's look at situations where it *does* make sense to refactor working code.

Code Smells

When talking about source code, the term "bad smell" refers to something that can be observed in the design and construction of a piece of software rather than something that necessarily affects its performance or produces wrong results. In *Refactoring*, author Martin Fowler attributes the concept of bad code smells to contributor Kent Beck, who said (my paraphrasing) that developers can sense when a piece of code needs to be changed in the same way parents can sense when their baby's diaper needs to be changed: in both cases, something just doesn't smell right.

Every experienced developer has probably encountered bad code smells at one time or another, either in their own code or in code they've inherited from someone else. Some common examples are overly long methods, duplicated code, awkward class construction, methods with too many parameters, and many more.

When a bad code smell is detected, the question becomes “how bad does it smell, and do I care enough to do anything about it?”. A developer’s typical thought process may look something like **Figure 1**.

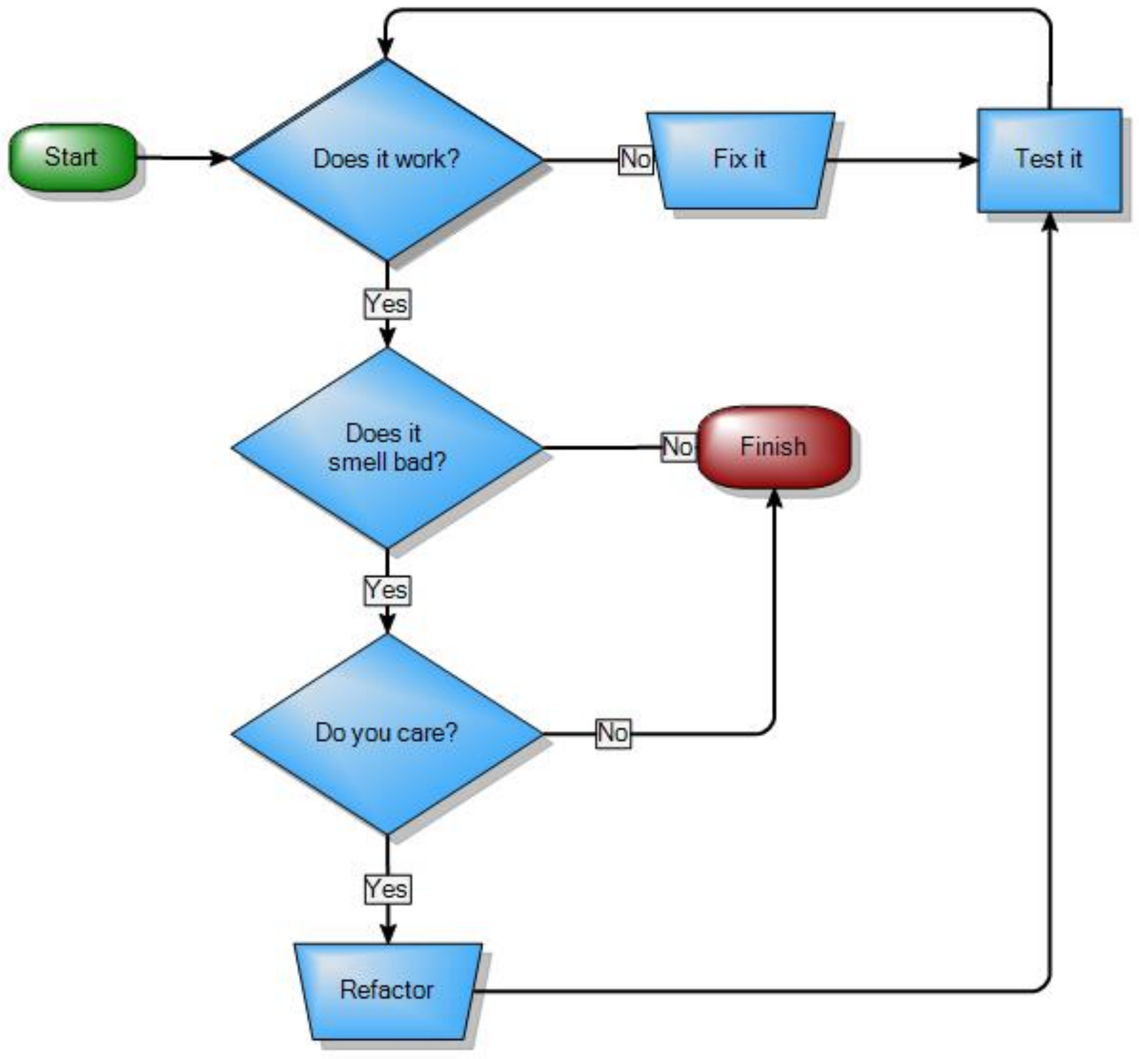


Figure 1. A developer’s typical thought process might look something like this.

Chapter 3 of *Refactoring* is devoted entirely to naming and describing several types of bad code smells. Knowing how to recognize them helps you identify places in code that are good candidates for refactoring.

The chapter describes many bad code smells, but this paper mentions only a few that I’ve commonly run across in my own work. (I hasten to add that I work mostly on my own code, so there's no doubt who's to blame for creating these smells in the first place!) The first five listed here are mentioned in *Refactoring*, while the remainder are ones I've added from my own experience.

Duplicated Code

How often have you had experiences like these?

- You're developing a new piece of code that needs to perform some function you know you've already programmed elsewhere, but you can't remember exactly where and you don't have time to go looking for it, so you end up re-writing essentially the identical code in a new place.
- You need a slightly modified version of some code that already exists elsewhere, and you even know where to find it, but you're short on time so instead of creating an abstract class and then specializing with subclasses, you copy the original code, paste it into the new location, tweak it to suit the new purpose, and forget about it.
- You find yourself writing essentially the same conditional statement in several different places in an app. It could be a simple IF…THEN…ELSE or a more complex CASE statement. Either way, you recognize that the code is making the same decisions over and over again.

All of these situations, and others, are root causes of duplicated code—one of the most common of all bad code smells.

Long Method

A long method is one that tries to do too much. Long methods are problematic because they're unwieldy, difficult to test, and risky to change. Personally, I don't believe there is any ideal length – say, one printed page or less – nor that there is any rigid definition of what constitutes "too long" or "too much". I think the definition of these terms depends on the context, but within a given context it's generally not difficult to spot methods that qualify as being too long or trying to do too much.

Large Class

Like an overly long method, an overly large class can lead to problems. When too many methods are crammed into a single class, the class can become unwieldy and the methods can become inter-dependent on one other, even if not intentionally. This type of informal coupling hampers reusability.

Long Parameter List

Methods with a long list of parameters can be cumbersome at best, even with tools like IntelliSense to prompt for each one as you create a method call. A long list of parameters may also be an indication that a method is trying to do too much, particularly if not all the parameters are needed in every path through the method code.

Switch Statements

Switch statements (CASE statements in VFP) are commonly used when the code needs to choose among two or more different but typically related execution paths depending on some condition. In some languages the logical condition on each leg of the switch statement is restricted to testing the value of a single variable. In VFP there is no such restriction, so VFP switch statements can become quite lengthy and diverse.

The question of whether or not a switch statement indicates a bad code smell hinges not only on its length but also on the level at which it occurs in the object hierarchy of the running application. The higher up a switch statement occurs, the more likely it is to be needed in more than one place and therefore the more likely it is to end up being duplicated elsewhere. A refactoring that pushes the decision down to lowest reasonable level in the object hierarchy helps reduce the potential for duplication.

Chaining Parameters

Chaining parameters refers to situations where one method passes a parameter to another method, which in turn passes it to a third method, and so on. While there may be situations where chaining parameters makes sense – for example, if the first method to accept the parameter validates or modifies it before passing it on – this is generally indicative of a structural problem. This is particularly true if the parameter is not actually used by the methods further up the chain, but rather is there merely so it can be passed on to lower methods where it is finally used.

Passing local memory variables as parameters within a class

The code smell arises when a method on a class passes one of its own local memory variables as a parameter to another method on the same class. There may be valid reasons for this, but it would generally be considered better design to use a class property in place of the local memory variable, if for no other reason than that to help avoid the "long parameter list" smell.

VFP Code Smells

There are some code smells that tend to be unique to Visual FoxPro apps. Most of them can be attributed to VFP's roots in FoxPro for DOS (FPD). Many VFP developers first "learned how to do it" in FPD and still carry some of those old habits with them when working in VFP. I am no exception here.

Procedural Code

VFP still allows procedural code, and the compiler doesn't impose any penalty for using it. As a result, it's not uncommon to find older VFP apps whose structure is a hybrid mixture of procedural and OOP design elements.

Other than a tiny main.prg to act as the stub for launching an app, procedural code in VFP qualifies as a bad code smell because the scope of variables can be vague, procedure and function calls appear to exist in a vacuum, it can't take advantage of encapsulation of data or polymorphism of method names, and so on. It's usually not difficult to convert procedural code to OOP, so there's really no excuse for allowing old procedural code to hang around in a VFP app.

Business logic in event methods

In FoxPro for DOS, it was common for business logic to be written directly in the Valid event of a field in a screen program, or at best delegated it to a function or procedure in the screen's Cleanup code. Either way, all the code was in the screen program itself. The concept of separating business logic from presentation wasn't widely used in those days.

As FPD developers transitioned to VFP, that kind of thinking sometimes came along for the ride and business logic ended up being written in the Click event method of a VFP form instead of in a separate business class. The problem with placing business logic in an event method is that it's not reusable. If the business logic that runs when a user clicks on a control on a form is embedded in the form itself, there's no good way to run that same code from another place in the app.

Technical Debt

We incur technical debt when we write code the quick and dirty way. The interest we pay on technical debt is the time it takes to go back and do it right. Refactoring is one way to pay off technical debt, but not all refactoring is debt payment.

A lot is being written and said these days about xDD: test-driven development (TDD), behavior-driven development (BDD), and so on. In the real world, a lot of us work under what I call deadline-driven development (DDD).[1]

When working under deadline-driven development, a developer may realize he or she is incurring technical debt even as the code is being written but may also choose to live with it temporarily because of the deadlines and other pressures. The problem is "temporary" often turns into "permanent" as other tasks come along and take priority, so the technical debt is never paid off.

Technical debt doesn't only result from hurry-up design. Over time, even the best designs decay as changes are made and additions tacked on. Whatever its cause, there is a point at which dealing with technical debt begins to suck up unreasonable amounts of time.

Other Reasons to Refactor

Bad code smells aside, the primary driving force for refactoring is the need to make changes to an application whose existing structure doesn't lend itself well to those changes. Evolving standards can be a stimulus for refactoring when code written to conform to earlier standards is made obsolete by newer standards or technologies. Even without code smells, technical debt, the need to make changes, or changing standards, refactoring can be employed to improve the clarity of existing code for other team members who may be unfamiliar with it.

[1] I know DDD also stands for domain-driven design, but that's a different story.

How to refactor

This section describes a few of the refactoring techniques I commonly use. With the exception of the first, the names of the techniques listed here are from *Refactoring*.

Rename Variables

Problem: The name of a variable does not describe the data it holds.
Solution: Rename the variable.

Renaming variables is not strictly a refactoring because it doesn't change the structure of the code, so you won't find it in Fowler's catalog of refactorings. In my own work, though, renaming variables is often the first thing I do when I begin working with an unfamiliar application whose meaning is not initially clear.

Compilers need only valid syntax, but developers need understanding. Cryptic variable names don't matter to the compiler, but they can be problematic for human beings because they obscure, or at best fail to reveal, the meaning of what they represent. When working with a piece of code whose variable names do not convey much meaning, changing the variable names is an easy way to improve understanding.

Renaming variables is a deceptively simple but nonetheless powerful technique. Not only does it help the developer who does the actual renaming, but it also improves the code's readability for others who may follow. Consider the following line of code:

```
lnT = lnP * ( 1.00 + lnM/100 )
```

It's clear that it multiplies two values and stores the result in a third, but it's not at all clear what the three variables represent. The person who wrote this code probably knew what each one stood for at the time, but the developer who later inherits this code doesn't get a clue.

Inspection of the context in which this code appears (not shown) reveals that `lnT` is a total amount, while `lnP` is a price and `lnM` is a markup percentage. A simple renaming makes this clear to anyone reading the code:

```
lnTotalPrice = lnBasePrice * ( 1.00 + lnMarkupPercent/100 )
```

The beauty of object oriented code is that changing the names of locally scoped variables has no effect on the rest of the system. The principle is the same for class properties, although it entails more work because changing the name of a property also requires changing it everywhere that property is referenced. The use of setter and getter methods can mitigate the effects of changing a property name – more on that later.

Rename Method

"The name of a method does not reveal its purpose.
Change the name of the method." [1]

Like *Rename Variables*, *Rename Method* is a simple but useful refactoring. The convention is that the name of a method should contain a verb and describe what the method does. For example, consider a method named *TotalPrice*, a name that is vaguely meaningful but which does not really describe what the method does. If it turns out the method calculates a total price, then a better name for that method would be *CalculateTotalPrice*.

Extract Method

"You have a code fragment that can be grouped together.
Turn the fragment into a method whose name explains the purpose of the method." [1]

As a method is initially being developed, it's not uncommon for the developer to embed a few lines of code that are required by the method but which don't necessarily belong inline. If such a fragment is likely to be needed elsewhere in the system, the design can be improved by extracting it from its original location and making it into a separate method. The *Extract Method* refactoring moves the fragment to a new method with its own name and replaces the inline code with a call to the new method.

In the following code, the second line truncates the fractional cents in a monetary amount. This is a generic function that's likely to be needed elsewhere, so it probably deserves to be a method of its own.

```
lnTotalPrice = lnBasePrice * ( 1.00 + lnMarkupPercent/100 )
lnFinalPrice = lnTotalPrice – mod( lnTotalPrice, .01)
```

After applying the *Extract Method* refactoring, the code looks like this:

```
lnTotalPrice = lnBasePrice * ( 1.00 + lnMarkupPercent/100 )
lnFinalPrice = this.TruncateFractionalCents( lnTotalPrice)

Function TruncateFractionalCents( tnValue as Number) as Number
Return tnValue – mod( tnValue, .01)
```

Note that the new method TruncateFractionalCents has a meaningful name.

Move Method

"A method is, or will be, using or used by more features of another class than the class on which it is defined.
Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether." [1]

Fowler describes moving methods as the "bread and butter of refactoring" [1]. *Move Method* is similar to *Extract Method*, but instead of creating a new method from a fragment, you're moving an entire method to another class. *Move Method* requires more analysis to determine if and when to do it, and it requires more work to implement, but the benefit is a looser and more flexible design.

In the previous example, *Extract Method* was applied to move the truncate fractional cents calculation from its inline position into its own method. The unanswered question is, which class does the new method belong in? If the methods in the original class are, for example, primarily devoted to calculating the price of line items on an invoice, then a generic method such as TruncateFractionalCents is probably more well suited to a class whose other methods are also generic arithmetic functions.

Continuing with the example above, the *Move Method* technique can be applied to move the TruncateFractionalCents method to a class named ArtithmeticFunctions. To implement this, first copy the method from the original class to the target class. Then delete the method from the original class and revise the method call(s) to reference the new class.

```
lnTotalPrice = lnBasePrice * ( 1.00 + lnMarkupPercent/100 )
loMath = NEWOBJECT( "ArithmeticFunctions") && If not already instantiated
lnFinalPrice = loMath.TruncateFractionalCents( lnTotalPrice)

DEFINE CLASS ArithmeticFunctions as Custom
Function TruncateFractionalCents( tnValue as Number) as Number
Return tnValue – mod( tnValue, .01)
ENDDEFINE
```

The benefit of this design change is that the TruncateFractionalCents method is no longer embedded in a class that also has all the methods and data required to work with invoices. The method can now be used anywhere simply by instantiating the ArithmeticFunctions class, which is presumably lighter weight because it does not have all the overhead of the invoice handling class. This makes the TruncateFractionalCents method suitable for use in places completely unrelated to handling invoices.

Extract Class

"You have one class doing work that should be done by two.
Create a new class and move the relevant fields and methods from the old class into the new class." [1]

The *Extract Class* refactoring is indicated when you realize a class has become overly complex, unwieldy, or is simply doing too many things. It's similar to *Move Method* but involves splitting an entire class in two instead of moving only a single method.

Classes grow and evolve over time, with new methods and responsibilities being added as the need arises. After a while a class can end up with methods that, while it may have been convenient to include them at the time, can now be seen as not closely related to the class's original purpose. When this happens, the design can be improved by extracting the methods that don't belong in the original class and putting them into a new class of their own.

As an example, consider a class whose basic purpose is to handle all things related to an invoice, such as calculating line item amounts, applying discounts, calculating the invoice total, and printing the invoice. When that class was originally written, invoices were always printed to paper – there were no other types of output. As time went by, methods were added to print the invoice to a PDF file instead of directly to paper, and then later to render it in HTML format suitable for emailing to the customer.

At first, the PDF and HTML methods were probably tacked on to the original invoice class, where they worked fine and caused no problems except that the class began to bloat. But now, when viewing the class with refactoring in mind, it becomes clear that the printing to paper, creating a PDF file, and rendering as HTML are each ways of generating report output and have nothing to do with invoices per se. Therefore, these three methods probably belong in a new class by themselves and *Extract Class* can be applied to accomplish this.

Remove Parameter

"A parameter is no longer used by the method body.
Remove it". [1]

Not much needs to be said here – this is about as simple as it gets. There is, as you might expect, a companion refactoring called *Add Parameter*. You've doubtless done both a hundred times and never thought about them as formal refactorings.

A design can be improved by one or the other of these techniques if the result reduces complexity or eliminates redundancy elsewhere in the code. Implementation is easy but requires a walk through the entire application because the method's signature is changed. It may also be necessary to add validation code when a new parameter is added.

Setter and Getter Methods

A setter method is a method whose purpose is to set the value of a class property. Conversely, a getter method is a method whose purpose is to return the value of a class property.

Setter and getter methods uncouple the name of a class property from its outside references. A property's name can therefore be changed without affecting anything outside of the class itself.

Some languages and some approaches to class design make extensive use of setter and getter methods, while others do not. In Visual FoxPro, setter and getter methods are optional. Unless defined as protected or hidden, a VFP class property's value can be written or retrieved by direct reference anywhere in the code. This has both advantages and disadvantages. The advantage is that the developer is free to reference the property wherever desired. The disadvantage is that there's no control over who can access the property's value, much less over what range of values or even what type of value can be assigned to it.

A getter method can be used to restrict access to a class property in some way, for example by checking user permissions. If a class property is defined as protected, a getter method is the only way to access the property's value from outside the class or one of its subclasses. If the property is defined as hidden, a getter method is the only way to access the property's value from anywhere outside the class.

A setter method is commonly used to control changes to the value of a property. As with getter methods, this could include checking user permissions, but setter methods can also be used to ensure that a valid data type and/or range of values is satisfied before allowing a property's value to be changed.

The name of setter and getter methods typically includes the word SET or GET along with the name of the property. For example, given a class with a character data-type property called cDataPath, the corresponding setter and getter methods might look like this:

```
Procedure SetDataPath( tcDataPath as String) as Void
If Vartype( tcDataPath) = “C”
   This.cDataPath = tcDataPath
Endif

Procedure GetDataPath() as String
Return this.cDataPath
```

In this example, the SetDataPath method protects the integrity of the cDataPath property by ensuring it can be assigned only character values. The GetDathPath method, on the other hand, is a pure getter that does nothing except return the value of cDataPath. In both cases, as long as the methods themselves are not defined as protected or hidden, they can be called from anywhere and work as expected even if the cDataPath property itself is protected or hidden.

VFP's native Access and Assign methods are a special form of getter and setter. You don't have to use them, but if you do, keep in mind they cannot be referenced outside of the class.

Adding setter and getter methods to a class constitutes a refactoring because it enhances and/or restricts the way the class's properties can be referenced, which might in turn require changes in the code that references them. Some developers may swear by the use of setter and getter methods while other developers spurn them as extraneous. Either way, as with many things in VFP, the choice is yours.

When to refactor

Knowing when to refactor is as important as knowing how to refactor. The short answer is, “Now!”.

In *Refactoring*, Fowler introduces the analogy of developers wearing two hats, one while writing new code and the other while refactoring existing code. Rather than wearing the developer hat for a long period of time and then changing to the refactoring hat, he suggests it's better to switch hats frequently, perhaps even multiple times per day. In other words, don't wait too long to refactor: write some new code, get it working, and then look for opportunities to refactor it almost immediately.

Others have commented on when to refactor, too. Ron Jeffries recently published an article on XProgramming.com [6] about the benefits of refactoring as you go along. It's a quick read, illustrated with some fun graphics. He talks about how to avoid the need for a huge refactoring job by "… [improving] the code where we work, and [ignoring] the code where we don't have to work." Using the analogy of weeds and bushes to represent problem areas in code, he writes "We take the next feature that we are asked to build, and instead of detouring around all the weeds and bushes, we take the time to clear a path through some of them." The benefit is that "With each new feature, we clean the code needed by that feature." Thus, refactoring is driven by change and accomplished in manageable chunks.

Both authors are saying essentially the same thing: "Don't wait to refactor." Do it as soon as you can, do it in small bites, and do it when the code is fresh in your mind. It's much easier to manage a dozen small refactorings over time than to wait until you have to tackle one big refactoring later on.

Jeff Atwood published a blog post in 2004 entitled "Don't be afraid to break stuff!" [7]. Although written in more general terms, his advice pertains directly to the challenges of refactoring. Face it: when you refactor you change code, and when you change code you're likely to break something. Jeff's point is that breaking stuff is a good way to figure out how it works. So go ahead and tackle that refactoring job. See what breaks, then fix it. You'll learn a lot in the process.

When not to refactor

Not all refactoring results in program improvement. In his Ph.D. thesis on refactoring, William Opdyke points out that "applying arbitrary refactorings to a program is more likely to corrupt the design rather than to improve it…" [2].

The important question is, when is it appropriate to refactor and when is it not? There's no one answer – what's right in one situation may be wrong in another, and vice-versa. The decision ultimately rests with the developer, but Opdyke provides this guiding principle: "A refactoring improves design if the resultant code units correspond to meaningful abstractions that make it easier to refine or extend the program. " [2]

Opdyke's thesis focuses on frameworks and therefore deals largely with abstractions, but not all refactorings are intended to create abstractions. Some are intended to create specialization by removing specific behavior from a higher-level class and encapsulating it in a lower-level class where it's easier to reuse.

In both cases, however, the same question should be asked: does the proposed refactoring improve the structure of the program, or does it unnecessarily complicate or obfuscate it? Again, the question can only be answered by the designer/developer in the context of the situation at hand.

Examples of refactoring in VFP

The First Example

The first example illustrates how a bad smell can develop in a piece of code over time, and how refactoring can help get rid of it. The code in this example resides in a method named GetPurchaseOrders on a form named frmTimer. A timer on that form fires the method at data-driven intervals to download and process incoming purchase order files from various customers via FTP. Each customer has its own unique IT system, so while the overall process of downloading and processing order files is essentially the same for all customers, the details – the address of the customer's FTP server, the type of FTP connection required, and the name, file type, and data content of the orders to be downloaded – are different for each customer. The application therefore has a separate business class to handle the specific requirements for each customer.

Listing 1 is the initial version of the GetPurchaseOrders method. At this point there was only one customer, Alpha Corp., but the developer used a CASE statement in anticipation of other customers being added later. The method:

- Stops the timer
- Sets a visible status indicator to show which company is being processed
- Instantiates the appropriate business class and call its GetIncomingFiles method
- Resets the visible status indicator to "Waiting"
- Restarts the timer

Listing 1 – The original code handles only one customer, Alpha Corp.

```
* frmTimer.GetPurchaseOrders()
LPARAMETERS tcCompany
thisform.StopTimer()
DO CASE
   CASE UPPER( tcCompany) = "ALPHACORP"
      thisform.SetStatus( "Processing", "Getting purchase orders for Alpha Corp")
      loAlphaCorp = NEWOBJECT( "AlphaCorp", "clsAlphaCorp.prg")
      loAlphaCorp.GetIncomingFiles()
   OTHERWISE
      *  Other companies may be added later
ENDCASE
thisform.SetStatus( "Waiting")
thisform.StartTimer()
```

This code isn't too bad. It's constructed as a stand-alone method on the form instead of residing in the event code of the control that runs it, and it delegates the details of the downloading and processing the incoming files to a method on a separate business class.

When the user of this application lands a contract with a second customer, Beta Corp., the code is easily extensible. The developer follows the original design concept and simply adds another leg to the CASE statement.

This works fine, but the seeds of a bad code smell are already planted: there's a clear duplication of code, the only difference being which business class gets instantiated. In addition, unique object names are used when a generic name would suffice. Listing 2 shows the code after adding the second customer – changes are indicated in green.

Listing 2 – The code after adding a second customer

```
* frmTimer.GetPurchaseOrders()
LPARAMETERS tcCompany
thisform.StopTimer()
DO CASE
   CASE UPPER( tcCompany) = "ALPHACORP"
      thisform.SetStatus( "Processing", "Getting purchase orders for Alpha Corp")
      loAlphaCorp = NEWOBJECT( "AlphaCorp", "clsAlphaCorp.prg")
      loAlphaCorp.GetIncomingFiles()
   CASE UPPER( tcCompany) = "BETACORP"
      thisform.SetStatus( "Processing", "Getting purchase orders for Beta Corp")
      loBetaCorp = NEWOBJECT( "BetaCorp", "clsBetaCorp.prg")
      loBetaCorp.GetIncomingFiles()
   OTHERWISE
      *  Other companies may be added later
ENDCASE
thisform.SetStatus( "Waiting")
thisform.StartTimer()
```

When the third customer, Gamma Corp., is added, this structure begins to look like it could become cumbersome. Maybe a CASE statement wasn't such a good idea after all. Listing 3 shows the code after the existing structure was extended by adding a third leg to the CASE statement.

Listing 3 – The code after adding a third company

```
* frmTimer.GetPurchaseOrders()
LPARAMETERS tcCompany
thisform.StopTimer()
DO CASE
   CASE UPPER( tcCompany) = "ALPHACORP"
      thisform.SetStatus( "Processing", "Getting purchase orders for Alpha Corp")
      loAlphaCorp = NEWOBJECT( "AlphaCorp", "clsAlphaCorp.prg")
      loAlphaCorp.GetIncomingFiles()
   CASE UPPER( tcCompany) = "BETACORP"
      thisform.SetStatus( "Processing", "Getting purchase orders for Beta Corp")
      loBetaCorp = NEWOBJECT( "BetaCorp", "clsBetaCorp.prg")
      loBetaCorp.GetIncomingFiles()
   CASE UPPER( tcCompany) = "GAMMACORP"
      thisform.SetStatus( "Processing", "Getting purchase orders for Gamma Corp")
      loGammaCorp = NEWOBJECT( "GammaCorp", "clsGammaCorp.prg")
      loGammaCorp.GetIncomingFiles()
   OTHERWISE
      *  Other companies are not programmed yet
ENDCASE
thisform.SetStatus( "Waiting")
thisform.StartTimer()
```

Fowler's *Refactoring* [1] introduces the "rule of threes", which goes like this:

- The first time you do something, just do it.
- The second time you do something similar, you wince at the duplication but you do the duplicate thing anyway.
- The third time you do something similar, you refactor.

The code in Listing 3 is a good example of the rule of threes. This code is beginning to smell bad. The unnecessarily unique object names mean it won't be easy to refactor, and the same commands are duplicated in all three legs of the CASE statement with the only difference being the business class that's instantiated. It's not hard to see how this structure will become burdensome as more and more customers are added. It's time to put on our refactoring hat!

The first refactoring consists of two changes. The first change involves the call to the SetStatus method, which is currently duplicated in each leg of the CASE statement. The duplication can be eliminated by removing the individual calls from within the CASE statement and replacing them with a single call up above. This is an example of the *Consolidate Duplicate Conditional Fragments* technique. A parameter can now be used to pass the name of the company to the SetStatus method.

The second change in this first refactoring is an example of *Rename Variables*, in which the unnecessarily unique variable names in each leg of the CASE statement are replaced with the single generic but descriptive object name *loCompanyHandler*. Listing 4 shows the code after the first refactoring is complete.

Listing 4 – The first refactoring sets the stage for the next one.

```
* frmTimer.GetPurchaseOrders()
LPARAMETERS tcCompany
thisform.StopTimer()
thisform.SetStatus( "Processing", "Getting purchase orders for " + ;
                   ALLTRIM( tcCompany))
LOCAL loCompanyHandler as Object
DO CASE
   CASE UPPER( tcCompany) = "ALPHACORP"
      loCompanyHandler = NEWOBJECT( "AlphaCorp", "clsAlphaCorp.prg")
      loCompanyHandler.GetIncomingFiles()
   CASE UPPER( tcCompany) = "BETACORP"
      loCompanyHandler = NEWOBJECT( "BetaCorp", "clsBetaCorp.prg")
      loCompanyHandler.GetIncomingFiles()
   CASE UPPER( tcCompany) = "GAMMACORP"
      loCompanyHandler = NEWOBJECT( "GammaCorp", "clsGammaCorp.prg")
      loCompanyHandler.GetIncomingFiles()
   OTHERWISE
      *  Other companies are not programmed yet
ENDCASE
thisform.SetStatus( "Waiting")
thisform.StartTimer()
```

This code is better, but there's still a lot of duplication and the CASE statement is still going to grow a new leg every time another company is added.

The second refactoring invokes the *Extract Method* technique, which is one of the most basic and frequently used. The CASE statement is extracted from the GetPurchaseOrders method and converted to a new method named GetIncomingFiles. This refactoring also applies *Consolidate Duplicate Conditional Fragments* to replace the three duplicate calls to loCompanyHandler.GetIncomingFiles with a single one below the CASE statement.

Listing 5 – The second refactoring uses *Extract Method* to create a new method for the CASE statement.

```
* frmTimer.GetPurchaseOrders()
LPARAMETERS tcCompany
thisform.StopTimer()
thisform.SetStatus( "Processing", "Getting purchase orders for " + ;
                   ALLTRIM( tcCompany))
thisform.GetIncomingFiles( tcCompany)
thisform.SetStatus( "Waiting")
thisform.StartTimer()

* frmTimer.GetIncomingFiles()
LPARAMETERS tcCompany
LOCAL loCompanyHandler as Object
DO CASE
   CASE UPPER( tcCompany) = "ALPHACORP"
      loCompanyHandler = NEWOBJECT( "AlphaCorp", "clsAlphaCorp.prg")
   CASE UPPER( tcCompany) = "BETACORP"
      loCompanyHandler = NEWOBJECT( "BetaCorp", "clsBetaCorp.prg")
   CASE UPPER( tcCompany) = "GAMMACORP"
      loCompanyHandler = NEWOBJECT( "GammaCorp", "clsGammaCorp.prg")
   OTHERWISE
      *  Other companies are not programmed yet
ENDCASE
loCompanyHandler.GetIncomingFiles()
```

The only thing this second refactoring has accomplished is to push the CASE statement down into a new method of its own, but that's still an important step in the right direction. The GetPurchaseOrders method on the form has now been shortened and simplified, and the decision as to which business class to instantiate method has been delegated to the new GetIncomingFiles method. So far this may look like a meaningless change, but it sets the stage for the final refactoring.

The third and final refactoring in this example gets rid of the CASE statement in the form's GetIncomingFiles method and replaces it with a call to a factory object. (You knew your knowledge of design patterns would pay off at some point, didn't you?) This reduces the form's GetIncomingFiles method to its essentials, and facilitates the addition of more companies by simply adding them to the factory class. Because the factory object could conceivably be needed elsewhere in the application, it's designed as a new class of its own. This is an example of the *Extract Class* refactoring technique.

Listing 6 – The final refactoring

```
* frmTimer.GetPurchaseOrders()
LPARAMETERS tcCompany
thisform.StopTimer()
thisform.SetStatus( "Processing", "Getting purchase orders for " + ;
                    ALLTRIM( tcCompany))
thisform.GetIncomingFiles( tcCompany)
thisform.SetStatus( "Waiting")
thisform.StartTimer()

* frmTimer.GetIncomingFiles
LPARAMETERS tcCompany
LOCAL loCompanyHandler as Object, loFactory as Object
loFactory = NEWOBJECT( "CompanyHandler")
loCompanyHandler = loFactory.CreateCompanyHandler( tcCompany)
loCompanyHandler.GetIncomingFiles()

*--------------------------------------*
*         Simple Factory Class
*--------------------------------------*
DEFINE CLASS CompanyHandler as Custom
FUNCTION CreateCompanyHandler( tcName)
LOCAL lcName
lcName = UPPER( ALLTRIM( tcName))
RETURN ICASE( tcName = "ALPHACORP", NEWOBJECT( "AlphaCorp", "clsAlphaCorp.prg"), ;
              tcName = "BETACORP",  NEWOBJECT( "BetaCorp", "clsBetaCorp.prg"), ;
              tcName = "GAMMACORP", NEWOBJECT( "GammaCorp", "clsGammaCorp.prg"), ;
              null )
ENDFUNC
ENDDEFINE
```

After this refactoring, the code that resides in the form itself has been simplified and abstracted to the point that no changes are required when a new company is added. Additions can now be made by adding a single line to the factory class. Also, the factory method can use the leaner ICASE statement in place of a full CASE statement structure.

What makes this the final refactoring? Without meaning to be glib, it's because this is where I decided to stop. Remember, refactoring changes are internal – the end user sees no difference, so the developer gets to define what "done" looks like.

The Second Example

The second example of refactoring begins with a method whose purpose is at first unknown to the developer assigned to work on it. The name of the method suggests it has something to do with a unit price, whatever that is, but it's difficult for the developer to figure out what the code actually does just by reading it, in part because the variable names are meaningless. I've made them artificially so for the sake of the example, but most developers have probably seen code similar to this in real life.

Listing 7 – It's difficult to figure out what this code does because variable names are meaningless.

```
FUNCTION UnitPrice( a, b, c, d, e, f, g, h)
LOCAL ln1, ln2, ln3, ln4
STORE 0.00 TO ln1, ln2, ln3, ln4
DO CASE
   CASE d = "01"
      ln4 = c
   CASE e = "WEIGHT"
      ln4 = b * c
   OTHERWISE
      ln4 = b
ENDCASE
DO CASE
   CASE h = 1
      ln1 = ROUND( ( ( a + g) / ln4), 3)
      ln1 = ln1 - MOD( ln1, .01)
   CASE h = 2
      ln1 = ROUND( ( ( a + g)/ ln4), 2)
   OTHERWISE
      ln1 = ROUND( ( ( a + g) / ln4), 3)
      ln3 = ( ln1 * 100) - INT( ln1 * 100)
      IF ln3 > 0
         ln1 = ln1 + .005
      ENDIF
      ln1 = ROUND(ln1, 2)
ENDCASE
RETURN ln1
ENDFUNC && UnitPrice
```

A good first step toward understanding what this method does is to figure out what each variable represents and then to give it a meaningful name. After researching how this method is used in the application, the developer sees that it calculates a unit price – in other words, the price of a single item – based on certain variables including the price of a full case of those same items. For example, if the thing in question is a case of 24 bottles of cola, this method calculates the price of a single bottle if one can be sold separately.

Once the developer understands this, she decides a good first step is to apply the *Rename Method* refactoring by adding a verb to the method name. Naming it CalculateUnitPrice instead of just UnitPrice helps make its purpose clear to anyone reading it.

After inspecting the calling code more thoroughly and seeing what data is being passed to it in each of the parameters, the developer determines that the first parameter is the price of a case, the second through the seventh are other factors affecting the calculation, and the last parameter specifies one of three types of rounding. Knowing this, she can apply *Rename Variables* to give the parameters meaningful names. This improves readability both for herself and for other developers who may follow, as shown in Listing 8.

Listing 8 – The method has been renamed and the parameters now have meaningful names.

```
FUNCTION CalculateUnitPrice( tnCasePrice, tnCount, tnSize, tcCategory, ;
                             tcUnitType, tcSpecial, tnUnitChrg, tnRoundingMethod)
```

```
LOCAL ln1, ln2, ln3, ln4
STORE 0.00 TO ln1, ln2, ln3, ln4
DO CASE
   CASE tcCategory = "01"
      ln4 = tnSize
   CASE tcUnitType = "WEIGHT"
      ln4 = tnCount * tnSize
   OTHERWISE
      ln4 = tnCount
ENDCASE
DO CASE
   CASE tnRoundingMethod = 1
      ln1 = ROUND( ( ( tnCasePrice + tnUnitChrg) / ln4), 3)
      ln1 = ln1 - MOD( ln1, .01)
   CASE tnRoundingMethod = 2
      ln1 = ROUND( ( ( tnCasePrice + tnUnitChrg) / ln4), 2)
   OTHERWISE
      ln1 = ROUND( ( ( tnCasePrice + tnUnitChrg) / ln4), 3)
      ln3 = ( ln1 * 100) - INT( ln1 * 100)
      IF ln3 > 0
         ln1 = ln1 + .005
      ENDIF
      ln1 = ROUND(ln1, 2)
ENDCASE
RETURN ln1
ENDFUNC && CalculateUnitPrice
```

The name of the local memory variable names are still cryptic, so the next step is to apply *Rename Variables* to them as well. While she's at it, the developer also figures out the difference between the three rounding techniques. The difference isn't immediately apparent from the code, so she adds a comment for each one. These comments increase readability and will also become useful in a later refactoring.

Listing 9 – The local memory variables now have meaningful names, too.

```
FUNCTION CalculateUnitPrice( tnCasePrice, tnCount, tnSize, tcCategory, ;
                             tcUnitType, tcSpecial, tnUnitChrg, tnRoundingMethod)
LOCAL lnPrice, lnCents, lnTenths, lnDivisor
STORE 0.00 TO lnPrice, lnCents, lnTenths, lnDivisor
DO CASE
   CASE tcCategory = "01"
      lnDivisor = tnSize
   CASE tcUnitType = "WEIGHT"
      lnDivisor = tnCount * tnSize
   OTHERWISE
      lnDivisor = tnCount
ENDCASE
DO CASE
   CASE tnRoundingMethod = 1  && truncate
      lnPrice = ROUND( ( ( tnCasePrice + tnUnitChrg) / lnDivisor), 3)
      lnPrice = lnPrice - MOD( lnPrice, .01)
   CASE tnRoundingMethod = 2  && normal rounding
      lnPrice = ROUND( ( ( tnCasePrice + tnUnitChrg) / lnDivisor), 2)
   OTHERWISE && round up
```

```
      lnPrice = ROUND( ( ( tnCasePrice + tnUnitChrg) / lnDivisor), 3)
      lnTenths = ( lnPrice * 100) - INT( lnPrice * 100)
      IF lnTenths > 0
         lnPrice = lnPrice + .005
      ENDIF
      lnPrice = ROUND(lnPrice, 2)
ENDCASE
RETURN lnPrice
ENDFUNC && CalculateUnitPrice
```

The method now makes pretty good sense to someone reading it, but it still has some issues. The developer notices that the local memory variable lnCents is never used, so she simplifies the code by getting rid of it. Removing of a local memory variable doesn't affect anything else, so this is an easy change.

The developer also observes that the tcSpecial parameter is not used anywhere in the method code, which suggests the *Remove Parameter* refactoring can be used to get rid of it. Of course, removing a parameter changes the method's signature and potentially breaks all the calling code, so the developer has a decision to make: is it better to leave the unused parameter in place, with the only penalty being some potential confusion as to why it's there, or is it worth the effort to remove it and then to locate and change all the calling code? In the real world the answer may well depend on how many places the method is called, or on how energetic the developer is feeling that day, but for purposes of this example she decides to remove it.

Another source of potential confusion is that the name of the local memory variable lnPrice isn't very descriptive. It would be more accurate to name it lnUnitPrice, because that name better describes the data it holds, so she does that too.

Looking at the two inline CASE statements, the developer can see that the first one sets the value of a divisor that's used to calculate the unit price, while the second one applies one of three rounding methods to a numeric value. Both of these appear to be somewhat generic functions that could conceivably be needed elsewhere in the application. She knows the main problem with inline code like this is that it's not reusable because it can't be called from anywhere else in the application.

The developer applies *Extract Method* to pull both of these CASE statements out of their inline location and to convert each into its own method, named CalculateUnitPriceDivisor and CalculateRoundedUnitPrice respectively. There are two advantages to this. One is that these calculations can be utilized anywhere else in the application, and the other is that by giving the new methods meaningful names, their purpose is clear wherever the method is referenced.

Listing 10 – The two CASE statements are extracted out of the original method and converted into methods of their own.

```
FUNCTION CalculateUnitPrice( tnCasePrice, tnCount, tnSize, tcCategory, ;
                             tcUnitType, tnUnitChrg, tnRoundingMethod)
LOCAL lnUnitPrice, lnDivisor
```

```
STORE 0.00 TO lnUnitPrice, lnCents, lnTenths, lnDivisor
lnDivisor = CalculateUnitPriceDivisor( tcCategory, tcUnitType, tnCount, tnSize)
lnUnitPrice = CalculateRoundedUnitPrice( tnCasePrice, tnUnitChrg, lnDivisor, ;
                                         tnRoundingMethod)
RETURN lnUnitPrice
ENDFUNC && CalculateUnitPrice

FUNCTION CalculateUnitPriceDivisor( tcCategory, tcUnitType, tnCount, tnSize)
LOCAL lnDivisor
DO CASE
   CASE tcCategory = "01"
      lnDivisor = tnSize
   CASE tcUnitType = "WEIGHT"
      lnDivisor = tnCount * tnSize
   OTHERWISE
      lnDivisor = tnCount
ENDCASE
RETURN lnDivisor
ENDFUNC && CalculateUnitPriceDivisor

FUNCTION CalculateRoundedUnitPrice( tnCasePrice, tnUnitChrg, tnDivisor, ;
                                    tnRoundingMethod)
LOCAL lnTenths
STORE 0.00 TO lnTenths
DO CASE
   CASE tnRoundingMethod = 1  && truncate
      lnUnitPrice = ROUND( ( ( tnCasePrice + tnUnitChrg) / lnDivisor), 3)
      lnUnitPrice = lnUnitPrice - MOD( lnUnitPrice, .01)
   CASE tnRoundingMethod = 2  && normal rounding
      lnUnitPrice = ROUND( ( ( tnCasePrice + tnUnitChrg)/ lnDivisor), 2)
   OTHERWISE && round up
      lnUnitPrice = ROUND( ( ( tnCasePrice + tnUnitChrg) / lnDivisor), 3)
      lnTenths = ( lnUnitPrice * 100) - INT( lnUnitPrice * 100)
      IF lnTenths > 0
         lnUnitPrice = lnUnitPrice + .005
      ENDIF
      lnUnitPrice = ROUND(lnUnitPrice, 2)
ENDCASE
RETURN lnUnitPrice
ENDFUNC && CalculateRoundedUnitPrice
```

Focusing on the new CalculateRoundedUnitPrice method, the developer sees that it actually performs two separate functions: it calculates the unit price based on the parameter values, and then rounds the result using one of three rounding methods. She also notices there's some duplication in the initial calculation of lnUnitPrice, where the first line of the CASE statement is essentially the same in all three legs. She therefore decides at least two more refactorings can be applied to further improve this code.

The calculation of the initial value of lnUnitPrice, currently duplicated in each leg of the CASE statement in the CalculateRoundedUnitPrice method, can be pulled out and replaced with a single calculation up in the CalculateUnitPrice method, where it's always rounded to three decimal positions. The remaining code in the CalculateRoundedUnitPrice method is now generic for rounding any numeric value in one of three ways, so the developer renames that method to be GetRoundedValue. The CalculateUnitPrice method can now pass the initial three-decimal value of lnUnitPrice to the new GetRoundedValue method to obtain the final price based on the desired rounding method.

Listing 11 – The initial calculation on lnUnitPrice has been moved up to the CalculateUnitPrice method, whose value is then passed to the generic GetRoundedValue method.

```
FUNCTION CalculateUnitPrice( tnCasePrice, tnCount, tnSize, tcCategory, ;
                             tcUnitType, tnUnitChrg, tnRoundingMethod)
LOCAL lnUnitPrice, lnDivisor
STORE 0.00 TO lnUnitPrice, lnCents, lnTenths, lnDivisor
lnDivisor = CalculateUnitPriceDivisor( tcCategory, tcUnitType, tnCount, tnSize)
lnUnitPrice = ROUND( ( ( tnCasePrice + tnUnitChrg) / lnDivisor), 3)
lnUnitPrice = GetRoundedValue( lnUnitPrice, tnRoundingMethod)
RETURN lnUnitPrice
ENDFUNC && CalculateUnitPrice

FUNCTION CalculateUnitPriceDivisor( tcCategory, tcUnitType, tnCount, tnSize)
LOCAL lnDivisor
DO CASE
   CASE tcCategory = "01"
      lnDivisor = tnSize
   CASE tcUnitType = "WEIGHT"
      lnDivisor = tnCount * tnSize
   OTHERWISE
      lnDivisor = tnCount
ENDCASE
RETURN lnDivisor
ENDFUNC && CalculateUnitPriceDivisor

FUNCTION GetRoundedValue( tnValue, tnRoundingMethod)
LOCAL lnTenths, lnRoundedValue
STORE 0.00 TO lnTenths, lnRoundedValue
DO CASE
   CASE tnRoundingMethod = 1  && truncate
      lnRoundedValue = tnValue - MOD( tnValue, .01)
   CASE tnRoundingMethod = 2  && normal rounding
      lnRoundedValue = ROUND( tnValue, 2)
   OTHERWISE && round up
      lnTenths = ( tnValue * 100) - INT( tnValue * 100)
      IF lnTenths > 0
         tnValue = tnValue + .005
      ENDIF
      lnRoundedValue = ROUND( tnValue, 2)
ENDCASE
RETURN lnRoundedValue
ENDFUNC && GetRoundedValue
```

As in the previous example, the GetRoundedValue method can now be easily moved to a different class if there is a more appropriate place for it.

Compare the refactored code in Listing 11 to the original code in Listing 7. Overall, the code is now much easier to read and understand. The CalculateUnitPrice method has a name that describes its purpose, and its parameters and local variables have names that describe the data they represent. The method also contains far less code than the original, because some of the calculations on which it depends have been extracted into separate methods that can also be referenced elsewhere in the application. Not only is the resulting code much easier to read and understand, it's also much easier to debug and change.

These two examples have focused on only a select few refactoring techniques. Much of refactoring involves not only making improvements to the code itself, but also making higher level decisions about how many classes there should be and which methods belong in which classes. If you use an application framework, many of these decisions have probably been made for you by the framework authors. The goal of refactoring is always the same, though – to improve the structure and readability of existing code in order to facilitate change.

Risks of refactoring

The primary risk of refactoring is the risk of breaking something that wasn't broken. As the old saying goes, "If it ain't broke, don't fix it!"

There are real risks involved in making any changes to working code, so the decision to refactor shouldn't be taken lightly. On the other hand, there are real benefits to be gained by refactoring working code, even code that's in pretty good shape to begin with.

Deciding if and when to refactor requires weighing the risks of breaking something against the potential benefits of improving it. The risk/reward ratio can be different in each situation, so it's a decision the developer or development team must make every time. Although there are guidelines, there is no hard and fast rule to determine when refactoring is required or even when it's beneficial.

Another risk of refactoring is the invasive creep of what I call the *"Where the heck did I put that??"* syndrome. This happens when the extract class, extract method, and other types of refactorings are overdone, resulting in a proliferation of new classes and subclasses to the extent that it becomes difficult to remember where a method is defined or where a piece of data resides. One goal of refactoring is to improve clarity and reveal meaning, but too much refactoring can obfuscate it. It's therefore important to know when to stop, or even when to just leave well enough alone in the first place.

Unit testing

The purpose of unit testing is to ensure nothing gets broken when changes are made to working code. It's the primary tool for mitigating the risks of refactoring, thereby shifting the balance toward the reward side of the risk/reward equation.

Doug Henning and others have written extensively about unit testing in VFP, so I'm not going to go into it in detail here. The most recent works are Doug's white paper from Southwest Fox 2013, *Unit Testing VFP Applications* [3], and his three-part series of the same name in the January, March, and May, 2014, issues of FoxRockX [4]. A video version of Doug's presentation is also available in the Online FoxPro Users Group (OFUG) archives [5].

By definition, refactoring involves changing the structure of code without changing its behavior. During and after refactoring, testing is needed to ensure that the refactored code exhibits the same behavior as the original code – in other words, to ensure that the refactoring didn't break anything. It goes without saying that refactoring should therefore be done only on code that passes all tests before modification.

You don't need a special-purpose tool to create unit tests, although a tool can be helpful if you're willing to invest the time to learn it. For VFP developers, the tool of choice is FoxUnit, which Doug covers in depth in the resources referenced above.

On the other hand, you can certainly write your own unit tests without a special-purpose tool. One way is to create a Test method for each method in a class and to incorporate the test method into the same class as the method itself. So for example, a class with a method named DoSomething might have a companion test method named DoSomething_Test.

One advantage of embedded test methods is that the test has access to all of the same runtime data available to the method being tested. This is useful if the method being tested takes parameters or accesses data whose values come from sources supplied at runtime. The #IFDEF preprocessor directive, along with an appropriate defined constant, can be used to include the test methods when compiling for development but exclude them when compiling for deployment.

In other situations it may make more sense to write the test code in external programs or class libraries. For that matter, although tests should certainly be written, they do not always need to be automated. Sometimes it's enough simply to write down a series of steps to be followed manually.

TDD purists would probably assert that unit testing per se is integral to the refactoring process. I take a little more relaxed view – testing is essential, but it's not necessary to create and run unit tests on every single method. Any type of test that ensures the refactored code exhibits the same behavior as the original code will do. If the code modifies data in a database, as most VFP applications do, comparing "before" and "after" snapshots of the affected data tables can be an effective way to ensure the refactored code produces the same result as the original for each set of inputs.

Tools for refactoring in VFP

Refactoring tools fall into two basic categories. Code inspection tools, sometimes called refactoring browsers, are designed to help locate and identify sections of code that may be candidates for refactoring, while automated refactoring tools actually change the code by applying one or more refactorings. Some products, such as ReSharper for Visual Studio, do all of this and more.

Visual FoxPro developers may not have access to all the tools available to developers working in other environments, but there are still several tools to help VFP developers with refactoring.

Basic search and replace

Many refactorings require the developer to search through code to find places where the refactored code is referenced, and then to make the appropriate changes. Common examples are when a variable or a method is renamed or when a method's signature is changed. In those situations, all of the existing references to those variables or methods must be located and changed throughout the entire program, class, or application.

VFP's native *Find* dialog is the basic tool when the scope of the search is a single program file, form, or class library. Along with the companion *Replace* dialog, this tool is often all that's needed to complete a refactoring when the scope of its effects is limited.

When a refactoring potentially affects several classes and program files throughout an entire VFP project, the next step up is the built-in *Code References* tool . This tool enables searching for and optionally replacing character strings within an entire project or folder. Like the Find and Replace dialogs, the Code References tool can search and replace within VFP binary files.

Matt Slay's *GoFish 4* is an enhanced search tool for VFP. Like the Code References tool, it can search within VFP's binary files as well as text files, but it features many more filters and options than the Code References tool. GoFish 4 is available for download on VFPX.

There are several third-party search and replace tools outside of the VFP-specific ones. *Search and Replace* and its successor, *Replace Studio Professional*, both from Funduc Software, are known for their speed. Both products enable you to specify a search string, an optional replacement string, the path to be searched, and a filter for the desired file extensions in that path. Although not suited for working with VFP's binary files, they are nonetheless useful for quickly searching for and optionally replacing character strings in folders containing VFP text files such as .prg and .h. They're also extremely well suited for working with Web applications involving HTML, CSS, and other text-based files.

Advanced navigation

Advanced navigation tools make it easy to jump to the places in an application where a variable or method is defined or referenced. While most VFP developers are probably familiar with the Code References tool and how to launch it from the Tools menu, it may be less well known that it can be launched directly from the code editor via the context menu. The code editor's context menu also has a companion View Definition item. Both of these work best within the scope of an actively open project in the VFP IDE.

To illustrate the use of these two navigation aids, suppose there is a project with three code files, main.prg, clsOne.prg, and clsTwo.prg. Both clsOne.prg and clsTwo.prg have a method named DoSomething. The main program instantiates both classes and calls their DoSomething method. Listing 12 shows these three program files.

Listing 12. The sample project has three files.

```
* Main.prg
LOCAL loOne as Object, ;
      loTwo as Object
loOne = NEWOBJECT( “clsOne”, “clsOne.prg”)
loOne.DoSomething()
loTwo = NEWOBJECT( “clsTwo”, “clsTwo.prg”)
loTwo.DoSomething()

* clsOne.prg
DEFINE CLASS clsOne as custom
cName = “One”
PROCEDURE DoSomething()
*   something
ENDPROC
ENDDEFINE

* clsTwo.prg
DEFINE CLASS clsTwo as custom
cName = “Two”
PROCEDURE DoSomething()
*   something
ENDPROC
ENDDEFINE
```

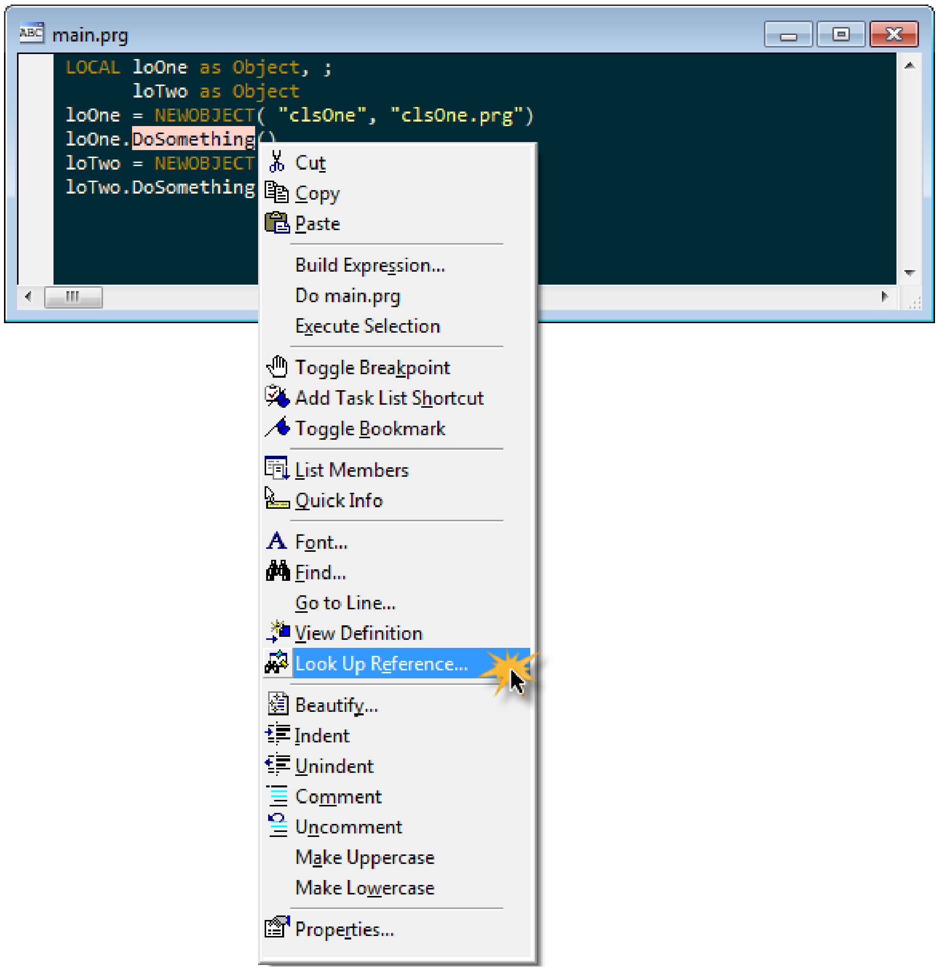
In this example it’s easy to see where all the references and definitions are because all the code can be seen at once, but think about it in terms of a real application comprising dozens of files with hundreds of lines each.

Look Up Reference

If the developer working on main.prg wants to locate and jump one of the places where the DoSomething method is defined in this project, the conventional way is to:

- open the Code References tool from VFP’s Tools menu
- click the Search button to bring up the Look Up Reference dialog
- type in the string to search for
- click that dialog’s Search button to generate a list of the references
- locate the desired reference in the list
- double-click the reference to open its location in the code editor

A quicker way is to accomplish the first three steps is to select the desired text from a line in the code editor, right-click to get the context menu, and then select Look Up Reference, as shown in **Figure 2**. This opens the Look Up Reference dialog with the highlighted text automatically inserted as the search string.



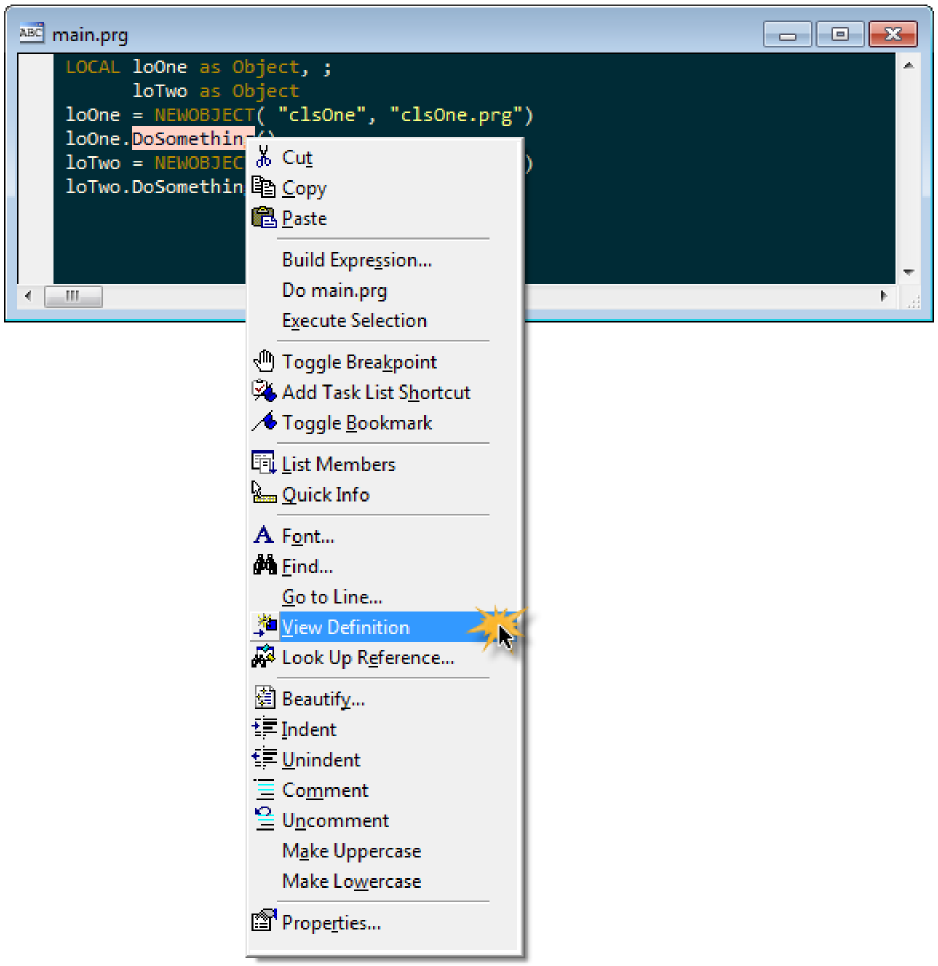
```
LOCAL loOne as Object, ;
      loTwo as Object
loOne = NEWOBJECT( "clsOne", "clsOne.prg")
loOne.DoSomething()
loTwo = NEWOBJECT
loTwo.DoSomething
```

Figure 2. The Code Reference tool can be launched from the Look Up Reference item on the context menu.

From there, the remaining steps are the same as if the Code References tool had been launched in the conventional manner.

View Definition

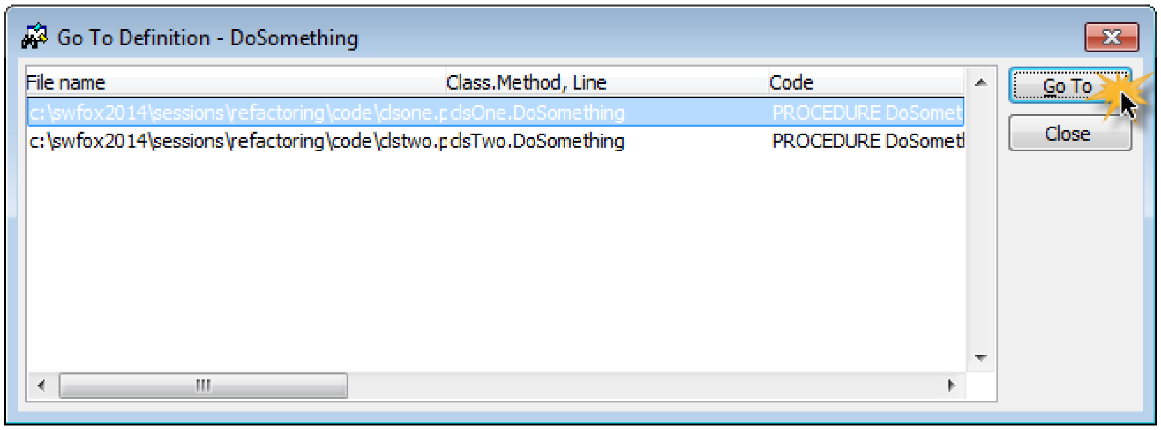
Sometimes the developer may want to jump directly to the definition of a code element without having to wade through all the references to it. VFP provides a quick way to do this, too. With the desired text selected in the editor, choose View Definition from the context menu as show in **Figure 3**.



```
LOCAL loOne as Object, ;
      loTwo as Object
loOne = NEWOBJECT( "clsOne", "clsOne.prg")
loOne.DoSomethin
loTwo = NEWOBJEC
loTwo.DoSomethin
```

Figure 3. Use View Definition on the context menu to jump to where an element is defined

If the selected element is defined in only one place, VFP opens the appropriate file and selects that location for you. One of the cool features of this tool comes into play when the selected element is defined in more than one place, as is true of the DoSomething method in this example. When an element is defined in more than one place, VFP brings up the Go To Definition dialog shown in **Figure 4**.

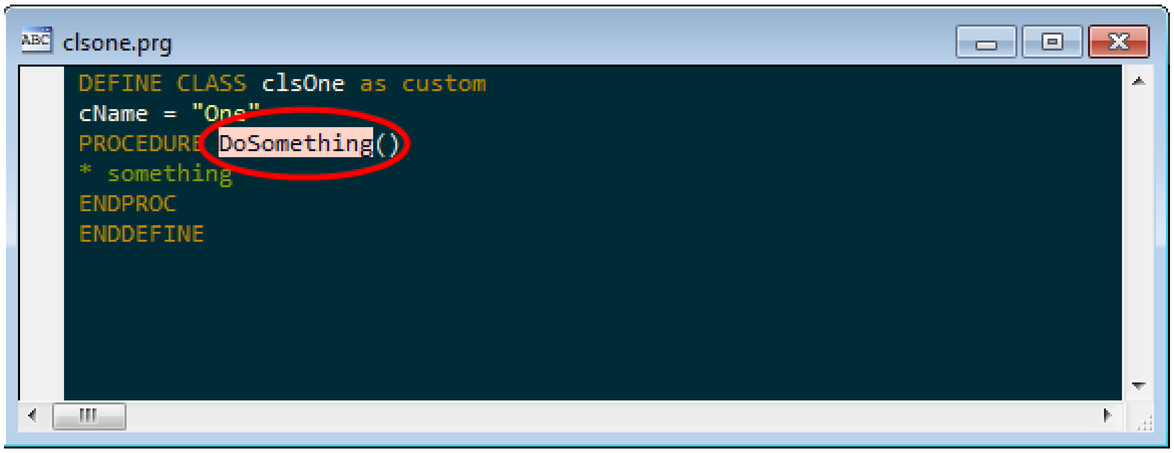


File name	Class.Method, Line	Code
c:\swfox2014\sessions\refactoring\code\clsone.p	clsOne.DoSomething	PROCEDURE DoSomet
c:\swfox2014\sessions\refactoring\code\clstwo.p	clsTwo.DoSomething	PROCEDURE DoSomet

Figure 4. The Go To Definition dialog lists the places where the selected element is defined.

The Go To Definition dialog lists the places where the selected element is defined. You can then jump directly to the location of the desired definition by selecting it in the list and clicking the Go To button.

In Figure 4, the definition of the DoSomething method in clsOne.prg is selected. When the Go To button is clicked, VFP therefore opens clsOne.prg in the code editor and highlights the location where its DoSomething method is defined, as shown in **Figure 5**. If the definition in clsTwo.prg had been selected, then clsTwo.prg would have been opened.



```
DEFINE CLASS clsOne as custom
cName = "One"
PROCEDURE DoSomething()
* something
ENDPROC
ENDDEFINE
```

Figure 5. After clicking the Go To button, the file containing the selected definition is opened and the definition of the selected element is automatically highlighted.

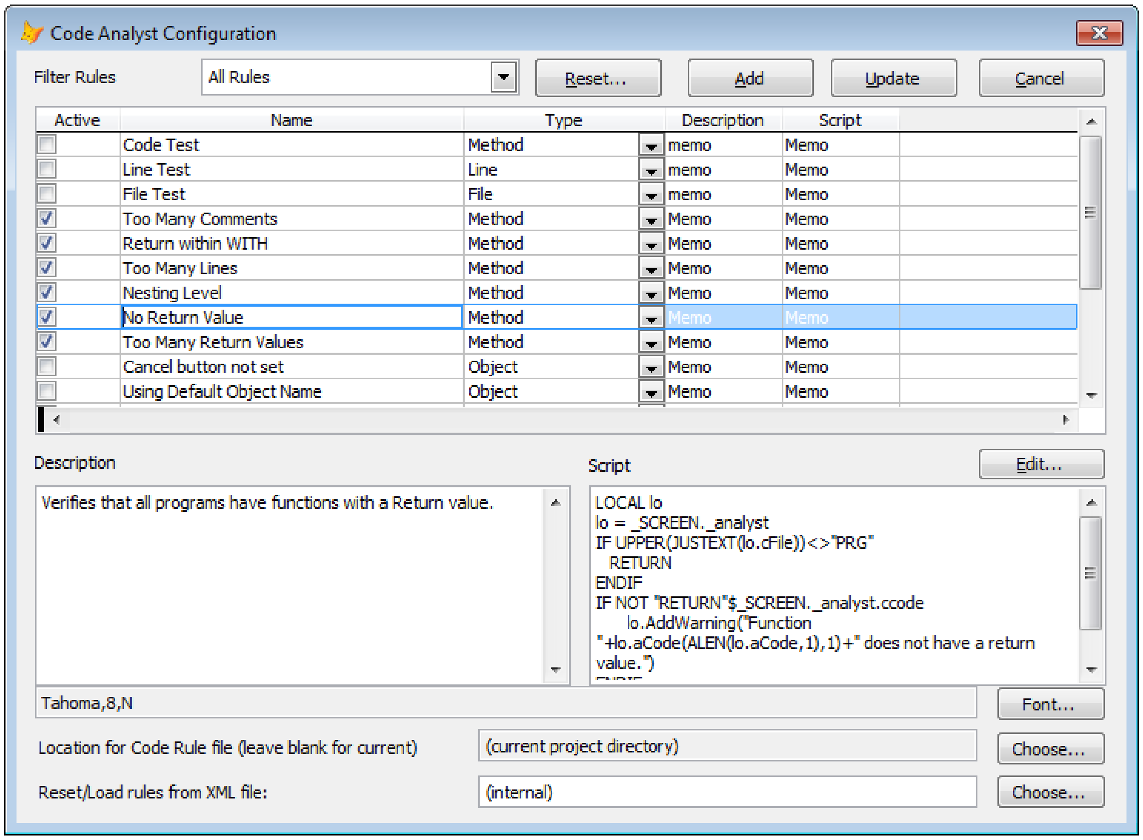
The Go To Definition dialog is non-modal and remains open even after clicking the Go To button. This enables you to open multiple files and view multiple definitions without having to start over from View Definition on the context menu.

The Look Up Reference and View Definition shortcuts on the VFP code editor's context menu are useful tools when refactoring because they enable you to quickly and easily find and jump to places in the code where changes may need to be made.

Code inspection

Code inspection tools enable developers to browse or search through code and identify places that could benefit from refactoring. For Visual FoxPro developers, the only tool I'm aware of is the *Code Analyst* tool on VFPX. This project is maintained by Andrew MacNeill. It's currently listed as being in release candidate status with the most recent build being January, 2013.

The tool comes with an extensible set of rules defining conditions to be flagged as potential problems or bad code smells, and which might be therefore be candidates for refactoring. These rules can be viewed, selected, and edited in the Code Analyst Configuration dialog, shown in **Figure 6**.



Active	Name	Type	Description	Script
☐	Code Test	Method	memo	Memo
☐	Line Test	Line	memo	Memo
☐	File Test	File	memo	Memo
✓	Too Many Comments	Method	Memo	Memo
✓	Return within WITH	Method	Memo	Memo
✓	Too Many Lines	Method	Memo	Memo
✓	Nesting Level	Method	Memo	Memo
✓	No Return Value	Method	Memo	Memo
✓	Too Many Return Values	Method	Memo	Memo
☐	Cancel button not set	Object	Memo	Memo
☐	Using Default Object Name	Object	Memo	Memo

Description

Verifies that all programs have functions with a Return value.

Script

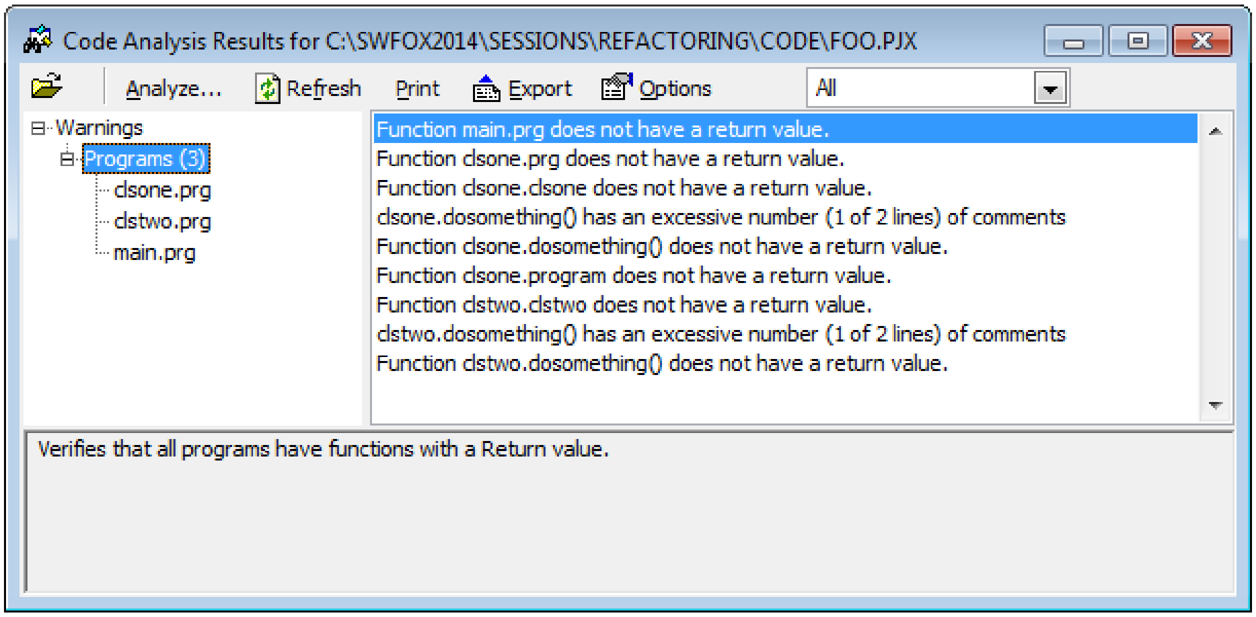
```
LOCAL lo
lo = _SCREEN._analyst
IF UPPER(JUSTEXT(lo.cFile))<>"PRG"
   RETURN
ENDIF
IF NOT "RETURN"$_SCREEN._analyst.ccode
      lo.AddWarning("Function
"+lo.aCode(ALEN(lo.aCode,1),1)+" does not have a return
value.")
```

Edit...
Tahoma,8,N
Font...
Location for Code Rule file (leave blank for current)
(current project directory)
Choose...
Reset/Load rules from XML file:
(internal)
Choose...

Figure 6. The Code Analyst tool comes with an extensible set of rules, which can be seen in the Configuration dialog.

The lower portion of the Configuration dialog shows the description of the selected rule and the code used to implement it. Inspecting the code is a good way to learn how the tool works.

When Code Analyst is launched, it prompts for the name of the VFP entity to be analyzed, which can be a single program file, form, or visual class library, or an entire project. The results obtained by running Code Analyst on the little project in Listing 12 are shown in **Figure 7**.



Function main.prg does not have a return value.
Function dsone.prg does not have a return value.
Function dsone.dsone does not have a return value.
dsone.dosomething() has an excessive number (1 of 2 lines) of comments
Function dsone.dosomething() does not have a return value.
Function dsone.program does not have a return value.
Function dstwo.dstwo does not have a return value.
dstwo.dosomething() has an excessive number (1 of 2 lines) of comments
Function dstwo.dosomething() does not have a return value.

Verifies that all programs have functions with a Return value.

Figure 7. The Code Analyst results dialog features a tree view for easy navigation and inspection of the results for each individual file.

I only recently began experimenting with Code Analyst, but I've used it enough to be intrigued. For my own purposes I can already see that some of the rules would need to be modified or ignored while new ones would probably need to be added, but the fact that the rules are editable and extensible opens up a lot of possibilities.

Automated Refactoring

Automated refactoring tools go a step beyond code inspection tools and actually make changes to your code. Their attraction is in their convenience, freeing the developer from such tedious tasks as searching for and replacing all occurrences of a renamed variable or method within a project, or having to manually copy and paste the code when extracting a method or a class. Automated refactoring tools don't do anything that can't be done manually, though, so although convenient they're not essential.

Visual Studio developers have automated refactoring tools like ReSharper, but to my knowledge there is no such tool for Visual FoxPro. I don't see this a real disadvantage, though. As Opdyke points out [2], there is always a human element to refactoring, so it can never be fully automated.

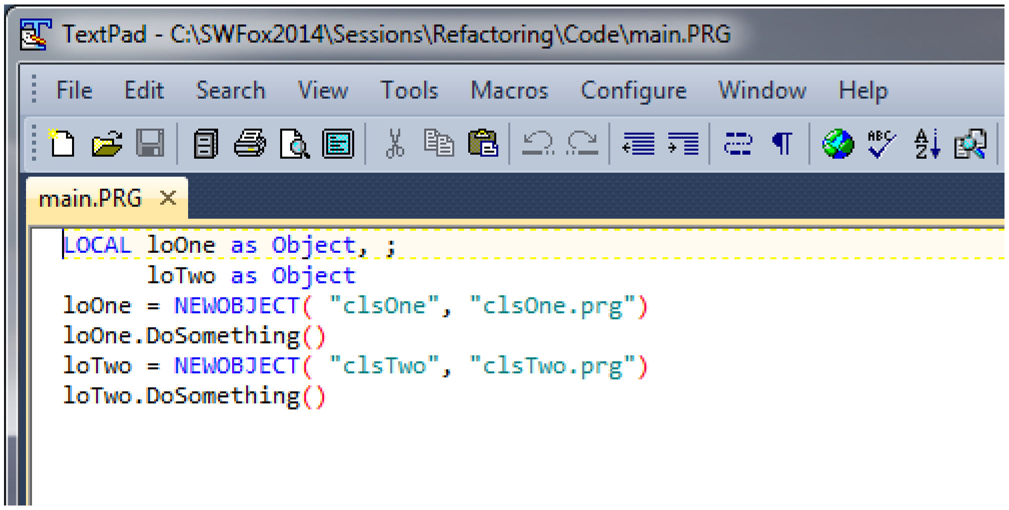
Unit testing tools

FoxUnit is the de facto unit testing tool for VFP developers. Originally from VisionPace, FoxUnit is now actively maintained on VFPX by Eric Selje. For more information about this tool please refer to the VFPX website and the references mentioned in the Unit Testing section earlier in this paper.

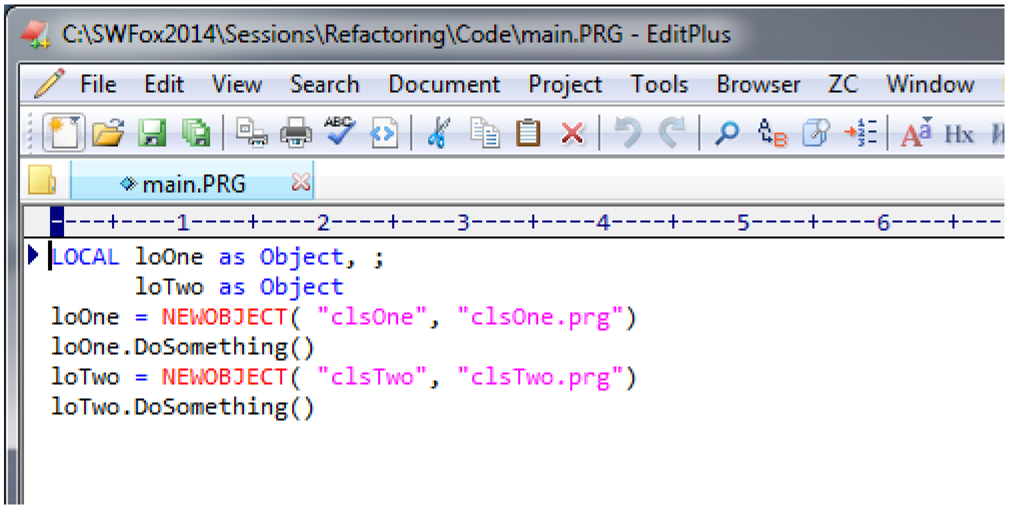
Other tools

During refactoring I often make use of external text editors that can recognize VFP syntax. My two favorites are *TextPad* and *EditPlus*. Both are generic text editors with free VFP syntax add-ons supplied by the community. Neither of these editors can provide code completion or other IntelliSense types of assistance, but when the appropriate VFP language add-on is installed they do provide VFP syntax coloring as an aid to readability.

Figure 8 shows how the little main.prg program file from an earlier example looks in each of these two editors. The default syntax coloring is different but useful in each.


TextPad - C:\SWFox2014\Sessions\Refactoring\Code\main.PRG
File Edit Search View Tools Macros Configure Window Help
main.PRG

```
LOCAL loOne as Object, ;
      loTwo as Object
loOne = NEWOBJECT( "clsOne", "clsOne.prg")
loOne.DoSomething()
loTwo = NEWOBJECT( "clsTwo", "clsTwo.prg")
loTwo.DoSomething()
```

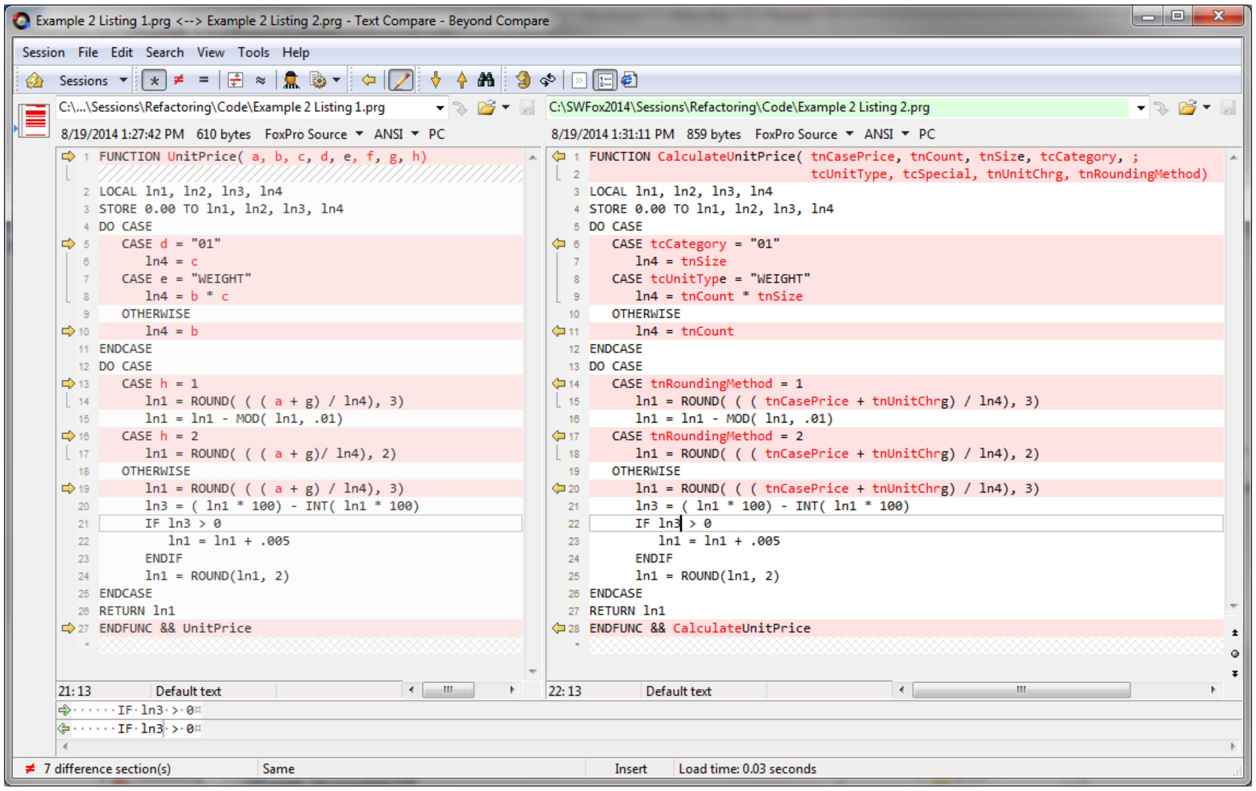

C:\SWFox2014\Sessions\Refactoring\Code\main.PRG - EditPlus
File Edit View Search Document Project Tools Browser ZC Window
main.PRG

```
LOCAL loOne as Object, ;
      loTwo as Object
loOne = NEWOBJECT( "clsOne", "clsOne.prg")
loOne.DoSomething()
loTwo = NEWOBJECT( "clsTwo", "clsTwo.prg")
loTwo.DoSomething()
```

Figure 8. With the appropriate VFP language add-ons installed, both TextPad and EditPlus provide syntax coloring for VFP files.

I'm also a big fan of the venerable *Beyond Compare* from Scooter Software. In combination with the external editors, these three tools make it easy to create and compare snapshots of code segments before and after refactoring. This visual inspection helps confirm that only the intended changes have been made. This is not a substitute for testing, but it is useful as one additional way to be sure everything looks good at various stages in the refactoring process.

By way of example, consider the refactorings applied to the UnitPrice method in the second VFP example earlier in this paper. To compare the code before and after refactoring, I first copied it from the VFP editor in its original form, pasted it into a blank document in one of the external text editors, and saved it as a VFP .prg file in a temporary folder. I then did the same thing with the code after the first refactoring. Finally, I used Beyond Compare to compare the two temporary files side by side so I could observe the differences, as shown in **Figure 9**. This enabled me to easily confirm that I hadn't made any unintended changes to the code on the right.


Example 2 Listing 1.prg <--> Example 2 Listing 2.prg - Text Compare - Beyond Compare

C:\...\Sessions\Refactoring\Code\Example 2 Listing 1.prg
8/19/2014 1:27:42 PM   610 bytes   FoxPro Source   ANSI   PC

```
FUNCTION UnitPrice( a, b, c, d, e, f, g, h)
LOCAL ln1, ln2, ln3, ln4
STORE 0.00 TO ln1, ln2, ln3, ln4
DO CASE
   CASE d = "01"
      ln4 = c
   CASE e = "WEIGHT"
      ln4 = b * c
   OTHERWISE
      ln4 = b
ENDCASE
DO CASE
   CASE h = 1
      ln1 = ROUND( ( ( a + g) / ln4), 3)
      ln1 = ln1 - MOD( ln1, .01)
   CASE h = 2
      ln1 = ROUND( ( ( a + g)/ ln4), 2)
   OTHERWISE
      ln1 = ROUND( ( ( a + g) / ln4), 3)
      ln3 = ( ln1 * 100) - INT( ln1 * 100)
      IF ln3 > 0
         ln1 = ln1 + .005
      ENDIF
      ln1 = ROUND(ln1, 2)
ENDCASE
RETURN ln1
ENDFUNC && UnitPrice
```

21: 13   Default text

C:\SWFox2014\Sessions\Refactoring\Code\Example 2 Listing 2.prg
8/19/2014 1:31:11 PM   859 bytes   FoxPro Source   ANSI   PC

```
FUNCTION CalculateUnitPrice( tnCasePrice, tnCount, tnSize, tcCategory, ;
                             tcUnitType, tcSpecial, tnUnitChrg, tnRoundingMethod)
LOCAL ln1, ln2, ln3, ln4
STORE 0.00 TO ln1, ln2, ln3, ln4
DO CASE
   CASE tcCategory = "01"
      ln4 = tnSize
   CASE tcUnitType = "WEIGHT"
      ln4 = tnCount * tnSize
   OTHERWISE
      ln4 = tnCount
ENDCASE
DO CASE
   CASE tnRoundingMethod = 1
      ln1 = ROUND( ( ( tnCasePrice + tnUnitChrg) / ln4), 3)
      ln1 = ln1 - MOD( ln1, .01)
   CASE tnRoundingMethod = 2
      ln1 = ROUND( ( ( tnCasePrice + tnUnitChrg) / ln4), 2)
   OTHERWISE
      ln1 = ROUND( ( ( tnCasePrice + tnUnitChrg) / ln4), 3)
      ln3 = ( ln1 * 100) - INT( ln1 * 100)
      IF ln3 > 0
         ln1 = ln1 + .005
      ENDIF
      ln1 = ROUND(ln1, 2)
ENDCASE
RETURN ln1
ENDFUNC && CalculateUnitPrice
```

22: 13   Default text

```
······IF·ln3·>·0¤
······IF·ln3·>·0¤
```

7 difference section(s)   Same   Insert   Load time: 0.03 seconds

Figure 9. Beyond Compare enables you to compare snapshots of a piece of code before refactoring (left side) and after refactoring (right side) for visual confirmation that the changes are as expected.

Although I use both editors for lots of reasons, EditPlus has one slight advantage when working with VFP files. In some cases, you may want to use the editor as a temporary holding area where you can view a chunk of code with no intention of ever saving it as a file. In EditPlus, you can specify a file type (VFP or other) when you first create a new document, so you don't have to save the file to get syntax coloring. TextPad doesn't apply syntax coloring until you save the file.

Links for both of these editors, along with those for other useful tools, can be found in the Tools section at the end of this paper.

Summary

Refactoring is the process of improving the internal structure of code without changing its external behavior. Without refactoring, the structure of an application's code can decay over time as changes are made and new features added in ways not planned for nor well supported by the original structure. Sometimes the structure of the code is not very good to begin with. Either way, it's difficult to make changes or add new features, and developers end up spending more time working around problems than doing productive work. Refactoring is a solution to these types of problems.

Biography

Rick Borup is owner and president of Information Technology Associates, LLC, a professional software development, computer services, and information systems consulting firm he founded in 1993. Rick earned BS and MBA degrees from the University of Illinois and spent several years developing software applications for mainframe computers before turning to PC database development tools in the late 1980s. He began working with FoxPro in 1991, and has worked full time in FoxPro and Visual FoxPro since 1993. He is a co-author of the books *Deploying Visual FoxPro Solutions* and *Visual FoxPro Best Practices For The Next Ten Years*. He has written articles for FoxTalk and FoxPro Advisor, and is a frequent speaker at Visual FoxPro conferences and user groups. Rick is a Microsoft Certified Solution Developer (MCSD) and a Microsoft Certified Professional (MCP) in Visual FoxPro.

Bibliography

[1] Martin Fowler, et al., *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, Inc., 1999

[2] William F. Opdyke, *Refactoring Object-Oriented Frameworks*, Thesis (Ph. D.), University of Illinois at Urbana-Champaign, 1992

[3] Doug Hennig, *Unit Testing VFP Applications*, white paper for Southwest Fox 2013, Gilbert, AZ, October, 2013

[4] Doug Hennig, *Unit Testing VFP Applications*, three-part series in FoxRockX, January, March, and May, 2014 (subscription required)
http://portaladmin.dfpug.de/dFPUG/Dokumente/FoxRockX/PDFIssues/

[5] Doug Hennig, *Unit Testing VFP Applications*, video presentation to the Online FoxPro Users Group (OFUG), Dec. 17, 2013
https://www.youtube.com/watch?v=J5PH1tKPYpI&feature=youtu.be

[6] Ron Jeffries, *Refactoring – Not on the backlog!*, XProgramming.com, July 29, 2014
http://xprogramming.com/articles/refactoring-not-on-the-backlog/

[7] Jeff Atwood, *Don't be afraid to break stuff!*, Coding Horror blog, Nov. 4, 2004
http://blog.codinghorror.com/dont-be-afraid-to-break-stuff/

Other References

Nancy Folsom, *Best Practices for Refactoring*, white paper for Great Lakes Great Database Workshop 2006, Milwaukee, WI, April, 2006, and also as Chapter 7 in *Visual FoxPro Best Practices for the Next Ten Years*, Hentzenwerke Publishing, 2006

Andrew MacNeill, *Visual FoxPro Refactoring Redux*
http://www.aksel.com/whitepapers/refactoring.htm

FoxPro Wiki – Refactoring
http://fox.wikis.com/wc.dll?Wiki~Refactoring

Nancy Folsom, *Refactoring: VFP form calls external PRG, uses Publics for shared data,*
http://nancyfolsom.wordpress.com/2011/04/15/refactoring-vfp-form-calls-external-prg-uses-publics-for-shared-data/

Andrew MacNeill, *Profiling and Refactoring,* Southwest Fox 2008

Doug Hennig, *FoxUnit is Cool!*
http://doughennig.blogspot.com/2006/05/foxunit-is-cool.html

H. Alan Stevens, *Test-Driven Development,* Southwest Fox 2007

Ted Roche, *Unit Testing in VFP*
https://speakerdeck.com/tedroche/unit-testing-in-visual-foxpro-2001
(published Nov. 9, 2011)

Andrew MacNeill, *Using FoxUnit for Test-Driven Development in VFP*
http://www.aksel.com/whitepapers/FoxUnit.htm

Tools

FoxUnit
http://vfpx.codeplex.com/wikipage?title=FoxUnit

VFPX – Code Analyst
vfpx.codeplex.com/wikipage?title=Code Analyst

GoFish 4
vfpx.codeplex.com/wikipage?title=GoFish

Search and Replace, and *Replace Studio Professional*
www.funduc.com

Beyond Compare
www.scootersoftware.com

EditPlus
www.editplus.com

TextPad
www.textpad.com

Appendix A

This appendix is an alphabetical list of selected refactorings I've used or referenced in this paper, quoted directly from Fowler's *Refactoring* book. The numbers in parentheses are the page in *Refactoring* on which each of them is defined and explained.

Consolidate Duplicate Conditional Fragments (243)

The same fragment of code is in all branches of a conditional expression.

Move it outside of the expression.

Convert Procedural Design to Objects (368)

You have code written in a procedural style.

Turn the data records into objects, break up the behavior, and move the behavior to the objects.

Decompose Conditional (238)

You have a complicated conditional (if – then – else) statement.

Extract methods from the condition, [the] then part, and [the] else part.

Extract Class (149)

You have one class doing work that should be done by two.

Create a new class and move the relevant fields and methods from the old class into the new class

Extract Method (110)

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

Move Method (142)

A method is, or will be, using or used by more features of another class than the class on which it is defined.

Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.

Remove Parameter (277)

A parameter is no longer used by the method body.

Remove it.

Rename Method (273)

The name of a method does not reveal its purpose.

Change the name of the method.

Replace Parameter with Method (292)

An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.

Remove the parameter and let the receiver invoke the method.

Separate Domain from Presentation (370)

You have GUI classes that contain domain logic.

Separate the domain logic into separate domain classes.