Part I: Introduction to Refactoring

In this introductory part, you are going to see what refactoring is in general terms, why it is important, what benefits refactoring brings to the development process, and how it can be even more relevant to Visual Basic programmers than to programmers in some other languages. You are also going to see a small demonstration of the refactoring process at work, explore the tools relevant to refactoring, and, finally, take a look at a sample application I will use throughout this book to illustrate refactorings and the refactoring process as it is applied.

Refactoring: What's All the Fuss About?

Take a look at any major integrated development environment (IDE) today and you are bound to discover "refactoring" options somewhere at the tip of your fingers. And if you are following developments in the programming community, you have surely come across a number of articles and books on the subject. For some, it is the most important development in the way they code since the inception of design patterns.

Unlike some other trends, refactoring is being embraced and spread eagerly by programmers and coders themselves because it helps them do their work better and be more productive. Without a doubt, applying refactoring has become an important part of programmers' day-to-day labor no matter the tools, programming language, or type of program being developed. Visual Basic is a part of this: at this moment, the same wave of interest for refactoring in the programming community in general is happening inside the Visual Basic community.

In this introduction,

- ❑ I start out by taking a look at what refactoring is and why it is important and then discuss a few of the benefits that refactoring delivers.
- ❑ I also address some of the most common misconceptions about refactoring.
- ❑ In the second part of this chapter, I want you to take a look at the specifics of Visual Basic as a programming language and how refactoring can be even more relevant for Visual Basic programmers because of some historic issues related to Visual Basic.

I'll start with some background on refactoring in general.

A Quick Refactoring Overview

When approaching some programming task, you have a number of ways in which you can go about it. You start off with one idea, but as you go along and get into more detail, you inevitably question your work along these lines: “Should I place this method in this class or maybe in this other class? Do I need a class to represent this data as a type or am I well off using the primitive? Should I break this class into more than one? Is there an inheritance relationship between these two classes or should I just use composition?” And if you share your thoughts with some of your peers, you are bound to hear even more options for designing your system. However, once you commit yourself to one approach, it may seem very costly to change these initial decisions later on. Refactoring teaches you how to efficiently modify your code in such a way that the impact of those modifications is kept at a minimum. It also helps you think about the design as something that can be dealt with at any stage of the project, not at all cast in stone by initial decisions. Design, in fact, can be treated in a very flexible way.

Definition: ***Refactoring*** **is a set of techniques used to identify the design flows and to modify the internal structure of code in order to improve the design without changing code’s visible behavior.**

All design decisions are the result of your knowledge, experience, and creativity. However, programming is a vast playfield, and it’s easy to get tangled in contradictory arguments. In VB .NET you are, first and foremost, guided by object-oriented principles and rules. Unfortunately, very often it is not so clear how these rules work out in practice. Refactoring teaches you some simple heuristics that can help improve your design by inspecting some of the visible characteristics of your code. These guidelines that refactoring provides will set you on the right path in improving the design of your code.

The Refactoring Process

Refactoring is an important programming practice and has been around for some time. Pioneered by the Smalltalk community, it has been applied in a great number of programming languages, and it has taken its place in many programmers’ bags of tricks. It will help you write your code in such a way that you will not dread code revision. Being a programmer myself, I know this is no small feat!

So, how do you perform refactoring? The refactoring process is fairly simple and consists of three basic steps:

1. Identify code smells.

   You’ll see what *code smell* means very soon, but, in short, this first step is concerned with identifying possible pitfalls in your code, and code smells are very helpful in identifying those pitfalls.

2. Apply the appropriate refactoring.

   This second step is dedicated to changing the structure of your code by means of refactoring transformations. These transformations can often be automated and performed by a refactoring tool.

3. Execute unit tests.

   This third step helps you rectify the state of your code after the transformations. Refactoring is not meant to change any behavior of your code observable from the “outside.” This step generally consists of executing appropriate unit tests that will prove the behavior of your code didn’t change after performing refactoring.

You might have noticed the word *design* used in the refactoring definition earlier in the chapter. This is a broad term and can take on very different meanings depending on your background, programming style, and knowledge. Design in this sense simply means that refactoring builds upon object-oriented theory with the addition of some very simple heuristics dedicated to identifying shortcomings and weak spots in your code. These antipatterns are generally referred to as *code smells* and a great part of refactoring can be seen simply as an attempt to eliminate code smells.

Definition: ***Code smell*** **is a sensation you develop that tells you that there might be a flaw in your code.**

The code smell can be something as simple as a very large method, a very large class, or a class consisting only of data and with no behavior. I'll dedicate a lot of time to code smells in the book, because improving your sense of code smell can be very important in a successful refactoring process.

The aim of refactoring is to improve the design of your code. You generally do this by applying modifications to your code. The refactoring methodology and its techniques help you in this task by making it easier to perform and even automate such modifications.

A Look at the Software Situation

As software developers, your success depends on being able to fulfill different types of expectations. You have to keep in mind many different aspects of your development work; here are just a few of the concerns:

- ❑ Very often you will hear that the most important one is satisfying user requirements, generally meaning that you should create software that does what the client paid for.
- ❑ You also need to guarantee the quality of your product. You strive to reduce defects and to release a program that has the minimum number of bugs.
- ❑ You have to think about usability, making programs that are easy to understand and exploit.
- ❑ You tend to be especially concerned about performance, always inventing new ways to minimize memory usage and the number of cycles needed in order to solve some problem.
- ❑ You need to do all of this in a timely manner, so you are always looking for ways to augment productivity.

These issues cause us to focus, and rightly so, on the final product (also known as the *binary*) and how it will behave for the final user. However, in the process of producing the binary, you actually work with *source code*. You create classes, add properties and methods, organize them into the namespaces, write logic using loops and conditions, and so on. This source code, at a click of a button, is then transformed, compiled, or built into a deliverable, a component, an executable, or something similar. There is an inevitable gap between the artifacts you work on — the *source* — and the artifacts you are producing — the *binary*.

In a way, this gap is awkward and not so common in the other areas of human activity. Take a look at stonemasonry, for example. While the mason chips away pieces of stone and polishes the edges, he or she can see the desired result slowly appearing under the effort. With software, the process is not at all as direct. You write source code that is then transformed into the desired piece of software. Even with the visual tools, which largely bridge this gap between source and binary, all you do in the end is create the source that is later on processed and turned into a compiled unit. Imagine a cook that can only write down a recipe and try the cooked meal, but is not allowed to handle the ingredients or taste the meal while it is being prepared.

What's more, there are many ways to write the source that will produce the same resulting binary. This can easily lead you to forget or sacrifice some qualities inherent to the source code itself, because the source code can be considered just a secondary artifact. While these qualities are not directly transformed to a final product, they have an immense impact on the whole process of creation and maintenance.

This leads to the following question: Can we distinguish between well written and poorly written code, even if the final result is the same? In the following sections, I'll explore this question, and you'll see how refactoring can clarify doubts you might have.

Refactoring Encourages Solid Design

No matter your previous programming experience, I am certain you will agree that you can indeed distinguish between good and bad code.

Assessing code may begin on a visual level. Even with a simple glance you can see if the code is indented and formatted so it is pleasing to view, if the agreed naming conventions are used, and so on.

At a less superficial level, you start to analyze code according to principles and techniques of software design. In Visual Basic, you follow the object-oriented software paradigm. You look into how well classes are structured and encapsulated, what their responsibilities are, and how they collaborate. You use language building blocks like classes and interfaces; and features like encapsulation, inheritance, and polymorphism in building a cohesive structure that describes the problem domain well. In a certain way you build your own ad-hoc language on top of a common language that will communicate your intentions and design decisions.

There are a number of sophisticated principles you need to follow in order to achieve a solid design. When you create software that is reusable, extendible, and reliable, and that communicates its purpose well, you can say you have reached your goal of creating well-designed code.

Refactoring gives you a number of recipes to ensure that your software conforms to the principles of well-designed code. And when you stray from your path, it helps you reorganize and impose the best design decisions with ease.

Refactoring Accommodates Change

Popular software design techniques like object-oriented analysis and design, UML diagramming, use-case analysis, and others often overlook one very important aspect of the software creation process: constant change. From the first moment it is conceived, software is in continuous flux. Every so often, requirements will change even before the first release, new features will be added, defects corrected, and even some planned design decisions, when confronted with real-world demands, overruled. Software construction is a very complex activity, and it is futile attempting to come up with a perfect solution up front. Even if some more sophisticated techniques like modeling are used, you still come short of thinking about every detail and every possible scenario. It is this state of flux that is often the biggest challenge in the process of making software. You have no choice but to be ready to adapt, count on change, and react readily when it happens. If you are not ready to react, the design decisions you made are soon obscured, and the dangerous malaise of rotting design settles in.

Refactoring is a relatively simple way to prepare for change, implement change, and control the adverse effects these changes can have on your design.

Refactoring Prevents Design Rot

Software is definitely one of the more ephemeral human creations. Driven by new advances and technologies, software creations are soon replaced with more modern or advanced versions. Even so, during its lifetime software will journey through a number of reincarnations. It is constantly modified and updated, new features are added and old ones removed, defects resolved and adaptations performed. It is quite common that more than one person will put their hands on the same piece of software, each with his or her own style and preferences. Rarely will it be the same team of people that will see the software from the beginning to the end.

Go back for a moment to the stonemason example. Now imagine that there is more than one person working on the same stone, that these people can change during the collaboration, and that the original plan is often itself changed with new shapes added or removed and different materials used. That may be a task for somebody of Michelangelo's stature, but definitely not for the ordinary craftsman.

No wonder then that initial ideas soon are forgotten, thought-out structure superseded by new solutions, and original design diluted. The initial intentions become less pronounced and the metaphors more difficult to comprehend, and the source is closer and closer to a meaningless cluster of symbols that still, but a lot less reliably, performs the intended function. This ailment steals in quietly, step by step, often unnoticed, and you end up with source that is difficult to maintain, modify, or upgrade.

What I've just described are the symptoms of rotting design, something that can occur even before the first release lives to see the light of a day. Refactoring helps you prevent design rot.

So, as you have moved along in this brief survey of the software landscape, I've pointed out several challenges that developers face and how refactoring can help. Next, I want to discuss refactoring in more detail.

The Refactoring Process: A Closer Look

I just discussed a few key areas of software development that can often lead to poor code. You need to stand guard for the quality of your code constantly. In effect, you need to have the design qualities of your code in mind at all times.

While this sounds sensible, thinking continuously about design and code quality can often be costly and quite complicated. The refactoring methodology and its techniques help you in this task by making it easier to perform and even automate modifications that will keep the design active.

In this section I'm going to take a look at the refactoring activities you would typically complete during a software development cycle.

Using Code Smells

As a first step in your refactoring activity, you take a look at the code in order to assess its design qualities. Refactoring teaches you a set of relatively simple heuristics called code smells that can help you with this task, along with well-known notions and principles of object-oriented design. Programming, being complex as it is, makes it difficult to impose precise rules or metrics, so these smells are more general guidelines and are susceptible to taste and interpretation. Along with gaining more experience and knowledge, you develop more expertise in identifying and eliminating bad smells in your code.

Transforming the Code

The next step leads you to modifying the code's internal structure. Here, refactoring theory has developed a set of formal rules that enable you to execute these *transformations* in such a way that, for a client, these modifications are transparent. You do not have to tackle the theory behind these rules. The toolmakers use these rules to make certain that refactoring modifies the code in a predictable way.

For example, let me illustrate this modification that preserves the original behavior of the code with an example. In Table 1-1, imagine you transformed the code at the left side into the code on the right side.

Table 1-1: Two Forms of Writing the Code that Will Execute in the Same Way

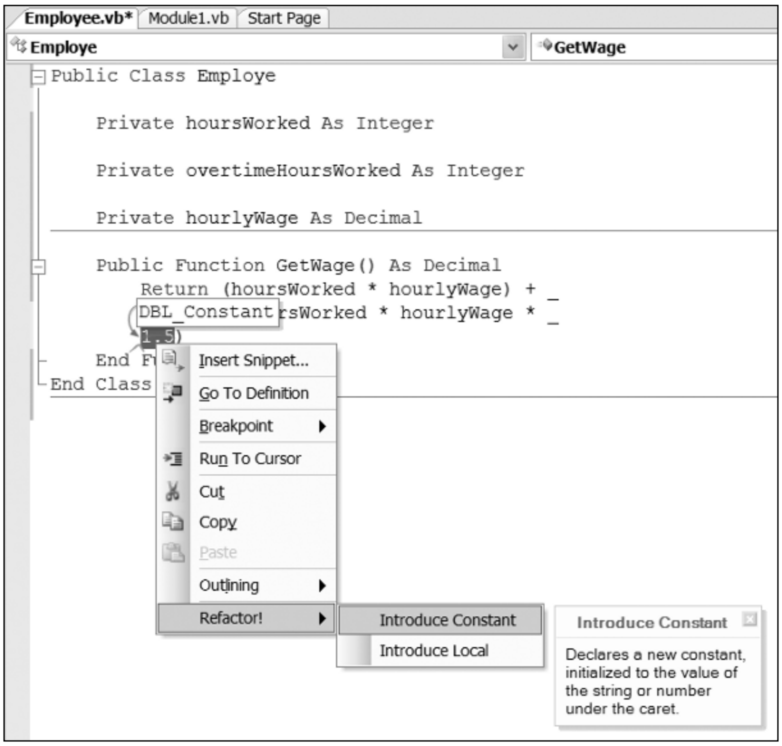
Free Literal Value	Literal as Constant
Public Class Employee	Public Class Employee
	Public Const OvertimeIndex _
	As Decimal = 1.5
Private hoursWorked As Integer	Private hoursWorked As Integer
Private overtimeHoursWorked _	Private overtimeHoursWorked _
As Integer	As Integer
Private hourlyWage As Decimal	Private hourlyWage As Decimal
Public Function GetWage() _	Public Function GetWage() _
As Decimal	As Decimal
Return (hoursWorked * _	Return (hoursWorked _
hourlyWage) + _	*hourlyWage) + _
(overtimeHoursWorked * _	(overtimeHoursWorked _
hourlyWage _	* hourlyWage _
* **1.5**)	* **OvertimeIndex**)
End Function	End Function
End Class	End Class

All you did here was to replace the literal value `1.5` with a constant `OvertimeIndex`. Executing the code on both sides provides identical results, but the one on the right can be a lot easier to maintain or modify. And it is definitely easier to understand. Now you understand that the literal `1.5` has a special meaning and has not been selected by chance.

Automating Refactoring Transformations

Refactoring rules have one great consequence: it is possible to automate a large number of these transformations. Automation is really the key to letting refactoring show its best. Refactoring tools will check for the validity of what you are trying to perform and let you apply a transformation only if it doesn't break the code. Even without a tool, refactoring is worth your while; however, manual refactoring can be slow and tedious.

Figure 1-1 shows the Refactor! for VB Visual Studio add-in from Developer Express integrated with Visual Studio 2005.



```
Public Class Employe

    Private hoursWorked As Integer

    Private overtimeHoursWorked As Integer

    Private hourlyWage As Decimal

    Public Function GetWage() As Decimal
        Return (hoursWorked * hourlyWage) + _
        DBL_Constant rsWorked * hourlyWage * _
        1.5)
    End F
End Class
```

Declares a new constant, initialized to the value of the string or number under the caret.

Figure 1-1

The Benefits of Refactoring

In light of all this, it is pertinent to ask what benefits refactoring brings. After all, it is not about adding new features or resolving bugs, and you end up with code that basically does what it used to, so why should you invest the time and money to perform this activity? What are the benefits of keeping your design optimal at all times? How does refactoring pay off?

Keeping the Code Simple

With the fact that software development is a continuous, evolving process, refactoring can bring important qualities to your code. Keeping your code lean at all times can be challenging, especially when you're under pressure to deliver the results quickly. So how does your code become overly complex? There are several ways this happens.

❑ In one typical scenario, you add a function here, a property there, another condition will crop up, and so on. This will soon produce a situation where classes and methods have grown and gone beyond their original purposes. They have too many responsibilities, communicate with many other elements, and are prone to change for many different reasons. It also becomes a breeding ground for duplicated code.

- In another scenario, you start off with a very thorough design that proves to be more than you really need. Simple code does only what it is supposed to do; you need not be so concerned much with trying to have your solution respond to any possible situation even before it happens. You can easily develop a tendency to overengineer your code, using complex structures when simple ones will do. (You can easily identify this school of thought by how many "what if" statements are used in discussions of the code.) This situation is motivated by an urge to anticipate future requirements even before they are expressed by the client.
- Performance has proved to be a lure for generations of programmers. You might spend numerous hours in order to obtain nanosecond gains in execution time. Without trying to lessen the importance of this key quality of software, you should bear in mind the right moment to deal with it. It can be very difficult to find the critical line you need to change in order to improve performance even for systems already in production; there is even less of a probability that you can find it while the system is in plain development and you are not sure what the rest of the pieces will end up looking like. Using the IDE as the performance-testing environment can be equally misleading.

How can you avoid such pitfalls? Once you become aware of them, you should deal with them quickly. Keeping things on the simple side will be greatly rewarded each time you need to add a feature, resolve a bug, or perform some optimization.

- If you see that a method has grown out of proportion, it is time to add a new method(s) that will take off some of the burden.
- If a class has too many members, maybe it can be restructured into a group of collaborating classes or into a class hierarchy.
- If a modification left some code without any use and you are certain that it will never be executed, there's no need to keep it; it should be eliminated.

All these solutions represent typical refactorings. After a smell is discovered, the solution is a restructuring of the problematic code.

When code is simple, it is easy to navigate — you don't lose time in long debugging sessions in order to find the right spot. The names of classes, methods, and properties are meaningful; code purpose is easy to grasp. This type of code won't have you reaching for documentation or desperately searching through the comments. Even after a short time spent with such code, you feel it does not hide any major mysteries. In simple words, you are in control.

Keeping the Code Readable

Programming is intellectually a very intense activity. You are often so immersed in your work that you tend to have a deep and detailed understanding of your creation in order to maintain complete control over it. You may try to memorize every single detail of the code. You feel proud when you are able to immediately correct a bug or change some behavior. After all, it is what makes you good in the work you perform. As you become more productive, you develop strategies and gain your own programming style. There is nothing wrong with being expert with the code you create, unless that expertise becomes the only weapon you have in your arsenal.

Unfortunately, sometimes you can forget one important fact; when developing software you seldom work alone. And in order to be able to work in a team, you must write code so it is easily comprehended by others. Others might need to modify, maintain, or optimize the code. In that case, if confronted with cryptic or hermetic code, others could lose numerous hours in a pure attempt to understand the code. Sooner or later you'll have a computer do all your bidding, but until then, writing source in such way that it is easy for others to understand can prove to be a much more difficult task. Ironically, you can find yourself in the "other person" role even with your own code. Your memory has its limits, and after a while you may not be able to remember every detail of the code you yourself wrote.

Readability can depend on different factors. Visual disposition is easily corrected and standardized with the IDE. Other factors, like the choice of identifier names, require a carefully thought-out approach. Because programmers often come from different backgrounds and have different experiences, the best bet is relying on natural language itself. You have to translate your decisions into code so they are easily understandable from a reading of the code, not only visible as a consequence of code execution. Code becomes really meaningful when a relation between it and a problem domain is correctly established.

As a programmer, you continuously develop your vocabulary. Using well-known idioms, patterns, and accepted conventions can increase the clarity of your code.

Reliance on comments and documentation can also affect the capacity of code to communicate with the reader. Because these artifacts never get executed, they are the first to suffer from obsolescence. Secondly, they are notorious for containing superfluous information.

I will try to illustrate this with two code snippets that perform equally during execution. Try reading first the snippet on the left side in Table 1-2, and then the one on the right.

Table 1-2: An Example of Code that is Difficult to Read and the Same Code in a More Readable Form

Difficult to Read	More Readable Code
`Dim oXMLDom As _`	`Dim portfolio As _`
`New DOMDocument40`	`New DOMDocument40`
`Dim oNodes As IXMLDOMNodeList`	`Dim stocks As IXMLDOMNodeList`
`'loads the file into XMLDom object`	
`oXMLDom.Load(App.Path + _`	`portfolio.Load(App.Path + _`
`"\ portfl.xml")`	`"\portfl.xml")`
`oNodes = _`	`stocks =`
`oXMLDom.selectNodes("//stock[1]/*")`	`portfolio.selectNodes("//stock[1]/*")`

If I have proved my point, you will find the second snippet more to your liking. In case you still are not convinced, as an interesting experiment, you can try obfuscating your code with some obfuscation tool and then trying to find your way around it. Even with the smallest code base, it soon becomes impossible to understand the code. No wonder, because obfuscation is a process completely opposite to refactoring.

Refactoring tools can help you improve readability by letting you rename identifiers in your code in a safe and systematic way and by letting you transform your code along well-known patterns and idioms — you use comments in a more profound manner. Strong structure in the code gives you confidence that the information you obtain from reading the code relates well to execution time.

All this sounds very good. However, you can often hear arguments against refactoring. While some of those arguments are well founded, let me first deal with some opinions often heard that are not very constructive.

79796c01.qxd:WroxPro 2/25/08 8:55 AM Page 12

Debunking Common Misconceptions

Like any topic that creates a huge amount of interest among developers, refactoring has produced an avalanche of opinions and contributions, some of more and others of less value. In certain cases I found those opinions so unfounded that I call them misconceptions. I feel it is worthwhile taking some time to debunk them, because they can add confusion and can lead you astray from a quest to adopt this valuable technique.

Refactoring Violates the Old Adage, "If It Ain't Broke, Don't Fix It"

Often portrayed as longstanding engineering wisdom, this posture only promotes complacency. Refactoring does teach against it, but for a reason.

Early on you learn how even a minuscule detail in code can make all the difference, often paying dearly for this knowledge. A small change can provoke software to break in a surprising manner and at the worst moment. So once you burn your hands you often become reluctant to make any change that is not absolutely necessary. This can work well for a moment, but then a situation comes up where bugs have to be resolved and petitions for new features cannot be evaded anymore. You are faced with the same code you tried not to confront.

Those who adopt this "if it ain't broke, don't fix it" position look upon refactoring as unnecessary meddling with something that already serves its purpose. Actually, this conformist posture that tries to maintain the "status quo" is the result of an intent to rationalize the fear of confronting the code and the fact that you do not have control over it.

Refactoring Is Nothing New

This misconception could be restated as, "Refactoring is just another word for what we all know already." Which means you have all learned about good code, object-oriented design, style, good practices, and so on, and refactoring is just another buzzword that someone invented to sell some books.

Okay, refactoring does not pretend to be imposing a radically new paradigm like object-oriented or aspect-oriented programming. What it does do is radically change the way you program: it defines rules that make it possible to apply complex transformations to code at the click of a button. You do not look at your code as some frozen construct that is not susceptible to change. Instead, you see yourself as capable of maintaining the code in optimum shape, responding efficiently to any new condition.

Refactoring Is Rocket Science

Programming is hard. It's a complex activity that requires a lot of intellectual effort. Some of the knowledge can be very difficult to grasp. With Visual Basic .NET, VB programmers had to acquire the ability to work in a fully capable object-oriented language. For many, this was baffling at first. The good part is it definitely pays off.

The great thing about refactoring is how simple it can be. It equips you with a very small set of simple rules to start off. This, coupled with a good tool, makes first steps in refactoring a breeze. Compared to other techniques an advanced programmer should know nowadays, like UML or design patterns, I'd say refactoring has the easiest learning curve, a lot like VB itself compared to other programming languages. Very soon, the time spent in learning refactoring will start to reap rewards. Of course, as with any other thing in life, gaining mastery requires a lot of time and effort.

Refactoring Causes Poor Performance

A longer way to state this might be, "Because after refactoring you usually end up with a larger number of more fine-grained elements like methods and classes, so much indirection must incur some performance cost."

If you go back in time a little, you'll discover that this argument curiously sounds like the one used to voice initial skepticism toward object-oriented programming. The truth is that the differences between refactored and unstructured code are, at best, minimal. Except in some very specialized systems, this is not a concern.

Experience shows that performance flows are generally afflicted by some precise spots in code. Fixing those during an optimization phase will get you the required levels of performance. Being able to easily identify the critical pieces of code can prove to be very valuable. By producing understandable code in which duplication and total size is minimized, refactoring greatly aids this task.

Refactoring Breaks Good Object-Oriented Design

Well-structured and refactored code can look awkward to an untrained eye. Methods are so short that they often seem without substance. Classes seem without enough weight, consisting of only a few members. It seems as if nothing ever happens in our code.

Having to manage a greater number of elements like classes and methods can imply that there is more complexity to deal with. This argument is actually misleading. The truth is that the same complexity was always present, only in refactored code it is expressed in such a cleaner, more structured way.

Refactoring Offers No Short-Term Benefits

Refactoring actually makes you program faster. So far, I do not know of any study that I could call upon in order to prove what I just said, but my own experience tells me this is the case. All the same, it is only logical that this is so. Because we have a smaller quantity of code overall, less duplication, and a clearer picture, unless we are dealing with some trivial and unrealistically small scale code, benefits become apparent very soon.

Refactoring Works Only for Agile Teams

Because it's often mentioned as one of the pillar techniques in agile methodologies, refactoring is interpreted as working only for teams adhering to these principles.

Refactoring is indispensable for agile teams. Even if your team has a different methodology, most of the time you are the one in charge charge of the way you code. Best results in refactoring are achieved if you adopt refactoring in small steps, performing it regularly while you code. Some practices, like strict code ownership or a waterfall process, can play against refactoring. If you can prove that refactoring makes sense from a programming point of view, you can start building your support base, first with your peers and then by spreading the word to the rest of your team.

That dispenses with some of the common misconceptions surrounding refactoring. At this point, you may be wondering how all of this relates to Visual Basic. That is the topic of the next section.

Visual Basic and Refactoring

It is fair to say that in the Visual Basic community, refactoring has had a slow start. One of the main reasons for this was the lack of proper tool support. While some tools with refactoring capabilities appeared on the market some years ago, only recently did dedicated refactoring tools for VB appear. Lack of tools coupled with lack of information and scarce literature suited for VB developers led to slow adoption of the technique. It seems, however, that in this case the developer community was ahead of the industry policy makers and commercial institutions. Refactoring support was voted the number-one desired feature for the 2005 edition of Visual Basic IDE. Realizing the importance this feature has for VB developers, Microsoft partnered with Developer Express to release a free Visual Basic refactoring add-in for Visual Studio 2005.

Visual Basic History and Legacy Issues

While refactoring was slow to take over in the Visual Basic community, it can be argued that refactoring has even greater importance for Visual Basic programmers than for programmers in some other languages. Visual Basic longevity means that VB developers need to deal with a host of legacy issues even today. In this effort, refactoring can be a great help by providing the programmer with the tools for unobtrusive transformation of legacy constructs into more appropriate contemporary code.

Visual Basic has been in existence for more than 15 years. During that time it has earned a huge following — it has become one of the most popular programming environments in existence. Thanks to its syntax based on the BASIC programming language and the graphical environment for drawing GUI elements, it has proven to be an easily accessible programming environment with a gradual learning curve. And thanks to Microsoft's policy of spreading the use of Visual Basic in other forms and other environments like Windows Scripting, Visual Basic for Applications used for Office automation, Active Server Pages, and others, the circle of VB adopters has significantly increased.

Visual Basic Evolution

Since its inception, Visual Basic has undergone a steady process of evolution and improvements. In version four, class constructs were added to Visual Basic, paving the way for object-oriented programming in Visual Basic. However, not until VB .NET did Visual Basic unleash the full power of object-oriented programming. In VB .NET, implementation inheritance support was added.

With the advent of the .NET Framework, Microsoft decided to make a more significant overhaul of the language and to make it more appropriate for the new platform. Significant improvements were made:

- ❑ Added inheritance support
- ❑ Optional static type checking (`Option Strict`)
- ❑ Structured Error Handling (`Try-Catch-Finally`)
- ❑ Attributes

Many other programming elements were removed or replaced in an attempt to clean up the language syntax and make it more in the spirit of the .NET Framework. In the .NET Framework, all code gets compiled into Intermediate Language and then traduced into native binary. This is true for code programmed in VB .NET also. This makes it possible for VB code to interact with code programmed in other languages.

VB programmers can use code programmed in C# or managed C++. The reverse is also true — C# or C++ programmers, or programmers in any other .NET language for that matter, can import and use assemblies programmed in Visual Basic .NET. This is not so different from the interoperability provided by COM in the pre-.NET era, with the distinction that it is also possible to inherit types programmed in different languages.

This continuous work on language improvement continues even today. For example, in Visual Basic 2005, support for generic types was added. In the 2008 version of VB, new features like LINQ, XML Literals, Extension Methods, and Lambda Expressions continue to keep VB at the forefront of .NET programming languages.

Along with market forces that are continuously moving Visual Basic forward in making it more powerful and advanced so that it stays as efficient as other programming languages, some frictional forces stand in the way of its progress. Its long history and success are the main reasons for keeping some of its older language elements that would normally be completely replaced with newer ones. With C# Microsoft had a clean slate for language design. With Visual Basic, it has to take into account a huge amount of existing code that should be migrated and a great number of developers who need to upgrade their skills.

Legacy Code

The huge popularity and widespread adoption of Visual Basic resulted in a great amount of pre-.NET code still in production even today, almost six years after the advent of .NET. Migrating VB6 or prior code to .NET is not a simple affair. While Microsoft provided an automated migration tool, this upgrade can not be realistically performed without user interaction. Code produced by the Migration Wizard will often leave portions of code that should be upgraded manually.

This in part demonstrates a somewhat brave decision by Microsoft not to subject VB .NET to upgrade necessities and backward-compatibility issues. But because a completely automated upgrade is not possible, a number of features were kept in VB .NET in order to provide at least some possibility of upgrade. Such features are:

❑ Old-style error handling (`On error...`)

❑ Optional static typing as opposed to mandatory static type checking (`Option Strict...`)

❑ Module language construct, etc.

Unfortunately, code with such features left over after the upgrade has not been fundamentally upgraded. It can execute in a new, .NET environment, but nevertheless it has kept old deficiencies.

Legacy Programming

In the computer industry, new technologies are the order of the day. As programmers, you are destined to upgrade your skills continuously and to acquire new knowledge. Unless you do so, the spectrum of employment opportunities and chances for career advancement can be greatly reduced. VB programmers were exposed to this process of continuous skill improvement throughout the history and evolution of VB, but never was the challenge as great as with the release of VB .NET.

By making VB a modern, fully object-oriented language with native access to the .NET Framework, Microsoft gave VB programmers a much more powerful tool. However, to be able to harness this power, programmers had to acquire new skills. It can be argued that with the advent of VB .NET a "paradigm shift" has been produced. The changes are significant and go beyond a simple upgrade. These changes require new skills and new ways of thinking.

However, because of many old elements that were kept in VB .NET, it is possible to program in VB .NET in the "old style," just as in VB6 or prior. But if that way is used, no benefits are gained from the new platform. This is not a new phenomenon in the history of programming. C++ and Java had their syntax based on C language syntax in order to attract and facilitate transfer of C programmers to these new languages. However, it was soon noticed that a number of these programmers started using new tools and environments in the old style, relying on old constructs and design applicable to the old programming language. With C++ and Java, this meant that programmers continued using procedural design instead of relying on new, object-oriented design. Admittedly, in VB the gap is not that great. Even in VB6 you can define a class, create an instance, or define and implement an interface. Nonetheless, using inheritance and generics and understanding the benefits of static typing are important challenges for someone coming from a classic VB background, and surmounting these challenges requires a major shift in the way programming is approached.

Legacy Language Elements

So these backward forces I've discussed led to a situation in which many of the obsolete language elements were kept in VB .NET for the purpose of upgrades of existing code or with the intention of facilitating the transition of programmers to a new platform.

Unfortunately, these features can just as easily be used in newly created VB .NET code, although there are other, much better alternatives. To be able to distinguish between the two, a solid command of VB .NET and object-oriented principles is required.

Dealing with Legacy Issues Through Refactoring

Building on the fundamentals of object-oriented theory, refactoring goes a step further than a traditional approach to programming does. With refactoring, it is easy to identify weak spots in your code and apply small-scale transformations that do not change the behavior of code but can deal with the shortcomings. In that sense, some of the legacy language elements can be considered undesirable, and, just like any other smell, this legacy code smell can be dealt with through the refactoring process. By defining procedures that can transform legacy elements and fundamentally upgrade the code, refactoring can greatly alleviate issues related to upgrade and make your old code fully capable and equally useful in the .NET world.

Summary

This introductory chapter has given you a brief overview of refactoring, explaining its relevance and benefits. You have seen how refactoring helps you design your applications and prevent design rot, and at the same time accommodate any change that your software might be exposed to.

You have learned the three important stages in each cycle of the refactoring process: smell identification, refactoring, and testing. These three stages are mandatory for successful refactoring. In order to make this process even more productive, you can rely on automated refactoring tools that take a lot of the drudgery and complexity out of refactoring, making it easily accessible and applicable.

You have also also been presented with some of the most common misconceptions of refactoring that you might come across, just so you won't be surprised by the diversity of opinions on the subject and can make your own informed choices.

In the second part of the chapter, I put Visual Basic into focus. A very popular and successful tool, it was not immune to changes and advances in the programming world. While it has evolved significantly, its longevity means that a host of legacy and backward-compatibility characteristics were preserved inside the language in an attempt to make the upgrade to new versions less upsetting. Unfortunately, this also meant that a lot of programmers continued to program in the legacy style, not reaping the benefits of the advances of this fully object-oriented language that came with th advent of VB .NET.

You have seen how refactoring can play a significant role in the transition and upgrade of legacy code to VB .NET. You can rely on it while you acquire new knowledge and sharpen your programming and design skills.

Now it's time to see some of this in practice. In the next chapter you are going to see refactoring at work. I will write some code and use a small application to illustrate the power of the refactoring process.