1

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 tips 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 programmers, it has become 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 enables them to 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. C# is a part of this; currently, a great number of developers are performing refactoring in C#, and a number of good and mature tools are available that you can use to automate refactoring process.

This chapter covers the following topics:

- What refactoring is and why it is important
- The benefits that refactoring delivers
- Common misconceptions about refactoring
- Specifics of C# as a programming language and how it relates to refactoring

First, however, let's 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 better 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?''

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. Refactoring teaches you how to efficiently modify your code in such a way that the impact of those modifications is kept to 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 flow and to modify the internal structure of code in order to improve the design without changing the code's visible behavior.

All design decisions are the result of your knowledge, experience, and creativity. However, programming is a vast playfield, and weighing the pros and cons of your design can be daunting. In C# 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 toward improving the design of your code.

The Refactoring Process

Refactoring is an important programming practice that 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. As a programmer myself, I know this is no small feat!

The refactoring process is fairly simple and consists of three basic steps:

1. **Identify code smells:** You'll learn 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 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 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 to ensure that 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 that can assume 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 anti-patterns are generally referred to as *code smells*; a great part of refactoring can be seen simply as an attempt to eliminate code smells.

Definition: ***Code smell*** **is a sense you develop that tells you 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, which is generally achieved by applying modifications to it. 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 a software developer, 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, including the following concerns:

- ❑ **User requirements:** This generally means that you should create software that does what the client paid for.
- ❑ **Quality:** To guarantee the quality of your product, you should strive to reduce defects and to release a program that has the minimum number of bugs.
- ❑ **Usability:** This includes making programs that are easy to understand and use.
- ❑ **Performance and efficiency:** As you develop, you want to find new ways to minimize memory usage and the number of cycles needed in order to solve the given problem.
- ❑ **Timeliness:** In order to achieve all of this in a timely manner, you should always be looking for ways to augment productivity.

These goals 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. However, there is an inevitable gap between the artifacts you work on — the *source* — and the artifacts you are producing — the *binary*.

This gap between the creation process and the finished product is not typical in other areas of human activity. Consider 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, such as Windows Forms Designer inside Visual Studio for example, which largely bridges this gap between source and binary, all you do in the end is create the source that is later processed and turned into a compiled unit.

What's more, there are many ways to write the source that will produce essentially 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 into a final product, they have an immense impact on the whole process of creation and maintenance.

This leads to the following question: Can you distinguish between well-written and poorly written code, even if the final result is the same? In addition, if the final result, the binary, is performing well from user's point of view, is it at all important how the source code is written? The following sections explore these questions, and you'll see how refactoring can clarify any doubts you might have in this respect.

Refactoring Encourages Solid Design

Regardless of 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 whether a bit of code is indented and formatted in a pleasing manner, whether the standard 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 C#, you first and foremost 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 such as classes and interfaces, and features such as encapsulation, inheritance, and polymorphism to build a cohesive structure that describes the problem domain well. In a certain sense, you build your own ad hoc language on top of a common language — one 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, extendable, and reliable, and that communicates its purpose well, you can say you have reached your goal of creating well-designed code.

Refactoring provides you with 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 such as 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. In some cases, requirements will change even before the first release, new features will be added, defects will be corrected, and even some planned design decisions, when confronted with real-world demands, will be overruled.

Software construction is a very complex activity, and it is futile to try to come up with a perfect solution up front. Even if you're using some sophisticated techniques such as modeling, you'll still come up short of thinking of every detail and every possible scenario. It is often this state of flux that presents the biggest challenge in the process of making software. You have no choice but to count on change, be ready to adapt, and react readily when it happens. If you are not ready to react, the design decisions you made are soon obsolete, 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 any changes can have on your original 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 revised or entirely new 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 are removed, defects are resolved and adaptations are performed. It is quite common to find more than one person putting their hands on the same piece of software, each with his or her own style and preferences. Rarely will the same team of people see the software process 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 it's definitely not for the ordinary craftsman.

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

This brief survey of the software landscape has pointed out several challenges that developers face. In order to understand how refactoring can help, the next section describes refactoring in more detail.

The Refactoring Process: A Closer Look

The previous section described a few key areas of software development that can often lead to poor code. Obviously, you need to stand guard over the quality of your code. In effect, you need to have the design quality of your code in mind at all times.

While this sounds sensible, thinking continuously about design and code quality can 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, you'll learn about the refactoring activities you would typically complete during a software development cycle.

Using Code Smells

As a first step in your refactoring activity, you'll 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, which can help you with this task (along with the well-known notions and principles of object-oriented design). Programming, being as varied and complex as it is, makes it difficult to impose precise rules or metrics, so these smells should be used as general guidelines only. Susceptible to taste and interpretation, they have to be applied using your own judgment in accordance with each specific situation.

Coding schools that promote strict coding guidelines, along with strict usage of static code analysis tools, might see this as unnecessary flexibility, but such a relatively liberal approach actually promotes programmer creativity and freedom, resulting in quality code that does not inhibit originality and innovation. Creative programmers thrive in an environment where refactoring is promoted as a first-class practice. In this sense, refactoring promotes programming excellence.

Therefore, in addition to 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, from a client's standpoint, the modifications are transparent. To make use of refactoring, you do not have to tackle the theory behind these rules. The toolmakers use these rules to ensure that refactoring modifies the code in a predictable way, so you can rely on your tool to perform a transformation without breaking the code.

To illustrate how refactoring preserves the original behavior of the code, let's look at an example. Using the contents of Table 1-1, imagine you transformed the code on the left side into the code on the right side.

All you did here was to replace the literal value 1.5m with a constant OvertimeIndex. Executing the code on both sides provides identical results, but the one on the right is a lot easier to maintain or modify. In case you use OvertimeIndex again, the value will be specified in a single place and not scattered around in your code.

What's more, the code on the right side of the table is definitely easier to understand. Once you understand that the literal 1.5m has a special meaning, you can easily relate its value to the business rules that govern your code.

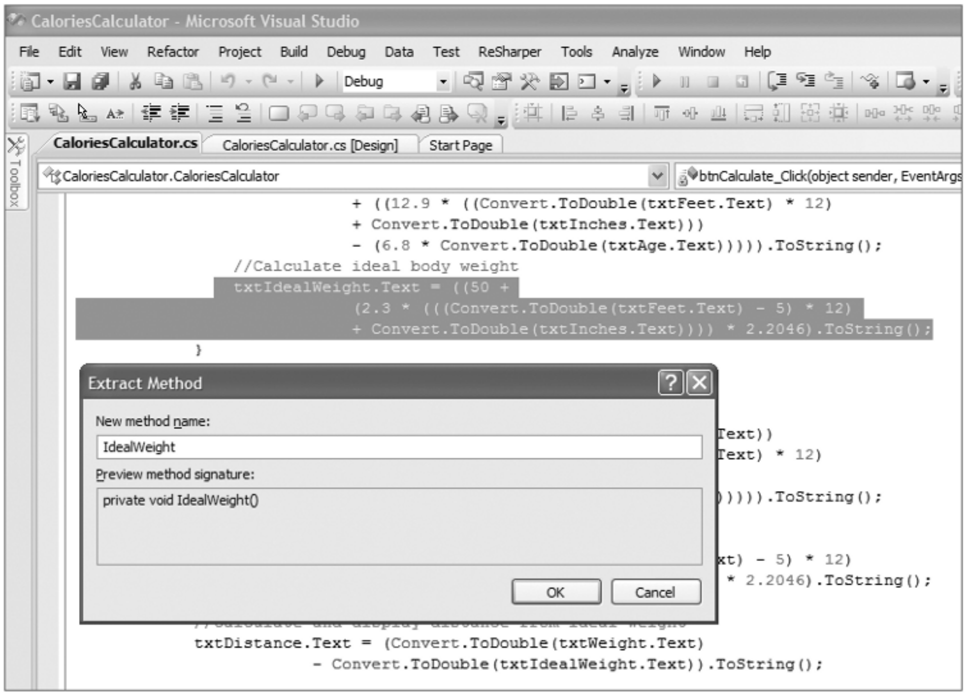
Automating Refactoring Transformations

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 put its best foot forward. Refactoring tools will check for the validity of what you are trying to perform, and then enable you to apply a transformation only if it doesn't break the code. Even without a tool, refactoring is worthwhile; however, manual refactoring can be slow and tedious.

Figure 1-1 shows extract method refactoring in progress in Visual Studio 2008.

Table 1-1: Two Ways of Writing Code That Execute in the Same Way

Free Literal Value	Literal as Constant
`public class Employee`	`public class Employee`
`{`	`{`
	`    public const decimal`
	`OvertimeIndex = 1.5m;`
`    private int hoursWorked;`	`    private int hoursWorked;`
`    private int overtimeHoursWorked;`	`    private int overtimeHoursWorked;`
`    private decimal hourlyWage;`	`    private decimal hourlyWage;`
`    public decimal GetWage()`	`    public decimal GetWage()`
`    {`	`    {`
`        return (hoursWorked *`	`        return (hoursWorked *`
`        hourlyWage) +`	`        hourlyWage) +`
`            (overtimeHoursWorked *`	`            (overtimeHoursWorked`
`           hourlyWage * 1.5m);`	`            * hourlyWage`
`    }`	`            * OvertimeIndex);`
`}`	`    }`
	`}`


CaloriesCalculator - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Data Test ReSharper Tools Analyze Window Help
Debug
Toolbox
CaloriesCalculator.cs
CaloriesCalculator.cs [Design]
Start Page
CaloriesCalculator.CaloriesCalculator
btnCalculate_Click(object sender, EventArgs
OK
Cancel

```
                    + ((12.9 * ((Convert.ToDouble(txtFeet.Text) * 12)
                    + Convert.ToDouble(txtInches.Text))
                    - (6.8 * Convert.ToDouble(txtAge.Text))))).ToString();
            //Calculate ideal body weight
            txtIdealWeight.Text = ((50 +
                    (2.3 * (((Convert.ToDouble(txtFeet.Text) - 5) * 12)
                    + Convert.ToDouble(txtInches.Text))) * 2.2046).ToString();
        }
```

Extract Method

New method name: IdealWeight

Preview method signature: private void IdealWeight()

```
Text))
Text) * 12)
))))).ToString();
xt) - 5) * 12)
 * 2.2046).ToString();
        txtDistance.Text = (Convert.ToDouble(txtWeight.Text)
                - Convert.ToDouble(txtIdealWeight.Text)).ToString();
```

Figure 1-1

The Benefits of Refactoring

In light of all the rules and techniques associated with refactoring, it is pertinent to ask what benefits refactoring offers. After all, it does not add new features or resolve 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

Because software development is a continuous, evolving process, refactoring can simplify the numerous aspects you need to juggle as a developer, streamlining your work and providing 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. Code can become overly complex in several ways:

- **Adding new features:** In one typical scenario, you have to add some functionality to your application. You add a function here, a property there, another condition crops up, and so on. This will soon produce a situation in which classes and methods have grown and ultimately exceeded their original purpose. They have too many responsibilities, communicate with many other elements, and are prone to change for many different reasons. This also becomes a breeding ground for duplicated code.
- **Big design up front:** 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 concerned with trying to have your solution anticipate and respond to every possible scenario before it even happens. You can easily develop a tendency to over-engineer your code, using complex structures when simple ones will suffice. (You can easily identify this tendency by how often ''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.
- **Premature optimization:** 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. Let me show you a few examples:

- If you see that a method has grown out of proportion, then it is time to add a new method, or methods, that will relieve some of the burden.
- If a class has too many members, maybe it can be restructured into a group of collaborating classes or a class hierarchy.
- If a modification left some code without any use and you are certain that it will never be executed, then 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. Such code follows the *rule of least surprise* and it is written in a uniform manner. This type of code won't have you reaching for documentation or desperately searching through the comments. Even after a short amount of time is 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 at 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 understood by others, who might need to modify, maintain, or optimize the code. In that case, when confronted with cryptic or hermetic code, other team members could lose numerous hours attempting to understand the code. Sooner or later you'll have a computer to 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 very 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 code you wrote yourself.

Readability can depend on different factors. Visual disposition is easily corrected and standardized with the IDE. Other factors, such as the choice of identifier names, require a carefully thought-out approach. Because programmers 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 truly 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 are never executed, they are the first to suffer from obsolescence. In addition, they are notorious for containing superfluous information.

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

Table 1-2: Difficult to Read Code vs. More Readable Code

Difficult to Read Code	More Readable Code
`XmlDocument oXMLDom =`	`XmlDocument portfolio = new`
`new XmlDocument();`	`XmlDocument();`
`//loads the file into XMLDom object`	`portfolio.Load(ApplicationPath  +`
`oXMLDom.Load(strAppPath  + "\\`	`"\\ portfolio.xml");`
`pfl.xml");`	
`XmlNodeList oNodes =`	`XmlNodeList stocks =`
`oXMLDom.SelectNodes("//stocks[1]/*");`	`XMLDom.SelectNodes("//stocks[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 an 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 enabling you to rename identifiers in your code in a safe and systematic way and by transforming 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’s first look at some baseless objections.

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 value and others of less value. In certain cases, I find the opinions so unfounded that I call them misconceptions. I feel it is worthwhile taking some time to debunk them, because they can add doubt and confusion, and lead you astray from a quest to adopt this valuable technique.

‘‘If It Ain’t Broke, Don’t Fix It’’

Often portrayed as longstanding engineering wisdom, the ‘‘if it ain’t broke, don’t fix it’’ posture only promotes complacency. Refactoring teaches against it, and for a good reason.

Early in your programming life you learn how failure to pay attention to even a minuscule detail in code can make huge difference, often paying dearly for this knowledge. A small change can provoke software to break in a surprising manner and at the worst time. Once you have burned your hands, you can become reluctant to make any change that is not absolutely necessary. This can work well for a while. Inevitably, however, a situation comes along whereby bugs have to be resolved and petitions for new features cannot be evaded anymore. You are faced with the same code you tried so hard not to confront.

Those who adopt the ‘‘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, which tries to maintain the ''status quo,'' often results from rationalizing the fear of confronting the code and the fact that you do not have control over it.

Many experienced programmers adopt this posture out of a legitimate intent to minimize any effort that is not truly necessary. For example, if the application is doing fine in terms of performance, then there is no need to squeeze that last processor cycle out for the sake of performance. The same is true for speculative design, often argued with ''prophetic'' programmer talk such as ''we might need that feature one day.''

In that sense, refactoring is pretty much on the same wavelength. When you refactor, you should eliminate dead code and avoid speculative design or premature optimization. However, for refactoring adepts, software can also be broken ''on the inside.'' If the design is flawed, or the code poorly written and badly structured, then even if the application is serving users correctly at the moment and is not broken visibly ''on the outside,'' it should still be refactored and have its design ''fixed.'' In that sense, refactoring insists on some less tangible but decisive characteristics of software, such as design, simplicity, improved source code comprehension, readability, and the like.

Refactoring will help you win back your command over your code. While this is hardly an easy task with a code base that has grown out of control, without refactoring the only solution you have left is the complete rewrite.

''Refactoring Is Nothing New''

This misconception could be restated as ''refactoring is just another word for what we all know already.'' In other words, you might think this means you have mastered good code practices, object-oriented design, style, good coding, and so on, and refactoring is just another buzzword that someone invented to sell some books.

Refactoring does not pretend to impose radically new paradigms such as object-oriented or aspect-oriented programming when they appeared for the first time. 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. Rather than see your code as some frozen construct that is not susceptible to change, you instead see yourself as capable of maintaining the code in optimum shape, responding efficiently to new challenges and changing the code without fear.

''Refactoring Is Rocket Science''

Programming is hard. It's a complex activity that requires a lot of intellectual effort, and some of the knowledge can be very difficult to grasp. With C#, programmers had to first acquire the ability to work in a fully capable object-oriented language. Nowadays, C# is being enriched by elements of functional and even dynamic language capabilities. For many, this can be baffling. Fortunately, learning new skills 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 begin with. This, coupled with a good tool, makes first steps in refactoring a breeze. Compared to other techniques an advanced programmer should know nowadays, such as UML or design patterns, I'd say that refactoring has the easiest learning curve.

Very soon, the time spent in learning refactoring will start to reap rewards. Of course, as with anything else in life, gaining mastery requires a lot of time and effort.

''Refactoring Causes Poor Performance''

This objection is really suggesting that because after refactoring you usually end up with a larger number of fine-grained elements, such as methods and classes, a new design with so much indirection must incur some performance cost.

If you go back in time a little, however, you'll discover that this argument sounds curiously similar to the one used to voice initial skepticism about object-oriented programming languages, as compared to procedural programming languages, such as comparing C with C++.

The truth is that the differences in performance between refactored and unstructured code are, at best, minimal. Except in some very specialized systems, this is not a real concern.

IO-intensive Code vs. Computation-intensive Code

The majority of typical enterprise applications are distributed, layered applications that involve a lot of input-output operations. This is especially true under today's prevailing architectural paradigm of service-oriented applications. These I/O operations can be network communication, inter-process communication, file I/O, and so on. For example, from an architectural point of view, an ASP.NET application's execution stack might look like this:

- ❑ IIS receives petition from browser (network I/O)
- ❑ ASP page communicates with remote COM+ component (network I/O)
- ❑ COM+ component communicates with several other remote components (network I/O)
- ❑ Next COM+ component communicates with remote web service (network I/O)
- ❑ Next COM+ component writes to log (file I/O)
- ❑ Next COM+ component communicates with database (network I/O)

Generally, I/O operation is by magnitudes slower than internal, computational operations. In such applications, the biggest performance gains are obtained by optimizing I/O operations. Such optimizations might include tactics such as connection pooling, optimizing database query performance, and filtering data in the database versus filtering data in the client, and so on. This leads you to the next premise related to refactoring and performance.

Application Performance Is Defined by Performance Bottlenecks, Not by Absolute Performance

Once you have committed yourself to a certain platform, you should try exploit all the benefits it provides. You could write a procedural program in C#, but it doesn't make sense. Aggressively structuring your code might produce some small overhead, but such overhead is well compensated by the benefits that refactored code provides.

When code is refactored and well structured, it is much easier to optimize it. Because there is less duplication, there is only a single place where such optimization needs to be performed. Even the computational part of your code will reap the same benefits from well-organized code. The alternative — not structuring your code in order to avoid method lookup overhead or applying similar optimization strategies — will confer only insignificant benefits. One of the reasons for this is the intelligence that the .NET platform has been bestowed with.

Compiler, Optimizations, and JIT

Compilers today are mature and extremely sophisticated pieces of technology. They are a critical piece in any software platform, so it is no surprise that so much time and so many resources have been put into them. .NET is by no means an exception. In contrast to other managed environments .NET produces a native code that makes no performance-related concessions. As a comparison, Java code is compiled to bytecode and then executed inside the Java Virtual Machine.

Just-in-Time (JIT) compilation is a technique used to improve the performance of programs by compiling the code (in the case of .NET Common Intermediate Language, or CIL) to native and executable binary file.

Even so, the .NET JIT compiler is capable of producing many very sophisticated optimizations. Moreover, code can be optimized for a specific platform, as it is compiled on the same machine on which it is executed, dead code can be eliminated, methods can be inlined, and so on. Theoretically, even runtime profiling data can be used to optimize code even further, but at this point we are entering a territory that is covered with the veil of secrecy. No wonder, as these technologies are critical for the platform's success and can represent an important competitive advantage.

This means that a lot of the overhead that aggressively structured, refactored code can introduce will be eliminated by the .NET runtime itself — and by the looks of it, it does a pretty good job of doing so. In the next section I plan to prove it.

Experimenting with Performance

Experience shows that performance flows are generally afflicted by some precise spots in the code. Fixing those during an optimization phase will enable you to reach 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, and by enabling changes to be performed in a single place, refactoring greatly aids the process of optimization.

These days, as CPU horsepower is constantly incremented, other aspects of code, such as maintainability, quality, scalability, and reliability, are forcing performance out of the top-priority code features. Now, don't use hardware speed as a pretext to write code that performs lousy, just don't go overboard with optimizing your code.

Just in case you would like to see some numbers here, I have prepared a small experiment. I will use two code samples. The first is an example of poorly structured code and has single `Main` method. The second example has a `Main` method inside a `Module` and a class `Circle` with a number of fine-grained methods. I originally used these samples to illustrate unstructured versus structured code style, so all it lacks is some measurement code.

I will measure the time it takes to execute a simple geometrical formula (circle circumference length); in order not to make this code computation-intensive, I will add some database query code. To even out the measurement, I will repeat execution inside a loop 10,000 times. Since I don't mean to make this an extremely precise experiment, I will use the `System.Diagnostics.Stopwatch` class to capture the interval; the precision of `Stopwatch` will suffice in this case.

In Listing 1-1 you can see an example of deliberately unstructured code.

Listing 1-1: Unstructured Code

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Diagnostics;

namespace RefactoringInCSharp.Chapter1
{
    struct Point
    {
        public double X;
        public double Y;
    }

    class CircleCircumferenceLength
    {

        public void Main()
        {
            Point center;
            Point pointOnCircumference;
            //read center coordinates
            Console.WriteLine("Enter X coordinate" +
                "of circle center");
            center.X = Double.Parse(Console.In.ReadLine());
            Console.WriteLine("Enter Y coordinate" +
                "of circle center");
            center.Y = Double.Parse(Console.In.ReadLine());
            //read some point on circumference coordinates
            Console.WriteLine("Enter X coordinate" +
                "of some point on circumference");
            pointOnCircumference.X = Double.Parse(Console.In.ReadLine());
            Console.WriteLine("Enter Y coordinate" +
                "of some point on circumference");
            pointOnCircumference.Y = Double.Parse(Console.In.ReadLine());
            //calculate and display the length of circumference
            Console.WriteLine("The length of circle" +
                "circumference is:");
            //calculate the length of circumference
            double radius;
            double lengthOfCircumference = 0;
            int i;
            //use stopWatch to measure transcurred time
            Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();
            //repeat calculation for more precise measurement
            for (i = 1; i <= 10000; i++)
            {
                //add some IO
                IDbConnection connection =
                    new SqlConnection("Data Source=TESLATEAM;" +
                        "Initial Catalog=RENTAWHEELS;" +
                        "User ID=RENTAWHEELS_LOGIN;"
                        + "Password=RENTAWHEELS_PASSWORD_123");
                connection.Open();
                IDbCommand command = new SqlCommand("SELECT GETDATE()");
                command.Connection = connection;
                IDataReader reader = command.ExecuteReader();
                reader.Read();
                reader.Close();
                connection.Close();
                radius = Math.Pow((Math.Pow((pointOnCircumference.X -
                    center.X), 2) +
                    Math.Pow((pointOnCircumference.Y - center.Y), 2)),
                    (1 / 2));
                lengthOfCircumference = 2 * 3.1415 * radius;
            }
            stopWatch.Stop();
            Console.WriteLine(stopWatch.Elapsed);
            Console.WriteLine(lengthOfCircumference);
            Console.Read();
        }
    }
}
```

Listing 1-2 shows the structured version, which will perform exactly the same operation as the code in Listing 1-1.

Listing 1-2: Structured Code

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Diagnostics;

namespace RefactoringInCSharp.Chapter1
{
    public struct Point
    {
        public double X;
        public double Y;
    }

    class CircleCircumferenceLength
    {

        public void Main()
        {
            Circle circle = new Circle();
            circle.Center = InputPoint("circle center");
            circle.PointOnCircumference =
                InputPoint("point on circumference");
            Console.WriteLine("The length of circle "
                + "circumference is:");
```

Continued

Listing 1-2: Structured Code *(continued)*

```
            double circumference = 0;
            int i;
            //use stopWatch to measure transcurred time
            Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();
            //repeat calculation for more precise measurement
            for (i = 1; i <= 10000; i++)
            {
                circumference =
                    circle.CalculateCircumferenceLength();
            }
            stopWatch.Stop();
            Console.WriteLine(stopWatch.Elapsed);
            Console.WriteLine(circumference);
            WaitForUserToClose();
        }

        public Point InputPoint(string pointName)
        {
            Point point;
            Console.WriteLine("Enter X coordinate "
                + "of " + pointName);
            point.X = Double.Parse(Console.In.ReadLine());
            Console.WriteLine("Enter Y coordinate "
                + "of " + pointName);
            point.Y = Double.Parse(Console.In.ReadLine());
            return point;
        }

        private void WaitForUserToClose()
        {
            Console.Read();
        }
    }
    public class Circle
    {
        private Point centerValue;
        private Point pointOnCircumferenceValue;
        public Point Center
        {
            get { return centerValue; }
            set { centerValue = value; }
        }
        public Point PointOnCircumference
        {
            get { return pointOnCircumferenceValue; }
            set { pointOnCircumferenceValue = value; }
        }

        public double CalculateCircumferenceLength()
        {
            QueryDatabase();
```

```
                return 2 * 3.1415 * CalculateRadius();
            }

            private double CalculateRadius()
            {
                return Math.Pow((Math.Pow((this.PointOnCircumference.X
                    - this.Center.X), 2) +
                    Math.Pow((this.PointOnCircumference.Y - this.Center.Y),
                    2)), (1 / 2));
            }

            private void QueryDatabase()
            {
                IDbConnection connection =
                    new SqlConnection("Data Source=TESLATEAM;" +
                        "Initial Catalog=RENTAWHEELS;"
                        + "User ID=RENTAWHEELS_LOGIN;"
                        + "Password=RENTAWHEELS_PASSWORD_123");
                connection.Open();
                IDbCommand command = new SqlCommand("SELECT GETDATE()");
                command.Connection = connection;
                IDataReader reader = command.ExecuteReader();
                reader.Read();
                reader.Close();
                connection.Close();
            }
        }
    }
```

After a few executions, you can see that the times for both samples are pretty close. On my machine, these values are in the neighborhood of 2.2 to 2.4 seconds. One small difference I could discern is that the best time for the unstructured sample was 1.9114800, whereas for the structured sample it was 2.0398497.

''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 also seem without enough weight, consisting of only a few members. It seems as if nothing ever happens in your code.

Having to manage a greater number of elements such as 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. In refactored code, however, it is expressed in a much cleaner, more structured manner.

''Refactoring Offers No Short-Term Benefits''

There is overwhelming consensus among refactoring converts that refactoring actually makes you program faster. So far, I do not know of any study that I could call upon in order to prove this, but my own experience tells me this is the case. It is only logical that this is so. Because you have a smaller quantity of code overall, less duplication, and a clearer picture, unless you are dealing with some trivial and unrealistically small-scale code, refactoring 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 of the way you code, and this is where refactoring enters the picture. Other team members or management might even be oblivious to the fact that from time to time you reach for the ''refactor'' option inside your IDE. There is nothing to prevent you from refactoring your code, regardless of the methodology your team might subscribe to.

The best results in refactoring are achieved if you adopt it in small steps, performing it regularly while you code. Some practices, such as strict code ownership or a waterfall process, can work 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.

''Refactoring Can Be Applied As a Separate Stage in the Development Process and Performed by a Separate Team''

This one is generally favored by managers. Being able to look at refactoring as a separate stage and placing it somewhere between the implementation and testing phases can suggest a false sense of control from a managerial point of view, where tasks and resources are often seen as bars on some Gantt chart that can be easily squeezed or moved around.

The truth is that in order to perform refactoring successfully, you should have a full understanding of the problem domain, and be aware of requirements, the design, and even implementation details. If you are not part of the team from the beginning — that is, you did not spend time working with clients, analyzing requirements, and thinking about the design — then you will have a hard time improving something constructed by the original team.

Following the model in which code can be refined *a posteriori*, akin to some substance in an industrial process, will generally bring few benefits. Without a good understanding of what the code is actually doing, refactoring that you can really perform with confidence will bring only small improvements. If you try to make any substantial change in such circumstances, the results will likely be quite the opposite of what is supposed to be achieved. Instead of making code relate better to the problem domain, you will probably make things worse and end up introducing bugs into your application.

Refactoring Can Work Just As Well Without Unit Tests

I imagine some simple refactorings can be performed even without unit testing in place. Refactoring tools and the compiler itself can offer a limited safety net that you can count on to catch some simple human mistakes. You can also test the code in traditional ways, using a debugger or performing functional tests, although these manual testing methods tend to be tedious and unreliable. Moreover, with refactoring, your code is more amenable to change than ever.

Avoid unnecessary problems by adding some NUnit tests to your project, and you will be able to test for errors with each small step you make. You will learn more about unit tests in Chapter 3. In the next section, you'll see how refactoring can be an indispensable tool in a team environment.

No Programmer Is an Island

A lot of the motivation for refactoring comes from one simple fact: Programmers work in teams. Programming in a team environment requires a major paradigm shift; not only does your code have to tell a computer what to do, it has to communicate your intentions to other members of your team.

Not every person in your organization will look inside source code, but a number of those will do so. Consider the typical programming ecosystem that revolves around source code, as shown in Figure 1-2:

- **Client programmer:** In today's modular and layered architectures, there is a good chance that you will work on a library that another programmer will (re)use in his or her project. For a client programmer, it is of paramount importance that your classes are designed and written so that they can be employed easily. They should be clearly related to the problem domain they were created to deal with. Simplicity and good naming strategies are critical for a client programmer. You will learn all the details about code naming strategies in Chapter 6.

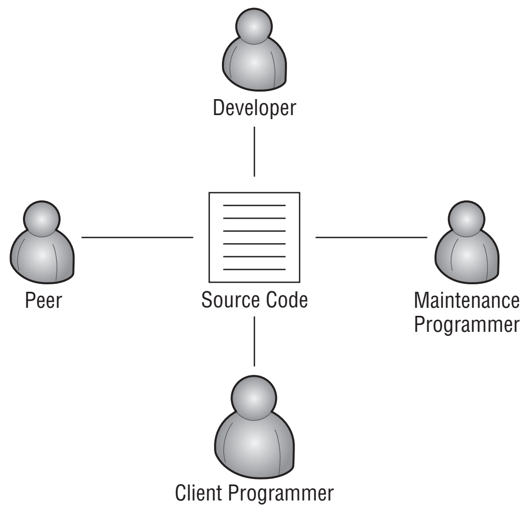


Figure 1-2

- **Maintenance programmer:** This is the person who will have to dig inside the internals of your code. To make this person's life easier, your code should be well structured, with small and simple methods and classes. The maintenance programmer also benefits from good naming strategies, and clean and well-formatted code.
- **Peer programmer:** Whether you work with a partner or go over your code with your peers in a code review session, it is your code that reflects your skills and professionalism. Good code earns you the respect of your colleagues.

Always keep others in mind when you write your code. This is not really altruism; it is a reciprocal interest for anyone who writes code. As you write code, you will use code written by others. The chances are good that you will have to modify code that you didn't create or review code that someone else has written.

Any organization involved with software will also employ people that do not write code. In the next section I will address some reasoning strategies that can be used to justify refactoring to managers and others.

Refactoring for Business People

Programming teams today have more and more liberty to organize themselves as they see fit, as long as this results in good productivity and a quality product. Such an approach is especially common in an agile software culture.

In a way, this is contradictory to a traditional approach to organization, whereby managers are not just enablers, but are also responsible for internal team organization and low-level task management. In such a more traditional environment, you might find yourself in a situation where you have to justify the work you put into refactoring your code. Remember that for someone not dealing with code, it might be quite difficult to grasp what benefits refactoring offers.

For the purpose of explaining the benefits of refactored code to a nontechnical audience, managers, and the like, you might find the following formula, which some authors use to represent the cost of software, useful:

$$\text{Cost(total)} = \text{Cost(develop)} + \text{Cost(maintain and evolve)}$$

According to many studies, and likely reflecting your own daily experience, the cost to maintain and evolve a system surpasses the cost to initially develop the system. Many times, for a programmer, there is no such thing as a clear line separating the development phase and the maintenance or evolution phase. For managers, release is more than an important milestone; after the release, the software is expected to start bringing in revenue. Therefore, bearing in mind that the benefits of refactoring are especially visible in the maintenance/evolution phase, the cost related to this phase can be further expressed as follows:

$$\text{Cost(maintain and evolve)} = \text{Cost(understand)} + \text{Cost(change)} + \text{Cost(test)} + \text{Cost(deploy)}$$

Refactoring will help you cut time and effort from each of these phases:

- ❑ Refactored code is written is such a manner that it is easier to understand, as it is written for comprehension. In addition, improved reuse will reduce the overall size of the code-base.
- ❑ Refactored code is easier to change. Less duplication means it is easier to discover the right place to make a change. Again, improved reuse reduces the overall size of the code-base, making this task easier to perform.
- ❑ Refactored code is easier to test thanks to readable code. Refactored code often includes ready unit tests that bring their own benefits to the refactoring process.
- ❑ Applications for which refactoring is put into practice often have better handled component-level dependencies, making them easier to deploy.

Now let's take a look at C# and how its modern language characteristics are well suited for the refactoring process.

C# and Refactoring

With the advent of the .NET platform, Microsoft decided it was time to begin with a clean slate — a new programming language. Created especially for .NET, C# is becoming the most popular language on the platform and today is in many cases the best choice for programming in .NET. C# has all the characteristics of a modern object-oriented programming language. Following are some of the reasons why C# is so popular:

- ❑ C-inspired syntax is immediately familiar to anyone with previous programming experience in C++, Java, C, or another language from the ''curly braces'' family.
- ❑ C# has an automatic memory management facility known as *garbage collector* that hugely simplifies the programming process and eliminates one of most problematic tasks in programming: reclaiming memory.
- ❑ C# is a strongly and statically typed language, meaning that a number of programming errors can be successfully detected during compilation.
- ❑ The inheritance hierarchy has a root class, `Object`, from which all classes are inherited.
- ❑ It has multiple-interface inheritance, meaning that a class can implement many interfaces, and it has single-implementation inheritance, meaning it can inherit only a single class.
- ❑ C# attributes can add powerful declarative capabilities to C# code.

Less than 10 years since it has appeared, C# has evolved significantly compared to its first version. While the language is still relatively simple and consistent, it has successfully integrated elements of other programming paradigms, such as functional, declarative, and dynamic programming. Here are some of the advances that C# has been endowed with:

- ❑ C# generics have further improved the C# type system.
- ❑ Extension methods offer an additional level of flexibility to programming in C#.
- ❑ Language Integrated Query (LINQ) enables you to query different sources of data in a uniform manner.

No wonder than that C# was the first language to obtain refactoring support in .NET. C# is still the only programming language with out-of-the-box support for refactoring in Visual Studio. There is also a healthy third-party refactoring tools market for C#, as you will see in Chapter 3.

C# is a code-loving programmer's language of choice for the .NET platform. The fundamental philosophy of power and simplicity that is characteristic of C# is naturally complemented with a refactoring approach to programming. Refactoring will sharpen your programming skills and help you climb another step on the skills ladder of the expert programmer.

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 remove a lot of the drudgery and complexity from refactoring, making it easily accessible and applicable.

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

Refactoring has a special significance in a team environment. Programming as an act of communication with other programmers instead of a programmer-to-computer monologue will require significant changes in the way you work and write code.

C# is a powerful, consistent language with an expressive syntax. Code written in such a language can especially benefit from refactoring, helping a programmer exploit the language and platform capabilities to their fullest extent.

Now it's time to see some of this in practice. In the next chapter you are going to see refactoring at work.