Refactoring with Microsoft Visual Studio 2010

Peter Ritchie

Chapter No.6
"Improving Class Quality"
In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.6 "Improving Class Quality"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Peter Ritchie is a software development consultant. Peter is president of Peter Ritchie Inc. Software Consulting Co., a software consulting company in Canada's National Capital Region specializing in Windows-based software development management, process, and implementation consulting. Peter has worked with such clients as Mitel, Nortel, Passport Canada, and Innovapost, from mentoring to architecture to implementation. Peter's range of experience ranges from designing and implementing simple stand-alone applications to architecting n-tier applications spanning dozens of computers; from C++ to C#.
Refactoring with Microsoft Visual Studio 2010

This book introduces the reader to improving a software system's design through refactoring.

It begins with simple refactorings and works its way through complex refactorings by building on the simple refactorings. You will learn how to focus changing the design of their software system and how to prioritize refactorings—including how to use various Visual Studio features to focus and prioritize design changes. The book also covers how to ensure quality in light of seemingly drastic changes to a software system. You will also be able to apply standard established principles and patterns as part of the refactoring effort with the help of this book.

What This Book Covers

Chapter 1, Introducing Refactoring, describes what refactoring is, its importance, and its priority in the software development effort. Comparison to re-writing and what "Technical Debt" is and how refactoring can be used to pay down technical debt is covered in this chapter.

Chapter 2, Improving Code Readability, begins detailing the refactorings built in to Visual Studio and how they can make code more readable. Code smells are introduced, and which code smells apply to readability, and how to detect and refactor them are detailed in this chapter.

Chapter 3, Improving Code Maintainability, continues to detail the refactorings built in to Visual Studio and how they can make code more maintainable. Code smells that apply to maintainability, how to detect and refactor them are detailed in this chapter. The importance of unit testing is covered in this chapter.

Chapter 4, Improving code navigation, continues with simple refactorings and how code can be refactored to improve its navigability in general and takes into account Visual Studio code navigation abilities.

Chapter 5, Improving design correctness, begins detailing complex refactorings. Design principles such as Liskov Substitution and Composition over Inheritance are introduced and how to perform refactorings related to these principles is covered in this chapter.

Chapter 6, Improving class quality, introduces code quality metrics like cohesion and coupling. Principles related to cohesion and coupling are introduced and refactorings that increase cohesion and decrease coupling are covered in this chapter.

For More Information:
Chapter 7, Refactoring to loosely-coupled, expands on coupling from the previous chapter and drills-down on loosely-coupled design. Principles related to loosely-coupled are introduced and complex refactorings related to loosening coupling are covered in this chapter.

Chapter 8, Refactoring to layers, continues with more complex refactorings that involve layered architectures. Typical layers, Model View Presenter, and Repository patterns and how and when to refactor to them are also detailed in this chapter.

Chapter 9, Improving architectural behavior, details complex refactorings to improve architectural behavior. Design behavior patterns, when and how to refactor to them are detailed in this chapter.

Chapter 10, Improving architectural structure, continues with architectural-related complex refactorings. Object-Relational Mapping (ORM) and refactoring Repository implementations are included in this chapter.

Chapter 11, Ensuring Quality with Unit Testing, details the importance of unit testing. How unit testing applies to refactoring, examples of unit testing to support the refactoring effort, and legacy code are also detailed in this chapter.

For More Information:  
Larry Constantine is attributed with the creation of systematic measurement of software quality. In the mid-to-late seventies, Larry Constantine (and Ed Yourdon) attributed several things to the quality of software code. Under the umbrella of structured design, among those attributes of quality software code were cohesion and coupling. At the time they associated quality with generality, flexibility, and reliability. We’re going to concentrate on generality and flexibility and how cohesion and coupling can be applied to increase code quality.

For More Information:  

Cohesion
Cohesion applies to many different disciplines. Cohesion in physics relates to the force that keeps molecules integrated and united and that makes the molecule what it is. A highly cohesive molecule is one that tends to remain autonomous and not adhere to or blend with other molecules. In geology, cohesion is the strength of substances to resist being broken apart. In linguistics, it's the degree to which text is related. Text whose content relates to the same subject is cohesive.
Cohesion in software is very similar to cohesion elsewhere. A cohesive block of code is a block of code that relates well together and has the ability to remain together as a unit. Object-oriented programming brings distinct cohesive abilities. All programming languages have certain cohesive abilities, such as their ability to group code in modules, source files, functions, and so on. A programming language's ability to define physical boundaries enables cohesiveness. A module, for example, defines a specific boundary for which some content is retained and other content is repelled. Code from one module can use code from another module, but only in specific and defined ways—usually independent of language syntax. All code within a module has innate cohesion: their relation amongst themselves as being contained within the module.

Any rule, principle, guideline, or practice needs to be implemented thoughtfully. This text isn't a manual on how you must perform your refactoring; it's a description of several types of refactoring and their impetus. By the same token, this text doesn't set out to prove the benefits of any particular rule, principle, guideline, or practice. "Mileage may vary" and incorrect usage will often negate most, if not all, benefits. I'll leave it as an exercise for the reader to find the research that "proves" the benefits of any particular rule, principle, guideline, or practice. This text assumes the generally accepted benefits of various principles and practices, including cohesion, as an indicator of quality. If you decide that the benefits aren't outweighing the costs, it's up to you to decide not to implement that principle.

**Class cohesion**

Object-orientation brings extra cohesive abilities to the programmer. The programmer has the ability to relate code together within a class. Other code can use the code within a class, but only through its defined boundaries (the class's methods and properties).

In object-oriented design, cohesion is generally much more than simply code contained within a class. Object-oriented cohesiveness goes beyond the physical relation of code within a class and deals with the relation of meaning of the code within a class.

Object-oriented language syntax allows the programmer to freely relate code to other code through a class definition, but this doesn't mean that code is cohesive. For example, let's revisit our Invoice class so far.

```csharp
/// <summary>
/// Invoice class to encapsulate invoice line items
/// and drawing
/// </summary>
public class Invoice
```
private IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy;
public Invoice(IEnumerable<InvoiceLineItem> invoiceLineItems, 
    IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy) 
{
    InvoiceLineItems = new 
        List<InvoiceLineItem>(invoiceLineItems);
    this.invoiceGrandTotalStrategy = invoiceGrandTotalStrategy;
}
private List<InvoiceLineItem> InvoiceLineItems 
{
    get;
    set;
}
public void GenerateReadableInvoice(Graphics graphics) 
{
    graphics.DrawString(HeaderText,
        HeaderFont,
        HeaderBrush,
        HeaderLocation);
    float invoiceSubTotal = 0;
    PointF currentLineItemLocation = LineItemLocation;
    foreach (InvoiceLineItem invoiceLineItem in 
        InvoiceLineItems)
    {
        float lineItemSubTotal = 
            CalculateLineItemSubTotal(invoiceLineItem);
        graphics.DrawString(invoiceLineItem.Description,
            InvoiceBodyFont,
            InvoiceBodyBrush,
            currentLineItemLocation);
        currentLineItemLocation.Y +=
            InvoiceBodyFont.GetHeight(graphics);
        invoiceSubTotal += lineItemSubTotal;
    }
    float invoiceTotalTax = 
        CalculateInvoiceTotalTax(invoiceSubTotal);
    float invoiceGrandTotal = 
        invoiceGrandTotalStrategy.CalculateGrandTotal(

For More Information:
Improving Class Quality

```csharp
invoiceSubTotal,
invoiceTotalTax);
CalculateInvoiceGrandTotal(invoiceSubTotal,
invoiceTotalTax);
graphics.DrawString(String.Format(
  "Invoice SubTotal: {0}",
invoiceGrandTotal - invoiceTotalTax),
InvoiceBodyFont, InvoiceBodyBrush,
InvoiceSubTotalLocation);
graphics.DrawString(String.Format("Total Tax: {0}",
invoiceTotalTax), InvoiceBodyFont,
InvoiceBodyBrush, InvoiceTaxLocation);
graphics.DrawString(String.Format("Invoice Grand Total: {0}",
invoiceGrandTotal), InvoiceBodyFont,
InvoiceBodyBrush, InvoiceGrandTotalLocation);
graphics.DrawString(FooterText,
FooterFont,
FooterBrush,
FooterLocation);
}

public static float CalculateInvoiceGrandTotal(
  float invoiceSubTotal, float invoiceTotalTax)
{
  float invoiceGrandTotal = invoiceTotalTax +
  invoiceSubTotal;
  return invoiceGrandTotal;
}

public float CalculateInvoiceTotalTax(
  float invoiceSubTotal)
{
  float invoiceTotalTax =
  (float)((Decimal)invoiceSubTotal * (Decimal)TaxRate);
  return invoiceTotalTax;
}

public static float
CalculateLineItemSubTotal(
  InvoiceLineItem invoiceLineItem)
{
  float lineItemSubTotal =
  (float)((decimal)(invoiceLineItem.Price - invoiceLineItem.Discount)
```

For More Information:
We have an operational Invoice class. It does some things, and they work. But, our Invoice class isn't very cohesive. The Invoice class has distinct groups of fields. Some are for the state of an invoice and some are for generating a readable invoice. Methods that deal with the behavior and attributes of an invoice don't use the fields that deal with generating a readable invoice.

Our Invoice class is implementing two distinct tasks: managing invoice state and generating a readable invoice. The data required to generate a readable invoice (over and above the data shown on an invoice) isn't used by Invoice when not generating a readable invoice.

Our Invoice class can be said to have multiple responsibilities: the responsibility of managing state and the responsibility of generating a readable invoice. What makes an invoice an invoice may be fairly stable; we may occasionally need to add, remove, or change fields that store data contained in an invoice. But, the act of displaying a readable invoice may be less stable: it may change quite frequently. Worse still, the act of displaying a readable invoice may depend on the platform it is running on.
The Single Responsibility Principle

In terms of focusing refactoring efforts towards cohesiveness of a class, the Single Responsibility Principle (SRP) gives us guidance on what a particular class should or should not do. The Single Responsibility Principle states that "there should never be more than one reason for a class to change". In the case of our invoice class there's a couple of reasons why we'd need to change the class:

- The way an invoice is displayed needs to change.
- The data that is contained in an invoice needs to change.

The stability of each responsibility shouldn't need to depend on the other. That is, any change to the Invoice class affects stability of the whole class. If I often need to change the way an invoice renders a displayable invoice, all of Invoice is instable—including its responsibility to the data an invoice contains.

The Single Responsibility Principle's focus is fairly narrow: class responsibility. A novice may simply accept that scope and use it only to focus cohesion efforts at the class level. The fact is that the Single Responsibility Principle is applicable at almost all levels of software design, including method, namespace, module/assembly, and process; that is, a method could implement too much responsibility, the types within an assembly or namespace could have unrelated responsibilities, and the responsibilities of a given process might not be focused effectively.

Simply saying "single responsibility" or detailing that something has too many responsibilities is simple. But the actual act of defining what a responsibility is can be very subjective and subtle. Refactoring towards Single Responsibility can take some time and some work to get right. Let's see how we can improve quality through better cohesion and the principle of single responsibility.

Refactoring classes with low-cohesion

Clearly single responsibility (and the heading of this section) suggests that we should refactor Invoice into multiple classes. But, how do we do that given our current scenario? The designer of this class has obviously thought that the functionality of Invoice included rendering a readable invoice, so how do we make a clean separation? Fortunately, the lack of cohesiveness gives us our separation points. What makes an invoice an invoice doesn't include those fields that deal solely with rendering a readable invoice. Fields like HeaderFont, HeaderBrush,HeaderText, HeaderLocation, FooterFont, FooterBrush, FooterText, FooterLocation, InvoiceBodyFont, InvoiceBodyBrush, LineItemLocation, InvoiceBustTotalLocation, InvoiceTaxLocation, and InvoiceGrandTotalLocation all deal with just the rendering responsibility.
The `Invoice` class is modeling a real-world invoice. When you hold an invoice in your hand or view it on the screen, it's already rendered. In the real-world we'd never think that a responsibility of rendering an invoice would be a responsibility of the invoice itself.

We know we want to retain our original `Invoice` class and we want to move the rendering responsibility to a new class. This new class will encapsulate the responsibility of rendering an invoice. Since this new class will take an `Invoice` object and help another class produce something useful, we can consider this an invoice rendering service.

In order to refactor our existing `Invoice` class to a new Invoice rendering service, we start with a new `InvoiceRenderingService` class and move the `HeaderFont`, `HeaderText`, `HeaderLocation`, `FooterFont`, `FooterBrush`, `FooterText`, `FooterLocation`, `InvoiceBodyFont`, `InvoiceBodyBrush`, `LineItemLocation`, `InvoiceBustTotalLocation`, `InvoiceTaxLocation`, and `InvoiceGrandTotalLocation` fields to the `InvoiceRenderingService`. Next, we move the `GenerateReadableInvoice` method to the `InvoiceRenderingService`. At this point, we basically have a functional class, but since the `InvoiceRenderingService` method was on the `Invoice` classes, the other properties that the `GenerateReadableInvoice` uses need an `Invoice` object reference—effectively changing it from "this" to a parameter to the `GenerateReadableInvoice` method. Since the original `Invoice` class was never expected to be used externally like this, we need to add a `CalculateGrandTotal` method that delegates to the `invoiceGrandTotalStrategy` object. The result is something like the following:

```csharp
/// <summary>
/// Encapsulates a service to render an invoice to a Graphics device.
/// </summary>
public class InvoiceRenderingService
{
    public void GenerateReadableInvoice(Invoice invoice, Graphics graphics)
    {
        graphics.DrawString(HeaderText, HeaderFont, HeaderBrush, HeaderLocation);
        float invoiceSubTotal = 0;
        PointF currentLineItemLocation = LineItemLocation;
        foreach (InvoiceLineItem invoiceLineItem in invoice.InvoiceLineItems)
        {
            // For More Information:
```
float lineItemSubTotal =
    Invoice.CalculateLineItemSubTotal(
    invoiceLineItem);

graphics.DrawString(invoiceLineItem.Description,
    InvoiceBodyFont,
    InvoiceBodyBrush,
    currentLineItemLocation);

    currentLineItemLocation.Y +=
        InvoiceBodyFont.GetHeight(graphics);
    invoiceSubTotal += lineItemSubTotal;
}

float invoiceTotalTax =
    invoice.CalculateInvoiceTotalTax(
    invoiceSubTotal);

float invoiceGrandTotal =
    invoice.CalculateGrandTotal(
    invoiceSubTotal,
    invoiceTotalTax);

Invoice.CalculateInvoiceGrandTotal(invoiceSubTotal,
    invoiceTotalTax);

graphics.DrawString(String.Format(
    "Invoice SubTotal: {0}",
    invoiceGrandTotal - invoiceTotalTax),
    InvoiceBodyFont, InvoiceBodyBrush,
    InvoiceSubTotalLocation);

graphics.DrawString(String.Format("Total Tax: {0} ",
    invoiceTotalTax), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceTaxLocation);

graphics.DrawString(String.Format(
    "Invoice Grand Total: {0}" ,
    invoiceGrandTotal), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceGrandTotalLocation);

graphics.DrawString(FooterText,
    FooterFont,
    FooterBrush,
    FooterLocation);
}

public string HeaderText { get; set; }

public Font HeaderFont { get; set; }

public Brush HeaderBrush { get; set; }

public RectangleF HeaderLocation { get; set; }
Alternatively, the whole use of invoiceGrandTotalStrategy can be moved into the InvoiceRenderingService—which is a better design decision.

**Detecting classes with low-cohesion**

So, we've seen a fairly simple example of making a not-so-cohesive class into two more-cohesive classes; but, one of the tricky parts of refactoring away classes with low cohesion is finding them. How do we find classes with low-cohesion?

Fortunately, many people have put time and effort over the years into defining what it means for a class to be cohesive. There have been various metrics researched and created over the years to define cohesion in classes. The most popular metric is **Lack of Cohesion of Methods (LCOM)**. Lack of Cohesion of Methods measures the degree to which all methods use all fields. The more segregated field usage is amongst methods of a class, the higher the Lack of Cohesion of Methods metric of the class will be. Lack of Cohesion of Methods is a measure of the entire class, so it won't point out where the class is not cohesive or indicate where the responsibilities can be separated.
Improving Class Quality

Lack of Cohesion of Methods is a measurement of the degree to which fields are used by all methods of a class. Perfection as defined by Lack of Cohesion of Methods is that every method uses every field in the class. Clearly not every class will do this (and arguably this will hardly ever happen); so, Lack of Cohesion of Methods is a metric, as most metrics are, that requires analysis and thought before attempting to act upon its value. LCOM is a value between 0 and 1, inclusive. A measure of 0 means every method uses every field. The higher the value, the less cohesive the class; the lower the value, the more cohesive the class. A typical acceptable range is 0 to 0.8. But, there's no hard-and-fast definition of specific value that represents cohesive; just because, for example, a class has an LCOM value of 0.9, that doesn't mean it can or should be broken up into multiple classes. Lack of Cohesion of Methods values should be used as a method of prioritizing cohesion refactoring work by focusing on classes with higher LCOM values before other classes (with lower LCOM values).

In the case of our Invoice class, it's apparent that its LCOM value does mean it can be split into multiple classes as we detailed in the previous section.

Method cohesion

Methods on their own can also suffer from low-cohesion. One symptom of a method with low-cohesion is size. Methods with low-cohesion are often large. A large method is generally doing more than it needs to. As we saw with the Large Method Code Smell in Chapter 2, refactoring is as simple as breaking the method up into multiple methods. Each method should take on a single responsibility.

Another symptom of a class that suffers from low-cohesion is one that has many parameters. A method that takes many parameters is probably doing too many things. Unfortunately, how to refactor depends on what is trying to be accomplished. One example of too many arguments is often constructors. For example:

```csharp
/// <summary>
/// Example of a class with low-cohesion
/// </summary>
public class Invoice
{
    public string GivenName { get; private set; }
    public string SurName { get; private set; }
    public string Street { get; private set; }
    public string City { get; private set; }
    public string Province { get; private set; }
    public string Country { get; private set; }
    public string PostalCode { get; private set; }
    public Invoice(IEnumerable<InvoiceLineItem>
```
invoiceLineItems, string givenName, 
string surName, string street, 
string city, string province, 
string country, string postalCode, 
Func<float, float, float> calculateGrandTotalCallback)
{
    GivenName = givenName;
    SurName = surName;
    Street = street;
    City = city;
    Province = province;
    Country = country;
    PostalCode = postalCode;
}
    //...
}

The Invoice class has taken on the extra responsibility of managing customer 
information (given name, surname, street, and so on) and thus one of its constructors 
has many parameters.

**Refactoring methods with low-cohesion**

Invoice could be refactored to make the constructor (and the class, for that matter) 
more cohesive by encapsulating customer information into a Customer class, 
encapsulating address information into an Address class and the Invoice class, 
and accepting a Customer parameter to the constructor that initializes a Customer 
property. The resulting refactoring would look like the following:

```csharp
/// <summary>
/// Customer shape to encapsulate
/// name and address
/// </summary>
public class Customer
{
    public String FirstName { get; private set; }
    public String LastName { get; private set; }
    public Address Address { get; private set; }
    public Customer(string firstName, string lastName, 
        Address address)
    {
        FirstName = firstName;
        LastName = lastName;
        Address = address;
```
For More Information:
We now have an Invoice class that makes use of the Customer class, and the Customer class manages the customer (including address) responsibility.

This provides a much more cohesive constructor and encapsulates customer and address information so that code to initialize that type of information does not have to be repeated. We can use the code that used to be contained in Invoice anywhere else with Customer without having to repeat that code within the other classes that need it.

In other cases, a method with too many parameters is simply trying to do too many things.

```csharp
/// <summary>
/// Invoice that uses callback for
/// grand total calculation
/// </summary>
public class Invoice
{
    private Func<float, float, float> calculateGrandTotalCallback;
    public Invoice(IEnumerable<InvoiceLineItem> invoiceLineItems,
                    Func<float, float, float> calculateGrandTotalCallback)
    {
        InvoiceLineItems = new
                        List<InvoiceLineItem>(invoiceLineItems);
        this.calculateGrandTotalCallback =
                        calculateGrandTotalCallback;
    }
    public List<InvoiceLineItem> InvoiceLineItems { get; set; }
    public float TaxRate { get; set; }
    public bool CalculateTotals(out float invoiceSubTotal, out float invoiceTotalTax, out float invoiceGrandTotal)
    {
        invoiceSubTotal = 0;
        foreach (InvoiceLineItem invoiceLineItem in
                        InvoiceLineItems)
        {
            invoiceSubTotal +=
                        (float)((decimal)(invoiceLineItem.Price
                        - invoiceLineItem.Discount)
                        * (decimal)invoiceLineItem.Quantity);
        }
    }
}
```

For More Information:
invoiceTotalTax = (float)((Decimal)invoiceSubTotal * 
(Decimal)TaxRate);
invoiceGrandTotal = 
calculateGrandTotalCallback(invoiceTotalTax, 
 invoiceTotalTax);
return true;
}
//...
}

CalculateTotals is a straightforward example that effectively returns three floating-point values from a method: calculating three values and assigning them to output parameters. What is common with methods with "multiple return values" is to return a Boolean result. This Boolean result is always true—and an indication that there's a design issue. In this example, the designer thought it would be useful to be able to query all the total values at the same time. The problem with this is that if only one value is needed, then the caller has to create a couple of dummy variables to needlessly store the unwanted values.

In order to refactor this, we simply need to perform Extract Method refactorings to split up the method into three methods: CalculateInvoiceGrandTotal, CalculateInvoiceTotalTax, and CalculateLineItemSubtotal.

/// <summary>
/// Example of method with too many parameters
/// </summary>
public class Invoice
{
private Func<float, float, float>
calculateGrandTotalCallback;

public Invoice(IEnumerable<InvoiceLineItem>
invoiceLineItems,
 Func<float, float, float> calculateGrandTotalCallback)
{
InvoiceLineItems = new
 List<InvoiceLineItem>(invoiceLineItems);
this.calculateGrandTotalCallback =
calculateGrandTotalCallback;
}

public List<InvoiceLineItem> InvoiceLineItems { get; set; }
public float TaxRate { get; set; }
public bool CalculateTotals(out float invoiceSubTotal,
 out float invoiceTotalTax, out float invoiceGrandTotal)
We've now introduced three methods. CalculateInvoiceGrandTotal extracts the portion of code that calculates the invoice grand total and calculates the grand total based on an invoice subtotal and invoice total tax. CalculateInvoiceTotalTax extracts the portion of code that calculates the total tax based on an invoice subtotal. CalculateInvoiceSubTotal extracts the portion of code that sums the line item subtotals. Each of these methods takes on a single responsibility from the original CalculateTotals—which had taken on multiple responsibilities.

Another example is method, which is both a query and a modifier. Sometimes, this is easy to detect, as the word And is usually in the method name: ChangeTaxAndCalculateGrandTotal. Sometimes, it's harder to detect and requires a bit of analysis of the method body. For example:

```csharp
/// <summary>
/// Example Command AND Query method
/// </summary>
/// <param name="taxRate"></param>
/// <returns></returns>
public float CalculateGrandTotal(float taxRate)
{
    TaxRate = taxRate;
    float invoiceSubTotal = 0;
    foreach (InvoiceLineItem invoiceLineItem in InvoiceLineItems)
    {
        invoiceSubTotal +=
            (float)((decimal)(invoiceLineItem.Price
             - invoiceLineItem.Discount)
             * (decimal)invoiceLineItem.Quantity);
    }
    invoiceTotalTax = (float)((Decimal)invoiceSubTotal * 
        (Decimal)TaxRate);
    invoiceGrandTotal =
        calculateGrandTotalCallback(invoiceTotalTax,
            invoiceTotalTax);
    return true;
}
```
{ invoiceSubTotal +=
  (float)((decimal)(invoiceLineItem.Price
   - invoiceLineItem.Discount)
  * (decimal)invoiceLineItem.Quantity);
}
float invoiceTotalTax = (float)((Decimal)invoiceSubTotal *
(Decimal)TaxRate);
return calculateGrandTotalCallback(invoiceTotalTax,
  invoiceTotalTax);
}

This method first sets the TaxRate property, then proceeds to calculate the grand total based on the TaxRate property. In the case of this method, it’s not clear from its signature (unlike ChangeTaxAndCalculateGrandTotal) that it is both a Command and a Query.

**Command**: A method that performs an action on an object; generally modifying state.

**Query**: A method or property that returns data to the caller.

In some circles, a Command is also known as a Modifier.

This can be refactored by performing the Separate Query from Modifier refactoring.

In our case, this is just a matter of removing the taxRate parameter from the method and removing the assignment to the TaxRate property, similar to the following code:

```csharp
/// <summary>
/// Refactored to Query and not Command
/// </summary>
/// <returns></returns>
public float CalculateGrandTotal()
{
  float invoiceSubTotal = 0;
  foreach (InvoiceLineItem invoiceLineItem in InvoiceLineItems)
  {
    invoiceSubTotal +=
    (float)((decimal)(invoiceLineItem.Price
      - invoiceLineItem.Discount)
    * (decimal)invoiceLineItem.Quantity);
  }
  float invoiceTotalTax = (float)((Decimal)invoiceSubTotal *
    (Decimal)TaxRate);
  return calculateGrandTotalCallback(invoiceTotalTax,
    invoiceTotalTax);
}
```
Code that needs to calculate the grand total with a different tax rate should simply use the TaxRate property before calling CalculateGrandTotal. For example, instead of this:

```csharp
float grandTotal = invoice.CalculateGrandTotal(.12f);
```

We'd refactor to this:

```csharp
TaxRate = .12f;
float grandTotal = invoice.CalculateGrandTotal();
```

In a more complex Separate Query From Modifier refactoring, a new method or property would have to be created to separate the Query, and the method performing the modification would remove the code that modifies state and is renamed (for example, ChangeTaxRateAndCalculateGrandTotal to CalculateGrandTotal) to be clear that it isn't modifying state. Calls to the method would have to be changed to the new query method, and a call to the modifier added before the call to the query.

### Namespace cohesion

As with any logical grouping of code, what is contained within the grouping may or may not be related. Syntactically, a namespace can contain any number of classes with any number of purposes related to any number of things. Grouping in a namespace is for the programmer; if it doesn't add any value, there's not much point to using it. Classes within a namespace should be related to one another in a particular way that adds value.

### Refactoring namespaces with low-cohesion

Unfortunately, there isn't a built-in way to move a class from one namespace to another. You can rename a namespace, but if there is more than one class within the namespace, you "move" all the classes to a new or existing namespace.
Let's say we have a class in the **Invoicing** namespace, and we want to move it to the **Invoicing.Domain** namespace because this class represents a fundamental domain entity and locating it in the **Domain** namespace will mean it will be cohesive with the other members of the **Domain** namespace.

```csharp
namespace Invoicing
{
    /// <summary>
    /// Address shape to encapsulate
    /// western-style addresses
    /// </summary>
    public class Address
    {
        public string Street { get; private set; }
        public string City { get; private set; }
        public string Province { get; private set; }
        public string Country { get; private set; }
        public string PostalCode { get; private set; }
        
        public Address(string street, string city, string province, string country, string postalCode)
        {
            Street = street;
            City = city;
            Province = province;
            Country = country;
            PostalCode = postalCode;
        }
    }
}
```

In order to perform a **Move to Another Namespace** refactoring, right-click the namespace name **Invoicing**, and select **Refactor\Rename**... then enter "**Invoicing.Domain**". This effectively "moves" the **Address** class to a new namespace, the **Invoicing.Domain** namespace. This results in the following:

```csharp
namespace Invoicing.Domain
{
    /// <summary>
    /// Address shape to encapsulate
    /// western-style addresses
    /// </summary>
    public class Address
    {
        public string Street { get; private set; }
        public string City { get; private set; }
    }
}
```

For More Information:
public string Province { get; private set; }
public string Country { get; private set; }
public string PostalCode { get; private set; }

public Address(string street, string city,
        string province, string country, string postalCode)
{
    Street = street;
    City = city;
    Province = province;
    Country = country;
    PostalCode = postalCode;
}
}

The only "heavy lifting" at this point you'll have to do is move the file this class lives in from one directory to another (if you’re synchronizing namespace names with directory names). This can be accomplished by dragging and dropping the file in the Solution Explorer.

1. If your namespace has many classes in it and you don't want all the classes to be moved, you'll have to manually perform the move:
2. Use Find All References to find all references to Address.
3. Change the namespace from Invoicing to Invoicing.Domain in Address.cs.
4. For each entry in the Find Symbol Results, double-click.
5. Add using directive for Invoicing.Domain.
6. Optionally move Address.cs to another folder with drag/drop in Solution Explorer.

Assembly cohesion
Assembly cohesion can be a bit of a red herring—it diverts attention away from assemblies' inherent features as a deployment strategy. Breaking up a solution into multiple assemblies simply to give related types a cohesive place to live is not the intention here. Assemblies are a deployment strategy; assemblies may need to exist for specific reasons unrelated to cohesion. For example, if two groups of different code need to be executed in separate processes or on separate computers, they need to live in different assemblies regardless of whether their packaging together would be more cohesive.
Improving Class Quality

If you're restructuring a system at the assembly level, you've got some pretty specific needs for what is contained in what assembly. That's not to say you won't have multiple assemblies per deployment platform. You may want to have assemblies that are shared amongst multiple applications, and they may need to be highly cohesive. .NET is a good example of this. .NET 2.0 deployed a certain set of assemblies. .NET 3.0 added types and features, but they're primarily added to assemblies new to .NET 3.0 to avoid changing existing binaries. For example, types in the System.Data.Linq namespace could have been included the System.Data.dll assembly and that assembly would have still been cohesive. But, because of the deployment issues (where System.Data.dll shouldn't be modified for anything other than show-stopper bugs) added, LINQ types and features were added to a new assembly: System.Data.Linq.dll.

Refactoring assemblies

When approaching refactoring assemblies by moving classes from one to another, the best starting point is to have all the projects associated with those assemblies in one Visual Studio® solution. Often, systems already have this type of organization. Larger projects may have to avoid this specific organization for performance and usability reasons within Visual Studio®. A temporary solution that contains these projects, in these cases, is the recommended place to start. When dealing with many project files, this may cause a bit of grief creating and loading the solution, but it will give a huge payoff if you want to move more than a couple of classes.

When approaching performing the Move Type to new Namespace from a single solution, performing the refactoring within Visual Studio® becomes very simple. Moving a class from one project to another becomes a simple process of selecting it in the Solution Explorer and dragging it to another project folder and dropping it while holding the Shift key down. Holding the Shift key down while dropping causes a move to occur instead of a copy. Once copied, the new file should be edited to change the original namespace to the destination namespace.

Coupling

Coupling is the degree to which two things are related. Coupling and cohesion go hand in hand. Something that is highly-cohesive generally has low coupling. Coupling is a form of dependency.

For More Information:
There are many different types of coupling when we're talking about software design and development. The effort here isn't to make a decoupled design, it's to change the coupling. At some level, one piece of code is coupled to every other piece of code in the system—there is rarely any change you can make to change that. Some code is more highly-coupled to other code and uses the other code directly. Some code is more loosely-coupled and uses other code indirectly. Efforts at refactoring towards a more loosely-coupled design are about the degree to which coupling has been made indirect.

Code can be coupled to other code by a shared data format (external coupling). Code can be coupled to other code by the fact that it results in the execution of other code (control coupling). Code can be coupled to other code by the fact that it results in executing other code by way of an abstract interface (message coupling). Code can be coupled to other code by the fact that they share data, usually in the form of parameters (data coupling). Code can be coupled to other code by the fact that it has a subclass relationship (subclass coupling). Code can also be coupled to other code by that fact that it directly references a type's public interface (content coupling). The direction and degree to which a type is coupled can also help focus our refactoring efforts. Afferent coupling is the degree to which a type is depended upon (inward coupling). Efferent coupling is the degree to which a type depends on other types (outward coupling). High afferent coupling can indicate that a type has too many responsibilities. It's trying to be everything to everyone and thus being used by everyone. High efferent coupling could indicate a type is very dependant. This becomes an issue when the types the class depends upon are in many different assemblies, suggesting a cohesion issue at the assembly layer.

Highly-coupled software is generally accepted to exhibit certain traits, and it can be hard to change. It's like pulling on the thread of a sweater; there are so many dependencies it's impossible to predict how much code will need to be modified in order to make a seemingly simple change. Highly-coupled code is also very rigid. It's hard to move or hard not to duplicate it outside its current context. It carries a lot of baggage (dependencies) with it that need to move with it as a unit. It's one thing to move the code you want to move, it's exponentially harder to move all its dependencies.

While good object-oriented design often promotes high cohesion, loosely coupled design and structure can easily fall to the wayside.

**Refactoring subclass coupling**

Refactoring subclass coupling involves removing the inheritance relationship between two classes. We discussed Composition over Inheritance in Chapter 5 and detailed how to refactor from inheritance to composition.
Refactoring content coupling

Content coupling is one class directly referencing another. There are several tactics for refactoring away this coupling. Depending on the situation, one tactic may be more applicable than another. One tactic is to use interface-based design and remove the coupling to the content of the class and replace it with coupling to an interface that the other class now implements. Another tactic is to replace method calls into the other class with delegate invocations. A final tactic is to use events instead of direct method calls.

For any particular refactoring, a combination of these tactics may be the best solution. You may find that despite a one-to-one coupling between two classes, it's more appropriate to use a combination of tactics to refactor away the content coupling.

Interface-based design

If you're already coupled to a particular class, replacing use of that class with an interface and having the other class implement that interface is the easiest way to change the coupling between two classes. This reduces coupling from content coupling to a more loosely coupled message coupling.

If the requirements of the other class are very complex or a series of members must come from a single source, using interfaces is often the best solution. Having to hook up several delegates or several events becomes tedious and error prone when a single reference to an object that implements a particular interface is so simple. Imagine if implementing a Windows Form wasn't as simple as deriving from Form and having to register a number of delegates or events.

If you find that implementers of the interface would find default or base implementation for them to be useful, implementing that interface may best be done with an abstract class.

Our Invoice class is a good example of something that can be more loosely coupled through interface-based design. It currently implements the calculation of grand totals through interface-based design and the strategy pattern (see Chapter 9). This could have easily been implemented through direct use of a particular class. For example:

```csharp
/// <summary>
/// Service to encapsulate calculation of
/// grand totals.
/// </summary>
public class InvoiceGrandTotalService
{
```
public float CalculateGrandTotal(float invoiceSubTotal, float invoiceTotalTax)
{
    return invoiceSubTotal + invoiceTotalTax;
}

/// <summary>
/// Invoice class that uses
/// <seealso cref="InvoiceGrandTotalService"/>
/// </summary>
public class Invoice
{
    InvoiceGrandTotalService invoiceGrandTotalService =
        new InvoiceGrandTotalService();
    public List<InvoiceLineItem> InvoiceLineItems { get; set; }
    public Invoice(IEnumerable<InvoiceLineItem> invoiceLineItems)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(invoiceLineItems);
    }
    public float CalculateGrandTotal(float invoiceSubTotal, float invoiceTotalTax)
    {
        return invoiceGrandTotalService.CalculateGrandTotal(
            invoiceSubTotal, invoiceTotalTax);
    }
    //...
}

In this example, we’ve created the InvoiceGrandTotalService class that contains the CalculateGrandTotal method. We then instantiate this class in the Invoice class and make reference to it in the CalculateGrandTotal method.
We’ve given away the surprise with this refactoring. We’re obviously going to replace direct use of the class with an interface. Since we essentially need a reference to an object right from the start, and to effectively loosen the coupling, we begin refactoring by accepting a reference to an `IInvoiceGrandTotalStrategy` object in the constructor. We then change our `InvoiceGrandTotalStrategy` field to an `IInvoiceGrandTotalStrategy` field and initialize it in the constructor. We finish our refactoring by replacing references from `invoiceGrandTotalService` to `invoiceGrandTotalStrategy`. The resulting refactoring will look similar to the following:

```csharp
/// <summary>
/// Invoice class that uses
/// <seealso cref="IInvoiceGrandTotalStrategy"/>
/// </summary>
public class Invoice
{
    private IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy;
    public List<InvoiceLineItem> InvoiceLineItems { get; set; } 
    public Invoice(IEnumerable<InvoiceLineItem> invoiceLineItems,
                    IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy)
    {
        InvoiceLineItems = new
        List<InvoiceLineItem>(invoiceLineItems);
        this.invoiceGrandTotalStrategy =
        invoiceGrandTotalStrategy;
    }
    public float CalculateGrandTotal(float invoiceSubTotal, float invoiceTotalTax)
    {
        return invoiceGrandTotalStrategy.CalculateGrandTotal(
            invoiceSubTotal, invoiceTotalTax);
    }
    //...
}
```

If you find that the relationship between the two classes is the invocation of one or two methods that return or update data, you may find that delegates are the best way of refactoring.
Delegates

Loosely coupling to delegates requires that another class (usually the other class) inform the class of what delegates to call. This is generally only useful if there are only one or two delegates. Use of another class that contains many methods that are used by the class becomes problematic to loosen coupling by using delegates simply due to the number of delegate initializations that need to occur.

Use of delegates is generally called callbacks. A callback is something that another class calls in order to obtain values or functionality from an external source. If you find that use of the other class requires values or functionality from the other class, use of callbacks may be an appropriate solution. This is particularly true if this is a comprehensive value or a single functionality.

In our Invoice class, we really only have one method that we need to inject into an Invoice object. This may be a perfect scenario for a callback.

Refactoring to a callback is much the same as refactoring to interface-based design. In our particular case, we begin by accepting a Func<float, float, float> parameter. Then, add a Func<float, float, float> field named calculateGrandTotalCallback to the Invoice class. Next we need to initialize the calculateGrandTotalCallback field in the constructor. Finally, we need to replace the call to CalculateGrandTotal to an invocation of the calculateGrandTotalCallback field. The refactoring should result in something similar to the following:

```csharp
/// <summary>
/// Example of using callback
/// instead of interface
/// </summary>
public class Invoice
{
    private Func<float, float, float> calculateGrandTotalCallback;

    public List<InvoiceLineItem> InvoiceLineItems { get; set; }
    public Invoice(IEnumerable<InvoiceLineItem> invoiceLineItems, 
                    Func<float, float, float> calculateGrandTotalCallback)
    {
        InvoiceLineItems = new List<InvoiceLineItem>(invoiceLineItems);
        this.calculateGrandTotalCallback = 
            calculateGrandTotalCallback;
    }
}
```

For More Information:
public float CalculateGrandTotal(float invoiceSubTotal,  
   float invoiceTotalTax)  
{  
    return calculateGrandTotalCallback(  
        invoiceSubTotal, invoiceTotalTax);  
}  
//...  
}

If you find that you are only passing data to the other class and not receiving any data from it, then events are the best way of refactoring.

**Events**

Although events are effectively callbacks, they have first-class status in C# and .NET with their own syntax and follow a specific protocol. Events are an optional one-way communication between one class and many subscribers. If you’re refactoring from a one-to-one coupling with another class and the use of that other class is to effectively notify it of particular values and not receive any information in return, events are a very apt refactoring.

Events are different from the average use of delegates in one important way: multicasting. This means the standard event interface automatically supports combining event listeners into a multicast delegate under the covers. You can certainly support multicast delegates in our delegate example, but we’d have to expand the interface to support "adding" and "removing" a callback. For example:

```csharp
/// <summary>
/// Invoice class that supports multicast delegates
/// </summary>
public class Invoice
{
    private Func<float, float, float> calculateGrandTotalCallback;
    //...
    public void AddCalculateGrandTotalCallback(
        Func<float, float, float> callback)
    {
        calculateGrandTotalCallback += callback;
    }
    public void RemoveCalculateGrandTotalCallback(
        Func<float, float, float> callback)
    {
        
        }  
    }
```
calculateGrandTotalCallback -= callback;
}
}

But, of course, this isn't what callbacks are intended for and thus not applicable for our exemplified purpose. We only get one result from executing multiple delegates with a multicast delegate. For the same reason, multicast delegates aren't suitable for delegates that return values; events that return values are not appropriate.

Let's return to a disposable Invoice for a moment. Let's say we wanted to inform a subscriber when we are disposed. One way of doing that is with delegates. For example:

```csharp
/// <summary>
/// Invoice that implements IDisposable
/// </summary>
public class Invoice : IDisposable
{
    private bool disposed = false;
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    private void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                //...
            }
            Action callback = disposedCallback;
            if (callback != null)
            {
                callback();
            }
            disposed = true;
        }
    }

    private List<InvoiceLineItem> invoiceLineItems;
    private Action disposedCallback;
    For More Information:  
public Invoice(IEnumerable<InvoiceLineItem> invoiceLineItems, 
Action disposedCallback)
{
    this.invoiceLineItems = new 
        List<InvoiceLineItem>(invoiceLineItems);

    this.disposedCallback = 
        disposedCallback;
    //...
}

This does what we want, but it limits us to just one subscriber—the subscriber 
that created the Invoice object. Clearly this isn't the best way to do this, plus it's 
uncommon and not intuitive. Using events is much more intuitive and supports 
multiple subscribers. Refactoring to events involves changing the delegate to a public 
event, removing the initialization of the delegate in the constructor, changing the 
invocation of the Invoice constructor, changing the method used for the delegate 
to include an object and an EventArgs parameter, using this new method to 
assign an event to the Invoice object, and changing the Dispose method to use an 
EventHandler object instead. This results in something like the following:

/// <summary>
/// Disposable Invoice and Disposed event
/// </summary>
public class Invoice : IDisposable
{
    private bool disposed = false;
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    private void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                //...
                EventHandler handler = Disposed;
                if (handler != null)
Refactoring external coupling

External coupling involves two pieces of code being coupled through the common use of a data format or protocol. Issues arise when coupling to a specific data format if the data format is less stable than the classes that use it. If the data format is less stable and tends to change quite often, you may want to refactor the code to not be directly coupled to the data format.

This coupling can be mitigating by performing the Introduce Adapter pattern. The Adapter Pattern converts the less stable data format to a more stable format. It’s common that going from a persistent storage source to an in-memory representation of data that a specific data format is used for persistent storage then used to create objects (in-memory representations). Coupling the object-oriented class to this flat data format means the class takes on the responsibility of conversion. This can become out of control if there need to be many different data formats for this object to support many different persistence mechanisms. Data formats often need to change independently of the class(es) that use it due to size and performance requirements, which leads to the domain class having a dependence on instability.

For example, we may need to instantiate an Invoice object based on a flat data format such as the following:

```csharp
/// <summary>
/// LineItemData shape for
/// serialization of line item data
/// </summary>
[StructLayout(LayoutKind.Sequential)]
struct LineItemData
{
    public float Price;
    public float Discount;
    public float Quantity;
    public String Description;
}

/// <summary>
/// InvoiceData shape for
/// serialization of invoice data
/// </summary>
[StructLayout(LayoutKind.Sequential)]
struct InvoiceData
{
    public LineItemData[] LineItemData;
}

public class Invoice
{
    public Invoice(InvoiceData invoiceData)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(
                invoiceData.LineItemData.Length);
        foreach (LineItemData lineItemData in
            invoiceData.LineItemData)
        {
            InvoiceLineItems.Add(new InvoiceLineItem()
            {
                Price = lineItemData.Price,
                Discount = lineItemData.Discount,
                Description = lineItemData.Description,
                Quantity = lineItemData.Quantity
            });
        }
    }
}

We have two structures intended to be used for getting data in and out of some sort of flat storage (file system, over the wire, and so on): LineItemData and InvoiceData. The Invoice class populates itself when constructed with an InvoiceData instance.
This obviously couples the Invoice class directly with InvoiceData and indirectly with LineItemData and their instability. We can get around this problem by performing the **Introduce Adapter** refactoring.

This refactoring starts with abstracting the InvoiceData class by creating an adapter, like InvoiceDataAdapter. The Invoice class would then be changed to make use of InvoiceDataAdapter instead of directly using InvoiceData. This refactoring would result in something like the following:

```csharp
/// <summary>
/// Provides a translation of InvoiceData into more appropriate interface
/// </summary>
public class InvoiceDataAdapter
{
    List<InvoiceLineItem> invoiceLineItems;

    public InvoiceDataAdapter(InvoiceData invoiceData)
    {
        invoiceLineItems = new List<InvoiceLineItem>(
            invoiceData.LineItemData.Length);
        foreach (LineItemData lineItemData in invoiceData.LineItemData)
        {
            invoiceLineItems.Add(new InvoiceLineItem()
            {
                Price = lineItemData.Price,
                Discount = lineItemData.Discount,
                Description = lineItemData.Description,
                Quantity = lineItemData.Quantity
            });
        }
    }

    public IEnumerable<InvoiceLineItem> InvoiceLineItems
    {
        get { return invoiceLineItems; }
    }
}

/// <summary>
/// Invoice class that uses InvoiceDataAdapter
/// </summary>
public class Invoice
{
    public List<InvoiceLineItem> InvoiceLineItems { get; set; }
}
```

For More Information:
public Invoice(InvoiceDataAdapter adapter)
{
    InvoiceLineItems = new
    List<InvoiceLineItem>(adapter.InvoiceLineItems);
}

//...

## Dependency cycles

We can't discuss coupling without discussing **Dependency cycles**. A dependency cycle (otherwise known as a **Circular dependency**) is when an item depends on something else and that something else, or its dependants, depend on that item. Where this presents a problem is at the deployment level. If something in assembly A depends on something in assembly B and something in assembly B depends on something in assembly A, the compiler won't be able to figure out which project to build first and generate an error. Assembly cycles are easy to detect, you've got an error to deal with. The **Acyclic Dependency Principle** details that the dependency structure of **packages** must have no cycles.

**Package** is a grouping of elements. How the elements are grouped is somewhat subjective, but generally means a physical grouping (for example, assembly in .NET). However, it can be interpreted as any grouping (for example, namespace).

I tend to view packages as all groupings: from assembly to class. The real problem with dependency cycles rears its head at the assembly level, but the assembly structure is dependent on the deployment requirements of the system. The deployment requirements of the system are independent of the logical design of the system. Sometimes, the physical deployment requirements of the system are not known when initial design is begun, and often changes throughout the evolution of a system.

Dependency cycles that don't span assembly boundaries aren't as easy to detect. If class A uses class B and class B uses class A and they're all in the same assembly, there will be no errors. That may or may not be a bad thing. Obviously, it hinders your ability to maintain the code such that you cannot break those two classes out into their own assembly.

---

For More Information:
Proper dependency design

Dependencies between packages should always occur in one direction. There are various guidelines about what direction makes for the most maintainable code. One guideline deals with stability. The Stable Dependencies Principle (DSP) details that a package should only depend on other packages that are more stable than it.

One means of providing stability is abstractiveness. The more abstract something is, the more likely it can be stable. Interfaces, for example, since they only contain a contract and no code, have no possibility that a code change will cause instability with an interface. This assumes that thorough analysis went into the design of the interface. This means the best kind of dependency is a dependency upon an abstraction. The corollary to that is that abstractions should never depend on concrete implementation.

Depending upon abstractions is covered in more detail in Chapter 7.

Another proper dependency design attribute has to deal with layering. Layers can be physical and explicit or logical and implicit. A layer is some sort of abstraction grouping of elements with a common goal. Some common contemporary layers are the Data Layer and the User Interface Layer. Layers have levels. Some are lower than other layers. The Data Layer, for example, is a lower-level layer compared to the User Interface Layer, which is a higher-level layer. Proper dependency structure between layers is always such that lower-level layers never depend on higher-level layers. The abstraction between layers means a lower-level layer can be used by any number of higher-level layers. Take the User Interface Layer and Data Layer layers for example. I may have multiple User Interface Layers: a WinForm layer, a WPF layer, a Web layer, and so on. If my Data Layer depended on one of the User Interface layers, that User Interface layer would have to be deployed with all other user interface layers. It would be catastrophic if we had to deploy WinForm code with our Web User Interface Layer on a web server, for example.

We’ll get more into refactoring as it relates to layers in Chapter 8.
Summary

We've reviewed what it is to be highly cohesive and loosely coupled. We've seen what it means to be non-cohesive and how to detect certain non-cohesiveness through metrics. With some simple refactoring we can make something cohesive where it was previously not.

By refactoring our code to be more cohesive and less coupled, we've improved the maintainability of our code base. Classes are easier to move or reuse and we're less likely to repeat ourselves. Changes are now easier to make because we've decreased the dependencies.

In future chapters, we'll see how we can expand our efforts at loosely coupling even further and discuss some of the benefits and features we can attain once there.
Where to buy this book

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.