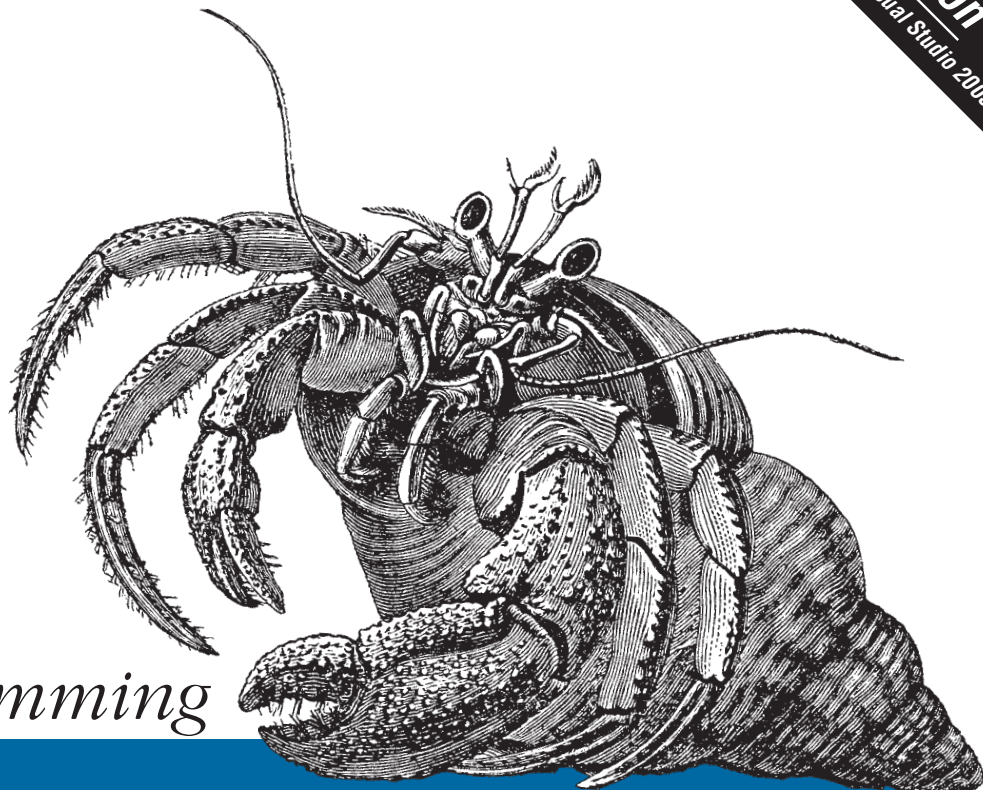


*Building Maintainable,  
Extensible, and Reusable .NET Applications*

Covers .NET 2.0 & Visual Studio 2005  
**2nd Edition**



*Programming*

# .NET Components

O'REILLY®

*Juwal Löwy*

# Interface-Based Programming

As explained in Chapter 1, separation of interface from implementation is a core principle of component-oriented programming. When you separate interface from implementation, the client is coded against an abstraction of a service (the interface), not a particular implementation of it (the object). As a result, changing an implementation detail on the server side (or even switching to a different service provider altogether) doesn't affect the client. This chapter starts by presenting .NET interfaces and describing what options are available to .NET developers when it comes to enforcing the separation of interface from implementation. It then addresses a set of practical issues involving the definition and use of interfaces, such as how to implement multiple interfaces and how to combine interfaces and class hierarchies. After a detailed look at generic interfaces, the chapter ends with a discussion of interface design and factoring guidelines.

## Separating Interface from Implementation

In both C# and Visual Basic 2005, the reserved word `interface` defines a CLR reference type that can't have any implementation, can't be instantiated, and has only public members. Saying that an interface can't have implementation means that it's as if all the interface's methods and properties were abstract. Saying it can't be instantiated means the same as if the interface were an abstract class (or `MustInherit` in Visual Basic 2005). For example, this interface definition:

```
public interface IMyInterface
{
    void Method1();
    void Method2();
    void Method3();
}
```

is almost equivalent to this class definition:

```
public abstract class MyInterface
{
    public abstract void Method1();
    public abstract void Method2();
    public abstract void Method3();
}
```

In traditional object-oriented programming, you typically use an abstract class to define a service abstraction. The abstract class serves to define a set of signatures that multiple classes will implement after deriving from the abstract class. When different service providers share a common base class, they all become polymorphic with that service abstraction, and the client can potentially switch between providers with minimum changes. There are a few important differences between an abstract class and an interface:

- An abstract class can still have implementation: it can have member variables or non-abstract methods or properties. An interface can't have implementation or member variables.
- A .NET class can derive from only one base class, even if that base class is abstract. However, a .NET class can implement as many interfaces as required.
- An abstract class can derive from any other class or from one or more interfaces. An interface can derive only from other interfaces.
- An abstract class can have nonpublic (protected or private) methods and properties, even if they are all abstract. In an interface, by definition, all members are public.
- An abstract class can have static methods and static members and can define constants. An interface can have none of those.
- An abstract class can have constructors. An interface can't.

These differences are deliberately in place, not to restrict interfaces, but rather to provide for a formal public contract between a service provider (the classes implementing the interface) and the service consumer (the client of the classes). Disallowing any kind of implementation details in interfaces (such as method implementations, constants, static members, and constructors) enables .NET to promote loose coupling between the service providers and the client. Because there is nothing in the contract that even hints at implementation, by definition the implementation is well encapsulated behind the interface, and service providers are free to change their implementations without affecting the client. You can even say that the interface acts like a binary shield, isolating each party from the other.

Because interfaces can't be instantiated, .NET forces clients to choose a particular implementation to instantiate. Having only public members in an interface complements the contract semantics nicely: you would not want a contract with hidden clauses or "fine print." Everything the contract implies should be public and well defined. The more explicit and well defined a contract is, the less likely it is that there will be conflicts down the road regarding exactly what the class providing the service is required to do. The class implementing the interface must implement all the interface members without exception, because it has committed to providing this exact service definition. An interface can extend only other interfaces, not classes. By deriving a new interface from an existing interface, you define a new and specialized contract, and any class that implements that interface must implement all members of the base interface(s). A class can choose to implement multiple interfaces, just as a person can choose to commit to multiple contracts.

## Interface Implementation

To implement an interface, all a class has to do is derive from it. Example 3-1 shows the class `MyClass` implementing the interface `IMyInterface`.

*Example 3-1. Defining and implementing an interface*

```
public interface IMyInterface
{
    void Method1();
    void Method2();
    void Method3();
}

public class MyClass : IMyInterface
{
    public void Method1()
    {...}
    public void Method2()
    {...}
    public void Method3()
    {...}
    //other class members
}
```

As trivial as Example 3-1 is, it does demonstrate a number of important points. First, interfaces have visibility—an interface can be private to its assembly (using the `internal` access modifier) or it can be used from outside the assembly (with the `public` access modifier), as in Example 3-1. Second, even though the methods the interface defines have no access modifiers, they are by definition public, and the implementing class has to declare its interface methods as public. Third, there is no need to use `new` or `override` to qualify the method redefinition in the subclass, because an interface method by its very nature can't have any implementation and

therefore has nothing to override. (If you aren't familiar with the new or override keywords, see the sidebar "C# Inheritance Directives" later in this chapter.) Finally, the class must implement all the methods the interface defines, without exception. If the class is an abstract class, it can redefine the methods without providing concrete implementation.

Example 3-2 shows how to define and implement an interface in Visual Basic 2005. In Visual Basic 2005, you need to state which interface method a class method corresponds to. As long as the signature (i.e., the parameters and return value) matches, you can even use a different name for the method. In addition, because the default accessibility in Visual Basic 2005 is public, unlike in C#, adding the Public qualifier is optional.

*Example 3-2. Defining and implementing an interface in Visual Basic 2005*

```
Public Interface IMyInterface
    Sub Method1()
    Sub Method2()
    Sub Method3()
End Interface

Public Class SomeClass
    Implements IMyInterface

    Public Sub Method1() Implements IMyInterface.Method1
        ...
    End Sub

    Public Sub Method2() Implements IMyInterface.Method2
        ...
    End Sub

    Public Sub Method3() Implements IMyInterface.Method3
        ...
    End Sub
End Class
```

To interact with an object using an interface, all a client has to do is instantiate a concrete class that supports the interface and assign that object to an interface variable, similar to using any other base type. Using the same definitions as in Example 3-1, the client code might be:

```
IMyInterface obj;
obj = new MyClass();
obj.Method1();
```

Interfaces promote loose coupling between clients and objects. When you use interfaces, there's a level of indirection between the client's code and the object implementing the interface. If the client wasn't responsible for instantiating the object, there is nothing in the client code that pertains to the object hidden behind the interface shield. There can be many possible implementations of the same interface, such as:

```
public interface IMyInterface
{...}
public class MyClass : IMyInterface
{...}
public class MyOtherClass : IMyInterface
{...}
```

When a client obtains an interface reference by creating an object of type `MyClass`, the client is actually saying to .NET “give me `MyClass`’s *interpretation* of the way `IMyInterface` should be implemented.”

Treating interfaces as binary contracts, which shields clients from changes made to the service providers, is exactly the idea behind COM interfaces, and logically, .NET interfaces have the same semantics as COM interfaces. If you are an experienced COM developer or architect, working with interfaces is probably second nature to you, and you will feel right at home with .NET interfaces.

However, unlike COM, .NET doesn’t enforce separation of the interface from the implementation. For example, using the definitions in Example 3-1, the client’s code can also be:

```
MyClass obj;
obj = new MyClass();
obj.Method1();
```

Because of the way the server in Example 3-1 implements the interface (as public members), nothing prevents the client from programming directly against the object providing the service, instead of the interface. I believe this is because .NET tries to make component-oriented programming accessible to all developers, including those who have trouble with the more abstract concepts of interface-based programming (see the section “.NET Adherence to Component Principles” in Chapter 1). The fact that something is possible, of course, doesn’t mean you should go ahead and do it. Disciplined .NET developers should always enforce the separation, to retain the benefits of interface-based programming.

## Explicit Interface Implementation

The way of implementing an interface shown in the previous section is called *implicit interface implementation*, because a public method with a name and signature that match those of an interface method is implicitly assumed to be an implementation of that interface method.

Example 3-3 demonstrates a simple technique that allows server developers to enforce the separation of the interface from the implementation. The server implementing the interface can actually prevent clients from accessing the interface methods directly by using *explicit interface implementation*. Implementing an interface explicitly means qualifying each interface member name with the name of the interface that defines it.

*Example 3-3. Explicitly implementing an interface*

```
public interface IMyInterface
{
    void Method1();
    void Method2();
}
public class MyClass : IMyInterface
{
    void IMyInterface.Method1()
    {...}
    void IMyInterface.Method2()
    {...}
    //Other methods and members
}
```

Note that the interface members must be implicitly defined as private at the class's scope; you can't use any explicit access modifiers on them, including private. The only way clients can invoke the methods of explicitly implemented interfaces is by accessing them via the interface:

```
IMyInterface obj;
obj = new MyClass();
obj.Method1();
```

To explicitly implement an interface in Visual Basic 2005, you need to explicitly set the method access to Private, as in Example 3-4.

*Example 3-4. Explicitly implementing an interface in Visual Basic 2005*

```
Public Interface IMyInterface
    Sub Method1()
    Sub Method2()
End Interface

Public Class SomeClass
    Implements IMyInterface

    Private Sub Method1() Implements IMyInterface.Method1
        ...
    End Sub

    Private Sub Method2() Implements IMyInterface.Method2
        ...
    End Sub
End Class
```

You should avoid mixing and matching explicit and implicit interface implementations, as in the following fragment:

```
//Avoid mixing and matching:
public interface IMyInterface
{
    void Method1();
```

```

    void Method2();
}
public class MyClass : IMyInterface
{
    void IMyInterface.Method1()
    {...}
    public void Method2()
    {...}
    //Other methods and members
}

```

Although .NET lets you mix and match implementation methods, for consistency, you should avoid it. Such mix and match forces the client to adjust its references depending on whether a particular method is accessible via an interface or directly via the object.

### **Assemblies with Interfaces Only**

Because interfaces can be implemented by multiple components, it's good practice to put them in a separate assembly from that of the implementing components. Maintaining a separate assembly that contains only interfaces allows concurrent development of the server and the client, once the two parties have agreed on the interfaces. Such assemblies also extend the separation of interface from implementation to the code-packaging units.

## **Working with Interfaces**

Now that you have learned the importance of using interfaces in your component-based application, it's time to examine a number of practical issues regarding working with interfaces and tying them to the rest of your application. Later in this chapter, you will also see the support Visual Studio 2005 offers component developers when it comes to adding implementation to your classes for predefined interfaces.



When you name a new interface type, you should prefix it with a capital I and capitalize the first letter of the domain term, as in IAccount, IController, ICalculator, and so on. Use the I prefix even if the domain term itself starts with an I (such as in IIDentity or IImage). .NET tries to do away with the old Windows and C++ Hungarian naming notations (that is, prefixing a variable name with its type), but the I prefix is a direct legacy from COM, and that tradition is maintained in .NET.

## Interfaces and Type Safety

Interfaces are abstract types and, as such, can't be used directly. To use an interface, you need to cast into an interface reference an object that supports it. There are two types of casting—implicit and explicit—and which type you use has an impact on type safety.

Assigning a class instance to an interface variable directly is called an *implicit cast*, because the compiler is required to figure out which type to cast the class to:

```
IMyInterface obj;  
obj = new MyClass();  
obj.Method1();
```

When you use implicit casts, the compiler enforces type safety. If the class `MyClass` doesn't implement the `IMyInterface` interface, the compiler refuses to generate the code and produces a compilation error. The compiler can do that because it can read the class's metadata and can tell in advance that the class doesn't derive from the interface. However, there are a number of cases where you cannot use implicit casting. In such cases, you can use *explicit cast* instead. Explicit casting means casting one type to another type:

```
IMyInterface obj;  
/* Some code here */  
obj = (IMyInterface)new MyClass();  
obj.Method1();
```

However, bear in mind that explicit casts to an interface are made at the expense of type safety. Even if the class doesn't support the interface, the compiler will compile the client's code, and .NET will throw an exception at runtime when the cast fails.

An example where implicit cast is unavailable is when dealing with non-generic class factories. In object-oriented programming, clients often don't create objects directly, but rather get their instances from a *class factory*—a known object in the system that clients ask to create objects they require, instead of creating them directly.\* The advantage of using a class factory is that only the factory is coupled to the actual component types that provide the interfaces. The clients only know about the interfaces. When you need to switch from one service provider to another you only need to modify the factory (actually, instantiate a different type of a factory); the clients aren't affected. When using a class factory that returns some common base type (usually `object`), you can use an explicit cast from the returned object to the interface type:

```
public interface IClassFactory  
{  
    object GetObject();  
}
```

\* See the Abstract Factory design pattern in *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley).

```

}

IClassFactory factory;
/* Some code to initialize the class factory */
IMyInterface obj;
obj = (IMyInterface)factory.GetObject();
obj.Method1();

```



When using generics (where the type parameter is defined at the factory or the method level), there is no need for the cast:

```

public interface IClassFactory<T>
{
    T GetObject();
}
IClassFactory<IMyInterface> factory;
/* Some code to initialize the class factory */
IMyInterface obj;
obj = factory.GetObject();
obj.Method1();

```

Generic interfaces are discussed in more detail later in this chapter.

Another example where implicit cast is impossible is when you want to use one interface that the class implements to get a reference to another interface that the class also supports. Consider the code in Example 3-5. Even when the client uses an implicit cast to get hold of the first interface, it needs an explicit cast to obtain the second.

*Example 3-5. Defining and using multiple interfaces*

```

public interface IMyInterface
{
    void Method1();
    void Method2();
}
public interface IMyOtherInterface
{
    void Method3();
}

public class MyClass : IMyInterface, IMyOtherInterface
{
    public void Method1()
    {...}
    public void Method2()
    {...}
    public void Method3()
    {...}
}
//Client-side code:
IMyInterface obj1;
IMyOtherInterface obj2;

```

*Example 3-5. Defining and using multiple interfaces (continued)*

```
obj1 = new MyClass();  
obj1.Method1();  
  
obj2 = (IMyOtherInterface)obj1;  
obj2.Method3();
```

In all these examples that use explicit casts, you must incorporate error handling, in case the type you are trying to cast from doesn't support the interface, and use try and catch statements to handle any exceptions.

There is, however, a safer, defensive approach to explicit casting—the as operator. The as operator performs the cast if it's legal and assigns a value to the variable. If a cast isn't possible, instead of throwing an exception, the as operator assigns null to the interface variable. Example 3-6 shows how to use the as operator to perform a safe cast that doesn't result in an exception in case of an error.

*Example 3-6. Using the as operator to cast safely to the desired interface*

```
SomeType obj1;  
IMyInterface obj2;  
  
/* Some code to initialize obj1 */  
  
obj2 = obj1 as IMyInterface;  
if(obj2 != null)  
{  
    obj.Method1();  
}  
else  
{  
    //Handle error in expected interface  
}
```



Interestingly enough, using the as operator to find out whether a particular object supports a given interface is semantically identical to COM's `QueryInterface()` method. Both mechanisms allow clients to defensively obtain an interface from an object and handle the situation where the interface isn't supported.

In general, you should always program defensively on the client side, using the as operator as shown in Example 3-6, instead of explicit casting. Never assume an object supports an interface—that leads both to robust error handling and to separation of the interface from the implementation, regardless of whether or not the server is using explicit interface implementation. Make it a habit on the client side to use the server via an interface and thus enforce the separation manually.

## Interface Methods, Properties, and Events

An interface isn't limited only to defining methods. An interface can also define properties, indexers, and events. Example 3-7 shows the syntax for defining all of these in an interface and the corresponding implementation.

*Example 3-7. An interface can define methods, properties, indexers, and events*

```
public delegate void NumberChangedEventHandler(int number);

public interface IMyInterface
{
    void Method1(); //A method
    int SomeProperty{ get; set; }//A property
    int this[int index]{ get; set;}//An indexer
    event NumberChangedEventHandler NumberChanged;//An event
}

public class MyClass : IMyInterface
{
    public event NumberChangedEventHandler NumberChanged;

    public void Method1()
    {...}
    public int SomeProperty
    {
        get
        {...}
        set
        {...}
    }
    public int this[int index]
    {
        get
        {...}
        set
        {...}
    }
}
```

## Interfaces and Structs

An interesting use of interfaces with properties involves structs. In .NET, a struct (a Structure in Visual Basic 2005) can't have a base struct or a base class, because it's a value type. However, .NET does permit structs to implement one or more interfaces. The reason for this is that sometimes you want to define abstract data storage, and there are a number of possible implementations for the actual structure. By defining an interface (preferably with properties only, but it can have methods as well), you can pass around the interface instead of the actual struct and gain the benefits of polymorphism, even though structs aren't allowed to derive from a common base struct. Example 3-8 demonstrates the use of an interface (with properties only) as a base type for structs.

*Example 3-8. Using an interface as a base type for structs*

```
public interface IMyBaseStruct
{
    int    SomeNumber{ get; set; }
    string SomeString{ get; set; }
}

struct MyStruct : IMyBaseStruct
{
    public int SomeNumber
    { get{...} set{...} }
    public string SomeString
    { get{...} set{...} }
    //Rest of the implementation
}
struct MyOtherStruct : IMyBaseStruct
{
    public int SomeNumber
    { get{...} set{...} }
    public string SomeString
    { get{...} set{...} }
    //Rest of the implementation
}
//A method that accepts a struct, without knowing exactly the type
public void DoWork(IMyBaseStruct storage)
{...}
```

## Interfaces and Partial Types

Partial types allow the component architect to define interface derivation for a class but have another developer implement it (similar to the old C++ distinction between header files and CPP files):

```
//In App.cs
public interface IMyInterface
{
    void Method1();
    void Method2();
}

public partial class MyClass : IMyInterface
{}
```

```
//In MyClass.cs

public partial class MyClass
{
    public void Method1()
    {...}
    public void Method2()
    {...}
}
```

With a partial class, each part of the class can choose to add interface derivation, or interface derivation and implementation:

```
public partial class MyClass
{

public partial class MyClass : IMyInterface
{
    public void Method1()
    {...}
    public void Method2()
    {...}
}
```

However, only a single part can implement an interface member.

## Implementing Multiple Interfaces

A class can derive from as many interfaces as required (see Example 3-5), but from at most one base class. When a class derives from a base class and from one or more interfaces, the base class must be listed first in the derivation chain (a requirement the compiler enforces):

```
public interface IMyInterface
{
}
public interface IMyOtherInterface
{
}
public class MyBaseClass
{
}
public class MySubClass : MyBaseClass,IMyInterface,IMyOtherInterface
{
}
```

Even such a trivial example raises a number of questions. What if both interfaces define identical methods? What are the available ways to resolve such collisions? What if the base class already derives from one or more of the interfaces?

When a class derives from two or more interfaces that define an identical method, you have two options: the first is to channel both interface methods to the same actual method implementation, and the second is to provide separate method implementations. For example, consider two interfaces that define the identical method `Method1()`:

```
public interface IMyInterface
{
    void Method1();
}
public interface IMyOtherInterface
{
    void Method1();
}
```

If you want to channel both interface methods to the same method implementation, all you have to do is derive from the interfaces and implement the method once:

```
public class MyClass : IMyInterface,IMyOtherInterface
{
    public void Method1()
    {...}
    //Other methods and members
}
```

Regardless of which interface the client of MyClass chooses to use, calls to Method1() will be channeled to that single implementation:

```
IMyInterface obj1;
IMyOtherInterface obj2;

obj1 = new MyClass();
obj1.Method1();

obj2 = obj1 as IMyOtherInterface;
Debug.Assert(obj2 != null);
obj2.Method1();
```

To provide separate implementations, use explicit interface implementation by qualifying the method implementation with the name of the interface that defines it:

```
public class MyClass : IMyInterface,IMyOtherInterface
{
    void IMyInterface.Method1()
    {...}
    void IMyOtherInterface.Method1()
    {...}
    //Other methods and members
}
```

Now, when the client calls an interface method, that interface-specific method is called. You can even have separate explicit implementations for some of the common methods and channel the others to the same implementation. However, as mentioned before, for the sake of consistency it's better to avoid mixing and matching.

If you want to both use explicit interface implementation and channel the implementation from one interface to the other, you will need to use the `this` reference to query for the desired interface and delegate the call:

```
public class MyClass : IMyInterface, IMOtherInterface
{
    void IMyInterface.Method1 ( )
    {...}
    void IMyOtherInterface.Method1 ( )
    {
        IMyInterface myInterface = this;
        myInterface.Method ( );
    }
    //Other methods and members
}
```



Using the `this` reference this way is the only way to call an explicit interface method by its own implementing class.

## Interfaces and Class Hierarchies

In component-oriented programming, you focus on defining and implementing interfaces. In object-oriented programming, you model your solution by using class hierarchies. How do the two concepts interact? The answer depends on the way you override or redefine the interface methods at the different levels of the class hierarchy. Consider the code in Example 3-9, which illustrates that when defining an interface only at the root of a class hierarchy, each level must override its base-class declarations to preserve semantics.

*Example 3-9. Overriding an interface in a class hierarchy*

using `System.Diagnostics`; //For the Trace class

```
public interface ITrace
{
    void TraceSelf();
}
public class A : ITrace
{
    public virtual void TraceSelf()
    {
        Trace.WriteLine("A");
    }
}
public class B : A
{
    public override void TraceSelf()
    {
        Trace.WriteLine("B");
    }
}
public class C : B
{
    public override void TraceSelf()
    {
        Trace.WriteLine("C");
    }
}
```

In a typical class hierarchy, the topmost base class should derive from the interface, providing polymorphism with the interface to all subclasses. The topmost base class must also define all the interface members as `virtual`, so that subclasses can override them. Each level of the class hierarchy can override its preceding level (using the `override` inheritance qualifier), as shown in Example 3-9. When the client uses the interface, it then gets the desired interpretation of the interface. For example, if the client code is:

## C# Inheritance Directives

In C#, you are required to explicitly indicate the semantics of inheritance when you supply a method with a name and signature identical to those of a base-class method. If you wish to override the base-class method when instantiating a base-class type with a subclass reference, you need to use the override directive:

```
public class BaseClass
{
    public virtual void TraceSelf()
    {
        Trace.WriteLine("BaseClass");
    }
}
public class SubClass : BaseClass
{
    public override void TraceSelf()
    {
        Trace.WriteLine("SubClass");
    }
}
BaseClass obj = new SubClass();
obj.TraceSelf(); //Outputs "SubClass"
```

If you want to provide the base-class behavior instead, use the new directive, with or without virtual, at the base class:

```
public class BaseClass
{
    public virtual void TraceSelf()
    {
        Trace.WriteLine("BaseClass");
    }
}
public class SubClass : BaseClass
{
    public new void TraceSelf()
    {
        Trace.WriteLine("SubClass");
    }
}
BaseClass obj = new SubClass();
obj.TraceSelf(); //Outputs "BaseClass "
```

```
ITrace obj = new B();
obj.TraceSelf();
```

the object traces “B” to the output window, as expected.

Things are less obvious if the subclasses use the new inheritance qualifier. The new modifier gives subclass behavior only when dealing with an explicit reference to a subclass, such as:

```
B obj = new B();
```

In all other cases, the base class implementation is used. If the code in Example 3-9 was written as:

```
public class A : ITrace
{
    public virtual void TraceSelf()//virtual is optional
    {
        Trace.WriteLine("A");
    }
}
public class B : A
{
    public new void TraceSelf()
    {
        Trace.WriteLine("B");
    }
}
public class C : B
{
    public new void TraceSelf()
    {
        Trace.WriteLine("C");
    }
}
```

then this client code:

```
ITrace obj = new B();
obj.TraceSelf();
```

would now trace “A” to the output window instead of “B.” Note that this is exactly why the new inheritance modifier is available. Imagine a client that somehow depends on the base class’s particular implementation. If a new subclass is used instead of the base class, the new modifier ensures that the client will get the implementation it expects. However, this nuance makes sense only when you’re dealing with clients that don’t use interface-based programming but rather program directly against the objects:

```
A obj = new B();
obj.TraceSelf();//Traces "A"
```

You can support such clients and still provide interface-based services to the rest of the clients. To achieve that, each class in the hierarchy can reiterate its polymorphism with the interface by explicitly deriving from the interface (in addition to having the base class derive from the interface). Doing so (as shown in Example 3-10) makes the new modifier yield the same results as the override modifier for the interface-based clients:

```
ITrace obj = new B();
obj.TraceSelf();//Traces "B"
```

Note that using `virtual` at the base-class level is optional.

In general, you should use the `override` modifier, as in Example 3-9, with virtual interface members at the topmost base class. Such code is readable and straightforward. Code such as that in Example 3-10 makes for an interesting exercise but is rarely of practical use.

*Example 3-10. Deriving from the interface explicitly at each level of the class hierarchy*

```
using System.Diagnostics; //For the Trace class

public interface ITrace
{
    void TraceSelf();
}
public class A : ITrace
{
    public virtual void TraceSelf()//virtual is optional
    {
        Trace.WriteLine("A");
    }
}
public class B : A, ITrace
{
    public new void TraceSelf()
    {
        Trace.WriteLine("B");
    }
}
public class C : B, ITrace
{
    public new void TraceSelf()
    {
        Trace.WriteLine("C");
    }
}
```

If you want to combine explicit interface implementation and class hierarchy, you should do so in a way that allows a subclass to call its base-class implementation. Because with explicit interface implementation, the implementation is private, you will need to add at the topmost base class a protected virtual method for each interface method. Only the topmost base class should explicitly implement the interface, and its implementation should call the protected virtual methods. All the subclasses should override the protected virtual methods:

```
public class A : ITrace
{
    protected virtual void TraceSelf()
    {
        Trace.WriteLine("A");
    }
    void ITrace.TraceSelf()
    {
        TraceSelf();
    }
}
```

```

    }
}
public class B : A
{
    protected override void TraceSelf()
    {
        Trace.WriteLine("B");
        base.TraceSelf();
    }
}

```

## Interfaces and Generics

Like classes or structures, interfaces too can be defined in terms of generic type parameters.\* Generic interfaces provide all the benefits of interface-based programming without compromising type safety, performance, or productivity. All of what you have seen so far with normal interfaces you can also do with generic interfaces. The main difference is that when deriving from a generic interface, you must provide a specific type parameter to use instead of the generic type parameter. For example, given this definition of the generic `IList<T>` interface:

```

public interface IList<T>
{
    void AddHead(T item);
    void RemoveHead(T item);
    void RemoveAll();
}

```

you can implement the interface implicitly and substitute an integer for the generic type parameter:

```

public class NumberList : IList<int>
{
    public void AddHead(int item)
    {...}
    public void RemoveHead(int item)
    {...}
    public void RemoveAll()
    {...}
    //Rest of the implementation
}

```

When the client uses `IList<T>`, it must choose an implementation of the interface with a specific type parameter:

```

IList<int> list = new NumberList();
list.AddHead(3);

```

\* If you are not familiar with generics, Appendix D provides a concise overview.

Generic interfaces allow you to define an abstract service definition (the generic interface) once, yet use it on multiple components with multiple type parameters. For example, an integer-based list can implement the interface:

```
public class NumberList : IList<int>
{...}
```

And so can a string-based list:

```
public class NameList : IList<string>
{...}
```

Once a generic interface is *bounded* (i.e., once you've specified types for it) it is considered a distinct type. Consequently, two generic interface definitions with different generic type parameters are no longer polymorphic with each other. This means that a variable of the type `IList<int>` cannot be assigned to a variable or passed to a method that expects an `IList<string>`:

```
void ProcessList(IList<string> names)
{...}

IList<int> numbers = new NumberList();
ProcessList(numbers); //Does not compile
```

You can maintain the polymorphism with generic interfaces if you keep the use of the interface in generic type parameter terms:

```
public class ListClient<T>
{
    public void ProcessList(IList<T> list)
    {...}
}

IList<int> numbers = new NumberList();
IList<string> names = new NameList();

ListClient<int> numbersClient = new ListClient<int>();
ListClient<string> namesClient = new ListClient<string>();

//Reuse of the code and algorithms of ProcessList():
numbersClient.ProcessList(numbers);
namesClient.ProcessList(names);
```

## Deriving from a Generic Interface

When you derive an interface from a generic interface, you have a number of options as to how to define the sub-interface. Usually, you will prefer to have the sub-interface be a generic interface and provide the sub-interface's generic type parameters as type parameters to the base interface:

```
public interface IBaseInterface<T>
{
    void SomeMethod(T t);
}

public interface ISubInterface<T> : IBaseInterface<T>
{...}
```

However, you can also specify a particular type to the base interface, thus making the sub-interface non-generic:

```
public interface ISubInterface : IBaseInterface<string>
{...}
```

Typically, when you derive from a generic interface you will do so in a generic subclass and let the client decide on the particular type parameters to use. This is an alternative to defining type-specific subclasses, and it does not limit the subclasses to the use of a particular type parameter:

```
public class List<T> : IList<T>
{
    public void AddHead(T item)
    {...}
    //Rest of the implementation
}
IList<int> numbers = new List<int>();
IList<string> names = new List<string>();
```

When a generic class or a generic interface derives from a generic interface, it cannot specify multiple naked generic types to the interface:

```
//Does not compile:
public class List<T,U> : IList<T>,IList<U>
{...}
```

Doing so unifies the interfaces when the same type is specified, which violates the uniqueness of interfaces. If that were allowed, the compiler would not know how to resolve a definition such as this:

```
List<int,int> list;
```

## Explicit Generic Interface Implementation

Just as with regular interfaces, you can implement a generic interface explicitly:

```
public class NumberList : IList<int>
{
    void IList<int>.AddHead(int item)
    {...}
    void IList<int>.RemoveHead(int item)
    {...}
    void IList<int>.RemoveAll()
    {...}
    //Rest of the implementation
}
```

Note the specification of the type parameter in the explicit implementation:

```
void IList<int>.AddHead(int item);
```

This is required because the `AddHead()` method is not just a method of the generic interface `IList<T>`; rather, it is a method of the generic interface `IList<T>` with an

integer as a type parameter. The same would be true if the implementing list were itself a generic class:

```
public class List<T> : IList<T>
{
    void IList<T>.AddHead(T item)
    {...}
    //Rest of the implementation
}
```

Explicit generic interface implementation is especially handy when the same type implements multiple versions of the generic interface, each with a different concrete type parameter. Although you can use implicit interface implementation, like this:

```
public class List : IList<int>,IList<string>
{
    public void AddHead(int item)
    {...}
    public void AddHead(string item)
    {...}
    public void RemoveAll()
    {...}
    //Rest of the implementation
}
```

it is preferable in such cases to use explicit interface implementation. The reason is that in methods such as `RemoveAll()`, you channel the implementation of both `IList<int>` and `IList<string>` to the same method. The client can use either `IList<int>` or `IList<string>`:

```
IList<int> list = new List();
list.RemoveAll();
```

Yet you have no way of knowing in the implementation of `RemoveAll()` which internal data structure you should clear when the client calls it. In addition, when you implement the same generic interface multiple times with different type parameters, you often have to use explicit implementation when the compiler cannot resolve the ambiguity. For example, if `IList<T>` has a method called `GetHead()`, defined as:

```
public interface IList<T>
{
    void AddHead(T item);
    void RemoveHead(T item);
    void RemoveAll();
    T GetHead();
}
```

you must use explicit interface implementation, because in C# you cannot overload methods only by a different returned type:

```
public class List : IList<int>,IList<string>
{
    int IList<int>.GetHead()
    {...}
}
```

```

    string IList<string>.GetHead()
    {...}
    //Rest of the implementation
}

```

## Generic Interfaces as Operators

C# 2.0 has an additional interesting use for generic interfaces. In C# 2.0, it is impossible to use operators such as + or += on generic type parameters. For example, the following code does not compile because C# 2.0 does not have operator constraints:

```

public class Calculator<T>
{
    public T Add(T argument1,T argument2)
    {
        return argument1 + argument2;//Does not compile
    }
    //Rest of the methods
}

```

The compiler does not know whether the type parameter the client will specify supports the + operator, so it will refuse to compile that code.

Nonetheless, you can compensate by using interfaces that define generic operations. Since an interface method cannot have any code in it, you can specify the generic operations at the interface level and provide a concrete type and implementation at the subclass level:

```

public interface ICalculator<T>
{
    T Add(T argument1,T argument2);
    T Subtract(T argument1,T argument2);
    T Divide(T argument1,T argument2);
    T Multiply(T argument1,T argument2);
}
public class Calculator : ICalculator<int>
{
    public int Add(int argument1, int argument2)
    {
        return argument1 + argument2;
    }
    //Rest of the methods
}

```

## Interface-Level Constraints

An interface can define constraints for the generic types it uses. For example:

```

public interface IList<T> where T : IComparable<T>
{...}

```

However, you should be very mindful about the implications of defining constraints at the scope of an interface. An interface should not have any shred of implementation details, to reinforce the notion of separation of interface from implementation. There are many ways to implement the generic interface, and the specific type parameters used are, after all, an implementation detail. Constraining them commonly couples the interface to specific implementation options.

For example, constraining a type parameter on an interface to derive from a particular class, like this, is a bad idea:

```
public class Customer
{...}
public interface IList<T> where T : Customer
{...}
```

This, in effect, makes the generic `IList<T>` useful only for managing lists of customers. If you want this level of strong typing with polymorphism, define a new interface dedicated to managing customers, instead of skewing the general-purpose definition of the generic `IList<T>`:

```
public interface ICustomerList : IList<Customer>
{...}
```

Constraining a default constructor, as follows, is also something to avoid:

```
public interface IList<T> where T : new()
{...}
```

Not all types have default public constructors, and the interface doesn't really care that they do not; the interface cannot contain any implementation code that uses such constructors anyway.

Even constraining an interface's generic type parameter to derive from another interface should be viewed with extreme caution. For example, you may think you should add a constraint for `IComparable<T>` to the list interface definition if you add a method that removes a specified item from the list:

```
public interface IList<T> where T : IComparable<T>
{
    void Remove(T item);
    //Rest of the interface
}
```

While implementing this method will often involve comparing the specified item to items in the list, which will in turn necessitate having the type parameter support `IComparable<T>`, this is still an implementation detail. Perhaps the implementing data structure has other ways of comparing the type parameters it uses, or perhaps it does not implement the method at all. All of that is irrelevant to the pure interface definition. Let the class implementing the generic interface add the constraint, and keep the interface itself constraint-free:

```
public class List<T> : IList<T> where T : IComparable<T>
{
    public void Remove(T item)
```

```

    {...}
    //Rest of the implementation
}

```

## Generic Derivation Constraints

When a generic class constrains one of its type parameters to derive from an interface, the client may provide as a specific type parameter a particular implementation of the interface:

```

public class ListClient<L,T> where L : IList<T>
{
    public void ProcessList(L list)
    {...}
}
public class NumberList : IList<int>
{...}

ListClient<NumberList,int> client = new ListClient<NumberList,int>();
NumberList numbers = new NumberList();
client.ProcessList(numbers);

```

However, you can also satisfy the constraint by specifying as a type parameter the very interface the type parameter is constrained against, not a particular implementation of it:

```

public class ListClient<L,T> where L : IList<T>
{
    public void ProcessList(L list)
    {...}
}
public class List<T> : IList<T>
{...}

ListClient<IList<int>,int> client = new ListClient<IList<int>,int>();
IList<int> numbers = new List<int>();
client.ProcessList(numbers);

```

or even:

```

public class AnotherClient<U>
{
    ListClient<IList<U>,U> m_ListClient;
}

```

This helps to separate the client code from particular interface implementations.

## Generics, Interfaces, and Casting

The C# compiler will only let you implicitly cast generic type parameters to object, or to constraint-specified types, as shown in Example 3-11. Such implicit casting is of course type-safe, because any incompatibility is discovered at compile time.

*Example 3-11. Implicit casting of generic type parameters*

```
public interface ISomeInterface
{...}
public class BaseClass
{...}
public class MyClass<T> where T : BaseClass,ISomeInterface
{
    void SomeMethod(T t)
    {
        ISomeInterface obj1 = t;
        BaseClass      obj2 = t;
        object         obj3 = t;
    }
}
```

The compiler will let you explicitly cast generic type parameters to any other interface, but not to a class:

```
public interface ISomeInterface
{...}
public class SomeClass
{...}
public class MyClass<T>
{
    void SomeMethod(T t)
    {
        ISomeInterface obj1 = (ISomeInterface)t;//Compiles
        SomeClass      obj2 = (SomeClass)t;      //Does not compile
    }
}
```

Such explicit casting is dangerous, because it may throw an exception at runtime if the specific type parameter used does not support the interface to which you explicitly cast. Instead of risking a casting exception, a better approach is to use the `is` and `as` operators, as shown in Example 3-12. You can use `is` and `as` both on naked generic type parameters and on generic classes with specific parameters.

*Example 3-12. Using `is` and `as` operators on generic type parameters*

```
public interface IMyInterface
{...}
public interface ISomeInterface<T>
{...}
public class MyClass<T>
{
    public void MyMethod(T t)
    {
        if(t is IMyInterface)
        {...}

        if(t is ISomeInterface<T>)
        {...}
    }
}
```

Example 3-12. Using *is* and *as* operators on generic type parameters (continued)

```
IMyInterface obj1 = t as IMyInterface;
if(obj1 != null)
{...}

ISomeInterface<T> obj2 = t as ISomeInterface<T>;
if(obj2 != null)
{...}
}
}
```

## Generic Interface Methods

In C# 2.0, an interface method can define generic type parameters, specific to its execution scope:

```
public interface IMyInterface<T>
{
    void MyMethod<X>(T t, X x);
}
```

This is an important capability, because it allows you to call the method with a different type every time, which is often very handy for utility classes.

You can define method-specific generic type parameters even if the containing interface does not use generics at all:

```
public interface IMyInterface
{
    void MyMethod<T>(T t);
}
```

This ability is for methods only. Properties or indexers can use only generic type parameters defined at the scope of the interface.

When you call an interface method that defines generic type parameters, you can provide the type to use at the call site:

```
public class MyClass : IMyInterface
{
    public void MyMethod<T>(T t)
    {...}
}
IMyInterface obj = new MyClass();
obj.MyMethod<int>(3);
```

That said, when the method is invoked, the C# compiler is smart enough to infer the correct type based on the type of parameter passed in, and it allows you to omit the type specification altogether:

```
IMyInterface obj = new MyClass();
obj.MyMethod(3);
```

This ability is called *generic type inference*. Note that the compiler cannot infer the type based on the type of the returned value alone:

```
public interface IMyInterface
{
    T MyMethod<T>();
}
public class MyClass : IMyInterface
{
    public T MyMethod<T>()
    {...}
}
IMyInterface obj = new MyClass();
int number = obj.MyMethod();//Does not compile
```

When an interface method defines its own generic type parameters, it can also define constraints for these types:

```
public interface IMyInterface
{
    void MyMethod<T>(T t) where T : IComparable<T>;
}
}
```

However, my recommendation to avoid constraints at the interface level extends to method-level generic type parameters as well.

## Designing and Factoring Interfaces

Syntax aside, how do you go about designing interfaces? How do you know which methods to allocate to which interface? How many members should each interface have? Answering these questions has little to do with .NET and a lot to do with abstract component-oriented analysis and design. An in-depth discussion of how to decompose a system into components and how to discover interface methods and properties is beyond the scope of this book. Nonetheless, this section offers a few pieces of advice to guide you in your interface-design effort.

### Interface Factoring

An interface is a grouping of logically related methods and properties. What constitutes “logically related” is usually domain-specific. You can think of interfaces as different facets of the same entity. Once you have identified (after a requirements analysis) all the operations and properties the entity supports, you need to allocate them to interfaces. This is called *interface factoring*. When you factor an interface, always think in terms of reusable elements. In a component-oriented application, the basic unit of reuse is the interface. Would this particular interface factoring yield interfaces that other entities in the system can reuse? What facets of the entity can logically be factored out and used by other entities?

Suppose you wish to model a dog. The requirements are that the dog be able to bark and fetch and that it have a veterinary clinic registration number and a property for having received shots. You can define the IDog interface and have different kinds of dogs, such as Poodle and GermanShepherd, implement the IDog interface:

```
public interface IDog
{
    void Fetch();
    void Bark();
    long VetClinicNumber{get;set;}
    bool HasShots{get;set;}
}
public class Poodle : IDog
{...}

public class GermanShepherd : IDog
{...}
```

However, such a composition of the IDog interface isn't well-factored. Even though all the interface members are things a dog should support, Fetch and Bark are more logically related to each other than to VetClinicNumber and HasShots. Fetch() and Bark() involve one facet of the dog, as a living, active entity, while VetClinicNumber and HasShots involve a different facet, one that relates it as a record of a pet in a veterinary clinic. A better approach is to factor out the VetClinicNumber and HasShots properties to a separate interface called IPet:

```
public interface IPet
{
    long VetClinicNumber{ get; set; }
    bool HasShots{ get; set; }
}
public interface IDog
{
    void Fetch();
    void Bark();
}
```

Because the pet facet is independent of the canine facet, other entities (such as cats) can reuse the IPet interface and support it:

```
public interface IPet
{
    long VetClinicNumber{ get; set; }
    bool HasShots{ get; set; }
}
public interface IDog
{
    void Fetch();
    void Bark();
}
public interface ICat
{
    void Purr();
}
```

```

    void CatchMouse();
}

public class Poodle : IDog,IPet
{...}
public class Siamese : ICat,IPet
{...}

```

This factoring, in turn, allows you to decouple the clinic-management aspect of the application from the actual pet type (be it dogs or cats). Factoring operations and properties into separate interfaces is usually done when there is a weak logical relation between methods. However, identical operations are sometimes found in several unrelated interfaces, and these operations are logically related to their respective interfaces. For example, both cats and dogs need to shed fur and feed their offspring. Logically, shedding is just a dog operation, as is barking, and is also just a cat operation, as is purring. In such cases, you can factor the interfaces into a hierarchy of interfaces instead of separate interfaces:

```

public interface IMammal
{
    void ShedFur();
    void Lactate();
}
public interface IDog : IMammal
{
    void Fetch();
    void Bark();
}
public interface ICat : IMammal
{
    void Purr();
    void CatchMouse();
}

```

## Factoring Metrics

As you can see, proper interface-factoring results in more specialized, loosely coupled, fine-tuned, and reusable interfaces, and subsequently, those benefits apply to the system as well. In general, interface factoring results in interfaces with fewer members. When you design a component-based system, however, you need to balance out two countering forces (see Figure 3-1).

If you have too many granular interfaces, it will be easy to implement each interface, but the overall cost of interacting with all those interfaces will be prohibitive. On the other hand, if you have only a few complex, large, poorly factored interfaces, the cost of implementing those interfaces will be a prohibitive factor, even though the cost of interacting with them might be low. In any given system, the total effort involved in designing and maintaining the components that implement the interfaces is the sum of those two factors. As you can see from Figure 3-1, there is an area of minimum

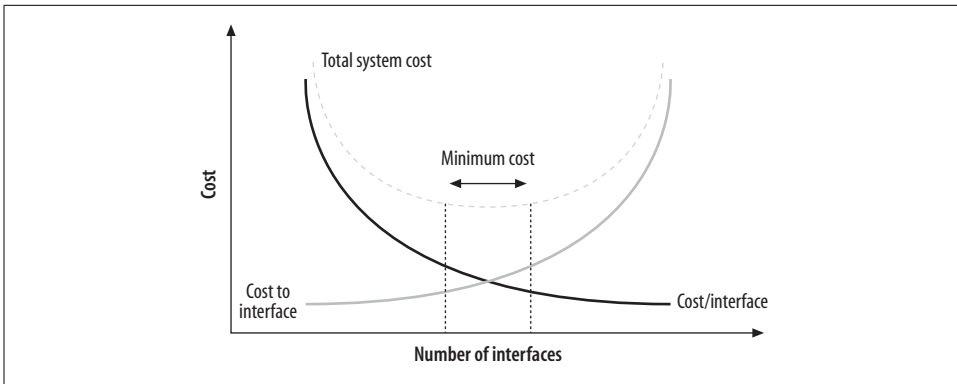


Figure 3-1. Balancing the number of components and their size

cost or effort in relation to the size and number of interfaces. Because these interface-factoring issues are independent of the component technology used, I can extrapolate from my own and others' experiences of factoring and architecting large-scale applications and share a few rules of thumb and metrics I have collected about interface factoring.

Interfaces with just one member are possible, but you should avoid them. An interface is a facet of an entity, and that facet must be pretty dull if you can express it with just one method or property. Examine that single method: is it using too many parameters? Is it too coarse, and therefore, should it be factored into several methods? Should you factor this method or property into an already existing interface?

The optimal number of interface members (in my opinion and experience) is between three and five. If you design an interface with more members—say, six to nine—you are still doing relatively well. However, try to look at the members to determine whether any can be collapsed into each other, since it's quite possible to over-factor methods and properties. If you have an interface with 12 or more methods, you should definitely find ways to factor the members into either separate interfaces or a hierarchy of interfaces. Your coding standard should set some upper limit never to be exceeded, regardless of the circumstances (say, 20).

Another rule involves the ratio of methods to properties among interface members. Interfaces allow clients to invoke abstract operations, without caring about actual implementation details. Properties are what is known as *just-enough-encapsulation*. It's much better to expose a member variable as a property than to give direct access to it, because then you can encapsulate the business logic of setting and reading that variable's value in the object, instead of spreading it over the clients. Ideally, you shouldn't bother clients with properties at all. Clients should invoke methods and let the object worry about how to manage its state. Consequently, interfaces should have more methods than properties, by a ratio of at least 2:1. The one exception is interfaces that do nothing except define properties; such interfaces should have properties only, with no methods.

It's best to avoid defining events as interface members, if possible. Leave it up to the object to decide whether it needs an event member variable or not. As you'll see in Chapter 6, there is more than one way to manage events.

### Is .NET Well-Factored?

After writing down my rules of thumb and metrics for interface factoring, I was curious to see how the various interfaces defined by the .NET Framework look in light of these points. I examined more than 300 interfaces defined by .NET. I excluded from the survey the COM interoperability interfaces redefined in .NET, because I wanted to look at native .NET interfaces only. I also excluded from the results the outliers—interfaces with 0 members and interfaces with more than 20 members. I consider an interface with more than 20 members to be a poorly factored one, not to be used as an example. On average, a .NET Framework interface has 3.75 members, with a methods-to-properties ratio of 3.5:1. Less than 3% of the members are events. These metrics nicely reaffirm the rules of thumb outlined in this section; you could say that on average, .NET interfaces are well-factored.

A word of caution about factoring metrics: rules of thumb and generic metrics are only tools to help you gauge and evaluate your particular design. There is no substitute for domain expertise and experience. Always be practical, apply judgment, and question what you do in light of the metrics.

## Interfaces in Visual Studio 2005

Visual Studio 2005 has excellent support for implementing and refactoring interfaces. As a component developer, your classes will occasionally need to implement an interface defined by another party. Instead of copying and pasting the interface definition, or typing it in, you can use Visual Studio 2005 to generate a *skeletal* implementation of the interface. A skeletal implementation is a do-nothing implementation: all implanted methods or properties throw an exception of type `Exception` and do not contain any other code. A skeletal implementation is required to at least get the code compiled as a starting point for your implementation of an interface, and it prevents clients from consuming a half-baked implementation. To have Visual Studio 2005 generate a skeletal interface implementation, you first add the interface to the class derivation chain. When you finish typing the interface name (such as `IMyInterface`), Visual Studio 2005 marks a little underscore tag under the `I` of the interface name. If you hover over `IMyInterface`, Visual Studio 2005 pops up a smart tag with a tool tip, "Options to implement interface." If you click the down arrow of the smart tip you can select from two options in the menu, implementing the interface either implicitly or explicitly (see Figure 3-2).

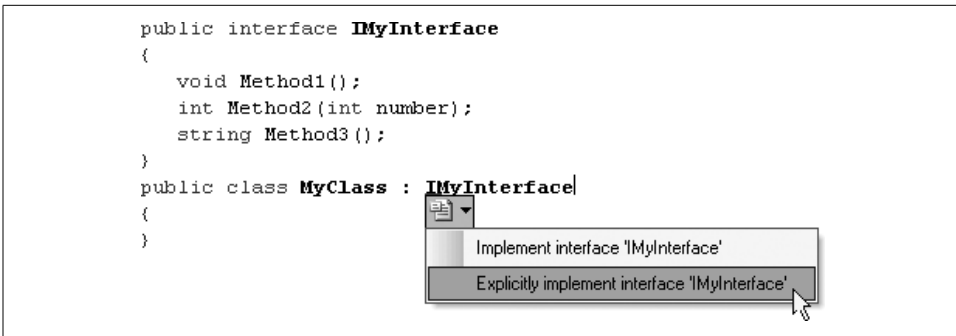


Figure 3-2. Using Visual Studio 2005 to provide a skeletal interface implementation

Once you select an option, Visual Studio 2005 creates a skeletal implementation of the interface on your class and scopes it with a collapsible #region directive. For example, consider this interface definition:

```
public interface IMyInterface
{
    void Method1();
    int Method2(int number);
    string Method3();
}
```

If you select explicit implementation, Visual Studio 2005 generates this skeletal implementation:

```
public class MyClass : IMyInterface
{
    #region IMyInterface Members
    void IMyInterface.Method1()
    {
        throw new Exception();
    }
    int IMyInterface.Method2(int number)
    {
        throw new Exception();
    }
    string IMyInterface.Method3()
    {
        throw new Exception();
    }
    #endregion
}
```

Implementing a skeletal interface also works with generic interfaces.



Visual Basic 2005 has only IntelliSense-level skeletal interface implementation, and it doesn't generate a region around the skeleton.

## Interface Refactoring

Code *refactoring* allows you to change the code structure without changing or affecting what the code itself actually does. Changing a variable name or packaging a few lines of code into a method are examples of code refactoring. Visual Studio 2005 contains a simple refactoring engine that enables several actions, including renaming types, variables, methods, or parameters; extracting a method out of a code section (and inserting a method call instead); encapsulating type members as properties; automating many formatting tasks; and auto-expanding common statements. The main difference between C# refactoring and doing a mere edit or find-and-replace is that you can harness the intelligence of the compiler to distinguish between code and comments, and so on.



In Visual Studio 2005, refactoring changes are limited to a single solution and do not propagate to client assemblies in different solutions.

You can invoke refactoring in two ways: you can select Refactor from the top-level Visual Studio 2005 menu, or you can select it from the pop-up context menu.

Of particular interest in the context of this chapter is the refactoring ability to extract an interface definition out of a set of public methods the type implements. For example, consider the following Calculator class:

```
public abstract class Calculator
{
    public int Add(int argument1,int argument2)
    {
        return argument1 + argument2;
    }
    public int Subtract(int argument1,int argument2)
    {
        return argument1 - argument2;
    }
    public virtual int Divide(int argument1,int argument2)
    {
        return argument1 + argument2;
    }
    public abstract int Multiply(int argument1,int argument2);
}
```

To extract an interface out of the Calculator class, right-click anywhere inside the class definition and select Extract Interface... from the Refactor menu. This will bring up the Extract Interface dialog box, shown in Figure 3-3.

The dialog box will propose a name for the interface: the type's name, prefixed with an I. The refactoring will use the default (also called *root*) namespace of the project, as configured in the project settings.

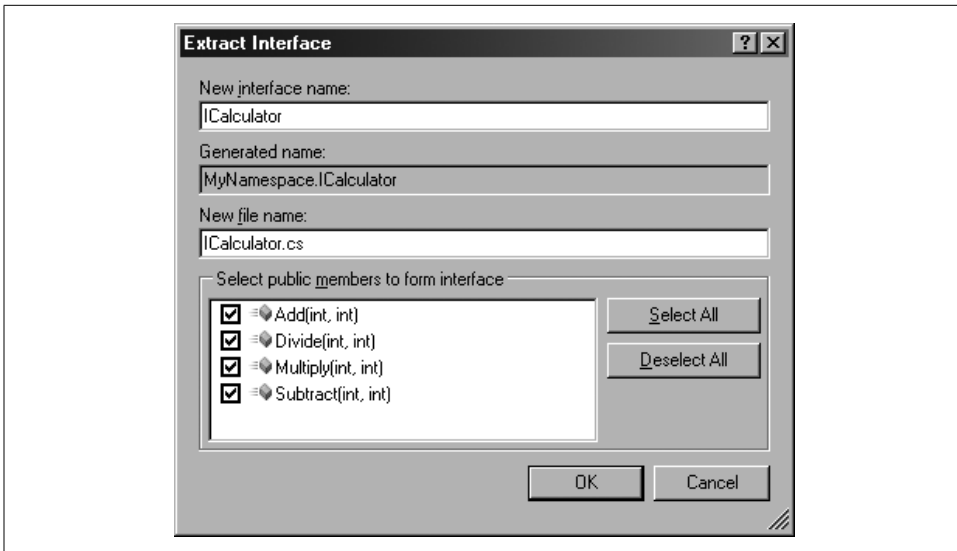


Figure 3-3. The Extract Interface dialog box

The interface will be extracted to a separate file, which will automatically be added to the project. You can provide a filename in the dialog. Finally, all the public methods (or properties) of the type will be listed in the dialog, regardless of whether they are public, virtual, or abstract. Note that when a class hierarchy is involved, the refactoring engine will only include public methods explicitly declared by the class or overridden by it. To include the suggested methods in the new interface definition, you must explicitly check the checkboxes to the left of each method. After you click the OK button, the new interface will be placed in the specified new file, and Visual Studio 2005 will add the interface derivation to the Calculator class, as shown here:

```
//In the file ICalculator.cs
interface ICalculator
{
    int Add(int argument1,int argument2);
    int Divide(int argument1,int argument2);
    int Multiply(int argument1,int argument2);
    int Subtruct(int argument1,int argument2);
}

//In the file Calculator.cs
public abstract class Calculator : ICalculator
{...}
```

Note that the extracted interface is internal, and that you have to explicitly make it public.

You can also extract one interface from the definition of another, in which case the new interface will be placed in a new file, but the original interface definition will not change (i.e., it will not inherit from the new interface). Visual Studio 2005 will not prefix the new interface name with an I, because that would result in a double II. Instead, it will append an ordinal number to the interface name.

Note that if the type already implements an interface implicitly, that interface's members will be included in the list in the Extract Interface dialog. Use explicit interface implementation on existing interfaces to avoid including them in the refactoring.