✓ **CONSIDER** naming factory types by concatenating the name of the type being created and `Factory`. For example, consider naming a factory type that creates `Control` objects `ControlFactory`.

The next section discusses when and how to design abstractions that might or might not support some features.

9.6 LINQ Support

Writing applications that interact with data sources, such as databases, XML documents, or Web Services, was made easier in the .NET Framework 3.5 with the addition of a set of features collectively referred to as LINQ (Language-Integrated Query). The following sections provide a very brief overview of LINQ and list guidelines for designing APIs related to LINQ support, including the so-called Query Pattern.

9.6.1 Overview of LINQ

Quite often, programming requires processing over sets of values. Examples include extracting the list of the most recently added books from a database of products, finding the e-mail address of a person in a directory service such as Active Directory, transforming parts of an XML document to HTML to allow for Web publishing, or something as frequent as looking up a value in a hashtable. LINQ allows for a uniform language-integrated programming model for querying datasets, independent of the technology used to store that data.

RICO MARIANI Like everything else, there are good and bad ways to use these patterns. The Entity Framework and LINQ to SQL offer good examples of how you can provide rich query semantics and still get very good performance using strong typing and by offering query compilation.

The Pit of Success notion is very important in LINQ implementations. I've seen some cases where the code that runs as a result of using a LINQ pattern is simply terrible in comparison to what you would write the conventional way. That's really not good enough—EF and LINQ to SQL let you write it nicely, and you get high-quality database interactions. That's what to aim for.

In terms of concrete language features and libraries, LINQ is embodied as:

- A specification of the notion of extension methods. These are described in detail in section 5.6.
- Lambda expressions, a language feature for defining anonymous delegates.
- New types representing generic delegates to functions and procedures: `Func<...>` and `Action<...>`.
- Representation of a delay-compiled delegate, the `Expression<...>` family of types.
- A definition of a new interface, `System.Linq.IQueryable<T>`.
- The Query Pattern, a specification of a set of methods a type must provide in order to be considered as a LINQ provider. A reference implementation of the pattern can be found in `System.Linq.Enumerable` class. Details of the pattern will be discussed later in this chapter.
- Query Expressions, an extension to language syntax allowing for queries to be expressed in an alternative, SQL-like format.

```
//using extension methods:
var names = set.Where(x => x.Age>20).Select(x=>x.Name);

//using SQL-like syntax:
var names = from x in set where x.Age>20 select x.Name;
```

MIRCEA TROFIN The interplay between these features is the following: Any `IEnumerable` can be queried upon using the LINQ extension methods, most of which require one or more lambda expressions as parameters; this leads to an in-memory generic evaluation of the queries. For cases where the set of data is not in memory (e.g., in a database) and/or queries may be optimized, the set of data is presented as an `IQueryable`. If lambda expressions are given as parameters, they are transformed by the compiler to `Expression<...>` objects. The implementation of `IQueryable` is responsible for processing said expressions. For example, the implementation of an `IQueryable` representing a database table would translate Expression objects to SQL queries.

9.6.2 Ways of Implementing LINQ Support

There are three ways by which a type can support LINQ queries:

- The type can implement `IEnumerable<T>` (or an interface derived from it).
- The type can implement `IQueryable<T>`.
- The type can implement the Query Pattern.

The following sections will help you choose the right method of supporting LINQ.

9.6.3 Supporting LINQ through `IEnumerable<T>`

✓ **DO** implement `IEnumerable<T>` to enable basic LINQ support.

Such basic support should be sufficient for most in-memory datasets. The basic LINQ support will use the extension methods on `IEnumerable<T>` provided in the .NET Framework. For example, simply define as follows:

```
public class RangeOfInt32s : IEnumerable<int> {
   public IEnumerator<int> GetEnumerator() {...}
   IEnumerator IEnumerable.GetEnumerator() {...}
}
```

Doing so allows for the following code, despite the fact that `RangeOfInt32s` did not implement a `Where` method:

```
var a = new RangeOfInt32s();
var b = a.Where(x => x>10);
```

RICO MARIANI Keeping in mind that you'll get your same enumeration semantics, and putting a LINQ façade on them does not make them execute any faster or use less memory.

✓ **CONSIDER** implementing `ICollection<T>` to improve performance of query operators.

For example, the `System.Linq.Enumerable.Count` method's default implementation simply iterates over the collection. Specific collection types can optimize their implementation of this method, since they often offer an `O(1)` - complexity mechanism for finding the size of the collection.

✓ **CONSIDER** supporting selected methods of `System.Linq.Enumerable` or the Query Pattern (see section 9.6.5) directly on new types implementing `IEnumerable<T>` if it is desirable to override the default `System.Linq.Enumerable` implementation (e.g., for performance optimization reasons).

9.6.4 Supporting LINQ through `IQueryable<T>`

✓ **CONSIDER** implementing `IQueryable<T>` when access to the query expression, passed to members of `IQueryable`, is necessary.

When querying potentially large datasets generated by remote processes or machines, it might be beneficial to execute the query remotely. An example of such a dataset is a database, a directory service, or Web service.

✗ **DO NOT** implement `IQueryable<T>` without understanding the performance implications of doing so.

Building and interpreting expression trees is expensive, and many queries can actually get slower when `IQueryable<T>` is implemented.

The trade-off is acceptable in the LINQ to SQL case, since the alternative overhead of performing queries in memory would have been far greater than the transformation of the expression to an SQL statement and the delegation of the query processing to the database server.

✓ **DO** throw `NotSupportedException` from `IQueryable<T>` methods that cannot be logically supported by your data source.

For example, imagine representing a media stream (e.g., an Internet radio stream) as an `IQueryable<byte>`. The `Count` method is not logically supported—the stream can be considered as infinite, and so the `Count` method should throw `NotSupportedException`.

9.6.5 Supporting LINQ through the Query Pattern

The Query Pattern refers to defining the methods in Figure 9-1 without implementing the `IQueryable<T>` (or any other LINQ interface).

Please note that the notation is not meant to be valid code in any particular language but to simply present the type signature pattern.

The notation uses `S` to indicate a collection type (e.g., `IEnumerable<T>`, `ICollection<T>`), and `T` to indicate the type of elements in that collection. Additionally, we use `O<T>` to represent subtypes of `S<T>` that are ordered. For example, `S<T>` is a notation that could be substituted with `IEnumerable<int>`, `ICollection<Foo>`, or even `MyCollection` (as long as the type is an enumerable type).

The first parameter of all the methods in the pattern (marked with `this`) is the type of the object the method is applied to. The notation uses extension-method-like syntax, but the methods can be implemented as extension methods or as member methods; in the latter case the first parameter should be omitted, of course, and the `this` pointer should be used.

Also, anywhere `Func<...>` is being used, pattern implementations may substitute `Expression<Func<...>>` for it. You can find guidelines later that describe when that is preferable.

```
S<T> Where(this S<T>, Func<T,bool>)

S<T2> Select(this S<T1>, Func<T1,T2>)
S<T3> SelectMany(this S<T1>, Func<T1,S<T2>>, Func<T1,T2,T3>)
S<T2> SelectMany(this S<T1>, Func<T1,S<T2>>)

O<T> OrderBy(this S<T>, Func<T,K>), where K is IComparable
O<T> ThenBy(this O<T>, Func<T,K>), where K is IComparable

S<T> Union(this S<T>, S<T>)
S<T> Take(this S<T>, int)
S<T> Skip(this S<T>, int)
S<T> SkipWhile(this S<T>, Func<T,bool>)

S<T3> Join(this S<T1>, S<T2>, Func<T1,K1>, Func<T2,K2>,
Func<T1,T2,T3>)

T ElementAt(this S<T>,int)
```

FIGURE 9-1: Query Pattern Method Signatures

✓ **DO** implement the Query Pattern as instance members on the new type, if the members make sense on the type even outside of the context of LINQ. Otherwise, implement them as extension methods.

For example, instead of the following:

```
public class MyDataSet<T>:IEnumerable<T>{...}
...
public static class MyDataSetExtensions{
     public static MyDataSet<T> Where(this MyDataSet<T> data, Func<T,bool>
query){...}
}
```

Prefer the following, because it's completely natural for datasets to support `Where` methods:

```
public class MyDataSet<T>:IEnumerable<T>{
     public MyDataSet<T> Where(Func<T,bool> query){...}
       ...
}
```

✓ **DO** implement `IEnumerable<T>` on types implementing the Query Pattern.

✓ **CONSIDER** designing the LINQ operators to return domain-specific enumerable types. Essentially, one is free to return anything from a `Select` query method; however, the expectation is that the query result type should be at least enumerable.

This allows the implementation to control which query methods get executed when they are chained. Otherwise, consider a user-defined type `MyType`, which implements `IEnumerable<T>`. `MyType` has an optimized `Count` method defined, but the return type of the `Where` method is `IEnumerable<T>`. In the example here, the optimization is lost after the `Where` method is called; the method returns `IEnumerable<T>`, and so the built-in `Enumerable.Count` method is called, instead of the optimized one defined on `MyType`.

```
var result = myInstance.Where(query).Count();
```

✗ **AVOID** implementing just a part of the Query Pattern if fallback to the basic `IEnumerable<T>` implementations is undesirable.

For example, consider a user-defined type `MyType`, which implements `IEnumerable<T>`. `MyType` has an optimized `Count` method defined but does not have `Where`. In the example here, the optimization is lost after the `Where` method is called; the method returns `IEnumerable<T>`, and so the built-in `Enumerable.Count` method is called, instead of the optimized one defined on `MyType`.

```
var result = myInstance.Where(query).Count();
```

✓ **DO** represent ordered sequences as a separate type, from its unordered counterpart. Such types should define `ThenBy` method.

This follows the current pattern in the `LINQ to Objects` implementation and allows for early (compile-time) detection of errors such as applying `ThenBy` to an unordered sequence.

For example, the Framework provides the `IOrderedEnumerable<T>` type, which is returned by `OrderBy`. The `ThenBy` extension method is defined for this type, and not for `IEnumerable<T>`.

✓ **DO** defer execution of query operator implementations. The expected behavior of most of the Query Pattern members is that they simply construct a new object which, upon enumeration, produces the elements of the set that match the query.

The following methods are exceptions to this rule: `All`, `Any`, `Average`, `Contains`, `Count`, `ElementAt`, `Empty`, `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Max`, `Min`, `Single`, `Sum`.

In the example here, the expectation is that the time necessary for evaluating the second line will be independent from the size or nature (e.g., in-memory or remote server) of `set1`. The general expectation is that this line simply prepares `set2`, delaying the determination of its composition to the time of its enumeration.

```
var set1 = ...
var set2 = set1.Select(x => x.SomeInt32Property);
foreach(int number in set2){...} // this is when actual work happens
```

✓ **DO** place query extensions methods in a "`Linq`" subnamespace of the main namespace. For example, extension methods for `System.Data` features reside in `System.Data.Linq` namespace.

✓ **DO** use `Expression<Func<...>>` as a parameter instead of `Func<...>` when it is necessary to inspect the query. See section 9.6.5 for more details.

As discussed earlier, interacting with an SQL database is already done through `IQueryable<T>` (and therefore expressions) rather than `IEnumerable<T>`, since this gives an opportunity to translate lambda expressions to SQL expressions.

An alternative reason for using expressions is performing optimizations. For example, a sorted list can implement look-up (`Where` clauses) with binary search, which can be much more efficient than the standard `IEnumerable<T>` or `IQueryable<T>` implementations.

9.7 Optional Feature Pattern

When designing an abstraction, you might want to allow cases in which some implementations of the abstraction support a feature or a behavior, whereas other implementations do not. For example, stream implementations can support reading, writing, seeking, or any combination thereof.

One way to model these requirements is to provide a base class with APIs for all nonoptional features and a set of interfaces for the optional features. The interfaces are implemented only if the feature is actually supported by a concrete implementation. The following example shows one of many ways to model the stream abstraction using such an approach.

```
// framework APIs
public abstract class Stream {
   public abstract void Close();
   public abstract int Position { get; }
}
public interface IInputStream {
   byte[] Read(int numberOfBytes);
}
```

```
public interface IOutputStream {
   void Write(byte[] bytes);
}
public interface ISeekableStream {
   void Seek(int position);
}
public interface IFiniteStream {
   int Length { get; }
   bool EndOfStream { get; }
}

// concrete stream
public class FileStream : Stream, IOutputStream, IInputStream,
ISeekableStream, IFiniteStream {
   ...
}

// usage
void OverwriteAt(IOutputStream stream, int position, byte[] bytes){
   // do dynamic cast to see if the stream is seekable
   ISeekableStream seekable = stream as ISeekableStream;
   if(seekable==null){
      throw new NotSupportedException(...);
   }
   seekable.Seek(position);
   stream.Write(bytes);
}
```

You will notice the .NET Framework's `System.IO` namespace does not follow this model, and with good reason. Such factored design requires adding many types to the framework, which increases general complexity. Also, using optional features exposed through interfaces often requires dynamic casts, and that in turn results in usability problems.

KRZYSZTOF CWALINA Sometimes framework designers provide interfaces for common combinations of optional interfaces. For example, the `OverwriteAt` method would not have to use the dynamic cast if the framework design provided `ISeekableOutputStream`. The problem with this approach is that it results in an explosion of the number of different interfaces for all combinations.

Sometimes the benefits of factored design are worth the drawbacks, but often they are not. It is easy to overestimate the benefits and underestimate the drawbacks. For example, the factorization did not help the developer who wrote the `OverwriteAt` method avoid runtime exceptions (the main reason for factorization). It is our experience that many designs incorrectly err on the side of too much factorization.

The Optional Feature Pattern provides an alternative to excessive factorization. It has drawbacks of its own but should be considered as an alternative to the factored design described previously. The pattern provides a mechanism for discovering whether the particular instance supports a feature through a query API and uses the features by accessing optionally supported members directly through the base abstraction.

```
// framework APIs
public abstract class Stream {
   public abstract void Close();
   public abstract int Position { get; }

   public virtual bool CanWrite { get { return false; } }
   public virtual void Write(byte[] bytes){
      throw new NotSupportedException(...);
   }

   public virtual bool CanSeek { get { return false; } }
   public virtual void Seek(int position){
      throw new NotSupportedException(...);
   }
   ... // other options
}

// concrete stream
public class FileStream : Stream {
   public override bool CanSeek { get { return true; } }
   public override void Seek(int position) { ... }
   ...
}

// usage
void OverwriteAt(Stream stream, int position, byte[] bytes){
   if(!stream.CanSeek || !stream.CanWrite){
      throw new NotSupportedException(...);
   }
   stream.Seek(position);
   stream.Write(bytes);
}
```

In fact, the `System.IO.Stream` class uses this design approach. Some abstractions might choose to use a combination of factoring and the Optional Feature Pattern. For example, the Framework collection interfaces are factored into indexable and nonindexable collections (`IList<T>` and `ICollection<T>`), but they use the Optional Feature Pattern to differentiate between read-only and read-write collections (`ICollection<T>.IsReadOnly property`).

✓ **CONSIDER** using the Optional Feature Pattern for optional features in abstractions.

The pattern minimizes the complexity of the framework and improves usability by making dynamic casts unnecessary.

> **STEVE STARCK** If your expectation is that only a very small percentage of classes deriving from the base class or interface would actually implement the optional feature or behavior, using interface-based design might be better. There is no real need to add additional members to all derived classes when only one of them provides the feature or behavior. Also, factored design is preferred in cases when the number of combinations of the optional features is small and the compile-time safety afforded by factorization is important.

✓ **DO** provide a simple Boolean property that clients can use to determine whether an optional feature is supported.

```
public abstract class Stream {
   public virtual bool CanSeek { get { return false; } }
   public virtual void Seek(int position){ ... }
}
```

Code that consumes the abstract base class can query this property at runtime to determine whether it can use the optional feature.

```
if(stream.CanSeek){
   stream.Seek(position);
}
```

✓ **DO** use virtual methods on the base class that throw `NotSupported-Exception` to define optional features.

```
public abstract class Stream {
   public virtual bool CanSeek { get { return false; } }
   public virtual void Seek(int position){
      throw new NotSupportedException(...);
   }
}
```

The method can be overridden by subclasses to provide support for the optional feature. The exception should clearly communicate to the user that the feature is optional and which property the user should query to determine if the feature is supported.

9.8 Simulating Covariance

Different constructed types don't have a common root type. For example, there would not be a common representation of `IEnumerable<string>` and `IEnumerable<object>` if not for a pattern implemented by `IEnumerable<T>` called Simulated Covariance. This section describes the details of the pattern.

Generics is a very powerful type system feature added to the .NET Framework 2.0. It allows creation of so-called parameterized types. For example, `List<T>` is such a type and it represents a list of objects of type `T`. The `T` is specified at the time when the instance of the list is created.

```
var names = new List<string>();
names.Add("John Smith");
names.Add("Mary Johnson");
```

Such generic data structures have many benefits over their nongeneric counterparts. But they also have some—sometimes surprising—limitations. For example, some users expect that a `List<string>` can be cast to `List<object>`, just as a `String` can be cast to `Object`. But unfortunately, the following code won't even compile.

```
List<string> names = new List<string>();
List<object> objects = names; // this won't compile
```

There is a very good reason for this limitation, and that is to allow for full strong typing. For example, if you could cast `List<string>` to a `List<object>` the following incorrect code would compile, but the program would fail at runtime.

```
static void Main(){
   var names = new List<string>();

   // this of course does not compile, but if it did
   // the whole program would compile, but would be incorrect as it
   // attempts to add arbitrary objects to a list of strings.
   AddObjects((List<object>)names);

   string name = names[0]; // how could this work?
}

// this would (and does) compile just fine.
static void AddObjects(List<object> list){
   list.Add(new object()); // it's a list of strings, really. Should we throw?
   list.Add(new Button());
}
```

Unfortunately, this limitation can also be undesired in some scenarios. For example, let's consider the following type:

```
public class CountedReference<T> {
   public CountedReference(T value);
   public T Value { get; }
   public int Count { get; }
   public void AddReference();
   public void ReleaseReference();
}
```

There is nothing wrong with casting a `CountedReference<string>` to `CountedReference<object>`, as in the following example.

```
var reference = new CountedReference<string>(...);
CountedReference<object> obj = reference; // this won't compile
```

In general, having a way to represent any instance of this generic type is very useful.

```
// what type should ??? be?
// CountedReference<object> would be nice but it won't work
static void PrintValue(??? anyCountedReference){
   Console.WriteLine(anyCountedReference.Value);
}
```

KRZYSZTOF CWALINA Of course, `PrintValue` could be a generic method taking `CountedReference<T>` as the parameter.

```
static void PrintValue<T>(CountedReference<T> any){
   Console.WriteLine(any.Value);
}
```

This would be a fine solution in many cases. But it does not work as a general solution and might have negative performance implications. For example, the trick does not work for properties. If a property needed to be typed as "any reference," you could not use `CountedReference<T>` as the type of the property. In addition, generic methods might have undesirable performance implications. If such generic methods are called with many differently sized type arguments, the runtime will generate a new method for every argument size. This might introduce unacceptable memory consumption overhead.

Unfortunately, unless `CountedReference<T>` implemented the Simulated Covariance Pattern described next, the only common representation of all `CountedReference<T>` instances would be `System.Object`. But `System.Object` is too limiting and would not allow the `PrintValue` method to access the `Value` property.

The reason that casting to `CountedReference<object>` is just fine, but casting to `List<object>` can cause all sorts of problems, is that in case of `CountedReference<object>`, the object appears only in the output position (the return type of `Value` property). In the case of `List<object>`, the object represents both output and input types. For example, `object` is the type of the input to the `Add` method.

```
// T does not appear as input to any members except the constructor
public class CountedReference<T> {
   public CountedReference(T value);
   public T Value { get; }
   public int Count { get; }
   public void AddReference();
   public void ReleaseReference();
}

// T does appear as input to members of List<T>
public class List<T> {
   public void Add(T item); // T is an input here
   public T this[int index]{
   get;
   set; // T is actually an input here
}
}
```

In other words, we say that in `CountedReference<T>`, the `T` is at covariant positions (outputs). In `List<T>`, the `T` is at covariant and contravariant (inputs) positions.

To solve the problem of not having a common type representing the root of all constructions of a generic type, you can implement what's called the Simulated Covariance Pattern.

Consider a generic type (class or interface) and its dependencies described in the code fragment that follows.

```
public class Foo<T> {
    public T Property1 { get; }
    public T Property2 { set; }
    public T Property3 { get; set; }
    public void Method1(T arg1);
    public T Method2();
    public T Method3(T arg);
    public Type1<T> GetMethod1();
    public Type2<T> GetMethod2();
}
public class Type1<T> {
   public T Property { get; }
}
public class Type2<T> {
   public T Property { get; set; }
}
```

Create a new interface (root type) with all members containing a `T` at contravariant positions removed. In addition, feel free to remove all members that might not make sense in the context of the trimmed-down type.

```
public interface IFoo<out T> {
   T Property1 { get; }
   T Property3 { get; } // setter removed
   T Method2();
   Type1<T> GetMethod1();
   IType2<T> GetMethod2(); // note that the return type changed
}
public interface IType2<T> {
   T Property { get; } // setter removed
}
```

The generic type should then implement the interface explicitly and "add back" the strongly typed members (using `T` instead of `object`) to its public API surface.

```
public class Foo<T> : IFoo<object> {
   public T Property1 { get; }
   public T Property2 { set; }
   public T Property3 { get; set;}
   public void Method1(T arg1);
   public T Method2();
   public T Method3(T arg);
   public Type1<T> GetMethod1();
   public Type2<T> GetMethod2();

   object IFoo<object>.Property1 { get; }
   object IFoo<object>.Property3 { get; }
   object IFoo<object>.Method2() { return null; }
   Type1<object> IFoo<object>.GetMethod1();
   IType2<object> IFoo<object>.GetMethod2();
}

public class Type2<T> : IType2<object> {
   public T Property { get; set; }
   object IType2<object>.Property { get; }
}
```

Now, all constructed instantiations of `Foo<T>` have a common root type `IFoo<object>`.

```
var foos = new List<IFoo<object>>();
foos.Add(new Foo<int>());
foos.Add(new Foo<string>());
```

```
...
foreach(IFoo<object> foo in foos){
   Console.WriteLine(foo.Property1);
   Console.WriteLine(foo.GetMethod2().Property);
}
```

In the case of the simple `CountedReference<T>`, the code would look like the following:

```
public interface ICountedReference<out T> {
   T Value { get; }
   int Count { get; }
   void AddReference();
   void ReleaseReference();
}

public class CountedReference<T> : ICountedReference<object> {
   public CountedReference(T value) {...}
   public T Value { get { ... } }
   public int Count { get { ... } }
   public void AddReference(){...}
   public void ReleaseReference(){...}

  object ICountedReference<object>.Value { get { return Value; } }
}
```

✓ **CONSIDER** using the Simulated Covariance Pattern if there is a need to have a representation for all instantiations of a generic type.

The pattern should not be used frivolously, because it results in additional types in the framework and can makes the existing types more complex.

✓ **DO** ensure that the implementation of the root's members is equivalent to the implementation of the corresponding generic type members.

There should not be an observable difference between calling a member on the root type and calling the corresponding member on the generic type. In many cases, the members of the root are implemented by calling members on the generic type.

```
public class Foo<T> : IFoo<object> {

  public T Property3 { get { ... } set { ... } }
  object IFoo<object>.Property3 { get { return Property3; } }
...
}
```

✓ **CONSIDER** using an abstract class instead of an interface to represent the root.

This might sometimes be a better option, because interfaces are more difficult to evolve (see section 4.3). On the other hand, there are some problems with using abstract classes for the root. Abstract class members cannot be implemented explicitly and the subtypes need to use the `new` modifier. This makes it tricky to implement the root's members by delegating to the generic type members.

✓ **CONSIDER** using a nongeneric root type if such type is already available.

For example, `List<T>` implements `IEnumerable` for the purpose of simulating covariance.

9.9 Template Method

The Template Method Pattern is a very well-known pattern described in much greater detail in many sources, such as the classic book *Design Patterns* by Gamma et al. Its intent is to outline an algorithm in an operation. The Template Method Pattern allows subclasses to retain the algorithm's structure while permitting redefinition of certain steps of the algorithm. We are including a simple description of this pattern here, because it is one of the most commonly used patterns in API frameworks.

The most common variation of the pattern consists of one or more nonvirtual (usually public) members that are implemented by calling one or more protected virtual members.

```
public Control{
   public void SetBounds(int x, int y, int width, int height){
      ...
      SetBoundsCore (...);
   }

   public void SetBounds(int x, int y, int width, int
   height, BoundsSpecified specified){
       ...
       SetBoundsCore (...);
   }
   protected virtual void SetBoundsCore(int x, int y, int width, int
   height, BoundsSpecified specified){
        // Do the real work here.
   }
}
```

The goal of the pattern is to control extensibility. In the preceding example, the extensibility is centralized to a single method (a common mistake is to make more than one overload virtual). This helps to ensure that the semantics of the overloads stay consistent, because the overloads cannot be overridden independently.

Also, public virtual members basically give up all control over what happens when the member is called. This pattern is a way for the base class designer to enforce some structure of the calls that happen in the member. The nonvirtual public methods can ensure that certain code executes before or after the calls to virtual members and that the virtual members execute in a fixed order.

As a framework convention, the protected virtual methods participating in the Template Method Pattern should use the suffix "Core."

✗ **AVOID** making public members virtual.

If a design requires virtual members, follow the template pattern and create a protected virtual member that the public member calls. This practice provides more controlled extensibility.

✓ **CONSIDER** using the Template Method Pattern to provide more controlled extensibility.

In this pattern, all extensibility points are provided through protected virtual members that are called from nonvirtual members.

✓ **CONSIDER** naming protected virtual members that provide extensibility points for nonvirtual members by suffixing the nonvirtual member name with "Core."

```
public void SetBounds(...){
   ...
   SetBoundsCore (...);
}
protected virtual void SetBoundsCore(...){ ... }
```

BRIAN PEPIN I like to take the template pattern one step further and implement all argument checking in the nonvirtual public method. This way I can stop garbage entering methods that were possibly overridden by another developer, and it helps to enforce a little more of the API contract across implementations.

The next section goes into designing APIs that need to support timeouts.

9.10 Timeouts

Timeouts occur when an operation returns before its completion because the maximum time allocated for the operation (timeout time) has elapsed. The user often specifies the timeout time. For example, it might take a form of a parameter to a method call.

```
server.PerformOperation(timeout);
```

An alternative approach is to use a property.

```
server.Timeout = timeout;
server.PerformOperation();
```

The following short list of guidelines describes best practices for the design of APIs that need to support timeouts.

✓ **DO** prefer method parameters as the mechanism for users to provide timeout time.

Method parameters are favored over properties because they make the association between the operation and the timeout much more apparent. The property-based approach might be better if the type is designed to be a component used with visual designers.

✓ **DO** prefer using `TimeSpan` to represent timeout time.

Historically, timeouts have been represented by integers. Integer timeouts can be hard to use for the following reasons:

- It is not obvious what the unit of the timeout is.
- It is difficult to translate units of time into the commonly used millisecond. (How many milliseconds are in 15 minutes?)

Often, a better approach is to use `TimeSpan` as the timeout type. `TimeSpan` solves the preceding problems.

```
class Server {
   void PerformOperation(TimeSpan timeout){
      ...
   }
}

var server = new Server();
server.PerformOperation(new TimeSpan(0,15,0));
```

Integer timeouts are acceptable if:

- The parameter or property name can describe the unit of time used by the operation, for example, if a parameter can be called `milliseconds` without making an otherwise self-describing API cryptic.
- The most commonly used value is small enough that users won't have to use calculators to determine the value, for example, if the unit is milliseconds and the commonly used timeout is less than 1 second.

✓ **DO** throw `System.TimeoutException` when a timeout elapses.

Timeout equal to `TimeSpan(0)` means that the operation should throw if it cannot complete immediately. If the timeout equals `TimeSpan.MaxValue`, the operation should wait forever without timing out. Operations are not required to support either of these values, but they should throw an `InvalidArgumentException` if an unsupported timeout value is specified.

If a timeout expires and the `System.TimeoutException` is thrown, the server class should cancel the underlying operation.

In the case of an asynchronous operation with a timeout, the callback should be called and an exception thrown when the results of the operation are first accessed.

```
void OnReceiveCompleted(Object source, ReceiveCompletedEventArgs
asyncResult){
   MessageQueue queue = (MessageQueue)source;
   // the following line will throw if BeginReceive has timed out
   Message message = queue.EndReceive(asyncResult.AsyncResult);
   Console.WriteLine("Message: " + (string)message.Body);
   queue.BeginReceive(new TimeSpan(1,0,0));
}
```

For more information on timeouts and asynchronous operation, see section 9.2.

✗ **DO NOT** return error codes to indicate timeout expiration.

Expiration of a timeout means the operation could not complete successfully and thus should be treated and handled as any other runtime error (see Chapter 7).

9.11 XAML Readable Types

XAML is an XML format used by WPF (and other technologies) to represent object graphs. The following guidelines describe design considerations for ensuring that your types can be created using XAML readers.

✓ **CONSIDER** providing the default constructor if you want a type to work with XAML.

For example, consider the following XAML markup:

```
<Person Name="John" Age="22" />
```

It is equivalent to the following C# code:

```
new Person() { Name = "John", Age = 22 };
```

Consequently, for this code to work, the `Person` class needs to have a default constructor. Markup extensions, discussed in the next guideline in this section, are an alternative way of enabling XAML.

CHRIS SELLS In my opinion, this one should really be a DO, not a CONSIDER. If you're designing a new type to support XAML, it's far preferable to do it with a default constructor than with markup extensions or type converters.

✓ **DO** provide markup extension if you want an immutable type to work with XAML readers.

Consider the following immutable type:

```
public class Person {
      public Person(string name, int age){
         this.name = name;
         this.age = age;
      }
      public string Name { get { return name; } }
      public int Age { get { return age; } }

      string name;
      int age;
}
```

Properties of such type cannot be set using XAML markup, because the reader does not know how to initialize the properties using the parameterized constructor. Markup extensions address the problem.

```
[MarkupExtensionReturnType(typeof(Person))]
public class PersonExtension : MarkupExtension {
   public string Name { get; set; }
   public int Age { get; set; }

   public override object ProvideValue(IServiceProvider serviceProvider){
      return new Person(this.Name,this.Age);
   }
}
```

Keep in mind that immutable types cannot be written using XAML writers.

✗ **AVOID** defining new type converters unless the conversion is natural and intuitive. In general, limit type converter usage to the ones already provided by the .NET Framework.

Type converters are used to convert a value from a string to the appropriate type. They're used by XAML infrastructure and in other places, such as graphical designers. For example, the string "#FFFF0000" in the following markup gets converted to an instance of a red `Brush` thanks to the type converter associated with the `Rectangle.Fill` property.

```
<Rectangle Fill="#FFFF0000"/>
```

But type converters can be defined too liberally. For example, the Brush type converter should not support specifying gradient brushes, as shown in the following hypothetical example.

```
<Rectangle Fill="HorizontalGradient White Red" />
```

Such converters define new "minilanguages," which add complexity to the system.

✓ **CONSIDER** applying the `ContentPropertyAttribute` to enable convenient XAML syntax for the most commonly used property.

```
[ContentProperty("Image")]
public class Button {
   public object Image { get; set; }
}
```

The following XAML syntax would work without the attribute:

```
<Button>
   <Button.Image>
      <Image Source="foo.jpg"
   </Button.Image>
</Button>
```

The attribute makes the following much more readable syntax possible.

```
<Button>
   <Image Source="foo.jpg"
</Button>
```

9.12 And in the End...

The process of creating a great framework is demanding. It requires dedication, knowledge, practice, and a lot of hard work. But in the end, it can be one of the most fulfilling jobs software engineers ever get to do. Large system frameworks can enable millions to build software that was not possible before. Application extensibility frameworks can turn simple applications into powerful platforms and make them shine. Finally, reusable component frameworks can inspire and enable developers to take their applications beyond the ordinary. When you create a framework like that, please let us know. We would like to congratulate you.