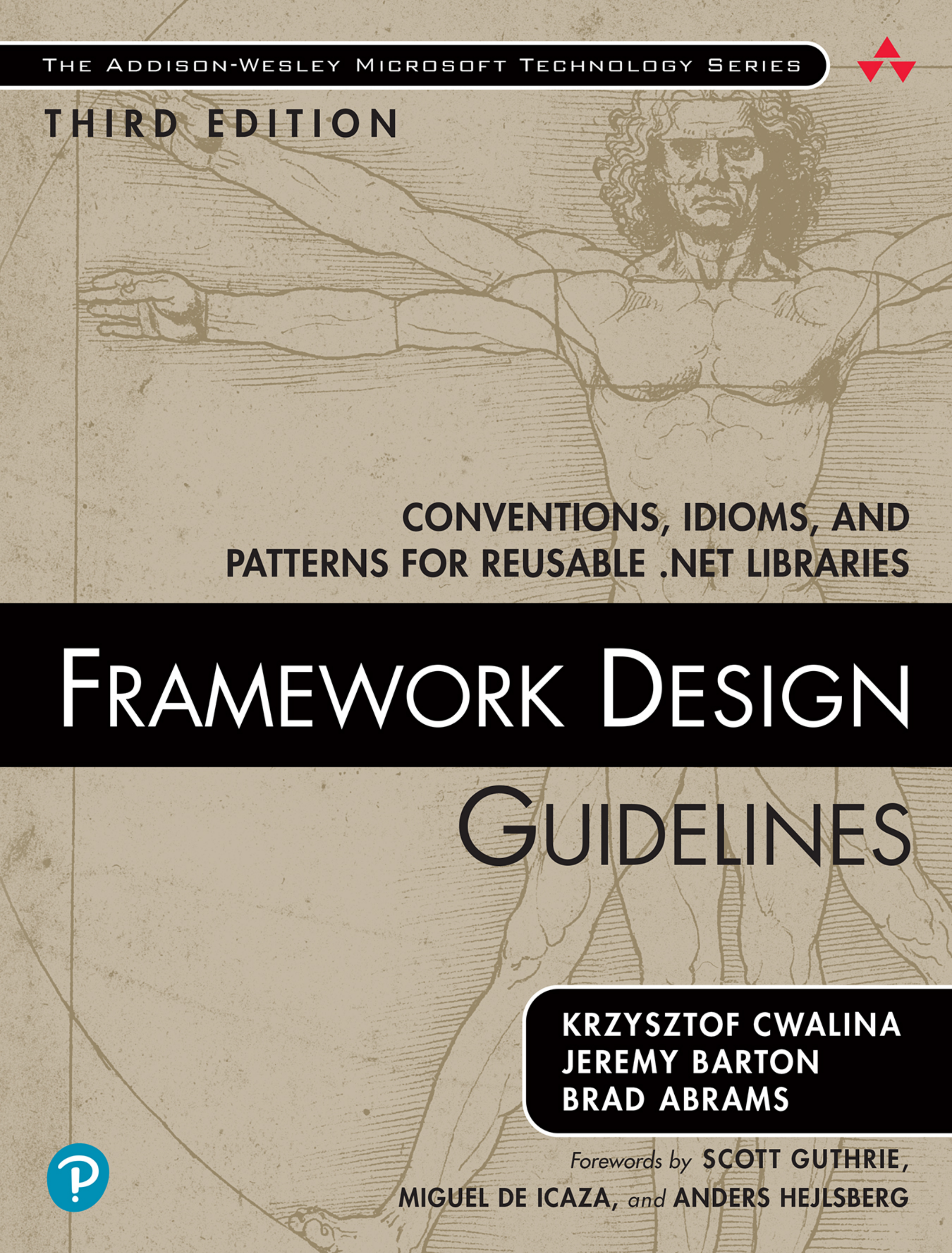
THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES

THIRD EDITION



CONVENTIONS, IDIOMS, AND PATTERNS FOR REUSABLE .NET LIBRARIES

FRAMEWORK DESIGN GUIDELINES

KRZYSZTOF CWALINA
JEREMY BARTON
BRAD ABRAMS

Forewords by SCOTT GUTHRIE, MIGUEL DE ICAZA, *and* ANDERS HEJLSBERG

Framework Design Guidelines
Third Edition

✓ **DO** provide a simple overload, with no default parameters, for any method with two or more defaulted parameters.

Usability studies have shown that default parameters are a comparatively advanced feature. Many developers are confused with the presentation in IntelliSense, and sometimes think that they have to provide the defaulted arguments as exactly the value shown by IntelliSense. Providing the simple overload makes your method easier to approach for users who need only the default behaviors.

```
public static BigInteger Parse(ReadOnlySpan<char> value)
   // Since a provider is 'specified' this calls the overload
   // which uses default parameters.
   => Parse(value, provider: null);

public static BigInteger Parse(
   ReadOnlySpan<char> value,
   NumberStyles style = NumberStyles.Integer,
   IFormatProvider provider = null) { ... }
```

✗ **DO NOT** use default parameters, for any parameter type other than `CancellationToken`, on interface methods or virtual methods on classes.

A disagreement between an interface declaration and the implementation on a default value creates an unnecessary source of confusion for your users. The same confusion also arises when a virtual method and an override of that method disagree on the default values for a parameter. The most straightforward solution to this problem is to avoid the situation, and only use default parameters on nonvirtual methods.

`CancellationToken` is specifically exempted from this guideline, because the guidance for asynchronous methods (section 9.2) is to always have a `CancellationToken` parameter with a default value. Since there's only one legal default value for a `CancellationToken`—that is, `default(CancellationToken)`—no default value disagreement is possible between implementations.

```
public interface IExample {
  void PrintValue(int value = 5);
}
```

```
public class Example : IExample {
  public virtual void PrintValue(int value = 10) {
    Console.WriteLine(value);
  }
}
public class DerivedExample : Example {
  public override void PrintValue(int value = 20) {
    base.PrintValue(value);
  }
}

...

// What gets printed?
// If you have to think about it, the API is not self-documenting.
DerivedExample derived = new DerivedExample();
Example e = derived;
IExample ie = derived;
derived.PrintValue();
e.PrintValue();
ie.PrintValue();
```

For types that use the Template Method Pattern (section 9.9), the longest overload of the public nonvirtual method can use default parameters when appropriate, then defer to the virtual implementation method without using default parameters.

```
public partial class Control {
  // The public method uses a default parameter
  public void SetBounds(
    int x,
    int y,
    int width,
    int height,
    BoundsSpecified specified = BoundsSpecified.All) {
    ...
    SetBoundsCore(x, y, width, height, specified);
  }

  // The protected (virtual) method does not use default parameters
  protected virtual void SetBoundsCore(
    int x,
    int y,
    int width,
    int height,
    BoundsSpecified specified) {
    // Do the real work here.
  }
}
```

Default parameters can be provided for interface methods via extension methods.

```
public interface IExample {
  void PrintValue(int value);
}

public static class ExampleExtensions : IExample {
  public static void PrintValue(
    this IExample example,
    int value = 10) {

    // While this may look like a recursive call,
    // the C# method resolution rules say the instance member
    // from the interface is a better match than this extension
    // method.
    example.PrintValue(value);
  }
```

One proposed alternative to not using default parameters in virtual methods was to say that virtual overrides should always match the defaults from the original method declaration. However, that guideline leads to versioning problems when derived types exist in a different assembly:

- Changing the last required parameter of a method to be a default parameter is generally not a breaking change, but would cause the override to provide a different set of defaults.
- The guidance for adding a new default parameter to an existing method (provided next) calls for removing the defaults from the current method overload, which can result in ambiguous method compile-time failures due to the overrides that have yet to be updated.

✗ **DO NOT** change the default value for a parameter once it has been publicly released.

Default parameters do not create logical overloads in the assembly that defines them, but are just a convenience mechanism to enable the compiler to fill in any missing values for the calling code at compile-time. When the value changes between two releases of a library, any existing callers will use the old value until they recompile, at which point they will switch to the new value. This can lead to confusion when diagnosing issues raised by users, when their reproduction cases don't exhibit the bad behavior due to the changed default value.

If you want to be able to change the value over time, use an otherwise illegal sentinel value such as zero or `-1` to indicate the runtime default value should be used instead.

```
// In this method, the style parameter should continue to use
// NumberStyles.Integer in all future versions of the library.
//
// Since the provider parameter has a default value of null,
// the library can change the default as an implementation detail
// (assuming that's an acceptable breaking change for the method).
public static BigInteger Parse(
    ReadOnlySpan<char> value,
    NumberStyles style = NumberStyles.Integer,
    IFormatProvider provider = null) { ... }
```

✗ **DO NOT** have two overloads of a method with "compatible" required parameters that both use default parameters.

✗ **AVOID** having two overloads of the same method that both use default parameters.

✗ **DO NOT** have different defaults for the same parameter in two overloads of the same method.

The only time that two overloads of the same method should both have default parameters is when their required parameters have incompatible signatures. The parameters shared in common by the two overloads should have the same default values, or no default value. This makes it abundantly clear which overload is being called.

```
// Given this overload already exists
public static OperationStatus DecodeFromUtf8(
  ReadOnlySpan<byte> utf8,
  Span<byte> bytes,
  out int bytesConsumed,
  out int bytesWritten,
  bool isFinalBlock = true) { ... }

// OK: default parameters have the same value,
// and the signatures are incompatible
public static byte[] DecodeFromUtf8(
  byte[] utf8,
  out int bytesConsumed,
  bool isFinalBlock = true) { ... }

// BAD: default parameters have a different value
public static byte[] DecodeFromUtf8(
  byte[] utf8,
  out int bytesConsumed,
  bool isFinalBlock = false) { ... }

// BAD: This longer overload's required parameters are compatible
// with the original overload
public static OperationStatus DecodeFromUtf8(
  ReadOnlySpan<byte> utf8,
  Span<byte> bytes,
  out int bytesConsumed,
  out int bytesWritten,
  bool isFinalBlock = true,
  int bufferSize = 512) { ... }
```

✓ **DO** move all default parameters to the new, longer overload when adding optional parameters to an existing method.

In the previous example, when adding the `bufferSize` parameter, the existing `DecodeFromUtf8` method should change to no longer have a default value for the `isFinalBlock` parameter when the new overload is added. This method cannot be deleted without introducing a runtime breaking change (a `MissingMethodException`) in upgrade scenarios. Since having a simple parameter after an `out` parameter makes this `DecodeFromUtf8` overload no longer conform to design guidelines, it is advisable to attribute it as `[EditorBrowsable(EditorBrowsable State.Never)]`. IntelliSense will only show the new overload with the additional optional parameters.

```
// This overload no longer has a default value for isFinalBlock,
// and has been marked as hidden from IntelliSense.
[EditorBrowsable(EditorBrowsableState.Never)]
public static OperationStatus DecodeFromUtf8(
  ReadOnlySpan<byte> utf8,
  Span<byte> bytes,
  out int bytesConsumed,
  out int bytesWritten,
  bool isFinalBlock) { /* call the longer overload */ }

public static OperationStatus DecodeFromUtf8(
  ReadOnlySpan<byte> utf8,
  Span<byte> bytes,
  out int bytesConsumed,
  out int bytesWritten,
  bool isFinalBlock = true,
  int bufferSize = 512) { ... }
```

5.1.2 Implementing Interface Members Explicitly

Explicit interface member implementation allows an interface member to be implemented so that it is only callable when the instance is cast to the interface type. For example, consider the following definition:

```
public struct Int32 : IConvertible {
   int IConvertible.ToInt32 () {..}
    ...
}

// calling ToInt32 defined on Int32
int i = 0;
i.ToInt32(); // does not compile
((IConvertible)i).ToInt32(); // works just fine
```

In general, implementing interface members explicitly is straightforward and follows the same general guidelines as those for methods, properties, or events. However, some specific guidelines apply to implementing interface members explicitly, as described next.