Framework Design Guidelines Digest

Krzysztof Cwalina

April 9, 2008

1 Introduction

This document is a distillation and a simplification of the most basic guidelines described in detail in a book titled Framework Design Guidelines by Krzysztof Cwalina and Brad Abrams.

Framework Design Guidelines were created in the early days of .NET Framework development. They started as a small set of naming and design conventions but have been enhanced, scrutinized, and refined to a point where they are generally considered the canonical way to design frameworks at Microsoft. They carry the experience and cumulative wisdom of thousands of developer hours over several versions of the .NET Framework.

2 General Design Principles

Scenario Driven Design
Start the design process of your public API by defining the top scenarios for each feature area. Write code you would like the end users to write when they implement these scenarios using your API. Design your API based on the sample code you wrote. For example, when designing API to measure elapsed time, you may write the following scenario code samples:

```
// scenario #1 : measure time elapsed
Stopwatch watch = Stopwatch.StartNew();
DoSomething();
Console.WriteLine(watch.Elapsed);

// scenario #2 : reuse stopwatch
Dim watch As Stopwatch = Stopwatch.StartNew()
```

```
DoSomething();
Console.WriteLine(watch.ElapsedMilliseconds)

watch.Reset()
watch.Start()
DoSomething()
Console.WriteLine(watch.Elapsed)

// scenario #3: ...
```

Usability Studies
Test usability of your API. Choose developers who are not familiar with your API and have them implement the main scenarios. Try to identify which parts of your API are not intuitive.

Self Documenting API
Developers using your API should be able to implement main scenarios without reading the documentation. Help users to discover what types they need to use in main scenarios and what the semantics of the main methods are by choosing intuitive names for most used types and members. Talk about naming choices during specification reviews.

Understand Your Customer
Realize that the majority of your customers are not like you. You should design the API for your customer, not for developers working in your close working group, who unlike majority of your customers are experts in the technology you are trying to expose.

3 Naming Guidelines

Casing and naming guidelines apply only to public and protected identifiers, and privately implemented interface members. Teams are free to choose their own guidelines for internal and private identifiers.

> ✔ **DO** use PascalCasing (capitalize the first letter of each word) for all identifiers except parameter names. For example, use TextColor rather than Textcolor or Text_color.

> ✔ **DO** use camelCasing (capitalize first letters of each word except for the first word) for all member parameter names.

✔ **DO** prefix descriptive type parameter names with 'T'.

```
public interface ISessionChannel<TSession>
  where TSession : ISession {
    TSession Session { get; }
}
```

✔ **CONSIDER** using T as the type parameter name for types with one single letter type parameter.

✔ **DO** use PascalCasing or camelCasing for any acronyms over two characters long. For example, use HtmlButton rather than HTMLButton, but System.IO instead of System.Io.

✘ **DO NOT** use acronyms that are not generally accepted in the field.

✔ **DO** use well-known acronyms only when absolutely necessary. For example, use UI for User Interface and Html for Hyper-Text Markup Language.

✘ **DO NOT** use of shortenings or contractions as parts of identifier names. For example, use GetWindow rather than GetWin.

✘ **DO NOT** use underscores, hyphens, or any other non-alphanumeric characters.

✘ **DO NOT** use the Hungarian notation.

✔ **DO** name types and properties with nouns or noun phrases.

✔ **DO** name methods and events with verbs or verb phrases. Always give events names that have a concept of before and after using the present particle and simple past tense. For example, an event that is raised before a Form closes should be named Closing. An event raised after a Form is closed should be named Closed.

✘ **DO NOT** use the "Before" or "After" prefixes to indicate pre and post events.

✔ **DO** use the following prefixes:

- "I" for interfaces.
- "T" for generic type parameters (except single letter parameters).

✔ **DO** Do use the following postfixes:

- "Exception" for types inheriting from System.Exception.
- "Collection" for types implementing IEnumerable.
- "Dictionary" for types implementing IDictionary or IDictionary<K,V>.
- "EventArgs" for types inheriting from System.EventArgs.
- "EventHandler" for types inheriting from System.Delegate.
- "Attribute" for types inheriting from System.Attribute.

✘ **DO NOT** use the postfixes listed above for any other types.

✘ **DO NOT** postfix type names with "Flags" or "Enum".

✔ **DO** use plural noun phrases for flag enums (enums with values that support bitwise operations) and singular noun phrases for non-flag enums.

✔ **DO** use the following template for naming namespaces:
<Company>.<Technology>[.<Feature>].
For example, Microsoft.Office.ClipGallery. Operating System components should use System namespaces instead for the <Company> namespaces.

✘ **DO NOT** use organizational hierarchies as the basis for namespace hierarchies. Namespaces should correspond to scenarios regardless of what teams contribute APIs for those scenarios.

4 General Design Guidelines

✔ **DO** use the most derived type for return values and the least derived type for input parameters. For example take IEnumerable as an input parameter but return Collection<string> as the return type.

✔ **DO** provide a clear API entry point for every scenario. Every feature area should have preferably one, but sometimes more, types that are the starting points for exploring given technology. We call such types Aggregate Components. Implementation of large majority of scenarios in given technology area should start with one of the Aggregate Components.

✔ **DO** write sample code for your top scenarios. The first type used in all these samples should be an Aggregate Component and the sample code should be straightforward. If the code gets longer than several lines, you need to redesign. Writing to an event log in Win32 API was around 100 lines of code. Writing to .NET Framework EventLog takes one line of code.

✔ **DO** model higher level concepts (physical objects) rather than system level tasks with Aggregate Components. For example File, Directory, Drive are easier to understand than Stream, Formatter, Comparer.

✘ **DO NOT** require users of your APIs to instantiate multiple objects in main scenarios. Simple tasks should be done with new statement.

✔ **DO** support so called "Create-Set-Call" programming style in all Aggregate Components. It should be possible to instantiate every component with the default constructor, set one or more properties, and call simple methods or respond to events.

```
var applicationLog = new EventLog();
applicationLog.Source = "MySource";
applicationLog.WriteEntry(exception.Message);
```

✘ **DO NOT** require extensive initialization before Aggregate Components can be used. If some initialization is necessary, the exception resulting from not having the component initialized should clearly explain what needs to be done.

✔ **DO** carefully choose names for your types, methods, and parameters. Think hard about the first name people will try typing in the code editor when they explore the feature area. Reserve and use this name for the Aggregate Component. A common mistake is to use the "best" name for a base type.

✔ **DO** run FxCop on your libraries.

✔ **DO** ensure your library is CLS compliant. Apply CLSCompliantAttribute to your assembly.

✔ **DO** prefer classes over interfaces.

✘ **DO NOT** seal types unless you have a strong reason to do it.

✘ **DO NOT** create mutable value types.

✘ **DO NOT** ship abstractions (interfaces or abstract classes) without providing at least one concrete type implementing each abstraction. This helps to validate the interface design.

✘ **DO NOT** ship interfaces without providing at least one API consuming the interface (a method taking the interface as a parameter). This helps to validate the interface design.

✘ **AVOID** public nested types.

✔ **DO** apply FlagsAttribute to flag enums.

✔ **DO** strongly prefer collections over arrays in public API.

✘ **DO NOT** use ArrayList, List<T>, Hashtable, or Dictionary<K,V> in public APIs. Use Collection<T>, ReadOnlyCollection<T>, KeyedCollection<K,V>, or CollectionBase subtypes instead. Note that the generic collections are only supported in the Framework version 2.0 and above.

✘ **DO NOT** use error codes to report failures. Use Exceptions instead.

✘ **DO NOT** throw Exception or SystemException.

✘ **AVOID** catching the Exception base type.

✔ **DO** prefer throwing existing common general purpose exceptions like ArgumentNullException, ArgumentOutOfRangeException, InvalidOperationException instead of defining custom exceptions.

✔ **DO** throw the most specific exception possible.

✔ **DO** ensure that exception messages are clear and actionable.

✔ **DO** use EventHandler¡T¿ for events, instead of manually defining event handler delegates.

✔ **DO** prefer event based APIs over delegate based APIs.

✔ **DO** prefer constructors over factory methods.

✘ **DO NOT** expose public fields. Use properties instead.

✔ **DO** prefer properties for concepts with logical backing store but use methods in the following cases:

- The operation is a conversion (such as Object.ToString())
- The operation is expensive (orders of magnitude slower than a field set would be)
- Obtaining a property value using the Get accessor has an observable side effect
- Calling the member twice in succession results in different results
- The member returns an array. Note: Members returning arrays should return copies of an internal master array, not a reference to the internal array.

✔ **DO** allow properties to be set in any order. Properties should be stateless with respect to other properties.

✘ **DO NOT** make members virtual unless you have a strong reason to do it.

✘ **AVOID** finalizers.

✔ **DO** implement IDisposable on all types acquiring native resources and those that provide finalizers.

✔ **DO** be consistent in the ordering and naming of method parameters. It is common to have a set of overloaded methods with an increasing number of parameters to allow the developer to specify a desired level of information. The Make sure all the related overloads have a consistent parameter order (same parameter shows in the same place in the signature) and naming pattern.

The only method in such a group that should be virtual is the one that has the most parameters and only when extensibility is needed.

```
public class Foo{
    readonly string defaultForA = "default value for a";
    readonly int defaultForB = 42;

    public void Bar(){
        Bar(defaultForA, defaultForB);
    }
    public void Bar(string a){
        Bar(a, defaultForB);
    }
    public void Bar(string a, int b){
        // core implementation here
    }
}
```

✘ **AVOID** out and ref parameters.

5 Resources

5.1 FxCop

FxCop is a code analysis tool that checks managed code assemblies for conformance to the Framework Design Guidelines.
http://code.msdn.microsoft.com/codeanalysis

5.2 Framework Design Studio

FDS is a tool useful for visualizing, comparing, and reviewing APIs.
http://code.msdn.microsoft.com/fds

5.3 Presentations

Overview of the Framework Design Guidelines:
http://blogs.msdn.com/kcwalina/archive/2007/03/29/1989896.aspx

TechEd 2007 Presentation about framework engineering:
http://blogs.msdn.com/kcwalina/archive/2008/01/08/FrameworkEngineering.aspx

5.4 Blogs

Design Guideline updates are posted to the following blog:
http://blogs.msdn.com/kcwalina