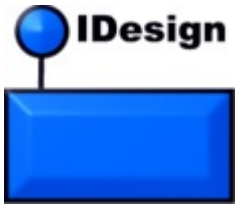

WCF Coding Standard

Guidelines and Best Practices
Version 4.0

Author: Juval Lowy
www.idesign.net

Table of Content

Preface

A comprehensive coding standard is essential for a successful product delivery. The standard helps in enforcing best practices and avoiding pitfalls, and makes knowledge dissemination across the team easier. Traditionally, coding standards are thick, laborious documents, spanning hundreds of pages and detailing the rationale behind every directive. While these are still better than no standard at all, such efforts are usually indigestible by the average developer. In contrast, the WCF coding standard presented here is very thin on the “why” and very detailed on the “what”. I believe that while fully understanding every insight that goes into a particular programming decision may require reading books and even years of experience, applying the standard should not. When absorbing a new developer into your team, you should be able to simply point him or her at the standard and say: "Read this first." Being able to comply with a good standard should come before fully understanding and appreciating it—that should come over time, with experience. The coding standard presented next captures dos and don'ts, pitfalls, guidelines, and recommendations. It uses the best practices and helper classes discussed in both the WCF Master Class and my book Programming WCF Services.

Juval Lowy

General Design Guidelines

1. All services must adhere to these principles:
   a) Services are secure.
   b) Service operations leave the system in a consistent state.
   c) Services are thread-safe and can be accessed by concurrent clients.
   d) Services are reliable.
   e) Services are robust.
2. Services can optionally adhere to these principles:
   a) Services are interoperable.
   b) Services are scale-invariant.
   c) Services are available.
   d) Services are responsive.
   e) Services are disciplined and do not block their clients for long.

Essentials

1. Place service code in a class library, not in any hosting EXE.
2. Do not provide parameterized constructors to a service class, unless it is a singleton that is hosted explicitly.
3. Enable reliability in the relevant bindings.
4. Provide a meaningful namespace for contracts. For outward-facing services, use your company's URL or equivalent URN with a year and month to support versioning. For example:

```
[ServiceContract(Namespace = "http://www.idesign.net/2009/06")]
interface IMyContract
{...}
```

   For intranet services, use any meaningful unique name, such as `MyApplication`. For example:

```
[ServiceContract(Namespace = "MyApplication")]
interface IMyContract
{...}
```

5. With intranet applications on prefer self-hosting to IIS hosting when the WAS is unavailable.
6. Do not mix and match named bindings with default bindings. Either have all your bindings be explicitly referenced, or use only default bindings.
7. Do not mix and match named behaviors with default behaviors. Either have all your behaviors be explicitly referenced, or use only default behaviors.
8. Always name all endpoints in the client config file.
9. Do not use SvcUtil or Visual Studio 2010 to generate a config file.

10. When using a tool such as Visual Studio 2010 to generate the proxy, do clean up the proxy.
11. Do not duplicate proxy code. If two or more clients use the same contract, factor the proxy to a separate class library.
12. Always close or dispose of the proxy.
13. When using discovery, prefer dynamic addresses.
14. When using discovery, do support the metadata exchange endpoint over TCP.
15. When using discovery, avoid cardinality of “some”.

Service Contracts

1. Always apply the `ServiceContract` attribute on an interface, not a class:

```
//Avoid:
[ServiceContract]
class MyService
{
   [OperationContract]
   public void MyMethod()
   {...}
}
//Correct:
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   void MyMethod();
}
class MyService : IMyContract
{
   public void MyMethod()
   {...}
}
```

2. Prefix the service contract name with an `I`:

```
[ServiceContract]
interface IMyContract
{...}
```

3. Avoid property-like operations:

```
//Avoid:
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   string GetName();

   [OperationContract]
   void SetName(string name);
}
```

4. Avoid contracts with one member.
5. Strive to have three to five members per service contract.

6. Do not have more than 20 members per service contract. Twelve is probably the practical limit.

Data Contracts

1. Avoid inferred data contracts (POCO). Always be explicit and apply the `DataContract` attribute.
2. Use the `DataMember` attribute only on properties or read-only public members.
3. Avoid explicit XML serialization on your own types.
4. Avoid message contracts.
5. When using the `Order` property, assign the same value to all members coming from the same level in the class hierarchy.
6. Support `IExtensibleDataObject` on your data contracts. Use explicit interface implementation.
7. Avoid setting `IgnoreExtensionDataObject` to `true` in the `ServiceBehavior` and `CallbackBehavior` attributes. Keep the default of `false`.
8. Do not mark delegates and events as data members.
9. Do not pass .NET-specific types, such as `Type`, as operation parameters.
10. Do not accept or return ADO.NET `DataSet`s and `DataTable`s (or their type-safe subclasses) from operations. Return a neutral representation such as an array.
11. Suppress the generation of a generic type parameter hash code and provide a legible type name instead.

Instance Management

1. Prefer the per-call instance mode when scalability is a concern.
2. If setting `SessionMode.NotAllowed` on the contract, always configure the service instancing mode as `InstanceContextMode.PerCall`.
3. Do not mix sessionful contracts and sessionless contracts in the same service.
4. Avoid a singleton unless you have a natural singleton.
5. Use ordered delivery with a sessionful service.
6. Avoid instance deactivation with a sessionful service.
7. Avoid demarcating operations.
8. With durable services, always designate a completing operation.

Operations and Calls

1. Do not treat one-way calls as asynchronous calls.
2. Do not treat one-way calls as concurrent calls.
3. Expect exceptions from a one-way operation.

4. Enable reliability even on one-way calls. Use of ordered delivery is optional for one-way calls.
5. Avoid one-way operations on a sessionful service. If used, make it the terminating operation:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
   [OperationContract]
   void MyMethod1();

   [OperationContract(IsOneWay = true,IsInitiating = false,IsTerminating =true)]
   void MyMethod2();
}
```

6. Name the callback contract on the service side after the service contract name, suffixed by `Callback`:

```
interface IMyContractCallback
{...}
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{...}
```

7. Strive to mark callback operations as one-way.
8. Use callback contracts for callbacks only.
9. Avoid mixing regular callbacks and events on the same callback contract.
10. Event operations should be well designed:
    a) `void` return type
    b) No out-parameters
    c) Marked as one-way operations
11. Avoid using raw callback contracts for event management, and prefer using the publish-subscribe framework.
12. Always provide explicit methods for callback setup and teardown:

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
   [OperationContract]
   void DoSomething();

   [OperationContract]
   void Connect();

   [OperationContract]
   void Disconnect();
}
interface IMyContractCallback
{...}
```

13. Use the type-safe `DuplexClientBase<T,C>` instead of `DuplexClientBase<T>`.
14. Use the type-safe `DuplexChannelFactory<T,C>` instead of `DuplexChannelFactory<T>`.

15. When debugging or in intranet deployment of callbacks over the `WSDualHttpBinding`, use the `CallbackBaseAddressBehavior` attribute with `CallbackPort` set to `0`:

```
[CallbackBaseAddressBehavior(CallbackPort = 0)]
class MyClient : IMyContractCallback
{...}
```

Faults

1. Never use a proxy instance after an exception, even if you catch that exception.
2. Avoid fault contracts and allow WCF to mask the error.
3. Do not reuse the callback channel after an exception even if you catch that exception, as the channel may be faulted.
4. Use the `FaultContract` attribute with exception classes, as opposed to mere serializable types:

```
//Avoid:
[OperationContract]
[FaultContract(typeof(double))]
double Divide(double number1,double number2);

//Correct:
[OperationContract]
[FaultContract(typeof(DivideByZeroException))]
double Divide(double number1,double number2);
```

5. Avoid lengthy processing such as logging in `IErrorHandler.ProvideFault()`.
6. With both service classes and callback classes, set `IncludeExceptionDetailInFaults` to `true` in debug sessions, either in the config file or programmatically:

```
public class DebugHelper
{
   public const bool IncludeExceptionDetailInFaults =
#if DEBUG
      true;
#else
      false;
#endif
}
[ServiceBehavior(IncludeExceptionDetailInFaults =
                 DebugHelper.IncludeExceptionDetailInFaults)]
class MyService : IMyContract
{...}
```

7. In release builds, do not return unknown exceptions as faults except in diagnostic scenarios.
8. Consider using the `ErrorHandlerBehavior` attribute on the service, both for promoting exceptions to fault contracts and for automatic error logging:

```
[ErrorHandlerBehavior]
class MyService : IMyContract
{...}
```

9. Consider using the `CallbackErrorHandlerBehaviorAttribute` on the callback client, both for promoting exceptions to fault contracts and for automatic error logging:

```
[CallbackErrorHandlerBehavior(typeof(MyClient))]
class MyClient : IMyContractCallback
{
   public void OnCallabck()
   {...}
}
```

Transactions

1. Never manage transactions directly.
2. Apply the `TransactionFlow` attribute on the contract, not the service class.
3. Do not perform transactional work in the service constructor.
4. Using this book's terminology, configure services for either Client or Client/Service transactions. Avoid None or Service transactions.
5. Using this book's terminology, configure callbacks for either Service or Service/Callback transactions. Avoid None or Callback transactions.
6. When using the Client/Service or Service/Callback mode, constrain the binding to flow transactions using the `BindingRequirement` attribute.
7. On the client, always catch all exceptions thrown by a service configured for None or Service transactions.
8. Enable reliability and ordered delivery even when using transactions.
9. In a service operation, never catch an exception and manually abort the transaction:

```
//Avoid:
[OperationBehavior(TransactionScopeRequired = true)]
public void MyMethod()
{
   try
   {
      ...
   }
   catch
   {
      Transaction.Current.Rollback();
   }
}
```

10. If you catch an exception in a transactional operation, always rethrow it or another exception.
11. Keep transactions short.
12. Always use the default isolation level of `IsolationLevel.Serializable`.
13. Do not call one-way operations from within a transaction.
14. Do not call nontransactional services from within a transaction.
15. Do not access nontransactional resources (such as the filesystem) from within a transaction.

16. With a sessionful service, avoid equating the session boundary with the transaction boundary by relying on auto-complete on session close.
17. Strive to use the `TransactionalBehavior` attribute to manage transactions on sessionful services:

```
[Serializable]
[TransactionalBehavior]
class MyService : IMyContract
{
   public void MyMethod()
   {...}
}
```

18. When using a sessionful or transactional singleton, use volatile resource managers to manage state and avoid explicitly state-aware programming or relying on WCF's instance deactivation on completion.
19. With transactional durable services, always propagate the transaction to the store by setting `SaveStateInOperationTransaction` to `true`.

Concurrency Management

1. Always provide thread-safe access to:
   a) Service in-memory state with sessionful or singleton services
   b) Client in-memory state during callbacks
   c) Shared resources
   d) Static variables
2. Prefer `ConcurrencyMode.Single` (the default). It enables transactional access and provides thread safety without any effort.
3. Keep operations on single-mode sessionful and singleton services short in order to avoid blocking other clients for long.
4. When using `ConcurrencyMode.Multiple`, you must use transaction auto-completion.
5. Consider using `ConcurrencyMode.Multiple` on per-call services to allow concurrent calls.
6. Transactional singleton service with `ConcurrencyMode.Multiple` must have `ReleaseServiceInstanceOnTransactionComplete` set to `false`:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
                 ConcurrencyMode = ConcurrencyMode.Multiple,
                 ReleaseServiceInstanceOnTransactionComplete = false)]
class MySingleton : IMyContract
{...}
```

7. Never self-host on a UI thread, and have the UI application call the service.
8. Never allow callbacks to the UI application that called the service unless the callback posts the call using `SynchronizationContext.Post()`.
9. When supplying the proxy with both synchronous and asynchronous methods, apply the `FaultContract` attribute only to synchronous methods.

10. Keep asynchronous operations short. Do not equate asynchronous calls with lengthy operations.
11. Do not mix transactions with asynchronous calls.

Queued Services

1. On the client, always verify that the queue (and a dead-letter queue, when applicable) is available before calling the queued service. Use `QueuedServiceHelper.VerifyQueues()` for this purpose.
2. Always verify that the queue is available when hosting a queued service (this is done automatically by `ServiceHost<T>`).
3. Except in isolated scenarios, avoid designing the same service to work both queued and non-queued.
4. The service should participate in the playback transaction.
5. When participating in the playback transaction, avoid lengthy processing in the queued service.
6. Avoid sessionful queued services.
7. When using a singleton queued service, use a volatile resource manager to manage the singleton state.
8. When using a per-call queued service, explicitly configure the contract and the service to be per-call and sessionless:

```
[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{...}
```

9. Always explicitly set contracts on a queued singleton to disallow sessions:

```
[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class MyService : IMyContract
{...}
```

10. The client should call a queued service inside a transaction.
11. On the client side, do not store a queued service proxy in a member variable.
12. Avoid relatively short values of `TimeToLive`, as they negate the justification for a queued service.
13. Avoid nontransactional queues.
14. When using a response queue, have the service participate in the playback transaction and queue the response in that transaction.
15. Have the response service participate in the response playback transaction.
16. Avoid lengthy processing in a queued response operation.

17. With MSMQ 3.0, prefer a response service to a poison queue service dealing with failures of the service itself.
18. With MSMQ 4.0, use `ReceiveErrorHandling.Reject` for poison messages unless you have advanced processing with `ReceiveErrorHandling.Move`. Avoid `ReceiveErrorHandling.Fault` and `ReceiveErrorHandling.Drop`.
19. With MSMQ 4.0, consider the use of a response service to handle service playback failures.
20. Unless dealing with a sessionful contract and service, never assume the order of queued calls.
21. Avoid multiple service endpoints sharing a queue.
22. Avoid receive context.

Security

1. Always protect the message and provide for message confidentiality and integrity.
2. In an intranet, you can use Transport security as long as the protection level is set to `EncryptAndSign`.
3. In an intranet, avoid impersonation. Set the impersonation level to `TokenImpersonationLevel.Identification`.
4. When using impersonation, have the client use `TokenImpersonationLevel.Impersonation`.
5. Use the declarative security framework and avoid manual configuration.
6. Never apply the `PrincipalPermission` attribute directly on the service class:

```
//Will always fail:
[PrincipalPermission(SecurityAction.Demand,Role = "...")]
public class MyService : IMyContract
{...}
```

7. Avoid sensitive work that requires authorization at the service constructor.
8. Avoid demanding a particular user, with or without demanding a role:

```
//Avoid:
[PrincipalPermission(SecurityAction.Demand,Name = "John")]
public void MyMethod()
{...}
```

9. Do not rely on role-based security in the client's callback operations.
10. With Internet clients, always use Message security.
11. Allow clients to negotiate the service certificate (the default).
12. Use the ASP.NET providers for custom credentials.
13. When developing a custom credentials store, develop it as a custom ASP.NET provider.
14. Validate certificates using peer trust.

The Service Bus

1. Prefer the TCP relay binding.
2. Make your services be discoverable.
3. Do use discrete events.
4. Do not treat buffers as queues.
5. With buffers, avoid raw WCF messages and use the strongly typed, structured calls technique of `BufferedServiceBusHost<T>` and `BufferedServiceBusClient<T>`.
6. Use message security.
7. Do not use service bus authentication for user authentication.
8. Strive for anonymous calls and let the service bus authenticate the calling application.

Resources

1 Programming WCF Services 3rd Edition

By Juval Lowy, O'Reilly 2010

2 The WCF Master Class

The world's best, most intense WCF training starts by explaining the motivation for service-orientation and then continues to discuss in depth how to develop service-oriented applications using WCF. You will see how to take advantage of built-in features such as service hosting, instance management, asynchronous calls, synchronization, reliability, transaction management, disconnected queued calls and security. While the class shows how to use these features, it sets the focus on the 'why' and the rationale behind particular design decisions, often shedding light on poorly-documented and understood aspects. You will learn not only WCF programming but also relevant design guidelines, best practices, and pitfalls. The material presented includes IDesign's original techniques and utilities and goes well beyond anything you can find in conventional sources. Don't miss out on this unique opportunity to learn WCF from the IDesign architects who have been part of the strategic design effort for WCF from the beginning, and who offer a profound insight on the technology and its applications. Any .NET developer or architect would benefit greatly from the class.

More at *www.idesign.net*

3 The Architect's Master Class

The Architect's Master Class is a 5 days training, and is the ultimate resource for the professional architect. The class has three parts, on process, technology & SOA, and the IDesign method for analysis and design. The class shows the architect how to take an active leadership role on all three aspects, as a continuum, since when executing a design, one cannot separate process from design from technology – all three have to work in concert. You will see relevant design guidelines, best practices, and pitfalls, and the crucial process required of today's modern architects. Don't miss on this unique opportunity to learn and improve your architecture skills with IDesign, and share their passion for architecture and software engineering.

More at *www.idesign.net*

4 The IDesign Serviceware Downloads

The IDesign serviceware downloads are a set of original techniques, tools, utilities and even breakthroughs developed by the IDesign architects. The utilities are largely productivity-enhancing tools, or they compensate for some oversight in the design of .NET or WCF. The demos are also used during our *Master Classes* to demystify technical points, as lab exercises or to answer questions. The classes' attendees find the demos useful not only in class but after it. The demos serve as a starting point for new projects and as a rich reference and samples source.

More at *www.idesign.net*