SharpMedia Coding Guidelines

© Žiga Osolin – SharpMedia project, document version: DRAFT

Contents

[Introduction 3](#_Toc166811919)

[Documentation 3](#_Toc166811920)

[Namespaces and Assemblies 3](#_Toc166811921)

[Classes 3](#_Toc166811922)

[Disposable 3](#_Toc166811923)

[Serialization 3](#_Toc166811924)

[MarshalByRefObject 3](#_Toc166811925)

[Interfaces 3](#_Toc166811926)

[Properties 3](#_Toc166811927)

[Events 3](#_Toc166811928)

[Methods 3](#_Toc166811929)

# Introduction

SharpMedia coding guidelines try to give each SharpMedia developer some sort of information about coding style used for this project. It is important that all developers write code in similar way, otherwise confusion can occur.

# Documentation

All public members of classes must be fully documented. This includes full method specifications, meaning of parameters and special remarks to SharpMedia developers or API usages. The documentation guideline is the same as proposed by Microsoft.

Note that internal, protected or private classes do not have to be commented. As long as API users won’t see them, it is okay to leave then uncommented. However, it is suggested that complex procedures and meaning of certain non-trivial entities are still documented.

# Namespaces and Assemblies

New namespace should be creating in following scenarios:

* Representation of logical unit that includes collection of publically visible classes (not just one class);
* To provide private implementations, if too many are needed;
* To isolate driver and API part;
* When too many classes are placed into one namespace.

On the other hand, new assembly should be created for:

* Each application in SharpMedia (so it can be run as SharpMedia process);
* For logical library unit with general functionality.

We prefer bigger assemblies because loading many smaller assemblies is slower than loading one big assembly. When thinking about separating one assembly into two, always check the usage cases or how often the other assembly will be needed if the first is needed.

# Classes

A class encapsulates some functionality. Static classes (containing only static methods) should include methods for working with something fairly specific. It should be easy for the user to guess which class to use to do something.

As for non-static classes, object orientated programming should be fully employed. A class must provide all functionality needed when it is used. Helper properties, events, static methods are desirable. Classes must be user friendly and allow fast development. A usability perspective is important; that means that everything must be possible, invalid states are hard/impossible to obtain and programming is in general fast (commonly used code can be provided as helpers).

Classes that implement interfaces do not have to document interface methods if behavior is the same as stated by interface method’s description. Special remarks must however be made if special behaviors can occur.

## Disposable

All classes that have unmanaged resources or track their lifetime policies must implement disposable interface. It is particularly useful because of *using* statement:

using (ITypedStream s = manager.Find("NodeName").OpenDefaultStream(OpenMode.Read))

{

// Do some work ...

}

## Serialization

Serialization is also one of the key benefits of CLR. All classes that will be once written to disk or need other kind of persistency or are remote objects without proxies, must have Serializable attribute. We also suggest implementing custom serialization methods if class is too complex to be written by default serializator. Don’t forget using version specific serialization attributes to not break back compatibility.

## MarshalByRefObject

Classes that can be called using proxies from other AppDomains should extend from this class.

# Interfaces

In general, common functionality that is shared among many classes should be placed in interfaces. Interfaces are good because they do not limit the class that implements it to not implement other interfaces (in general, a class can inherit from only *one* other class but from *multiple* interfaces).

Common interfaces, defined by .NET should also be employed. This is especially true for structures that need to implement equality tests, cloning, enumerating … Inheriting from .NET classes allows easy processing with functionality already provided by .NET.

## Common Interface

Those interfaces are placed into SharpMedia namespace and should be reused as much as possible. We do not document them here, only some exceptional cases are documented.

### ICloneable<T>

A generic version of .NET interface named **ICloneable**. Some classes may depend on this interface. The interface inherits **ICloneable** for .NET compatibility. The non-generic cloning should be implemented explicitly.

# Properties

Querying properties with Boolean should begin with Is, e.g. IsDisposed, IsNice, IsBig. Other properties’ names must also represent their real meaning. If one variable is used to hold much information, it is advised to provide properties for each of this information (plus the property for variable itself).

# Events

Events are also a powerful way to customize classes and provide side processing. At the same time, events must not be misused. Events should be a rare thing to happen (e.g. not triggered 1000 times a second). Be careful to use correct event firing!

Whenever possible, use events with **Action<T>** generic delegate or some other existent delegate. Do not invent new delegate types. Don’t use Microsoft guidelines for delegates (sender and event descriptor) but place description not as a separate class but as parameters. Only if there are too many, pack them into some class. This way, we avoid nasty casts and new classes, only new delegates must be created (one liner in C#).

# Methods

Methods should be correctly named. Whenever possible, properties or indexers should be used instead of methods (for debugging and code style purposes). Method overloading is a good technique to allow fast programming if some operands are commonly not used or defaults are usually selected.

A method should be virtual if at some point, customization is desirable.