BrianHetrick.com
Application Note AN-9

The Singleton Pattern in .NET

Contents

Goal

This Application Note describes the singleton design pattern in .NET. It discusses the design pattern itself and the properties of the pattern, including the circumstances when it is useful. In investigates three different implementation strategies for the design pattern, and recommends one as particularly suited for most circumstances. An example of an application-wide facility as is typically addressed by the singleton pattern is examined in detail, showing how the recommended implementation strategy can be used to adjust the characteristics of the implementation in use.

The .ZIP file accompanying this Application Note includes both the example facility and the test driver used to demonstrate the implementation characteristics.

Introduction

A rule of thumb in software design and implementation is that there are three numbers of interest: zero, one, and infinity (by which is meant a finite but unbounded number). The zero case is typically trivial to implement: simply do not reference something at all. The unbounded case is also typically also trivial: provide a reference to the particular thing in which one is interested at the moment. The one case, however, is special. The one instance of a class in this situation is termed a *singleton*. The singleton design pattern is as acknowledgment that sometimes there needs to be exactly one of something, and its scope is the entire application. The example typically used is that of a logging facility: one typically wants all logging messages, application wide, to appear in a single place.

There is no doubt that implicitly sharing a single instance of a class among all components of an application is frequently useful. Classes that perform a critical glue function holding the application design together are typically of this nature. This Application Note takes as at exemplar of this type of function the event publication mechanism. Any application using events for inter-component notifications must have an event publication mechanism. As this mechanism is purely functional, having a single instance of the mechanism shared across the application is sensible.

The singleton design pattern has its detractors, and there is some truth to the assertion that the singleton pattern is in fact an anti-pattern. The typical singleton implementation does not separate the interface of the singleton object from the implementation; the typical singleton implementation does not provide for replaceable implementations; the typical singleton implemen-

tation provides implicit application global state which hides dependencies on it. Nor is the example of a logging system truly an argument for a singleton. A single web server's logs are typically and usefully maintained separately for each web site it serves up, for example; and an audit trail is essentially a log with requirements conflicting with those of a debugging log. However, as this Application Note will attempt to show, these criticisms targets not the pattern itself but its typical implementation technique.

This Application Note shows how a jacket class forwarding to a single implementing object can be used to address these criticisms of the singleton pattern. This technique provides a default instance of the implementing object, but others can easily be used when desired; this technique provides an instance of a default class, but others can easily be used (even application-wide) when desired; and this technique strongly separates the interface from the implementation. This technique is most natural in C#, due to the availability of static classes, but also applies to other languages with static class members, such as C++ and Java.

The Pattern and Its Properties

Distinguishing Characteristics

The distinguishing characteristic of the singleton pattern is that a class have a single instance, and this instance be used whenever the application needs the functionality exposed by that class.

This is opposed to the more typical case where an instance of a class is created when the application needs the functionality exposed by that class. The class instance is created, the function performed, and the instance is discarded. Absent other arrangements, any state held in the instance is lost.

It is entirely possible for the state held by an instance to be static, that is, class-wide rather than instance-wide. In this arrangement, although class instances may be created and destroyed freely, the class is arguably a singleton: there is a single state accessed through the class, regardless of appearances. Using class state rather than instance state is the source of the unnecessary singleton flaw, where what should be per-instance state is incorrectly shared among instances. This typically has undesirable effects.

A requirement of the singleton pattern is that the singular instance can be accessed. Since there is only one instance, by definition, passing around a reference to the instance is unusual. Typically a static member of a class is used either to hold the reference or to return the reference.

Note that dependency injection frameworks can also typically provide singletons. The singleton may be directly requested from the framework when desired, or may be injected into a class through a constructor argument or through property injection. That a singleton is to be used to satisfy requests for a type, rather than new instances, is an element of the lifetime management policy attached to the type. This implementation, while convenient, disguises the singleton nature of the singleton class; this may be viewed as an advantage or a disadvantage.

Benefits

The benefit of the singleton pattern is that it essentially provides a language extension rather than an application function. The singleton instance can typically be accessed without an explicit reference to it being part of an object's state. Instead, a client object can reference the static member containing the reference to the singleton object. The member, being static, does not require a reference to the containing instance: there is no containing instance. Instead the singleton instance can be obtained whenever desired.

Costs

The cost of the singleton pattern is that it allows certain objects to be used without an explicit reference to them. This makes it difficult to track the use of the object instance—references to it do not appear in parameter lists or as property values. It also prohibits fine-grained dependency injection, where the application may need to use one implementation of an interface one place and another implementation another place.

Implementation Alternatives

This section describes typical implementation approaches to the singleton design pattern, and compares them to one another.

Private Constructor and Static Instance Property

The simplest implementation of the singleton design pattern makes the class constructor `private`, and provides a class member (rather than instance member) `getInstance()` method or `Instance` property. The method or property returns a reference to the single instance; the use of a private constructor ensures no out-

```
public class ImplementationMethod1
{
    private static ImplementationMethod1 _instance;

    public static ImplementationMethod1 Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new ImplementationMethod1 ();
            }
            return _instance;
        }
    }

    private ImplementationMethod1 ()
    { }

    public void ShowInstance ()
    {
        Console.WriteLine
            ("This is the singleton instance of ImplementationMethod1.");
    }
}
```

Display 1. Singleton lazy initialization (single threaded).

side class can create an instance of the class intended to be a singleton.

The actual instance of the class for which a singleton is provided can be created when the class is loaded or activated, through a static initializer; or the instance can be created when it is first requested. The latter technique, called lazy initialization, is typically used when creation of the instance is an expensive operation and moving that operation out of program start-up is desired. Lazy initialization of a singleton reference can be implemented in a single threaded environment with code as shown in Display 1.

The `Instance` property code is straightforward: if an instance does not exist, create one; and return the instance. In a multi-threaded environment, however, this code creates a race condition: the test as to whether an instance exists is not atomic with the creation of a new instance and the storage of a reference to that new instance. Multiple threads executing this code simultaneously can each observe the lack of an instance, each independently create a new instance, and each independently set the instance reference. This circumstance then results in multiple instances of the "singleton." This can be prevented by forcing sequential operation on the `Instance` property. The cleanest way to force sequential operation is to lock on a lock object, and perform the test and possible set within the scope of that lock. This straightforward approach has the disadvantage that the lock is taken even when there is already an instance that can be used directly. Typically this is not worth worrying about—taking an uncontested lock in .NET is a tens-of-nanoseconds operation—but in a heavy load environment a cumulative effect may be observable. This consideration gives rise to the *double test* idiom, shown in Display 2 (next page). Although this idiom first arose in Java, details of the Java memory model imply the idiom does not actually work in Java. It does, however, work in .NET.

Note how the `Instance` method first tests the singleton instance reference; if there is one (as there will be after the first calls), it returns it directly. Only when there is no instance does the `Instance` method lock the lock object, then retest: some other thread may have created an instance between the testing and the

```
public class ImplementationMethod2
{
    private static ImplementationMethod2 _instance;
    private static object _lockObject = new object ();

    public static ImplementationMethod2 Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lockObject)
                {
                    if (_instance == null)
                    {
                        _instance = new ImplementationMethod2 ();
                    }
                }
            }
            return _instance;
        }
    }

    private ImplementationMethod2 ()
    { }

    public void ShowInstance ()
    {
        Console.WriteLine
            ("This is the singleton instance of ImplementationMethod2.");
    }
}
```

Display 2. Singleton lazy initialization (thread safe).

granting of the lock. Only if the retest still shows the lack of an instance method is an instance created and the reference set.

This idiom is sometimes attributed, even by persons who should know better, to "a race condition in locking." That attribution is incorrect. If locking has a race condition, the operating system is severely broken. The race condition is between the test for the need for an instance and the subsequent creation of an instance. These two operations must be atomic: the reference tested must be the reference set. The atomicity of the compound operation (test and if necessary create and set) is provided by the lock. The outer test is simply an optimization to avoid the overhead of the lock when it can be shown to be unnecessary.

Static Class

The C# type system provides another opportunity for implementing singletons, the static class. A static class has no instances, and all members of the static class must themselves be static. They are therefore members of the class itself, rather than members of the instances of the class (of which, with a static class, there can be none).

Static classes can expose properties, methods, variables, events, and so forth, just as a non-static class can. These class members are referenced with *classname.membername* rather than *instancereference.membername*. The effect is to introduce a namespace containing what are effectively global procedures and global variables. Another way of thinking

```
public static class ImplementationMethod3
{
    public static void ShowInstance ()
    {
        Console.WriteLine ("This is the static class ImplementationMethod3.");
    }
}
```

Display 3. Singleton as a static class.

of this is that a static class implicitly has exactly one instance, known by the class name. This is in some respects similar to pre-.NET Visual Basic, where forms had a "default instance" that could be referred to with the form type name.

There are two major limitations of static classes: they cannot participate in a class hierarchy, and there can be no references to them. A static class cannot inherit from another class or from an interface, and a class cannot inherit from a static class. As there are no instances of a static class, there can be no references to instances of the static class; this implies that the static class cannot, for example, be passed as a parameter. (The `Type` instance corresponding to the class can be passed as a parameter; a delegate to a static class's method can be passed as a parameter; but if some method needs a reference to a provider of a function, that provider cannot be a static class.)

A static class used as a singleton can be extraordinarily simple. This is shown by the code in Display 3, which is functionally equivalent to the previous examples.

When using a static class, there is no need to mess about with creating and using an `Instance` property and protecting against multithreading race conditions: just refer to *`classname.membername`* and go. This is straight procedural code, almost like good old C: just pretend that dot is an underscore. And yet there are reasons the world moved from the procedural model to the object-oriented model, so perhaps this is not quite a good thing.

Static Jacket

The convenience of the static `Instance` method approach and the static class approach are notable. However, the drawbacks are notable as well. In both approaches discussed so far, the use of the class is tied to the identity of the class, and the class itself must be written as a singleton. The implementation of the functionality provided by the singleton cannot be replaced without modifying all use sites to refer to the new implementation class. A class cannot be used as a singleton without being designed as a singleton. Finally, a class designed as a singleton cannot have multiple instances created even when such use makes sense.

These deficits can be addressed by combining the two approaches discussed so far. The single instance of the singleton class need not be provided by the class itself: the single instance can be provided by another class, which can be static and hence available without a reference. This is the *static jacket* approach to implementing a singleton.

In this approach, a (static) class—the jacket class—provides a single class member static reference to a single instance of an implementation of an interface. The jacket class can provide a default implementation which is an instance of an implementing class. The jacket class provides an instance property which returns a reference to the single instance of the implementing class. This reference can be used as a parameter or assignment source or anywhere else an object instance in required. The jacket class can also provide convenience methods that delegate to the methods of the implementing class. Finally, the jacket class can expose the instance property as read-write, rather than read-only. This then permits the application to replace the instance of the implementing class with an instance of a different implementing class for performance or other reasons.

As there is nothing necessarily special about the implementing class (other than its use by the jacket class), the application can freely create and use other instances where needed. These other instances could be referred to by their own jacket classes. Thus the conflicting requirements of an audit trail and a debugging log could be addressed by a single logging class with differently configured instances, and perhaps different convenience methods in the respective jacket classes.

```
public interface IImplementationMethod4
{
    void ShowInstance ();
}

public static class ImplementationMethod4
{
    private static object _lockObject = new object ();
    private static IImplementationMethod4 _instance;

    public static IImplementationMethod4 Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lockObject)
                {
                    if (_instance == null)
                    {
                        _instance = new ImplementationMethod4Implementation1 ();
                    }
                }
            }
            return _instance;
        }

        set
        {
            IDisposable disposable;
            lock (_lockObject)
            {
                disposable = _instance as IDisposable;
                _instance = value;
            }
            if (disposable != null)
            {
                disposable.Dispose ();
            }
        }
    }

    public static void ShowInstance ()
    {
        Instance.ShowInstance ();
    }
}
```

Display 4. Singleton as a static jacket class (part 1).

```
public class ImplementationMethod4Implementation1: IImplementationMethod4
{
    public void ShowInstance ()
    {
        Console.WriteLine
            ("This is an instance of ImplementationMethod4Implementation1.");
    }
}

public class ImplementationMethod4Implementation2: IImplementationMethod4
{
    public void ShowInstance ()
    {
        Console.WriteLine
            ("This is an instance of ImplementationMethod4Implementation2.");
    }
}
```

Display 4. Singleton as a static jacket class (part 2).

This arrangement results in a multi-part implementation, as shown in Display 4. This multi-part implementation is functionally equivalent to the previous examples.

The jacket class approach to the singleton pattern is composed of the following parts:

- An interface to the singleton object. The interface (`IImplementationMethod4` here) separates the use of the singleton class from the implementation of the singleton class. Knowledge of the identity of the class implementing the singleton is both unnecessary and largely useless.
- A class holding a static reference to the singleton object. The class (`ImplementationMethod4` here) creates an instance of a default implementation class on first use and provides methods which invoke that implementation class instance. The `Instance` property is read-write; thus the application can use the default implementation class by doing nothing, or can replace the default implementation instance with an instance of another class that implements the `IImplementationMethod4` interface. The convenience method `ShowInstance` of the static class delegates to the `ShowInstance` method on whatever the implementing class is at the current time. Thus if the function is required, the convenience method of the static class can be used; if a reference to an implementing instance is required, the `Instance` property of the static class can be used.
- Implementation classes. The implementation class or classes (`ImplementationMethod4Implementation1` and `ImplementationMethod4Implementation2` here) are typical classes which implement the desired interface (`IImplementationMethod4` here). Instances of these classes can, if required, be freely created and used by the application; or the application can use the singleton instance conveniently offered by the static class.

This approach is evidently structurally more complicated than the previous approaches: a minimum of three classes (the interface, the static class, and the interface implementation) are required rather than a single class. In return for this added complexity, however, there is a substantial increase in functionality and usability.

Extended Example

As an extended example of the singleton pattern in actual use, this Application Note takes the example of an event publication facility.

Event Publication

The eventing pattern is .NET's original implementation of the Observer design pattern. An event in .NET parlance is a delegate of a particular form. A delegate in .NET is a list of object and class methods. When de-

clared with the `event` keyword, the delegate's addition and removal accessors are exposed as the scope keyword requires, but its other accessors remain private. This permits other objects to add to and remove from the list of methods, but not to otherwise manipulate the delegate. The methods on the list can be invoked by the object owning the delegate; in the eventing pattern, this is done when a particular situation arises. This serves to notify the objects on the list—the *subscribers* to the event—that the situation has arisen.

Event subscribers take two parameters: a reference to the object publishing the event, and a reference to an object describing the event. The first parameter serves to disambiguate multiple sources, the second multiple events; this lets subscriber methods handle multiple events if desired. The object describing the event is traditionally a descendant of the framework-provided `EventArgs` class, although this tradition seems to be fading.

Although the .NET languages have syntax for invoking all the targets of a delegate, this unadorned invocation has semantics not particularly well-suited to the eventing pattern. In particular, exceptions thrown by subscribers invoked in this way both terminate the (iterative) delegate list invocation and propagate back to the event publisher. For this reason, the idiom in .NET for event publication is to expand the delegate list into its elements, iterate through the elements, and invoke each separately while in a `try-catch` statement. This ensures all subscribers to the event are notified, and prevents exceptions thrown by subscribers from affecting the event publisher.

The code to expand the delegate and invoke its individual members is not particularly difficult, but it is too lengthy to be a convenient in-line idiom. Therefore this code is traditionally extracted to a method which performs the mechanics of event publication. This code can be shared among all event publishers—making it a reasonable application for the singleton design pattern.

In this section, then, this Application Note investigates two mechanisms for event publication. Both mechanisms perform event publication, but their semantics differ slightly and their resource requirements can differ significantly depending on application structure and behavior. An application might choose to depend on the details of semantics, or may find the resource requirements of one to be exorbitant; and therefore require one implementation or the other.

Two Implementations

The first implementation of event publication is straightforward, and is presented in the `SynchronousEventPublisher` class in the attached .ZIP file. The `PublishEvent` method accepts an `EventDescription` object; the `EventDescription` object contains a reference to the publishing object, a reference to an `EventArgs` object, and a reference to the `Delegate` containing the event invocation list. The method expands the delegate into single-target delegates using the `GetInvocationList()` method of the delegate, and then within a `try-catch` statement invokes each of the target methods. Should the target throw an exception, the exception is caught, translated into an `ExceptionDuringEventNotification` event, and published. The event arguments to the `ExceptionDuringEventNotification` event give information sufficient to identify the offending event handler. The use of an event permits the application to choose its response to the exception during event notification: to log it and continue, to terminate the application, or to ignore it, for example.

A subtlety of this implementation is that event subscribers are notified before the `PublishEvent` method returns to its caller. This has the effect of publishing events "depth-first:" should an event handler for one event itself publish a second event, the second event is delivered immediately—while the first event is still in mid-publication. This has three implications:

- One implication is that events can be delivered out of order. That is, an object might subscribe to two events, where one event gives rise to a second event, and receive the second event before it receives the first event. This can violate an assumption that events are received in the order in which they are published. Maintaining the temporal order of events is *not* a property of synchronous event publication, and assuming this property can cause an application to malfunction.
- A second implication is that events can serve as a bidirectional communication mechanism. That is, the event arguments have been seen (and possibly modified) by all event subscribers when control returns to the event publisher. This property is exploited by elements of the .NET framework such as the `Form.FormClosing` event in the Windows Forms.framework: should a subscriber to the `FormClosing` event set the `Cancel` property of

the event arguments to `true`, the form close process is aborted.

- A third implication is that stack storage is put under stress. Arbitrarily many events may be in publication at once—any event handler may itself publish events. For each event in publication, stack frames for both the publication mechanism and the event handler must be created. This can be a significant factor in resource constrained systems, or in applications using an event-driven mechanism.

Of these, the first may be regarded as a curiosity, the second as an advantage, and the third as a disadvantage. It may be that the disadvantage of the third implication—that of putting stress on stack storage—outweighs the advantage of the second—that of bidirectional communication. Is such a case, an alternative implementation of event publication may be desired. For such cases, a second implementation of event publication is provided.

The second implementation of event publication is similarly straightforward, and is presented in the `AsynchronousEventPublisher` class in the attached .ZIP file. Again a `PublishEvent` method accepts an `EventDescription` object as before. However, instead of publishing the event immediately, this implementation instead enqueues the event description on a per-thread queue of event descriptions of pending event publications. If event publication is already occurring on the current thread, the implementation immediately returns to its caller. If event publication is not already occurring on the current thread, events on the queue are published until there are no more pending event publications.

The interactions between threads and events is discussed in detail in The Event-Based Asynchronous Operation Pattern in .NET. For the present discussion, simply noting that events are associated with a particular thread is sufficient.

This arrangement depends on a the reentrant behavior of the `PublishEvent` method. The outermost invocation of the `PublishEvent` method is the invocation that publishes events. Other invocations of the `PublishEvent` method—which occur only when code executed as a result of the event publication itself publishes an event—merely queue event descriptions. These subsequent events will be published in turn by the outermost invocation. Because the non-outermost invocations of `PublishEvent` do not transfer control out of the method, there are at most two `PublishEvent` stack frames in use: one for invocation publishing events, and one for the invocation queueing an event.

Now, this second, asynchronous, implementation also has implications. To some extent, these implications are the opposites of the implications of the first, synchronous, implementation. These implications are:

- One implication is that events are delivered in the order in which they were published. An object subscribing to events from multiple event sources will never observe a response event delivered before a stimulus event.
- A second implication is that events become unidirectional communications pathways. When control returns to the event publisher, the event might not yet have been published. Consequently, the subscribers to the event have not yet necessarily seen the event arguments, and so might not have been able to modify them. This implementation removes the "back channel" from the event subscriber to the event publisher provided by the synchronous event publication approach.
- A third implication is that heap storage, but not stack storage, is put under stress. Arbitrarily many events may queued for publication at once. For each event queued for publication, the event description must be maintained in heap storage. This can be a significant factor in resource constrained systems, or in applications using an event-driven mechanism.

Moving storage demands from the stack to the heap may be an advantage or a disadvantage, depending on the application structure and environment. The loss of the communication channel from subscriber to publisher is a decided disadvantage of asynchronous event publication; and the delivery of events in the same order as that in which they were published may be an advantage.

The asynchronous event publisher in the attached .ZIP file happens to delegate actual event publication to an instance of the synchronous event publisher. The synchronous event publisher instance is private to the asynchronous event publisher. One implication of this delegation is that exception during event delivery events are published by the synchronous event publisher, but the subscribers to the event are registered with the asynchronous event publisher. The asynchro-

```
public interface IEventPublisher
{
    /// <summary>
    /// The exception during event notification event. This event is published
    /// when a subscriber to an event throws an exception during event
    /// notification.
    /// </summary>
    event EventHandler<ExceptionDuringEventNotificationEventArgs>
        ExceptionDuringEventNotification;

    /// <summary>
    /// Publish an event.
    /// </summary>
    /// <param name="eventDescription">
    /// The description of the event to publish.
    /// </param>
    void PublishEvent (EventDescription eventDescription);
}
```

Display 5. The interface offered by the selectable singleton.

nous event publisher therefore subscribes to the exception during event delivery event of the synchronous publisher, and relays the event to its own subscribers.

This event relay can be either synchronous or asynchronous. The asynchronous event publisher relays the event synchronously, rather than asynchronously. The reason for this design choice is that when used with a static jacket, there is another relay involved—that of the static jacket—and this relay will be (if the asynchronous event publisher is in use) asynchronous. Thus, the exception during event publication event would be relayed twice: once by the asynchronous event publisher's relay, and once by the static jacket's relay. Each relay, being a new event publication, effectively reschedules event delivery; so the exception during event delivery event will be twice delayed. However, the delay in event relays is a property of all event relays in the asynchronous event delivery implementation, and special casing this particular event relay is arguably a wrong thing to do.

Implementation-Agnostic Access to Implementations

A convenient way to embody an application-wide choice among competing implementations of a single function—such as the event publication function investigated here—is to have a single application-wide reference to the implementation of choice. This is the purpose of the static jacket class recommended in this Application Note. The .ZIP file attached to this Application Note includes an `EventPublisher` class. The text of this somewhat lengthy class is presented in Appendix A.

The `EventPublisher` static class offers a public `Implementation` property, which when accessed returns the singleton instance of the `IEventPublisher` interface. This interface is shown in Display 5.

A reference to the singleton implementation can be obtained and then used. Alternately, the static jacket class also provides a set of convenience `PublishEvent` static methods which forward to the `IEventPublisher` implementation currently in use.

Now, the event publisher currently in use also publishes `ExceptionDuringEventNotification` events. An interested object can retrieve the event publisher in use and subscribe to this event. Alternately, the static jacket class also provides an `ExceptionDuringEventNotification` static event, which republishes the event from the `IEventPublisher` implementation currently in use. Subscribing to the static jacket class's event maintains the subscription even if the `IEventPublisher` implementation in use changes, which can be an advantage.

The overall effect of the static jacket class is to completely decouple the objects consuming the event publication functionality from the objects providing the event publication functionality. The client objects do

not need to know the type of the provider; they do not need a reference to the provider; they do not need to subscribe to the provider's events. They can, should they choose, obtain a reference to the provider, but this reference is filtered through an interface. The implementation of the singleton object can be replaced with another implementation of the interface at will. The providing classes do not need to be designed as singletons, and it is entirely feasible for there to be multiple instances of the providing classes: clients do not need to access "the singleton" to obtain the services provided by the implementing class.

Driver Program

To demonstrate the effect of the differing semantics of the two implementations, as well as switching implementations of the singleton at run time, the .ZIP file accompanying this Application Note includes a driver program. The driver program consists of three classes: `MainClass`, `ClassA`, and `ClassB`. `MainClass` publishes an event `MainEvent`, and `ClassA` publishes an event `AEvent`. Both `ClassA` and `ClassB` subscribe to `MainEvent`; `ClassB` in addition subscribes to `AEvent`.

When the driver program starts, the default event publisher is `SynchronousEventPublisher`. `MainClass` publishes `MainEvent`. When `ClassA` receives `MainEvent`, it notes the reception and also publishes `AEvent`. When `ClassB` receives `MainEvent` or `AEvent`, it notes it. `MainClass` then changes the active event publisher to `AsynchronousEventPublisher` and publishes `MainEvent` a second time. Again `ClassA` publishes `AEvent` and both `ClassA` and `ClassB` note event receptions.

The output of the program is as follows:

```
ClassA: Subscribing to MainEvent.
ClassB: Subscribing to MainEvent.
ClassB: Subscribing to AEvent.
Main: Using default SynchronousEventPublisher implementation.
Main: Publishing MainEvent.
ClassA: Received MainEvent from Main.
ClassA: Publishing AEvent.
ClassB: Received AEvent from ClassA.
ClassB: Received MainEvent from Main.
Main: Switching to AsynchronousEventPublisher implementation.
Main: Publishing MainEvent.
ClassA: Received MainEvent from Main.
ClassA: Publishing AEvent.
ClassB: Received MainEvent from Main.
ClassB: Received AEvent from ClassA.
Main: Done. Enter ENTER:
```

Note particularly how the order of events reported by `ClassB` differs with the event publisher in use. When the `SynchronousEventPublisher` is in use, `ClassB` receives `AEvent` before it receives `MainEvent`, even though `MainEvent` is the stimulus that gave rise to `AEvent`. When the `AsynchronousEventPublisher` is in use, `ClassB` receives `MainEvent` and then – after `MainEvent` has been fully published – receives `AEvent`.

Summary and Recommendations

This Application Note has described three approaches to implementing the singleton design pattern.

One of these approaches, that of a jacket class with a single static reference to the singleton object, has been demonstrated to address several of the objections to the singleton pattern. This approach might therefore be thought to be superior to the other approaches. An extended example using an event publisher as the singleton of interest has demonstrated the use of this implementation technique and of the ability it gives to select implementations both at design time and at run time.

The recommended approach is therefore that of the wrapper class containing a static (class-wide, rather than instance-wide) reference to the singleton object.

Further Information

This Application Note and a Visual Studio solution containing all source code for the sample application are contained in a .ZIP file at: http://brianhetrick.com/an/AN-9.zip.

The MonoDevelop software used to produce the code described in this Application Note is available for no charge at http://www.monodevelop.com/.

The Raspbian operating system on which this software was produced and on which this Application Note was produced is available for no charge at https://www.raspberrypi.org/.

The LibreOffice office suite used to create this Application Note itself is available for no charge at http://www.libreoffice.org.

This Application Note uses the Book Antiqua font for body text, the Source Sans Pro fonts for headings, headers, and footers, and the Source Code Pro font for computer code. The Book Antiqua font can be licensed from Monotype at http://catalog.monotype.com/family/monotype/book-antiqua and is included in several of Microsoft's products. The Source Sans Pro font is available at no charge at https://adobe-fonts.github.io/source-sans-pro/ . The Source Code Pro font is available at no charge at https://github.com/adobe-fonts/source-code-pro .

Disclaimer

The author believes all content of this Application Note and any software and design it describes is a straightforward application of existing well-known concepts and techniques and is evident to anyone versed in the state of the art. Despite this, the author makes no claim or warranty of any kind that any item developed based on this Application Note, the software it describes, the designs it describes, or any portion of any or all, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country.

Copyright and Grant of License

Appendix A. The static EventPublisher class

```
public static class EventPublisher
{
    #region Non-Public Class (Static) Variables
    private static IEventPublisher _implementation;
    private static object _lockObject = new object ();
    #endregion

    #region Non-Public Class (Static) Methods and Properties
    /// <summary>
    /// Handle an exception during event publication event from the current event
    /// publisher. Republish the event to the subscribers of the event on this
    /// class.
    /// </summary>
    /// <param name="publisher">
    /// The publisher of the event. Ignored.
    /// </param>
    /// <param name="eventArgs">
    /// The event arguments describing the event.
    /// </param>
    private static void Implementation_ExceptionDuringEventPublication
       (object publisher,
        ExceptionDuringEventNotificationEventArgs eventArgs)
    {
        EventDescription description = new EventDescription
           (typeof (EventPublisher),
            ExceptionDuringEventNotification,
            eventArgs);
        description.NotifyOfException = false;
        PublishEvent (description);
    }
    #endregion

    #region Public Class (Static) Methods and Properties
    /// <summary>
    /// Get or set the event publisher implementation is use.
    /// </summary>
    /// <remarks>
    /// If there is no event publisher implementation in use, create a
    /// SynchronousEventPublisher and make that the event publisher
    /// implementation in use.
    ///
    /// An event publisher implementation implementing IDisposable is disposed
    /// when replaced.
    /// </remarks>
    public static IEventPublisher Implementation
    {
        get
        {
            if (_implementation == null)
```

```
            {
                lock (_lockObject)
                {
                    if (_implementation == null)
                    {
                        _implementation = new SynchronousEventPublisher ();
                        _implementation.ExceptionDuringEventNotification +=
                            Implementation_ExceptionDuringEventPublication;
                    }
                }
            }
            return _implementation;
        }

        set
        {
            lock (_lockObject)
            {
                if (_implementation != null)
                {
                    _implementation.ExceptionDuringEventNotification -=
                        Implementation_ExceptionDuringEventPublication;
                }
                IDisposable disposable = _implementation as IDisposable;
                if (disposable != null)
                {
                    disposable.Dispose ();
                }

                _implementation = value;

                if (_implementation != null)
                {
                    _implementation.ExceptionDuringEventNotification +=
                        Implementation_ExceptionDuringEventPublication;
                }
            }
        }
    }

    /// <summary>
    /// Publish an event.
    /// </summary>
    /// <param name="eventDescription">
    /// The event description of the event to be published.
    /// </param>
    public static void PublishEvent (EventDescription eventDescription)
    {
        if (eventDescription == null)
        {
            throw new ArgumentNullException (nameof (eventDescription));
```

```
        }

        Implementation.PublishEvent (eventDescription);
    }

    /// <summary>
    /// Publish an event.
    /// </summary>
    /// <param name="publisher">
    /// The object on behalf of which the event is to be published. Must not be
    /// <c>null</c>.
    /// </param>
    /// <param name="subscribers">
    /// The list of subscribers to the event. May be <c>null</c> to indicate the
    /// absence of subscribers.
    /// </param>
    /// <param name="eventArgs">
    /// The event arguments describing the event. Must not be <c>null</c>.
    /// </param>
    public static void PublishEvent
        (object publisher, Delegate subscribers, EventArgs eventArgs)
    {
        if (publisher == null)
        {
            throw new ArgumentNullException (nameof (publisher));
        }
        if (eventArgs == null)
        {
            throw new ArgumentNullException (nameof (eventArgs));
        }

        if (subscribers != null)
        {
            EventDescription eventDescription = new EventDescription
                (publisher, subscribers, eventArgs);
            PublishEvent (eventDescription);
        }
    }

    /// <summary>
    /// The exception during event notification event. This event is published
    /// when an event subscriber throws on exception while receiving an event.
    /// This event is not published when subscribers throw an exception when
    /// receiving this event.
    /// </summary>
    public static event EventHandler<ExceptionDuringEventNotificationEventArgs>
        ExceptionDuringEventNotification;
    #endregion
}
```