David Berry 24 August 2011

133110 views 376 23

Designing C# Software With Interfaces

> The best way to understand how interfaces improve software design is to see a familiar problem solved using interfaces. First, take a tightly-coupled system design without interfaces, spot its deficiencies and then walk-through a solution of the problem with a design using interfaces.

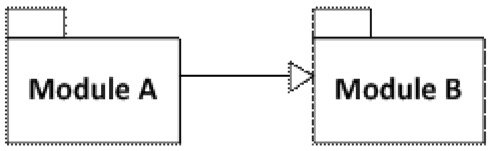
As systems grow in size and complexity, software design must increasingly achieve the separation of concerns, adaptability for future changes and loose coupling. To do this, it is essential to design applications using interfaces. Interfaces are one of the most powerful concepts in modern object orientated languages such as C#, VB.NET or Java. Through the use of interfaces, developers can clearly define the relationship between different modules within a system. This allows a better definition of the boundaries of responsibility between different modules. The result of this is that individual modules are loosely coupled from each other, making the entire system more adaptable to change.

While most C# developers are familiar with the syntax and concept of interfaces, fewer have mastered their use sufficiently to design software around interfaces. Introductory texts on C# provide the basic syntax and a few simple examples, but do not have enough detail and depth to help the reader understand how powerful interfaces are, and where they should best be used. Books on design patterns catalog a vast array of patterns, many focused around interfaces, but often seem overwhelming to those developers who are making their first foray into software design. Consequently, too many developers reach a plateau in their abilities and their careers, and are unable to take the next steps into working with, and designing, more complex software systems.

I believe that many developers can benefit from learning to design with interfaces, and that the best way to learn about how to design with interfaces is to see an example of a practical, familiar problem solved using interfaces. First, I am going to show a system design without using interfaces and discuss some of the deficiencies that can result from this tightly coupled design. Then I am going to re-solve the problem with a design using interfaces, walking the reader through step by step of what decisions I am making and why I am making them. In understanding this solution, the reader will see how this is a more flexible, more adaptable design. Finally, it is hoped the reader will then be able to take these same techniques and apply them to other aspects of system design they may be dealing with.

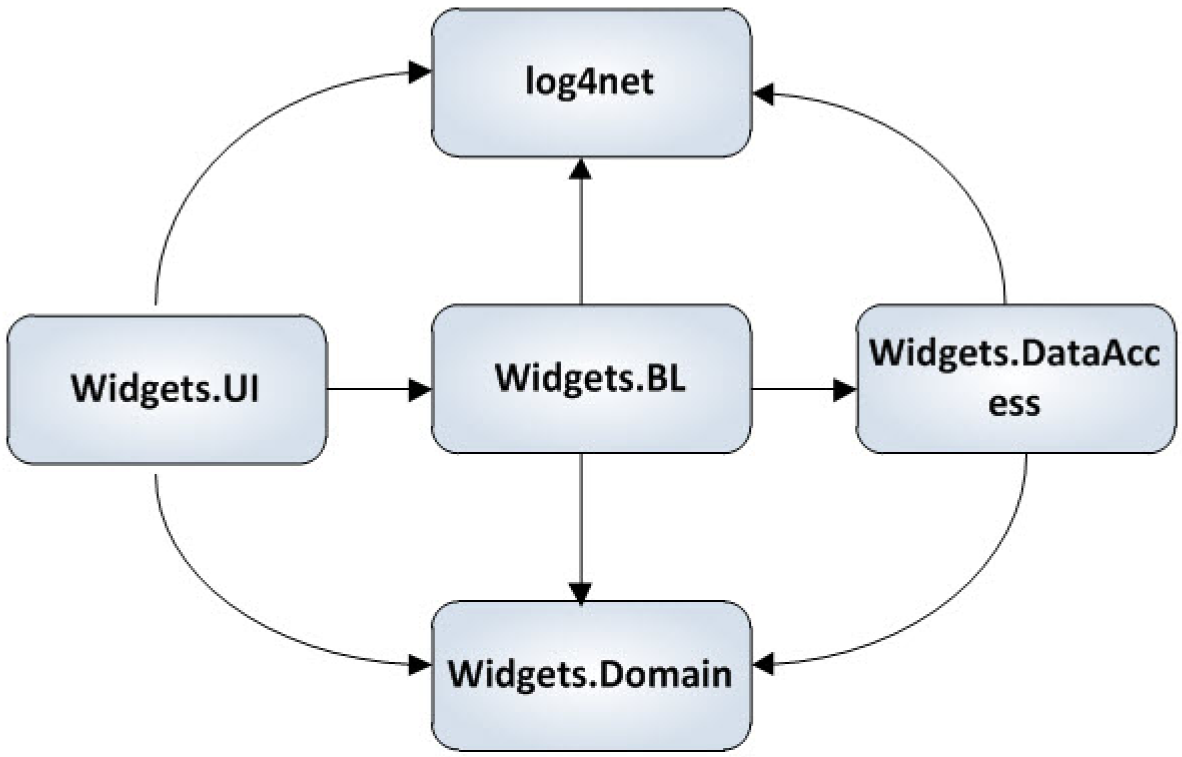
The High Cost of Tightly Coupled Systems

Every university computer science student knows that hardcoding values in a program **is bad design practice**. By writing code with modules that are tightly coupled together, you are in just about as bad of shape. What do we mean by tightly coupled? Imagine we have two modules in our system, module A and module B. If module A depends on module B, then module A and B are said to be tightly coupled. If you make a change to module B, whether a change in a method signature, class layout or any other change, you are going to also have to change module A. Module A has significant knowledge of how module B works, so any changes on module B are going to cascade up to module A.



At this point you are saying, “Of course module A has to know about module B. My classes in module A need to use my classes in module B”. That is a fair statement, and there has to be some amount of coupling for a system work. Otherwise none of the modules or layers in a system could talk to any of the other modules or layers of the system. Where we run into trouble though is when we are tightly coupled to a module that can change. The following example will demonstrate this point.

Almost every application needs to perform some sort of application logging. Several high quality logging frameworks are freely available, including log4net from the Apache Foundation and Enterprise Library from Microsoft. In addition, many companies may have developed their own internal logging frameworks. Let us imagine for a moment that we have developed a typical n-tier system for managing widgets. As part of this system, we have separate projects for each of the major layers of the system-user interface, business logic, domain objects and data access. As part of our design, we have decided to use the log4net framework for logging. The diagram below shows the major components of this system and their dependencies (the arrow points in the direction of the dependency).



We feel as we have done everything right in our design. We have separated out the layers of our system, put our business logic in a separate layer and separated out our data access. Yet, all of our modules in this system remain tightly coupled to each other. For the purposes of this example, we are going to focus on the logging module. Consider the following two scenarios:

- A new CIO just started at our company, and he wants to standardize on Enterprise Library. All projects need to convert to using the logging framework in Enterprise Library within the next three months. In this case, we need to not just change out the reference to log4net with Enterprise Library, but we need to track down all of the logging statements in all of our code and replace them with their Enterprise Library equivalents.
- Our widget application is wildly successful. Another team wants to integrate widget information into their Intranet web application. But this team uses their own, home grown logging framework. So using our widget libraries, we will either be writing log messages using two different frameworks (and hence multiple files) or we need to either convert our libraries to use their logging framework or adapt their application to use log4net.

In both of these cases, we are faced with unpalatable options. Instead of spending time adding features to our system that our business users want, we are ripping out one set of plumbing and replacing it with another. The fundamental problem here is that our system is tightly coupled to a specific logging implementation. Changing from one implementation (log4net) to another (Enterprise Library) represents a significant change to our code. The second example is even worse. Do we change our application to use the homegrown logging system of the other group, or do we try to convince them to use our chosen logging solution (log4net). Another alternative would be to maintain two sets of the widget libraries, one for each logging framework in use. This is clearly unsatisfactory though, because the two libraries will immediately start to grow apart. Finally, they could choose just to go and write their own widget library rather than use ours. But once again, our company is now maintaining two sets of source code to perform the same tasks.

What we are lacking here is any sort of interchangeability with regards to our logging system. By being tightly integrated to one solution to the problem (in this case, log4net), we have sacrificed any sort of flexibility to change to a different solution. If we want to proceed with this change, it comes at a very high cost-namely rewriting and recompiling significant portions of our system.

A Better Way – Designing with Interfaces

Our goal in the above system is to design an intermediate layer which will allow us to easily switch out logging subsystems for whatever is needed. In doing so, we also want to allow flexibility so that a new logging framework could easily be plugged in at a later date. All of this should be accomplished so that any changes to the logging sub-system do not require us to make any changes to our existing application code. Interfaces provide a simple yet elegant solution to precisely this problem.

First, an enum will be defined in our common logging library. The value of the enum will represent the different levels at which log messages can be written out. Log levels are used to filter what messages are actually written to the log in a log system. Generally, levels such as FATAL and ERROR will always be configured to be included in the log system. Levels such as INFO and VERBOSE are only included in development systems or when attempting to debug a problem. Our common interface will define the log levels shown below. The level FATAL is considered most important and VERBOSE the least important. Each message that is written to the log will be assigned one of these levels.

```
    ///<summary>
    /// Enum defining log levels to use in the common logging inter
    /// </summary>
    public enum LogLevel
    {
        FATAL = 0,
        ERROR = 1,
        WARN = 2,
        INFO = 3,
        VERBOSE =4
    }
```

The next step in our refactored design is create an interface that defines how any other code we write will talk to the logging sub-system.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DesigningWithInterfaces.LoggingInterface
{

    /// <summary>
    /// Defines the common logging interface specification
    /// </summary>
    public interface ILogger
    {
        /// <summary>
        /// Writes a message to the log
        /// </summary>
        /// <param id="category"">A String of the category to write
        /// <param id="level"">A LogLevel value of the level of thi
        /// <param id="message"">A String of the message to write t
        void WriteMessage(string category, LogLevel level, string m

    }
}
```

Of all of the code in this article, this interface is the most important. It defines how the rest of the modules in our application or any application are going to talk to the logging subsystem. Any piece of code that wants to put a message in a log will have to do it according the specification defined above. Furthermore, any backend logging system we want to plugin and use must adhere to this specification. The interface defined above is the bridge between the two subsystems. On one side you have the module that wants to write messages to a log. On the other is the concrete logging module (like log4net or Enterprise Library) that will perform the actual details of writing the message out . This interface serves as an agreement about how those two modules will communicate.

The problem we face now is that we have the interface we defined above, but none of our actual concrete logging systems implement the above interface. That is, there is no class in log4net, Enterprise Library or any other logging system out there that directly implements this interface. In fact, they all do logging a little bit differently. What is required is a class that will adapt our logging interface to an actual implementation of a logging framework. This class will fulfill our interface defined above and effectively translate those calls that are defined in our interface into method calls that work with the logging implementation that we have chosen. Such a class is shown below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DesigningWithInterfaces.LoggingInterface;
using log4net;

namespace DesigningWithInterfaces.Log4Net
{
    /// <summary>
    /// Driver class to adapts calls from ILogger to work with a lo
    /// </summary>
    internal class Log4NetLogger : ILogger
    {

        public Log4NetLogger()
        {
            // Configures log4net by using log4net's XMLConfigurato
            log4net.Config.XmlConfigurator.Configure();
        }

        /// <summary>
        /// Writes messages to the log4net backend.
        /// </summary>
        /// <remarks>
        /// This method is responsible for converting the WriteMess
        /// the interface into something log4net can understand.  I
        /// by doing a switch/case on the log level and then callin
        /// appropriate log method
        /// </remarks>
        /// <param id="category"">A string of the category to log t
        /// <param id="level"">A LogLevel value of the level of the
        /// <param id="message"">A String of the message to write t
        public void WriteMessage(string category, LogLevel level, s
        {
            // Get the Log we are going to write this message to
            ILog log = LogManager.GetLogger(category);

            switch (level)
            {
                case LogLevel.FATAL:
                    if (log.IsFatalEnabled) log.Fatal(message);
                    break;
                case LogLevel.ERROR:
                    if (log.IsErrorEnabled) log.Error(message);
                    break;
                case LogLevel.WARN:
                    if (log.IsWarnEnabled) log.Warn(message);
                    break;
                case LogLevel.INFO:
                    if (log.IsInfoEnabled) log.Info(message);
                    break;
                case LogLevel.VERBOSE:
                    if (log.IsDebugEnabled) log.Debug(message);
                    break;
            }
        }
```

The comments in the code above speak for themselves. The above class translates our common interface to something that log4net can understand. This is the way that we can write the rest of our system to work against our common interface, but yet still use a high quality, full featured logging system like log4net to take care of the actual work of writing our logging messages out to a file or a database or whatever we desire.

The real beauty though is that we can define multiple classes that implement the interface, in this case, each class serving as an adapter to a different logging backend. We can write a second class which implements the ILogger interface to support an Enterprise Library backend. All of the application code that uses logging is the same. It is programmed against our ILogger interface. To change which logging backend we use, we only have to substitute a different adapter class to be used by the system. Furthermore, if a new logging backend comes along at some point in the future, all we have to do to use it is to write a new adapter class that fulfills our interface and plug in to the new framework. This is a huge win because we can switch to something in the future we don't even know about yet without rewriting our system. In fact, we can switch to a new framework without modifying any of our application or library code at all, just by writing one simple adapter class. Such a class is shown below.

```
    /// <summary>
    /// Adapter class to use Enterprise Library logging with the co
    /// logging interface
    /// </summary>
    internal class EnterpriseLibraryLogger : ILogger
    {

        public void  WriteMessage(string category, LogLevel level,
        {
            // First thing we need to do is translate our generic l
            // into a priority for Enterprise Library.  Along the w
            // assign a TraceEventType value
            TraceEventType eventSeverity = TraceEventType.Informati
            int priority = -1;
            switch (level)
            {
                case LogLevel.FATAL:
                    eventSeverity = TraceEventType.Critical;
                    priority = 10;
                    break;
                case LogLevel.ERROR:
                    eventSeverity = TraceEventType.Error;
                    priority = 8;
                    break;
                case LogLevel.WARN:
                    eventSeverity = TraceEventType.Warning;
                    priority = 6;
                    break;
                case LogLevel.INFO:
                    eventSeverity = TraceEventType.Information;
                    priority = 4;
                    break;
                case LogLevel.VERBOSE:
                    eventSeverity = TraceEventType.Verbose;
                    priority = 2;
                    break;
            }

            // This creates an object to specify the log entry and
            // values to the appropriate properties
               LogEntry entry = new LogEntry();
            entry.Categories.Add(category);
            entry.Message = message;
            entry.Priority = priority;
            entry.Severity = eventSeverity;

            // This line actually writes the entry to the log(s)
            Logger.Write(entry);
        }
    }
```

The final problem that must solved is how to we select which one of our adapter classes to use. Somewhere in our code we need to locate the correct adapter and create an actual instance class of our adapter to talk to our backend logging framework. At first blush, one might think of writing a code snippet like the following:

```
        public static ILogger GetLogger()
        {
            string logger_key = ConfigurationManager.AppSettings["L
            if (logger_key.Equals("log4net"))
            {
                return new Log4NetLogger();
            }
            else if (logger_key.Equals("EnterpriseLibrary"))
            {
                return new EnterpriseLibraryLogger();
            }
            else
            {
                throw ApplicationException("Unknown Logger");
            }
        }
```

One major problem this code suffers from is that it needs to know about all of the available logging implementations up front, when we write this code. A second problem is that the ILogger interface, all of the adapter classes that implement ILogger and the above piece of code all tend to end up in a single project in order to avoid circular dependencies. Further, each adapter class will have a reference to its individual backend logging assemblies, which will all come along for the ride when we compile the above code. This heavyweight, tightly coupled project is exactly what we were trying to avoid in the first place. In this case, we have used an interface, but we really have not gained very much.

What we really desire is a solution that is truly decoupled and interchangeable, interchangeable to the point that at any time in the future we can substitute in a completely new adapter class and new logging backend. By utilizing the Reflection API in .NET, we can accomplish just that. The Reflection API allows us to dynamically instantiate an object of a class. That is, we can create an object of a class without calling new() on the class, but by providing the name of the class and assembly to the Reflection API. The following code shows how to do this.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Configuration;
using System.Reflection;

namespace DesigningWithInterfaces.LoggingInterface
{

    /// <summary>
    /// Factory class to get the appropriate ILogger based on what
    /// the App.Config file
    /// </summary>
    public class LoggerFactory
    {

        #region Member Variables

        // reference to the ILogger object.  Get a reference the fi
        private static ILogger logger;

        // This variable is used as a lock for thread safety
       private static object lockObject = new object();

        #endregion


        public static ILogger GetLogger()
        {
            lock (lockObject)
            {
                if (logger == null)
                {
                    string asm_name = ConfigurationManager.AppSetti
                    string class_name = ConfigurationManager.AppSet

                    if (String.IsNullOrEmpty(asm_name) || String.Is
                        throw new ApplicationException("Missing con

                    Assembly assembly = Assembly.LoadFrom(asm_name)
                    logger = assembly.CreateInstance(class_name) as

                    if (logger == null)
                        throw new ApplicationException(
                            string.Format("Unable to instantiate IL
                            asm_name, class_name));
                }
                return logger;
            }
        }
    }
}
```

This is a factory class. To get a reference to the appropriate ILogger object, we don't call new() in our application code, we call the static method LoggerFactory.GetLogger(). The first time this method is called, it will look into the app.config file and get the assembly and class name of the adapter class for the logging system we want to use. It then uses the reflection API to load that assembly into the .NET Runtime and then get an instance of the class. It then keeps a reference to this ILogger object to use on subsequent calls to this method. Finally, all of this logic is wrapped in a lock statement to assure thread safety.

The last paragraph may seem complex, but when you boil it down, what is really happening is that we are creating the appropriate ILogger to use based on two entries in the app.config file. Now, we can change what logging system is being used by our application simply by changing entries in the config file. We do not have to make any code changes or recompile any code, simply to change the values in the config file. The .NET runtime still has to be able to locate and load the DLL with the adapter class in it (and in turn, locate and load the logging framework DLL's you intend to use). But our system is truly decoupled now. Our application code is programmed against an interface such that any backend logging system can be used. If we want to support some new logging backend that hasn't been invented yet, we simply have to write a new adapter class, compile it, copy the DLL to where our application can find it along with the new logging system DLL's and change the config file. We have achieved true interchangeability within our system, so now different logging frameworks are merely plugins, and we can select whichever one suits our needs best.

Putting It All Together

Here is what the new design looks like in terms of a UML diagram:

The classes Log4NetLogger and EnterpriseLibraryLogger both implement a common interface, ILogger. Log4NetLogger and EnterpriseLibraryLogger both serve as adapter classes, adapting the common logging interface we defined in ILogger to specific implementations for their respective backend logging frameworks. The class LoggerFactory is a factory class. It contains just one static method, GetLogger(), which will figure out the appropriate implementation of ILogger we are using, create the appropriate adapter class via reflection (Log4NetLogger or EnterpriseLibraryLogger) and return it to us.

From the perspective of any application, the application only knows about three types: the enum LogLevel since it defines the logging levels, LoggerFactory which is used to get a reference to ILogger and ILogger itself. Any code that uses logging has no knowledge of whether Log4Net, Enterprise Library or another system is being used. That is the purpose of this design. The application is programmed against an interface (in this case ILogger), but has no knowledge of what specific implementation (log4net or Enterprise Library) is being used. The design is now said to be loosely coupled.

Every design has tradeoffs, and this design is no exception. By programming to a common specification (the interface), we have gained the ability to easily switch out different logging frameworks without any impact to our application code. What we have given up though is the capability to use any advanced or unique features of an individual logging system. By programming to an interface, we can only make use of the features exposed by that interface, not any of the other capabilities that underlying implementation may possess. For example, Enterprise Library contains a number of additional features not in our logging interface:

- The ability to specify a title as well as a message for each log entry
- The ability to specify multiple categories for a log message, thereby potentially directing it to multiple destinations
- The ability to define more granular levels through the use of the priority field in addition to the event severity
- The ability to specify an Event ID value to further categorize event messages

Of course, I have purposely left the logging interface very generic and simplistic for the purposes of this article. But this list of sacrificed features still serves to illustrate an important point. To program to an interface, you will most likely give up access to any unique features and advanced capabilities that a particular implementation provides. In this case, if we merely want to perform general purpose logging, we can accept this tradeoff because our desire to achieve flexibility an interchangeability outweighs our need for the unique features above. However, if those unique features were determined to be essential requirements, we would be faced with two choices. First, we could forgo the use of an interface and program directly against our target implementation. If those features are that important to us, this may be an acceptable tradeoff. Second, we could refactor our interface to take additional parameters that could be passed on to our adapter class and ultimately our target framework. In this case though, every adapter class must be modified to fulfill the interface. For example, let’s say that having the ability to specify an event id was a required feature for our logging interface. We would have to modify ILogger to accept event id in its WriteMessage() event, and both Log4NetLogger and EnterpriseLibraryLogger. Since log4net doesn’t have a concept of event id, we have to figure out something to do with the event id passed in. We could throw the value away, but more likely, we may add the value into the message in some formatted fashion. In this case, the tradeoff is additional complexity. To include a feature that is unique to Enterprise Library, we have to add some logic and therefore complexity to the Log4netLogger. Any system design will include such tradeoffs. What is important is to evaluate what is most important in each particular design and make the right tradeoffs for the system being implemented.

Design Patterns

The design above is what is known as the Bridge design pattern. This pattern is also sometimes known as the plugin pattern because each specific implementation can be easily "plugged in" to a program. The concept of the bridge pattern is that as long as an implementation fulfills the defined interface, implementations can be freely substituted for each other. Since the application is programmed against the interface, it is unaffected by switching to a new implementation.

The bridge pattern is very commonly used with regards to database drivers. ADO.NET, OLE DB and ODBC are all standard data access technologies for which Microsoft publishes a written specification. Other database vendors then take these specification and develop database drivers for their specific database management system. As long as those drivers adhere to the published specification, they can be easily used by any application without the application needing to know what specific database backend is used. An example of this is Microsoft Excel. Excel knows how to query data from ODBC. Excel doesn't know if your backend database is SQL Server, Oracle, MySQL or even a text file. Excel just knows that it is talking to ODBC, and there is a driver that implements the ODBC specification. As long as you have an ODBC driver for your data source, Excel can get data from it. This is a classic example of the bridge pattern and the power of programming against an implementation rather than a specific interface.

The second design pattern present is the Adapter pattern. Both Log4NetLogger and EnterpriseLibraryLogger serve are examples of the adapter pattern in action, effectively translating the ILogger interface methods into something that each backend logging framework can understand. Without the adapter classes in the middle, ILogger and the logging frameworks would be incompatible. With the adapter in the middle, the two are able to work together.

The third and final design pattern here is the factory pattern. The factory pattern is a design pattern that encapsulates the details of how to get an instance of an object within the factory. In this example, we do not want code in other modules to use new() to create a new ILogger instance. Doing so would defeat the whole purpose of putting using an interface in the first place, because then some code somewhere would still be bound to a specific implementation. Therefore, we have created the LoggerFactory class to act as a factory that handles the details of obtaining an instance of the correct object. In this case, these details involve looking in the app.config file for the name of the appropriate class and using the reflection API to obtain an instance. All of this code is encapsulated in the factory class, so other modules can, with ease, obtain a reference to the correct Logger object without knowing all of these details.

Design patterns have become a hot topic in the .NET community over the last few years, with literally hundreds of books, articles and screencasts to explain the details of all of the various patterns available. What is more important than being able to rattle off the intricacies of each and every pattern at the drop of a name is to understand the design concepts that are involved. The focus in software design should be on good design principles like clean separation of modules, separating the interface from implementation and decoupling different parts of the system. Design patterns offer established solutions to common problems in software development. However, you shouldn't become a slave to patterns or force fit patterns into your design. Follow the principles above, and the places where a pattern can effectively help solve a problem will naturally emerge.

When Should You Design with Interfaces

The last step in learning how to successfully design with interfaces is being able to recognize when using an interface will result in a superior design. The following are some classic examples of when designing with and coding to an interface is appropriate and generally results in a better design.

- Whenever a third party component or service is used. Unfortunately, libraries change, companies merge and acquire new products and stop supporting old ones. If you are able to distill out the important operations that a component or service performs into an interface, you are generally in a much better position if you use the design outlined above and call the component or service via an interface. This way, if you want to switch to a competitor's product in the future, you have the flexibility to do so simply by writing a new adapter class. Secondly, if the component or service ever changes or is upgraded, it will now be much easier to test, because you are testing that the new implementation fulfills the interface. As long as the new implementation fulfills the interface, you can have high confidence the entire system will still work without having to test each and every feature in the system as a whole.
- Another classic example is to hide your data access code behind an interface or series of interfaces. In some cases this is done so you can have vendor specific implementations of the data access layer-that is, one data access implementation for SQL Server, one for Oracle, one for DB2, etc. In this way, each data access implementation can utilize vendor specific features (example, identity columns in SQL Server, sequences in Oracle) but the rest of the application does not need to know about these details

- Application settings can be loaded from different places, including the app.config file, a database, an XML file on a network share or the Windows registry. It would be very straightforward to abstract the operation of getting an application setting to an interface, and then provide a specific implementation to read the settings from each location. This technique could be useful if for example, today application settings are stored in the Windows registry for legacy reasons, but in the future, you are planning to transition these settings to a different location, like an XML file in the user's application settings directory. Programming to an interface allows you to provide one implementation for legacy compatibility (the Windows registry) and a second implementation for your desired state (a custom XML file). The power of using an interface though is that when you make the switch, you simply have to update a configuration setting somewhere, rather than rewrite or even recompile code.

When designing with interfaces, there are some design guidelines you should follow. One of the most important is to keep your interfaces focused on the problem you are trying to solve. Interfaces that perform multiple unrelated tasks tend to be very difficult to implement in a class. A class may only want to implement part of the interface because that is all that is needed, but is required to implement the entire interface. An interface should clearly and concisely communicate what purpose it serves and what functionality it provides. This makes it clear to implementers what is expected and clear to users of the interface what functions the module can perform. When an interface starts trying to perform too many tasks, it is too easy for the original purpose of the interface to become lost, defeating much of the value of having an interface in the first place.

A second guideline is to make sure the interface does not contain too many methods. Too many methods makes implementing the interface difficult as the implementing class has to provide for each and every method in the interface. At best, this is tedious. At worst, the implementer may be tempted to "stub out" methods that they don't consider important by providing an empty or underdeveloped implementation. In this case, while the class provides a method for each method in the interface, it does not truly fulfill the interface. This becomes a problem if you have application code expecting certain functionality of an interface since a method is provided, but the implementation is but a stub. By keeping the number of methods in an interface reasonable and keeping those methods focused on the functionality that the interface is supposed to provide, it is much more likely the interface will be used and correctly implemented.

A third guideline to remember is to not allow implementation specific functionality creep up into the interface. Too often, when a development team has an existing module from which they are trying to extract an interface, they will include functionality in the interface that is very specific to the current implementation that exists. This becomes a problem when you want to write a different class that implements the interface. This then limits the usefulness of the interface, because the interface itself is really tied to a specific implementation, which is not the point. An interface should define the common functionality that the module or subsystem will perform. Any implementation specific logic must be contained inside of an implementing class, not exposed as a method on the interface itself. The interface is a definition of what functionality the module provides, not a constraint on how an implementing class must provide that functionality.

A fourth and final guideline is to keep in mind is that while some level of abstraction is positive, too many levels of abstraction lead to code that is over-complex and difficult to maintain. In this article, I have focused on the use of interfaces to provide a level of abstraction between different subsystems of an overall application. In this case, providing a level of abstraction between the logging system and the application code has a clear purpose and helps to provide a level of separation between the logging component and the rest of the application. Including additional levels of abstraction through would probably only serve to make the design more difficult to understand. Using an interface should help more clearly define what the role of a module or unit of code is, and therefore lead to a design that is clearer to understand, not more complex. When in doubt, ask a colleague to review the design and explain it back to you. If the design is correct, they should be able to provide an explanation of the purpose of each interface in the system. If they struggle to understand the purpose of a particular interface, you may want to rethink what you were trying to accomplish with that interface in the first place and whether the code reflects your intentions.

Conclusion

Designing with interfaces will result in cleaner separation of responsibilities between different subsystems within an application. By programming to an interface, applications become much more flexible because different subsystems can be easily switched out if the need arises. Furthermore, if a subsystem does have to be switched out, the burden of testing is reduced, because now you can focus most of your testing efforts on insuring that the new implementation properly fulfills the interface rather than testing the entire system as a whole.

This article covered an example of how interface design could be applied to the problem of application logging frameworks. I hope that, by understanding the example given in this article, more developers will come to recognize the power and flexibility that interfaces bring and will start using interfaces to design more flexible software systems.