.NET Logging Wrapper 3.0 Component Specification

1. Design

The .NET Logging Wrapper component provides a standard logging API with support for pluggable back-end logging solutions. Components utilizing the .NET Logging Wrapper are not tied to a specific logging solution. A change to the pluggable back-end logging solution does not require a code change to the .NET Logging Wrapper component.

LogManager is able to create the Logger instance from a specified or default configuration namespace. By default, configuration is loaded using the File Based Configuration component. This makes the component backwards-compatible with the previous version that used ConfigManager. If an application using this component wants to switch to another logging implementation, we can simply make the change to the configuration file related to the namespace used to create the Logger without changing any code of the application. A logger can also be created by passing any IConfiguration instance. The IConfiguration can be any implementation of the interface. It does not have to be file based.

The Logger instances created by the LogManager from different namespaces are different, and they are instantiated with those different configuration files. This allows us to have several different logs which are configured differently.

This component may be used heavily in applications or components, and can be used in multi-threaded environment, so it is preferable to keep this component as lightweight as possible. All existing classes in this component are stateless or immutable to ensure thread-safety. New Logger implementations added to this component should be immutable if possible.

Log4NETImpl should always use strongly named log4net assembly (which is in the release directory of the log4net distribution), since this component will be signed too.

All the old public APIs are kept for backward compatibility, but they are labeled with Obsolete. They should not be used in the future. (This applies to both versions 2 and 3.)

When used in the ASP.NET environment, proper authorizations should be granted to the ASP.NET process to make sure this component functions well. Here are some details about it:

Two checks should be performed before accessing resources from ASP.NET applications: code access security and operating system/platform security. The resources can only be accessed only if both checks succeed.
The Web Application can run in two modes: Full Trust and Partial Trust. When running in the Full Trust mode, it has unrestricted code access permissions, so we

only have to ensure the user running the Web Application is granted enough permission (i.e can read configuration files and write to log files/directory). When running in the Partial Trust mode, other than the former requirement, we should configure Web Application policy properly in the web.config file. I will only discuss the Full Trust case here, since the Partial Trust Web Application requires that those using strongly named assemblies should be labeled with **AllowPartiallyTrustedCallersAttribute** attribute. Other than this, the configuration of the policy file is relatively easy. You can refer to the following link for more information:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod82.asp

DiagnosticImpl class will use the EventLog to log messages. So if using DiagnosticImpl in Web Application, we should ensure the process account running the app is able to create event sources. To be more specific, it should be able to create registry keys beneath:
`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog.`
Or you can create the events sources before running the app.

Log4NETImpl class will use the log4net component to log messages, which needs to access a configuration file to configure the log4net. So you should grant the read permission in the ACL of the file to the ASP.NET process account at least.

The Logging Application Block backend needs similar permissions as Log4NETImpl when running in ASP.NET.

The Enterprise Logging logger will need permission to create outgoing network connections to the enterprise logging endpoint. Assuming that you are using the Enterprise Logger in conjunction with other web services, no special configuration should be needed to make this possible.

With regards to deployment and packaging issues, the .NET runtime does not load (or need available) any class until the first time it is used in an application. In regards to this component, this means that there is no need to break the component up into separate assemblies. Distribution of the ELSImpl/ELSAppender class does not require the Enterprise Logging Service or WCF libraries to be deployed as long as no loggers are configured to use these implementations. Similarly for the Logging Application Block using classes in this component.

Notes on this document:

- Changes from version 2.0 to 3.0 are marked in blue text
- Additions in version 3.0 are marked in red text

In addition to adding support for all the requirements in version 3.0, this component features one significant addition:

- EnterpriseLibraryLogger, which is a Logger implementation that allows using the Enterprise Library Logging Application Block as the backend logging solution.

It also features one major change from previous versions:

- Log methods are now permitted to throw LoggingException. Any logger can be wrapped in ExceptionSafeLogger to ensure it does not throw any exceptions. Wrapping with ExceptionSafeLogger is the default, and maintains backwards compatible behavior. Wrapping loggers in an ExceptionSafeLogger can be disabled. This is useful when developing a component, as it allows developers to detect when they are calling log methods incorrectly (for example, passing an incorrect number of parameters to format the message). Previously this kind of mistake was much harder to detect.

1.1 Design Patterns

LogManager implements Factory pattern to create Logger instances.

DiagnosticImpl and Log4NETImpl implement the Adapter pattern to wrap EventLog and log4net respectively. EnterpriseLibraryLogger and ELSLogger also implement the Adapter pattern to wrap the Logging Application Block and Enterprise Logging Service backends.

ExceptionSafeLogger and LevelFilteredLogger implement the Decorator pattern to add functionality to the underlying loggers.

1.2 Industry Standards

XML

1.3 Required Algorithms

1.3.1 Zero-configuration for log4net

The InitializeZeroConfiguration method of the Log4NETImpl class is responsible for creating an appropriate log4net configuration file for each of the zero configuration options. Details on configuring the appenders can be found at http://logging.apache.org/log4net/release/config-examples.html. The configurations that should be used is as follows:

Test:
The configuration should use a FileAppender that writes to the file ../../test_files/log.txt

Component:
The configuration should use a FileAppender that writes to the file ./log.txt.

Certification:

The configuration should use a RollingFileAppender to write to the logs folder. Files should be rolled over daily, using a pattern of “yyyy-mm-dd" for the files. The configuration should be set up so as never to delete old logs.

Client Debug:
This is the same as certification, but configured to keep only the 30 most recent logs.

Client Stress:
This is the same as client debug, but configured so that only messages at the Level.Error (this is the log4net level) or higher are recorded.

Release:
This is the same as client debug, but configured so that only messages at the Level.Warn (this.is the log4net level) or higher are recorded.

1.4 Component Class Overview

LogManager
LogManager provides one overloaded CreateLogger methods to create Logger instances from the configuration values loaded from an IConfiguration instance. By default, app.config file used to get the default logger name to be created. If no specified logger name, DiagnosticImpl will be created.

Logger
Logger abstract class includes the log name and default logging level properties. And it has public Log method to log the message directly. Loggers that are designed to be dynamically created based on configuration settings should have a constructor that takes an IConfiguration argument. They should also have a static InitializeZeroConfiguration method that takes a ZeroConfigurationOption argument and an IConfiguration argument.

Level
Level enumeration indicates the logging level when logging messages.

DiagnosticImpl
DiagnosticImpl class extends from Logger abstract class to log the message with the EventLog class of .NET framework.

Log4NETImpl
Log4NETImpl class extends from Logger abstract class to log the message with the log4net.ILog of log4net 3rd party component.

Log4NETLevel
Log4NETLevel is an inner class of Log4NETImpl to map the logging level of this component to that of log4net.

NamedMessage

The NamedMessage class is a tiny data storage class that stores all the information for one named message. It does not have any behavior, only a few getters. This class is used by the Logger class (and implementations) to translate between a message name and all the information that is needed to actually log a message.

ExceptionSafeLogger
The ExceptionSafeLogger class wraps another Logger to guarantee that none of the Log methods (and the IsLevelEnabled method) throw an exception. Each method of this class forwards to the corresponding method of the underlying logger. If an exception results, it tries to log the resulting exception to the exception logger. If this attempt to log the exception throws an exception, this second exception is caught and ignored. This class is not designed to be created dynamically, so it does not have the IConfiguration constructor or the InitializeZeroConfiguration method.

LevelFilteredLogger
The LevelFilteredLogger wraps another Logger to filter out any calls to the Log methods that are logged at any of the filtered levels. Each method of this class forwards to the corresponding method of the underlying logger if the level is not filtered. This class is used by LogManager when creating a logger if the filtered_levels property is present in the configuration. This class is not designed to be created dynamically, so it does not have the IConfiguration constructor or the InitializeZeroConfiguration method.

ZeroConfigurationOption
The ZeroConfigurationOption enum defines the valid values that can be specified in the default_config configuration setting to make use of the zero-configuration setup of this component. This value is read by the LogManager and then one of these enum values is passed to the InitializeZeroConfiguration method of the configuration specified logger.

ELSImpl
The ELSImpl class is a simple Logger implementation that forwards all Log and LogNamedMessage calls to an instance of the EnterpriseLoggingService. Like all Logger implementations, it has the IConfiguration constructor and required InitializeZeroConfiguration static method.

ELSAppender
The ELSAppender is a custom log4NET appender that forwards logging events to a TopCoder Enterprise Logging Service instance. It follows the basic pattern for an ELSAppender, inheriting from a skeleton appender provided by the framework.

LoggingWrapperTraceListener
The LoggingWrapperTraceListener class is an extension of the TraceListener class that is designed to be plugged into the Enterprise Library Logging Application Block. All messages logged through the Logging Application Block will be directed to this

trace listener, which will then relay them to a Logger from this component, which will then relay them to a backend logging solution.

EnterpriseLibraryLogger
The EnterpriseLibraryLogger is a TopCoder logger that allows using the Enterprise Library: Logging Application Block as the backend logging destination. This class performs the inverse function to LoggingWrapperTraceListener. All messages logged to this class are redirected to the Logging Application Block, using the specified category.

1.5 Component Exception Definitions

ConfigException [Custom]
This exception will be thrown from the CreateLogger methods and constructor taking a properties dictionary of Logger and its subclasses if the logger cannot be created successfully from the configuration values.

LoggingException [Custom]
LoggingException is the base class for all exceptions thrown from the Log and LogNamedMessage methods of the Logger class. Generally, a LoggingException indicates that the backend logging solution encountered an error. For the specific condition that message formatting failed, a MessageFormattingException is thrown instead. Future versions of the component may add more specific exceptions.

MessageFormattingException [Custom]
The MessageFormattingException is thrown from the Log and LogNamedMessage methods of the Logger class. It indicates that the parameters to these methods could not correctly be combined with the message string. Commonly, this will indicate a failure of a String.Format call. For some logging implementations, it may indicate a similar failure in the backend logging solution.

PluginException [Custom & Obsolete]
Exception class for all pluggable implementation exceptions.

InvalidPluginException [Custom & Obsolete]
Exception for incorrect pluggable logging implementations (without implementation of Logger class).

ArgumentNullException
This exception will be thrown when the given argument is null. Refer to documentation tab for more details.

ArgumentException
This exception will be thrown when the given string is empty. Refer to documentation tab for more details.

1.6 Thread Safety

Forum thread explicitly states that this component is not necessary to consider the obsolete methods when implementing the thread-safety functionality.
This component is thread-safe. The LogManager is stateless, and the Logger abstract class and its two subclasses are all immutable. log4net documentation explicitly states log4net component is thread-safe, so it always safe to log with log4net in Log4NETImpl class in multiple-threaded environment. And the EventLog documentation does not guarantee its thread-safety, so we should lock on the EventLog instance when logging in the DiagnosticImpl class to ensure it.
If new subclasses of Logger are added, special attention should be paid on the thread-safety aspect.

In version 3.0, almost all additions are immutable and hence thread-safe.

2. Environment Requirements

2.1 Environment

.NET Framework 3.0

2.2 TopCoder Software Components

Configuration API 1.0
Used by LogManger to get configuration values to use in creating loggers.

File Based Configuation 1.0.1
Used by LogManager as the default way to get configuration values (for backwards compatibility to read ConfigManager files) and for ELSAppender to read the configuration.

Enterprise Logging Service 1.0
Provides the logging backed for the ELSImpl logger (not required by this component to compile and test, but for ELS usage)

Set Utility 1.0
Used by Configuration API

2.3 Third Party Components

log4net (any version)

The Log4NETImpl class uses this component to log the messages. The log4net assembly must have strong name in order to work this component correctly. And the assembly in the **release** directory of the downloaded package is strongly named.

Enterprise Library 3.1

(http://msdn2.microsoft.com/en-us/library/aa480464.aspx)

The Enterprise Library Logging Application Block is used as the backend for the EnterpriseLibraryLogger, and can be configured to log to the TopCoder system though the use of the LoggingWrapperTraceListener.

3. Installation and Configuration

3.1 Package Name

TopCoder.LoggingWrapper
TopCoder.LoggingWrapper.EntLib
TopCoder.LoggingWrapper.ELS

3.2 Configuration Parameters

3.2.1 Configuration values shared by all the Logger implementations:

Parameter	**Description**	**Values**
logger_class	The full name of the Logger implementation class. This field is required.	TopCoder. LoggingWrapper. Log4NETImpl
logger_assembly	The assembly name of the Logger implementation class. This field is optional.	LoggingWrapper.dll
logger_name	The name of the created Logger. This field is required.	any non-empty valid string.
propagate_exceptions	Tells whether exceptions are propagated from the logger, or caught and swallowed by the logger. Optional, default is false.	A boolean value.
ExceptionLogger	If propagate_exceptions is false, this nested namespace contains the configuration for a logger to log exceptions emanating from the main logger. Optional.	<Nested namespace>
default_config	Used to specify the type of zero-configuration setup to use. Optional. Its absence means not to use zero-configuration.	One of the ZeroConfigurationOptionValues, such as Test.
filtered_levels	Tells which levels of log messages should be filtered out and not logged. A multi-valued property. Optional.	Valid values of the Level enum. DEBUG, ERROR, etc.
default_level	The default logging level of the Logger, the level string should be able to be parsed into Level enumeration values. This field is optional. Default to Level.DEBUG.	Valid value of the Level enum. DEBUG, ERROR, etc.

3.2.2 *Configuration for named messages*

Named messages are configured in a nested namespace called NamedMessages under the configuration for the logger. Within the NamedMessages namespace, there can be any number of nested namespaces with the following configuration values:

Parameter	**Description**	**Values**
text	The text of the message. This should include any backend specific parameter references. Required.	Any string (except an empty string). For example: “Component {0} malfunctioned due to invalid input of type {1}”. An example intended for the log4NET backend is “Component %property{component} malfunctioned due to invalid input of type %property{type}”
default_level	The default level at which the message is to be logged, if a specific level is not provided in the LogNamedMessage call. Optional, defaults to the default level of logger it belongs to or Level.DEBUG if it belongs to no logger.	Any of the defined Level values. DEBUG, ERROR, etc.
parameters	Names to map parameters to when passing to the backend logging system (e.g. log4NET). This setting can have multiple values, as it is always Optional (if not present, the message will take no arguments).	Any non-empty strings. May be 0 or more occurrences. For example, to make use of the log4NET example above, the two values “component” and “type” should be configured.

Each such namespace under the NamedMessages namespace takes the name of the namespace as its message identifier to be used when calling the LogNamedMessage method.

3.2.3 *Configuration values specific to DiagnosticImpl class:*

Parameter	**Description**	**Values**

source	The source name used to create EventLog. Required (optional in zero-configuration mode, defaults to “TopCoder Logger”).	Any non-empty string.

3.2.4 *Configuration values specific to Log4NETImpl class:*

Parameter	**Description**	**Values**
config_file	The config file to set up the log4net.LogManager. Required (optional in zero-configuration mode, defaults to “log4net.config”).	Must be a valid reference to a file. Example: ../../conf/log4net.config

3.2.5 *Configuration values specific to EnterpriseLibraryLogger class:*

Parameter	**Description**	**Values**
Category	The category under which messages are logged to the Logging Application Block backend. Required (optional in zero-configuration mode, defaults to “TopCoder Logger”).	Any non-empty string.

3.2.6 *Configuration values specific to ELSImpl and ELSAppender class:*

Parameter	**Description**	**Values**
`loggingService`	A configuration section for generating the client used to connect to the service. The object is created using an embedded Configuration API Object Factory.	required

3.2.7 *Configuration for ELS log4net appender and Logging Application Block TraceListener*

Documentation for configuring these two frameworks is readily found on the internet. For specific examples of configurations that use the classes in this component, please see the demo section.
Note that for ELSAppender configuration is read from "TopCoder.LoggingWrapper.ELS.ELSAppender" configuration. You should use the preload.xml to enable this configuration.

Here is an example configuration file to be used by Configuration API:

```
<configuration>
    <TopCoder.LoggingWrapper.LogManager logger_class="TopCoder.LoggingWrapper.DiagnosticImpl"
```

```
                                        logger_assembly="TopCoder.LoggingWrapper.Test.dll"
                                        logger_name="LogTest"
                                        propagate_exceptions="false"
                                        default_config="Test"
                                        default_level="INFO"
                                        source="DefaultSource"
                                        filtered_levels="FATAL;ERROR">

        <ExceptionLogger logger_class="TopCoder.LoggingWrapper.DiagnosticImpl"
                         logger_name="LogTest"
                         propagate_exceptions="true"
                         source="ExceptionSource"/>

        <NamedMessages>
            <SimpleMessage text="The parameters are {0} and {1}"
                           default_level="INFO"
                           parameters="myParam1;myParam2"/>
            <Log4NetMessage text="The parameters is %property{myParam}"
                            default_level="WARN"
                            parameters="param"/>
        </NamedMessages>
    </TopCoder.LoggingWrapper.LogManager>

    <!-- Example configuration with minimum setting -->
    <TopCoder.LoggingWrapper.Minimum logger_class="TopCoder.LoggingWrapper.SimpleLogger"
                                     logger_name="MinimumLogger"/>

    <!-- Example configuration with exception propagated -->
    <TopCoder.LoggingWrapper.ExceptionPropagated
logger_class="TopCoder.LoggingWrapper.SimpleLogger"
                                                 logger_name="ExceptionPropagatedLogger"
                                                 propagate_exceptions="true"/>

    <!-- Example configuration with levels filtered -->
    <TopCoder.LoggingWrapper.LevelsFiltered
logger_class="TopCoder.LoggingWrapper.SimpleLogger"
                                            logger_name="LevelsFilteredLogger"
                                            propagate_exceptions="true"
                                            filtered_levels="FATAL;ERROR"/>

    <!-- zero configuration for test -->
    <TopCoder.LoggingWrapper.ZeroConfiguration
logger_class="TopCoder.LoggingWrapper.SimpleLogger"
                                               logger_name="LevelsFilteredLogger"
                                               propagate_exceptions="true"
                                               default_config="Release"/>

    <!-- Example configuration for DiagnosticImpl -->
    <TopCoder.LoggingWrapper.DiagnosticImpl
logger_class="TopCoder.LoggingWrapper.DiagnosticImpl"
                                            logger_name="LogTest"
                                            propagate_exceptions="true"
                                            source="SourceForDiagnosticImpl"/>

    <!-- Example configuration for Log4NETImpl -->
    <TopCoder.LoggingWrapper.Log4NETImpl logger_class="TopCoder.LoggingWrapper.Log4NETImpl"
                                         logger_name="Log4NETLogger"
                                         propagate_exceptions="true"
                                         config_file="..\..\test_files\v3\log4net.config"/>

    <!-- Example configuration for ELSImpl -->
    <TopCoder.LoggingWrapper.ELSImpl logger_class="TopCoder.LoggingWrapper.ELS.ELSImpl"
                                     logger_name="LogTest"
                                     default_level="INFO"
                                     propagate_exceptions="true">
        <object name="loggingService">
            <assembly value="TopCoder.LoggingWrapper.Test.dll"/>
            <type_name value="TopCoder.LoggingWrapper.ELS.ILoggingServiceDummyImpl"/>
        </object>
    </TopCoder.LoggingWrapper.ELSImpl>
    <!-- Example configuration for ELSAppender-->
    <TopCoder.LoggingWrapper.ELSAppender logger_class="TopCoder.LoggingWrapper.ELS.ELSImpl"
                                     logger_name="LogTest"
                                     default_level="INFO"
                                     propagate_exceptions="true">
        <object name="loggingService">
            <assembly value="TopCoder.LoggingWrapper.Test.dll"/>
            <type_name value="TopCoder.LoggingWrapper.ELS.ILoggingServiceDummyImpl"/>
        </object>
    </TopCoder.LoggingWrapper.ELSAppender>

    <!-- Example configuration for EnterpriseLibraryLogger -->
    <TopCoder.LoggingWrapper.EnterpriseLibraryLogger
logger_class="TopCoder.LoggingWrapper.EntLib.EnterpriseLibraryLogger"
                                                     logger_name="EnterpriseLibraryLogger"
                                                     propagate_exceptions="true"
                                                     Category="MyCategory"/>

    <!-- configuration using SimpleLogger for testing -->
    <TopCoder.LoggingWrapper.SimpleLogger logger_class="TopCoder.LoggingWrapper.SimpleLogger"
                                          logger_name="SimpleLogger"/>

        </configuration>
```

3.3 Dependencies Configuration

The file configuration should be properly configured to make it work correctly. Log4Net and the Enterprise Library should be properly deployed if those backends are being used.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘nant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Install third-party component (if needed), then follow demo.

4.3 Demo

4.3.1 Create Logger from the LogManager

// create the logger from the specified namespaces, one for each of the
// configured namespaces.

```
using CM = System.Configuration.ConfigurationManager;
CM.AppSettings[TestHelper.DEFAULT_LOGGER_CLASS_APP_SETTING_NAME]
         = "TopCoder.LoggingWrapper.Log4NETImpl";
     // logger1 is a Log4NETImpl instance
     Logger logger1 = LogManager.CreateLogger();

     CM.AppSettings[TestHelper.DEFAULT_LOGGER_CLASS_APP_SETTING_NAME]
         = "TopCoder.LoggingWrapper.DiagnosticImpl";
     // logger2 is a DiagnosticImpl instance
```

```
        Logger logger2 = LogManager.CreateLogger();

        CM.AppSettings[TestHelper.DEFAULT_LOGGER_CLASS_APP_SETTING_NAME]
            = "TopCoder.LoggingWrapper.ELSImpl";
        // logger3 is an ELSImpl instance
        Logger logger3 = LogManager.CreateLogger();

        CM.AppSettings[TestHelper.DEFAULT_LOGGER_CLASS_APP_SETTING_NAME]
              = "TopCoder.LoggingWrapper.EnterpriseLibraryLogger";
        // logger4 is an EnterpriseLibraryLogger instance
              Logger logger4 = LogManager.CreateLogger();
// create the logger from the Default Namespace
// If the namespace is not specified, the namespace defaults to
// TopCoder.LoggingWrapper.LogManager.
Logger logger = LogManager.CreateLogger();

// It is also possible to create a logger from a non-file based
// configuration
IConfiguration config = new DefaultConfiguration("default");
config.SetSimpleAttribute("logger_class", "TopCoder.LoggingWrapper.DiagnosticImpl");
config.SetSimpleAttribute("logger_name", "LogTest");
config.SetSimpleAttribute("default_level", "INFO");
config.SetSimpleAttribute("source", "source of demo");
logger = LogManager.CreateLogger(config);
```

4.3.2 Log message with the Logger directly

```
// create the logger using default configuration
Logger logger = LogManager.CreateLogger();

// log simple message with default level
logger.Log("Hello World");

// log formatted message with default level
logger.Log("Hello {0}", "World");

// log formatted message with specified logging level
logger.Log(Level.WARN, "Hello {0}", "World");

// log a named message that has been defined in configuration
logger.LogNamedMessage("SimpleMessage", "p1", "p2");

// Even if we specify the wrong number of arguments, this log call
// will still work, because exception propagation is turned off
logger.LogNamedMessage("SimpleMessage", "p1");

// This call will not log anything at all because the logger has been
// configured to filter out the ERROR level.
logger.Log(Level.ERROR, "error message");
// On the other hand, a call at a non-filtered level will log the message
logger.Log(Level.INFO, "info message");

// If we has exception propagation turned on, we'll get a MessageFormattingException
// because it has not enough params are specified.
```

```
logger = LogManager.CreateLogger("TopCoder.LoggingWrapper.DiagnosticImpl");
try
{
      logger.Log("String to be formatted {0}, {1}, {2}", "only 1 arg");
}
catch (MessageFormattingException e)
{
      System.Console.WriteLine(e.ToString());
}
```

4.3.3 *Use zero-configuration option*

```
// The zero-configuration can be as simple as specifying the logging
// class and the zero-configuration type that should be used.

// The configuration varies for different loggers. For DiagnosticImpl logger,
// The source will be set to "TopCoder Logger" if it doesn't exist before.
Logger logger = LogManager.CreateLogger();
```

4.3.4 *Use ELS custom appender for log4net*

```
// The appender should be configured in the log4net config file like so
// <appender name="ELSAppender" type="TopCoder.LoggingWrapper.ELS.ELSAppender" >
// </appender>
// <logger name="Some Logger">
//   <appender-ref ref="ELSAppender" />
// </logger>

// load the config file
XmlConfigurator.Configure(new FileInfo(@"..\..\test_files\v3\log4net.config"));

// get the logger
log4net.ILog log = log4net.LogManager.GetLogger("Some Logger");

// log to ELS
log.Debug("Message to go to ELS");
```

4.3.5 *Use trace listener for Enterprise Library Logging Application Block*

```
// The listener should be configured in the app.config file like so
<configuration>
  <configSections>
    <section name="loggingConfiguration"
type="Microsoft.Practices.EnterpriseLibrary.Logging.Configuration.LoggingSettings,
Microsoft.Practices.EnterpriseLibrary.Logging, Version=3.1.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" />
    <section name="dataConfiguration"
type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings,
Microsoft.Practices.EnterpriseLibrary.Data, Version=3.1.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" />
  </configSections>
  <loggingConfiguration name="Logging Application Block" tracingEnabled="true"
    defaultCategory="General" logWarningsWhenNoCategoriesMatch="true">
    <listeners>
      <add loggerNamespace="TopCoder.LoggingWrapper.LogManager"
listenerDataType="Microsoft.Practices.EnterpriseLibrary.Logging.Configuration.CustomTrace
```

```
ListenerData, Microsoft.Practices.EnterpriseLibrary.Logging, Version=3.1.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
        traceOutputOptions="None"
type="TopCoder.LoggingWrapper.EntLib.LoggingWrapperTraceListener,
TopCoder.LoggingWrapper.Test, Version=3.0.0.0, Culture=neutral, PublicKeyToken=null"
        name="Logging Wrapper TraceListener" initializeData="" formatter="Text Formatter" />
    </listeners>
    <formatters>
      <add template="Timestamp: {timestamp}&#xD;&#xA;Message:
{message}&#xD;&#xA;Category: {category}&#xD;&#xA;Priority: {priority}&#xD;&#xA;EventId:
{eventid}&#xD;&#xA;Severity: {severity}&#xD;&#xA;Title:{title}&#xD;&#xA;Machine:
{machine}&#xD;&#xA;Application Domain: {appDomain}&#xD;&#xA;Process Id:
{processId}&#xD;&#xA;Process Name: {processName}&#xD;&#xA;Win32 Thread Id:
{win32ThreadId}&#xD;&#xA;Thread Name: {threadName}&#xD;&#xA;Extended Properties:
{dictionary({key} - {value}&#xD;&#xA;)}"
        type="Microsoft.Practices.EnterpriseLibrary.Logging.Formatters.TextFormatter,
Microsoft.Practices.EnterpriseLibrary.Logging, Version=3.1.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a"
        name="Text Formatter" />
    </formatters>
    <categorySources>
      <add switchValue="All" name="General">
        <listeners>
          <add name="Logging Wrapper TraceListener" />
        </listeners>
      </add>
    </categorySources>
    <specialSources>
      <allEvents switchValue="All" name="All Events" />
      <notProcessed switchValue="All" name="Unprocessed Category" />
      <errors switchValue="All" name="Logging Errors &amp; Warnings">
        <listeners>
          <add name="Formatted EventLog TraceListener" />
        </listeners>
      </errors>
    </specialSources>
  </loggingConfiguration>
</configuration>
// Then using any of the Enterprise Library Logging Application Block API
// will cause the messages to get to the TopCoder logger.  For example:
LogEntry entry = new LogEntry();

// set the entry
entry.Message = "This is my message";
entry.Severity = TraceEventType.Error;
entry.Priority = 5;

// write the entry
Microsoft.Practices.EnterpriseLibrary.Logging.Logger.Write(entry);
```

5. Future Enhancements

- Remove the obsolete methods.
- More Logger implementations.
- Support for timing blocks of code
- Support for categories (as in the Enterprise Library Logging Application Block)