Logging Wrapper 2.0 Component Specification

1. Design

The Logging Wrapper component provides a standard logging API with a pluggable back-end logging implementation. Utilization of the Logging Wrapper insures that components are not tied to a specific logging solution. More importantly, a change to the back-end logging solution does not require a code change to existing, tested components.

The design is based on a factory pattern. LogFactory is the factory that produces Log implementation instances. The Log interface abstracts a logging implementation (concrete instances are basic System.out/err logging, JDK 1.4 logging and Log4j logging). LogFactory can be used to get an instance of a logger. This class of the instance is configurable. This allows implementations to be swapped with no code modifications.

The changes to this design are presented below. Some changes were due to the new requirements and some were the result of a refactoring cycle, meant to fix some problems with the previous design. Note that despite of the drastic changes, compatibility is still preserved with old client code (the main goal was to support the LogFactory.getInstance().getLog(name) call).

One of the biggest changes of this design is the removal of the factory implementation classes. The factory classes were redundant and they only complicated the design with no benefit. Since the factory classes where build themselves using Java reflection, it made sense creating the Log instances directly, using Java reflection. The intermediation done by the factory class served to no purpose at all. Simply put, the flaw of the previous version was the fact that this class was a *factory of factories*, an obvious overkill. This removal implied also the removal of the abstract getLog method and of the abstract modified for this class.

Here is a summary of the changes:

- removed all LogFactory subclasses
- removed LogFactory.getLog abstract method and LogFactory abstract modifier
- removed createInstance (see the forum) and made getInstance deprecated
- added static getLog and getLog(name) methods with all exceptions silently caught
- added static getLogWithExceptions (same as getLog but with exceptions thrown, to allow the user to debug problems if needed)
- fixed some loadConfiguration method inconsistencies (see loadConfiguration)
- LogException uses the standard BaseException component
- Level uses the standard Typesafe Enum component
- added Level.hashCode (because whenever equals is overwritten, hashCode should be overwritten too)
- removed exceptions from the implementations of Log.log and Log.isEnabled, loadConfiguration and getAttribute
- changed visibility of the constants from BasicLog to private, because they do not need to be used by outside code
- enhanced the javadocs in all classes.

1.1 Design Patterns

- The Log implementations are adapters for the classes that do the actual logging
- LogFactory is a singleton.
- The LogFactory has factory methods that produce Log instances. It creates instances of the Log interface implementation based on a class name that is configurable. It is done using reflection. This way, the Log implementations can be swapped with no code modification.

1.2 Industry Standards

- None

1.3 Required Algorithms

1.3.1 Creating Log implementation instances using reflection

The following code can be used to create instances dinamically:

```
String logClass = null;
try {
  logClass =
     ConfigManager.getInstance().getString(NAMESPACE, LOG_CLASS);
} catch (ConfigManagerException e) {
  logClass =
   ConfigManager.getInstance().getString(NAMESPACE, LOG_FACTORY);
  if (logClass.endsWith("Factory")) {
    logClass = logClass.substring(0, logClass.length() - 7);
  }
}
Constructor constructor =
    Class.forName(logClass).getConstructor(new Class[]
     {String.class});
Log log = (Log) constructor.newInstance(new String[] {name});
return log;
```

1.3.2 Preserving the backwards compatibility with the previous versios of this component

There are some backwards compatibility provisions. The mail goal was to support the LogFactory.getInstance().getLog(name) construction (used to get a Log instance in the older versions). getInstance() is supported easily by returning a singleton instance of this class itself. There is no change to the signature. The getLog(name) method is conveniently supported by the new static method (it can be called on an instance too).

Another compatibility provision is changing LogException to be unchecked. LogFactory.getInstance/getLog do not throw any exceptions. However, old code expects LogException. In a try catch block, if the code block within try does not throw the caught exception, and that exception is checked, we have a compilation error. That's why the solution to the problem is to make it unchecked.

Another provision is the Configuration Manager property that specifies the LogFactory class (class present in versions up to 1.1) to be used for logging. This configuration property is currently deprecated and is mantained only for back compatibility purposes. The idea is to make this version support configuration files from the previous versions, too. Since logFactory was replaced with logClass, in order to support old configuration files, we have to deduce the LogClass name from the logFactory property. Fortunately, this is rather easy, because the factory classes had the name of the corresponding Log classes with the "Factory" suffix. Obviously, this assumption can fail for some custom old

Log/LogFactory implementations, so it is a best effort situation, made to ensure backwards compatibility in the vast majority of the cases.

1.4 Component Class Overview

com.topcoder.util.log.LogFactory

The Logging Wrapper component provides a standard logging API with a plugable back-end logging implementation. Utilization of the Logging Wrapper insures that components are not tied to a specific logging solution. More importantly, a change to the back-end logging solution does not require a code change to existing, tested components. Support exists for log4j and java 1.4 Logger as back-end logging implementations.

This class is factory of Log instances. Client code uses this class to retrieve Log instances that can be used afterwards for the actual logging. The basic usage of this class is LogFactory.getLog(name). Other methods are. present as well, such as getLog() (uses DEFAULT_NAME as name), getInstance() (returns an instance of this class - necessary for backwards compatibility), getLogWithExceptions(name) (creates a Log instance and throws exception in case an error occurs - opposed to the getLog methods who silently catch any exception).

The main idea of this factory class is to create actual Log instances based on a configuration file. By simply changing the configuration file, different implementations will be created by this class. This allows the easy swapping of different logging implementation with no code modifications.

This class is a singleton but only for backwards compatibility reasons (see the getInstance method). In fact, this class leans more towards an utility class (because all methods are static as the new requirements call for, no direct instatiation can be done), and would be one, if it weren't for the getInstance() method that breaks the utility class pattern. However, future versions will make this class most likely a true utility class.

One of the biggest changes of this design is the removal of the factory implementation classes. The factory classes were redundant and they only complicated the design with no benefit. Since the factory classes where build themselves using Java reflection, it made sense creating the Log instances directly, using Java reflection. The intermediation done by the factory class served to no purpose at all. Simply put, the flaw of the previous version was the fact that this class was a factory of factories, an obvious overkill. This removal implied also the removal of the abstract getLog method and of the abstract modified for this class.

There are some backwards compatibility provisions also. The mail goal was to support the LogFactory.getInstance().getLog(name) construction (used to get a Log instance in the older versions). getInstance() is supported easily by returning a singleton instance of this class itself. There is no change to the signature. The getLog(name) method is conveniently supported by the new static method (it can be called on an instance too).

com.topcoder.util.log.Log

The Log interface should be extended by classes that wish to provide a custom logging implementation. The <tt>log</tt> method is used to log a message using the underlying implementation, and the <tt>isEnabled</tt> method is used to determine if a specific logging level is currently being logged.

This class is not meant to be instantiated directly. The LogFactory class should be used to create instances. The main idea of this factory class is to create actual Log instances based on a configuration file. By simply changing the configuration file, different implementations will be created by this class. This allows the easy swapping of different logging implementation with no code modifications.

Note: All implementations of this class must implement a public constructor that accepts a String as parameter (the name for the instance of the Log implementation that is to be created).

com.topcoder.util.log.Level

The Level class maintains the list of acceptable logging levels. It provides the user this easy access to predefined logging levels thought the constants defined in this class.

Extends the Enum class from the Typesafe Enumeration component.

com.topcoder.util.log.basic.BasicLog

This is the basic implementation of the Log interface.

com.topcoder.util.log.jdk14.Jdk14Log

This is the JDK 1.4 specific implementation of the Log interface.

com.topcoder.util.log.log4j.Log4jLog

This is the Log4J specific implementation of the Log interface.

1.5 Component Exception Definitions

LogException:

Exception class for all logging exceptions thrown from this API. It provides the ability to reference the underlying exception via the <tt>getCause</tt> method, inherited from BaseException. It is essentially a wrapper for other exceptions. It is used in LogFactory.getLogWithExceptions/getInstance methods.

Except for this case, all exception all caught silently and never thrown to the client code.

1.6 Component Benchmark and Stress Tests

Performance is important for this component because logging is present everywhere in a non-trivial application. There should be benchmarks for logging a huge quantity of data.

This component is very likely to be used in a concurrent environment to thread safe is essential. That's why some concurrency testing should be performed. Different threads will attempt to log simultaneously. Both the correctness of the logging and the speed should be taken into account.

The following benchmarks should be performed:

- Log 1,000 messages using the basic logger. Since the complexity of logging one message is O(1) this should not take long. However, the actual outputting of text may be the bottleneck here.
- Log 10,000 messages using the JDK 1.4 logger, to a file. Since the complexity of logging one message is O(1) this should not take long, especially since file logging is reasonably fast (compared to screen logging).
- Log 10,000 messages using the Log4j logger, to a file. Since the complexity of logging one message is O(1) this should not take long, especially since file logging is reasonably fast (compared to screen logging).

- Repeat the above tests in a concurrent environment with at least 10 threads. The above quantities should be decreased appropriately (for each thread the task should be smaller). There shouldn't be a significant performance difference.

2. Environment Requirements

2.1 Environment

- At minimum, Java1.4 is required for compilation and executing test cases.
- Java 1.2 or higher can be used for Basic and Log4j logging implementations.
- Java 1.4 or higher must be used for Java 1.4 built in logging (Jdk14Log class).

2.2 TopCoder Software Components

- Base Exception 1.0 (provides the base for the LogException in a uniform manner across JDK 1.4 and previous JDK versions)
- Type Safe Enumeration 1.0 (the Level class was a type safe enumeration before, with some minor problems, especially at serialization; using this component fixes the problem and makes the enumeration handling consistent across the component catalog)
- Configuration Manager 1.2

NOTE: The default location for this file is../lib/tcs/configmanager/1.2 relative to this component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

- Xerces 1.4.4 (indirectly, by the Configuration Manager): download
- Log4j-1.2.6 or higher (only in the Log4jLog class): download

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Names

- com.topcoder.util.log
- com.topcoder.util.log.basic
- com.topcoder.util.log.jdk14
- com.topcoder.util.log.log4j

3.2 Configuration Parameters

Parameter	Description	Values
com.topcoder.util.log	Property specified in the configuration manager properties file to point to the logging configuration file.	The relative path to the logging XML file containing the logging configuration information. For example: com/topcoder/util/log/Logging.xml
factoryClass	Deprecated	
logClass	Property specified in the logging configuration file to indicate which logging implementation to use.	Currently one of: com.topcoder.util.log.jdk14.Jdk14Log com.topcoder.util.log.log4j.Log4jLog com.topcoder.util.log.basic.BasicLog
config.file	Optional property used in the logging configuration file to specify an additional file containing log4j specific properties	The relative path to a properties file containing log4j logging configuration data. For example: log4j.properties
basic.log.target	Optional property used with the basic logger to indicate where the logged messages will be written.	Must be one of: System.out System.err If neither is specified, System.err is assumed.

3.3 Dependencies Configuration

- JDK 1.2 or above
  - If logging is to be supplied by the JVM, then JDK 1.4 or above must be used; otherwise, only JDK 1.2 is required.
- Logging configuration
  - If jdk1.4 logging is used, the logging configuration must be specified according to the JRE requirements. By default, the logging.properties file located in the lib directory of the JRE is used.
  - If log4j logging is used, the logging configuration must be specified according to log4j requirements. The config.file configuration parameter can be used to help specify a configuration file.
- Log4j jar file

  The build script uses Log4j-1.2.8. If you use a different version of Log4j either:
  - Modify the log4j.jar property in the build.xml to point to the version you are using.
    OR
  - Add the following to the topcoder_global.properties to override all references to log4j in TopCoder Software components.
    Log4j.jar=*PATH*
    Where PATH is the location and name of the log4j jar on your system.

4. Usage Notes

It should be noted that the logging configuration is loaded the first time LogFactory.getInstance/getLog() is called. But the configuration can be changed at runtime and the logging wrapper will respond accordingly once the loadConfiguration method is invoked.

There is no way to specify a logging level within the logging wrapper itself. All level configuration must be done in an implementation –specific way. Consult the specific documentation for the logging implementation you are using for details.

If the basic logger is used, there is no concept of level. All logging messages are written to the log regardless of level. Therefore, the isEnabled() method will always return true for the basic logger.

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.
    - Executing 'ant test' will execute tests for all logging implementations. The tests will fail if each implementation is not properly configured.
    - To remove tests for certain logging implementations:
        1. Open the build.xml file.
        2. Locate the "test" target.
        3. Comment out the tests that should not be executed. To comment a section, use <!-- -->

    NOTE: The Logging Wrapper component requires Java1.4 to compile and execute test cases.
- Make sure that the specific logging implementation is logging at the appropriate level for the tests and that the logging output file (if necessary) is located in the required directory for the tests.

The stress tests are covered in their specific section. The failure tests should be created using the invalid and valid arguments present in the Poseidon documentation. They provide a full coverage of the failure situations that need to be tested.

The accuracy tests should cover the following areas:

- The configuration logic should be checked (see if the instances actually produced by the factory class is indeed what the configuration file is specifying)
- The basic logging should be tested to see if output is actually generated.
- The JDK 1.4 logging should be tested, especially the level conversion. There should be tests to verify if the levels work as they should.
- The Log4j logging should be tested, especially the level conversion. There should be tests to verify if the levels work as they should.
- A compatibility test should be created. This test would do the logging as in the previous versions. Its purpose is to ensure backward compatibility now and in the future versions.

4.2 Required steps to use the component

- Place the logging-1.2.jar in your classpath.
- Import the appropriate classes from the com.topcoder.util.log package and appropriate subpackages.
- Get an instance of the log by name. It can be done in three ways:

- ```
  try {
     Log log = LogFactory.getInstance().getLog(
                   "some name");
  } catch (LogException e) {
     ...
  }

  (this is the backwards compatibility way)
  ```

- ```
  try {
     Log log = LogFactory.getLogWithExceptions(
                   "some name");
  } catch (LogException e) {
     ...
  }

  (this is the new way, with exceptions)
  ```

- ```
  Log log = LogFactory.getLog("some name");

  (this is the new way, with no exceptions thrown)
  ```

- Write messages to the log based on level restrictions:
  - `log.log(Level.INFO, "informational message");`
- Or in one step:
  - ```
    LogFactory.getLog("some name").log(
              Level.INFO, "informational message");
    ```

4.3 Demo

The code above shows all that is needed to use the component.

5. Future Enhancements

Maybe some formatter methods could be added to help the user create parametric messages. These methods would get a format string containing placeholders, and an array of objects. The final string would be obtained by replacing the placeholders.