Logging Wrapper 1.3 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.

Version 1.2 Changes

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.

Version 1.3 Changes

All changes to this document (beyond simple formatting and minor spelling corrections) will be marked in blue.

This enhancement to the Logging Wrapper has the following goals:

1. Backwards compatibility. Backwards compatibility was achieved by keeping the same API signatures and functionality. The functionality provided by this change was either completely new or extended existing functionality.
2. A new composite logger that allows dynamic changes to the underlying loggers. The composite logger will log the given message to all loggers within the composite. The logger can be initialized from a set of loggers or dynamically changed programmatically.
3. A database logger that will log messages to a database table. This logger implementation uses the database connection factory to acquire a connection and will use a user specified insert statement to insert logging messages. This allows the user to create custom logging tables and customize the actual insert being done.

In addition to those changes, the following changes occurred:

1. An asynchronous logger implementation was provided. This logger will return immediately and provide logging in different thread using the Synchronous Processing Wrapper component. This logger was provided for any logging implementation that has high latency – such as the database logger.
2. A filtered logger implementation was provided. This logger will determine if the message matches a provided regex pattern before the underlying logger will be called. This allows the composite logger to act as a centralized logger that can filter logging messages to specific loggers.
3. Enhanced the property retrieval from the configuration manager. The property retrieval was enhanced to first look for the logger name + property name before looking for just the property name. This allows custom configuration setup for specific logging names.

Summary of Changes to existing classes:

LogFactory

- Changed getAttribute(string) to use a new argument name.
- Added a new method getAttribute(string,string)
- Change the getLogWithException(string) to call the new getLogWithException(string,string) method
- Added a new method getLogWithException(string,string)

Level

- Add a new method parseLevel(string)

BasicLog

- Changed to inherit from AbstractLog for storage of the name
- Changed log message to call getAttribute(getName(), SYSTEM_ERR)

Log4jLog

- Changed to inherit from AbstractLog for storage of the name
- Change constructor to call getAttribute(name, CONFIG_FILE)

Jdk14Log

- Changed to inherit from AbstractLog for storage of the name
- Change constructor to call getAttribute(name, CONFIG_FILE)

Summary of new classes:

- AbstractLog
- DatabaseLog
- FilteredLog
- CompositeLog
- AsynchronousLog

Summary of other changes:

- Updated all diagrams and documents to reflect standard styling.

Synchronous Processing Wrapper Update

This design relies on an update to the Synchronous Processing Wrapper component that will allow it to process the messages using a fixed set of threads from a thread pool. This update was approved and specified in the Event Engine 2.0 component.

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.
- The decorator pattern has been used in to add asynchronous and filtered logging to other logging implementations
- The composite pattern was used to allow a group of log implementations to be used as a single unit.

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 dynamically:

```
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);
  }
}
return getLog(name, logClass);
```

The getLog(name, logClass) will then:

```
Class clazz = Class.forName(logClass);
Constructor ctor = clazz.getConstructor(new Class[] {String.class});
Return (Log) ctor.newInstance(new String[] { name });
```

1.3.2 *Preserving the backwards compatibility with the previous versions 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 maintained 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.

1.3.3 *Composite Logger*

The composite logger allows a group of loggers to act as a single logger. The composite logger implements the List interface to allow the application to work with the composite in an easy manner. The composite logger and inherits from AbstractList to provide most of the List implementation and implements only those methods specified by the AbstractList to implement a mutable list. The add and set methods will ensure that the objects are of Log type – ensuring that all objects within the logger are the proper type.

isEnabled(level)

The Log contract specifies that true should be returned if the level is enabled. Since the composite is a group of loggers, we will interpret this to mean that if any internal Log returns true – a true should be returned. In other words:

```
for(Iterator itr = loggers.iterator(); itr.hasNext();) {
   Log log = (Log) itr.next();
   if (log.isEnabled(level)) return true;
```

```
        }
        return false;
```

log(level, message)
The composite logger will log the message to each internal log that is enabled for the level:

```
        for(Iterator itr = loggers.iterator(); itr.hasNext();) {
           Log log = (Log) itr.next();
           if (log.isEnabled(level)) {
              log.log(level,message);
           }
        }
        return false;
```

add(index,element)/set(index,element)
The only algorithm that needs to be addressed in this method is to ensure that the logger being added cannot be tracked back to 'this' (to avoid infinite loops in the log method). Since logging will never be very complex, a simple brute force search is acceptable. This can be:

```
   validate(Log element) throws IllegalArgumentException {
       if (element==this) throw new IllegalArgumentException();
       if (element instanceof CompositeLog) {
          CompositeLog log = (CompositeLog) element;
          for(Iterator itr = log.iterator();itr.hasNext();) {
             validate((Log) itr.next());
          }
       }
   }
```

1.3.4 Database Logger

The database logger will use one connection per instance and will use a repeating timer task to close/null the insert statement if a certain amount of inactivity has passed. If the insert statement is null, the statement will automatically be recreated when it's next needed. The basic algorithm for logging is:

```
        boolean useTimestamp = Boolean.parseBoolean(
            LogFactory.getAttribute(getName(),USE_TIMESTAMP));

        boolean useString = Boolean.parseBoolean(
            LogFactory.getAttribute(getName(),USE_STRING_LEVEL));

        long now;
        synchronized (this) {
           lastLogTime = System.currentTimeMillis();
           now = lastLogTime
        }

        String statement =
            LogFactory.getAttribute(getName(),INSERT_STATEMENT));

        PreparedStatement ins;
        synchronized (this) {
           if (connection == null
           || connection.isClosed()
           || connection.isReadyOnly()) {
              ... get the db connection namespace
              ... and name from the logFactory.getAttribute

              DBConnectionFactory dcf =
                        new DBConnectionFactoryImpl(namespace)

              Connection conn = dcf.createConnection(name);
              ins = conn.prepareStatement(statement);
           }
           else {
              ins = conn.prepareStatement(statement);
```

```
    }
}


int colIdx = 0;

if (useTimestamp) {
    ins.setTimestamp(colIdx++, new Timestamp(now));
}

if(useString) {
    ins.setString(colIdx++,level.toString())
} else {
    ins.setInt(colIdx++,level.intValue())
}
ins.setObject(colIdx++,message);
ins.executeUpdate();
```

The database log implementation will work with any insert statement where the level and message columns are parameterized. This allows the maximum flexibility in allowing the application to design the logging table as they see fit (including column types), provides the flexibility to handle various database systems and to provide static, default values or SQL function values to any other columns defined (such as logging the time of insert).

1.4 Component Class Overview

com.topcoder.util.log.LogFactory

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. 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 instantiation 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.loggers.AbstractLog

This is an abstract implementation of the Log interface that can provide common services to super class log implementations. This abstract base, currently, only provides services to store and retrieve the name assigned to the logger

com.topcoder.util.log.loggers.AsynchronousLog

This implementation of the Log contract will provide asynchronous logging capabilities around another log implementation. This logging implementation could be used when the underlying logging implementation has high latency and the logging (and timing) is not critical. This class does not make any guarantees

as to when the logging occurs or the order the log messages will be processed (they may be processed out of order). These limitations are based on the Synchronous Wrapper component processing and may be true as that component is later enhanced.

com.topcoder.util.log.loggers.FilteredLog

This implementation of the Log contract will provide logging capabilities around another log implementation only if the message matches a specified regex pattern. This logging implementation could be used where the message format use some application defined pattern and should only be logged when that some pattern is met. This is especially useful when used in conjunction with a CompositeLog (allowing centralized application logging to be log to different logs based on various patterns).

com.topcoder.util.log.loggers.DatabaseLog

This implementation of the Log contract will provide logging to a database table. A connection will be made to the database specified by the connection name in the LogFactory configuration file and defined by the DBConnectionFactory. An insert statement (defined also in the LogFactory configuration file) will then be run for each log message received.

com.topcoder.util.log.loggers.CompositeLog

This implementation of the Log contract will provide logging capabilities over many other logging implementations. This logging implementation could be used when logging should be done to one or more loggers. The composite logger can also be used as a centralized logger that will distribute logging to the appropriate logger if used in conjunction with the filtered log implementation. This class implements the List interface that can only accept Log instances.

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 Thread Safety

The log implementations added in Version 1.3 are completely thread safe. Each implementation takes the following approaches:

- AbstractLog is immutable
- FilteredLog is immutable
- AsynchronousLog is immutable
- CompositeLog will synchronize on loggers on access or modification to protect concurrent modification of the internal loggers
- DatabaseLog has the log method synchronized to protect concurrent modification of the insert statement and will synchronize on levelsEnabled on access or modification to protect concurrent modification of the levels that are enabled.

2. Environment Requirements

2.1 Environment

- At minimum, Java1.4 is required for compilation and executing test cases.
- 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 2.1.4
- Synchronous Processing Wrapper X.X was utilized to allow logging to asynchronously occur in the AsynchronousLog implementation. This component will need to wait on a modification to this component to provide a fixed number of threads for processing (this was approved and specified in the Event Engine 2.0 project).
- DB Connection Factory 1.0 was utilized to create connections for the DatabaseLog implementation.

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

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
- com.topcoder.util.loggers

3.2 Configuration Parameters

Parameters valid for the LogFactory

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.	The fully qualified classname of a Log implementation

*Parameters valid for the BasicLog implementation***

Parameter	Description	Values
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.

*Parameters valid for the Jdk14Log/Log4jLog implementations***

Parameter	Description	Values
config.file	Optional property used in the logging configuration file to specify an additional file containing the logging specific properties	The relative path to a properties file containing the logging configuration data. For example: log4j.properties

*Parameters valid for the FilteredLog implementation***

Parameter	Description	Values
filtered.log.class	The classname for the Log implementation that will be called if the message matches the specified regex pattern.	The fully qualified classname of a Log implementation
filtered.log.regex	The regex pattern that must be matched to call the underlying Log	A regex pattern or none to specify all messages.

*Parameters valid for the AsynchronousLog implementation***

Parameter	Description	Values

asynchronous.log.class	The classname for the Log implementation that will be called asynchronously.	The fully qualified classname of a Log implementation

*Parameters valid for the CompositeLog implementation***

Parameter	Description	Values
composite.log.classes	The Log implementation to initialize the composite with in the form of: name,classname;name,classname;etc.	A non-null, non-empty string

*Parameters valid for the DatabaseLog implementation***

Parameter	Description	Values
database.log.namespace	The configuration namespace to use for the DB Connection Factory component	A non-null, non-empty string
database.log.connection.name	The connection name to use (that is managed by the DB Connection Factory Component)	A non-null, non-empty string
database.log.insert.statement	The insert statement that will be run. The first parameter will the level and the second parameter will be the message	A non-null, non-empty string
database.log.use.string.level	Whether to use the string version of the level (true) or the integer version when inserting	A boolean
database.log.levels.enabled	The logging levels to enable	A comma separated list of levels – either their integer or string version (see Level.parseLevel)
database.log.use.timestamp	Whether to create a timestamp or not	A boolean
database.log.close.time.interval	The inactivity interval of time (in milliseconds) that must pass before the connection is closed	A long

** all parameters for this implementation can be customized for a specific log name. For example, if you had a configuration file like:

```
LogClass = com.topcoder.util.log.basic.BasicLog
basic.log.target = System.err
acme.basic.log.target = System.out
```

All logging will then appear in System.err except for the logs that are named “acme” - those would go to System.out:

LogFactory.getLog().log(...) - writes to System.err
LogFactory.getLog(getClass().getName()).log(...) - writes to System.err
LogFactory.getLog("acme").log(...) - writes to System.out

The developer should note that when "use timestamp" is specified, the time will be generated on the local PC using the local system clock and time zone (this may not prove to be accurate). The better solution would be to use database side timestamps instead by adding a "CURRENT TIMESTAMP" (for DB2 – use the equivalent for the database in question) to the insert statement itself and set this flag to false.

3.3 Dependencies Configuration

- 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.

- Please review the documentation for the Synchronous Processing Wrapper and DB Connection Factory for specifics on their configuration needs.

- Database table and insert statement
  The DatabaseLog implementation can work with any database table that contains at least the following:
  - A level column – either number or string based.
  - A message column – depends on the messages being sent but will typically be some string based column

  The insert statement (see configuration above) can then be specified where the level column will be the first parameter in the insert statement and the message will be the second parameter.

  The following is an example table and insert statement that can be:

```
CREATE TABLE logging
  (
   log_time      TIMESTAMP not null
  ,log_level     INTEGER   not null
  ,message       VARCHAR(254) NOT NULL
  ,PRIMARY KEY (log_time)
 )

INSERT INTO logging
   VALUES (?, ?, ?);
```

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.

Specification of logging level is dependent upon the log implementation. The only logger that supports direct log level enabling is the DatabaseLog implementation. All other implementations will depend on something external to the component – please see the specific document 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 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 sub packages.
- 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

4.3.1 Filtered Logging

The filtered logging provides a wrapper around another logger and will only call that logger if the toString() of the message matches a specified regex pattern.

To use the filtered logging to the console, you can setup the configuration file like:

```
LogClass = com.topcoder.util.log.loggers.FilteredLog
filtered.log.class = com.topcoder.util.log.basic.BasicLog
filtered.log.regex = Exception.*
basic.log.target = System.out
```

The following would log messages to the console:

```
LogFactory.getLog().log(Level.ERROR, “Exception occurred in this message”);
LogFactory.getLog(“stuff”).log(Level.INFO, “Exceptional logging”);
```

The following would not log any messages to the console:

```
LogFactory.getLog().log(Level.INFO, “This is a log message”);
```
```

LogFactory.getLog("stuff").log(Level.WARN, "warning – no logging here");

4.3.2 Database Logging

The database logging will log it's messages to some database table. This demo assumes that a table similar to that specified in section 3.3 and that the db connection factory has been setup to connect to it (both the namespace and connection names listed below can be anything and the ones specified are strictly for example).

To use the database logging, you can setup the configuration file like:

```
LogClass = com.topcoder.util.log.loggers.DatabaseLog
database.log.namespace = connection.factory
database.log.connection.name = tcsinformix
database.log.insert.statement = << statement from section 3.3 >>
database.log.use.string.level = false
database.log.levels.enabled = "info,warn,error"
database.log.use.timestamp = true
```

The following would log messages:

```
LogFactory.getLog().log(Level.ERROR, "Exception occurred in this message");
LogFactory.getLog("stuff").log(Level.INFO, "Exceptional logging");
```

The following would not log any messages:

```
LogFactory.getLog().log(Level.DEBUG, "This is a log message");
LogFactory.getLog("stuff").log(Level.FATAL, "fatal message");
```

After the above logging – the following would appear in the table:

log_time	**log_level**	**Message**
2006-07-04 12:15:12.293489	500	Exception occurred in this message
2006-07-04 12:15:12.883483	600	Exceptional logging

4.3.3 Asynchronous Logging

The asynchronous logging will log messages asynchronously – allowing the application to continue working while the log message is processed. This is especially useful for loggers that have a high latency in their logging. This demo will asynchronously call the database logger that was specified in section 4.3.1.

To use the asynchronous logging, you can setup the configuration file like:

```
LogClass = com.topcoder.util.log.loggers.AsynchronousLog
asynchronous.log.class = com.topcoder.util.log.loggers.DatabaseLog
database.log.namespace = connection.factory
database.log.connection.name = tcsinformix
database.log.insert.statement = << statement from section 3.3 >>
database.log.use.string.level = false
database.log.levels.enabled = "info,warn,error"
```

Running the same demo as in section 4.3.1 will result in the same output (as shown in 4.3.1).

4.3.4 Composite Logging

The composite logging will allow the logging to occur over multiple loggers. This demo will create a logger that will log messages in the same was as section 4.3.1 and section

4.3.2 for each message.

To use the composite logging, you can setup the configuration file like:

```
LogClass = com.topcoder.util.log.loggers.CompositeLog
composite.log.classes = com.topcoder.util.log.loggers.FilteredLog;
                        com.topcoder.util.log.loggers.FilteredLog,error;
                        com.topcoder.util.log.loggers.DatabaseLog
filtered.log.class = com.topcoder.util.log.basic.BasicLog
filtered.log.regex = Exception.*
basic.log.target = System.out
error.filtered.log.class = com.topcoder.util.log.basic.BasicLog
error.filtered.log.regex = Error.*
error.log.target = System.err
database.log.namespace = connection.factory
database.log.connection.name = tcsinformix
database.log.insert.statement = << statement from section 3.3 >>
database.log.use.string.level = false
database.log.levels.enabled = “info,warn,error”
```

This configuration also demonstrates the ability to use specialized configuration based on the name of the logger. The first filtered log has no name assigned to it and will default to the normal configuration options. The second filtered log will be assigned the 'error' name and will use the “error.xxx” configuration options instead.

Running the following logs would result in the log messages appearing on the console and being added to the database table. The first two log messages will appear in the System.out stream, the last one will appear in the System.err stream:

```
LogFactory.getLog().log(Level.ERROR, “Exception occurred in this message”);
LogFactory.getLog(“stuff”).log(Level.WARN, “Exceptional logging”);
LogFactory.getLog().log(Level.ERROR, “Error – something happened”);
```

Running the following logs would result in the log messages only appearing in the database table (the message does not match the regex pattern):

```
LogFactory.getLog().log(Level.ERROR, “NullPointer occurred”);
```

Running the following logs would not be logged at all (the either don't match the filter or are disabled levels for the database):

```
LogFactory.getLog().log(Level.DEBUG, “This is a log message”);
LogFactory.getLog(“stuff”).log(Level.FATAL, “fatal message”);
```

Dynamically changing the composite logger is also possible using standard List operations:

```
CompositeLog logger = (CompositeLog) LogFactory.getLog();
logger.add(new BasicLog(“acme”));
logger.remove(0);
logger.clear();  // etc
```

4.3.5 Specialized configuration for the logger name.

Logging can be specialized by it's name. When the logger is created, it will first check for the configuration options prepended with the name (and a period).

With the following configuration file, we setup the all loggers to use the System.out stream. However, if the logger was called “acme” or “widget” - System.err will be used instead:

```
LogClass = com.topcoder.util.log.loggers.BasicLog
```

```
basic.log.target = System.out
acme.basic.log.target = System.err
widget.log.target = System.err
```

The following logging will go to System.out:

```
LogFactory.getLog().log(Level.INFO, “Exception occurred in this message”);
LogFactory.getLog(“stuff”).log(Level.WARN, “Exceptional logging”);
```

The following logging will go to System.err:

```
LogFactory.getLog(“acme”).log(Level.INFO, “Portable hole was created”);
LogFactory.getLog(“widget”).log(Level.WARN, “Widget A was incomplete”);
```

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.

The synchronous processing wrapper should be updated to support a configurable thread pool to limit the number of threads that are utilized by that component. Please note that an RFQ has been entered in the customer forum for that component.