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. The LogManager will be used by the application to obtain Log instances from the underlying LogFactory. The Log interface abstracts a logging implementation (concrete instances are basic print stream logging, JDK 1.4 logging and Log4j logging) and provides an API to delay the formatting of the logging message as long as possible (sometimes even into the underlying logging system). The LogManager can be used to get an instance of a logger for a given logger name. The LogManager can be configured to specify the type of LogFactory to use and to specify the various Object Formatters to format a message object (such as an event or transaction). By utilizing Object Formatters, the application can separate the logging format of an object from the object itself (and/or centralize the formatting of objects into a single location rather than at each logging line).

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 Type Safe Enumeration 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 2.0 Changes

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

Please note that the project file (.zuml) will mark added items in blue and updated items in red. The only exceptions to this are:

1. All the design documentation tabs (class, variables, and methods) were updated to reflect the v1.2 distribution code – if the method logic itself was unchanged, this type of update would not be marked in red.
2. Converting the project file to Poseidon 5.0 destroyed the existing sequence diagrams and marking the differences would be too difficult. Since most of these diagrams were fully updated anyway – no markings were made.

This enhancement to the Logging Wrapper has the following goals:

1. The configuration manager was eliminated from this component. All references to it were removed and all corresponding methods were updated and associated fields dropped.
2. Formatting of the message should be put off as much as possible. This goal was satisfied by providing functions that allows the application to specify the formatting and have that formatting delayed as much as possible. In both the log4j and java logging instances, specific situations were even pushed past this component and into the native processing of those logging systems.
3. Stack trace support was added to the component to allow any throwable to be specified for logging. In the case of log4j/jdk14 logging, the throwable is simply passed to the appropriate method. In the case of the basic logger, the throwable will print the stack trace out to the PrintStream.

In addition to those changes, the following changes occurred:

1. Full support for message **object** formatting was introduced by utilizing the Object Formatter component (PM required approval of all components in the forum - see PM approval message in docs directory). This provides separation between a message object and its formatting logic. The application simply logs with the message and the string format of the object is then controlled via the Object Formatter (also allowing for a centralized area to update the formatting).
2. The basic logger was enhanced to work with any PrintStream (not just System.out/System.err). This would allow the BasicLogger to also write the information out to files or sockets.
3. The .zuml file was fully updated to current standards/colors and the documentation changed to reflect the v1.2 development.
4. Added more constructors to the LogException to allow it to be used by other

LogFactory types.

5. Added a convenience method to parse a level from a string. This enables an application to easily store and recreate a level by either the string form or the integer form.
6. An error in the mapping of the log4j levels was corrected.

Summary of Changes to existing classes:

LogManager (formerly known as LogFactory)

- Name was changed to LogManager.
- Removed Configuration Manager fields, variables and references.
- Support for the new LogFactory and Object Formatter were added.
- Made the BasicLogFactory the default factory for the LogManager if none was specified.
- Changed the logger DEFAULT_NAME to be public.

Level

- Add a new convenience method parseLevel(string).
- Made the level integer final

LogException

- Added two new constructors.

BasicLog

- Removed Configuration Manager fields, variables and references.
- Changed to inherit from AbstractLog for storage of the name.
- Changed to print to a generic PrintStream that is specified in the constructor.
- Changed to print the stack trace after the message.

Log4jLog

- Removed Configuration Manager fields, variables and references.
- Fixed an error in the level mapping.
- Changed to inherit from AbstractLog for storage of the name
- Dropped the current level variable and processing (was not needed).
- Changed the constructor to package private since it should only be created by the associated factory now.
- Changed its logging method to implement the new signature defined by the AbstractLog
- Changed the logging method to call the appropriate underlying logging method depending if a throwable was specified or not.
- Overrode some of the AbstractLog methods to allow message object formatting to be processed internally by a Log4J Layout if the application specified one (delaying the formatting further than this component).
- Made the logger variable final.

Jdk14Log

- Removed Configuration Manager fields, variables and references.
- Changed to inherit from AbstractLog for storage of the name
- Dropped the current level variable and processing (was not needed).
- Changed the constructor to package private since it should only be created by the associated factory now.
- Changed its logging method to implement the new signature defined by the AbstractLog

- Changed the logging method to call the appropriate underlying logging method depending if a throwable was specified or not.
- Overrode some of the AbstractLog methods to allow message string formatting to be processed internally by a java logging API (delaying the formatting further than this component).
- Made the logger variable final.

Summary of new classes:

- AbstractLog
- LogFactory
- BasicLogFactory
- Jdk14LogFactory
- Log4jLogFactory

1.1 Design Patterns

- The Log implementations are adapters for the classes that do the actual logging
- The factory pattern is used by each LogFactory to produce Log implementations specific to that factory implementation.
- The strategy pattern was used to allow interchangeability of the various LogFactory implementations and their associated Log implementation.
- The template pattern was used to in AbstractLog to allow subclasses to override specific methods without affecting the overall structure

1.2 Industry Standards

- None

1.3 Required Algorithms

1.3.1 Message String Formatting

This component can format a message format string pattern with a list of arguments. To format the list, the AbstractLog will simply:

```
// Format the string pattern with the arguments
String formattedMessage = MessageFormat.format(messageFormat,args);
```

The formatted message is then passed to the underlying logger's log method.

Please note that the Jdk14 API provides an API to do this internally and the Jdk14Log implementation will override this functionality to simply pass the message format pattern and its arguments directly to the Jdk14 Logger (thus delaying the formatting even further). Note: this only occurs on the non-Throwable version of this API (the Jdk14 Logger API only provides a non-Throwable version of this).

1.3.2 Message Object Formatting

This component can also format a message object utilizing the Object Formatter

component. This allows the application to separate the formatting of the object from the object itself and from the logging locations. When the AbstractLog needs to format an object, it will simply:

```
// Get the Object Formatter (if not specified via the API)
ObjectFormatter of = LogManager.getObjectFormatter();

// Format the object
String formattedMessage = of.format(message);

// As a backup – if it wasn't formatted, default to the toString()
If (formattedMessage == null) formattedMessage = message.toString();
```

The formatted message is then passed to the underlying logger's log method.

Please note that the Log4J API provides the ability to format message objects via a Layout implementation. This component provides the API to turn off the above logic in the Log4J factory if the application will be specifying object formatting via the Layout itself. This would allow the component to delay the formatting of the message object even further.

1.4 Component Class Overview

com.topcoder.util.log.LogManager

This is the main class for the Logging Wrapper component. 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 the console, log4j and java1.4 Logger as back-end logging implementations. This class will default to the console logger unless a new one is specified. Additionally, logging a message object can be generically formatted via the Object Formatter component prior to logging.

com.topcoder.util.log.LogFactory

This interface defines the contract for implementations that will produce Log instances. The LogFactory will implement the createLog method that takes the name to assign to the Log and return a Log instance for it.

com.topcoder.util.log.Log

The Log interface should be extended by classes that wish to provide a custom logging implementation. The various log method(s) are used to log a message using the underlying implementation, and the isEnabled method is used to determine if a specific logging level is currently being logged. This class has various overridden methods to allow flexible logging. All log methods will attempt to delay the formatting of the logging message to the latest possible moment (which may be in the underlying logging mechanism if supported). At the highest level, the message will not be formatted if the logging level is not enabled. Beyond that level, it's entirely dependent upon the underlying logger. Example: the java logger can delay message format processing until the last possible moment natively and the log4j logger could delay message object formatting until the last possible moment.

com.topcoder.util.log.AbstractLog

This is an abstract implementation of the Log interface that can provide common services to Log implementations. This abstract base, currently, provides services to store and retrieve the name assigned to the logger and provides default implementations to the various log methods in the Log interface. This abstract base will be responsible for converting the message into a string form and then calling an abstract method (that the subclass will provide) to log the message. Please note that subclasses may override any of these methods to provide specific functionality from the underlying implementation.

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 though the constants defined in this class.

com.topcoder.util.log.basic.BasicLogFactory

This is the implementation of the LogFactory interface that will create BasicLog instances based on the print stream given.

com.topcoder.util.log.basic.BasicLog

This is the basic implementation of the Log interface that will write the logging message to the specified print stream.

com.topcoder.util.log.jdk14.Jdk14LogFactory

This is the implementation of the LogFactory interface that will create Jdk14Log instances based on the java logger for the given name.

com.topcoder.util.log.jdk14.Jdk14Log

This is the implementation of the Log interface that will write the logging messages to the specified java logging system.

com.topcoder.util.log.log4j.Log4jLogFactory

This is the implementation of the LogFactory interface that will create Log4jLog instances based on the log4j logger for the given name.

com.topcoder.util.log.log4j.Log4jLog

This is the implementation of the Log interface that will write the logging messages to the specified log4j logging system.

1.5 Component Exception Definitions

com.topcoder.util.log.LogException:

This class is an exception class for all logging exceptions thrown from this API. It provides the ability to reference the underlying exception via the getCause method,

inherited from BaseException.

1.6 Thread Safety

This component has been made reasonably thread-safe.

Typically, configuration of the Logging component is done before the application is up and running. Since configuration is done in a public manner, that configuration can technically be called at any time. In this case, to fully support thread safety, the normal logging path would have to synchronize whenever a configurable item is touched and since logging is executed quite a bit – that synchronization overhead would likely impact the performance of the application. However, since configuration is typically done at startup and typically not touched afterwards, it's reasonable to assume that the configuration API won't be called during logging and the synchronization can safely be ignored.

This component implements this reasonable thread safety for the setup type of items: the logging factory and the object formatter. Both of these setups should be done prior to actual logging usage. If either is done any other time during the lifetime of the application, it may work 9,999 times in a row – but there is no guarantee that it will work the 10,000 time since this component assumes the setup work will always be done prior to usage.

Beyond the setup information, the component itself is thread-safe by having either stateless or immutable state information classes. However, reasonably thread safe still applies since we assume the underlying logging system is thread-safe also (which may not be the case).

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)
- Object Formatter Component 1.0 is used to provide the ability to format message objects automatically.

2.3 Third Party Components

Log4j-1.2.12 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

No configuration is necessary.

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-1.2.12. 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

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 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 log4j-1.2.12.jar in your classpath.
- Import the appropriate classes from the com.topcoder.util.log package and appropriate sub packages.

4.3 Demo

4.3.1 What needs to change when upgrading from v1.2 to v2.0:

There are two breakage points when upgrading from v1.2 to v2.0:

a. The loss of specifying the logger implementation via the configuration file.
b. The main class name that changed.

Fortunately, both breakage points can be rectified with one simple change.

In v1.2 – you would load the configuration with the following line:

```
LogFactory.loadConfiguration();
```

To convert this to v2.0 – you simply replace the line with the following (using the appropriate logging factory):

```
LogManager.setLogFactory(new Log4jLogFactory());
```

This corrects both the name change and specification of the underlying logging system.

4.3.2 Logging setup

Before any logging is done, the Logging Wrapper component should be setup. Setup involves two steps – specification of the underlying logging factory and specification of

the message object formatting (if needed).

```
// set the underlying logging factory to use the Log4jLog
// the logging configuration of log4j must be specified according to
log4j requirements
LogManager.setLogFactory(new Log4jLogFactory());
// set the message object formatting to use PrimitiveFormatter
ObjectFormatter objectFormatter =
ObjectFormatterFactory.getPrettyFormatter();
// get the Log instance, it should be Log4jLog
Log log = LogManager.getLog();
// log the object using PrimitiveFormatter
log.log(Level.INFO, new Integer(123456789), objectFormatter);
```

Both steps are optional. If the underlying logging factory is not specified, logging to the console (System.out) will be used by default. If the message object formatting is not specified, a simply object.toString() will be used for formatting.

4.3.2.1 Specification of the Logging Factory

In version 2.0, the component provides three basic logging systems:

a) A Basic Log Factory that will log to a specified print stream.
b) A Java Log Factory that will log to the Java 1.4+ logging API
c) A Log4J Log Factory that will log to the Log4J logging API

Although not required, it's highly recommended that the logging factory be specified as early as possible and only be specified once. Here is an example of using the Basic Log Factory to write logging to a file.

```
// create a print stream to the file with auto flushing
PrintStream ps = new PrintStream(new
FileOutputStream("test_files/log.txt"), true);
// specify the basic logger with the above print stream
// it's highly recommended that the logging factory is only
specified once
LogManager.setLogFactory(new BasicLogFactory(ps));
// application code ...
// any logging from this point on will go to the
"test_files/log.txt" file
// get the Log instance, it should be BasicLog
Log log = LogManager.getLog();
// log the object using PrimitiveFormatter
log.log(Level.INFO, new Integer(123456789));
```

4.3.2.2 Specification of Object Formatting

This component makes use of the Object Formatter component to separate the formatting of an object from the object itself or from the logging line itself.

If an application needed to log property change events in a GUI application, the logging code would look something like:

```
logger.log(Level.DEBUG, event.getPropertyName()
  + " changed to a new value of " + event.getNewValue());
```

Or like:

```
logger.log(Level.DEBUG, event.toString());
```

Using the new message format API, you could change this to:

```
logger.log(Level.DEBUG, "{0} changed to a new value of {1}",
      event.getPropertyName(), event.getNewValue());
```

The problem with any of those is that the formatting of the event is duplicated in every logging line (especially across the application) or the formatting of the event is left up to its toString method (which ties the formatting directly to the object or provides more/less information than you need). If you wanted to add in the old value from the event, you'd have to modify every location.

A better way would be to setup the application with specific object formatters:

```
public static void main(String[] args) {

      // specify Log4J logging
      LogManager.setLogFactory(new Log4jLogFactory());

      // specify a format method for all PropertyChangeEvent classes
      LogManager.getObjectFormatter().setFormatMethodForClass(
            PropertyChangeEvent.class,
            new ObjectFormatMethod() {
                  public String format(Object o) {
                        PropertyChangeEvent e =
      (PropertyChangeEvent) o;
                        return e.getPropertyName() + " changed from
      " + e.getOldValue() + " to " + e.getNewValue();
                  }
            },
            true);
      // application code...
      // get the Log instance, it should be Log4jLog
      Log log = LogManager.getLog();
      // then the logging line would be simply:
      log.log(Level.INFO, new PropertyChangeEvent(new Object(),
      "propertyName", "oldValue", "newValue"));
}
```

If you ever needed to change the format of the logging, you'd just have to change the ObjectFormatMethod above and all instances of the event logging would change automatically. This provides separation of formatting code from both the object itself and the logging locations.

4.3.3 *Traditional Message Logging with Exception Stack Trace*

Logging a message with or without an exception is fairly straight forward. You simply acquire a Log then use the API to log message and/or errors:

```
Log log = LogManager.getLog("acme");
// other codes ...
// Log the entry into the message
log.log(Level.DEBUG, "Entering doUpdate");
try {
      // some SQL based updates
      doSQLUpdates();
} catch (SQLException e) {
      // Log the exception with a full stack trace
      log.log(Level.ERROR, e, "Update failed");
}
```

4.3.4 *Message Formatting Logging*

The performance aspect of logging is an area that needs constant attention.

Example: the following logging code has a fairly high overhead because it constructs a String regardless if the message will be logged or not:

```
log.log(Level.DEBUG, someString + " happened at " +
System.currentMillis());
```

If only Level.INFO and above is enabled, the above line will waste time by constructing the string first then determining nothing should be logged.

Traditionally, you could get around that issue by writing the code like:

```
if (log.isEnabled(Level.DEBUG)) {
      log.log(Level.DEBUG, someString + " happened at " +
      System.currentMillis());
}
```

However, this is fairly involved code for logging and programmers tend to forget the performance aspect of that.

Instead, the message format API can be used:

```
log.log(Level.DEBUG, "{0} happened at {1}", someString,
System.currentMillis());
```

This minimizes the overhead by allowing the formatting to happen as late as possible (certainly after a check to see if logging is enabled). This allows the code to be more compact and easier to write (and less performance error prone).

The API provides overloaded "log" methods for specification of up to three arguments directly. More arguments can be specified by using the object array API:

```
// get the BasicLog
      Log log = LogManager.getLog();
      String tradeDesc = "tradeDesc";
     Integer numOfShares = new Integer(3);
      Double pricePerShare = new Double(1.5);
      Date tradeTime = new Date();
      // log the message formatting
      log.log(Level.DEBUG, "Trade {0} for {1} shares at {2} occurred
at {3}",
            new Object[] {tradeDesc, numOfShares, pricePerShare,
tradeTime});
```

4.3.5 *NIO Server example*

The following is an example of the usage of the Logging Wrapper component in a NIO Server. This example demonstrates the setup and API usage of the component:

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
```

```java
import java.net.UnknownHostException;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.spi.SelectorProvider;
import java.util.Iterator;

import com.topcoder.util.format.ObjectFormatMethod;
import com.topcoder.util.log.jdk14.Jdk14LogFactory;


/**
 * <p>
 * This class is an example of the usage of the Logging Wrapper component
in a NIO Server.
 * </p>
 *
 * <p>
 * This example demonstrates the setup and API usage of the component.
 * </p>
 *
 * <p>
 * This class will be used in DemoTests.
 * </p>
 *
 *
 * @author TCSDEVELOPER
 * @version 2.0
 */
public class NioServer implements Runnable {

        /**
         * <p>
         * Represents the InetAddress with host:port combination to listen
on.
         * </p>
         */
        private InetAddress hostAddress;

        /**
         * <p>
         * Represents the port to listen on.
         * </p>
         */
        private int port;

        /**
         * <p>
         * Represents the channel on which we'll accept connections.
         * </p>
         */
        private ServerSocketChannel serverChannel;

        /**
         * <p>
         * Represents the selector we'll be monitoring.
         * </p>
```

```
         */
        private Selector selector;

        /**
         * <p>
         * Represents the logger we'l be using.
         * </p>
         */
        private Log log = LogManager.getLog("nioserver");

        /**
         * <p>
         * Constructs a NioServer.
         * </p>
         *
         * @param args the arguments to create a NioServer
         */
        public NioServer(String[] args) {
              try {
                    // parse arguments and get the initial selector
                    this.hostAddress =
InetAddress.getByName(args[0]);
                    this.port = Integer.parseInt(args[1]);
                    this.selector = this.initSelector();
              } catch (UnknownHostException e) {
                    // log error with exception stack trace
                    log.log(Level.ERROR, e, "Invalid host name: {0}",
args[0]);
              } catch (NumberFormatException e) {
                    // log error the number parsing error
                    log.log(Level.ERROR, "Specified port '0}'was not a
valid number", args[1]);
              } catch (IOException e) {
                    // log the IO error
                    log.log(Level.ERROR, e, "IOException occurred
initializing server");
              }
        }

        /**
         * <p>
         * Creates and initializes the selector.
         * </p>
         *
         * @return the selector which has been created and initialized
         *
         * @throws IOException if fail to create and initialize the
selector
         */
        private Selector initSelector() throws IOException {
              // create a new selector
              Selector socketSelector =
SelectorProvider.provider().openSelector();

              // create a new non-blocking server socket channel
              this.serverChannel = ServerSocketChannel.open();
              serverChannel.configureBlocking(false);
```

```
                // bind the server socket to the specified address and port
                InetSocketAddress isa = new
InetSocketAddress(this.hostAddress, this.port);
                serverChannel.socket().bind(isa);

                // register the server socket channel, indicating an
interest in
                // accepting new connections
                serverChannel.register(socketSelector,
SelectionKey.OP_ACCEPT);

                return socketSelector;
          }

          /**
           * <p>
           * The entry point method of this thread.
           * </p>
           */
          public void run() {
                log.log(Level.INFO, "Server started for {0} listening to
port {1}", hostAddress, new Integer(port));

                while (true) {
                      try {
                            // wait for an event one of the registered
channels
                            this.selector.select();
                            // iterate over the set of keys for which events
are available
                            Iterator selectedKeys =
this.selector.selectedKeys().iterator();
                            while (selectedKeys.hasNext()) {
                                  SelectionKey key = (SelectionKey)
selectedKeys.next();

                                  // log the selection event
                                  log.log(Level.DEBUG, key);

                                  selectedKeys.remove();
                                  if (!key.isValid()) {
                                        continue;
                                  }

                                  // check what event is available and deal
with it
                                  if (key.isAcceptable()) {
                                        this.accept(key);
                                  }
                            }
                      } catch (IOException e) {
                            // log the exception and break
                            log.log(Level.ERROR, e, "Server encountered
an exception during selector processing");
                            break;
                      }
```

```
                }
                // log server stopping message
                log.log(Level.INFO, "Server stopped");
        }

        /**
         * <p>
         * Accepts a connections.
         * </p>
         *
         * @param key the selectionKey representing the registration of
a SelectableChannel with a Selector.
         */
        public void accept(SelectionKey key) {
                // ... application processing code
        }

        /**
         * <p>
         * The Main entry point.
         * </p>
         *
         * @param args the arguments used for Main entry point
         */
        public static void main(String[] args) {
                // set logging to use the Java Logging API
                LogManager.setLogFactory(new Jdk14LogFactory());

                // create an object formatter for a selection key

   LogManager.getObjectFormatter().setFormatMethodForClass(
                                SelectionKey.class,
                                new ObjectFormatMethod() {
                                        public String format(Object o) {
                                                SelectionKey k = (SelectionKey)
o;
                                                return "InterestOps: " +
k.interestOps() + ", ReadyOps: " + k.readyOps();
                                        }
                                },
                                true);
                // start the server
                new Thread(new NioServer(args)).start();
        }
   }
```

5. Future Enhancements

Restore the logger implementations from v1.4:

a) The composite logger implementation was a good implementation to combine a generic logging system with some application specific need. Example: in the Xmpp Assembly contest – the Xmpp needed to send log levels above a certain level to a terminal session. A composite logger would have been useful to provide this specialized logging in addition to the standardized logging.

b) A filtering logging wrapper would be useful to filter in/out messages based on

some criteria (and would be doubly useful when combined with a composite logger).

c) A database logger implementation would be useful to log messages to a database for storage.

d) An asynchronous logging wrapper is useful when implementing a database or other high latency logging implementation.