

Streamline Common Base Event logging

Reduce complexity and instrumentation code size

Skill Level: Intermediate

[Paul Slauenwhite \(paules@ca.ibm.com\)](mailto:paules@ca.ibm.com)

Software Developer

IBM<reg></reg>

03 Jan 2006

Common Base Event logging provides problem-determination data rich in activity and control-flow information that the Log and Trace Analyzer uses to accelerate problem determination and increase system serviceability and quality. This tutorial shows how to streamline Common Base Event logging instrumentation in the Java™ programming language to reduce complexity and instrumentation code size. You'll learn techniques and best practices that reduce maintenance requirements and let you increase customized, vendor-specific content.

Section 1. Before you start

About this tutorial

The Log and Trace Analyzer (LTA) in the IBM Autonomic Computing Toolkit monitors, correlates, analyzes, diagnoses, and resolves run-time problems in complex heterogeneous systems. The LTA collects and consolidates practical problem-determination data in the Common Base Event format from disparate systems into a single management tool. You can then use the LTA for viewing, navigating, sorting, filtering, searching, correlating, and analyzing.

My previous tutorial, "Common Base Event Logging" (see [Resources](#)), showed how to configure Common Base Event logging in the Java language. This gives you a

rich source of activity and control-flow information to accelerate problem determination and increase system serviceability and quality. The LTA can then use logged Common Base Events to detect and resolve configuration errors, performance degradation, exception states, resource starvation, security failures, communication delays, deadlocking, and other problems. However, a common criticism of Common Base Event logging is that it is too heavyweight for simple message logging, causing increased complexity, instrumentation code size, and maintenance. This tutorial shows how to streamline Common Base Event logging to mitigate these issues.

The tutorial:

- Reviews the concept of problem determination and the Common Base Event model. The review details the benefits of and concerns with logging events and problems in the Common Base Event format.
- Teaches best practices that let you streamline Common Base Event logging instrumentation in the Java programming language to reduce complexity and instrumentation code size.
- Shows how to leverage the Common Base Event factory to create a customized XML configuration template for populating Common Base Events with static system-specific content.
- Shows how to extend the Common Base Event factory to create a customized Event Factory Home for retrieving the same static system-specific content set as part of the Java Logging configuration (for example, `java.util.Properties` configuration file) and retrieved from Java Logging's Log Manager.
- Shows how to create a customized Content Handler for populating Common Base Events with dynamically resolved run-time content.
- Shows how to merge the Event Factory Home and Content Handler for streamlined Common Base Event logging based on the system's logging configuration file and run time to decrease maintenance and increase customized, vendor-specific content.

Developers can use the best practices and techniques in this tutorial when instrumenting their code for Common Base Event logging, but they are intended primarily for system serviceability architects. Serviceability architects provide a logging strategy for the complete system that individual developers follow when instrumenting Common Base Event logging in their components. System serviceability architects can use these best practices and techniques to craft a logging strategy to ensure consistent and useful problem-determination data to increase system serviceability and quality.

Note that the Common Base Event model is not associated with a particular

consumer, such as a logging facility. This tutorial uses the Java Logging APIs because they provide an integrated and extensible logging facility for the Java language useful for illustrative purposes. However, numerous proprietary and open source logging facilities for the Java language are available. The tutorial is not intended to be an exhaustive aid in using this or any other logging facility.

Prerequisites

This tutorial is intended for serviceability architects and developers who want to streamline Common Base Event logging in new and existing systems written in the Java language. Basic knowledge of generic logging concepts and Common Base Events, and moderate experience with the Java language and Eclipse, will help you complete the tutorial's tasks and understand its examples.

This tutorial requires the Autonomic Computing Toolkit V3.0.0. The Autonomic Computing Toolkit V3.0.0 requires Eclipse V3.0.3. Eclipse V3.0.3 requires the Java Runtime Environment (JRE) V1.4.x (see [Resources](#)).

To run the code and configuration in this tutorial, you need the Common Base Event V1.0.1 Java implementation and support in the Log and Trace Analyzer V3.3.0, which is part of the Autonomic Computing Toolkit V3.0.0. Several plug-in Java Archive (JAR) files located in the *<Autonomic Computing Toolkit installation directory>\eclipse\plugins* directory are required for this tutorial (see [Table 1](#)).

Table 1. Plug-in JAR files required to run the code and configuration in this tutorial

JAR file	Plug-in
hlcbe101.jar	org.eclipse.hyades.logging.core_3.3.0
hlcore.jar	org.eclipse.hyades.logging.core_3.3.0
ecore.jar	org.eclipse.emf.ecore_2.0.2\runtime
common.jar	org.eclipse.emf.common_2.0.1\runtime
hl14.jar	org.eclipse.hyades.logging.java14_3.3.0
hexr.jar	org.eclipse.hyades.execution.remote_3.3.0

See [Resources](#) or *<Autonomic Computing Toolkit installation directory>\eclipse\plugins\org.eclipse.hyades.logging.core_3.3.0\doc.cbe101\index.html* for a link to the Common Base Event V1.0.1 Java implementation API Javadoc documentation.

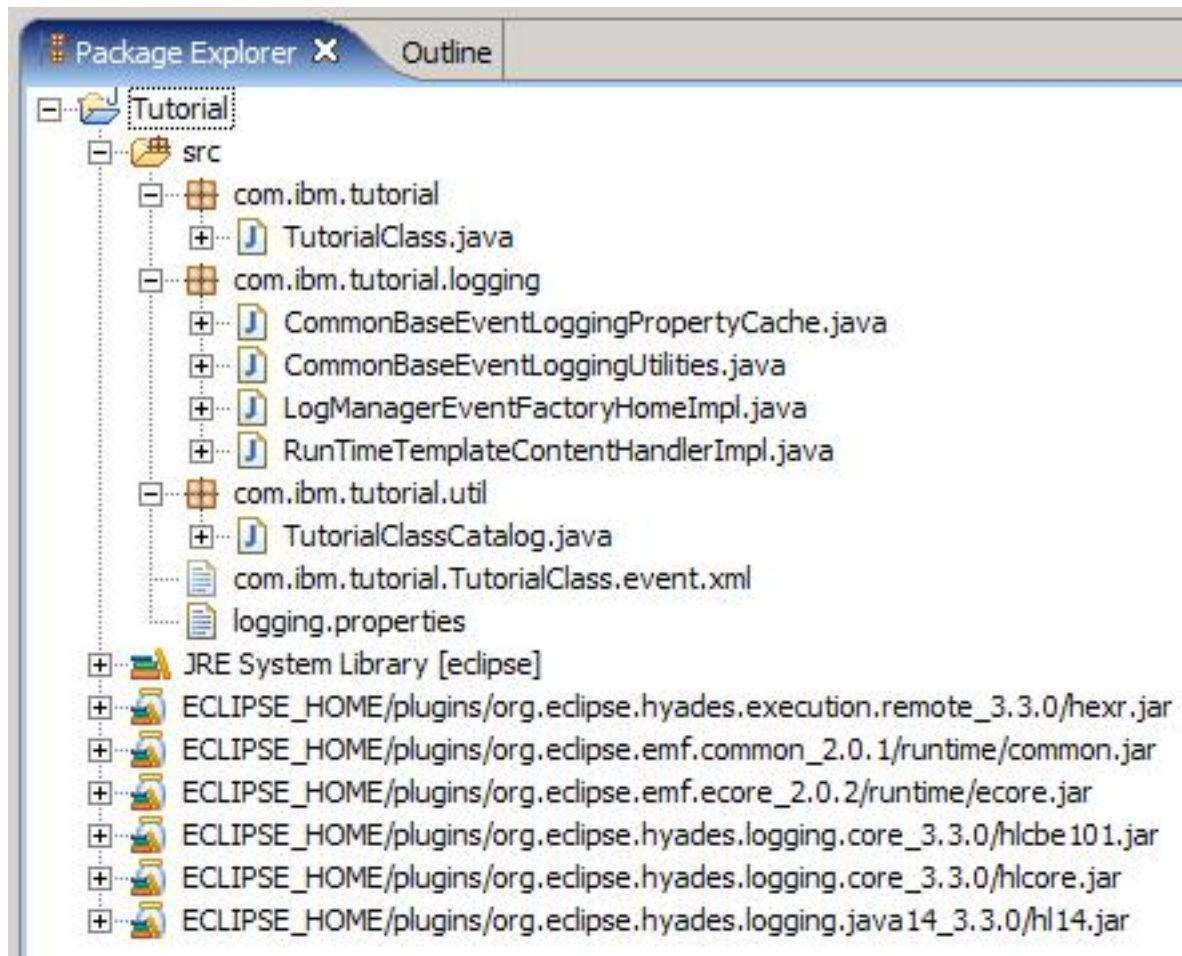
Sample code and configuration

This tutorial guides you through streamlining Common Base Event logging instrumentation using the Java Logging APIs. The sample code and configuration are provided for reference as an Eclipse Java project. The sample code (`com.ibm.tutorial.TutorialClass`) provides four distinct scenarios, ranging from standard to fully streamlined Common Base Event logging. These four scenarios illustrate the gradual reduction of complexity and of lines of instrumentation code that you can achieve by using the best practices and techniques you'll learn in this tutorial. To make the project available, follow these steps:

1. Save the [Tutorial_Java_Project.zip](#) file to a local directory.
2. Start the Autonomic Computing Toolkit.
3. Open the Java Perspective by selecting **Window > Open Perspective > Other... > Java > OK**.
4. Create a new Java project by selecting **File > New > Project... > Java > Java Project > Next**.
 1. Enter a project name, for example `Tutorial`.
 2. Select **Finish**.
5. Import the `Tutorial_Java_Project.zip` file into the newly created Java project by selecting the project in the Package Explorer of the Java Perspective and selecting **File > Import... > Zip file > Next**.
 1. Select **Browse...** and navigate to the `Tutorial_Java_Project.zip` file saved in the local directory in Step 1.
 2. Select the `Tutorial_Java_Project.zip` file and select **Open**.
 3. Select **Finish**.
 4. Select **Yes** in the question dialog.

[Figure 1](#) illustrates the resultant Java project in the Package Explorer of the Java Perspective.

Figure 1. Java project in the Package Explorer of the Java Perspective



Section 2. Problem determination and the Common Base Event model

This section provides context for this tutorial by reviewing the concept of problem determination and the Common Base Event model, including the benefits of and concerns with logging events and problems in the Common Base Event format. If you are familiar with these concepts, feel free to skip to the next section, [Best practices for Common Base Event logging](#).

What is problem determination?

Problem determination involves detecting, isolating, and resolving run-time problems in computing systems by monitoring, correlating, and analyzing the system's

execution activity. As computing systems become increasingly complex and fragmented because of the rapid advancements in raw computing capabilities and the proliferation of distributed technologies, so does the importance of management-related activities such as problem determination. For instance, architectures often incorporate numerous subsystems, such as servers and databases, and multiple components, such clients and applications, across internal and external networks. Problem determination is fundamental to the successful and sustainable operation of these complex heterogeneous systems.

Fortunately, most existing computing systems leverage *logging*. Logging consists of identifying, capturing, and persisting common events and problems during normal execution. The resulting log data, or problem-determination data, is often rich in execution-activity and control-flow information essential for problem determination.

The "problem" with problem-determination data

Problem-determination data often contains varying levels of detail. For instance, log messages might correctly describe an event or problem but not its source or context. Furthermore, problem-determination data generated by heterogeneous systems with multiple subsystems and components typically contains vendor-, product-, and version-specific formats and terminology. For instance, time stamps that denote when an event or problem was captured and logged might be represented in varying formats. Also, problem-determination data is often oriented toward a specific component and not the entire system. Finally, problem-determination data frequently uses varying natural languages and grammars to describe events and problems. For example, a log message denoting a component's initialization might be represented in English as "Component A has started" and in French as "Le commencement du component A est accompli" ("The beginning of component A is completed").

The varying detail, terminology, and format of problem-determination data unnecessarily complicates problem-determination activities. Deciphering events and problems reported by the system's numerous subsystems and components can be challenging and tedious.

The Common Base Event model

The Common Base Event model seeks to circumvent problem-determination data's varying formats, terminology, and level of detail. The model provides a standardized taxonomy, unified format, and consistent terminology for encoding events and problems occurring in hardware and software. A *common* representation for all events and problems aids in their consistent interpretation and standardized exchange. For instance, the Common Base Event's timestamp property -- named `creationTime` -- is defined as an XML Schema `dateTime` type, thereby forcing agreement on its format. Furthermore, the model enables the interoperability of problem-determination technologies and tooling -- such as the Autonomic

Computing Toolkit -- thereby reducing maintenance costs and increasing system stability, security, availability, resiliency, quality, and asset utilization.

Common Base Events are based on a structured *3-tuple* format:

- Situation, including descriptive information
- Component that is reporting the situation
- Component that is affected by the situation (which can be the same as the component reporting the situation)

Version 1.0.1 of the Common Base Event model was originally drafted by IBM. The model was submitted as a specification to the Eclipse Test and Performance Tool Platform (TPTP) open source project (formally Hyades) for adoption and implementation. The resulting Common Base Event V1.0.1 Java implementation is included in the Log and Trace Analyzer V3.0.0. The model and implementation are based on open source standards to promote industry-wide acceptance and adoption.

See [Resources](#) for the current Common Base Event specification and an article explaining the model in greater detail.

Benefits of Common Base Event logging

The most effective method of capturing vital problem-determination data in a system is *Common Base Event logging*. By using the Common Base Event model's standardized taxonomy, unified format, and consistent terminology to encode events and problems, you can circumvent the potential for varying levels of detail, vendor-, product-, and version-specific terminology and formats in problem-determination data. However, you need to create and configure Common Base Events in close proximity to the represented event or problem to fully leverage the model's intended benefits. More-detailed and exact information can be captured and persisted in the Common Base Event when it is created in close proximity to the represented event or problem. For instance, situational and contextual information can be extracted directly from the event or problem in conjunction with the pertinent run time configuration and execution information. Creating, configuring, and logging Common Base Events at the source of the event or problem provides the detailed, exact, standardized, and consistent data required for accelerated problem determination and increased system serviceability and quality.

Concerns with Common Base Event logging

Complexity and instrumentation code size are common concerns with Common

Base Event logging. The Common Base Event model was designed to describe all events and problems abstractly within a system, subsystem, or component, but to be granular enough to describe any occurrence of an event or problem. As a result, the model is often considered heavyweight for simple message logging. Disregarding the tangible serviceability benefits of leveraging the model, the argument is valid. In order for the Common Base Event model to provide a standardized taxonomy, unified format, and consistent terminology for encoding all events and problems occurring in hardware and software, by definition its design is complex. For instance, the Common Base Event model defines 12 core situations, each with multiple qualifiers just to define the type of the event or problem. Beginning users of the model and implementation often experience an elongated learning curve when instrumenting Common Base Event logging.

The complex design of the Common Base Event model directly contributes to increased Common Base Event logging instrumentation code size compared to alternative, simple message-logging approaches. In order to ensure the *common* representation of all events and problems, the Common Base Event model enforces several required properties for a Common Base Event and its complex properties. For example, the minimum required fields for a Common Base Event are:

- `creationTime`
- `sourceComponentId`
- `situation`

Coupled with the additional optional Common Base Event properties, instrumenting code size can stretch from 20 to 100 lines of code to create one Common Base Event. Moreover, the model enforces defined formats for each of its properties to circumvent the varying formats and terminology in problem-determination data. For instance, the Common Base Event's *severity* property captures severity values ranging from 0 to 70. These defined formats also contribute to increased instrumentation code size when vendor-, product-, and version-specific formats and terminology need to be converted or transformed to the analogous Common Base Event format.

The rather large amount of the instrumentation code required for simple message logging requires a significant percentage of total code for the system to be dedicated to Common Base Event logging instrumentation, resulting in decreased readability and maintainability. Potential adopters are often discouraged from realizing the tangible serviceability benefits of leveraging Common Base Event logging.

The rest of this tutorial will provide you with the best practices and techniques required to streamline Common Base Event logging instrumentation by controlling complexity and decreasing the lines of instrumentation code. The result will be decreased maintenance and increased customized, vendor-specific content.

Section 3. Best practices for Common Base Event logging

The first approach in streamlining Common Base Event logging instrumentation is to use best practices. This section examines several best practices for controlling the complexity and size of code in Common Base Event logging instrumentation.

Controlling complexity

Identifying core problem-determination data

Core problem-determination data required for system serviceability and quality

Several types of core problem-determination data are required for system serviceability and quality:

- Event time stamps
- Event severities and priorities
- Component, package, or class name reporting the event
- Location of the component, package, or class name reporting the event in a distributed system
- Exception states and stack traces
- Descriptive information

A preliminary step in controlling complexity when you instrument an application for Common Base Event logging is to identify core problem-determination data for the system. Understanding this data forces you to identify the required activity and control-flow information required in your logging instrumentation to accelerate problem determination and increase system serviceability and quality.

The core problem-determination data typically consists of the situational and contextual information on execution state-change events in conjunction with the relevant run time configuration and execution information. If logging is instrumented in the system, this is a simple exercise of aggregating the core problem-determination data currently being logged.

Otherwise, the exercise is somewhat more involved, because you must identify and aggregate the core problem-determination data required for system serviceability

and quality. After it's identified, the core problem-determination data provides the necessary foundation for the quantity and composition of logging instrumentation. As a best practice, identify the pertinent execution state-change events and aggregate the following information for each event or problem:

- Properties of the problem or event, such as situation, severity, timestamp, and message
- Location of the problem or event, such as package, class, and method names
- Properties of the run time, such as version, IP address, and thread ID

Mapping properties

After you identify the system's core problem-determination data, the next step in controlling complexity is to craft a logical mapping between key properties in the core problem-determination data and Common Base Event properties. (An analogous process is followed when you use the Generic Log Adapter in the Autonomic Computing Toolkit to convert proprietary problem-determination data into the Common Base Event format for use by the Log and Trace Analyzer. The Generic Log Adapter is a rules-based tool that uses a user-provided mapping file, called an *adapter*, to map proprietary log-data properties to Common Base Event properties.)

In preparation for deriving this logical mapping, consult the numerous resources available to help you understand the Common Base Event model (see [Resources](#)). Clear and concise understanding of the model will simplify this process and ensure the correct and accurate use of the model. The exercise of exploring the Common Base Event model for properties most relevant to the system's core problem-determination data typically shortens the elongated learning curve often experienced by users of the Common Base Event model and implementation. The logical mapping, once finalized, provides a high-level design for the Common Base Event logging instrumentation.

If Java Logging is currently instrumented in the system, you can write a customized log formatter (extension of the `java.util.logging.Formatter` class) based on the logical mapping to convert proprietary log records to Common Base Events -- analogous to the formatting support in the LTA to convert `java.util.logging.LogRecord` to Common Base Events (see the `org.eclipse.hyades.logging.java.XmlFormatter` class). The existing logging instrumentation site does not require refactoring to benefit from Common Base Event logging. And the same code can be used as a generic logging API for the system, thereby reducing the code at the logging instrumentation site to a single line.

Identifying static and dynamic data

After you've derived a logical mapping of core problem-determination data to Common Base Event properties, identify the static and dynamic data. Static data consists of Common Base Event properties that do not change during or between the execution of the system -- for example, run time version, package names, and class names. The static data becomes the focus of the logging configuration. The next two sections of this tutorial discuss the several best practices and techniques for dealing with static data using the Common Base Event factory's configuration template.

Conversely, dynamic data consists of Common Base Event properties that are specific to the particular run time instance of the system, event, or problem -- for example, situation, method names, IP addresses, thread IDs, severities, timestamps, and messages. The dynamic data becomes the focus of the actual logging instrumentation code and the Content Handler of the Common Base Event factory discussed in the following two sections. The division between static and dynamic data is an important step in preparing to decrease the number of lines of instrumentation code.

Decreasing lines of instrumentation code

Regrettably, terse Common Base Event logging instrumentation that fully exploits its intended serviceability benefits is simply not possible because of the model's elaborate design. For instance, 20 lines of code are required to create a Common Base Event with the minimum required fields. Unfortunately, there is no silver bullet to achieve both high serviceability and terse logging instrumentation. However, you can use many best practices and techniques to decrease the lines of code at the actual instrumentation site. Given that high serviceability and terse logging instrumentation are mutually exclusive, the only recourse is to move lines of code from the actual instrumentation site to common components and modules, thereby decreasing maintenance and accommodating vendor-specific content.

Caching recurrent properties

The first approach to decreasing lines of instrumentation code is to cache all recurrent Common Base Event properties, generic factories, and run time properties that do not change state during the system's life cycle. Each property in the Common Base Event logging property cache is statically initialized for reuse throughout the system. As with all caching strategies, using a Common Base Event logging property cache promotes decreased memory consumption, increased performance, improved maintenance, and decreased lines of code.

For instance, each Common Base Event instantiated in the system requires a `situation` property that translates into seven to nine lines of code for its creation, configuration, and association with the Common Base Event. Fortunately, the Common Base Event model identifies a finite number of core situations -- 12 to be

exact -- used to identify the type of events and problems. Caching the situations most commonly used in the logging instrumentation reduces the code for setting these cached situations on the `situation` property at the instrumentation site to one line. Although additional situation objects for the system's life cycle cause some memory overhead, the reuse of these instances throughout the system results in a decrease of situation objects at any point in time of the system's life cycle.

The Common Base Event logging property cache can also contain generic factories for creating individual Common Base Event properties to be reused throughout the system. Common Base Events and their complex properties (for example, `java.lang.Object`) can be created with a cached anonymous Event Factory instance resolved from the simple Event Factory Home. Common Base Event properties that do change state during the system's life cycle can also be cached for reuse -- for example, a running counter of all logged Common Base Events in the system used as the `sequenceNumber` Common Base Event property. However, complex Common Base Event properties (for example, `java.lang.Object`) require cloning by the caller. For example, the `msgDataElement` property can be cached, cloned, and reused with the applicable `msgId`, `msgCatalogId`, and `msgCatalogTokens` properties set in the instrumentation code.

Finally, you can store pertinent run time proprieties, such as configuration and execution information, in the Common Base Event logging property cache. For example, [Listing 1](#) illustrates the static initialization of the local host's IP address, Report Situation, and Stop Situation.

Listing 1. Local host's IP address, Report Situation, and Stop Situation statically initialized for reuse in the Common Base Event property cache

```
/**
 * IP address of the local host, otherwise "127.0.0.1". The current IP address
 * of the local host is assumed to be an IPv4 address.
 */
public static String LOCAL_HOST_IP_ADDRESS = null;

/**
 * Situation for INTERNAL, SUCCESSFUL, and STOP COMPLETED stopping events.
 */
public static Situation INTERNAL_SUCCESSFUL_STOP_COMPLETED_SITUATION = null;

/**
 * Situation for INTERNAL and HEARTBEAT reporting events.
 */
public static Situation INTERNAL_HEARTBEAT_REPORT_SITUATION = null;

//Static initialization block:
static {

    //Resolve the current IP address of the local host:
    try {
        LOCAL_HOST_IP_ADDRESS = InetAddress.getLocalHost().getHostAddress();
    } catch (UnknownHostException u) {
        LOCAL_HOST_IP_ADDRESS = "127.0.0.1";
    }
}
```

```

//Create and populate an INTERNAL, SUCCESSFUL, and STOP COMPLETED Stop
//Situation:
INTERNAL_SUCCESSFUL_STOP_COMPLETED_SITUATION = ANONYMOUS_EVENT_FACTORY
    .createSituation();
INTERNAL_SUCCESSFUL_STOP_COMPLETED_SITUATION.setStopSituation("INTERNAL",
    "STOP COMPLETED", "SUCCESSFUL");

//Create and populate an INTERNAL and HEARTBEAT Report Situation:
INTERNAL_HEARTBEAT_REPORT_SITUATION = ANONYMOUS_EVENT_FACTORY.createSituation();
INTERNAL_HEARTBEAT_REPORT_SITUATION.setReportSituation("INTERNAL", "HEARTBEAT");
}

```

Utility methods in the Common Base Event implementation

The

`org.eclipse.hyades.logging.events.cbe.util.EventHelpers` class contains several utility methods to help decrease lines of instrumentation code:

- Conversion between an XML Schema `dateTime`, a millisecond time stamp, and a textual time stamp
- Conversion of a `java.lang.Object` to a Common Base Event
- Conversion of a `java.lang.Throwable` to an Extended Data Element
- Factory for the `values` property of an Extended Data Element to ensure that the 1,024-character limit is respected for each index of the array

Leveraging utility methods

Following on the concept of caching all recurrent complex Common Base Event properties, you can factor recurrent Common Base Event operations out to static utility methods in a Common Base Event logging utilities class. Often event or problem situational and contextual information, and pertinent run time configuration and execution information, must be converted to a different format suitable for the Common Base Event model -- for example, converting Java Logging levels to Common Base Event severities. Furthermore, creating and populating various Common Base Event properties can be abstracted to common utility methods. For example, [Listing 2](#) illustrates a static utility method for creating a Message Data Element using a combination of default properties and parameter properties.

Listing 2. Static utility code for creating a Message Data Element using default and the parameter properties

```

/**
 * Static utility method for creating a Message Data Element using default and
 * the parameter properties.
 * <p>
 *

```

```

* @param msgId
*     Message ID.
* @param msgCatalogId
*     Message catalog ID.
* @param msgCatalogTokens
*     Message catalog tokens as a String array.
* @return
*/
public static MsgDataElement createMsgDataElement(String msgId, String msgCatalogId,
    String[] msgCatalogTokens) {

    MsgDataElement msgDataElement = CommonBaseEventLoggingPropertyCache
        .ANONYMOUS_EVENT_FACTORY.createMsgDataElement();
    msgDataElement.setMsgIdType("IBM4.4.1");
    msgDataElement.setMsgCatalog(TutorialClassCatalog.TUTORIAL_CLASS_CATALOG_NAME);
    msgDataElement.setMsgCatalogType("Java");
    msgDataElement.setMsgLocale(Locale.getDefault().getLanguage().concat("-").concat(
        Locale.getDefault().getCountry()));
    msgDataElement.setMsgId(msgId);
    msgDataElement.setMsgCatalogId(msgCatalogId);
    msgDataElement.setMsgCatalogTokensAsStrings(msgCatalogTokens);

    return msgDataElement;
}

```

Periodically, problem-determination data does not cleanly map to one or more Common Base Event properties. In these cases, the problem-determination data can be mapped to one or more name-type-value properties in the `extendedDataElement` property. You can write such transforms as common utility methods, thereby promoting containment and reuse. An example is the `convertToExtendedDataElement()` method in the `org.eclipse.hyades.logging.events.cbe.util.EventHelpers` class that converts a `java.lang.Throwable` to an Extended Data Element. The Common Base Event V1.0.1 Java implementation provides several other static utility methods for a variety of common Common Base Event operations.

Recurrent logging operations can also be factored out to a static utility method in a Common Base Event logging utilities class -- for example, instantiating and configuring an `org.eclipse.hyades.logging.java.CommonBaseEventLogRecord`. Because the Java Logging APIs support logging only `java.lang.String` and `java.util.logging.LogRecord` log records, the LTA provides the `org.eclipse.hyades.logging.java.CommonBaseEventLogRecord` subclass of the proprietary `java.util.logging.LogRecord` log record required to log a Common Base Event in Java Logging. Because the `org.eclipse.hyades.logging.java.CommonBaseEventLogRecord` must be instantiated with a Common Base Event and configured with a logger's name before being logged, factoring this operation out to a static utility method translates three lines of code to a single line at the instrumentation site.

Using convenience methods

Finally, the Common Base Event V1.0.1 Java implementation provides various

set on the Common Base Event at the instrumentation site have precedence over the properties populated by the Content Handler. *The factory is the core mechanism for decreasing the lines of instrumentation code in Common Base Event logging.* All of the static problem-determination data and a subset of the dynamic problem-determination data identified in the previous section ([Best practices for Common Base Event logging](#)), can be abstracted to the factory's configuration template and Content Handler. Leveraging the Common Base Event factory results in reduced code size, decreased maintenance, and increased customized, vendor-specific content.

A layered design

The factory's design has a layered architecture with four levels of abstraction:

- Event Factory Context
- Event Factory Home
- Event Factory
- Event

The Event Factory Context layer provides an optional look-up service for Event Factory Homes based on the fully qualified package and class name of the Event Factory Home implementation. This look-up service eliminates the need to import and instantiate typed Event Factory Homes directly in instrumentation code but is not required to create an Event Factory Home.

The Event Factory Home layer provides Event Factory instantiation based on a unique factory name. The Event Factory Home caches named Event Factory instances for added performance with multiple accesses. Cached Event Factory instances can be updated (for example, by invoking the `updateEventFactory()` method) and released from the cache (for example, by invoking the `releaseEventFactory()` method) at any time during their life cycle. Event Factory Homes are tightly coupled with a specific type of configuration template used by a specific type of Content Handler to populate Common Base Events. Event Factory Homes resolve and reload a typed Content Handler based on a typed configuration template from a locatable source. The Event Factory Home and Content Handler are extended to provide run time- and system-specific content, which you'll read about in the next section, [Extending the Common Base Event factory](#). The Content Handler resolved by the Event Factory Home is set on the instantiated Event Factory for association with all newly created Common Base Events.

The Event Factory layer provides a named factory for creating Common Base Events. Event Factories are named using a hierarchical dot-delimited naming convention. As a best practice, factory names should resolve to the component,

package, or class name, depending on the granularity of the configuration template and in line with the system's serviceability initiative. The Event Factory associates its Content Handler with each newly created Common Base Event. Moreover, the Event Factory can be configured to automatically populate or *complete* the newly created Common Base Event (for example, invoke the Common Base Event's `complete()` method). This eliminates the need to invoke the `complete()` method at the instrumentation site. Although this auto-complete feature has negative performance implications if the Common Base Event is never consumed, it eliminates a notable number of lines of code from the instrumentation site.

Finally, the Event layer is the Common Base Event instance. You can *complete* Common Base Events at any time during their life cycle by invoking the `complete()` method. This method calls its Content Handler to populate the properties of the Common Base Event. As a best practice for performance reasons, do not invoke the `complete()` method until the Common Base Event is about to be consumed or traverse system boundaries (for example, serialized to XML and sent across a network).

Using the event XML file Event Factory Home

The Common Base Event V1.0.1 Java implementation provides a reference implementation of both the Content Handler and the Event Factory Home. The reference implementation of the Content Handler is the *Template Content Handler*, as defined in the `TemplateContentHandlerImpl` class. The Template Content Handler uses a Common Base Event instance as the configuration template to *complete* Common Base Events.

The reference implementation of the Event Factory Home is the event XML file Event Factory Home, as defined in the `EventXMLFileEventFactoryHomeImpl` class. The event XML file Event Factory Home resolves and reloads a Template Content Handler based on a template XML file as the configuration template. The naming convention used for the template XML file is `<Event Factory's name>.event.xml`.

The template XML file is a well-formed XML document consisting of a single Common Base Event XML fragment with one or more defined properties. The XML document conforms to a predefined XML schema (`templateEvent.xsd`) provided in the Common Base Event V1.0.1 Java implementation. You should define in the template XML file the static Common Base Event properties identified in the logical mapping of core problem-determination data to Common Base Event properties. For example, [Listing 3](#) illustrates a template event XML file named `com.ibm.tutorial.TutorialClass.event.xml` with various static properties for the tutorial class.

Listing 3. Template event XML file with various properties set for the tutorial

class

```
<TemplateEvent version="1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="templateEvent.xsd">

  <CommonBaseEvent
    severity="18"
    version="1.0.1">
    <sourceComponentId
      executionEnvironment="Java"
      application="Tutorial"
      componentIdType="Application"
      componentType="Java_Application"/>
    <situation
      categoryName="ReportSituation">
      <situationType
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="ReportSituation"
        reasoningScope="INTERNAL"
        reportCategory="LOG"/>
      </situation>
    </CommonBaseEvent>
  </TemplateEvent>
```

As a best practice, you can define default Common Base Event properties in the template XML file. As a consequence, you need to set only nondefault Common Base Event properties on the Common Base Event at the instrumentation site. For example, the default `ReportSituation` and `severity` properties in [Listing 3](#) are set on all newly created Common Base Events in the `Tutorial` class. Leveraging the event XML file Event Factory Home to populate Common Base Events with static problem-determination data drastically reduces the lines of instrumentation code, decreases maintenance, and increases customized, vendor-specific content.

Resolving the template event XML file

In order for the event XML file Event Factory Home to resolve the template event XML file, the file must be accessible by the Event Factory Home's class loader (for example, on the classpath). The event XML file Event Factory Home searches for the template event XML file based on the hierarchical factory name, starting from the full factory name (for example, all segments) and iterating its ancestors to the highest ancestor (for example, first segment). For example, resolving the `com.ibm.tutorial.TutorialClass` Event Factory intern causes the event XML file Event Factory Home to resolve the template event XML file named `com.ibm.tutorial.TutorialClass`. If this template event XML file cannot be resolved, the event XML file Event Factory Home tries to resolve the template event XML file named `com.ibm.tutorial`, continuing until `com`. Template event XML files can be specified for any logical grouping of the system (for example, component, package, or class name) in line with its serviceability initiative.

[Listing 4](#) illustrates resolution of an Event Factory from the `EventXMLFileEventFactoryHomeImpl` Event Factory Home class using the

template event XML file in [Listing 3](#) for creating populated Common Base Events.

Listing 4. Resolving an Event Factory for creating populated Common Base Events

```

/**
 * Event XML file Event Factory Home to resolve Event Factory instances.
 */
public static EventFactoryHome EVENT_XML_FILE_EVENT_FACTORY_HOME = null;

/**
 * Event Factory instance for this tutorial class resolved from the event XML
 * file Event Factory Home.
 */
private static EventFactory eventXMLFileEventFactory = null;

/**
 * Constant for the fully qualified name (package and class name) of this
 * tutorial class.
 */
public final static String TUTORIAL_CLASS_NAME = TutorialClass.class.getName();

//Static initialization block:
static {

    //Retrieve the event XML file Event Factory Home instance from the
    //singleton instance of the Event Factory Context:
    EVENT_XML_FILE_EVENT_FACTORY_HOME = EventFactoryContext
        .getInstance()
        .getEventFactoryHome(
            "org.eclipse.hyades.logging.events.cbe.impl.EventXMLFileEventFactoryHomeImpl");

    //Resolve the Event Factory instance for this tutorial class
    //resolved from the event XML file Event Factory Home:
    eventXMLFileEventFactory = EVENT_XML_FILE_EVENT_FACTORY_HOME
        .getEventFactory(TUTORIAL_CLASS_NAME);
}

```

Section 5. Extending the Common Base Event factory

The extensibility of the Common Base Event factory provides the most effective mechanism for streamlining Common Base Event logging instrumentation. This section shows how to extend the Common Base Event factory's Event Factory Home and Content Handler to automatically populate Common Base Events with dynamic run time- and vendor-specific content from system-specific resources.

Creating a customized Event Factory Home

The Common Base Event factory provides a customizable and extensible architecture for creating and populating Common Base Events. Recall from the

preceding section ([Common Base Event factory](#)) that Event Factory Homes are tightly coupled with a specific type of configuration template used by a specific type of Content Handler to populate Common Base Events. The Event Factory Home resolves and reloads a typed Content Handler based on a typed configuration template from a locatable source. For example, the event XML file Event Factory Home resolves and reloads a Template Content Handler based on a well-formed XML document consisting of a single Common Base Event XML fragment with one or more defined properties as the configuration template. The Event Factory Home can be extended to resolve one or all of the following:

- Customized Content Handler
- Specific type of configuration template
- Specific location of configuration template

The Common Base Event factory provides an abstract implementation of the Event Factory Home, named

`org.eclipse.hyades.logging.events.cbe.impl.AbstractEventFactoryHome`. The abstract Event Factory Home is responsible for Event Factory instantiation and caching based on a unique factory name. Cached Event Factory instances can be updated (for example, by invoking the `updateEventFactory()` method) and released from the cache (for example, by invoking the `releaseEventFactory()` method) at any time during their life cycle. The abstract Event Factory Home provides the parent or *super* class for all customized Event Factory Homes. A customized Event Factory Home, namely the Log Manager Event Factory Home, extends the abstract Event Factory Home and implements the `resolveContentHandler()` and `createContentHandler(String)` methods. For customized Event Factory Homes that do not have an associated typed Content Handler, both of these methods should simply return `null`.

Resolving an uninitialized instance of a customized Content Handler

The `createContentHandler(String)` method calls the `resolveContentHandler()` method to resolve an uninitialized instance of the associated typed Content Handler for the customized Event Factory Home. This method permits extensions of existing Event Factory Homes implementations (for example, the event XML file Event Factory Home) for associating a customized Content Handler with the specific type and location of configuration template resolved by the parent or super Event Factory Home. Moreover, this method can be used to locate and resolve a Content Handler from a specific location, for instance, using the Java Naming and Directory Interface (JNDI). Conversely, for customized Event Factory Homes that do not resolve a customized Content Handler, this method should simply return an uninitialized instance of the Template Content Handler.

The customized Event Factory Home shown in this tutorial is associated with a typed

Content Handler, namely the run time Template Content Handler, as you'll see in [Creating a customized content handler](#). Listing 5 illustrates the customized Event Factory Home resolving an uninitialized instance of the customized Content Handler.

Listing 5. Resolving an uninitialized instance of the customized Content Handler

```
/**
 * @see org.eclipse.hyades.logging.events.cbe.impl.AbstractEventFactoryHome
 * #resolveContentHandler()
 */
public ContentHandler resolveContentHandler() {
    return (new RunTimeTemplateContentHandlerImpl());
}
```

The abstract Event Factory Home calls the `createContentHandler(String)` method to initialize the Content Handler returned from the `resolveContentHandler()` method. The Content Handler is initialized from a specific type of configuration template resolved by the customized Event Factory Home from a specific location. Specifically, this method:

- Resolves the location of the configuration template based on some identifiable marker (for example, the parameter Event Factory name)
- Retrieves its contents
- Configures the Content Handler with the contents of the configuration template using a common format (for example, Common Base Event object)

For example, the event XML file Event Factory Home's `createContentHandler(String)` method initializes a Template Content Handler returned from its `resolveContentHandler()` method based on a well-formed XML document consisting of a single Common Base Event XML fragment with one or more defined properties as the configuration template. The configuration template is resolved by searching for the named (`<Event Factory's name>.event.xml`) template XML file on the classpath. As you'll learn in [Creating a customized Content Handler](#), the Content Handler then can merge the contents of the configuration template into a Common Base Event, upon invoking the `complete()` method.

As you know, the core function of a customized Event Factory Home is to resolve a specific type of configuration template from a specific location. Configuration templates can consist of in-memory Common Base Event objects resolved using the JNDI and Lightweight Directory Access Protocol (LDAP) or Common Base Event XML documents or fragments resolved from a Web service, HTTP server, or relational database using Java DataBase Connectivity (JDBC).

One example is retrieving the configuration template from the system's logging facility. Often, systems using the Java Logging APIs configure their logging instrumentation using various key-value pairs in a logging configuration file. The `java.util.Properties` logging configuration file is loaded by the Java Logging's Log Manager and used to configure loggers, handlers, and the Log Manager. Localizing all related logging configuration, including configuration templates, in the system's logging configuration file increases serviceability and maintainability.

Adding the configuration template to the logging configuration file

For this tutorial, the template event XML file in [Listing 3](#) is added to the logging configuration file. The name of the template event XML file (for example, `com.ibm.tutorial.TutorialClass.event.xml`) is the name of the configuration entry. The single Common Base Event XML fragment in the XML document contained in the template event XML file is the value of the configuration entry. The customized Event Factory Home in this tutorial queries the Log Manager using the parameter Event Factory name and the `.event.xml` suffix as the name of the logging configuration property. The returned Common Base Event XML fragment is deserialized to a Common Base Event object, used to configure the customized Content Handler.

Recall that the event XML file Event Factory Home permits template event XML files for any logical grouping of the system (for example, component, package, or class name) by iterating all hierarchical Event Factory name segments until a configuration template is resolved or all name segments are attempted. The customized Event Factory Home in this tutorial provides analogous function by continuously querying the Log Manager with all of the hierarchical Event Factory name segments and the `.event.xml` suffix until the configuration entry is found or all name segments were attempted. [Listing 6](#) illustrates the customized Event Factory Home initializing a customized Content Handler instance for a named Event Factory based on the configuration template specified in the Log Manager configuration file.

Listing 6. Initializing a customized Content Handler instance for a named Event Factory based on the configuration template specified in the Log Manager configuration file

```
if ((factoryName != null) && (factoryName.trim().length() > 0)) {  
  
    String parentFactoryName = factoryName;  
    String propertyValue = null;  
    int dotIndex = -1;  
  
    while (true) {  
  
        //Resolve the configuration template property for this Event Factory  
        //from the Log Manager's properties:  
        propertyValue = LogManager.getLogManager().getProperty(  
            parentFactoryName.concat(".event.xml"));  
  
        if (propertyValue != null) {
```

```

try {
    CommonBaseEvent templateEvent = EventFormatter
        .eventFromCanonicalXML(propertyValue.trim());

    if (templateEvent != null) {

        TemplateContentHandler contentHandler =
            ((TemplateContentHandler) (resolveContentHandler()));

        contentHandler.setTemplateEvent(templateEvent);

        return contentHandler;
    }
} catch (FormattingException f) {
    //Ignore since the configuration template property for this Event
    //Factory is invalid so delegate to the parent factory:
}
}

//The configuration template property for this Event Factory cannot be
//loaded so delegate to the parent Event Factory:
dotIndex = parentFactoryName.lastIndexOf('.');

if (dotIndex != -1) {
    parentFactoryName = parentFactoryName.substring(0, dotIndex);
} else {
    break;
}
}
}
}

```

Creating a customized Content Handler

The Content Handler is the in-memory realization of the configuration template. It *merges* the properties defined in the associated configuration template into a Common Base Event. Recall that the Template Content Handler uses a Common Base Event instance as the configuration template to *complete* Common Base Events. However, the Content Handler can be extended to populate Common Base Events with dynamically resolved run time content.

Customized Content Handlers provide an effective mechanism for streamlining Common Base Event logging instrumentation by automatically populating newly created Common Base Events with dynamic problem-determination data, thereby drastically reducing the lines of instrumentation code, decreasing maintenance, and increasing customized, vendor-specific content. Customized Content Handlers can dynamically resolve and populate varying quantities and types of problem-determination data (for example, IP addresses, thread IDs and timestamps) specific to the particular run time instance of the system, event, or problem without requiring additional lines of code at the instrumentation site.

In this tutorial's example, the customized Event Factory Home is associated with a customized Content Handler, namely the run time Template Content Handler. The customized Content Handler is extended from the Template Content Handler, which

uses a Common Base Event instance as the configuration template to *complete* Common Base Events. The customized Content Handler overloads the Template Content Handler's `completeEvent(CommonBaseEvent)` method to populate the parameter Common Base Event with dynamically resolved run time content.

After confirming that parameter Common Base Event is not `null` and has not been previously *completed*, the `completeEvent(CommonBaseEvent)` method is invoked on the parent or *super* class to merge the Common Base Event properties defined in the configuration template into the parameter Common Base Event. This call should appear early in the customized Content Handler's overloaded `completeEvent(CommonBaseEvent)` method if the content in the configuration template has precedence over the properties populated by the Content Handler. Alternatively, the call can appear later in the customized Content Handler's overloaded `completeEvent(CommonBaseEvent)` method if the properties populated by the Content Handler have precedence over the content in the configuration template.

For the customized Content Handler, the content in the configuration template has precedence over the properties it populates. Because a Common Base Event can be populated by the Content Handler at any time during their life cycle, the customized Content Handler must respect existing properties set on the Common Base Event. Only unset and uninitialized Common Base Event properties should be set by the customized Content Handler.

Resolving dynamic problem-determination data

Next, the dynamic problem-determination data resolved at invocation time is set on the parameter Common Base Event. This data can be resolved at invocation time or retrieved from the Common Base Event logging property cache. For example, [Listing 7](#) illustrates the customized Content Handler populating dynamic problem-determination data resolved at invocation time and retrieved from the Common Base Event logging property cache.

Listing 7. Resolving and populating dynamic problem-determination data based on the system's runtime instance and the Common Base Event logging property cache

```
//If the global instance ID is not set, assign a new Globally Unique
//Identifier (GUID):
if (commonBaseEvent.getGlobalInstanceId() == null) {
    commonBaseEvent.setGlobalInstanceId(Guid.generate());
}

//If the creation time is not set, assign the current time:
if (!commonBaseEvent.isSetCreationTime()) {
    commonBaseEvent.setCreationTimeAsLong(System.currentTimeMillis());
}

//If the sequence number is not set, increment the running counter of all
//logged Common Base Event and assign the new value:
```

```

if (!commonBaseEvent.isSetSequenceNumber()) {
    commonBaseEvent
        .setSequenceNumber(++CommonBaseEventLoggingPropertyCache.COMMON_BASE_EVENT_COUNT);
}

//Retrieve the source component ID:
//Assumption: The source component ID is not null since it has been
//set in the configuration template.
ComponentIdentification sourceComponetId = commonBaseEvent.getSourceComponentId();

//If the thread ID is not set, assign the current thread's name:
if (sourceComponetId.getThreadId() == null) {
    sourceComponetId.setThreadId(Thread.currentThread().getName());
}

//If the location is not set, assign the local host's IP address and set
//the location type to 'IPV4':
if (sourceComponetId.getLocation() == null) {

    //Set the source component ID's location property to the local host's
    // IP
    //address:
    sourceComponetId
        .setLocation(CommonBaseEventLoggingPropertyCache.LOCAL_HOST_IP_ADDRESS);

    //Set the source component ID's locationType property to 'IPV4':
    //Assumption: The current IP address of the local host is an IPV4
    //address.
    sourceComponetId.setLocationType(ComponentIdentification.LOCATION_TYPE_IPV4);
}
}

```

Moreover, the customized Content Handler can resolve dynamic problem-determination data from the caller -- typically the logging instrumentation site. For example, [Listing 8](#) illustrates the customized Content Handler resolving and populating dynamic problem-determination data based on the logging instrumentation site.

Listing 8. Resolving and populating dynamic problem-determination data based on the logging instrumentation site

```

//Resolve the current call stack for resolving the caller's class name
//and method name:
//Note: This code handles the three possible call stacks specific to the
//tutorial class' programming model.
StackTraceElement[] stackTraceElements = new Throwable().getStackTrace();
int callersStackTraceElementIndex = 2;
String callersClassName = stackTraceElements[callersStackTraceElementIndex]
    .getClassName();

if (callersClassName
    .equals("org.eclipse.hyades.logging.events.cbe.impl.EventFactoryImpl")) {

    callersStackTraceElementIndex = 3;
    callersClassName = stackTraceElements[callersStackTraceElementIndex]
        .getClassName();

    if (callersClassName
        .equals("com.ibm.tutorial.logging.CommonBaseEventLoggingUtilities")) {

        callersStackTraceElementIndex = 4;
        callersClassName = stackTraceElements[callersStackTraceElementIndex]
            .getClassName();
    }
}

```

```
    }  
  }  
  
  //Retrieve the source component ID:  
  //Assumption: The source component ID is not null since it has been  
  //set in the configuration template.  
  ComponentIdentification sourceComponetId = commonBaseEvent.getSourceComponentId();  
  
  //If the component is not set, assign the caller's class name:  
  if (sourceComponetId.getComponent() == null) {  
    sourceComponetId.setComponent(callersClassName);  
  }  
  
  //If the sub-component is not set, assign the caller's method name:  
  if (sourceComponetId.getSubComponent() == null) {  
    sourceComponetId  
      .setSubComponent(stackTraceElements[callersStackTraceElementIndex]  
        .getMethodName());  
  }  
}
```

Section 6. Summary

Conclusion

The Log and Trace Analyzer uses problem-determination data in the Common Base Event format, providing a rich source of activity and control-flow information to accelerate problem determination and increase system serviceability and quality. This tutorial has detailed how to streamline Common Base Event logging in the Java language to reduce complexity and instrumentation code size in order to decrease maintenance and increase customized, vendor-specific content.

The tutorial:

- Reviewed the concept of problem determination and the Common Base Event model
- Reviewed the benefits and concerns with Common Base Event logging
- Streamlined Common Base Event logging instrumentation using the Java Logging APIs by leveraging the Common Base Event V1.0.1 Java implementation and support in the LTA
- Identified the essential best practices required to control and reduce the complexity of Common Base Event logging
- Extended the Common Base Event factory to reduce the Common Base Event logging instrumentation code size

Downloads

Description	Name	Size	Download method
Source code for the examples in this tutorial	Tutorial_Java_Project.zip	29KB	HTTP

[Information about download methods](#)

Resources

Learn

- The tutorial "[Understand problem determination](#)" (developerWorks, March 2004) describes the state of problem determination today, looks at available logging options, and shows how events can be recognized as part of larger situations.
- For an article detailing and explaining the Autonomic Computing Toolkit, see "[ABCs of the Autonomic Computing Toolkit](#)" (developerWorks, November 2004).
- The Common Base Event model specification and schema is provided in this Eclipse.org whitepaper: "[Canonical Situation Data Format: The Common Base Event V1.0.1.](#)"
- For an article detailing and explaining the Common Base Event model, see "[Standardize messages with the Common Base Event model](#)" (developerWorks, October 2004).
- The [W3C's XML Schema](#) Web site provides resources for understanding the Common Base Event schema.
- The Eclipse [Test and Performance Tools Platform Project Web site](#), formally Hyades, provides the Common Base Event V1.0.1 Java implementation API JavaDoc documentation for understanding the Java implementation of the Common Base Event V1.0.1 model.
- For a document detailing and explaining the Common Base Event factory in the Common Base Event V1.0.1 Java implementation, see "[Common Base Event Factory Design](#)" at the Eclipse Web site.
- The [Java Logging APIs](#) in Java 2 Platform, Standard Edition (J2SE) V1.4.0 and above provide an integrated and extensible logging facility for the Java programming language.
- "[Common Base Event logging](#)" (developerWorks, April 2005) shows how to instrument Common Base Event logging using the Java Logging APIs to provide a rich source of activity and control-flow information to accelerate problem determination and increase system serviceability and quality.
- For illustrative examples of the Java implementation of the Common Base Event V1.0.1 model and Common Base Event logging, see **File > New > Example... > Examples > Common Logging** and **File > New > Example... > Examples > Hyades Logging** in the Autonomic Computing Toolkit.

Get products and technologies

- The [Autonomic Computing Toolkit V3.0.0](#) offers the Log and Trace Analyzer for problem determination of complex heterogeneous systems.

- [Eclipse V3.0.3](#) is required by the [Autonomic Computing Toolkit V3.0.0](#).
- The [Java Runtime Environment \(JRE\) V1.4.x](#) is required by the [Autonomic Computing Toolkit V3.0.0](#).
- The Eclipse [Test and Performance Tools Platform Project Web site](#), formally Hyades, provides Java and C implementations of the Common Base Event V1.0.1 model.

About the author

Paul Slauenwhite



Paul Slauenwhite is a Software Developer on the IBM Autonomic Computing Tools and Technologies project and contributes to the Eclipse Test and Performance Tool Platform (TPTP) open source project (formally Hyades). After receiving a B.Sc. degree in computer science from Dalhousie University, Paul joined IBM in 2000 and worked on the WebSphere Object Level Trace (OLT) project. In 2001, he joined the IBM WebSphere Studio Team, where he developed logging and tracing technologies. Paul moved to the Eclipse Hyades project at its inception in 2002, focusing on log and trace data, collection, and correlation. Paul is currently working on an M.Math degree in software engineering at the University of Waterloo.