# org.jax.mgi.shr.dla.log

Author: Mike Walker

Created: March 20, 2003

Last Modified: February 28, 2005 13:56

## 1 Purpose of Document

This document describes the classes belonging to the org.jax.mgi.shr.dla.log package and provides source code examples for some common usage patterns.

## 2 Introduction

This package is comprised of classes that provide functionality for managing application logging in accordance with the Data Load Architecture (DLA) standards. This includes the handling of four standard log files, a process log, a curator log, a diagnostic log and a validation log (see DLA standards for more information about these logs). Also included in this package are classes designed for message formatting and for exception handling.

This package integrates with the lib_java_core product; namely, the log, config and exception packages.

You must be pointing to Java1.4 in your classpath when using this product.

## 3 Overview of Classes

The **DLALogger** class is a singleton class (one instance per JVM) which performs logging operations for a load application. An application can obtain multiple references to the **DLALogger**, but they all point to the same instance. It performs methods for logging messages to the process log, curator log, diagnostic log and validation log. It implements the **Logger** interface, so it can be used by both the application classes and by the frameworks classes.

The **DLALoggerFactory** is a class which implements the **LogFactory** interface and is used to create an implementation of the **Logger** interface (in this case, the **DLALogger**). This provides a way to “plug in” the **DLALogger** to the framework classes.

The **DLALoggerCfg** class is used to configure a **DLALogger** object. It reads parameters directly from the configuration files and java system properties and provides the **DLALogger** access to them.

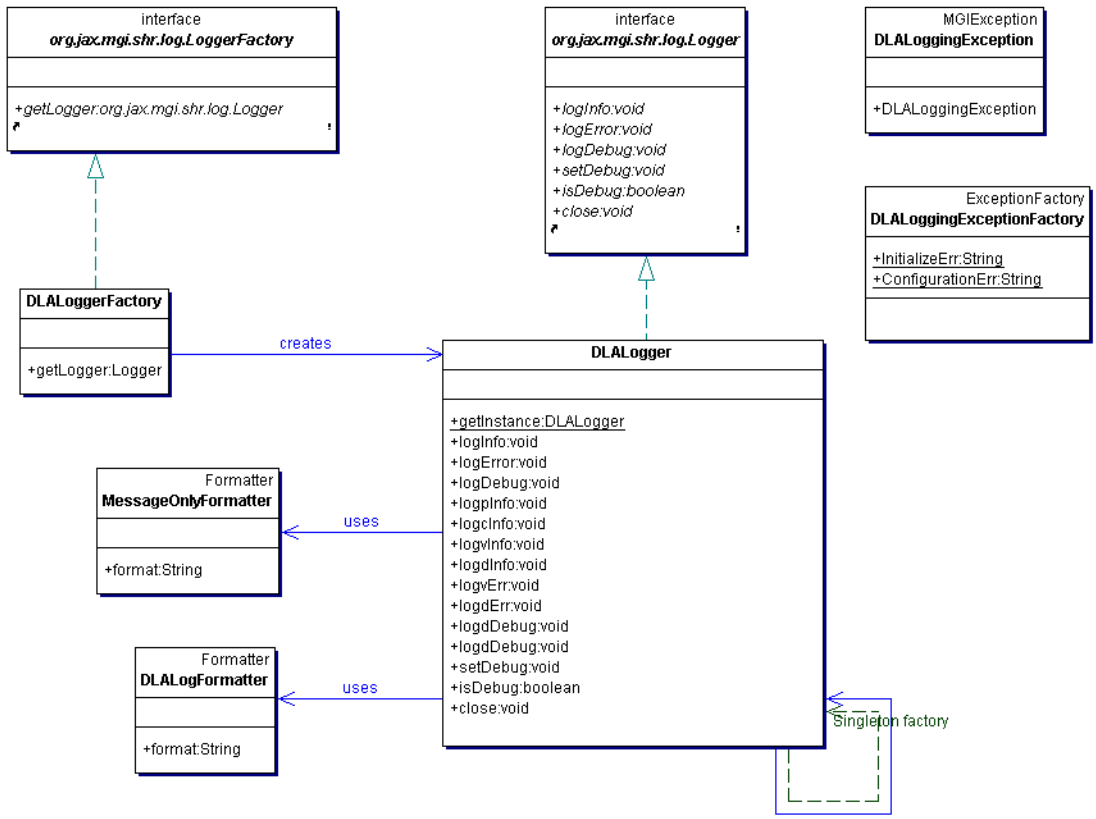
The **DLALogFormatter** class is used by the **DLALogger** to format log messages with time stamped headers. See Section 5 for more information.

The **MessageOnlyFormatter** class is used by the **DLALogger** to format messages for the log files which do not contain time stamped header information.

The **DLALoggingException** is a class which extends **MGIException** (see the documentation for the exception package from lib_java_core). It is used as the primary object for transporting logging exceptions throughout the system.

The **DLALoggingExceptionFactory** is a class which extends **MGIExceptionFactory** (see the documentation for the exception package from lib_java_core). It stores known **DLALoggingExceptions** and provides access to them by name.

# 4 Class Diagram




# 5 Configuration

Configuration is accomplished by use of the org.jax.mgi.shr.config package from the lib_java_core product. See the documentation for this package for more information about the config base classes. A configurator class, DLALoggerCfg, was created for managing the configuration of the logger classes. Because of protected access to methods within the config package that are required by the configurator class, DLALoggerCfg must be a member of the org.jax.mgi.shr.config package. Details of how the configurator classes work can be found in the config documentation, but in brief, the configuration parameters are read in from configuration

files and the java system properties. The configuration file names are indicated on the java command line by use of the CONFIG system property (see example 1).

**EXAMPLE 1. setting the configuration**

```
java -DCONFIG=configfile -DPARM1=OVERRIDE -DPARM2=OVERRIDE2 <app>
```

The configuration parameters are as follows:

**LOG_PROC**

The name of the process log. The default is derived by appending the string ".proc.log" to the value of the LOG_DEFAULTNAME parameter.

**LOG_DIAG**

The name of the diagnostic log. The default is derived by appending the string ".diag.log" to the value of the LOG_DEFAULTNAME parameter

**LOG_CUR**

The name of the curator log. The default is derived by appending the string ".cur.log" to the value of the LOG_DEFAULTNAME parameter.

**LOG_VAL**

The name of the validation log. The default is derived by appending the string ".diag.log" to the value of the LOG_DEFAULTNAME parameter.

**LOG_DEFAULTNAME**

The default name of any unnamed log file. If neither the curator, diagnostic, validation or process log have not been configured then this value is used as a base name to derive the corresponding log name. The default value is "dataLoad". For example if neither the LOG_PROC or this parameter is set, then the process log will be called "dataLoad.proc.log". Furthermore, if this value is configured to be "seqloader" and the LOG_PROC parameter is unset, then the process log will be called "seqloader.proc.log".

**LOG_PATH**

The path name of the directory where the process, curator, diagnostic and validation log will be created.

**LOG_DEBUG**

This parameter takes a boolean value (‘1’, ‘0’, ‘true’, ‘false’, ‘yes’, ‘no’) and determines whether debug messages should be enetered into the log file. The default is false.

# 6 Severity Levels and Header Stamping

A message can have one of three levels of severity. These are informational, debug, and error. Informational messages can be logged to any of the four logs and can be stamped with a conventional header or not. The conventional header includes more information about the message including date/time, severity level, class and method name of the caller. See Example 2 for a sample of the header format.

**EXAMPLE 2. Conventional Message Header**

```
May 25, 2002 5:46:02 PM org.jax.mgi.app.rpciload interpret
ERROR: Badly formatted record from rpci_clones.dat, id=RPC23-345C23
```

All the logging methods are summarized in table 2.

**Table 1: Logging method summary**

| method | log file | header stamp | severity |
|---|---|---|---|
| logpInfo(String, boolean) | process | depends on boolean | I |
| logcInfo(String, boolean) | curator | depends on boolean | I |
| logvInfo(String, boolean) | validation | depends on boolean | I |
| logdInfo(String, boolean) | diagnostic | depends on boolean | I |
| logdDebug(String, boolean) | diagnostic | depends on boolean | D |
| logvErr(String) | validation | yes | E |
| logdErr(String) | diagnostic | yes | E |
| logdDebug(String) | diagnostic | yes | D |

```
I=Informational, E-Error, D=Debug
```

Debug messages are used for testing and debugging applications. The user can add debug statements throughout the code and have them displayed in the log by calling the setDebug() method on the **DLALogger** with a value of true or setting the configuration parameter LOG_DEBUG=true. The default is false.

The following examples illustrate a couple of patterns of usage for the logging methods.

**EXAMPLE 3. Logging Informational Messages**

```
// log informational message to the process log
logger.logpInfo('Beginning rpci load initialization', true);
```

**EXAMPLE 4. Logging Debug Messages**

```
// log debug message to the diagnostic log
logger.logdDebug('Entering data validation code for id = ' + seqid);
```

one since all methods are static. The handleException() method takes an MGIException object as a parameter. See Example 6.