University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

How to Use Logging in Evolizer Core

Overview and Examples

1 Overview

EVOLIZERCORE provides sophisticated logging facilities. They are based on log4j[1], but provide fine-grained configuration possibilities on a per-plug-in basis. For example, it is possible to configure logging to write messages to console, to Eclipse's `ILog`, and/or to a file by simply editing `log4j.properties` without the need to change any code at all. This document describes how to use and configure logging in plug-ins which are based on the EVOLIZER-plattform in an appropriate way.

[1] http://logging.apache.org/log4j/

2 Using Logging in a Plug-In

This section describes the steps that are necessary to use logging in an EVOLIZERCORE-context.

2.1 Implementation Details

To use the logging facilities provided by EVOLIZERCORE, some prerequisites have to be met. First of all, some plug-in dependencies must be fulfilled. Implementors require the following two modules:

- org.evolizer.util.logging
- org.apache.junit

The latest revisions of both of the plug-ins are located in Haydn's svn repository[2] and can be added to the **Required Plug-ins**-list under the **Dependencies**-tab of the PDE-view or by editing the plug-in's manifest. Either way, the following lines need to be added to MANIFEST.MF:

```
Require-Bundle: org.apache.log4j,
 org.evolizer.util.logging
```

Listing 1: META-INF/MANIFEST.MF

Second, a log4j-configuration (see Section 3 for details) must be provided. Third, the plug-in's `Activator`-class has to be enhanced by additional code (note that some statements, that are not directly related to logging, were left out):

[2] https://haydn.ifi.unizh.ch/svn/evolizer_ng/core/trunk/

```
public class Activator extends Plugin {
   static final String LOG_PROPERTIES_FILE = "config/log4j.properties";
   PluginLogManager fLogManager;

   public void start(BundleContext context) throws Exception {
      super.start(context);
      configure();
   }

   public void stop(BundleContext context) throws Exception {
      Activator.plugin = null;
      if (fLogManager != null) {
         fLogManager.shutdown();
         fLogManager = null;
      }
      super.stop(context);
   }

   public static InputStream openBundledFile(String filePath)
                                 throws IOException {
         return Activator.getDefault().getBundle()
                              .getEntry(filePath).openStream();
   }

   public static PluginLogManager getLogManager() {
      return getDefault().fLogManager;
   }

   private void configure() {
      try {
         InputStream propertiesInputStream
                  = openBundledFile(LOG_PROPERTIES_FILE);
         if (propertiesInputStream != null) {
            Properties props = new Properties();
            props.load(propertiesInputStream);
            propertiesInputStream.close();
            fLogManager = new PluginLogManager(this, props);
         }
         propertiesInputStream.close();
      } catch (Exception e) { /*Exception-handling*/ }
   }
}
```

Listing 2: org.evolizer.foo.Activator – Establishing logging for a custom plug-in

Clients can now retrieve a logger-instance from the `Activator` by invoking the following code:

```
public class Bar {
   Logger logger
      = Activator.getLogManager().getLogger(Bar.class.getName());
}
```

Listing 3: org.evolizer.foo.Bar – Getting a logger-instance

Note that each plug-in receives its own logger hierarchy and configuration. This opens the possibility to turn of logging of one plug-in completely, while *e.g.,* printing detailed logging information of a second plug-in to console. This is very convenient, since it reduces information overload significantly.

To add a message to the log, several methods can be used, depending on the severity of the logging cause:

```
public class Bar {
   ...
   void doSomething(){
      logger.debug("this is a debugging statement");
      logger.warn("this is a warning statement");
      logger.info("this is an info statement");
      logger.error("this is an error statement");
   }
   ...
}
```

Listing 4: org.evolizer.foo.Bar – Logging

The next section discusses the circumstances under which a certain logging level may be appropriate.

2.2 Guidelines - Under construction

This section defines logging conventions with the following goals in mind:

- User and developer friendly logging
- Expressiveness of logging statements
- Uniformity of logger use

Not every log statement that is interesting for a developer might be useful for the user. Severity levels like debug, warn, info, and error provide a convenient way to handle this issue:

1. Use `logger.debug()` to log information that is not usefull for the user, but only for the developer.
2. Use `logger.warn()` and `logger.info()` for information that the user should be noticed of. Use these methods wisely, since they will be sent *e.g.,* to Eclipse's log and might clutter the Log View.
3. Use `logger.error()` for example when an exception occurs and the normal program path cannot be continued.

3 Configuration Example

Configuration can be done without changing a single line of code. Instead, each plug-in needs a folder `config/` containing a file `log4j.properties`. The following listing shows an example configuration:

```
# Set root logger level to [LEVEL] and its appender to [APPENDER].
# Suggested levels are: DEBUG, WARN, INFO, ERROR, OFF.
log4j.rootLogger=DEBUG, A1, A2, R1

#Plug-in-specific loggers
log4j.logger.org.evolizer.foo.Bar=ERROR, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A2 is set to be a EclipseLogAppender
log4j.appender.A2=org.evolizer.util.logging.EclipseLogAppender
log4j.appender.A2.verbose=true
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%p %t %c - %m%n

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n

# R1 is set to be a RollingFileAppender
log4j.appender.R1=org.apache.log4j.RollingFileAppender

log4j.appender.R1.File=myfile.log
log4j.appender.R1.MaxFileSize=10MB
# Keep one backup file
log4j.appender.R1.MaxBackupIndex=100

log4j.appender.R1.layout=org.apache.log4j.PatternLayout
log4j.appender.R1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
}
```

Listing 5: log4j.properties – Configuration

The properties listed above configure the logging system to behave as follows:

1. All log messages will be sent to the appenders, since the logger level is set to `DEBUG` – except for the class `org.evolizer.foo.Bar`, whose logger level is set to `ERROR`.
2. All log statements are written to console (appender A1), to the Eclipse log (appender A2), and to a file called `myfile.log`. Again, `org.evolizer.foo.Bar` is an exception since its errors are only sent to console.

This configuration will lead to many entries in Eclipse's log, which is very likely to be undesired behavior. Changing

```
log4j.appender.A2.verbose=true
```

to

```
log4j.appender.A2.verbose=false
```

takes care of this issue by telling the EclipseLogAppender to handle only errors. See the log4j-documentation for details on appenders and layouts.