Kurze Einführung in log4j

Version 1.4
09.03.2006

Dirk Schnelle `dirk.schnelle@web.de`

Dieses Dokument beschreibt die log4j API, ihre besonderen Merkmale und Design Grundlagen. Log4j ist ein Open Source Projekt und umfasst die Arbeit vieler Entwickler. Es gibt dem Entwickler die Möglichkeit auf die Art und Granularität der Loggingausgaben Einfluss zunehmen. Es ist zur Laufzeit voll konfigurierbar, da es auf externe Konfigurationsdateien aufsetzt. Dabei ermöglicht es einen einfachen Einstieg und hat nach oben hin wenig Grenzen.

Copyright ©2004-2006 Dirk Schnelle

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Wenn Sie beim Lesen dieses Tutorials auf Fehler stoßen oder aber Stellen auffallen, die nicht ausführlich genug oder *nur fast richtig* beschrieben wurden, können Sie mich gern über `dirk.schnelle@web.de` kontaktieren.

Danksagungen

Danke an Ceki Gülcü. Dieses Tutorial hat seine Wurzeln in seiner Dokumentation und ist im Wesentlichen eine Übersetzung seines Artikels. Der Originalartikel kann auf [4] nachgelesen werden.
Vielen Dank auch an all die, die sich die Mühe gemacht haben und mich auf Fehler in diesem Tutorial aufmerksam gemacht haben.

Inhaltsverzeichnis

1 Einführung **3**

2 Installation **4**

3 Logger, Appender und Layouts **4**
3.1 Logger Hierarchie 4

4 Appender und Layouts **8**

5 Konfiguration **10**

6 Standard Initialisierungsverfahren **15**

7 Beispiel Konfigurationen **16**
7.1 Standard Initialisierung bei Tomcat 16
7.1.1 Beispiel 1 16
7.1.2 Beispiel 2 16
7.1.3 Beispiel 3 17
7.1.4 Beispiel 4 17
7.1.5 Initialisierung durch ein Servlet 17

8 Nested Dialog Kontexte **19**

9 Performanz **20**
9.1 Logging Performanz bei deaktiviertem Logging 20
9.2 Performanz der Entscheidung ob ein Logger aktiv ist 21
9.3 Performanz der Logausgabe 21

10 SUNs Logging API **22**

11 Zusammenfassung **22**

A Konvertierungsmuster **25**

1 Einführung

My software never has bugs,
it just develops random features.

Fast jede große Anwendung bringt ihre eigene Logging oder Tracing API mit. Gemäß dieser Regel verfuhr Anfang 1996 auch das E.U. SEMPER Projekt [13]. Nach unzähligen Umstellungen und Erweiterungen entstand aus dieser API log4j [9], eine beliebte Logging Bibliothek für Java. Die Bibliothek wird unter der Apache Software License [2] verteilt. Diese Lizenz ist durch die Open Source Initiative [10] zertifiziert. Mittlerweile wurde log4j in andere Sprachen, wie C,C++, C#, Perl, Python, Ruby und Eiffel übertragen. Aktuell wird versucht diese unter einem Top Level Projekt bei Apache [1] zu vereinheitlichen.

Das Einfügen von Logging Statements ist eine low-tech Methode, um den Quellcode zu Debuggen. Manchmal ist dieses auch der einzige Weg, weil z.B. Debugger nicht verfügbar sind, oder nicht eingesetzt werden können. In der Regel ist dieses bei Anwendungen der Fall, die viele Threads nutzen oder bei großen verteilten Anwendungen.

Die Erfahrung zeigt, das Logging ein wichtiger Faktor im Entwicklungszyklus ist. Logging bietet einige Vorteile. Es beinhaltet den genauen *Kontext* eines Durchlaufs der Anwendung. Wurden die Logging Statements erst einmal in den Quellcode eingefügt, ist für die Ausgabe des Loggings kein weiteres Eingreifen nötig. Darüber hinaus kann die Ausgabe auf einem persistenten Medium gespeichert werden und kann somit auch zu einem späteren Zeitpunkt genau untersucht werden. Weiterhin kann ausreichendes Logging auch als Audit Instrument eingesetzt werden.

Brian W. Kernighan und Rob Pike haben in ihrem ausgezeichneten Buch [8] geschrieben:

> As personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugging sessions are transient.

Logging hat aber auch seine Nachteile. Es kann die Performanz einer Anwendung verringern. Wenn es zu exzessiv eingesetzt wird, kann das Inspizieren der Ausgabe zu einer wahren Scrolling Orgie ausarten, bei der

die wesentlichen Punkte nicht mehr ins Auge fallen. Da Logging kaum die Hauptaufgabe einer Anwendung ist, wurde die log4j API bewusst so gehalten, dass sie einfach zu verstehen und zu benutzen ist.

2 Installation

Die aktuelle Version, einschließlich dem vollständigen Quellcode und Dokumentation kann von `http://logging.apache.org/log4j` frei herunter geladen werden.

Um log4j zu installieren, genügt es, das heruntergeladene *tar.gz* oder *zip* Archiv in ein geeignetes Verzeichnis zu entpacken, z.B. */usr/local*. Das Archiv entpackt sich hierbei in den Ordner *jakarta-log4j-VERSION*.

Im Unterverzeichnis *dist/lib* befindet sich ein JAR-Archiv, *log4j-VERSION.jar*. *VERSION* ist hierbei durch die Versionsnummer der heruntergeladenen Version zu ersetzen. Dieses JAR-Archiv muss zum Übersetzen und Ausführen ihrer Programme, die log4j benutzen, in den *CLASSPATH* aufgenommen werden.

In früheren Versionen gab es dort zwei JAR-Archive, die sich im Umfang unterschieden. Eine davon hatte die Endung *-core* und beinhaltete nur die wichtigsten Elemente, war dafür aber auch kleiner.

Eine kurze Beschreibung der Installation, inklusive einer Anleitung zum Testen, ob die Installation geglückt ist, findet sich auch in der Datei *INSTALL*.

3 Logger, Appender und Layouts

Log4j hat drei Hauptkomponenten: *Logger*, *Appender* und *Layouts*. Diese drei arbeiten zusammen, und ermöglichen den Entwicklern Logmeldungen entsprechend ihrem Typ und Level in ihren Quellcode zu integrieren und zur Laufzeit zu kontrollieren, wie diese Logmeldungen formatiert und wo sie ausgegeben werden.

3.1 Logger Hierarchie

Der erste und wichtigste Vorteil jeder Logging API gegenüber einem einfachen `System.out.println` besteht in der Möglichkeit einige Logging Statements auszublenden, während andere ungehindert ausgegeben werden. Das setzt aber voraus, das die Entwickler sich die Mühe gemacht haben, ihre Logmeldungen auf irgendeine Art und Weise zu Kategorisieren. Aus diesem Grund wurde auch *category* ein zentrales Konzept von log4j. Trotzdem wurde seit log4j Version 1.2 die Klasse *Category* durch die Klasse *Logger* ersetzt. Für die, die frühen Versionen von log4j kennen, kann die *Logger*-Klasse als ein Alias für die *Category*-Klasse betrachtet werden.

Logger sind *named entities*. Die Namen von Loggern unterscheiden Groß- und Kleinschreibung und sind durch ihre Namen hierarchisch geordnet.

Definition 1 *Ein Logger ist ein **Vorfahre** eines anderen Loggers, wenn sein Name, gefolgt von einem Punkt der Präfix des Namens eines **Kind** Loggers ist. Ein Logger ist ein **Vater** eines **Kind** Loggers, wenn es zwischen ihm und dem Nachkomme Logger keine weiteren Vorfahren gibt.*

Zum Beispiel ist ein Logger mit dem Namen *com.foo* der Vater eines Loggers mit dem Namen *com.foo.Bar*. Entsprechend ist *java* ein Vater von *java.util*. Dieses Namenschema ist den meisten Entwicklern vertraut.

An der Spitze der Hierarchie steht der *root*-Logger. Er ist in zweifacher Hinsicht eine Ausnahme:

1. Ihn gibt es immer

2. Man kann ihn nicht durch seinen Namen bestimmen

Der *root*-Logger kann nur durch die statische Methode *Logger.getRoot-Logger* bestimmt werden. Alle anderen Logger werden durch die statische *Logger.getLogger* Methode instantiiert und bestimmt. Diese Methode hat den Namen des gesuchten Loggers als Argument. Einige der Hauptmethoden sind im Folgenden aufgeführt:

```
1   public class Logger {
        // Methoden zur Instantiierung und zum Holen
        // einer Referenz:
        public static Logger getRootLogger();
        public static Logger getLogger(String name);
6
        // Ausgabe Methoden:
        public void debug(Object message);
        public void info(Object message);
        public void warn(Object message);
11      public void error(Object message);
        public void fatal(Object message);

        // Generische Ausgabe Methode:
        public void log(Level l, Object message);
16  }
```

Loggern kann ein Level zugewiesen werden. Die möglichen Level *DEBUG*, *INFO*, *WARN*,*ERROR* und *FATAL* sind in der Klasse *org.apache.log4j.Level* definiert. Es ist zwar nicht zu empfehlen, aber es ist möglich, eigene Level einzuführen, indem man eine Klasse einführt, die von der Klasse *Level* erbt.

Wenn einem Logger kein Level zugeordnet wurde, erbt er den Level des nächsten Vorfahren, dem ein Level zugewiesen wurde. Etwas formaler:

Tabelle 1: Beispiel 1 für Vererbung von Leveln

Logger Name	**Zugewiesener Level**	**Ererbter Level**
root	Proot	Proot
X	keiner	Proot
X.Y	keiner	Proot
X.Y.Z	keiner	Proot

Tabelle 2: Beispiel 2 für Vererbung von Leveln

Logger Name	**Zugewiesener Level**	**Ererbter Level**
root	Proot	Proot
X	Px	Px
X.Y	Pxy	Pxy
X.Y.Z	Pxyz	Pxyz

Tabelle 3: Beispiel 3 für Vererbung von Leveln

Logger Name	**Zugewiesener Level**	**Ererbter Level**
root	Proot	Proot
X	Px	Px
X.Y	keiner	Px
X.Y.Z	Pxyz	Pxyz

Definition 2 *Der* ***ererbte Level*** *eines gegebenen Loggers C entspricht dem ersten Level in der Logger Hierarchie der nicht* null *ist, wobei bei C begonnen wird und zum* root-*Logger in der Hierarchie aufgestiegen wird.*

Damit das auch immer funktioniert und ein Logger immer einen Level erben kann, ist dem *root*-Logger immer ein Level zugewiesen. In den Tabellen 1 bis 4 wird dieses an einigen Beispielen gezeigt.

In Beispiel 1 aus Tabelle 1 wurde nur dem *root*-Logger ein Level zugewiesen. Dessen Level *Proot* wird von allen anderen Loggern X, $X.Y$ und $X.Y.Z$ geerbt.

In Beispiel 2 aus Tabelle 2 wurde jedem Logger ein Level zugewiesen. Deswegen erbt auch kein Logger einen Level.

In Beispiel 3 aus Tabelle 3 wurde nur den Loggern *root*, X und $X.Y.Z$ ein Level zugewiesen. Der Logger $X.Y$ erbt seinen Level von seinem Vater X.

Tabelle 4: Beispiel 4 für Vererbung von Leveln

Logger Name	**Zugewiesener Level**	**Ererbter Level**
root	Proot	Proot
X	Px	Px
X.Y	keiner	Px
X.Y.Z	keiner	Px

In Beispiel 4 aus Tabelle 4 haben nur die Logger *root* und *X* einen zugewiesenen Level. Die Logger *X.Y* und *X.Y.Z* erben ihre Level von ihren Vorfahren.

Die Logmeldungen werden durch die Ausgabe-Methoden der Logger Instanzen ausgegeben. Diese Ausgabe-Methoden sind *debug*, *info*, *warn*,*error*, *fatal* und *log*.

Per Definition bestimmt die aufgerufene Methode den Level der Logmeldung. Ist zum Beispiel *c* eine Logger Instanz, dann soll durch das Statement *c.info("...")* eine Logmeldung des Levels *INFO* ausgegeben werden.

Die Ausgabe einer Logmeldung ist *aktiv*, wenn ihr Level höher oder gleich dem Level ihres Loggers ist. Andernfalls ist sie *inaktiv*. Etwas formaler:

Definition 3 *Eine Logmeldung mit dem Level P eines Logger mit dem (zugewiesenen oder ererbten) Level q ist* ***aktiv****, wenn gilt $p \geq q$.*

Diese Regel bildet den Kern von log4. Für die Standardlevel gilt:

$$\texttt{DEBUG} < \texttt{INFO} < \texttt{WARN} < \texttt{ERROR} < \texttt{FATAL} \quad (1)$$

Um all diese formalen Definitionen etwas verständlicher zu machen, soll die Anwendung dieser Regeln an einem Beispiel vorgeführt werden.

```
// Hole eine Logger Instanz mit dem Name "com.foo"
Logger logger = Logger.getLogger("com.foo");

// Setzen des Levels. Das ist nicht die uebliche
// Vorgehensweise.
// Normalerweise wird der Level durch
// Konfigurationsdateien gesetzt und nicht durch
// das Programm.
logger.setLevel(Level.INFO);

Logger barlogger = Logger.getLogger("com.foo.Bar");

// Diese Logmeldung ist aktiv, da WARN >= INFO.
logger.warn("Low fuel level.");
```

```
// Dies Logmeldung ist inaktiv, da DEBUG < INFO.
logger.debug("Starting search for nearest station.");

// Die Instanz des Loggers barlogger mit dem Namen
// "com.foo.Bar" erbt ihren Level vom Logger
// "com.foo". Die folgende Logmeldung ist deswegen
// aktiv INFO >= INFO.
barlogger.info("Located nearest gas station.");

// Diese Logmeldung ist inaktiv,da DEBUG < INFO.
barlogger.debug("Exiting gas station search");
```

Der Aufruf der Methode `getLogger` liefert bei jedem Aufruf für den gleichen Namen das gleiche Logger Objekt.

Zum Beispiel:

```
Logger x = Logger.getLogger("wombat");
Logger y = Logger.getLogger("wombat");
```

x und y referenzieren *genau* das gleiche Logger Objekt.

Hierdurch ist es möglich, einen Logger zu konfigurieren und dessen Instanz irgendwo im Code zu holen und zu verwenden, ohne irgendwelche Referenzen mitschleifen und übergeben zu müssen. Ein grundsätzlicher Unterschied zur biologischen Vater-Kind Beziehung, bei der immer erst ein Vater da sein muss, bevor es ein Kind geben kann, können Logger in beliebiger Reihenfolge erzeugt und auch konfiguriert werden. Ein Vater Logger wird immer sein Kind und sonstige Nachkommen finden, auch wenn diese vor ihm erzeugt wurden.

Die Konfiguration der log4j-Umgebung erfolgt üblicherweise bei der Initialisierung der Anwendung. Dabei ist die Konfiguration durch eine Konfigurationsdatei der übliche Weg. Dieser Ansatz soll kurz umrissen werden.

Durch log4j ist es relativ einfach Logger nach einer *Software Komponente* zu benennen. Das ist eine nützliche und direkte Methode um Logger zu definieren. Da die Logmeldung den Namen des verwendeten Loggers kennt, ist es dann relativ einfach, die Quelle der Logmeldung zu bestimmen. Wie auch immer ist das nur eine Möglichkeit die Benennung von Loggern zu strukturieren. Log4j macht hinsichtlich der Strukturierung keine Einschränkungen. Der Nutzer kann seine Logger so benennen und strukturieren, wie es ihm gefällt. Die Benennung eines Loggers nach der Klasse in der er verwendet wird scheint bis jetzt jedoch die beste Strategie von allen zu sein.

4 Appender und Layouts

Die Möglichkeit, einzelne Logger beliebig zu aktivieren oder deaktivieren ist nur ein Teil der Möglichkeiten. Log4j erlaubt die parallele Ausgabe der Logmeldungen auf verschiedene Ausgabemedien. In der Sprache von log4j wird dies *appender* genannt. Aktuell gibt es Appender für die Konsole,

Tabelle 5: Beispiel für Additive Appender

Logger Name	**Appender**	**add. flag**	**Ausgabe**
root	A1		A1
x	A-x1, A-x2	true	A1, A-x1, A-x2
x.y	keiner	true	A1, A-x1, A-x2
x.y.z	A-xyz1	true	A1, A-x1, A-x2, A-xyz1
security	A-sec	false	A-sec
security.access	keiner	true	A-sec

Dateien, GUI Komponenten, Sockets, JMS, NT Event Logger UNIX Syslog daemons. Es ist sogar möglich asynchron zu loggen.

Jedem Logger können mehrere Appender zugewiesen werden.

Die Zuweisung eines Appenders zu einem Logger erfolgt durch die Methode *addAppender*. Jede Logmeldung wird sowohl an die Appender des verwendeten Loggers als auch an die Appender weitergeleitet, die in der Hierarchie über dem Logger stehen. Das bedeutet, das Logger weiter vererbt werden, indem sie zu den Appendern der Nachkommen hinzugefügt werden. Wird zum Beispiel dem *root* Logger ein Konsolen Appender zugewiesen werden alle aktiven Logmeldungen auf der Konsole ausgegeben. Wird zudem für einen Logger *C* ein File Appender zugefügt, dann erfolgt die Ausgabe von *C* und seinen Nachkommen in die Datei und auf der Konsole. Dieses Standardverhalten kann über die Methode *setAdditivity* auch abgeschaltet werden, indem das *additivity flag* auf *false* gesetzt wird.

Definition 4 *Die Ausgabe der Logmeldungen des Loggers* C *werden an alle Appender von* C *und seinen Nachkommen weitergeleitet. Dieses Verhalten wird durch den Ausdruck* ***Additive Appender*** *beschrieben.*

Ein Beispiel zeigt Tabelle 5.

Manchmal kommen Benutzer auch auf die Idee, neben dem Ausgabemedium auch das Ausgabeformat beeinflussen zu wollen. Hierzu wird einem Appender ein *Layout* zugewiesen. Das Layout ist für die Formatierung der Logmeldung zuständig während ein Appender dafür zuständig ist, auf welches Ausgabemedium die Logmeldung gesendet wird. Mit dem *PatternLayout*, einem Bestandteil der Standard log4j Distribution, kann der Benutzer das Ausgabeformat mit Konvertierungsmustern, ähnlich denen der *printf* Funktion der Programmiersprache C, angeben. Eine Übersicht über die Konvertierungsmuster findet sich in Anhang A.

Zum Beispiel gibt das Konvertierungsmuster

```
%r [%t ] %-5p %c-%m%n
```

eine Logmeldung etwa folgendermaßen aus:

```
176 [main] INFO org.foo.Bar-Locate nearest gas station
```

Das erste Feld ist die Anzahl Millisekunden, die seit dem Programmstart verstrichen sind. Das zweite Feld ist der Thread, in dem die Logmeldung ausgegeben wurde. Das dritte Feld ist der Level der Logmeldung. Das vierte Feld ist der Name des Loggers und der Text nach dem Bindestrich ist der Text der Logmeldung.

Die Logmeldungen können entsprechend den Vorgaben des Benutzers gerendert werden. Nimmt man zum Beispiel an, das Orangen Früchte sind und ein *FruitRenderer* registriert wurde, werden alle Früchte, einschließlich Orangen mit dem *FruitRenderer* gerendert. Es sei denn, es wird ein spezieller *OrangeRenderer* registriert.

Renderer müssen das *ObjectRenderer*-Interface implementieren.

5 Konfiguration

Das Einfügen von Logmeldungen in den Code einer Anwendung erfordert einiges an Planung und Aufwand. Beobachtungen haben gezeigt, dass Logging etwa 4 Prozent des Gesamtcodes ausmacht. Das bedeutet, dass selbst mittelgroße Anwendungen schnell auf einige Tausend Logmeldungen kommen können. Wenn man diese Zahl vor Augen hat, kann man sich leicht vorstellen, dass diese unmöglich alle per Hand gepflegt werden können.

Die log4j Umgebung ist aus dem Programm heraus voll konfigurierbar. Dennoch ist es weitaus flexibler, log4j über Konfigurationsdateien zu steuern. Aktuell können Konfigurationsdateien im XML oder Java properties[1] Format geschrieben werden.

Um einen Geschmack davon zu bekommen, wie so etwas aussehen kann, stellen wir uns vor, wir hätten eine imaginäre Anwendung MyApp, die log4j einsetzt.

```
import com.foo.Bar;

// Importieren der benoetigten log4j Klassen.
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class MyApp {
    // Definition eines statischen Loggers, der
    // die Logger Instanz mit dem Namen "MyApp"
    // referenziert.
    static Logger logger =
        Logger.getLogger(MyApp.class);

    public static void main(String[] args) {
        // Aufsetzen einer einfachen Konfiguration,
```

[1] Key-Value Paare im Format key=value

```
        // die Logmeldungen auf der Konsole ausgibt.
        BasicConfigurator.configure();

        logger.info("Entering application.");
        Bar bar = new Bar();
        bar.doIt();
        logger.info("Exiting application.");
    }
}
```

MyApp beginnt damit, dass es die benötigten log4j Klassen importiert. Danach wird ein statischer Logger mit dem voll qualifizierten Namen der Klasse*MyApp* definiert.

MyApp benutzt die Klasse *Bar* aus dem Paket *com.foo*.

```
package com.foo;

import org.apache.log4j.Logger;

public class Bar {
    static Logger logger =
        Logger.getLogger(Bar.class);

    public void doIt() {
       logger.debug("Did it again!");
    }
}
```

Der Aufruf von *BasicConfigurator.configure* erzeugt eine einfache log4j Umgebung. Hierbei ist der *root* Logger als *ConsoleAppender* hart verdrahtet. Die Ausgabe verwendet ein *PatternLayout* mit dem Konvertierungsmuster

```
%-4r [%t] %-5p %c %x - %m%n
```

Dem *root* logger wird der Level *DEBUG* zugewiesen.

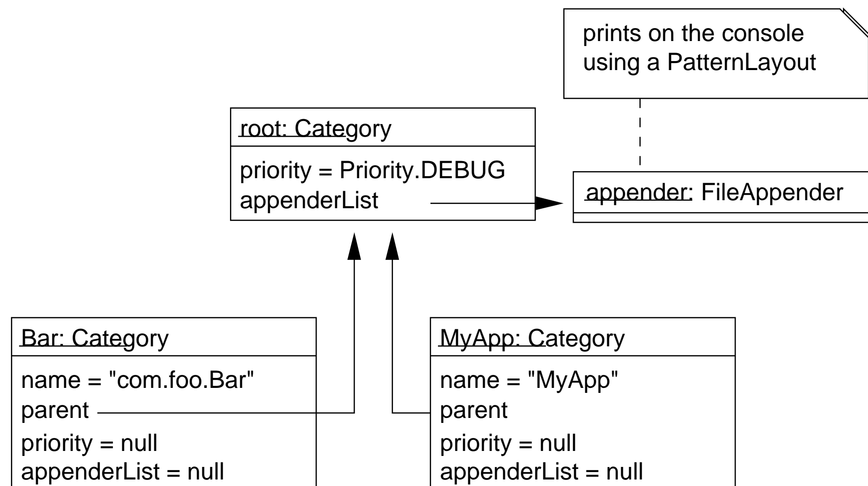
Die Ausgabe beim starten von *MyApp* sieht so aus:

```
0    [main] INFO  MyApp  - Entering application.
36   [main] DEBUG com.foo.Bar  - Did it again!
51   [main] INFO  MyApp  - Exiting application.
```

Abbildung 1 zeigt das Objektdiagramm von *MyApp* nach dem Aufruf der Methode *BasicConfigurator.configure*.

Eine kleine Zwischenbemerkung: Die Kinder von log4j Loggern verbinden sich nur mit ihren existierenden Vorfahren. So verbindet sich der Logger *com.foo.Bar* direkt mit dem *root*-Logger und umgeht die nicht benutzten Logger *com* und *com.foo*. Hierdurch wird sowohl die Performanz verbessert, als auch der Speicherbedarf gering gehalten.

Abbildung 1: Objekt Diagramm von MyApp



Das obige Beispiel gibt immer die gleichen Logmeldungen aus. Glücklicherweise ist es relativ einfach, *MyApp* so zu verändern, dass die Kontrolle zur Laufzeit erfolgt. Hier ist die leicht überarbeitete Version:

```
import com.foo.Bar;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class MyApp {

    static Logger logger =
        Logger.getLogger(MyApp.class.getName());

    public static void main(String[] args) {
        // BasicConfigurator durch
        // PropertyConfigurator ersetzt.
        PropertyConfigurator.configure(args[0]);

        logger.info("Entering application.");
        Bar bar = new Bar();
        bar.doIt();
        logger.info("Exiting application.");
    }
}
```

Diese Version von *MyApp* weist den *PropertyConfigurator* an, eine Konfigurationsdatei zu parsen und die Logging Umgebung entsprechend einzurichten.

Eine Konfigurationsdatei, die das gleiche Verhalten bewirkt, wie das vorhergehende Beispiel mit dem *BasicConfigurator* könnte folgendermaßen aussehen:

```
# Set root logger level to DEBUG and its only appender
# to A1.
log4j.rootLogger=DEBUG, A1

# A1 ist ein ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

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

Neben dem Properties Format ist es, wie schon zu Beginn gesagt, auch möglich, die Konfiguration im XML Format anzugeben.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration
    xmlns:log4j='http://jakarta.apache.org/log4j/'>
    <appender name="A1"
        class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern"
                value="%-4r [%t] %-5p %c %x - %m%n"/>
        </layout>
    </appender>

    <root>
        <priority value ="debug" />
        <appender-ref ref="A1" />
    </root>

</log4j:configuration>
```

Wenn wir nun feststellen, dass die Ausgabe von Komponenten des *com.foo* Packages uninteressant ist, kann dessen Ausgabe über Änderungen an der Konfigurationsdatei ausgeblendet werden. Das kann zum Beispiel so aussehen:

```
log4j.rootLogger=DEBUG, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=\
        org.apache.log4j.PatternLayout

# Ausgabe des Datums im ISO 8601 Format
```

```
log4j.appender.A1.layout.ConversionPattern=\
        %d [%t] %-5p %c - %m%n

# Im Package com.foo sollen nur noch Meldungen
# ausgegeben werden, die mindestens Warnungen
# sind.
log4j.logger.com.foo=WARN
```

Die Ausgabe ändert sich entsprechend:

```
2000-09-07 14:07:41,508 [main] INFO  MyApp - Entering appli\
cation.
2000-09-07 14:07:41,529 [main] INFO  MyApp - Exiting appli\
cation.
```

Da der Logger *com.foo.Bar* keinen eigenen Level hat, erbt er den von *com.foo*. In der Konfigurationsdatei wurde dessen Level auf *WARN* gesetzt. Das Logging Statement in der Methode *Bar.doIt* hat den Level *DEBUG*, also einen geringeren Level als *WARN*. Deswegen wird diese Logmeldung heraus gefiltert.

Hier ein Beispiel einer Konfigurationsdatei, die mehrere Appender verwendet:

```
log4j.rootLogger=debug, stdout, R

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=\
        org.apache.log4j.PatternLayout

# Ausgabe des Dateinamens und der Zelennummer
log4j.appender.stdout.layout.ConversionPattern=\
        %5p [%t] (%F:%L) - %m%n

log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=example.log

log4j.appender.R.MaxFileSize=100KB
# Keep one backup file
log4j.appender.R.MaxBackupIndex=1

log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=\
        %p %t %c - %m%n
```

Wenn nun *MyApp* gestartet wird, wird auf der Konsole folgendes ausgegeben:

```
 INFO [main] (MyApp2.java:12) - Entering application.
DEBUG [main] (Bar.java:8) - Doing it again!
 INFO [main] (MyApp2.java:15) - Exiting application.
```

Zusätzlich wurde dem *root* Logger ein weiter Appender zugewiesen. Es gibt also eine weitere Ausgabe in die Datei *example.log*. Diese Datei wird neu geschrieben, wenn ihre Größe 100 KB überschreitet. Wenn das passiert, wird die bisherige Datei zuvor automatisch in *example.log.1* verschoben.

Das Schöne daran ist jetzt, das hierfür nicht eine Zeile des Quellcodes verändert werden musste. Genauso gut könnte die Ausgabe an den UNIX Syslog daemon weiter geleitet werden, die Ausgabe von *com.foo* an den NT Event Logger oder an einen entfernten log4j Server, der eine eigen lokale Logging Strategie verfolgt.

6 Standard Initialisierungsverfahren

Die log4j Bibliothek setzt keine besonderen Einstellungen in der Umgebung voraus. Deswegen gibt es auch keine Standard log4j Appender. Wenn einige, genau definierte Umstände eintreten, wird log4j jedoch versuchen, sich automatisch selbst zu konfigurieren. Wie das geht, soll im Folgenden beschrieben werden.

Die Spezifikation von Java garantiert, dass der statische Initialisierer jeder Klasse genau einmal aufgerufen wird, und zwar dann, wenn die Klasse in den Speicher geladen wird. Man sollte im Hinterkopf behalten, dass einige Classloader eventuell mehrere Kopien der gleichen Klasse in den Speicher laden. Diese Kopien stellt die JVM in keinen Bezug zueinander. Die Standard Initialisierung ist dann hilfreich,wenn der Einstiegspunkt in die Anwendung von der Laufzeitumgebung abhängt. Zum Beispiel ist es denkbar, dass eine Anwendung als stand-alone Anwendung, als Applet oder als Servlet unter der Kontrolle eines Webservers ausgeführt wird.

Das Standard Initialisierungsverfahren erfolgt folgendermaßen:

1. Das Setzen des System Properties *log4j.defaultInitOverride* auf einen anderen Wert als *false* veranlasst log4j, das Standard Initialisierungsverfahren, also das hier beschriebene, nicht auszuführen.

2. Der Wert der *resource* String-Variable wird auf den Wert des System Properties *log4j.configuration* gesetzt. Die Angabe der Konfigurationsdatei über das System Property *log4j.configuration* ist der bevorzugte Weg. Wurde das System Property nicht gesetzt, wird *resource* auf den Standardwert *log4j.properties* gesetzt.

3. Versuche den Wert der *resource* Variablen in eine URL zu konvertieren.

4. Konnte die *resource* Variable nicht in eine URL konvertiert werden, z.B. auf Grund einer *MalformedURLException*, wird versucht, die *resource* durch Aufruf von

```
log4j.helpers.Loader.getResource(resource,
                                 Logger.class)
```

aus dem *CLASSPATH* zu bestimmen. Dieser Aufruf liefert nach erfolgreicher Suche dann die gewünschte URL.

5. Konnte trotz aller Bemühungen keine URL bestimmt werden, bricht das Verfahren an dieser Stelle ab.

   Zum Parsen und letztendlichem Konfigurieren von log4j wird der *PropertyConfigurator* verwendet, es sei denn, die URL endet mit *.xml*. In diesem Fall wird der *DOMConfigurator* verwendet. Zum Konfigurieren kann jede Klasse verwendet werden,die das *Configurator* Interface implementiert. Der Konfigurator wird durch Setzen des System Properties *log4j.configuratorClass* bekannt gemacht.

7 Beispiel Konfigurationen

7.1 Standard Initialisierung bei Tomcat

Das Initialisierungsverfahren ist besonders im Umfeld eines Webservers nützlich. Bei Tomcat 3.x und 4.x wird die *log4j.properties* einfach in das *WEB-INF/classes*-Verzeichnis der Webanwendung kopiert. Log4j findet dort die Konfigurationsdatei und initialisiert sich selbst. Das ist quietscheinfach und funktioniert.

Alternativ kann auch das System Property *log4j.configuration* gesetzt werden, bevor Tomcat gestartet wird. Bei Tomcat 3.x erfolgt dies in der *TOMCAT_OPTS* Umgebungsvariable, die zum Aufbau der Kommandozeile verwendet wird. Bei Tomcat 4.x nimmt man hierzu die Umgebungsvariable *CATALINA_OPTS*.

7.1.1 Beispiel 1

Das UNIX Shell Kommando

```
export TOMCAT_OPTS="-Dlog4j.configuration=foobar.txt"
```

teilt log4 mit, das *foobar.txt* die Konfigurationsdatei ist. Diese Datei wird in das *WEB-INF/classes*-Verzeichnis der Webanwendung kopiert. Die Datei wird dann durch den *PropertyConfigurator* ausgewertet. Jede Webanwendung verwendet dabei ihre eigene Konfigurationsdatei, da diese nur relativ zur Webanwendung ist.

7.1.2 Beispiel 2

Das UNIX Shell Kommando

```
export TOMCAT_OPTS=\
    "-Dlog4j.debug -Dlog4j.configuration=foobar.xml"
```

teilt log4j mit, seine internen debug Informationen auszugeben und die Datei *foobar.xml* als Konfigurationsdatei zu benutzen. Diese Datei wird wiederum in das *WEB-INF/classes*-Verzeichnis der Webanwendung kopiert. Da der Name der Datei mit *.xml* endet, wird zur Auswertung der *DOMConfigurator* benutzt. Jede Webanwendung verwendet wieder ihre eigene Konfigurationsdatei, da diese stets relativ zur Webanwendung ist.

7.1.3 Beispiel 3

Das Windows Shell Kommando

```
set TOMCAT_OPTS=\
    "-Dlog4j.configuration=foobar.lcf \
   -Dlog4j.configurationClass=com.foo.BarConfigurator"
```

teilt log4j mit, *foobar.lcf* als Konfigurationsdatei zu verwenden. Diese Datei wird wieder im *WEB-INF/classes*-Verzeichnis der Webanwendung erwartet. Entsprechend der Konfiguration der Systemeigenschaft *log4j.configuratorClass* wird diese Datei durch die Klasse *com.foo.BarConfigurator* ausgewertet. Jede Webanwendung verwendet wieder ihre eigene Konfigurationsdatei, da diese stets relativ zur Webanwendung ist.

7.1.4 Beispiel 4

Das Windows Shell Kommando

```
set TOMCAT_OPTS="-Dlog4j.configuration=file:/c:/foobar.lcf"
```

teilt log4 mit, die Datei *c:\foobar.lcf* als Konfigurationsdatei zu verwenden. Durch die Angabe einer voll qualifizierten URL *file:/c:/foobar.lcf* gilt diese Konfigurationsdatei für alle Webanwendungen.

Die einzelnen Webanwendungen laden ihre log4j Klassen durch ihre jeweiligen Classloader. Das kann auch unangenehmen Nebenwirkungen haben, denn hierdurch ist es möglich, dass jeder Teil der log4j-Umgebung unabhängig behandelt werden kann und auch ohne eine globale Synchronisation auskommt. Hierdurch kommt es zum zum Beispiel dazu, dass FileAppender, die für mehrere Webanwendungen gleich konfiguriert wurden auch versuchen, in die gleiche Datei zu schreiben. Das Ergebnis ist dann nicht sehr berauschend. Man muss deswegen sicher stellen, das die einzelnen log4j Konfigurationen der Webanwendungen nicht die gleiche System Ressource benutzen.

7.1.5 Initialisierung durch ein Servlet

Es ist auch möglich, ein spezielle Servlet für die Initialisierung von log4j zu verwenden. Dieses kann zum Beispiel folgendermaßen aussehen:

```
package com.foo;

import org.apache.log4j.PropertyConfigurator;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.io.IOException;

public class Log4jInit extends HttpServlet {
    public
    void init() {
        String prefix = getServletContext()
                        .getRealPath("/");
        String file = getInitParameter("log4j-init-file");
        // if the log4j-init-file is not set, then no
        // point in trying
        if(file != null) {
            PropertyConfigurator.configure(prefix + file);
        }
    }

    public void doGet(HttpServletRequest req,
        HttpServletResponse res) {
    }
}
```

Das Servlet wird in der web.xml der Webanwendung so konfiguriert:

```
<servlet>
  <servlet-name>log4j-init</servlet-name>
  <servlet-class>com.foo.Log4jInit</servlet-class>

  <init-param>
    <param-name>log4j-init-file</param-name>
    <param-value>
      WEB-INF/classes/log4j.lcf
    </param-value>
  </init-param>

  <load-on-startup>1</load-on-startup>
</servlet>
```

Das Schreiben eines Initialisierungsservlets ist der flexibelste Weg, log4j zu initialisieren. Hier gibt es keine Einschränkungen für den Code in der *init()* Methode des Servlets.

8 Nested Dialog Kontexte

Viele Systeme müssen mehrere Clients parallel bedienen können. Typischerweise werden die einzelnen Clients durch mehrere Threads bedient. Hierbei ist insbesondere Logging eine gut geeignete Debugging und Tracing Methode. Der traditionelle Ansatz ist der, die Logausgabe der einzelnen Clients durch separate Logger für jeden Client zu realisieren. Das erhöht die Anzahl der benötigten Logger und auch den Verwaltungsaufwand für sie.

Eine einfachere Technik ist es, jeder Logausgabe eines Clients einen Client spezifischen Stempel aufzudrücken. Diese Methode wurde von Neil Harrison in [7] beschrieben.

Um jeder Loganfrage einen eindeutigen Stempel aufzudrücken, werden Kontext-Informationen im NDC[2] benötigt. Diese Information muss durch den Benutzer angegeben werden. Die NDC Klasse sieht folgendermaßen aus:

```
public class NDC {
    // Zur Ausgabe der Diagnostic.
    public static String get();

    // Entfernen des naechsten Kontextes aus NDC.
    public static String pop();

    // Dem aktuelle Thread einen weiteren Kontext
    // hinzufuegen.
    public static void push(String message);

    // Diagnostic Kontext fuer diesen Thread entfernen.
    public static void remove();
}
```

NDC wird pro Thread als *Stack* abgelegt. Alle Methoden der Klasse *org.apache.log4j.NDC* sind statisch. Man geht davon aus, das die NDC Ausgabe für jede Loganfrage aktiviert wird. Dadurch hat die entsprechende log4j Komponente Zugriff auf den ganzen NDC Stack des aktuellen Threads. Das Alles erfolgt ohne Zutun des Benutzers, der nur dafür sorgen muss, dass der NDC über *push* und *pop* an wenigen wohldefinierten Codestellen mit den richtigen Informationen versorgt wird. Im Gegensatz dazu erfordert der Ansatz der getrennten Logausgabe pro Client einen hohen Codierungsaufwand.

Das Ganze soll wieder an einem Beispiel etwas anschaulicher werden. Hierzu kann ein Servlet dienen, das mehrere Clients mit Informationen versorgen soll. Das Servlet erstellt den NDC zu Beginn des Requests, bevor irgend ein anderer Code ausgeführt wird. Die Kontextinformation kann hier z.B. der Hostname des Clients und andere Informationen sein, die dem Request entnommen werden können, wie z.B. der Inhalt von Cookies. Selbst

[2] Abkürzung für **N**ested **D**iagnostic **C**ontext

wenn mehrere Clients nun gleichzeitig auf das Servlet zugreifen, können die Logausgaben der Clients, die alle denselben Logger nutzen, unterschieden werden, weil alle einen anderen NDC Stack haben. Im Gegensatz dazu steht der erhöhte Aufwand des Ansatzes, der für jeden Client einen eigenen Logger instantiieren muss.

Daneben gibt es aber auch Anwendungen, wie z.B. Virtual Hosting Web Server, die ihre Logausgaben geordnet nach dem Kontext des virtuellen Hosts machen müssen. Nebenbei muss auch noch die Komponente unterschieden werden, die die Logausgabe gemacht hat. Die neueren log4j Versionen unterstützen deswegen mehrere hierarchische Bäume. Diese Erweiterung erlaubt jedem virtuellen Host eine eigene Kopie der Logger Hierarchie.

9 Performanz

Das Argument, das am häufigsten gegen Logging verwendet wird, ist das der Rechenzeitkosten. Dieser Vorwurf ist sicherlich gerechtfertigt, denn selbst Anwendungen von moderater Größe können Tausende von Logausgaben produzieren. Deswegen wurde viel Aufwand getrieben, um den tatsächlichen Bedarf an Rechenleistung zu bestimmen und diesen so gut wie möglich zu optimieren. Log4j legt Wert auf Geschwindigkeit und Flexibilität. Dabei wird immer zuerst an die Geschwindigkeit gedacht.

Der Nutzer sollte sich folgender Kriterien hinsichtlich der Performanz im Klaren sein.

9.1 Logging Performanz bei deaktiviertem Logging

Wenn das Logging vollständig oder auch nur für eine bestimmte Menge von Leveln deaktiviert ist, bestehen die Kosten, die durch eine Logausgabe entstehen, aus dem Aufruf einer Methode und einem Integer Vergleich. Auf einem Computer mit einem Pentium II und 233MHz liegt das etwa in einem Bereich von 5 bis 50 Nanosekunden. Daneben gibt es aber auch noch *versteckte* Kosten, die durch die Auswertung eines Parameters entstehen. Als Beispiel wird ein Logger betrachtet, der folgende Ausgabe machen soll:

```
1 logger.debug("Entry number: " + i + " is "
                + String.valueOf(entry[i]));
```

Hier entstehen zusätzliche Kosten durch die Auswertung des Parameters. In diesem Fall bestehen diese aus der Umwandlung der integer *i* und *entry[i]* in einen String und die Verkettung der auftretenden Strings. Dieses erfolgt unabhängig davon, ob dies tatsächlich ausgegeben wird oder nicht. Im Allgemeinen können diese zusätzlichen Kosten sehr hoch werden, abhängig von der Menge der einbezogenen Parameter.

Um die Kosten der Parameterauswertung zu umgehen sollte der Ausdruck folgendermaßen umgeschrieben werden:

```
if (logger.isDebugEnabled()) {
    logger.debug("Entry number: " + i + " is "
                 + String.valueOf(entry[i]));
}
```

Ist das Logging deaktiviert, entstehen durch die Auswertung des Parameters keine Kosten. Bei aktiviertem Logging entstehen aber andererseits zusätzliche Kosten, da nun zweimal überprüft wird, ob dieser Logger nun ausgeben soll oder nicht. Das erste Mal bei der Auswertung von *isDebugEnabled* und das zweite Mal bei *debug*. Dieser zusätzliche Aufwand fällt aber kaum ins Gewicht, wenn man sich vor Augen führt, dass die Überprüfung eines Loggers nur etwa 1% der Zeit einer tatsächlich ausgeführten Logausgabe kostet. Hierdurch wird eine messbare Reduzierung des Aufwands einer Loganfrage auf Kosten der Flexibilität erreicht.

Einige Nutzer verwenden JAVA Präprozessoren oder speziell Compiliertechniken, um die Logausgaben bei Bedarf vollständig zu entfernen. Hierdurch wird eine optimale Performanz erzielt. Da die Übersetzten Dateien dann aber keinerlei Logausgaben mehr beinhalten, muss man natürlich neu compilieren, um das Logging wieder zu aktivieren. Das ist ein relativ hoher Preis für einen geringen Zuwachs an Performanz.

9.2 Performanz der Entscheidung ob ein Logger aktiv ist

Die Performanz der Entscheidung, ob eine Logausgabe erfolgen soll oder nicht, entspricht der Performanz des Entlangwanderns in der Logger-Hierarchie. Wenn Logging aktiviert ist, muss log4j immer noch den Level der Logausgabe mit dem Level des Loggers vergleichen. Da ein Logger seinen Level von seinen Vorfahren erben kann, kann das auch bedeuten, das der nächste Vorfahre mit einem zugewiesenen Level gesucht werden muss. Es floss viel Gehirnschmalz darein, dieses Entlangwandern an der Hierarchie so effizient wie möglich zu gestalten. So verweisen Kind Logger z.B. immer nur auf einen existierenden Vorfahren. Im Beispiel des Loggers *com.foo.Bar* aus Abschnitt 3.1 wurde dieser direkt mit dem *root*-Logger verbunden. Die nicht existierenden Logger *com* und *com.foo* wurden dabei einfach weggelassen. Besonders in einfach aufgebauten Hierarchien wird hierbei das Entlangwandern stark beschleunigt. Die typischen Kosten des Entlangwanderns an der Hierarchie sind in etwa 3 mal so hoch, als wenn das Logging vollständig deaktiviert wurde.

9.3 Performanz der Logausgabe

Hierbei handelt es sich um die Kosten, die ein Logger benötigt, um die Logausgabe zu formatieren und sie auf dem Ausgabemedium auszugeben. Auch hier wurden viele Anstrengungen unternommen, um die Layouts (Formatierer) so schnell wie möglich zu machen. Gleiches gilt für die Appender. Die

typischen Kosten, um eine Logausgabe auszugeben, belaufen sich auf etwa 100 bis 300 Mikrosekunden. Aktuelle Information hierzu findet sich bei [11].

Obwohl log4j viele Features hat, wurde es doch in erster Linie in Hinblick auf Geschwindigkeit entworfen. Einige Komponenten von log4j wurden mehrere Male neu geschrieben, um die Performanz zu verbessern. Aber es gibt auch noch heute immer wieder neue Vorschläge für Optimierungen. So haben Performanztests mit dem *SimpleLayout* ergeben, dass hier log4j genauso schnell ist, wie *System.out.println*.

10 SUNs Logging API

Leider hat sich SUN mit dem JDK 1.4 entschieden, ein sehr ähnliches, aber in Details unterschiedliche Logging-API aufzunehmen [14]. Ein tiefgreifender Vergleich dieser beiden APIs würde jedoch den Rahmen dieses Tutorials sprengen. Es soll aber auch nicht ganz unter den Tisch gekehrt werden, dass SUN mit dem JDK 1.4 eine eigene Logging-API zur Verfügung stellt.

Seimet stellt in [12] beide APIs kurz vor und geht auch auf Unterschiede und Gemeinsamkeiten ein. Er kommt zu dem Schluss, dass die beiden APIs ähnlich, aber nicht identisch sind. Einen tieferen Einblick in beide Logging APIs bekommt man in dem Buch von Gupta [6].

SUN's API kommt aber zu spät. Zu diesem Schluss kommt auch Bernhard Bablok in [3]. Mittlerweile hat log4j Einzug in viele populäre Programme, wie z.B. Tomcat oder JBoss, gefunden und hat sich somit zu einem Quasi-Standard entwickelt, der auch mit früheren JDK Versionen problemlos funktioniert. Fraglich ist, ob sich die Programmierer in die Abhängigkeit des JDK 1.4 begeben, wenn sie mit log4j eine vollständige Lösung vorfinden, die schon mehrere Jahre Zeit hatte zu reifen und sich so zu einem ausgewachsenen, stabilen Produkt entwickeln konnte.

Seimet fasst dieses in [12] so zusammen:

> Wer seine Projekte unabhängig von Bibliotheken Dritter halten will und mit dem JDK 1.4 entwickelt, ist mit der dort integrierten Logging API gut bedient. Mehr Freiheiten, geringere Leistungseinbußen und die Filterung der Protokollinformationen durch grafische Werkzeuge bietet dagegen log4j. Darüber hinaus zeichnet es sich durch gut strukturierte Konfigurationsmechanismen und zahlreiche Ausgabemethoden aus.

11 Zusammenfassung

Bei log4j handelt es sich um eine populäre JAVA Logging-Bibliothek. Eine ihrer herausragenden Eigenschaften ist die Vererbung bei Loggern. Die Benutzung von Logger Hierarchien erlaubt eine feingranulare Kontrolle über die

einzelnen Logausgaben. Hierdurch ist eine Reduzierung der Menge der Logausgaben bei gleichzeitiger Minimierung der Kosten für das Logging möglich.

Ein Vorteil der log4j API ist der geringe Verwaltungsaufwand durch den Nutzer. Sind die log Statements erst einmal im Code, können sie einfach durch Konfigurationsdateien gesteuert werden. Sie können beliebig aktiviert oder deaktiviert werden und auch auf unterschiedlichen Ausgabemedien in vom Nutzer vorgegebenen Formatierungen ausgegeben werden. Die log4j Bibliothek wurde so entworfen, dass die Logausgaben in der fertigen Anwendung verbleiben können, ohne große Performanzeinbußen hinnehmen zu müssen.

Wer sich einen tieferen Einblick in die Arbeit mit log4j verschaffen möchte, sollte sich das Buch [5] von Ceki Gülcü anschauen.

Literatur

[1] Apache logging. `http://logging.apache.org/`.

[2] Apache software license. `http://www.apache.org/licenses/LICENSE-2.0`.

[3] Bernhard Bablok. Zu den akten. *Linux-Magazin*, April 2002.

[4] Ceki Gülkü. Short introduction to log4j. März 2002.

[5] Ceki Gülkü. *The complete log4j manual*. QOS, Februar 2003.

[6] Samudra Gupta. *Logging in Java with the JDK 1.4 Logging API and Apache log4j*. Apress, 2003.

[7] Neil Harrsion. *Patterns for Logging Diagnostic Messages*. Addison-Wesley, 1997.

[8] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Reading, MA, 1999.

[9] log4j. `http://logging.apache.org/log4j`.

[10] Open source initiatitive. `http://www.opensource.org`.

[11] org.apache.log4j.performance.logging. `http://logging.apache.org/log4j/docs/api/org/apache/log4j/performance/L%ogging.html`.

[12] Uwe Seimet. Mithören erwünscht. *iX*, Dezember 2003.

[13] Semper. `http://www.semper.org`.

[14] Suns logging api. `http://java.sun.com/j2se/docs/1.4/docs/api/java/util/logging/package-su%mmary.html`.

A Konvertierungsmuster

Hier sind alle Kürzel für das Pattern Layout beschrieben.

c **Beschreibung:** Ausgabe der Kategorie des Logging Events. Optional kann die Ausgabe durch die Angabe einer dezimalen Konstanten in Klammern begrenzt werden.

Wenn die Ausgabe begrenzt wird, werden nur die gegebene Anzahl Komponenten von rechts ausgegeben. Standardmäßig wird die Kategorie voll ausgeschrieben.

Beispiel:
Die Kategorie *a.b.c* wird über das Pattern *%c{2}* als *b.c* ausgegeben.

C **Beschreibung:** Ausgabe des voll qualifizierten Klassennamens des Aufrufers. Optional kann die Ausgabe durch die Angabe einer dezimalen Konstanten in Klammern begrenzt werden.

Wenn die Ausgabe begrenzt wird, werden nur die gegebene Anzahl Komponenten von rechts ausgegeben. Standardmäßig wird der Klassenname voll ausgeschrieben.

Beispiel:
Die Klasse *org.apache.xyz.SomeClass* wird über das Pattern *%C{1}* als *SomeClass* ausgegeben.

Warnung: Die Bestimmung des Klassennamens des Aufrufers ist langsam und sollte deswegen vermieden werden, es sei denn, die Ausführungsgeschwindigkeit ist kein Thema.

d **Beschreibung:** Ausgabe des Datums des Logging Events. Optional kann das Format in Klammer angegeben werden. Standardmäßig wird das ISO8601 Format verwendet.

Die Angabe des Formats entspricht dem Pattern String des *SimpleDateFormat*s. Obwohl dieser eine Bestandteil des Standard JDK ist,ist die Performanz eher schlecht.

Beispiel:
%d{HH:mm:ss.SS} oder *%d{dd MMM yyyy HH:mm:ss,SS}*

Um performantere Ergebnisse zu erzielen wird die Verwendung des log4j eigenen Formatierers empfohlen. Hier kann das Format über String eingestellt werden.

Beispiel: *%d{ISO8601}* oder *%d{ABSOLUTE}*

F **Beschreibung:** Ausgabe des Dateinamens von dem die Logging Anfrage kam.

WARNUNG: Die Bestimmung dieser Ortsinformation ist extrem langsam und sollte deswegen vermieden werden, es sei denn, die Ausführungsgeschwindigkeit ist kein Thema.

l **Beschreibung:** Ausgabe der Ortsinformation des Aufrufers des Logging Events.

Die Ortsinformation hängt von der Implementierung der JVM ab, besteht aber normalerweise aus einem voll qualifizierten Namen der aufrufenden Methode gefolgt vom Dateinamen und der Zeilennummer in Klammern.

L **Beschreibung:** Ausgabe der Zeilennummer von der der Aufruf des Logging Events kam.

Warnung: Die Bestimmung dieser Ortsinformation ist extrem langsam und sollte deswegen vermieden werden, es sei denn, die Ausführungsgeschwindigkeit ist kein Thema.

m **Beschreibung:** Ausgabe der Logmeldung, die zu einem Logging Event assoziiert ist.

M **Beschreibung:** Ausgabe des Methodennamens, in dem der Aufruf des Logging Events ausgelöst wurde.

Warnung: Die Bestimmung dieser Ortsinformation ist extrem langsam und sollte deswegen vermieden werden, es sei denn, die Ausführungsgeschwindigkeit ist kein Thema.

n **Beschreibung:** Ausgabe des plattformabhängigen Zeilentrennungszeichens.

Die Verwendung dieses Kürzels bietet die gleiche Performanz als ob das plattformabhängige Zeilentrennungszeichen
n oder
r
n verwendet wird und ist deswegen der bevorzugte Weg, Zeilen zu trennen.

P **Beschreibung:** Ausgabe der Priorität des Logging Events.

r **Beschreibung:** Ausgabe der Anzahl Millisekunden, die zwischen dem Start der Anwendung und der Erstellung des Logging Events verstrichen sind.

t **Beschreibung:** Ausgabe des Threads, der das Logging Event erzeugt hat.

x **Beschreibung:** Ausgabe des NDC (Nested Dialog Context), der mit dem Thread assoziiert ist, der das Logging Event erzeugt hat.

X **Beschreibung:** Ausgabe des MDC (Mapped Dialog Context), der mit dem Thread assoziiert ist, der das Logging Event erzeugt hat. Dieses Kürzel*muss* von einem Schlüssel für die Map in Klammern angegeben werde. Der Wert, der in der MDC zu diesem Schlüssel assoziiert ist, wird ausgegeben.

Beispiel:
In *%X{clientNumber}* ist *clientNumber* der Schlüssel für die Map.

%% **Beschreibung:** Ausgabe eines einzelnen %-Zeichens.

Die verschiedenen Formate zur Begrenzung von Ausgabe, z.B.für *c* oder *m* sind nachfolgend beispielhaft beschrieben.

Format	**links**	**min**	**max**	**Beschreibung**
%20c	ja	20	keine	Linksbündig, aufgefüllt mit Leerzeichen, wenn der Name der Kategorie kürzer als 20 Zeichen ist.
%-20c	nein	20	keine	Rechtsbündig, aufgefüllt mit Leerzeichen, wenn der Name der Kategorie kürzer als 20 Zeichen ist.
%.30c	n/a	keine	30	Abschneiden am Anfang, wenn der Name der Kategorie länger als 30 Zeichen ist.
%20.30c	nein	20	30	Linksbündig, aufgefüllt mit Leerzeichen, wenn der Name der Kategorie kürzer als 20 Zeichen ist. Abschneiden am Anfang, wenn der Name der Kategorie länger als 30 Zeichen ist.
%20.30c	ja	20	30	Rechtsbündig, aufgefüllt mit Leerzeichen, wenn der Name der Kategorie kürzer als 20 Zeichen ist. Abschneiden am Anfang, wenn der Name der Kategorie länger als 30 Zeichen ist.