Logging in Java

Verteilte und Parallele Systeme

Seminararbeit von
Marc Berendes und Philipp Wever

Fachhochschule Bonn-Rhein-Sieg
Studiengang Angewandte Informatik

Betreuender Dozent: Prof. Dr. Rudolf Berrendorf

Stand: 01.06.2003

“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.”

Brian W. Kernighan and Rob Pike, *"The Practice of Programming"*

Inhaltsverzeichnis

1. Einführung

Diese Seminararbeit befasst sich mit dem Thema Logging in Java. Sie konzentriert sich dabei hauptsächlich auf die beiden Logging Frameworks „Java Logging API“ und „Jakarta Log4J“. Außerdem soll eine kleine Einführung in „Syslog“ zur Erweiterung des Wissens und für einen wirklichkeitsnahen Einsatzzweck dargestellt werden.

Da die gesamten Methoden der beiden Frameworks als sehr umfangreich angesehen werden müssen, dient diese Arbeit sicherlich nicht als ausführliche Anleitung, sondern nur als Einführung in das Thema.

2. Motivation

Während der Entwicklungsphase einer Anwendung ist es zur Gewohnheit geworden, Fehler mit Hilfe eines Debuggers zu eliminieren. Dies gilt als recht zuverlässige und schnelle Methode. Was passiert jedoch im alltäglichen Produktionsbetrieb, wo das setzen von Breakpoints oder einrichten von Watches nicht immer wieder von Neuem vorgenommen werden kann?

Bei dieser Frage stößt man auf das Konzept des Logging. Jedem Java-Programmierer sollte wohl der `System.out.println()` Befehl, ein Begriff sein. Häufig wird er verwendet um bestimmte Warnungen zur Lokalisierung von Fehlern auszugeben. Logging geht jedoch weit über diese Funktion hinaus. Es bietet eine einfache Möglichkeit Fehler, nicht nur zur Entwicklungszeit, einzugrenzen und Programmabläufe zu überprüfen. Befinden sich erst einmal die Logging-Routinen im Programm können diese jederzeit aktiviert und dazu benutzt werden, das Verhalten der Anwendung zur Laufzeit zu überwachen.

Durch Loggersysteme wird also nicht nur dem Programmierer die Fehlererkennung und Beseitigung erleichtert. Es wird sogar die Wartung und Überwachung einer Applikation erheblich vereinfacht, indem zum Beispiel die Ressourcenverwaltung, Systemabstürze oder auftretende Fehler aufgezeichnet und später ausgewertet werden können.

3. Java Logging API

Seit der Version 1.4 des Java Development Kits von Sun wird ein eigenes Logging Interface zur Entwicklungsumgebung mitgeliefert. Es ist im Paket java.util.logging abgelegt. Das Logging-Package bietet unter anderem die Möglichkeit der dynamischen Konfiguration, mehrere Log Levels und verschiedene Ausgabeformate.

3.1 Framework

Es folgt eine kurze Übersicht der Kassen und Interfaces, die das Java Logging API beeinhaltet.

Interfaces	
Filter	Ein Filter wird dazu verwendet, um zu spezifizieren, was geloggt werden soll.
Klassen	
ConsoleHandler	Schreibt eine log Meldung auf die Konsole
ErrorManager	Kann zu einem Handler hinzugefügt werden, um auf Fehlermeldungen zu reagieren
FileHandler	Schreibt eine log Meldung in eine Datei
Formatter	Konvertiert Protokolleinträge in ein vorgegebenes Format
Handler	Übernimmt Protokolleinträge von Loggern, konvertiert sie mit einem Formatter und schickt sie an ein vorgegebenes Ziel.
Level	Durch einen (Log) Level kann die Wichtigkeit eine Meldung vorgegeben werden.
Logger	Nimmt Protokolleinträge entgegen und veranlasst deren Verarbeitung
LoggingPermission	Weist Rechte zur Veränderung der Konfiguration zu
LogManager	Zentrale Komponente beim Zugriff auf Logger
LogRecord	Repräsentiert einen einzelnen Protokolleintrag
MemoryHandler	Legt die log Meldung in einem Puffer ab
SimpleFormatter	Konvertiert einen Protokolleintrag in Klartext
SocketHandler	Schreibt die log Meldung auf einen Socket
StreamHandler	Schreibt die log Meldung in einen OutputStream
XMLFormatter	Konvertiert einen Protokolleintrag in ein Standard XML Format

3.2 Funktionsweise

Die gängigsten Funktionen der Java Logging API sind Logger, Log Levels, Handler, Formatter und Log Manager. Im Folgenden wird beschrieben, wie die einzelnen Klassen verwendet werden.

3.2.1 Logger

Ein Logger ist ein Objekt, das in jede Applikation implementiert werden muss, von der aus etwas gelogged werden soll. Jeder Logger bekommt einen Namen, der aus einer Punkt-Zeichen-Sequenz besteht, die im so genannten *namespace* hierarchisch geordnet ist. Der Name basiert normalerweise auf dem Namen des Paketes und dem Klassennamen. Optional ist es auch möglich „anonyme“ Logger zu erstellen, die nicht in die Logger Hierarchie aufgenommen werden.

Beispiel für die Generierung eines Loggers der Klasse `Log` im Paket `xxx`:

```
static Logger logger = Logger.getLogger(“xxx.Log“);
```

Die Methode `getLogger()` gibt einen passenden Logger zurück, falls dieser schon existiert. Andererseits wird ein neuer Logger erzeugt.

Jeder Logger bezieht sich auf seinen direkten Vorfahren im namespace (hier: xxx). Er erbt von ihm unter anderem den Log Level.
Ein anonymer Logger kann mit der Methode `getAnonymousLogger()` zurückgegeben bzw. erzeugt werden. Anonyme Logger haben immer den root-Logger im namespace als Vorfahren.
Mit den Methoden `entering()` und `exiting()` ist es möglich, das Starten und Beenden von Applikationen aufzuzeichnen. Diesen Methoden muss der Name der zu überwachenden Klasse und Methode übergeben werden.

Beispiel für eine Klasse `Bank`, die die Methode `konto_eroeffnen` enthält:

```
Logger.entering("Bank","konto_eroeffnen");
Logger.exiting("Bank","konto_eroeffnen");
```

Jedes Mal, wenn nun eine Kontoeröffnung eingeleitet bzw. beendet wird, wird eine entsprechende Meldung aufgezeichnet.

3.2.2 Log Levels

Die Log Levels der Java Logging API dienen dazu, die Dringlichkeit einer Meldung festzulegen. Die Klasse `Level` enthält eine Reihe von standardisierten Dringlichkeitsstufen, die dazu verwendet werden können, die Logging Ausgabe zu steuern.

Die Log Levels in absteigender Ordnung sind:

- SEVERE (höchste Dringlichkeit)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (niedrigste Dringlichkeit)

Zusätzlich gibt es noch die Level OFF und ALL, um das Logging auszuschalten bzw. jede Meldung zu loggen.

Wird ein Logger mit einem gewissen Log Level initialisiert, so werden alle Meldungen mit diesem Log Level und mit allen darüber liegenden aufgezeichnet.

Im folgenden Beispiel wird der Logger `logger` auf das Log Level Info gesetzt:

```
logger.setLevel(Level.INFO);
```

Bei diesem Logger werden nun alle Meldungen mit den Dringlichkeitsstufen INFO, WARNING und SEVERE aufgezeichnet.

3.2.3 Handler

Jeder Logger hat Zugriff auf eine Anzahl von Handlern. Ein Handler Objekt übernimmt eine log Meldung von einem Logger und exportiert sie zu einem bestimmten Ziel. Ein Handler kann mit der Methode `setLevel(Level.OFF)` ausgeschaltet werden und entsprechend mit der `setLevel()` Methode wieder aktiviert werden.

In JDK 1.4 stehen momentan folgende Handler zur Verfügung:

- ConsoleHandler Gibt die log Meldung auf der Konsole aus.
- FileHandler Schreibt die log Meldung in eine Datei.
- MemoryHandler Legt die log Meldung in einem Puffer ab.
- SocketHandler Schreibt die log Meldung auf einen Socket.
- StreamHandler Schreibt die log Meldung in einen OutputStream

Es ist auch möglich mehrere Handler hintereinander zu schalten. So kann man zum Beispiel einen File Handler hinter einen Memory Handler schalten. Nach der Pufferung einer bestimmten Anzahl von Nachrichten kann dann der Inhalt des Memory Handler in eine Datei übertragen werden. Des Weiteren ist es möglich eigene Handler zu erstellen, indem man die Klasse `Handler()` erweitert.

Hinweis: Die Log Levels INFO und höher werden, unabhängig vom benutzten Handler, standardmäßig automatisch auf die Konsole geschrieben.

3.2.4 Formatter

Normalerweise besitzt jeder Handler einen Formatter, mit dem das Format der log Meldung vorgewählt werden kann.

Momentan sind folgende Formatter verfügbar:

- SimpleFormatter Generiert die log Meldung als Klartext
- XMLFormatter Generiert die log Meldung im Standard XML Format

Ein Beispiel für die Aufzeichnung einer Meldung in XML Format:

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2003-05-12T19:39:21</date>
  <millis>1052761161400</millis>
  <sequence>2</sequence>
  <logger>LogA</logger>
  <level>WARNING</level>
  <class>LogA</class>
  <method>doA</method>
  <thread>10</thread>
  <message>Fehler in LogA!</message>
</record>
</log>
```

3.2.5 Log Manager

Der Log Manager stellt eine zentrale Komponente der Logger dar, die folgende Funktionen bereitstellt:

- Verwaltung und Erstellung der Logger im namespace
- Erzeugen von anonymen Loggern
- Verwaltung der Konfigurationseinstellungen

Systemweit ist nur ein globales Exemplar des Log Managers verfügbar, das über die Methode `LogManager.getLogManager()` abgerufen werden kann.
Bei jedem Start des Anwendungsprogramms wird die Log Manager Klasse automatisch aufgerufen. Alle Einstellungen, die für die Ausführung der Protokollierung von Bedeutung sind, sind in der Datei `<java_home>/jre/lib/logging.properties` hinterlegt. Diese wird automatisch in den Log Manager geladen.

Es wird auch eine Methode readConfiguration() zur Verfügung gestellt, die es ermöglicht, die Konfiguration neu zu lesen. Dabei werden alle Einstellungen, die während der Programmausführung verändert wurden, auf die Voreinstellungen zurückgesetzt.

Durch Property-Dateien, wie zum Beispiel `java.util.logging.config.class` kann eine Einstellung in den Log Manager nachgeladen werden. Somit ist es möglich eine Applikation zur Laufzeit zu rekonfigurieren.

3.3 Implementierungsbeispiel

Das folgende Beispiel beschreibt den Quellcode einer einfachen Anwendung des Logging.

Klasse LoggingA

```
import java.io.IOException;
import java.util.logging.*;

/**
 * @author Philipp Wever
 * @version 20.05.2003
*/

public class LoggingA {
    static Logger logger;
    Handler file_handler;
    Formatter klartext;


    // ------------ Konstruktor für Klasse LoggingA ----------------
    public LoggingA() throws IOException {
        // Logger erzeugen
        logger = Logger.getLogger("LoggingA");

        // File Handler erzeugen
        file_handler = new FileHandler("LoggingA.txt");

        // Formatter erzeugen
        klartext = new SimpleFormatter();
        file_handler.setFormatter(klartext);
        logger.addHandler(file_handler);
    }

    // ------------------- Methode A ----------------
    public void methodeA() {
        double r = 0.6;
        if(r<0.5){
            logger.warning("Fehler in LoggingA!");
        }
        else{
            logger.info("LogA ok :)");
        }
    }
}
```

Klasse LoggingB

```
import java.io.IOException;
import java.util.logging.*;

/**
* @author Philipp Wever
* @version 20.05.3003
*/

public class LoggingB {
    static Logger logger = Logger.getLogger("LoggingB");
    static Handler handler;
    LoggingA log_a;

    public LoggingB(){
        // Console Handler erzeugen
        handler = new ConsoleHandler();

        // Log Level des Handlers auf FINEST setzen
        handler.setLevel(Level.FINEST);
        logger.addHandler(handler);
        logger.setUseParentHandlers(false);
        try{
            log_a = new LoggingA();
            logger.config("LoggingA erzeugt.");
        }
        catch (Exception e){
            logger.severe("LoggingA nicht erzeugt!");
            System.exit(1);
        }
    }
    public void methodeB() {
        logger.info("Verarbeite Methode B.");
        log_a.methodeA();
    }
}
```

Klasse LoggingC

```
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
* @author Philipp Wever
* @version 20.05.2003
*/

public class LoggingC {
    static Logger logger = Logger.getLogger("LoggingC");
    public static void main(String[] args) {
        logger.info("Betrete Applikation");
        LoggingB log_b = new LoggingB();
        log_b.methodeB();
        logger.log(Level.INFO, "Verlasse Applikation");
    }
}
```

Ausgabe auf der Konsole

```
31.05.2003 14:02:05 LoggingC main
INFO: Betrete Applikation
31.05.2003 14:02:05 LoggingB methodeB
INFO: Verarbeite Methode B.
31.05.2003 14:02:05 LoggingA methodeA
INFO: LogA ok :)
31.05.2003 14:02:05 LoggingC main
INFO: Verlasse Applikation
```

Inhalt der erzeugten Datei LoggingA.txt

```
31.05.2003 14:02:05 LoggingA methodeA
INFO: LogA ok :)
```

4. Einführung in Syslog

Syslog ist ein, vor allem unter UNIX Betriebssystemen, weit verbreiteter Standard zum Aufzeichnen von Systemmeldungen. Es läuft ein Syslog Server als Deamon im Hintergrund, der auf einen bestimmten Port (514, bzw. 10514) hört und Systemmeldungen protokolliert. Dies müssen nicht nur eigene Systemmeldungen sein. Hier können unter Anderem auch Router oder andere Peripheriegeräte zum Einsatz kommen.
Unter UNIX Systemen ist der Syslog Daemon meisst standardmäßig enthalten. Auf NT Betriebssystemen gibt es den NTEventLog. Es sind allerdings auch kommerzielle Produkte, wie z.B. Winsyslog (http://www.winsyslog.com) erhältlich.

Logging - Nachrichten an einen Syslog Daemon zu schicken ist vor allem dann sinnvoll, wenn der Systemadministrator direkt auf Fehler hingewiesen werden soll und nicht erst spezifische Logfiles lesen muss.
Wir finden es sehr wichtig an dieser Stelle auf Syslog zu verweisen, da uns aufgefallen ist, daß Syslog unter den Studierenden nicht immer ein Begriff ist, aber sicherlich zu einem der wichtigsten Logging Einrichtungen in der UNIX Welt gehört.

5. Jakarta Log4J Projekt

Jakarta ist ein Projekt der Apache Software Foundation und beschäftigt sich überwiegend mit Javaprogrammierung. Derzeit gibt es neben Log4J noch 25 weitere Produkte, die unter dem Jakarta Projekt erstellt wurden. Allesamt sind Bibliotheken, APIs, kleine Tools, Serveranwendungen, Frameworks oder Engines für Java. Das bekannteste Produkt ist zweifelsohne der Tomcat Server. Apache Produkte, einschließlich Log4J, werden als Open Source Software angeboten.

Log4J ist eine freie Alternative zur Java Logging API und wurde im Laufe der Zeit für diverse andere Programmiersprachen, wie z.B. ANSI C, C++, Perl, Python, Ruby, .NET Plattform, PHP, PL/SQL… portiert. Zusätzlich sind von Drittanbietern diverse Erweiterungen zu Log4J erhältlich.

Weiterführende Informationen zu Log4J bekommt man unter:
http://jakarta.apache.org/log4j/.

5.1 Framework

Die komplette Javadoc Dokumentation zu Log4J kann im Internet unter folgender Adresse eingesehen werden:

http://jakarta.apache.org/log4j/docs/api/index.html

Die nachfolgende Tabelle gibt einen kurzen Überblick über die in Log4j, Version 1.2.8 spezifizierten Packages:

Package	
apache.org.log4j	Das Hauptpaket
apache.org.log4j.chainsaw	Chainsaw ist ein GUI zur Auswertung des Log4J Paketes
apache.org.log4j.config	Setzen und Abfragen von Einstellungen
apache.org.log4j.helpers	Zur internenVerwendung
apache.org.log4j.jdbc	Speichern von LogEvents in einer Datenbank
apache.org.log4j.jmx	Einstellungen per JMX verwalten
apache.org.log4j.lf5	LogFactor5 – nicht komplett implementiert
apache.org.log4j.net	Remote Logging
apache.org.log4j.nt	NT Event Logging
apache.org.log4j.or	ObjectRenders – Rendern von Nachrichten, abhängig vom Klassentyp
apache.org.log4j.or.jms	MessageRenderer – Objekt vom Typ javax.jms.Message
apache.org.log4j.or.sax	AttributesRenderer – Objekt vom Typ org.xml.sax.Attributes
apache.org.log4j.performance	Leistungsmessung der Log4J Komponenten
apache.org.log4j.spi	Beinhaltet Teile des System Programming Interface zur Erweiterung von Log4J
apache.org.log4j.varia	Verschiedene Filter, Appender, etc.
apache.org.log4j.xml	XML basierte Komponenten
apache.org.log4j.xml.examples	Beispiele zu den XML Komponenten

In dieser Seminararbeit werden wir uns ausschließlich mit dem Hauptpaket, apache.org.log4j, beschäftigen. Dieses Paket besteht aus folgenden Interfaces und Klassen:

Interface: Appender

Klassen: AppenderSkeleton, AsyncAppender, BasicConfigurator, Category, ConsoleAppender, DailyRollingFileAppender, FileAppender, Hierarchy, HTMLLayout, Layout, Level, Logger, LogManager, MDC, NDC, PatternLayout, Priority, PropertyConfigurator, RollingFileAppender, SimpleLayout, TTCCLayout, WriterAppender.

Da die einzelnen Klassen sehr umfangreich sind, gehen wir hier nicht auf jede einzeln ein, sondern erläutern die einzelnen Klassen an entsprechender Stelle.

5.2 Funktionsweise

Das Log4J Paket funktioniert ähnlich wie die Logging API von Java. Es besteht aus drei Hauptkomponenten: Logger, Appender und Layout. Diese Werden in den folgenden Unterkapiteln schrittweise erklärt.

Zur Installation von Log4J ist zu bemerken, daß die gelieferten Dateien in ein zentrales Verzeichnis entpackt werden müssen. Danach muss der CLASSPATH angepasst werden. Hier ist zu beachten, daß der richtige Pfad der .jar Datei angegeben wird.

Beispiel: *CLASSPATH = C:\Programme\Java\jakarta-log4j-1.2.8\dist\lib\log4j-1.2.8.jar;.*

5.2.1 Logger

Logger in Log4J werden wie in der Standard API hierarchisch angeordnet. Der RootLogger steht in der Hierarchie ganz oben. Nachfolgende Logger werden, wie in der Java API, hierarchisch benannt. So ist "logger.unterlogger" Vorgänger von "logger.uterlogger.speziallogger". Dies ist z.B. vergleichbar mit: "java.util" und "java.util.Vector".
Zu beachten ist, daß der RootLogger nicht namentlich angesprochen werden kann. Hier muss die Funktion Logger.GetRootLogger benutzt werden. Eigenschaften von Loggern werden, wenn sie nicht genauer spezifiziert sind, vom Vorgänger geerbt. Die Eigenschaften eines Loggers können mit Logger.GetLogger abgefragt werden.

5.2.2 Log Levels

Apache Log4j verfügt über sieben verschiedene LogLevels. Diese sind im Hauptpaket in der Klasse Level näher spezifiziert.

DEBUG	(niedrigste Stufe)
INFO	
WARN	
ERROR	
FATAL	(höchste Stufe)

ALL	(alles wird gelogged)
OFF	(Logging ausgeschaltet)

Theoretisch könnte man noch eigene LogLevels hinzufügen, indem man die Klasse Level ableitet. Davon wird jedoch von der Apache Software Foundation abgeraten.
LogLevel müssen nicht explizit zugewiesen werden. Wird kein LogLevel zugewiesen, wird der Level des Vorgängers übernommen. Sollte dieser auch keinen explizit zugewiesen Level haben, wird die Hierarchie bis zum bis zum root-logger durchsucht.
Es werden nur Level gelogged, die das LogLevel des Loggers mit einschließen.
Eine Logging Anfrage vom Level p wird nur gelogged, wenn der Logger das Level q hat und p>=q ist.

5.2.3 Appender

Die Appender aus dem Log4J Paket sind mit den Handlern aus der Java Logging API vergleichbar. Sie definieren das Ausgabeziel für das Logging. Appender werden aus der abstrakte Klasse AppenderSkeleton abgeleitet. Dies hat den Vorteil, daß man eigene Appender entwickeln kann.
Appender sind wie Handler additiv, d.h. man kann sie nach einander schalten.

Folgende Appender sind u.A. im Log4J Standardpaket enthalten:

Appender	
ConsoleAppender	Logging auf die Konsole
SyslogAppender	Logging an den Syslog Daemon
FileAppender	Logging in eine Datei
SocketAppender	Logging auf ein Socket
JMSAppender	Logging JMS – Objekt vom Typ javax.jms.Message
NTEventLogAppender	Logging für Windows NT Plattformen
AsyncAppender	Asynchroner Appender
JDBCAppender	Logging in eine Datenbank
SMTPAppender	Logging per Mail versenden

5.2.4 Konfiguration

Anwendungen können sehr viele Logging Statements haben, jedoch werden im Anwendungsfall nur sehr wenige davon benötigt. Um nicht den kompletten Sourcecode abzuändern, kann man den Logger so konfigurieren, dass nur bestimmte Events gelogged werden (siehe z.B. Kapitel 5.2.2 – LogLevels). Damit die Konfiguration nicht in dem Sourcecode fest eingebettet sein muss, gibt es die Möglichkeit externe Konfigurationsdateien anzulegen. Dies geht im XML Format oder im Java Properties Format.

Hier ein Beispiel einer kleinen Datei im JavaProperties Formats:

```
# Setzt root logger level auf DEBUG und den Appender auf A1.
log4j.rootLogger=DEBUG, A1

# Appender A1 wird als ConsoleAppender definiert.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

...
```

Die Konfigurationsdatei kann dann mit der Methode PropertyConfigurator.configure eingelesen werden.

5.3 Implementierungsbeispiel

Konfigurationsdatei (TestLog.conf)

```
# Setzt root logger level auf DEBUG und den Appender auf A1.
log4j.rootLogger=DEBUG, A1

# Appender A1 wird als ConsoleAppender definiert.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

#Layout definieren
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=[%t] %-5p %c %x - %m%n
```

Beispielimplementation (TestLog.java)

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

class TestLog
{
     public static void main(String[] args)
     {
          // Logger Instanz "com.foo" erzeugen
          Logger  logger = Logger.getLogger("com.foo");

          // Konfigurationsdatei laden
          PropertyConfigurator.configure("TestLog.conf");


          // Zweiten Logger erzeugen, der Daten von erstem erbt.
          Logger barlogger = Logger.getLogger("com.foo.Bar");

          // Dieser Event wird ausgegeben, da WARN >= INFO.
          logger.warn("Low fuel level.");

          // Dieser Event wird nicht ausgegeben, da DEBUG < INFO.
          logger.debug("Starting search for nearest gas station.");

          // Event wird angezeigt, da INFO >= INFO.
          barlogger.info("Located nearest gas station.");

          // Dieser Event wird nicht angezeigt, da DEBUG < INFO.
          barlogger.debug("Exiting gas station search");
     }
}
```

Ausgabe:

```
[main] WARN  com.foo  - Low fuel level.
[main] INFO  com.foo.Bar  - Located nearest gas station.
```

6. Fazit

Abschließend ist zu sagen, daß Logging eine sehr wichtige Methode zur Überwachung von Programmabläufen ist und nicht nur in der Entwicklungsphase eingesetzt wird.
Ein großer Vorteil von Logging ist, dass man LogLevels setzen und diese entsprechend filtern kann.

Java kommt von Haus aus mit einer guten Logging API, jedoch bieten Drittanbieter, wie die Apache Software Foundation mit dem Jakarta Log4J Projekt, umfangreichere Lösungen zum Thema Logging an.

Es ist sehr schwer eine Empfehlung für ein Produkt auszusprechen. Dies sollte man von Fall zu Fall entscheiden und sich die genauen Anforderungen an sein Logging vor Augen führen.

Fakt ist jedoch, daß Log4J sehr populär und gefragt ist, denn ansonsten wäre es sicher nicht auf diverse andere Programmiersprachen übertragen worden. Man kann also davon ausgehen, daß Log4J auf jeden Fall weiterentwickelt wird.

Ein Nachteil von Log4j ist, daß man es zusätzlich installieren muß. Dies sollte jedoch kein großes Problem darstellen.

7. Literaturverzeichnis

Java Logging API
http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/overview.html

Jakarta Projekt
http://jakarta.apache.org/

Log4J Homepage
http://jakarta.apache.org/log4j/

hier sind insbesondere folgende Seiten zum Einsatz gekommen:
http://jakarta.apache.org/log4j/docs/manual.html
http://jakarta.apache.org/log4j/docs/api/index.html

Homepage von Winsyslog
http://www.winsyslog.com