Java/GTK

Elmar Ludwig (elmar.ludwig@uos.de)

Fachbereich Mathematik/Informatik
Universität Osnabrück

Motivation

Der erste Versuch von Sun, eine einheitliche grafische Oberfläche für Java zur Verfügung zu stellen, war das AWT (Abstract Window Toolkit) Paket. Ziel dabei war, auf allen unterstützten Plattformen eine einheitliche Programmierschnittstelle für das dort „üblicherweise“ verwendete Toolkit zu bieten (Windows, Macintosh, X11+Motif), auch wenn ein Fenster-System wie X11 eigentlich kein bevorzugtes Toolkit kennt.

Die – durchaus beabsichtige – Folge davon war das, was heute als „native Look & Feel“ bezeichnet wird: Jede Plattform verwendet zur Implementierung der Grafik jeweils ihre eigenen Oberflächenelemente, mit denen der Anwender der Applikation bereits vertraut sein sollte. Leider hat dieser Ansatz zwei gravierende Probleme deutlich gemacht, die Sun in der zweiten Runde mit dem JDK-1.1 und vor allem Swing zu vermeiden versucht hat:

- Durch die Verwendung der Plattform-eigenen Komponenten schlichen sich stellenweise viele kleine (aber bei der Programmierung sehr lästige) Plattform-Unterschiede ein, so daß die Programme trotz einheitlicher Programmierschnittstelle nicht auf allen Plattformen gleich funktionieren. Typische Beispiele dafür sind Unterschiede in den auftretenden Events bzw. deren Reihenfolge und Unterschiede in der grafischen Darstellung.
- Das wesentliche Problem ist aber, daß das AWT aufgrund des Designs dem Programmierer eben nur die Schnittmenge der Fähigkeiten aller Plattformen zur Verfügung stellen kann. Da die Programme portabel sein sollen, mußte man sich auf die Komponenten beschränken, die auf allen Plattformen existieren. Daher fehlen dem AWT einfach viele nützliche Oberflächenelemente.

Ein anderes Problem ist die immer noch wenig elegante Eventbehandlung im AWT. Die Art der Eventverarbeitung wurde zwar für die Java Version 1.1 noch einmal komplett überarbeitet (Observer-Modell, Listener), ist aber leider noch weit von einer eleganten Lösung entfernt, da die Events über

die Listener-Interfaces nur an Methoden mit fest vorgegebenen Namen (und Argumenttypen) zugestellt werden können.

Aufgrund dieser fehlenden Flexibilität muß man in der Realität für nahezu jede Reaktion auf einen Event von der Oberfläche eine eigene Klasse im Programm erfinden, was die Programmstruktur nicht gerade übersichtlicher macht. Sun hat dafür die Idee von „inneren Klassen“ in die Sprache Java integriert, womit man – allerdings mit einer sehr gewöhnungsbedürftigen Syntax – mit sehr wenig Programmtext leicht neue Klassen erzeugen kann.

Obwohl die Sprache Java über das `java.lang.reflect` Paket durchaus die Möglichkeit zu mehr Flexibilität in diesem Bereich hätte, wird davon sowohl beim AWT wie auch bei der Weiterentwicklung zu Swing wohl aus Effizienzgründen kein Gebrauch gemacht.

Andererseits sieht man dem Swing-Paket deutlich die Altlasten an, die durch das Aufsetzen auf der alten Klassenhierarchie des AWT entstehen:

Die neuen Swing-Komponenten stammen nicht etwa alle von `JComponent` ab, wie man eigentlich erwarten würde, sondern wesentliche Klassen sind jeweils einfach unterhalb der entsprechenden AWT-Komponenten eingehängt (`JApplet`, `JFrame` etc.). Das führt aber leider dazu, daß Methoden, die in `JComponent` vorhanden sind, nur für einen Teil der Swing-Komponenten zur Verfügung stehen oder aber in vielen Klassen implementiert werden müssen.

Designziele

Ich möchte hier einen alternativen Ansatz für den Anschluß von Java an eine moderne grafische Oberfläche vorstellen, der die oben skizzierten Probleme mit den vorhandenen Oberflächen für Java (d.h. im wesentlichen AWT und Swing) vermeidet. Man sollte dabei mindestens die folgenden Designziele erfüllen können:

- unabhängig von `java.awt.*`

  Die Implementierung sollte völlig unabhängig von den vorhandenen Klassen in den Paketen `java.awt` und `javax.swing` sein, damit man die bereits oben erwähnten „Altlasten“ des AWT (speziell bezüglich der Klassenhierarchie) nicht immer mit sich herumschleppt. Allerdings muß man – wenn man einen radikalen Schnitt macht – dafür in gewisser Weise einen Preis in Form von fehlender Kompatibilität bezahlen:

  Die Klasse `java.applet.Applet` stammt von `java.awt.Container`

ab, ist also ebenfalls eine AWT-Komponente, und kann damit *nur* AWT-Komponenten enthalten. Damit verliert man leider zunächst die Fähigkeit, die eigenen Klassen in einem Java-Applet verwenden zu können.

- Plattformunabhängigkeit

  Sofern das zugrundeliegende Toolkit (für die Test-Implementierung zur Zeit das GTK) auf verschiedenen Plattformen verfügbar ist, sollte die Implementierung der Java-Anbindung auf all diesen Plattformen auch gleich funktionieren. In diesem Punkt kann man sich das Leben deutlich leichter machen, wenn man die Implementierung so weit wie möglich direkt in Java realisiert.

- verwendbar im Umfeld von (vielen) Threads

  Threads sind – im Gegensatz zu vielen anderen Programmiersprachen – in Java fester Bestandteil der Sprache und die Anweisungen zur Thread-Synchronisation sind ebenfalls in Java integriert.

  Allerdings machen sowohl das alte AWT als auch dessen Nachfolger Swing sehr wenig Gebrauch davon. Alternative Betriebssysteme wie z.B. BeOS machen dagegen vor, daß es sehr wohl anders gehen kann und man durchaus Vorteile von vielen Threads in einer Oberfläche hat.

- effiziente Implementierung der fertigen Komponenten

  Sun hat, um das Problem der kleinen (aber sehr störenden) Plattformunterschiede zu vermeiden, das Swing-Paket komplett in der Sprache Java implementiert, wobei es jedoch auf dem vorhandenen AWT aufsetzen muß. Leider hat das dann zu den für Java typischen Performance-Problemen geführt, womit Swing auf einem etwas älteren Rechnersystem kaum bzw. nur sehr zäh zu benutzen ist.

  Wenn man das Problem der Plattformunterschiede auf einer anderen Ebene in den Griff bekommen kann (unterhalb von Java), kann man diese Performance-Probleme zumindest für die fertigen Komponenten vermeiden, da man wesentliche Teile der Darstellung dann z.B. in C erledigen kann, bzw. fertige Komponenten das schon tun. Allerdings kann man dann nicht mehr mit dem – bei Sun so beliebten – Slogan „*100% pure Java*“ werben...

Was bringt nun Java/GTK?

Das GTK (die Abkürzung steht für das *Gimp Toolkit*, siehe [4]) ist eine Bibliothek für C und andere Programmiersprachen zum Erstellen von grafischen Benutzeroberflächen auf verschiedenen Systemen (zur Zeit werden X11 und Win32 unterstützt, Versionen für BeOS und Mac OS sind noch in Entwicklung).

Ich habe mit dem GTK eine relativ einfache Beispiel-Implementierung einer Anbindung an die Sprache Java realisiert, um dabei leicht verschiedene Ansätze ausprobieren zu können. Im Gegensatz zu der von Sun für X11 verwendeten *Motif*-Bibliothek bietet das GTK eine deutlich einfachere Programmierschnittstelle an und hat außerdem den Vorteil, daß der Code unverändert gleich auf mehreren Plattformen läuft.

Aber: Alle hier vorgestellten Ideen bezüglich der Integration einer externen Grafik-Bibliothek in Java sind ohne weiteres auch auf jede andere Bibliothek (weitere Kandidaten dafür wären Tk oder Qt) übertragbar.

Mögliche Ansätze

Man hat im Prinzip viele verschiedene Möglichkeiten der Anbindung des GTK (bzw. einer C-Bibliothek allgemein) an Java. Die meisten müssen allerdings an irgendeiner Stelle auf das *Java Native Interface* (JNI) [3] aufsetzen, denn sonst kann man Funktionen in einer C-Bibliothek nicht direkt verwenden. Über das JNI können einzelne Methoden einer Java-Klasse als *native* markiert und dann in C oder C++ implementiert werden.

Wenn man den Weg über das Java Native Interface wählt, kann man zunächst auf drei verschiedene Ansätze kommen, die ich im folgenden jeweils kurz vorstellen möchte.

Möglichkeiten zur Anbindung an Java mittels JNI:

- Analog zum vorhandenen AWT mit Peer-Klassen, die einen Teil der Methoden auf entsprechende plattformspezifische C-Funktionen abbilden. Da hier viele Threads potentiell gleichzeitig aufrufen können, die Abarbeitung auf der C-Seite aber synchron (ein Aufruf nach dem anderen) passieren muß, muß man Zugriffe irgendwo synchronisieren.
- Man bildet die Funktionen der grafischen Komponenten einfach 1:1 auf Java-Methoden ab, die dann alle native implementiert sind. Auch hier

muß man den gleichzeitigen Zugriff aus vielen Threads entsprechend synchronisieren.

- Alternativ kann man auch alle Grafik-Klassen komplett in Java zu implementieren versuchen. Der Trick hierbei ist, die Methodenaufrufe über *eine* zentrale Stelle an die C-Seite weiterzuleiten und auch die Antworten entsprechend zurück zu schicken. Dadurch erreicht man eine sehr lose Koppelung zwischen der Java- und der C-Welt.

Die aktuelle Test-Implementierung verwendet den dritten Ansatz mit „Marshalling“ (dazu später mehr), dabei ist die Verwendung von JNI auf ein Minimum reduziert.

Vergleich der verschiedenen Ansätze

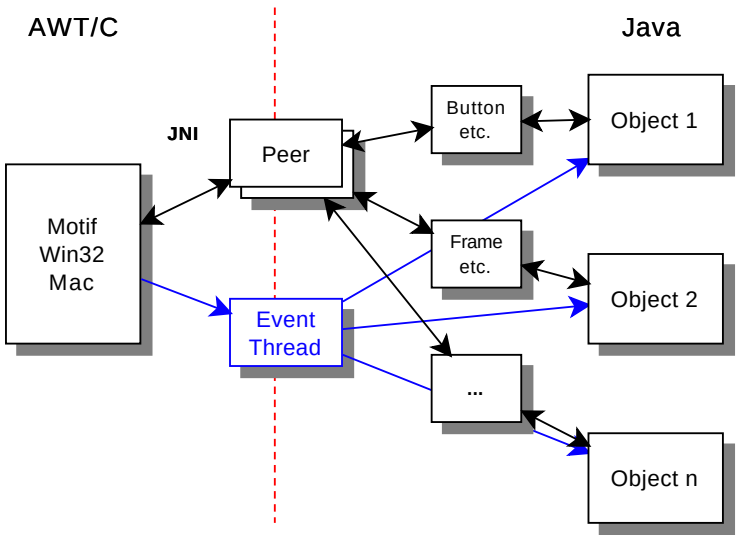

Abbildung 1: Ansatz des AWT (bzw. Swing)

Der „klassische“ Ansatz, den das AWT (und damit auch Swing) verfolgt, verwendet sogenannte *Peer*-Klassen (siehe Abbildung 1): Für jede Klasse, die ein Objekt der Benutzeroberfläche repräsentiert (wie z.B. `Button`), gibt es eine zugehörige Peer-Klasse (wie `ButtonPeer`), die jeweils den plattformspezifischen Code verkapselt. Bestimmte Teile der Peer-Klassen sind dabei in native-Methoden implementiert, z.B. das Erzeugen und Manipulieren der darunter liegenden grafischen Komponenten.

Für die Eventverarbeitung gibt es einen eigenen Event-Thread, der alle auftretenden Events vom System entgegen nimmt und selbst bei den dafür regi-

strierten Listener-Objekten entsprechende Methoden aufruft. Daraus ergibt sich natürlich, daß der Event-Thread – und damit die gesamte Oberfläche der Applikation – blockiert, falls eine Listener-Methode nicht rasch zu Ende geht. Parallel dazu gibt es intern noch einen Paint-Thread, der sich bei Bedarf um die Darstellung der Komponenten kümmert.

Hier ergibt sich nun folgendes Problem: Die Java VM (*virtual machine*) verwendet zur Abbildung der Java-Threads in der Regel echte Threads auf der Plattform, auf der das Programm ausgeführt wird. Für den Fall, daß die Plattform selbst keine Threads unterstützt, kann die JVM diese auch in Software emulieren (die sogenannten *green threads*). Leider sind aber die wenigsten C-Bibliotheken dafür eingerichtet, Aufrufe aus mehreren Threads gleichzeitig verkraften zu können, daher muß man zwangsläufig die gesamte Interaktion mit der Grafik-Bibliothek auf C-Ebene so synchronisieren, daß immer nur ein Thread zur Zeit in diesem Bereich aktiv sein kann. Das immer richtig hinzubekommen, ist nicht ganz einfach.

Ein erster Ansatz mit dem GTK als Toolkit könnte etwa wie in Abbildung 2 dargestellt aussehen:

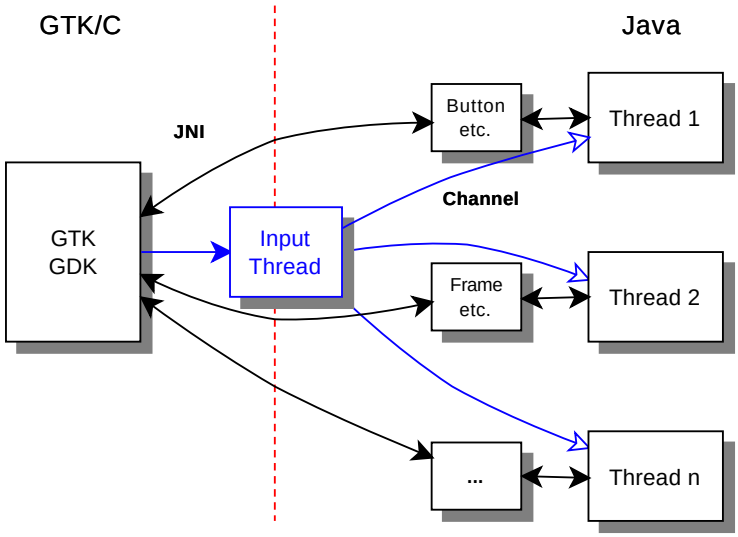

Abbildung 2: Ein erster Versuch mit GTK

Die Idee ist hier, daß man die Events über den Event-Thread nur noch an andere Threads verteilt, anstatt direkt vom Event-Thread aus die Listener-Methoden aufzurufen. Damit können diese das Antwortverhalten der Oberfläche nicht mehr negativ beeinflussen: Falls eine der dabei beteiligten Handler-Methoden lange rechnet, stapeln sich höchstens bei diesem Thread die

Events, es kann aber nichts blockieren.

Um die Events leicht verteilen zu können, bietet sich das schon bei Plan 9 und Inferno verwendete *Channel*-Konzept an: Ein Channel ist so etwas wie ein „Nachrichtenkanal“ zwischen Threads, die durch den Channel Objekte (und damit beliebige Daten) austauschen können.

Man bekommt auf diese Weise fast automatisch ein Modell, in dem auch viele Threads auf verschiedene Events von der Oberfläche warten und reagieren können. Beispielsweise kann man leicht ein Programm realisieren, daß für jedes Fester einen oder sogar mehrere eigene Threads verwendet, so daß rechenintensive Operationen in einem Fester die anderen Fester bzw. die anderen in diesem Fenster aktiven Threads nicht betreffen können.

Voraussetzung dafür ist jedoch, daß man nun für jeden Thread eine eigene Hauptschleife besitzt, die Events nur für diesen Thread entgegen nimmt und dafür registrierte Aktionen auslöst.

Leider hat man auch hier immer noch das Problem der Thread-Synchronisation zu lösen, das Problem hat sich lediglich von den (jetzt nicht mehr vorhandenen) Peer-Klassen direkt in die Klassen der Grafik-Komponenten verschoben, die nun viele native-Methoden enthalten müssen.

Die Peers sind bei diesem Ansatz nicht mehr erforderlich, da deren ursprüngliche Aufgabe – einen Ausgleich zwischen den Plattformunterschieden herzustellen – hier nicht erforderlich ist.

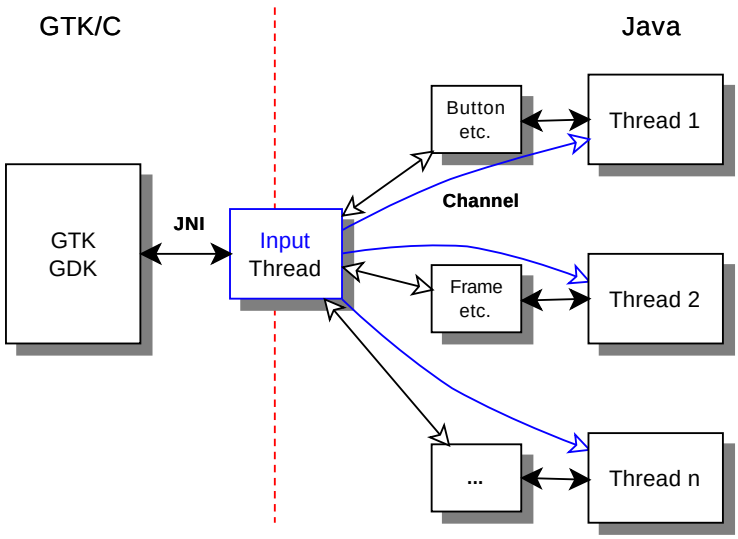

Abbildung 3: Der aktuelle Ansatz mit GTK

Man kann jetzt aber noch einen kleinen Schritt weiter gehen (Abbildung 3) und auch noch die Methodenaufrufe bei den einzelnen Komponenten als „Requests“ wieder über einen `Channel` an den InputThread schicken. Dadurch löst man recht elegant zwei Probleme:

Man braucht überhaupt keine native-Methoden mehr in den Grafik-Komponenten, da die *gesamte* Kommunikation mit der Plattform-Bibliothek über die Klasse `InputThread` läuft. Diese Klasse muß nun als einzige noch einige wenige native-Methoden enthalten.

Da im Programm nur genau eine Instanz der Klasse `InputThread` existiert, erreicht man außerdem die notwendige Thread-Synchronisation bereits dadurch, daß die Aufrufe über den InputThread verschickt werden. Der Channel sorgt hier dafür, daß die Aufträge stets nacheinander und nicht gleichzeitig ankommen. Dieses Konzept ist mit dem Begriff „Marshalling“ gemeint, das oben bereits kurz erwähnt wurde.

Theoretisch ist natürlich auch eine Anbindung ganz ohne JNI denkbar: Man hat dann im Prinzip ein sehr ähnliches Modell wie bei dem gerade beschriebenen Fall, die gesamte Kommunikation wird aber nicht mehr über Channels abgewickelt, sondern über einen äquivalenten Mechanismus (Sockets, Pipelines) in einen separaten Prozeß verschickt, der sich um die Events und die Darstellung kümmert. Letztlich hat man damit einen externen „Grafik-Server“ und verteilt alle Aufrufe als *Remote Procedure Calls*. Dieser Ansatz hat aber den wesentlichen Nachteil, daß sehr viel Prozeßkommunikation erforderlich ist, wodurch man mit ernsthaften Performance-Problemen rechnen müßte.

Implementierung

Die entscheidende Arbeit auf Java-Seite leisten die Klassen `InputThread`, die sich um die Weiterleitung der Events und Requests kümmert, und `Base` (die Basisklasse aller grafischen Komponenten) sowie `RunLoop`. RunLoop realisiert dabei die Arbeitsschleife, die ankommende Events innerhalb eines Threads an die dafür registrierten Objekte verteilt.

Üblicherweise hat jeder Thread eine eigene RunLoop (die *default RunLoop*), obwohl das nicht fest vorgegeben ist. Theoretisch können sich auch mehrere Threads eine RunLoop teilen oder ein Thread kann – je nach Programmlogik – nacheinander verschiedene RunLoops betreiben.

Die gesamte Kommunikation zwischen den beteiligten Objekten läuft wie beschriebenen über *Channels*, das sind praktisch Message-Queues für Objek-

te. Da diese Channels beliebig viele Objekte speichern können, kann dabei nichts blockieren, außer natürlich, wenn man von einem leeren Channel zu lesen versucht.

Die einzige Klasse mit native-Methoden in diesem Design ist InputThread, da die gesamte Kommunikation zwischen Java (Requests an die Grafik) und dem native-Teil (ankommende Events zur Weiterleitung an die RunLoops) in dieser einen Klasse verkapselt ist. Bei Programmstart wird implizit genau ein InputThread-Objekt erzeugt und gestartet, das auch für das Weiterlaufen der Applikation sorgt, wenn sich der Main-Thread beendet.

Man sieht hier, wie jeweils abwechselnd Events und Requests verarbeitet werden. `processEvents` und `doRequest` sind dabei native-Methoden. Es zeigt sich, daß ein Request hier im Prinzip nichts anderes ist als ein Object-Array, in dessen erstem Element der diesem Request zugeordnete Antwort-Channel eingetragen ist bzw. der Wert `null`, wenn keine Antwort erwartet wird. Wie das verwendet wird, zeigt zum Beispiel der folgende Ausschnitt aus der Klasse `Label` (Abbildung 5).

`initWithArgs` hat nur die Aufgabe, eventuell für das Toolkit gedachte Kommandozeilenargumente zu übermitteln, beispielsweise „–display“ unter X11, hat aber sonst keine weitere Bedeutung. Interessant sind die Methoden `processEvents` und `doRequest` (Abbildung 6).

`processEvents` sorgt hier implizit über den GTK-Callback Mechanismus dafür, daß alle ankommenden Events über die richtigen Channels an die darauf wartenden Java-Threads weitergeleitet werden, während `doRequest` in der Gegenrichtung ankommende Requests über eine interne Tabelle an die dazu gehörigen C-Funktionen verteilt.

Beispiel

Um beurteilen zu können, ob man mit diesen Ideen zu einem eleganteren Programmiermodell als bei AWT/Swing kommt, sollte man natürlich zunächst das klassische „Hello, World“-Programm implementieren. Dieses könnte dann zum Beispiel etwa wie in Abbildung 7 gezeigt aussehen.

Die hier verwendete `quit`-Methode bekommt beim Aufruf als Argument den Auslöser des Events, hier also hier das Fenster, als `Object` übergeben, was an dieser Stelle aber nicht weiter gebraucht wird.

Natürlich kann man das gleiche Programm auch ganz analog zu der Eventbehandlung im AWT mit einer inneren Klasse als ActionListener aufschreiben,

```
package gtk;

public class InputThread extends Thread
{
    public static Channel chan = new Channel();

    static { System.loadLibrary("native"); }

    public static void sleep ()
    {
        try { Thread.sleep(20); } catch (InterruptedException e) {}
    }

    public native Object initWithArgs (String args[]);
    protected native void processEvents ();
    protected native Object doRequest (Object request);

    public void run ()
    {
        for (;;) {
            Object req[];

            processEvents();
            while ((req = (Object[]) chan.available()) != null) {
                Object result = doRequest(req);

                if (req[0] != null) ((Channel) req[0]).send(result);
            }
            sleep();
        }
    }
}
```

Abbildung 4: InputThread.java

benötigt dann aber zwei Klassen statt einer.

```
public void setText (String label)
{
  InputThread.chan.send(new Object[] {null, this, "Label_setText", label});
}
public String getText ()
{
  InputThread.chan.send(new Object[] {chan(), this, "Label_getText"});
  Object _res = chan().recv();
  return (String)_res;
}
```

Abbildung 5: Auszug aus Label.java

```
JNIEXPORT void JNICALL
Java_gtk_InputThread_processEvents (JNIEnv *env, jobject self)
{
    while (gtk_events_pending())
        gtk_main_iteration();
}

JNIEXPORT jobject JNICALL
Java_gtk_InputThread_doRequest (JNIEnv *env, jobject self, jobject array)
{
    jstring fkt_name = ELEMENT(array, 2);
    const char *name = (*env)->GetStringUTFChars(env, fkt_name, 0);
    jobject result = find_entry(name)(env, array);

    (*env)->ReleaseStringUTFChars(env, fkt_name, name);
    return result;
}
```

Abbildung 6: native-Teil der Klasse InputThread

```
import gtk.Application;
import gtk.Label;
import gtk.RunLoop;
import gtk.Window;

public class Hello
{
    public static void main (String args[])
    {
        new Application(args);
        new Hello();
        RunLoop.defaultRunLoop().run();
    }

    public Hello ()
    {
        Window window = new Window(Window.TOPLEVEL);
        Label label = new Label("Hello, World!");

        try {
            window.connect("destroy", this, "quit");
        } catch (NoSuchMethodException e) {
            throw new RuntimeException(e.toString());
        }

        window.add(label);
        window.showAll();
    }

    public void quit (Object obj)
    {
        System.err.println("exiting now...");
        System.exit(0);
    }
}
```

Abbildung 7: Hello.java

Literaturverzeichnis

[1] Java Homepage (bei Sun), *http://www.javasoft.com/*

[2] Java Tutorial (bei Sun), *http://java.sun.com/docs/books/tutorial/*

[3] Java Native Interface Specification,
http://java.sun.com/j2se/1.3/docs/guide/jni/

[4] Homepage des GTK (Tutorial, Reference Documentation),
http://www.gtk.org/