Von Ameisen, Fühlern und Eklipsen

J2ME MIDlet-Entwicklung mit Eclipse, Ant und Antenna

Kai Tödter

J2ME (Java 2 Micro Edition) und insbesondere MIDP (Mobile Device Information Profile) ist mittlerweile auf Millionen von mobilen Geräten verfügbar. Insbesondere auf Handys ist J2ME so sehr verbreitet, dass ein neuer interessanter Markt auch für professionelle Java-Applikationen entstanden ist. Es gibt eine Vielzahl professioneller Entwicklungsumgebungen, die das Erstellen von MIDlets vereinfachen und unterstützen. Dieses Tutorial beschreibt, wie man mit Hilfe von frei verfügbaren Werkzeugen, insbesondere Eclipse, Ant und Antenna, eine professionelle Entwicklungsumgebung für MIDlets konfigurieren und effizient nutzen kann.

Eine kleine Einführung in J2ME

Bevor wir mit dem eigentlichen Tutorial beginnen, hier erstmal eine kleine Einführung in J2ME, die Java 2 Micro Edition. Anders als seine großen Brüder, J2SE und J2EE, ist die Micro Edition wiederum aufgeteilt in Konfigurationen und Profile. In unserem Fall interessieren wir uns für die Konfiguration CLDC (Connected Limited Device Configuration) und das Profil MIDP (Mobile Information Device Profile). MIDP 1.0 wurde im Dezember 2000 verabschiedet und ist mittlerweile auf Millionen von Endgeräten, z.B. Handys, verfügbar. CLDC spezifiziert die Mindestanforderungen an die Hardware der Geräte, z.B. Speicher-, Display-, Eingabe- und Netzwerk-Voraussetzungen, sowie die Eigenschaften der Java Virtual Machine (JVM) und die mitgelieferten Java-Bibliotheken. Im Falle von CLDC sind das die Packages java.lang, java.io, java.util und java.microedition.io. Die Hauptunterschiede zwischen J2SE und CLCD 1.0 basierten Profilen aus Sicht des Java-Entwicklers sind:

- Keine Fließkommaarithmetik
- Eingeschränkte Fehlerbehandlung (Exception-Handling ist vorhanden, aber nur eingeschränkte Error-Klassen verfügbar)
- Kein Java Native Interface (JNI)
- Keine benutzerdefinierten Classloader
- Keine Reflection
- Keine Thread-Gruppen
- Keine Finalisierung (kein object.finalize())
- Keine Weak References
- 2 geteilter Verifikationsprozess (siehe unten)

Genaueres zu den genannten Einschränkungen finden Sie in der CLDC-Spezifikation [CLDC]. Abgesehen von diesen Einschränkungen ist die Java-Entwiclung für CLDC/MIDP der Java Desktop-Entwicklung sehr ähnlich, die größte Umstellung im Entwicklungsprozess ist das Verifizieren (preverify) der Java-Klassen auf Entwicklungsseite, bevor diese auf einem Gerät ausgeführt werden können. Dazu kommen wir später noch. Wer tiefer in die Details von CLDC Bytecode-Verifikation einsteigen möchte, dem sei der Appendix 1 der CLDC 1.1-Spezifikation [CLDC1.1] empfohlen. Doch kommen wir wieder zurück zu MIDP. Es gibt mittlerweile schon die Version 2.0 des Mobile Information Device Profiles. Dieses Tutorial ist sowohl für MIDP 1.0, wie auch für 2.0 anwendbar. In MIDP wird ein Applikations-Modell eingeführt, das wohl am ehesten mit dem Applet-Modell zu vergleichen ist. Analog zu Applets gibt es unter MIDP so genannte MIDlets, die durch ein vordefiniertes API auf den mobilen Endgeräten ablaufen und dort eingeschränkte Rechte besitzen. Neben der eigentlichen Programmierung der MIDlets gibt es noch bei der Auslieferung einige Besonderheiten. Die verifizierten Java-Klassen ein oder mehrerer MIDlets werden zur Auslieferung in ein JAR-File gepackt, welches genau dem JAR-Standard von J2SE entspricht. Weiterhin kann optional ein so genannter Java Application Descriptor mitgeliefert werden, der

Meta-Informationen über die MIDlets enthält. Dies ist sehr wichtig, da ja die Ressourcen auf den mobilen Geräten sehr eingeschränkt sind und Software-Download oft auch recht teuer ist. Vor dem Download wird also erst das JAD-File interpretiert und dem Endbenutzer die wesentlichen Daten wie Name, Größe, etc. dargestellt. Dieser kann dann entscheiden, ob er das eigentliche JAR-File herunterladen will. Hier ein Beispiel für ein JAD-File:

```
MIDlet-Name: TestMIDlet
MIDlet-Vendor: Kai Toedter
MIDlet-Version: 1.0
MIDlet-Jar-Size: 897
MIDlet-Jar-URL: TestMidlet.jar
MIDlet-1: TestMidlet, ,com.toedter.j2me.test.TestMIDlet
```

Laut Spezifikation muss das JAD-File immer die Extension .jad haben. Man kann, wie schon erwähnt, mehrere MIDlets zu einer so genannten MIDlet Suite in ein JAR-File packen. Diese MIDlets teilen sich dann eine JVM auf dem mobilen Endgerät und können Ressourcen austauschen. Weiterhin lassen sich benutzerdefinierte Attribute über die Methode `MIDlet.getAppProperty(String key)` aus dem JAD-File lesen, so könnte z.B. ein Attribut

```
Logging: fine
```

definiert werden. Im Internet finden Sie unter [Brontofundus] eine sehr gute deutschsprachige Einführung in das Thema J2ME.

J2ME Entwicklungsumgebungen

Es gibt einige kommerzielle Produkte, welche die MIDP-Entwiclung sehr gut unterstützen. Zum einen gibt es freie Versionen wie z.B. Borland JBuilder 8 plus Mobile Set [JBuilder] oder Sun ONE Studio 4 update 1, Mobile Edition [SunONE]. Zum anderen gibt es kostenpflichtige kommerzielle IDEs (Integrated Development Environments), wie z.B. IBMs WebSphere Studio Device Developer [WSDD]. WSDD ist eine Erweiterung von Eclipse und unterstützt auch die Entwicklung von MIDlets. Ein weiteres Produkt JetBrains IntelliJ IDEA [IntelliJ], für das es auch J2ME-Plug-ins gibt. Dieses Tutorial geht nicht weiter auf diese Entwicklungsumgebungen ein sondern beschäftigt sich mit einer anderen frei verfügbaren Entwicklungsumgebung: Eclipse [Eclipse]. Eclipse erfreut sich bei Java-Entwicklern zunehmender Beliebtheit, zum einen wegen der freien Verfügbarkeit, zum anderen wegen der guten Performance und der flexiblen, erweiterbaren Architektur. Durch eine Vielzahl frei verfügbarer Plug-ins lässt sich mit Eclipse eine auf viele individuelle Wünsche angepasste Entwicklungsumgebung konfigurieren. Es gibt mittlerweile einige Ansätze für J2ME-Plug-ins, z.B. von Siptech [Siptech] oder das Open-Source Eclipseme-Projekt [Eclipseme], das sich allerdings noch in einer sehr frühen Phase befindet. Diese Plugins sind viel versprechend, dieses Tutorial geht aber nicht weiter auf sie ein. Vielmehr wird der Focus auf ein weiteres freies Java-Werkzeug gelegt, nämlich Ant, das dazu standardmäßig in der Eclipse SDK-Distribution enthalten ist. Ant, ein Open-Source-Projekt der Apache Faundation, ist ein Werkzeug, welches den Build-Prozeß von Java-Applikationen unterstützt. Von der Idee her ähnelt es am ehesten dem allseits bekannten Make-Tool. Bei Ant erfolgt die Beschreibung der Build-Targets durch eine XML-basierte Beschreibung, die relativ intuitiv zu verstehen ist. Gegenüber proprietären Build-Prozessen, welche sehr eng an bestimmte IDEs gebunden sind, hat Ant einige entscheidende Vorteile:

- Durch die IDE-Unabhängigkeit kann der Build-Prozess jederzeit außerhalb der IDE angestoßen werden, automatische Builds sind also kein Problem
- Ant bietet ein reichhaltiges Angebot von Standard-Funktionen
- Ant lässt sich über ein Java-API leicht erweitern und an individuelle Bedürfnisse anpassen.

Gerade der letzte Punkt ist für und von sehr großer Bedeutung, da Ant standardmäßig keine J2ME-spezifischen Eigenheiten des Build-Prozesses unterstützt. Jörg Plaumann greift genau hier an und stellt mit seinem Werkzeug Antenna [Antenna] eine hervorragende Ant-Erweiterung kostenlos zur Verfügung, mit der sich die meisten J2ME-spezifischen Aufgaben gut lösen lassen.

Installation von Eclipse, Ant, Antenna und dem WTK

Nachdem wir uns in vorherigen Abschnitten mit der Theorie befasst haben, installieren wir erstmal unsere Entwicklungsumgebung. Zunächst brauchen wir ein Java 2 Standard Edition (J2SE) SDK. Dieses können wir kostenlos von Sun [J2SE] herunterladen und installieren. Ich empfehle, die aktuelle Version 1.4.2 zu benutzen. Das Eclipse-SDK ist unter [Eclipse] zu finden, Ant ist dort schon enthalten. Alle Tools sind plattform-unabhängig, der Einfachheit halber werde ich alle konkreten Beispiele und Screenshots für die Windows-Versionen zeigen. Die Windows-Version von Eclipse wird als einfaches Zip-File geliefert, das nur ausgepackt werden muss. Einzige Voraussetzung ist eine installierte Java 2 Laufzeitumgebung, die wir ja mit dem Java 2 SDK schon haben. Nun brauchen wir noch die aktuelle Version von Antenna, die wir im Downloadbereich von [Antenna] finden. Wir brauchen sowohl die Sourcen (daraus eigentlich nur ein Beispiel-Script), wie auch die Binär-Version, die sich im JAR-File antenna-bin.jar befindet. Sourcen und JAR-File laden wir herunter und speichern es lokal, z.B. unter d:/java/antenna. Nach dem Starten von Eclipse (eclipse.exe) müssen wir noch den Classpath von Ant erweitern, damit wir die neuen Antenna Tasks verwenden können. Dafür gehen wir in Eclipse zum Menüpunkt Window/Preferences/Ant/Runtime und tragen das antenna-bin.jar in den Ant-Runtime-Classpath ein. Weiterhin fügen wir dort noch die Datei tools.jar aus dem lib-Verzeichnis unseres J2SE SDKs hinzu. Abbildung 1. zeigt diese Konfiguration.

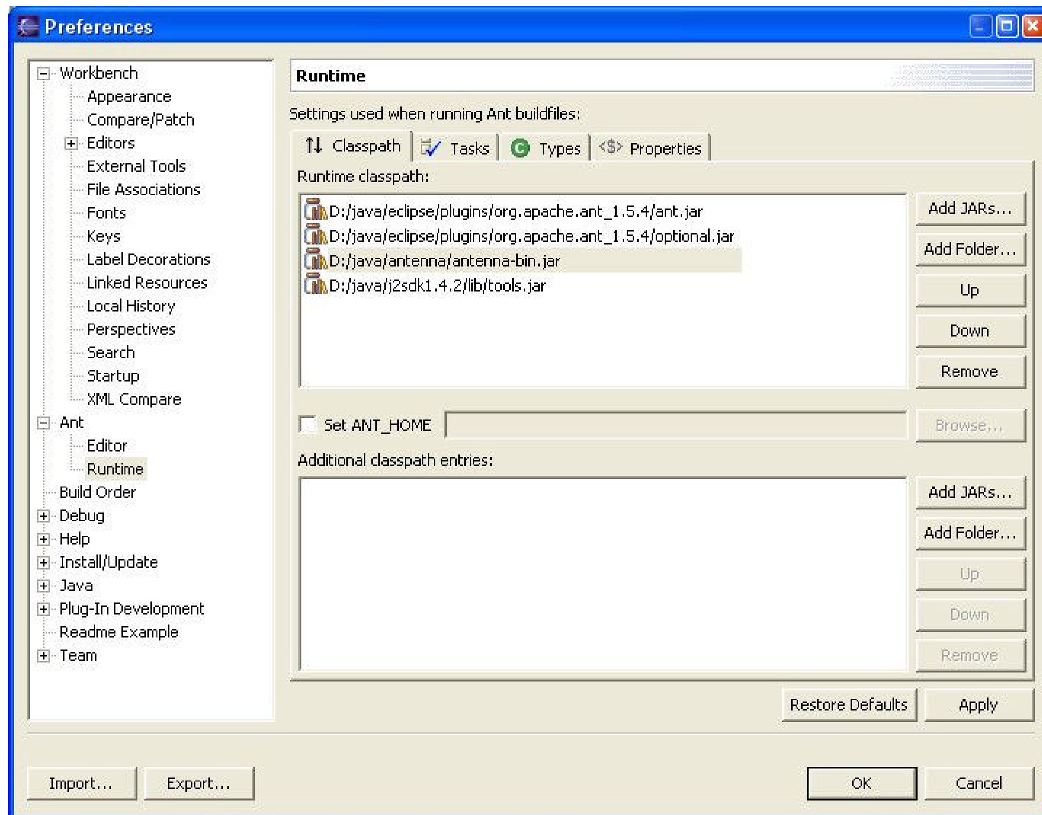

Abbildung 1: Eclipse Ant Runtime Classpath

Zu guter Letzt brauchen wir noch mindestens eine MIDP-Testumgebung. Dies kann entweder eine Handy-spezifische Umgebung sein, wie sie von Siemens [SiemensSMTK] oder Nokia [NokiaDS] kostenlos zur Verfügung gestellt werden, oder aber das Wireless Toolkit (WTK) von Sun Microsystems, welches sich sehr gut als generische Testumgebung eignet. Um die Siemens SMTKs für verschiedene Siemens-Handys zu bekommen, müssen Sie sich nur kostenlos beim Siemens-Portal registrieren. Das Tutorial wird im weiteren Verlauf sowohl Siemens SMTKs wie

auch Suns WTK benutzen. Installieren Sie bitte also noch die MIDP SDKs Ihrer Wahl, für dieses Tutorial brauchen Sie mindestens Suns WTK.

Das erste MIDlet

So, nun ist es soweit, unser erstes kleines MIDP-Projekt zu starten. Da dieses Tutorial nur einen Einstieg in die J2ME-Entwicklung geben soll, starten wir mit einem extrem simplen MIDlet, das nur ein paar geräteinterne System-Properties ausgeben soll. Hier der Source-Code:

```
package com.toedter.j2me.test;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.midlet.MIDlet;

/**
 * Das PropetyMIDlet zeigt einige System-Properties an:
 * microedition.platform,microedition.configuration,
 * microedition.profiles, microedition.encoding,
 *
 * @author Kai Toedter
 */
public class PropertyMIDlet extends MIDlet implements CommandListener {

	private Command exitCommand;
	private Form form;
	private Display display;

	public PropertyMIDlet() {
		display= Display.getDisplay(this);
		form = new Form("Property MIDlet");
		exitCommand = new Command("Exit", Command.EXIT, 2);
		form.addCommand(exitCommand);
		form.setCommandListener(this);

		form.append("Platform: "
				+ System.getProperty("microedition.platform") + "\n");
		form.append("Configuration: "
				+ System.getProperty("microedition.configuration") + "\n");
		form.append("Profiles: "
				+ System.getProperty("microedition.profiles")+ "\n");
		form.append(
				"Encoding: " + System.getProperty("microedition.encoding")
				+ "\n");
	}

	protected void startApp() {
		display.setCurrent(form);
	}

	protected void pauseApp() {
	}

	protected void destroyApp(boolean arg0) {
	}

	public void commandAction(Command command, Displayable displayable) {
		if (command == exitCommand) {
				destroyApp(false);
```

```
                    notifyDestroyed();
            }
        }
}
```

Listing 1: PropertyMIDlet

Hier noch ein paar kleine Anmerkungen zum Source-Code. Wichtig auch bei einfachen MIDlets ist das Verständnis des MIDlet-Lebenszyklus. Das MIDlet befindet sich immer in einem der drei Stati: Aktiv, Angehalten oder Gelöscht. Abbildung 2 zeigt ein Bild vom MIDlet-Lebenszyklus:

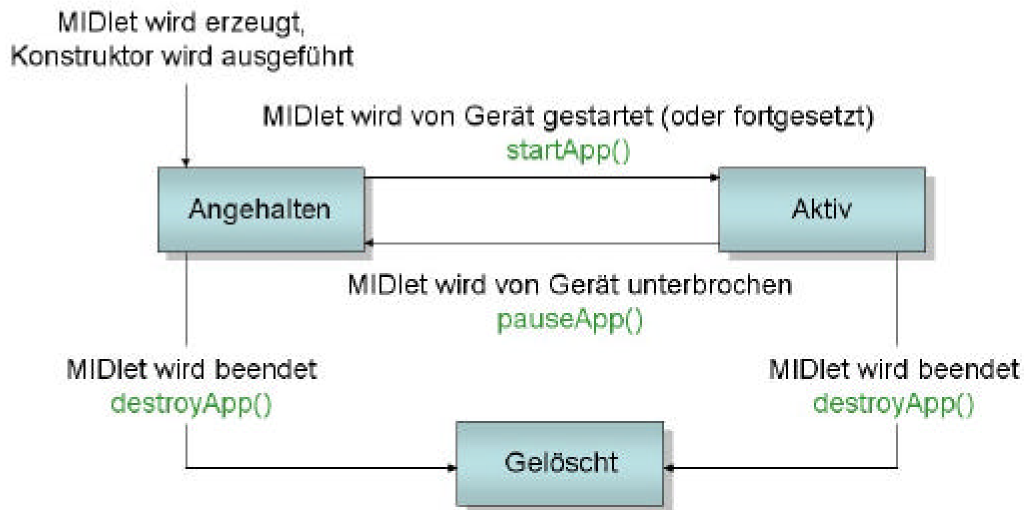

Abbildung 2: Der MIDlet-Lebenszyklus

Wichtig beim Lebenszyklus ist, dass die Methode startApp() mehrfach aufgerufen werden kann. Läuft das MIDlet zum Beispiel auf einem Handy, wird es normalerweise bei einem ankommenden Anruf durch Aufruf von `pauseApp()` angehalten und nach Beendigung des Anrufes durch wiederholtes Aufrufen von `startApp()` fortgesetzt. Dinge, die nur einmal initialisiert werden sollen, gehören also nicht in die Methode `startApp()`. Die Methode `destroyApp()` wird vom System aufgerufen, wenn das MIDlet beendet werden soll. Hier sollten alle Aufräumarbeiten passieren. Der `boolean`-Parameter `unconditional` gibt an, ob das MIDlet eventuell Einspruch erheben kann. Ist der Wert `true`, wird das MIDlet auf jeden Fall beendet, beim Wert `false` kann das MIDlet noch eine `MIDletStateChangeException` werfen, um dem Gerät anzugeben, dass es nicht beendet werden möchte. Wenn das MIDlet ordnungsgemäß durch den Endbenutzer beendet worden ist, sollte dies dem System durch den Aufruf der Methode `notifyDestroyed()` mitgeteilt werden. In unserem kleinen Beispiel-MIDlet initialisieren wir deswegen alle Objekte im Kontruktor und schalten bei `startApp()` nur auf unser Display um, welches eine einfache Form enthält, die nur die Textausgaben einiger System-Properties beinhaltet. Um das MIDlet durch eine Benutzereingabe beenden zu können, implementieren wir noch das `CommandListener`-Interface, um in der Methode `commandAction`() das MIDlet ordnungsgemäß zu beenden.

Ein Eclipse J2ME-Projekt

Wir starten Eclipse und legen unter File/New/Projekt ein Java-Projekt an, das wir „Property MIDlet" nennen. Innerhalb des „Java Build Path" legen noch das Verzeichnis „src" für unseren Source-Code an und geben als Ausgabe-Verzeichnis für die Java-Klassen das Verzeichnis „classes" an. Da wir ja nicht für J2SE entwickeln, löschen wir noch die standardmäßig eingestellte Java–

Laufzeitumgebung aus dem „Java Build Path“ und fügen stattdessen die Datei midpapi.zip aus Suns Wireless Toolkit (liegt dort im lib-Verzeichnis) hinzu. An dieser Stelle müssen wir entscheiden, ob das Projekt aus Sicht von Eclipse MIDP 1.x oder 2.0 unterstützt, da Eclipse ständig die Java-Syntax prüft. Da MIDP 2.0 eine Obermenge von MIDP 1.0 darstellt, können wir für unser Beispiel ruhig das API aus Suns MIDP 2.0 Referenz-Implementierung wählen. Im Package-Explorer von Eclipse klicken Sie jetzt auf `src` und erzeugen ein neues Package `com.toedter.j2me.test`. Natürlich können Sie auch einen anderen Packetnamen wählen, müssen dann aber auch die entsprechenden kleinen Anpassungen am Source Code und den Build-Files vornehmen. Kopieren Sie den Beispiel-Source-Code des PropertyMIDlets in das angelegte Package. In diesem Stadium ist Eclipse schon in der Lage, das Source-File automatisch zu kompilieren und erzeugt im classes-Verzeichnis die entsprechende class-Datei. Wir können aber das MIDlet noch nicht ausführen und testen, da uns noch einige Dinge fehlen: Zum einen der Java Application Descriptor, zum anderen müssen wir das MIDlet noch verifizieren und packen. Hier kommt die Verbindung Ant/Antenna ins Spiel. Kopieren Sie sich aus Antennas samples/siemens-Verzeichnis (war in der Source-Distribution) die Datei build.xml in Ihr Eclipse-Projektverzeichnis und passen Sie sie wie in Listing 2 gezeigt, an.

```
<?xml version="1.0"?>

<project name="SysProps" default="build" basedir=".">

    <!-- Define the Mobility Toolkit home directory, e.g the WTK home directory -->

    <property name="wtk.home" value="d:\java\wtk20"/>

    <!-- Define some additional properties for this project. Not required. -->

    <property name="midlet.name" value="sysprops"/>
    <property name="midlet.home" value="${basedir}/src"/>

    <!-- Define the tasks. –>

    <taskdef name="wtkjad" classname="de.pleumann.antenna.WtkJad"/>
    <taskdef name="wtkbuild" classname="de.pleumann.antenna.WtkBuild"/>
    <taskdef name="wtkpackage" classname="de.pleumann.antenna.WtkPackage"/>
    <taskdef name="wtkmakeprc" classname="de.pleumann.antenna.WtkMakePrc"/>
    <taskdef name="wtkrun" classname="de.pleumann.antenna.WtkRun"/>
    <taskdef name="wtkpreverify" classname="de.pleumann.antenna.WtkPreverify"/>
    <taskdef name="wtkobfuscate" classname="de.pleumann.antenna.WtkObfuscate"/>

    <target name="clean">
        <delete failonerror="false" dir="classes"/>
        <delete failonerror="false" file="${midlet.name}.jar"/>
    </target>

    <target name="build" depends="clean">

      <!-- Create a JAD file. -->
      <wtkjad jadfile="${midlet.name}.jad"
            jarfile="${midlet.name}.jar"
            name="SysProps"
            vendor="<IHR NAME ☺>"
            version="1.0.0"
            update="true">

         <midlet name="SysProps" class="com.toedter.j2me.test.PropertyMIDlet"/>
```

```
        </wtkjad>

        <!-- Make sure we have a fresh classes directory. -->
        <mkdir dir="classes"/>

        <!-- Compile everything, but don't preverify (yet). -->
        <wtkbuild srcdir="${midlet.home}" destdir="classes" preverify="false"/>

        <!-- Package everything.  -->
        <wtkpackage jarfile="${midlet.name}.jar" jadfile="${midlet.name}.jad">
            <fileset dir="classes"/>
        </wtkpackage>

        <!-- Preverify –>
        <wtkpreverify jarfile="${midlet.name}.jar" jadfile="${midlet.name}.jad"/>

        <!-- Start the MIDlet suite  -->
        <wtkrun jadfile="${midlet.name}.jad" />

    </target>

</project>
```

Listing 2: Ant/Antenna XML-Build-File

Zum Anstoßen des Build-Prozesses klicken Sie jetzt mit der rechten Maustaste auf die Datei build.xml in Ihrem Projekt und wählen die Funktion „Run Ant…“ aus. Jetzt erscheint ein Dialog, in dem Sie bestimmte Ant-Targets auswählen können. Lassen Sie es bei der Default-Einstellung „Build“ und klicken Sie auf „Run“. Nun wird der Build-Prozess angestoßen und nach erfolgreicher Erzeugung der JAR- und JAD-Dateien das MIDlet im WTK-Emulator gestartet, siehe Abbildung 3.

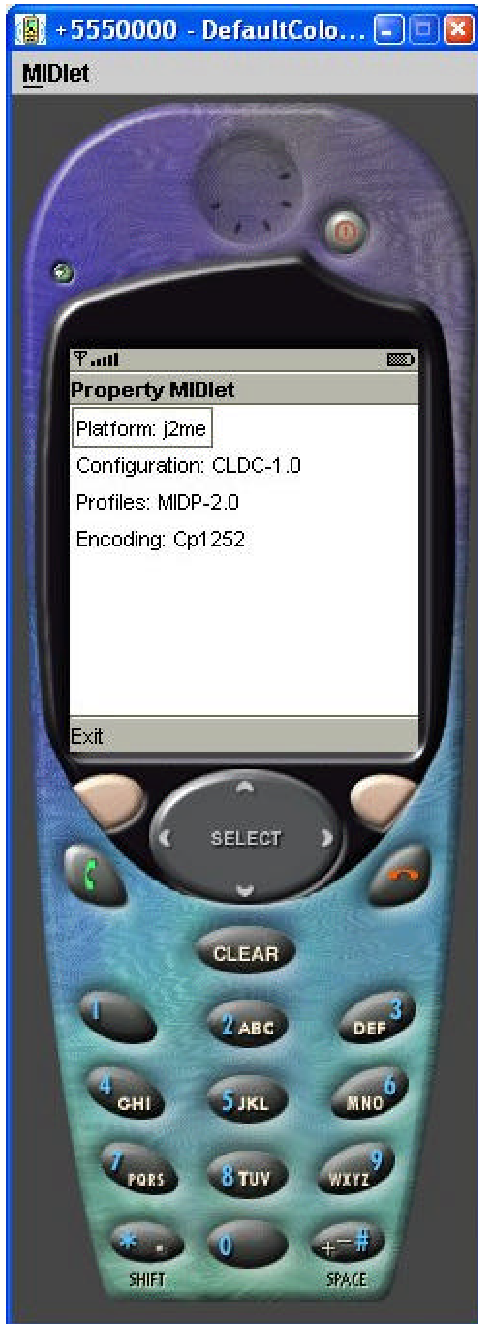

Abbildung 3: Unser Proprty MIDlet in Suns Wireless Toolkit 2.0 Emulator

Doch nun wieder zurück zu unserem Ant/Antenna Build-File. Zuerst spezifizieren wir einen Namen, das Default-Target und ein Basis-Verzeichnis. Steht dort ein „.“, wird automatisch das Eclipse-Projektverzeichnis genommen. Als nächstes müssen wir definieren, wo sich das Home-Verzeichnis unserer MIDP-Testumgebung befindet. In unserem Fall tragen Sie bitte den Verzeichnispfad vom Wireless Toolkit ein. Da wir später das gleiche Build-File auch für das Siemens SMTK nutzen wollen, empfiehlt es sich, diese Property außerhalb des Files zu setzten. Dies kann man in Eclipse sehr einfach bewerkstelligen: Nach dem ersten Lauf des Build-Files finden Sie unter „External Tools…“ (Das Läufer-Icon mit dem roten Koffer) unter den Ant-Build-Files den Eintrag „Proprty MIDlet build.xml“. Editieren Sie diesen und fügen unter dem Reiter „Properties“

einfach den Eintrag für „wtk.home“ hinzu. Später können Sie sich den Eintrag einfach kopieren (mit dublicate) und entsprechende Einträge für alle MIDP-Testumgebungen z.B. von Siemens oder Nokia einrichten. Doch wieder zurück zum build.xml. Mit den beiden Properties „midlet.name“ und „midlet.home“ bestimmen sie die MIDlet-spezifischen Parameter für den weiteren Build. Nun müssen Ant noch die Antenna-spezifischen Tasks bekannt geben werden. Auf diese werde ich im Detail noch später zu sprechen kommen. Als erstes Target im Build-File haben wir „clean“. Hier können Sie später in weiteren Projekten alle Aufräumarbeiten wie das Löschen dynamisch erzeugter Dateien unterbringen. Das für uns interessante Target ist build, welches vom clean-Target abhängt und dies deshalb automatisch aufruft. Wie schon oben erwähnt, brauchen wir ja unbedingt ein JAD-File. Dieses können wir entweder von Hand erzeugen, oder aber durch die Antenna-Task wtkjad automatisch erzeugen lassen. Hier tragen Sie einfach die wesentlichen MIDlet-Daten wie JAD-File-Name, JAR-File-Name, Name der MIDlet Suite, Vendor, Version, MIDlet-Name und -Klasse. Sie sollten auf jeden Fall den Parameter update=“true“ spezifizieren, da dann Ihr JAD-File nicht jedes Mal gelöscht und neu erzeugt wird. Dies ist wichtig, wenn Sie per Hand noch eigene Attribute zum JAD-File hinzufügen wollen. Es besteht auch die Möglichkeit, mit dem Parameter auto-version die Versionsnummer automatisch hochzuzählen. Als nächsten Schritt erzeugen wir ein neues classes-Verzeichnis, das ja durch den automatischen Aufruf vom clean-Target gelöscht wurde. Nun möchten wir noch mal alle Java-Klassen kompilieren und zwar genau mit dem in der Testumgebung durch wtk.home festgelegten Environment. Dafür nehmen wir die Antenna-Taks wtkbuild. Sie bekommt als Parameter nur unser Source-Verzeichnis, das Ausgabe-Verzeichnis „classes“ sowie die Anweisung, noch kein „preverify“ auf die erzeugten Klassen anzuwenden. Nachdem wir die Klassen erzeugt haben, verwenden wir die Task wtkpackage, um das entsprechende JAR-File zu erzeugen. Die Größe des JAR-Files wird dabei im JAD-File automatisch eingetragen. Als vorletzten Schritt müssen wir noch alle Klassen des JAR-Files durch den Preverifier jagen, was die Task wtkpreferify erledigt. Wir hätten uns diesen Schritt auch sparen können, wenn wir bei der Task wtkpackage den Parameter preverify auf `true` gesetzt hätten. Ich habe es im Beispiel als Extra-Schritt gemacht, um zu veranschaulichen, dass das Pre-Verifizieren logisch gesehen der letzte Schritt des Build-Prozesses ist. Nach erfolgreichem Build wollen wir unser MIDlet gleich starten, und benutzen dafür die Task wktrun mit dem JAD-File als Parameter. Defaultmäßig wartet wtkrun auf das Beenden des Emulators, was den Vorteil hat, dass noch abschließende Ausgaben angezeigt werden. Natürlich bieten die vorgestellten Antenna-Tasks noch eine Vielzahl von nützlichen Konfigurations-Möglichkeiten, welche alle unter [Antenna] sehr gut dokumentiert sind. Weiterhin gibt es noch zusätzliche Tasks, die das Deployen der MIDlet Suite auf einen OTA-Server (OTA = „Over The Air“) ermöglichen. Der Anbindung eines Obfuscators und dem Antenna Preprozessor möchte ich zwei eigene Abschnitte widmen.

Obfuscating

Ein Obfuscator, zu deutsch etwa Verwirrer oder Verunstalter, ist ein Tool, was den Java-Bytecode so modifiziert, dass die Semantik der Java-Anwendung exakt erhalten bleibt, der decompilierte Java Source-Code aber möglichst unleserlich wird. Sämtliche Bezeichner bekommen meistens neue Namen wie “a“,“b“,“c“ etc. Neben der Unleserlichmachung haben Obfuscatoren aber einen anderen sehr positiven Nebeneffekt, der für die MIDlet-Entwicklung von entscheidender Bedeutung ist: Sie schrumpfen die Bytecode-Größe teilweise erheblich. Die bekanntesten frei verfügbaren Obfuscatoren sind ProGuard [ProGuard] und RetroGuard [RetroGuard], die beide von Antenna unterstützt werden. Um z.B. ProGuard in Verbindung mit Antenna verwenden zu können, muss das entsprechende JAR-File nur im Classpath sein oder ins bin-Verzeichnis des MIDP-Emulators kopiert werden. Dann kann entweder die separate Antenna-Task wtkobfuscate herangezogen werden, oder aber das Attribut `obfuscate` in der Task wtkpackage auf `true` gesetzt werden. Verwenden wir ProGuard in unserer wtkpackage-Task, schrumpft die Größe unseres JAR-Files von 2109 Bytes auf 1911 Bytes, was knapp 10% entspricht. Bei größeren Projekten kann dieser Prozentanteil auch wesentlich größer werden, 30 – 40% sind keine Seltenheit.

Antenna-Preprosessor

Der Antenna-Preprozessor modifiziert den Java-Source-Code entsprechend selbstdefinierter Bezeichner, bevor er kompiliert wird. Viele von Ihnen werden jetzt denken: „So was kommt mir nicht ins Haus, das hat mir schon zu C- und C++-Zeiten viel Ärger bereitet!“. Dies war auch mein erster Gedanke, aber nach kurzem Nachdenken sind die Vorteile für die J2ME-Entwicklung recht offensichtlich. In der realen Welt ist es nämlich nicht etwa so, dass ein MIDlet auf allen Geräten gleich (gut) läuft. Oft sind Unterschiede in der Display-Auflösung, der Farbtiefe und vor allen den unterstützten optionalen oder proprietären Java-APIs ausschlaggebend für die erfolgreiche Ausführung eines MIDlets. Hier ein Beispiel aus der Praxis: Stellen Sie sich vor, Sie möchten ein Spiel entwickeln, welches die MIDP 2.0 Game-API benutzt, und somit auf allen MIDP 2.0 Kompatiblen Handys laufen sollte. Weiterhin soll das Spiel auch auf den aktuellen MIDP 1.0 kompatiblen Handys von Siemens laufen können. Da Siemens die MIDP 2.0 Game-API entwickelt hat, ist diese nämlich auf den aktuellen Siemens-Handys implementiert. Allerdings mit dem kleinen, aber entscheidenden Unterschied, dass alle Game-spezifischen Klassen in Siemens-Packages liegen. Hier lässt sich der Antenna-Preprozessor hervorragend einsetzten, wie z.B. in Listing 3 und 4 zu sehen ist.

In build.xml:

```
<wtkpreprocess srcdir="src" destdir="siemens" symbols="siemens" verbose="true"/>
```

Im Java Source-Code:

```
//#ifdef midp20
import javax.microedition.lcdui.game.Sprite;
//#endif

//#ifdef siemens
//# import com.siemens.mp.game.Sprite
//#endif
```

Der Java Source-Code ist für MIDP 2.0 kompilierbar, alle Preprozessor-Direktiven sind Java-Kommentare. Soll nun für Siemens-Handys kompiliert werden, interpretiert der Antenna-Preprozessor alle definierten Symbole, hier also „siemens“ und kommentiert beziehungsweise unkommentiert die entsprechenden Code-Stellen. Das Resultat im Siemens-Source-Verzeichnis wäre also:

```
//#ifdef midp20
//# import javax.microedition.lcdui.game.Sprite;
//#endif

//#ifdef siemens
import com.siemens.mp.game.Sprite
//#endif
```

Das spart auf jeden Fall eine Menge Arbeit, da man jetzt die gleiche Source-Basis benutzen kann.

Eclipse-Launcher für verschiedene Emulatoren

Wollen Sie nur das gleiche MIDlet in verschiedenen Emulatoren testen, können Sie sich einfach unter „External Tools“ Launcher für die verschiedenen Emulatoren einrichten. Klicken Sie dafür einfach auf den Pfeil rechts neben dem Läufer-Icon mit dem roten Koffer und wählen Sie den Eintrag „External Tools…“. Dann bekommen Sie ein neues Dialog-Fenster, in dem Sie neue Launcher-Konfigurationen erstellen können. Klicken Sie auf das Icon „Programs“ im linken Baum-Fenster und danach auf „New“ und geben Sie Ihrer neuen Konfiguration einen sinnvollen Namen.

Im Reiter Main browsen Sie im Feld Main nach der Datei emulator.exe Ihrer MIDP - Testumgebung. Im Feld „Working Directory“ klicken Sie auf „Browse Workspace“ und wählen das Property MIDlet. Nun müssen Sie noch die Emulator-spezifischen Parameter eingeben. Für den Siemens-Emulator für das MC60 sind dies:
`-Xdevice:MC60 -Xdescriptor:"sysprops.jad"`. Abbildung 4 zeigt diese Launcher-Konfiguration. Nachdem Sie diese einmal gestartet haben, finden Sie einen Eintag unter dem Läufer-Icon, den Sie mit einem Klick wieder starten können. In Abbildung 5 sehen Sie die komplette Eclipse-Entwicklungsumgebung zusammen mit dem aus Eclipse heraus gestarteten Siemens MC60-Emulator.

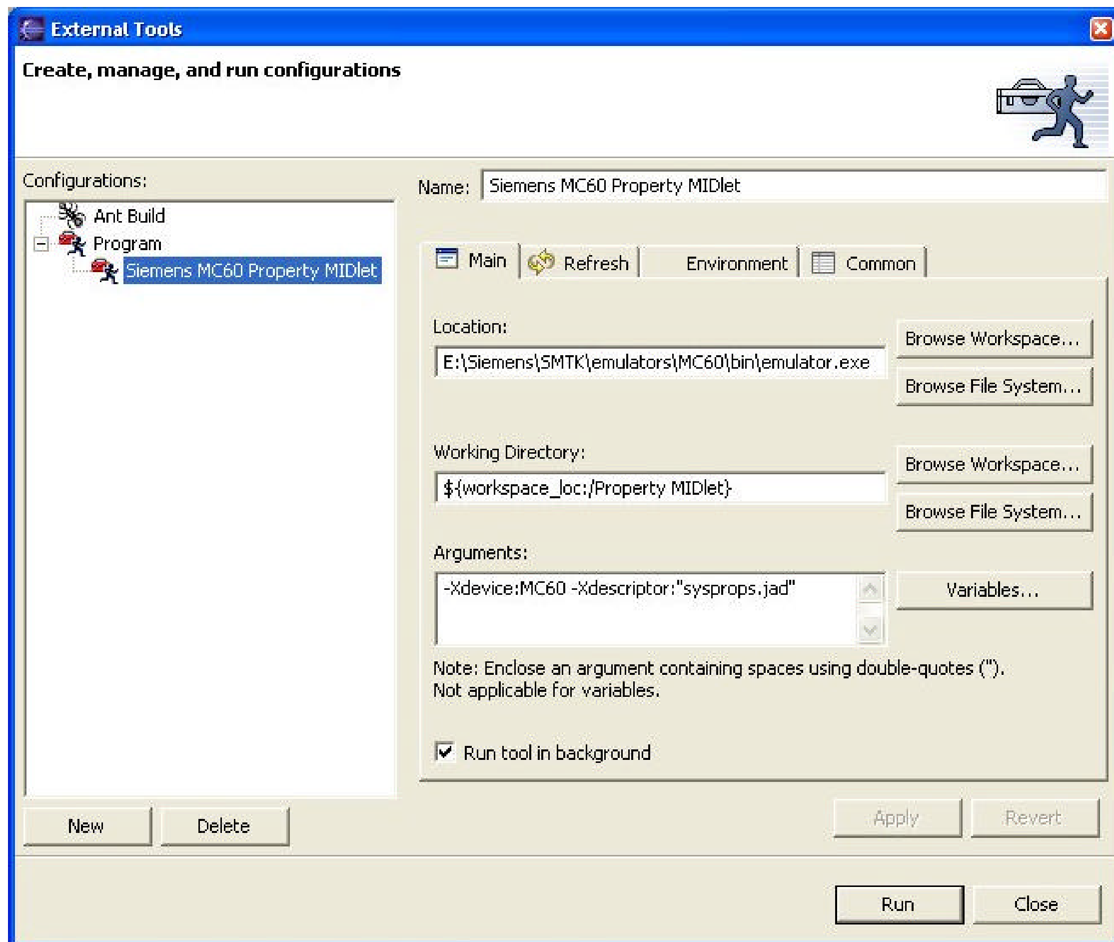

Abbildung 4: Eine Eclipse Launcher-Konfiguration für den Siemens MC60-Emulator

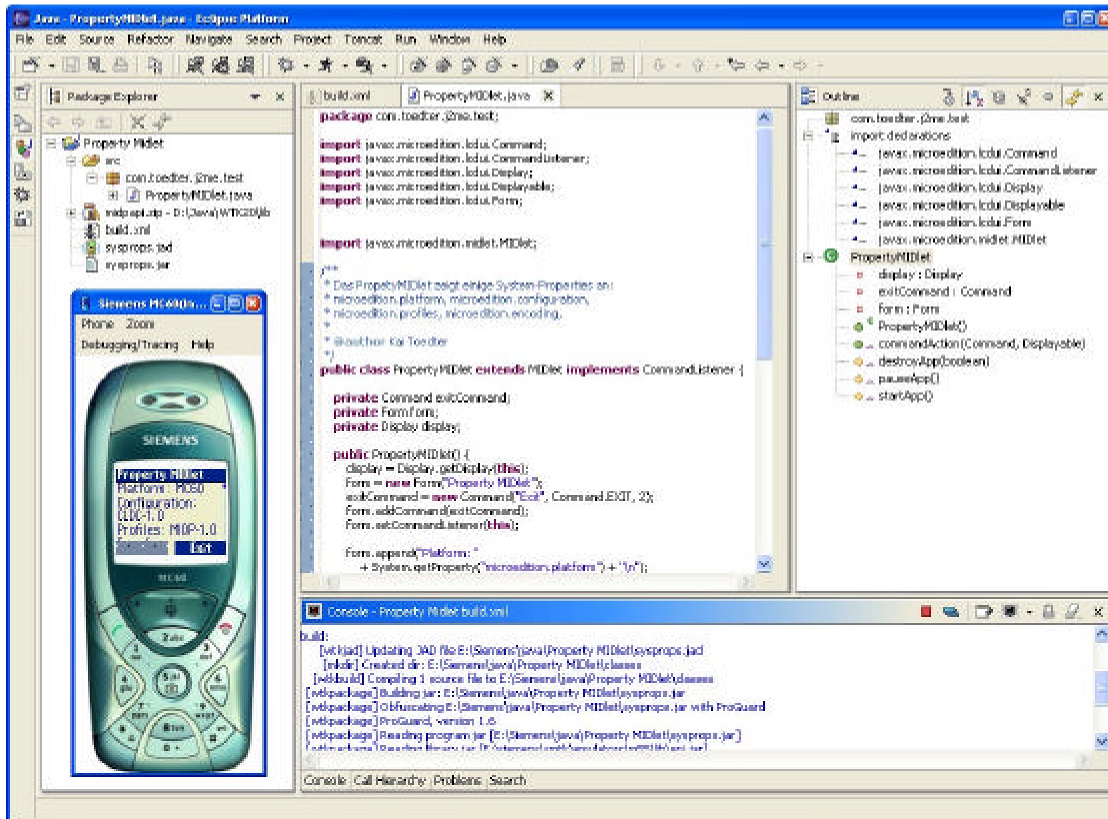

Abbildung 5: Die Eclipse -Entwicklungsumgebung mit Siemens MC60-Emulator

Debugging

Ein wichtiges Thema bei der Entwicklung für MIDP ist natürlich das Debugging. Während Umgebungen wie JBuilder und SunONE Studio komfortables Debuggen innerhalb der Emulatoren auf Source-Code-Ebene unterstützen, ist dies leider mit der hier vorgestellten Eclipse/Antenna-Konfiguration nicht möglich. Die Eclipse-APIs bieten durchaus die Möglichkeit, Source-Ebenen-Debugger anzubinden, wie z.B. WSDD zeigt. WSDD implementiert dazu eine eigene J2ME-"Launch Configuration", die mittels dem offiziellen Eclipse API implementiert wurde. Ich bin deshalb sehr zuversichtlich, dass es bald frei verfügbare Plug-ins geben wird, die auch genau dieses Feature implementieren und somit die Lücke schließen. Will man für MIDP 1.0 entwickeln, gibt es noch eine weiter Möglichkeit, um in den Genuss von Source-Ebene-Debuggen zu kommen: Man benutzt das frei verfügbare Werkzeug ME4SE [ME4SE]. Das Verfahren ist denkbar einfach. Nach der Installation von ME4SE erzeugen Sie ein kleines Eclipse Java-Projekt. Es reicht quasi ein leeres Projekt, das nur die Bibliothek me4se.jar beinhaltet. Dann erzeugen Sie einen Eclipse-Launcher, welcher die Klasse `org.me4se.MIDletRunner` aus dem ME4SE-Projekt ausführt und spezifizieren als Parameter (unter „Arguments“) `–jad <unser sysprops.jad>`. Weiterhin wählen Sie als „Working directory“ unser Propty MIDlet. Im Classpath des Runners fügen Sie noch unser Projekt „Property MIDlet“ hinzu. Fertig! Wenn Sie den Runner nun starten, erhalten Sie Abbildung 6.

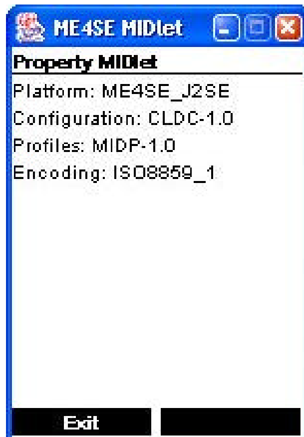

Abbildung 6: Unser Property MIDlet im ME4SE-Runner

Der Nachteil dieses Verfahrens ist natürlich, dass Sie aus dem Verhalten von ME4SE nicht unbedingt auf das Verhalten eines echten Endgerätes schließen können. Das Verfahren hat allerdings auch Vorteile. Da Sie die J2SE-Umgebung nicht verlassen, brauchen Sie kein Preverify auf die erzeugten Java-Klassen anzuwenden. Wenn Sie z.B. innerhalb von Eclipse eine kleine Änderung an einem Source-File vornehmen, wird diese für ME4SE sofort sichtbar, ohne dass Sie den Ant-Build-Prozess neu anstoßen müssen. Das gibt natürlich hervorragende Turnaround-Zeiten. Der größte Vorteil ist natürlich, dass Sie jetzt Eclipses hervorragende J2SE-Debugger-Unterstützung nutzen können. Befindet sich zum Beispiel irgendwo in Ihrem MIDP-Code eine Nullpointer-Exception, finden Sie diese auch sehr schnell durch Debuggen mit ME4SE. Es bleibt zu hoffen, dass ME4SE auch auf MIDP 2.0 erweitert wird, für MIDP 1.0-Entwicklungen leistet es auf jeden Fall gute Dienste.

Fazit

Es gibt einige sehr gute Tools, sowohl kommerzielle, wie auch frei verfügbare, die MIDP-Entwicklung unterstützen. Wenn Sie allerdings nicht auf Eclipse verzichten wollen, können Sie sich mit Hilfe von Antenna schnell eine professionelle MIDP-Entwicklungsumgebung konfigurieren. Größter Wermutstropfen dürfte die fehlende Source-Level Debugging-Unterstützung der vorgestellten Eclipse/Antenna-Konfiguration innerhalb von MIDP-Emulatoren darstellen. Ich bin aber guter Hoffnung, dass diese Lücke bald durch frei verfügbare Plug-ins geschlossen wird.

Literatur und Links

- [CLDC] Java Community Process, JSR-000030 J2ME Connected, Limited Device Configuration (Final Release), siehe: http://www.jcp.org/aboutJava/communityprocess/final/jsr030/
- [CLDC1.1] Java Community Process, JSR-000139 Connected Limited Device Configuration 1.1 (Final Release), siehe: http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html
- [Brontofundus] Der Brontofundus, J2ME – MIDP, siehe: http://bf.monis.ch/prog/java/midp/index.html
- [SunONE] Sun Microsystems, Sun ONE Studio 4 update 1, Mobile Edition, siehe: http://wwws.sun.com/software/sundev/jde/studio_me/index.html
- [JBuilder] Borland, JBuilder und Mobileset, siehe: http://www.borland.com/products/downloads/download_jbuilder.htm
- [WSDD] IBM, WebSphere Studio Device Developer, siehe: http://www-3.ibm.com/software/wireless/wsdd/
- [IntelliJ] Jetbrain, IntelliJ IDEA, siehe: http://www.intellij.com/idea/
- [Eclipse] Eclipse, siehe: http://www.eclipse.org/

- [Siptech] Siptech J2ME Plugin for Eclipse, siehe: http://www.siptech.com/index?url=j2meplugin
- [Eclipseme] Eclipseme Open-Source Projekt, siehe: http://sourceforge.net/projects/eclipseme
- [Ant] Apache Ant Projekt, siehe: http://ant.apache.org/index2.htm
- [Antenna] Antenna, An Ant-to-End Solution For Wireless Java, siehe: http://antenna.sourceforge.net
- [J2SE] Sun Microsystems, Java 2 Standard Edition, siehe: http://java.sun.com/j2se/1.4.2/download.html
- [SiemensSMTK] Siemens Developer Tools, siehe: http://www-my-siemens.com/developer
- [NokiaDS] Nokia Developer's Suite for J2ME, siehe: http://americas.forum.nokia.com/sdk/default.asp
- [ProGuard] ProGuard Obfuscator Open-Source Projekt, siehe: http://proguard.sourceforge.net
- [RetroGuard] Retrologig Systems, RetroGuard Obfuscator, siehe: http://www.retrologic.com/
- [Me4SE] ME4SE, MIDP für J2SE, siehe: http://www.me4se.org

Autor

Kai Tödter ist Principal Engineer in der Corporate Technology Abteilung der Siemens AG. Er beschäftigt sich seit vielen Jahren mit Java-Technologie, sein aktuelles Projekt bewegt sich im J2ME-Umfeld. Kai war bis letztes Jahr als „Technical Liaison Manager for Java“ für Siemens in den USA tätig und hat Siemens im Executive Commitee des Java Community Process vertreten.
E-Mail: kai.toedter@siemens.com