Midlet Entwicklung

mit Hilfe von

J2ME Polish

Eine Seminararbeit von

Thomas Kraft
MatrNr. 011270

Betreut durch Prof. Fuchß

FH Karlsruhe
Hochschule für Technik
Moltkestraße 30
76133 Karlsruhe

J2ME developers face a serious problem: either they use only the lowest common denominator of J2ME devices – the MIDP/1.0 standard – or they choose to support only a few handsets, for which they optimize their applications with a lot of energy. The open source tool J2ME Polish promises a solution to this problem – and more.

Robert Virkus, Enough Software

Inhaltsverzeichnis

1. Einleitung

Immer wieder steht man bei der Entwicklung von mobilen Anwendungen vor dem Problem, das es mittlerweile eine fast unüberschaubare Anzahl von verschiedenen Geräten mit sehr unterschiedlichen Hardware Spezifikationen gibt. Die einzelnen Geräte unterscheiden sich dabei nicht nur durch solch offensichtliche Sachen wie Displaygrösse oder Speicherkapazität. In der J2ME Welt gibt es auch noch die diversen APIs, zum Beispiel für die Behandlung von Nachrichten wie SMS oder eMail, die bei weitem nicht alle Geräte unterstüzen. Auch haben mittlerweile viele Hersteller eigene APIs für Ihre Geräte entwickelt, welche das wilde Durcheinander innerhalb der Welt der mobilen Anwendungen eher noch unterstützen.

In diesem Seminar soll exemplarisch die Entwicklung von mobilen Anwendungen mit Hilfe des Frameworks „J2ME Polish“ erläutert werden, welches eine grosse Anzahl von Hilfsmitteln zur Verfügung stellt, die dem Entwickler der Anwendung einiges an Arbeit abnehmen. Hauptsächlich für die Probleme der vielen unterschiedlichen Hardware Spezifikationen und mannigfaltigen (auch proprietären) APIs hält J2ME Polish hier sehr hilfreiche Funktionen bereit. Diese Funktionen bringen einem gerade beim praktischen Entwickeln einer Applikation, welche später auf vielen Endgeräten laufen soll, sehr viel Zeitersparnis und gewährleisten, daß das Projekt mit vertretbarem Aufwand auf möglichst viele Zielplattformen portiert werden kann und somit eine grosse Anzahl potentieller Kunden bzw. Anwender erreicht.

J2ME Polish bietet einen reichhaltigen Fundus von nützliche Hilfsmitteln, wie Preprocessing, Ressourcen Bundling, Lokalisation, Logging, Obfuskation und das zusammenpacken zu einer lauffähigen Anwendung.

Darüber hinaus bietet es eine in XML geschriebene „Device Database“, in welcher die verschiedenen Eigenschaften einzelner Geräte angeben sind. Die aktuelle Version der Device Database umfasst zur Zeit 182 Geräte inklusive deren Spezifikationen wie Auflösung der integrierten Kamera, die Farbtiefe des verwendeten Displays oder aber die unterstützten APIs. Mit Hilfe dieser Datenbank ist es möglich, seine Applikationen perfekt an das entprechende Gerät anzupassen.

1.1 J2ME was ist das?

Sun hatte bei der Entwicklung von JAVA eine Vison. Sie wollten eine universelle Steuerungssprache für Geräte aller Art entwickeln. Diese Steuerungssprache sollte dem Entwickler die Möglichkeit bieten, mit Hilfe einer sicherheitstechnisch ausgereiften Programmiersprache Programme für nahzu alle Sorten von Hardware zu schreiben. Aus dieser Vision, JAVA auf allen möglichen Endgeräten zur Verfügung zu haben, enstand auch das wohl bekannte, von Sun immer wieder gern aufgegriffene Paradigma: „write once, run everywhere".

Im Lauf der Zeit stellte sich jedoch heraus das dieser Ansatz leider in der realen Welt der mannigfaltigen Hardwarekonfigurationen nicht voll umzusetzen war. Zu sehr differieren die Anforderungen in den jeweiligen Bereichen und vor allem differieren auch die zur Verfügung stehenden Plattformen auf denen JAVA lauffähig sein sollte.

Aufgrund dieser Unzulänglichkeiten wurde die Sprache JAVA recht schnell in drei verschiedene „Bereiche" unterteilt:

- Server Anwendungen / Businesslogik
- Desktop Anwendungen / Internet Anwendungen
- Kleingeräte / Mobile Geräte (Handhelds, Mobiltelefone usw.)

Für diese drei Bereiche wurde jeweils eine optimierte JAVA Version entwickelt:

- JAVA 2 Enterprise Edition (J2EE)
- JAVA 2 Standard Edition (J2SE)
- JAVA 2 Micro Edition (J2ME)

Somit bezeichnet „J2ME" die speziell für Geräte mit streng limitierten Hardware Ressourcen entwickelte JAVA Version. Diese Geräte zeichnen sich oft dadurch aus, das sie eine langsame CPU, wenig Speicher und nur ein eingeschränktes Display besitzen, was in Zusammenhang mit anderen Einschränkungen, zum Beispiel nicht vorhandene Tastatur oder Netzwerkverbindung dazu führt das es gerade im Bereich der J2ME mannigfaltige Laufzeitumgebungen gibt, die sehr unterschiedlich sein können. In diesem Seminar möchte ich darauf eingehen, wie man sich durch geschickten Einsatz des J2ME Polish Frameworks eine Menge Optimierungsaufwand für unterscheidlichste Hard-

wareprofile einsparen kann und als Entwickler in die Lage versetzt wird Code zu schreiben der beim compilieren ein aus Sicht der Zielplattform möglichst optimales Ergebnis bringt.

1.2 MIDP1

MIDP1 ist die Bezeichnung der ersten Version von konkreten Libraries für die J2ME. MIDP1 bezeichnet also eine Sammlung von Klassen und Interfaces, mit deren Hilfe man entsprechende JAVA Programme für diese Geräte entwickeln kann. Ähnlich den Klassen der „normalen" J2SE hat der Entwickler die aus dem JAVA Umfeld bereits bekannten Basisklassen und primitiven Datentypen zur Verfügung. Aufgrund der beschränkten Resourcen müssen jedoch bereits in den Basisklassen diverse Einschränkungen in Kauf genommen werden. Bekannte Klassen wie z.B. `Thread` besitzen in der J2ME Variante deutlich weniger Methoden, die funktionalen Möglichkeiten wurden hier auf das absolut Notwendige beschränkt. Hier wurde vor allem dem oft sehr kleinen Speicher und langsamen Prozessor Rechnung getragen. Auf manche Klassen wie zum Beispiel den Datentyp `float` muss der J2ME Entwickler sogar komplett verzichten.

MIDP1 bietet jedoch trotz alledem mannigfaltige Möglichkeiten der Anwendungsentwicklung, wenn auch im etwas ungewohnt kleinen Stil, und wird von sehr vielen mobilen Geräten heutzutage unterstützt. Elementare Ein- bzw. Ausgabeelemente wie Listen und Formulare sind vorhanden und werden von der KVM direkt unterstützt, so das sich der Entwickler hier auch über die Darstellung auf dem konkreten Display keine Gedanken machen muss. Vor allem Mobiltelefone wurden in der Vergangenheit mit MIDP1 ausgestattet. Aktuell unterstützen ungefähr 63% aller am Markt befindlichen JAVA – fähigen Mobiltelefontypen den MIDP 2 Standard. [1]

MIDP1 unterstützt den Entwickler zwar mit grundlegenden Sprachelementen wie Forms und einer Zeichenfläche (`Canvas`). Die Möglichkeiten der API gehen jedoch meist nicht über sehr rudimentäre Funktionen hinaus. Gerade beim Versuch grafisch ein wenig aufwändigere Applikationen (zum Beispiel Spiele) zu programmieren, stösst der Entwickler ziemlich schnell an die

Grenzen des Systems. Dies wurde auch von Sun schnell erkannt und man erweiterte die MIDP1 Spezifikation erheblich um neue Funktionen um dem Programmierer mehr Möglichkeiten zu bieten, sein Programm auch optisch nach seinen Wünschen zu gestalten. Der neue Standard im Bereich mobiles JAVA heisst seit dem „MIDP2“

Praktisch alle neu auf den Markt kommenden mobilen Geräte unterstützen mittlerweile den MIDP2 Standard, und da die durchschnittliche Lebensdauer eines solchen Gerätes auch durch die Vertragspolitik der Mobilfunkanbieter, welche dem Kunden meist automatisch nach 1 oder 2 Jahren ein neues Gerät anbieten, recht kurz ist, wird in kurzer Zeit dieser Standard vom Grossteil der Geräte unterstützt werden.[2]

1.3 MIDP2

Es ist zwar auch möglich mit dem MIDP1 Standard einige lustige kleine Spiele zu programmieren, allerdings ist die Version 2 deutlich mehr Funktionalität in dieser Richtung ausgestattet, so das es dem Entwickler möglich ist, wesentlich schönere und reichhaltigere Anwendungen zu schreiben.

Das wichtigste Paket das in MIDP2 neu hinzugekommen ist, stellt sicherlich `javax.microedition.lcdui.game` dar, welches ein spezielles Paket zur optimierten Darstellung von Grafiken und den optimierten Zugriff auf die Steuerung ist. Hier sind wohl speziell der `GameCanvas` und die Möglichkeit `Image` Objekte zu transformieren und natürlich die Klasse `Sprite` zu erwähnen. All diese, für eine professionelle Entwicklung von grafisch aufwändigen Anwendungen erforderlichen Funktionen stehen erst ab der zweiten Version des MIDP Standards zur Verfügung.

So ist es nun auch endlich möglich, eine Sound Unterstützung in seinen Anwendungen zu integrieren, was mit MIPD1 faktisch nicht möglich war.

In der Richtung Konnektivität hat sich auch einiges getan mit der Einführung des neuen Standards. Die Unterstützung von Verbindungen und Sockets die dem HTTP bzw dem HTTPS Standard entsprechen, sind gerade für Anwendungen, die wirklich auf die Mobilität des Gerätes abzielen, sehr wichtig. Somit ist es nun auch ohne Probleme möglich, einen Client für Webanwendungen zu bauen, der mit Hilfe von Standarisierten HTTP Sockets auf eine Webschnittstelle eines Servers entweder im Inter- oder natürlich auch dem Intranet zugreift. Damit ist ein wichtiger Schritt auch im Bereich der Businesstauglichkeit der Midlets getan worden, auch wenn, wie bei MIDP1, die Socket Connections bei MIDP2 immer noch optional sind. Faktisch werden sie von allen dem Autor bekannten MIDP2 Geräten unterstützt.

2. J2ME Polish

J2ME Polish stellt einen recht gelungenen Versuch dar, der Midlet Entwicklergemeinde ein umfangreiches Framework zur Entwicklung von J2ME Anwendungen in die Hand zu geben, welches viel der meist umfangreich nötigen Anpassungsarbeit für einen erledigt. Herausgekommen ist dabei nicht nur ein hochspezialisiertes Framework, welches einen grossen Beitrag zum effektiveren Programmieren für die unterschiedlichsten Endgeräte leistet, sondern auch eine umfangreiche Datensammlung über sowohl Hardware- als auch Softwarespezifikationen der unterschiedlichsten Modelle der einzelnen Hersteller.

Wenn man früher immer umständlich auf den Internetseiten der jeweiligen Handy Hersteller versucht hat, die benötigten Informationen zu sammeln, die man für die Entwicklung seiner Anwendungen benötigt hat, reicht jetzt ein Blick in die Device Datenbank von J2ME Polish und man hat meist alle relevanten Informationen auf einen Blick übersichtlich und in XML Notation dargestellt.

J2ME Polish unterstützt mit seinen Umfangreichen Tools den Entwicklungsprozess ganz erheblich, und bringt einen deutlich schnellere Time-To-Market als die meisten anderen J2ME Frameworks, zumindest aus der Erfahrung des Autors heraus.

Der Vollständigkeit halber sollen hier die Konkurenzprodukte nicht unerwähnt bleiben. Diese Frameworks sind meist, wie auch J2ME Polish, in zwei verschiedenen Lizenzmodellen zu haben. Einmal eine Community Version, die meist quelloffen und unter der GPL zu haben ist, als auch eine kommerzielle Version für den gewerblichen J2ME Entwickler.

- kAWT (GPL / kommerziell)
- Synclast UI API (GPL / kommerziell)
- JTGL (LGPL / kommerziell)
- antenna (LGPL)

3. Praktischer Einsatz von J2ME Polish

3.1 Preprocessing

Unter Preprocessing versteht man das gezielte Verändern von Quellcode, bevor er kompiliert wird. Dadurch lassen sich die wirklich dem Compiler vorgelegten Klassen durch ein vorgegebenes Regelset verändern um zu erreichen, das der schlußendlich compilierte Programmcode keine unnützen Teile mehr enthält, welche die daraus resultierende Applikation unnötig aufblähen und größer machen würde. Und da gerade die spätere Größe einer Anwendung nicht selten den Ausschlag gibt, ob auf einem bestimmten Endgerät die Applikation ausgerführt werden kann oder nicht, bietet J2ME Polish dem Entwickler durch Unterstützung von Preprocessing ein sehr mächtiges Werkzeug, das ihm hilft, optimale Programme für die jeweiligen Geräte zu erzeugen.

Der Preprozessor wird bei J2ME Polish durch sogenannte Direktiven getriggert, welcher entweder Symbole oder Variablen dazu benutzt, zu entscheiden ob er Code einbauen oder verändern soll. Symbole sind dabei für den Preprozessor wie boolsche Werte, entweder ein Symbol ist definiert und damit vorhanden, oder aber es ist eben nicht vorhanden. Variablen hingegen haben immer einen Wert, welche dann innerhalb der speziellen Preprozessor Direktiven dazu verwendet werden kann, zum Beispiel den Wert der Variable in den erzeugten Code einzubauen, oder aber den Wert der Variablen mittels eines Operators für Bedingungen zu nutzen.

So ist es Beispielsweise möglich, eine bestimmte Methode, je nach Unterstützung einer speziellen API durch das Zielgerät, entweder mit dieser konkreten Implementierung zu füllen oder bei Nichtvorhandensein der benötigten API eine externe Bibliothek einzubinden, die die benötigte Funktionalität zur Verfügung stellt, ohne diesen zusätzlichen Programmcode zu haben, wenn das Zielgerät die entsprechende API schon von Haus aus besitzt.

Symbole und Variablen befinden sich in verschiedenen Dateien:

- *build.xml*
  Symbole als auch Variablen werden hier mit dem Attribut **`"symbols"`** des **`<build>`** Elementes und mit dem **`<variables>`** Element angegeben
- *vendors.xml*, *groups.xml*, *devices.xml*: Symbole werden durch das **`<features>`** Element definiert. Variablen werden im **`<capability>`** Tag von den Geräte Eigenschaften definiert, welche auch meist noch Symbole definieren.

Gerade die mitgelieferte, sehr umfangreiche Device Datenbank, welche sehr viele Eigenschaften der einzelnen Geräte nicht nur erfasst, sondern auch als Symbole und Variablen automatisch zur Verfügung stellt, macht das Preprozessing in J2ME Polish erstens sehr effektiv, und zweitens sehr einfach zu benutzen.

Sämtliche Preprozessor Anweisungen beginnen immer mit folgenden Zeichen:

`//#`

Dadurch ist zuallererst einmal gewährleistet, das die für den Preprozessor bestimmten Programmzeilen einen „normalen“ JAVA Compiler, welcher versucht, den mit Preprozessoranweisungen erweiterten Code zu compilieren, nicht beeinflussen, da für diesen die Anweisungen als Kommentare erkannt und ignoriert werden.

Darüber hinaus hat sich auch diese Schreibweise mittlerweile im J2ME Entwickler Umfeld als Quasi – Standard etabliert, da auch das deutlich weiter verbreitete *Antenna* diese Syntax benutzt. Um die Kompatibilität zu *Antenna* zu wahren, werden alle *Antenna* Preprozessor Direktiven auch von J2ME Polish unterstützt, so das bei einer eventuellen Portierung eines bereits laufenden Projektes relativ wenige bis gar keine Anpassungen am Quellcode vorgenommen werden müssen. J2ME Polish unterstützt über die Antenna Spezifikationen hinaus noch zahlreiche weitere Preprozessor Direktiven und ist damit von der Flexibilität und dem Funktionsumfang dem bisherigen Marktführer in diesem Segment deutlich überlegen.[3]

3.1.1 Logging

Das integrierte Logging in J2ME Polish ist im Grunde genommen nur eine Untermenge des Preprocessing. Es steht auf Grund der geringen Kapazität der Hardware natürlich kein komplexes Logging Framework zur Verfügung, welches sich mit einem so prominenten Vertreter der Gattung wie Log4J messen könnte. Trotz alledem sind bei der Entwicklung von komplexen Anwendungen Log Ausgaben sehr sinnvoll, da sie bei Bugs sehr hilfreich beim Aufspüren von diesen versteckten Fehlern sind. Und gerade bei algorithmischen Modellen die einige Dutzend male durchlaufen werden, bis der zu suchende Fehler auftritt, ist ein normaler Debugger schlichtweg nicht geeignet.

Mangels Möglichkeiten, direkt in einer Datei zu loggen, bedient sich J2ME Polish hier eines einfachen, aber genialen Tricks: Es benutzt zum loggen die Standard Ausgabe, sprich: Die Konsole. Diese Standardausgabe ist auch bei mobilen Anwendungen verfügbar, wird normalerweise jedoch komplett vor dem Endanwender verborgen. Diese Art der Log Ausgabe auf die Standardausgabe ist somit vor allem für den Entwicklungszyklus mit dem Test auf den Emulatoren der jeweiligen Handys interessant.

Ein kleines Beispiel für einen einfache Log Ausgabe auf der Konsole:

```
try {
    callComplexMethod();
    //#debug info
    System.out.println("complex method succeeded.");
} catch (MyException e) {
    //#mdebug error
    System.out.println("complex method failed:");
    System.out.println(e.getMessage(), e);
    //#enddebug
}
```

Single line und multiple Line Debugging Beispiel 1

In diesem Beispiel sind mehrere Aspekte von Interesse, es werden die zwei grundlegenden Debug Preprozessor Direktiven **`//#debug`** und **`//#mdebug`** benutzt. Während die erste Anweisung dem Preprozessor mitteilt, das die direkt darauf folgende Zeile Code eine Debug Anweisung ist, wird mit der zweiten ein mehrzeiliger Debug Bereich begonnen, der explizit mit **`//#enddebug`** wieder beendet werden muss.

Dadurch ist es nun also problemlos möglich, seinen Code mit Anweisungen zu versehen, der nur in die entsprechende Applikation mit eincompiliert wird, wenn im dazugehörigen Build-File den entsprechende Debug Level (oder grösser) gesetzt ist. Im obigen Beispiel werden die zwei Debug Level „info“ und „error“ genutzt. Standardmässig verfügt J2ME Polish über folgende Debug Level:

`debug < info < warn < error < fatal < user-defined`[4]

Es ist hier natürlich möglich eigene Debug Level einzuführen um bei Bedarf eine noch feinere Granularität der Debug Ausgaben zu erreichen. Vorstellbar wäre hier zum Beispiel ein Level „benchmark“ um das Laufzeitverhalten des Programms zu überprüfen und zu optimieren.

Eine zweite Log Möglichkeit, welche J2ME Polish bietet, ist die Hilfsklasse **`Debug`**, die mit dem Framework mitgeliefert wird. Mit der Hilfe dieser Klasse ist es möglich, die Debug Ausgaben nicht nur auf die Kommandozeile zu loggen, sondern J2ME Polish bietet hier eine Möglichkeit an, diese Debug Ausgaben durch Benutzung der so genannten „Debug GUI“ auf echten mobilen Endgeräten sichtbar zu machen. Dazu werden von der Hilfsklasse **`Debug`** die diversen Debug Ausgaben in eine **`Form`** geschrieben, so das man sich die Debug Ausgaben auch auf den „echten“ Geräten anzeigen lassen kann. Sehr nützlich bei der Fehlersuche, wenn gewisse Probleme nur auf echten Geräten auftreten und nicht im Emulator reproduziert werden können.

Beispiel 2 zeigt exemplarisch eine solche Möglichkeit auf. Hier wird unter anderem durch eine Preprozessor Anweisung der entsprechende Menüpunkt zum Anzeigen der Debug **`Form`** nur dann in die Anwendung hineinkompiliert, wenn in der *build.xml* der „GUI Debugging Mode“ aktiviert ist. Dadurch ist dann automatisch das Preprozessor Symbol **`"polish.useDebugGui"`** gesetzt und kann für bedingte Abfragen verwendet werden. In dem vorliegenden Beispiel wir das **`Command`** nur dann angelegt, wenn eben dieses Preprozessor Symbol gesetzt ist, genauso wir nur bei Vorhandensein desselbigen auf das Eintreffen des **`Command`** entsprechend reagiert. Wenn man in der *build.xml* den „Gui Debugging Mode“ jedoch ausschaltet, wird der erzeugte Code automatisch keine der zu Debug Zwecken benutzten Anweisungen mehr enthalten, so das zum Erzeugen einer Release Version keine Änderungen am Quellcode mehr vorgenommen werden müssen.

```
import de.enough.polish.util.Debug;

public class MyMIDlet extends MIDlet {
  //#ifdef polish.useDebugGui
  private Command logCmd;
  logCmd = new Command("show log", Command.SCREEN, 10);
  //#endif

  private Screen mainScreen;
  [...]

  public MyMIDlet() {
    [...]
    //#ifdef polish.useDebugGui
    this.mainScreen.addCommand(this.logCmd);
    //#endif
  }

  public void commandAction(Command cmd,
                            Displayable screen) {
    [...]
    //#ifdef polish.useDebugGui
    if (cmd == logCmd) {
      this.display.setCurrent(
                  Debug.getLogForm(true, this));
    } else if (cmd == Debug.RETURN_COMMAND) {
      this.display.setCurrent(this.mainScreen);
    }
    //#endif
  }
```

Erstellen eines speziellen Debug Menüpunktes Beispiel 2

3.1.2 Styles und andere Dinge

Die Möglichkeiten des Preprocessing innerhalb von J2ME Polish sind natürlich mit dem oben beschriebenen noch lange nicht ausgereizt. So bietet der Preprozessor zusammen mit den Klassen von J2ME Polish einen mächtigen Zusammenschluss, der sehr viel Potiential für effektive und schnelle Programmierung bietet.

Ein Beispiel hierfür ist unter anderem auch die Möglichkeit innerhalb des Codes die Direktive `//#style` zu benutzen, um dem verwendeten Design Element, wie einem Listeneintrag, einen Style zu geben, der wiederum in der J2ME Polish eigenen css Datei definiert wird. So ist es sehr einfach, der eigenen Anwendung abhängig von bestimmten äusseren Einflüssen, wie zum Beispiel Farbtiefe des Displays, einen speziellen Look zu verpassen. Hiermit kann also auch das komplette Applikationsdesign dynamisch angepasst werden.

Das Preprocessing ist eigentlich das zentrale Feature in J2ME Polish, es ist ein sehr mächtiges Werkzeug, mit dessen Hilfe man sehr einfach Programmcode schreiben kann, welcher der JAVA Syntax entspricht, das heisst, auch einfach ohne Vorhandensein von J2ME Polish kompiliert werden kann. Zusätzlich ist es bei Zuhilfenahme des Frameworks und der dazugehörigen Klassenbibliotheken sehr einfach möglich, für mehrere Geräte speziell auf die spezifischen Eigenschaften wie Speicher, Display oder verfügbare APIs zugeschnittene Programm Archive zu bauen. Da der komplette Funktionsumfang der Preprocessing API den Umfang dieser Arbeit sprengen würde, sei hierfür die weitergehende Dokumentation auf der J2ME Polish Webseite empfohlen.

Dort werden dann unter anderem die Möglichkeiten beschrieben, wie es möglich ist, auch komplexe Bedingungen innerhalb von Preprozessor Direktiven zu erstellen, zum Beispiel mehre in der *build.xml* oder in den anderen XML Dateien definierten Variablen mit Operatoren vergleichen und anhand des Ergebnisses dann Code einbinden oder auch nicht. Es ist sogar möglich, innerhalb einer Direktive den Wert der Preprozessor Variablen gezielt zu verändern, dafür stehen insgesamt 6 verschiedene Funktionen zur Verfügung, welche im Folgenden kurz vorgestellt werden:

`uppercase`, `lowercase`: Wandeln einen gegebenen String entweder komplett in Großbuchstaben oder aber in Kleinbuchstaben um. Ein kleines Beispiel zur Verwendung dieser Funktionen findet sich in Beispiel 3

`bytes`, `kilobytes`, `megabytes`, `gigabytes`: Gibt den berechneten Wert des angegebenen Speicherwertes in der angegebenen Grössenordnung zurück. Verwendet werden kann diese Funktion unter anderem so wie in Beispiel 4 exemplarisch aufgezeigt

```
//#ifdef start-url:defined
    //#= private String url = "${ lowercase(start-url) }";
//#else
    private String url = "http://192.168.101.101";
//#endif
```

Stringmanipulation in Preprozessor Direktiven Beispiel 3

```
//#if ${ bytes( polish.HeapSize ) } > ${ bytes(100 kb) }
  [...]
//#endif
```

Umrechnung von Speicherwerten in Preprozessor Direktiven Beispiel 4

3.2 API wrapping

Ein weiterer Punkt der J2ME Polish so interessant für den Entwickler von Midlets macht ist sicherlich die Tatsache des „API wrapping“. Was ist damit gemeint?

Wie bereits im zweiten Kapitel erwähnt, gibt es deutliche Unterschiede im Funktionsumfang zwischen MIDP1 und MIDP2, hauptsächlich was die Möglichkeit anbelangt, grafisch ansprechende Anwendungsdesigns mit Hilfe des `GameCanvas` bzw `Sprite` zu erstellen. Als Entwickler steht man also immer vor der schwierigen Entscheidung, entweder massenkompatibel zu programmieren, um möglichst viele Endgeräte abzudecken, sprich: Nur die Methoden und Funktionen aus dem MIDP1 Standard zu verwenden, oder aber den zweiten Weg zu beschreiten, und komfortabel zu programmieren, mit allen Annehmlichkeiten, welche der MIPD2 Standard zu bieten hat, aber im Gegenzug auf die Kompatibilität mit vielen aktuell am Markt vertretenen Geräten zu verzichten.

J2ME Polish bietet hier die optimale Lösung. Es bringt sogenannte Wrapper Klassen mit. Das sind Klassen, welche die Funktionalität des MIDP2 Standards mit den Methoden des MIDP1 Standards implementieren. Wenn mann nun also mit J2ME Polish seine Anwendungen entwickelt, braucht man sich um die mangelnde Unterstützung benutzter Befehle in einigen Geräten keine Sorgen mehr zu machen. Das Framework wird beim Erstellen von Programmarchiven für MIDP1 Geräte automatisch dafür sorgen, daß die Methoden der Wrapper Klassen benutzt und diese auch mit in das Archiv aufgenommen werden.

3.3 *api.xml & devices.xml*

Die beiden XML Dateien *api.xml* und *devices.xml* bilden sozusagen das Herzstück von J2ME Polish, auf dem alles andere aufbaut. In diesen Dateien befindet sich sozusagen geballtes Wissen über verschiedenste Geräte komprimiert und in ein XML Schema eingepasst. Mit Hilfe dieser zwei Dateien kann J2ME Polish beim Preprocessing Vorgang und beim anschliessenden Compiliervorgang die optimale Konfiguration für das gewünschte Endgerät bestimmen und einen möglichst optimalen Bytecode erzeugen lassen.

In der Datei *devices.xml* stehen sehr viele (aktuell 182) verschiedene mobile Engeräte mit entsprechenden Information über Speichergrösse, Diplaygröße, Farbtiefe, unterstützte APIs und vieles mehr. So ist diese Datei hauptverantwortlich dafür, die benötigten Informationen zu liefern wenn es darum geht zum Beispiel aus unterschiedlichen Resourcen (wie Grafikdateien) die passenden herauszusuchen und nur diese nachher auch wirklich in das Programm Archiv hineinzupacken, so das am Ende ein möglichst kleines Archiv zustanden kommt.

Nachfolgend ist im Beispiel 5 der Inhalt der *devices.xml* exemplarisch anhand des Nokia N-Gage Handys aufgeführt.

```
<device>
  <identifier>Nokia/N-Gage_QD</identifier>
  <groups>Series60</groups>
  <capability name="OS" value="Symbian OS 6.1" />
  <capability name="ScreenSize" value="176x208" />
  <capability name="BitsPerPixel" value="12" />
  <capability name="JavaPackage" value="mmapi, wmapi" />
  <capability name="JavaPlatform" value="blah" />
  <capability name="VideoFormat" value="3gpp" />
  <capability name="SoundFormat" value="midi,amr,wav" />
  <capability name="HeapSize" value="16 MB" />
  <capability name="MaxJarSize" value="4 MB" />
</device>
```

Spezifikation der gerätespezifischen Eigenschaften, devices.xml Beispiel 5

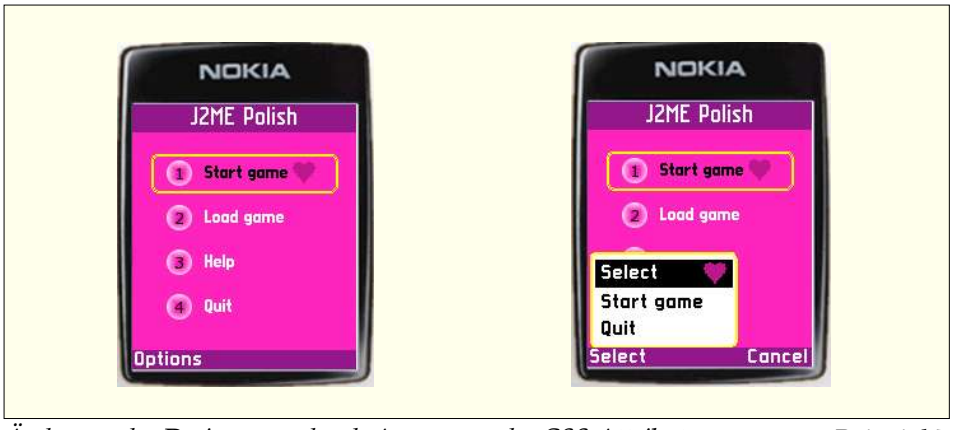
3.4 *polish.css*

In der Datei *polish.css* werden, wie der Name schon sagt, die CSS Styles definiert und verwaltet. Diese CSS Styles werden benutzt um das Erscheinungsbild der Applikation komfortabel zu designen. Mit Hilfe von CSS Styles in Verbindung mit der schon erwähnten Preprozessor Direktive `//#style` kann der Applikation ein ansprechendes Äußeres verpasst werden, welches man durch blosses ändern der CSS Attribute in der *polish.css* ändern kann. Dies kann auch sehr schön anhand eines kleinen Beispieles gezeigt werden:

Beispiel 6 zeigt das Applikationsdesign einer kleinen Beispielanwendung, welche mit Hilfe von CSS designed wurde.

Beispielapplikation mit exemplarischem Design Beispiel 6

Beispiel 7 zeigt genau die gleiche Anwendung, es wurden nur die CSS Styles innerhalb der *polish.css* Datei geändert.



Änderung des Designs nur durch Anpassung der CSS Attribute Beispiel 7

3.5 Einbinden von Obfuskatoren

J2ME Polish bietet die Möglichkeit sogenannte Obfuskatoren einzubinden. Unter einem Obfuskator versteht man ein Programm, welches vor dem eigentlichen Kompiliervorgang (hier: zwischen dem Preprozessor und dem JAVA Compiler) den Quellcode noch einmal durchgeht und umschreibt. So werden von einem Obfuskator zum Beispiel überflüssige, weil nicht benutzte, Pakete aus dem `import` Zweig herausgelöscht, nicht benutzte Variablen und Methoden werden eliminiert, immer mit dem Ziel, den resultierenden Bytecode möglichst klein zu halten. Damit wird der meist sehr geringen Speichergröße der mobilen Endgeräte Rechnung getragen.

Zusätzlich zum Eliminieren des überflüssigen Programmcodes werden auch sämtliche Namen von Klassen, Variablen und Methoden auf einen möglichst kurzen Namen geändert, dies dient nicht nur der nochmaligen, deutlichen Verkleinerung des schlussendlichen Bytecodes, zusätzlich wird der auch von „überflüssigen" Kommentaren und Debug Informationen befreite Code dadurch auch sehr sehr schwer lesbar, und ist für einen potentiellen Schaduser, der versucht sich die Programmlogik durch ein Decompilieren anzueignen, fast nicht mehr lesbar.

Durch diese „Security by obscurity" wird ein Ideenklau zwar nicht unbedingt verhindert, aber auf jeden Fall deutlich erschwert. Viele User sind durch die nach dem Obfuskator entstehenden Programmzeilen mit Ihren meist nur ein oder zwei Zeichen langen Variablen, Klassen und Methoden schon abgeschreckt. Ein Nachvollziehen des Programmes und der einzelnen Algorithmischen Schritte wird dadurch fast bis zur Unmöglichkeit erschwert.

Standardmässig ist in J2ME Polish der Obfuskator *ProGuard* integriert. Man kann aber auch den ebenfalls mitgelieferten *RetroGuard* verwenden, ebenfalls ein leistungsfähiger Obfuskator. J2ME Polish erlaubt es auch, noch andere, zusätzliche Obfuskatoren einzubinden. Dies geschieht über das Einfügen von verschiedenen `<obfuscator>` Elementen innerhalb der *build.xml*. Die Praxis hat jedoch gezeigt, das unter Verwendung von *ProGuard* als Obfuskator der kleinste Bytecode entsteht. Bei Verwendung von mehreren, zum Beispiel *ProGuard* und *RetroGuard* zusammen entsteht sogar grösserer Bytecode als bei Verwendung nur einer der beiden Obfuskatoren.

3.6 Erstellen von optimierten Jars

Durch die integrierte Device Datenbank in der Datei *devices.xml* ist es mit Hilfe von J2ME Polish sehr einfach möglich, speziell optimierte JARs für die einzelnen Geräte zu bauen. Man kann dabei die gewünschten Geräte entweder direkt mit ihrem Namen angeben, so wie sie im **<identifier>** Tag beschrieben sind, oder aber man kann auch hier wieder komplexe Bedingungen einführen, um so zum Beispiel für alle J2ME Polish bekannten Geräte, welche eine bestimmte API unterstützen oder deren Display mindestens eine Farbtiefe von 8 Bit hat, ein spezielles JAR zu bauen,.

```
<deviceRequirements>
  <requirement name="Identifier"
               value="Nokia/6230, Siemens/S65" />
</deviceRequirements>
```

Device Requirements für ein Siemens und ein Nokia Gerät Beispiel 8

Beispiel 8 zeigt einen solchen „einfachen“ Eintrag in der *build.xml* mit dessen Hilfe J2ME Polish genau 2 JARs und die dazugehörigen JADs bauen wird. In diesem Falle wäre es eine Version für ein Nokia 6230 und ein Siemens S65. Im Falle einer gewünschten generellen Abdeckung aller Geräte die zum Beispiel dem MIDP2 Standard entsprechen und zusätzlich noch die API zur Verarbeitung von Nachrichten wie SMS implementieren, würde der Eintrag folgendermassen aussehen:

```
<deviceRequirements>
  <requirement name="JavaPlatform" value="MIDP/2.0+" />
  <requirement name="JavaPackage" value="wmapi" />
</deviceRequirements>
```

Device Requirements für alle Geräte mit MIDP2 und wmapi Beispiel 9

Erklärung des Verfassers

Hiermit erkläre ich an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nichtveröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt.

Der Fachhochschule Karlsruhe, vertreten durch den Fachbereich Informatik, wird für die Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Seminararbeit einschliesslich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Karlsruhe, 12. Dezember 2004 Thomas Kraft

LiteraturVerzeichnis:

[1] J2ME Polish (2004): devices.xml

[2] Hamer, Carol (2004): „J2ME Games with MIDP2“, Berkeley: Apress S. XV

[3] Virkus, Robert (2004): „The Complete Guide to J2ME Polish“. URL: http://www.j2mepolish.org/downloads/Complete_Guide_to_J2ME_Polish.pdf

[4] J2ME Polish online documentation (2004). URL: http://www.j2mepolish.org/docs/directives.html

Linkverzeichnis:

Antenna: http://antenna.sourceforge.net/

kAWT: http://www.kawt.de/

Synclast UI API: http://www.synclast.com/

JTGL: http://sourceforge.net/projects/jtgl

ProGuard: http://proguard.sourceforge.net/

RetroGuard: http://www.retrologic.com/