iPhone meets Android

XML Virtual Machine

Wolfgang Korn, Arno Puder, Sascha Häberling

Das iPhone erfreut sich seit seiner Verfügbarkeit im Jahr 2007 großer Beliebtheit. Bei der Werkzeugunterstützung zur Entwicklung von iPhone-Anwendungen hat sich aber seitdem wenig verändert. Apple setzt auf die Sprache Objective-C und die Entwicklungsumgebung Xcode. Im folgenden Artikel wird ein innovativer Ansatz zur Entwicklung von iPhone-Anwendungen beschrieben, der zudem auch Portabilität verspricht. Mithilfe des Open-Source-Projekts XMLVM lassen sich Anwendungen entwickeln, die sowohl unter dem auch auf Smartphones zum Einsatz kommenden Betriebssystem Android als auch auf dem iPhone genutzt werden können.

Smartphones, wie das iPhone, Android-basierte Endgeräte oder der kürzlich erschienene Palm Pre verdrängen die herkömmlichen Mobiltelefone in zunehmendem Maße. Einer der Gründe für diese Entwicklung ist sicherlich die Möglichkeit, auf diesen Geräten beliebige Anwendungen installieren und ausführen zu können. Insbesondere für die beiden erstgenannten Geräte ist bereits eine Vielzahl frei verfügbarer und kommerzieller Anwendungen verfügbar. Vergleicht man die Hardware der verschiedenen Geräte, stellt man fest, dass diese sehr ähnlich sind. Große Displays, WLAN, Beschleunigungsmesser und GPS gehören inzwischen zur Standardausstattung.

Implementierungsvielfalt

So ähnlich die Hardware auch ist, umso unterschiedlicher gestaltet sich die Art, wie Anwendungen für die verschiedenen Geräte entwickelt werden können. Android-Anwendungen werden in Java entwickelt und verwenden ein Android-spezifisches API, welches auch eine Untermenge des Standard-Java-API enthält. Als Entwicklungsumgebung kann damit prinzipiell jede Java-IDE nebst den von Google bereitgestellten Entwicklungswerkzeugen verwendet werden. Darüber hinaus existiert für Eclipse auch ein Android-Plug-In, welches die Entwicklung vereinfacht.

Deutlich eingeschränkter sieht es auf der iPhone-Seite aus. Hier ist die einzige verfügbare Programmiersprache Objective-C, mit welcher in der von Apple zur Verfügung gestellten Entwicklungsumgebung Xcode entwickelt wird. Als API zur Entwicklung von iPhone-Anwendungen kommt Cocoa Touch zum Einsatz. Erschwerend kommt hinzu, dass Apple die Nutzung jeglicher Interpreter- und Virtual Machine-Technologien auf dem iPhone untersagt. So erklärt sich, warum eine so verbreitete Sprache wie Java für dieses Gerät nicht genutzt werden kann.

Anwendungsentwicklung für den Palm Pre wiederum erfolgt mithilfe von Webtechnologien wie HTML und JavaScript.

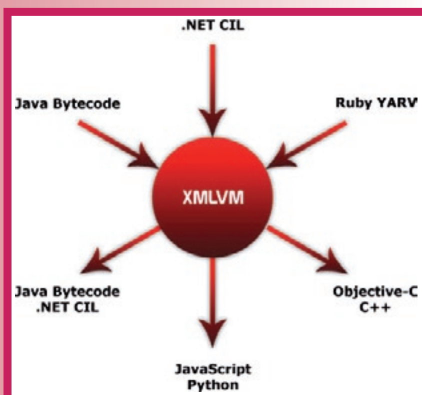

Abb. 1: XMLVM-Crosscompilierung

Aufgrund der geschilderten Vielfalt besteht für Anwendungen, die auf mehr als einer Plattform verfügbar sein sollen, derzeit nur die Möglichkeit von mehreren voneinander unabhängigen Implementierungen. Abhilfe verspricht das Open-Source-Projekt XMLVM. Nach einer Vorstellung der zugrunde liegenden Konzepte des Projekts zeigt der Beitrag, wie mithilfe von XMLVM Anwendungen entwickelt werden, die eine gemeinsame Code-Basis haben und ohne Portierungsaufwand sowohl unter Android als auch auf dem iPhone genutzt werden können.

XMLVM im Überblick

Den Kern von XMLVM bildet ein flexibles Crosscompilierungs-Werkzeug. Im Gegensatz zu anderen Crosscompilern operiert XMLVM nicht auf Quelltextebene, sondern auf Bytecode-Ebene. Die Eingabe für XMLVM sind Bytecode-Instruktionen von Suns Java Virtual Machine (JVM) oder von Microsofts Common Language Runtime (CLR). Hieraus generiert XMLVM Quelltext für verschiedene Zielsprachen, die dann entweder mit einem passenden Compiler wiederum in Binärcode übersetzt oder durch einen Interpreter ausgeführt werden können.

Abbildung 1 zeigt die derzeit unterstützten Quellen und Zielsprachen. Die Vollständigkeit der Unterstützung variiert momentan noch zwischen den Sprachen und ist für Java, .NET und Objective-C am vollständigsten. Die Anwendungsmöglichkeiten für diese Technologie sind vielfältig. Die Generierung von Ajax-Anwendungen auf Basis von Java-Desktop-Anwendungen [XMLVMJS] ist ebenso möglich wie die portable Entwicklung mobiler Anwendungen, wie es in diesem Beitrag vorgestellt wird.

XML Virtual Machine

Die grundlegende Arbeitsweise von XMLVM ist unabhängig von Quell- und Zielsprache immer gleich und erklärt auch den Namen des Projekts. In einem ersten Schritt überführt ein Compiler-Frontend das Quellprogramm in das interne XMLVM-Format. Hierbei handelt es sich um eine XML-Repräsentation des Programms, die im Wesentlichen Bytecode-Instruktionen von Suns JVM enthält. Wie die JVM implementiert auch Microsofts CLR Stack-basierte Operationen. Wenngleich zwischen den Operationen dieser beiden VMs Unterschiede bestehen, ist eine Abbildung zwischen ihnen möglich. Quellprogramme, die CLR-Bytecode enthalten, werden von XMLVM daher ebenfalls in dieses Format überführt. Details zur Übersetzung des CLR-Bytecodes in Java-Bytecode finden sich unter [XMLVM].

NUR für Abonnenten: Haben Sie schon Ihren Log-In für den kostenlosen Download aller Artikel-pdfs? Kontakt: Susanne.Herl@sigs-datacom.de

Die XMLVM-Darstellung einer einfachen Klasse zur Fakultätsberechnung zeigt Listing 1. Innerhalb des Elements `<vm:class>` finden sich die Details der Klasse `Fact`. Die geschachtelten Elemente `<vm:method>` spezifizieren die Methoden der Klasse. Neben der Spezifikation der Methodensignatur finden sich hier insbesondere auch Informationen über die Zahl der lokalen Variablen und über den benötigten Platz auf dem Stack. Schließlich folgen in den Zeilen 12 bis 37 die Bytecode-Instruktionen der Methode, eingebettet in das Element `<vm:code>`.

In den Zeilen 13 und 14 werden die lokalen Variablen und Parameter deklariert. Die Variable mit Index 0 ist immer die this-Referenz. Unter Index 1 findet sich der Aufrufparameter n. Die Zeilen 16 bis 19 liefern das Ergebnis für `fact(0)`. Dazu wird mittels `<jvm:iload>` der Wert von n auf dem Stack abgelegt. `<jvm:ifne>` testet, ob das oberste Element des Stacks ungleich 0 ist, und springt gegebenenfalls zum angegebenen Label. Für n=0 ist dies jedoch nicht der Fall und es wird in Zeile 18 durch die Instruktion `<jvm:iconst>` die Konstante 1 auf dem Stack abgelegt, die nach einem `<jvm:goto>` zum Label mit id=1 mittels `<jvm:ireturn>` als Ergebnis geliefert wird.

In den Zeilen 21 bis 24 erfolgt die rekursive Berechnung von `n*fact(n-1)`. Zunächst erfolgt dazu der Aufbau des Stacks. Mittels `<jvm:iload>` werden die Inhalte der lokalen Variable n und mittels `<jvm:aload>` die this-Referenz für den nachfolgenden rekursiven Aufruf auf den Stack gespeichert. Der rekursive Aufruf erfolgt durch die `<jvm:invokevirtual>`-Instruktion, welche die Referenz des aufzurufenden Objekts und alle Parameter vom Stack liest und den Return-Wert auch wieder dort ablegt. Die abschließende Multiplikation erfolgt in Zeile 34 mithilfe der `<jvm:imul>`-Instruktion.

```
01 public class Fact {
02   public int fact(int n) {
03     return n == 0 ? 1 : n * fact(n - 1);
04   }
05 }

01 <vm:xmlvm xmlns:vm="http://xmlvm.org"
02           xmlns:jvm="http://xmlvm.org/jvm">
03   <vm:class name="Fact" isPublic="true"
04             isSynchronized="true"
05             extends="java.lang.Object">
06     <vm:method name="fact" isPublic="true"
07                stack="4" locals="2">
08       <vm:signature>
09         <vm:return type="int" />
10         <vm:parameter type="int" />
11       </vm:signature>
12       <vm:code language="ByteCode">
13         <jvm:var name="this" id="0" type="Fact" />
14         <jvm:var name="arg0" id="1" type="int" />
15         <jvm:label id="2" />
16         <jvm:iload type="int" index="1" />
17         <jvm:ifne label="0" />
18         <jvm:iconst type="int" value="1" />
19         <jvm:goto label="1" />
20         <jvm:label id="0" />
21         <jvm:iload type="int" index="1" />
22         <jvm:aload type="java.lang.Object"
23                    index="0" />
24         <jvm:iload type="int" index="1" />
25         <jvm:iconst type="int" value="1" />
26         <jvm:isub />
27         <jvm:invokevirtual class-type="Fact"
28                            method="fact">
29           <vm:signature>
30             <vm:return type="int" />
31             <vm:parameter type="int" />
32           </vm:signature>
33         </jvm:invokevirtual>
34         <jvm:imul />
35         <jvm:label id="1" />
36         <jvm:ireturn />
37       </vm:code>
38     </vm:method>
39   </vm:class>
40 </vm:xmlvm>
```

Listing 1: Fakultätsberechnung in XMLVM-Darstellung

Quelltextgenerator

Ausgehend von dieser XML-Repräsentation des Eingabeprogramms generieren die Compiler-Backends Quelltext für die verschiedenen unterstützten Zielsprachen. Dabei wird unabhängig von der Zielsprache grundsätzlich das Verhalten der Stack-basierten VM auch in der Zielsprache simuliert.

Listing 2 zeigt anhand der bereits besprochenen `<jvm:imul>`-Instruktion (Integer-Multiplikation), wie mithilfe einer XSL-Transformation die verschiedenen Instruktionen in die jeweiligen Zielsprachen übersetzt werden. `<jvm:imul>` liest zwei Integer-Werte vom Stack, ermittelt das Produkt und schreibt das errechnete Ergebnis wieder auf den Stack zurück. `_stack` repräsentiert den Stack, der über die Variable `_sp` (Stackpointer) referenziert wird. Prädekrement- und Postinkrement-Operationen simulieren das Verhalten der Stack-Operationen POP und PUSH. Die Deklaration der verwendeten Hilfsvariablen erfolgt separat für jede Methode in einem Template, welches das Tag `<vm:method>` verarbeitet.

Java für iPhone

Die oben gezeigte Abbildung von Java auf Objective-C hat die Voraussetzung geschaffen, ausgehend von einer Java-Implementierung Objective-C-Quellcode zu generieren. Dieser kann mit dem von Apple bereitgestellten Objective-C-Compiler in eine native iPhone-Anwendung übersetzt werden. Die Sprachabbildung ist neben weiteren Details, wie z. B. die nicht näher erläuterte Abbildung der Garbage Collection, auf dem Weg zu einer in Java implementierten iPhone-Anwendung aber nur ein Teil der Lösung.

Ebenso wichtig ist die Portierung des Cocoa-Touch-API in die Java-Welt, da Anwendungen erst hierdurch Zugriff auf die zahlreichen iPhone-spezifischen Widgets ermöglicht wird. Eine prototypische Java-Anbindung des Cocoa-Touch-API ist Bestandteil von XMLVM. Das Java-API lehnt sich eng an die native Cocoa-Touch-Version an, sodass Erfahrungen mit dem nativen Cocoa-Touch-API hierauf übertragbar sind.

Listing 3 zeigt eine Java-Implementierung des klassischen HelloWorld-Programms für das iPhone. Einstiegspunkt in die Anwendung ist, wie auch bei reinen Objective-C-Implementierungen, die Methode `applicationDidFinishLaunching`. In den Zeilen 4 bis 6 wird zunächst ein Fenster mit der gesamten Größe des Bildschirms erzeugt. Anschließend wird dem Fenster eine Sicht mit der gleichen Größe hinzugefügt. In den Zeilen 12 bis 16 wird schließlich ein Label in Bildschirmgröße erzeugt, der den Text „Hello World“ anzeigt. Abschließend wird in Zeile 18 das Fenster sichtbar gemacht.

```
<xsl:template match="jvm:imul">
  <xsl:text>
    _op2.i = _stack[--_sp].i;           // Pop op1
    _op1.i = _stack[--_sp].i;           // Pop op2
    _stack[_sp++].i = _op1.i * _op2.i;  // Multiply
  </xsl:text>
</xsl:template>
```

Listing 2: Objective-C-Mapping der jvm:imul-Instruktion

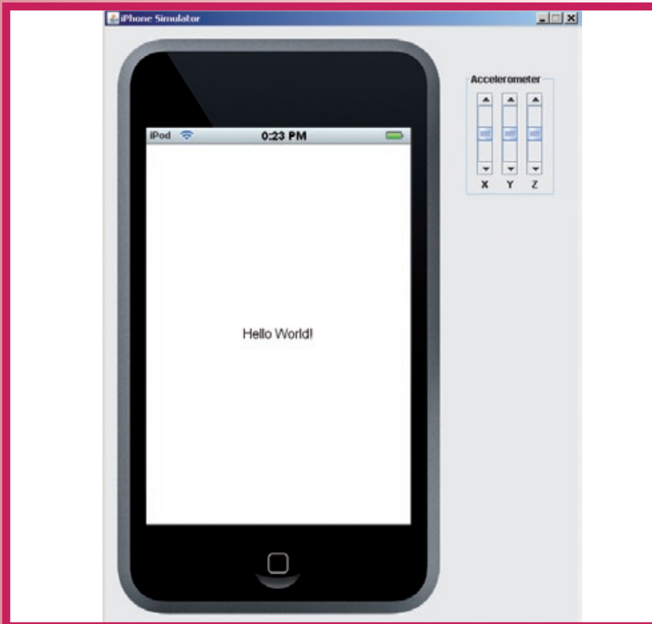

Abb. 2: XMLVM-iPhone-Simulator

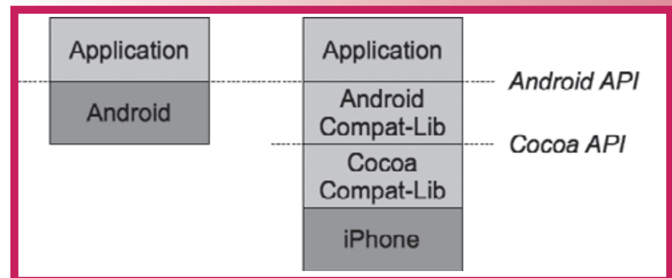

Abb. 3: Beziehung Android/iPhone-Anwendung

Neben der Anbindung des Java-API an die entsprechenden Objective-C-APIs stellt XMLVM auch eine reine Java-Implementierung des API bereit. Diese wird genutzt, wenn während der Entwicklung Anwendungen in XMLVMs eigenem Simulator ausgeführt werden. Dieser in Abbildung 2 gezeigte Simulator unterstützt effektiv die Fehlersuche in Java-iPhone-Anwendungen, da sich hiermit Anwendungen in jedem Java-Debugger ausführen lassen.

```
01 public class HelloWorld extends UIApplication {
02  public void applicationDidFinishLaunching(
03                 NSNotification n) {
04    UIScreen screen = UIScreen.mainScreen();
05    CGRect rect = screen.applicationFrame();
06    UIWindow window = new UIWindow(rect);
07
08    rect.origin.x = rect.origin.y = 0;
09    UIView mainView = new UIView(rect);
10    window.addSubview(mainView);
11
12    UILabel title = new UILabel(rect);
13    title.setText("Hello World!");
14    title.setTextAlignment(
15        UITextAlignment.UITextAlignmentCenter);
16    mainView.addSubview(title);
17
18    window.makeKeyAndVisible();
19  }
20 }
```

Listing 3: HelloWorld für das iPhone

Kanonisches API

Die Verfügbarkeit einer Java-Implementierung des Cocoa-Touch-API rückt das Ziel einer gemeinsamen Code-Basis für Android- und iPhone-Anwendungen in greifbare Nähe. Die Lücke der unterschiedlichen Implementierungssprachen auf den beiden Plattformen wurde durch die Sprachabbildung und eine passende Anbindung des iPhone-API geschlossen. Was bleibt, sind die unterschiedlichen APIs, die bei der Programmierung von Android- und iPhone-Applikationen zum Einsatz kommen. Auf Basis der bereits beschriebenen Techniken lässt sich diese letzte Lücke jedoch leicht schließen.

Zur Entwicklung von Anwendungen, die für beide Plattformen verfügbar sein sollen, wird das Android-API zum kanonischen API erklärt, welches für beide Plattformen Gültigkeit besitzt. Da mit XMLVM eine gemeinsame Implementierungssprache zur Verfügung steht, lässt sich durch Implementierung einer Kompatibilitäts-Bibliothek das Android-API auf das Java-Cocoa-Touch-API abbilden.

Abbildung 3 verdeutlicht diesen Ansatz. Auf der linken Seite wird die Android-Applikation gezeigt, die unmittelbar auf dem Android-API aufsetzt und so auf einem Android-Gerät lauffähig ist. Da für beide Plattformen die gleiche Code-Basis genutzt wird, enthält der Code der Anwendung keinerlei Hinweise darauf, dass auch die Lauffähigkeit auf dem iPhone erreicht werden soll. Die rechte Seite der Abbildung zeigt die Struktur der gleichen Anwendung in der iPhone-Version. Hier setzt die Anwendung auch auf dem Android-API auf. Allerdings handelt es sich bei dieser Version des API um eine Reimplementierung, welche das Android-API auf das Java-Cocoa-Touch-API abbildet. Letzteres übernimmt, wie oben vorgestellt, die Aufrufe des nativen iPhone-API. Sowohl Applikation als auch die Android-Kompatibilitäts-Bibliothek sind in Java implementiert und durchlaufen daher die durch XMLVM durchgeführte Crosscompilierung.

Der in Listing 4 gezeigte Auszug aus der Kompatibilitäts-Bibliothek verdeutlicht, wie die Abbildung der APIs aufeinander funktioniert. Der Code entstammt der Reimplementierung der Android-Klasse `Button` und demonstriert die Registrierung eines Handlers, der das Drücken des Buttons verarbeitet. Anhand der deklarierten Instanz-Variablen `button` ist erkennbar, dass ein Android-Button auf dem iPhone auf `UIButton` abgebildet wird. Wird für den Android-Button ein

```
package android.widget;

public class Button extends View {
  protected UIButton button;

  // ...

  public void setOnClickListener(OnClickListener
                listener) {
    final OnClickListener theListener = listener;
    button.addTarget(new UIControlDelegate() {
      @Override
      public void raiseEvent() {
        theListener.onClick(Button.this);
      }
    }, UIControl.UIControlEventTouchUpInside);
  }
}
```

Listing 4: Auszug aus der Android-Kompatibilitäts-Bibliothek

NUR für Abonnenten: Haben Sie schon Ihren Log-In für den kostenlosen Download aller Artikel-pdfs? Kontakt: Susanne.Herl@sigs-datacom.de

Handler registriert, löst dies die Erzeugung eines anonymen `UIControlDelegate` aus, welches den Handler aufruft. Das `UIControlDelegate` wird als Target am `UIButton` registriert. Bei Betätigung des Buttons wird zunächst das `UIControlDelegate` aufgerufen, welches den Aufruf an den gemäß Android-API implementierten `ClickListener` weiterleitet.

Die Tragfähigkeit dieses Ansatzes wird durch verschiedene auf [XMLVM] gezeigte Showcases untermauert. Unter anderem findet sich hier eine Implementierung des bekannten Sokoban-Spiels, welches sowohl unter Android als auch auf dem iPhone lauffähig ist.

Fazit

Das vorgestellte Open-Source-Projekt XMLVM verfolgt einen innovativen Ansatz zur Überbrückung der unterschiedlichen Technologien bei der Implementierung von Smartphone-Anwendungen. Durch die unterschiedlichen bereits vorhandenen Crosscompiler-Backends beschränken sich die unterstützbaren Plattformen nicht nur auf das iPhone. Auf Basis des JavaScript-Backends ist es bereits möglich, ausgehend von der gleichen Code-Basis auch Anwendungen für den Palm Pre zu erzeugen. Aufgrund der Mächtigkeit der aufeinander abzubildenden Bibliotheken ist bei der Implementierung der Kompatibilitäts-Bibliotheken jedoch noch viel Arbeit zu leisten, die auch nur mit Unterstützung weiterer Entwickler bewältigt werden kann. Aber bereits die vorhandene Implementierung erlaubt die Erstellung sinnvoller Anwendungen.

Neben der Entwicklung von Smartphone-Anwendungen existieren auch weitere interessante Einsatzmöglichkeiten für die vorgestellte Technologie. Weitere Informationen, wie z. B. Online-Dokumentation und Quelltexte, finden sich auf der XMLVM-Homepage unter [XMLVM].

Links

[XMLVM] XMLVM-Homepage, `http://xmlvm.org`
[XMLVMJS] XMLVM-JavaScript-Backend, `http://xmlvm.org/javascript/`

Wolfgang Korn ist Core Developer im XMLVM-Projekt und arbeitet als Consultant für die blueCarat AG in Köln. Die blueCarat AG unterstützt XMLVM aktiv durch die Bereitstellung von Entwicklerkapazitäten.
E-Mail: wolfgang@xmlvm.org.

Arno Puder ist Begründer des XMLVM-Projekts und lehrt als Professor an der San Francisco State University.
E-Mail: arno@xmlvm.org.

Sascha Häberling ist Core Developer im XMLVM-Projekt und arbeitet als Entwickler für Google in Zürich.
E-Mail: sascha@xmlvm.org.