Baccalaureatsarbeit im Projekt

Management verteilter Geschäftsprozesse für mobile Anwendungsbereiche

Vergleich der Entwicklung von Diensten für JavaSE, JavaME und Android

Cristina Mossoni

7cmossoni@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 5055284
Fachsemester 24

Inhaltsverzeichnis

Einleitung

Mobile Kommunikation spielt heutzutage eine immer größere Rolle. Nicht nur zum Telefonieren sondern für eine Vielzahl von Funktionalitäten, die vom Alltag bis zur Arbeitswelt reichen, kann man sich immer komplexer werdender mobilen Lösungen bedienen. Der Markt boomt zurzeit mit Innovationen in diesem Bereich. Verschiedene Unternehmen versuchen folglich, sich Anteile eines so wichtigen Marktes zu sichern, grenzen sich jedoch auch durch unterschiedliche Technologien voneinander ab. Einige davon sind schon etabliert, viele neue entstehen und Programmierer haben nun die Wahl zwischen verschiedenen Systemen und Programmiersprachen.
Diese Studie beschäftigt sich mit einem Vergleich einiger Technologien und insbesondere mit einer neuen Erscheinung: *Android*[1] von Google, und stellt diesen dem bisher de facto Standard *J2ME*[2] gegenüber, welcher zum jetzigen Zeitpunkt am meisten verbreitet ist und von den meisten Geräten unterstützt wird: Ist Android eine positive Neuerung oder bringt es nur Verwirrung auf dem ohnehin schon sehr vielfältigen Markt der Lösungen für mobile Geräte? Weiterhin ist zu klären, inwiefern Android als eine Java Variante angesehen werden kann.

Android wurde 2007 von der *Open Handset Alliance*[3] ins Leben gerufen, die sowohl Software als auch Hardware entwickelt und vertreibt. Aufgrund der an der Allianz beteiligten Unternehmen und der Tatsache, dass Teile des Quellcodes offengelegt wurden, kann man davon ausgehen, dass Android in Zukunft eine große Rolle für mobile Endgeräte spielen wird.

Ein Vergleich zwischen Java, JavaME und Android bedeutet, dass sowohl Laufzeitumgebungen als auch Programmiersprachen verglichen werden:
Java ist als einfache Programmiersprache entstanden und hat sich langsam zu einem kompletten System entwickelt. Den aktuellen Stand bildet die *Java 2 Standard Edition*[4] *(JavaSE) 1.6,* welche inzwischen ein großes Spektrum von Technologien bietet. Dazu gehören eine Laufzeitumgebung (JRE), ein Entwicklung-Framework (SDK) und einiges mehr. Wenn von Java die Rede ist, geht es also zum einen um die Sprachmerkmale aber

[1] Android Homepage: *http://www.android.com/*

[2] Sun Java Micro Edition: *http://java.sun.com/j2me*

[3] Open Handset Alliance: *http://www.openhandsetalliance.com/* Der Allianz gehören u.a. Carriers wie T-Mobile, Vodafone; Hardwarehersteller wie Asus, Acer, LG Electronics, Motorola, Samsung, Sony Ericsson; Softwarehersteller wie Google, ebay, Nuance, SVOX.. an.

[4] Sun Java Standard Edition: *http://java.sun.com/j2se*

zum anderen um eine gesamte Plattform, welche erlaubt, Java Programme zu entwickeln und auch auf verschiedenen Systemen auszuführen.

Die reduzierte Variante JavaME, die für die Programmierung mobiler Geräte eingesetzt wird, ist auch mehr als eine Programmiersprache: Damit ist ebenso ein gesamtes System gemeint, zum einen das API Framework und zum anderen die Laufzeitumgebung mit ihren verschiedenen Konfigurationen.

Android wird mal als Betriebssystem, mal als Programmiersprache, mal als Technologie verstanden. Es handelt sich auch hier in der Tat um eine ganze Plattform, konzipiert, um auf verschiedenen Geräten eigenständig zu laufen und ausgestattet mit allem, was für die Ausführung der Programme und Funktionalitäten benötigt wird. Ein Teil davon ist die eingesetzte Programmiersprache, welche mit JavaME nicht viel gemeinsam hat, außer dass sie ebenso auf Java basiert und ebenso eine Auswahl von Java-Merkmalen darstellt. Diese Programmiersprache wird in dieser Arbeit ab und zu als „Android-Java“ bezeichnet, wenn es um die reinen Sprachmerkmale geht; andernfalls betrachten wir mit dem Begriff „Android“ die gesamte Plattform.

Zum Aufbau dieser Arbeit:
Nach einer Beschreibung der im Projekt “Management verteilter Geschäftsprozesse für mobile Anwendungsbereiche“ realisierten Anwendung, wird im zweiten Teil die Implementierung unter JavaME und Android kontrastiv untersucht und soweit dies sinnvoll ist, werden diese Lösungen dem „gewöhnlichen“ Java gegenübergestellt.
Als roter Faden wird eine Beispielanwendung aus dem Projekt (ein Location Based Service: die Ermittlung und Verwendung von GPS-Positionsdaten) erweitert und vertieft.
Der Vergleich umfasst die Architektur der Plattformen und ihre Virtual Machines, die vielen Sprachunterschiede trotz der gemeinsamen Herkunft aus „Java“, den Einsatz von Entwicklungswerkzeuge, den strukturellen Aufbau der Applikationen bis hin zu den grafischen Bibliotheken und den Mechanismen der Verwaltung und Speicherung persistenter Daten.
Die letzten beiden Themen sind besonders wichtig: Die grafischen Oberflächen wegen ihren häufigen Einsatz im mobilen Anwendungsbereich und der Umgang mit persistenten Daten, weil es bei mobilen Geräten ein besonders heikles Gebiet ist: Aufgrund der beschränkten Ressourcen ist es nämlich erforderlich, besondere Strategien für die Datenspeicherung zu entwickeln. Darüber hinaus stellen generell mobile Geräte durch ihre Beschaffenheit und insbesondere ihre Mobilität neue Herausforderungen an das Thema „Datenbanken“.

1 Projekt „Management verteilter Geschäftsprozesse für mobile Anwendungsbereiche“

Mit der Zunahme der Mobilität der Anwender wird langsam das „Desktop-Modell“, also das Modell des festen Arbeitsplatzes, vom Modell des „Ubiquitous Computing“ abgelöst, in dem die Mensch-Computer-Interaktion viel häufiger und alltäglicher stattfindet: Unternehmensmitarbeiter sind heutzutage immer „mobiler“ und können ihre Arbeit an verschiedenen Orten und Endgeräten erledigen, Besitzer von mobilen Telefonen haben jederzeit Zugang zu Diensten und Informationen, Sensordaten und Kontextdaten stehen vielen Anwendungen stets zur Verfügung.

Im Projekt wurden der Entwurf, die Realisierung und die Analyse verteilter (Geschäfts-) Prozesse untersucht, die zum Teil auf mobilen Geräten laufen sollen.
Geschäftsprozesse sind als Abfolgen von Aktivitäten zur Erreichung eines übergeordneten Ziels zu verstehen.
Prozesse werden in der Regel in mehrere Dienste aufgeteilt: d.h. Softwarebausteinen, die jeweils eine in sich abgeschlossene Funktionalität kapseln und die man verschiedenen Teilnehmern zur Ausführung zuweisen kann; die Zusammenarbeit der Teilnehmern ergibt dann die Ausführung eines verteilten Prozesses.
Zur Koordination ist eine „Workflow Engine“ notwendig, welche die Steuerung der verschiedenen Komponenten übernimmt[5].

Wenn Dienste für *mobile* Geräte entwickelt werden, stellen sich neue Anforderungen, vorwiegend muss mit dem Problem der unzuverlässigen Netzverbindung umgegangen werden. Die im Projekt eingesetzte Middleware *Demac*[6] ist eine für mobile Prozesse konzipierte Workflow Engine. Auf der anderen Seite muss berücksichtigt werden, dass mobile Geräte im Vergleich zu Stationären ressourcenarm sind, so dass sie in der Regel nur Teile einer größeren Aufgabe bearbeiten können.
Im Projekt wurde eine solche verteilte Anwendung für ein Geschäftsprozess realisiert, die mit Demac gesteuert wird und deren Dienste teilweise von einem mobilen Gerät (Android) ausgeführt werden.

---

[5] L. Pajunen and S. Chande: *Developing Workflow Engine for Mobile Devices*, IIn EDOC ’07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, S. 279. IEEE Computer Society, 2007.

[6] Verteilte Systeme und Informationssysteme, Department Informatik, Universität Hamburg, DEMAC Homepage*: http://vsis-www.informatik.uni-hamburg.de/projects/demac/*

1.1 Die Projektaufgabe: ein Tankassistent

Als Beispiel einer verteilten Anwendung auf mobilen Geräten wurde ein Tankassistenten realisiert, der in Form eines Bord Computers bei Bedarf den Autofahrer bei der Suche nach einer günstigen Tankstelle entlang seiner Fahrstrecke unterstützt.
Dabei werden verschiedene Dienste in Anspruch genommen, die sich in separaten Anwendungen befinden und von Demac koordiniert werden.

1.1.1 Der Weg zur Implementierung

Gemäß dem Lebenszyklus eines Geschäftsprozesses ist die Anwendung in mehreren Schritten realisiert worden. Nach einer anfänglichen Anforderungsanalyse steht die Modellierungsphase, in der ein Prozess auf abstrakter Ebene beschrieben wird. In Anschluss wird der Prozess konkret in eine Datei für die Workflow Engine erfasst. Es folgt die Implementierung der einzelnen Diensten und schließlich ihre Ausführung (standalone zunächst, dann im gesamten Prozess). Bei der Ausführung des Prozesses kann man noch verschiedene Prozess Management Techniken zur Überwachung und Analyse anwenden, dadurch kann man die Applikation noch verbessern oder erweitern.

Anforderungsanalyse

Der erste Schritt besteht in einer Analyse der Anforderungen, die zum Beispiel von einem Auftraggeber in natürlichsprachlicher Form vorgelegt werden könnten. Aufgabe der Informatiker ist es, diese Anforderungen zu analysieren, auf Machbarkeit zu überprüfen und zu formalisieren.
In unserem Fall war die initiale Idee, einen Tankassistenten für das Autofahren zu realisieren unter Verwendung eines mobilen Geräts oder eines GPS-fähigen Geräts.

Abstrakte modellierung des Prozesses

Auf der Basis einer informalen Beschreibung ist der Prozess modelliert und mittels der *Business Process Modelling Language* BPMN[7] abstrakt beschrieben worden. Durch eine solche Beschreibung wird deutlich, wie und wann einzelne Schritte (sogenannte *Activities*). geschehen sollen (s. Abb. 1-1, 1-2). Dabei wurde darauf geachtet, dass nicht nur die

[7] Business Process Modeling Notation, *http://www.bpmn.org/*

zeitliche Abfolge der Activities sichtbar wird, sondern auch die Zuweisung der Dienste an die verschiedenen Teilnehmer. Letzteres ist durch die Verwendung verschiedener „Swim Lanes“ möglich.

In der „Hauptbahn“ stehen die Tätigkeiten der Hauptapplikation (Bord Computer); verschiedene andere Bahnen stellen die Inanspruchnahme externer Dienste dar, welche immer von der Hauptapplikation aus aufgerufen werden. Die externen Dienste sind hier entweder geschlossene Java-Projekte, die auf anderen Rechnern laufen, oder Android-Projekte, die auf Android (genauer gesagt auf dem Android Emulator) ebenso separat laufen. Alle führen ihren Teil der Berechnung aus und liefern ein Ergebnis zurück, zur weiteren Verwendung.

Im Folgenden wird das abstrakte Design mittels BPMN Diagramm erläutert.

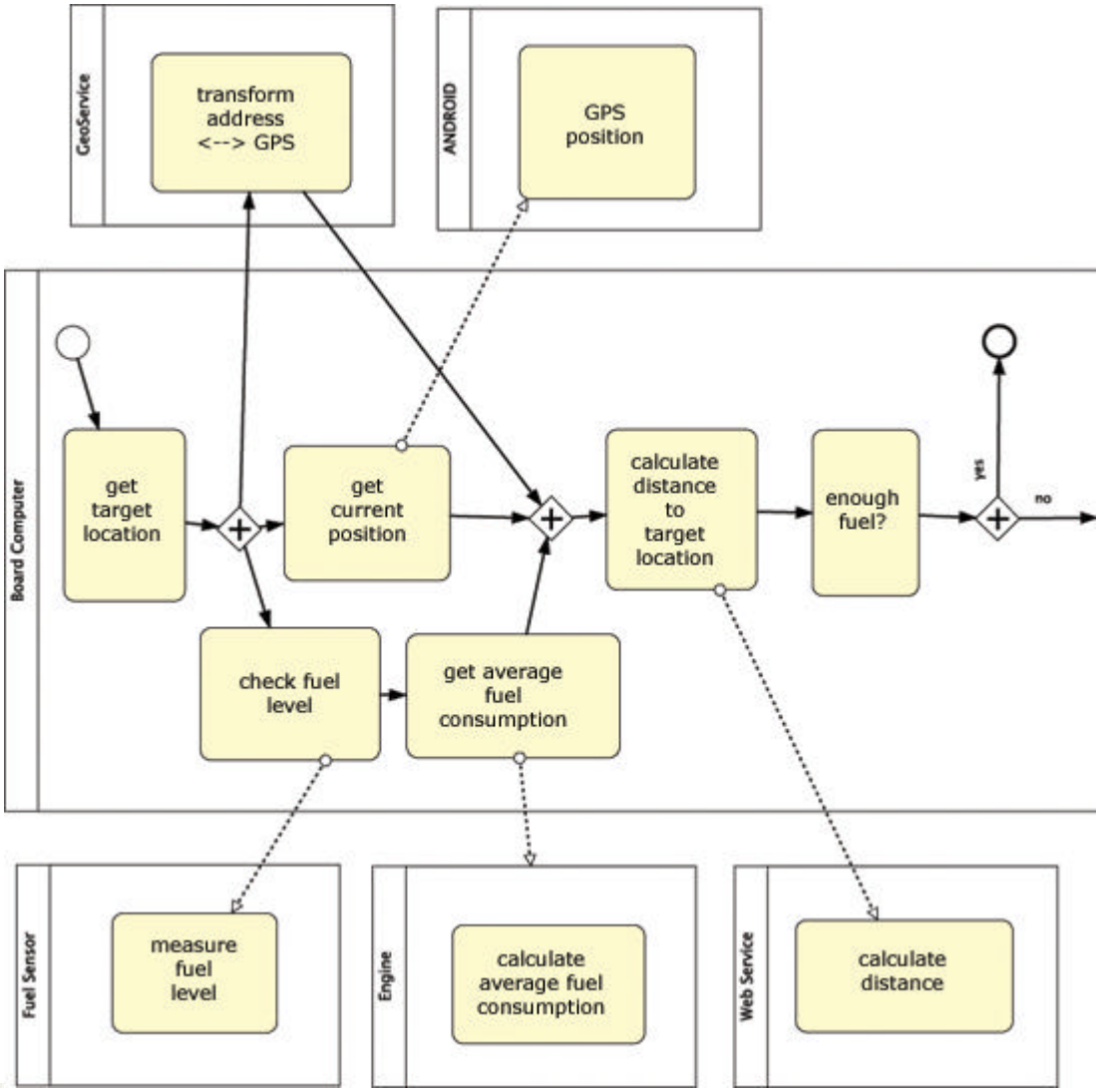

Abbildung 1-1 BPMN – Diagramm – Abschnitt 1

Abbildung 1-1: Die Hauptanwendung, vom Autofahrer bei Bedarf initiiert, fordert als erstes den Benutzer auf, sein Reiseziel in Form einer Adresse einzugeben (Activity *get target location*). Das Ziel wird in Geo-Koordinaten umgewandelt, gleichzeitig[8] wird der Tankstand abgefragt und die aktuelle Position von Android angefordert. Nach einer Berechnung des Durchschnittsverbrauchs und der Entfernung zum Ziel entscheidet das Hauptprogramm, ob eine Tankstellensuche notwendig ist. Andernfalls endet die Ausführung (idealerweise mit einer Meldung des Systems).

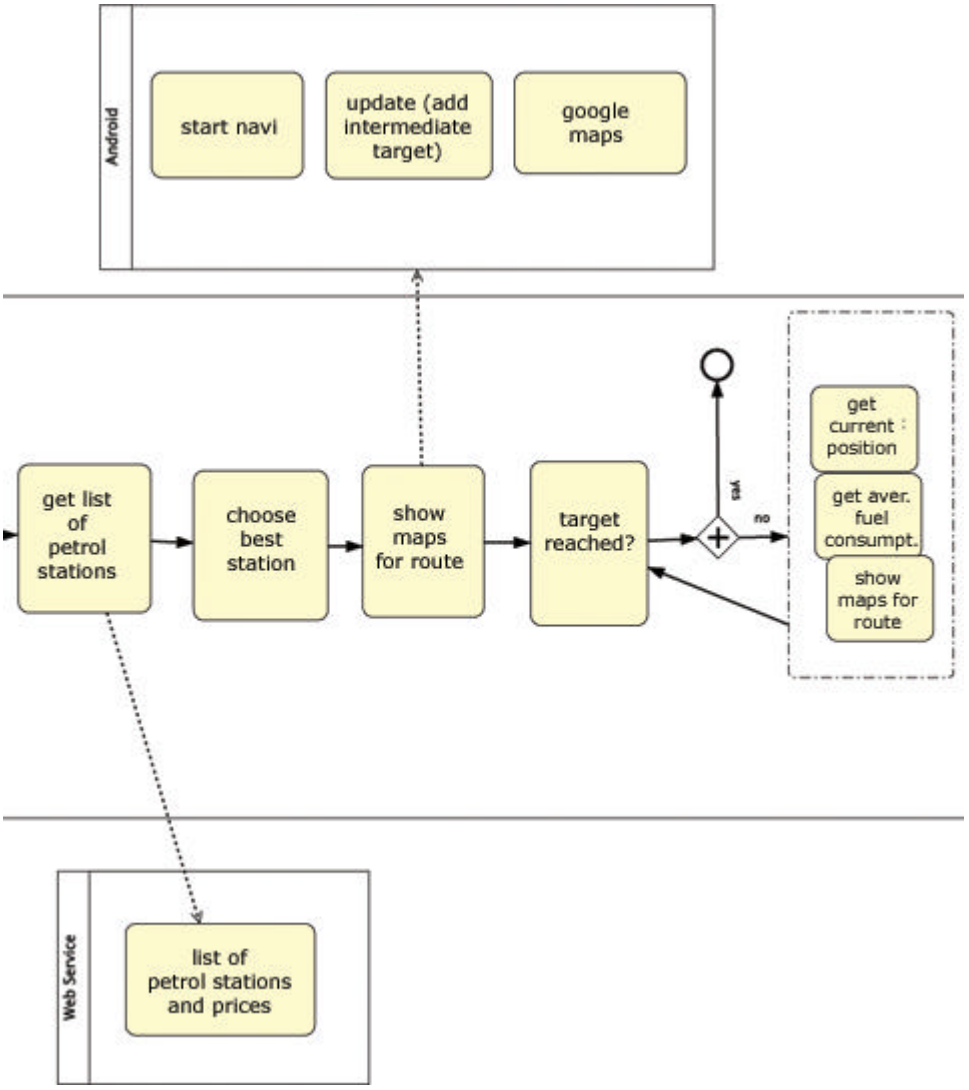

Abbildung 1-2 BPMN – Diagramm – Abschnitt 2

[8] Die Parallelität ging aus der theoretischen Überlegung hervor, dass voneinander unabhängige Dienste zeitsparend gleichzeitig ausgeführt werden können. Ob dies in diesem Fall die günstigste Lösung sei, sei dahingestellt.

Weiter mit Abb. 1-2: Im positiven Fall wird eine Liste der Tankstellen in der Umgebung (anhand der aktuellen PLZ) von einem Web Service angefragt und daraus wird die Günstigste ermittelt. Schließlich wird die Route mit dem Zwischenstopp auf dem Android angezeigt. Die Anzeige der Karten wird so lange aktualisiert, bis das Ziel erreicht worden ist (d.h. wenn die aktuellen Koordinaten gleich der Koordinaten des Ziels sind). Die Berechnung des Benzinverbrauches wird ebenso in der Schleife wiederholt, um auf einen stark verändertem Verbrauch entsprechend reagieren zu können.

Konkrete modellierung des Prozesses: der Workflow.

Im zweiten Schritt des Lebenszyklus unseres Prozesses sind die Schnittstellen zu den verschiedenen Diensten definiert (mittels Signatur, also Methodennamen und Parameterbeschreibung bzw. in Fall von Webdiensten mittels der *Web Service Description Language*: wsdl[9]) und bei DEMAC registriert worden.

Als Workflow Management System ist Demac das, was unsere Applikation zusammenhält: In der dpdl-Datei (*Demac Process Description Language*[10], in Anlehnung an XPDL), die beim Starten der Applikation an Demac übergeben wird, ist der gesamte Prozess beschrieben, und zwar in Form von *Activities*, *DataFields* (gemeinsame Variablen, auf die alle Activities zugreifen können) und *Transitionen*, die den zeitlichen Ablauf steuern und derer Ausführung mit Bedingungen versehen werden kann. Transitionen unterstützen auch Parallelität, was in diesem Prozess verwendet wurde.

Wie bei jeder XML-artigen Datei stellt sich bei der dpdl-Datei das Problem der Editierung: Über IDs sind die Elemente miteinander verknüpft, doch die Lesbarkeit einer solchen Datei ist teilweise schwierig. Ohne einen Editor, der auf Inkonsistenzen prüft, ist die Wahrscheinlichkeit von Flüchtigkeitsfehlern recht hoch.
Interessant waren auch die unerwarteten Auswirkungen der Benamung der IDs einiger Transitionen (IDs geben entgegen der Erwartungen die Reihenfolge an, in der Transitionen ausgeführt werden, eine Benamung wie folgt: Id="t1a" Id="t1b" führte zu einem Fehler in der Ausführung, wohingegen Id="t1" und Id="t2" einwandfrei funktionierte).
Die Tatsache, dass nur Strings als Parameter weitergegeben werden können hat einige Anpassungen der Schnittstellen erfordert.
Wenn man einmal ein Verständnis dafür entwickelt hat, bietet DPDL eine übersichtliche und solide Art, die Anwendung formal zu beschreiben.

---

[9] W3C Beschreibung zu WSDL: *http://www.w3.org/TR/wsdl20/*

[10] DPDL Dokumentation von der Uni Hamburg (Stand Sept 2009): *http://vsis-www.informatik.uni-hamburg.de/teaching/ss-09/mvg/dokumentation/materialien/doku_dpdl.pdf*

Im Fehlerfall ist die XML-Logdatei von Demac sehr hilfreich, weil man darin den aktuellen Stand der Variablen prüfen und somit schnell erschließen kann, welche Abschnitte ausgeführt wurden und mit welchen Werten.

Implementierung der Dienste

Schließlich sind die Dienste implementiert worden. Durch die Verwendung verschiedener Dienste ist unser Bord Computer in der Lage, viele Funktionalitäten zur Verfügung zu stellen, obwohl er nicht besonders leistungsstark ist, kein GPS besitzt und nur über ein kleines Display verfügt, das zur Anzeige einfacher Meldungen ausreicht, nicht aber von Bildern bzw. Kartenmaterial. Einige Dienste können komplexere Berechnungen ausführen, die Hautapplikation bekommt aber über Demac immer nur das „einfache“ Ergebnis.

1.1.2 Dienste für den Tankassistent

Eine Auswahl der Dienste, die Anhand der BPMN-Beschreibung identifiziert werden konnten, wird im diesem Abschnitt beschrieben.

Transformation Adresse ← → Geokoordinaten

Die Benutzereingabe des Ziels erfolgt in Form einer Adresse, oder des Namens eines besonderen Ortes bzw. einer Stadt. Für die Berechnungen (z.B. die Ermittlung der Entfernung vom Ziel) werden aber normierte Angaben in Form von geografischen Koordinaten benötigt.
Für solche aufwändigen Berechnungen bot es sich an, die Dienste von Google[11] zu verwenden, die zu einer Ortseingabe eine Reihe von Informationen liefern, darunter die Koordinaten. Unser Dienst (Activity: “Transform Address <-> GPS” in Abb. 1) erhält die erhaltene Benutzereingabe (String) und berechnet daraus ein Objekt vom Typ GoogleCoordinates. Daraus wird lediglich die Information der Koordinaten extrahiert und als String in dem benötigten Format („latitude, longitude“) an die Hauptanwendung zurückgegeben.

[11] Geocoding über Google Maps: *http://maps.google.com*

Berechnung der aktuellen Position

Es ist vorgesehen, dass der Autofahrer zusätzlich zum Bord Computer auch über ein Android Gerät verfügt. Dieser ist in der Lage, die aktuelle GPS-Position zu ermitteln und wird dazu von der Hauptapplikation verwendet.
Die Ermittlung der aktuellen GPS-Koordinaten (Activity: "GPS Position" in Abb. 1-1) erfolgt also in einer separaten Android Applikation, die mit der Hauptapplikation kommuniziert (Activity: "get current Position" in Abb. 1-1).
Auf Android steht für die Berechnung von lokalen Koordinaten die Klasse *android.location LocationManager* zur Verfügung, die u.a. Methoden bietet, verschiedene Providers (d.h. Dienstanbieter, die in der Lage sind, die Position eines Geräts zu ermitteln, in der Regel über Funknetze) durchzugehen und diese nach der aktuellen Position abzufragen.
Auf Android wird also ein Service gestartet, welcher eine Instanz von LocationManager mit dem aktuellen Kontext erzeugt.
Hier werden die Providers mit der Methode `requestLocationUpdates()`angefragt (s. Listing 1.1), der man als Parameter mitgeben kann, wie oft nach GPS-Daten gefragt werden soll (hier zum Beispiel: alle 60 Sekunden und nach 50 Metern Positionsänderung).
Das Ergebnis (ein *Location* Objekt) wird im Anschluss noch bearbeitet (Aufruf der Methode `makeStringLocation()`), um daraus nur die zwei Koordinaten zu entnehmen und in dem benötigten Format zu speichern. Der String mit den kommaseparierten Koordinaten ist das, was von dieser Android Applikation zurückgegeben wird.

```
public class LocationService extends Service {

 public String getCurrentLocation(Context ctx) {

  LocationManager manager = (LocationManager) ctx
                       .getSystemService(Context.LOCATION_SERVICE);
  Location result = null;
  Iterator<String> providerIterator = manager.getProviders(
                                  new Criteria(), true).iterator();
  while (providerIterator.hasNext())
  {
      String provider = providerIterator.next();
      manager.requestLocationUpdates(provider, 60000, 50, this);

      Location nextLoc = manager.getLastKnownLocation(provider);

      //it could be that a Provider returned null, then try with next one
      if (result == null) {
        result = nextLoc;
      }
  }
  return makeStringLocation(result);
 } }
```

Listing 1.1 Positionsanfrage über LocationManager von Android

Die Hauptschwierigkeit bestand bei diesem Dienst zum einen darin, dass zu den Location Diensten auf Android noch relativ wenig Dokumentation existiert[12], zum anderen beim Zusammenspiel zwischen Android und Hauptapplikation, also wie diese auf die Koordinaten zugreifen kann.
Zum Datenaustausch ist Socketkommunikation eingesetzt worden:
Die Applikation ruft über Demac an der benötigten Stelle eine Client-Methode auf, welche bei Android die GPS-Daten anfragt.
Android startet seinerseits mit dem GPS-Service einen Server und horcht dort auf einem bestimmten Port, bis eine Anfrage vom Client kommt.
Wenn Demac die ClientAnwendung aufruft, findet der Austausch der Daten über den `DataInputStream` statt (s. Listing 1.2) und der String mit den Koordinaten kann in der Applikation gespeichert werden.

```
private void sendToServer(){
    serversock = new ServerSocket(6666);
    while (true){
         Socket theclient = serversock.accept();
         DataOutputStream output = new DataOutputStream
                                        (theclient.getOutputStream());
         output.writeUTF(LocationService.resultCoordinates);
         output.flush();
         output.close();
         theclient.close();
       }
 }
```

Listing 1.2 Senden der GPS-Koordinaten über Socket Verbindung

Für die Verbindung mit dem Android Emulator muss auf dem Rechner, auf dem die Instanz des Emulators läuft, eine Port Redirection erstellt werden, um den verwendeten Port des Rechners auf dem verwendeten Port des Emulators zu mappen:

```
redir add <protocol>:<host-port(Rechner)>:<guest-port(Emulator)>
```

[12] Nicht alle Wege haben mit dem Android Emulator erfolgreich funktioniert, zum Beispiel die in der c't (vom 25.5.09) vorgeschlagene `manager.isProviderEnabled(LocationManager.GPS_PROVIDER)` und `manager.isProviderEnabled(LocationManager.NETWORK_PROVIDER)` haben keine Provider ermitteln können. Auch konnten dem Emulator mittels DDMS-Perspektive (S.Abschnitt 2.3.3) keine Koordinaten geschickt werden, dies funktioniert nur über die Telnet Konsole mit dem Befehl „geo fix <Koordinaten>".

Ermittlung der günstigsten Tankstelle

Die Activity „choose best station“ aus Abb. 1-2 basiert auf Ergebnissen der vorangehende Activity (das Suchen einer Tankstellenliste) und wählt die günstigste Tankstelle aus.
In der vorangehenden Activity wurde aus dem Parsen der Ergebnisse für eine bestimmte Postleitzahl aus einer Webseite[13] zunächst dynamisch einen String erzeugt, der eine Liste der dort gefundenen Tankstellen und der dazugehörigen Daten enthält. Dabei wird in den String gleich xml-Syntax geschrieben, beispielsweise:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
   <tankstellenliste>
      <tankstelle>
            <name>STAR</name>
            <strasse>Wöhlerstr. 4 </strasse>
            <plz>22113</plz><ort>Hamburg</ort>
            <preis>1.279 </preis>
     </tankstelle>
</tankstellenliste>
```

Dieser String kann nun mühelos unter Verwendung von `java.io.PrintWriter` in eine XML-Datei geschrieben werden. Die Codierung mit ISO-8859-1 ist dabei wichtig, weil sie auch die Umlaute korrekt übernimmt.
Die Liste der erhaltenen Tankstellen wird für den Benutzer angezeigt, das System hat allerdings noch die Aufgabe, daraus die Günstigste zu ermitteln und diese anschließend mitzuteilen sowie ihre Koordinaten als Zwischenstopp in die Route hinzuzufügen.
Dazu wird die XML-Datei untersucht[14], bis das Kindelement „Preis“ gefunden und mit allen anderen „Preis“-Elementen verglichen werden kann. Ist das Element mit dem niedrigsten Wert gefunden worden, so merkt sich die Applikation die zugehörige Adresse und schickt diese Angabe dem Dienst zur Transformation der Adresse in Geokoordinaten, da die Applikation für ihre Berechnungen immer Geokoordinaten benötigt.
Das Ergebnis davon liefert dieser Dienst am Ende zurück.
Demac schreibt das Ergebnis in das dafür vorgesehene Datafield für den Zwischenstopp der dpdl-Datei. Später wird der Routen- und Kartendienst diese Variable für die Route benötigen und sie von dort auslesen.
Der genaue Code ist im Anhang zu finden.

[13] Clever Tanken: *http://www2.clever-tanken.de*

[14] Unter Verwendung von Methoden aus *javax.xml.parsers* und aus *org.w3c.dom*

1.1.3 Prozessmanagement

Ist einmal die Applikation zu Ende implementiert und getestet, so kann man sich in einem letzten Schritt der Analyse, Überwachung und Wartung der Applikation widmen. Unter den verschiedenen Möglichkeiten des Prozessmanagements ist für den Tankassistenten das Eventmanagement mit *Esper*[15] eingesetzt worden.
Dadurch ist es möglich, an relevanten Stellen der Anwendung sogenannte „Events“ zu senden, die während der Laufzeit der Applikation ausgewertet werden können. Ein Beispiel ist der Event, der gesendet wird beim Auftreten von „Sonderfällen“, beispielhaft wenn keine Tankstelle in der Umgebung gefunden werden kann: In dem Fall kann das Event sofort ein anderes Verhalten der Applikation bewirken und vermeidet ein Fehlverhalten. Ein anderes Beispiel ist das Senden der aktuell ermittelten GPS-Positionen an die Applikation, welche damit ein Fahrtenbuch automatisch erstellt.
Events werden in Form von Nachrichten (serialisierbare Java Objekte) über das Netzwerk an Esper geschickt.

Im Fall der GPS-Position enthält der Event einen String mit den Koordinaten und wird nach dem Ermitteln jeder neuen Position erzeugt (als *Serializable*), mit der passenden set-Methode gefüllt und über den *ObjectOutputStream* verschickt.

In der Esper Engine sind Listeners registriert für die Events bzw. für das Auftreten bestimmter Muster unter den Events (in SQL-ähnlicher Syntax) :

```
Configuration config = new Configuration();
config.addEventTypeAutoAlias("de.hamburg.vsis.eventmanagement");

EPServiceProvider engine = EPServiceProviderManager
    .getDefaultProvider(config);

String pattern2 = "select * from Tankstelle where foundOne=false";
EPStatement s2 = engine.getEPAdministrator().createEPL(pattern2);
s2.addListener(new FuelstationListener());

String pattern3 = "select * from GPSPositionEvent";
EPStatement s3 = engine.getEPAdministrator().createEPL(pattern3);
s3.addListener(new GPSPositionListener());
```

Listing 1.3 Esper Patterns

[15] Esper ist eine Engine für Event Streaming Processing (ESP) und Complex Event Processing (CEP). Esper Homepage, *http://esper.codehaus.org/* Stand September 2009

In diesem Beispiel tritt pattern2 dann auf, wenn keine Tankstelle gefunden werden kann und pattern3 immer, da es sich um das Empfangen von GPS-Positionen für das Fahrtenbuch handelt und jede neu ermittelte Position aufgenommen werden muss.
Angenommen, ein GPSPositionEvent sei in der Esper Engine angekommen, nun tritt der dafür registrierte EventListener in Kraft, der GPSPositionListener.
Dieser führt eine Aktion aus. In diesem Fall wird der String mit den Koordinaten in eine Textdatei geschrieben. Dabei wird noch der Fall ausgeschlossen, dass sich genau dieselben Koordinaten als letzter Eintrag in der Textdatei schon befinden: in dem Fall hat sich der Fahrer seit dem letzten GPS-Aufruf nicht bewegt und diese Position soll nicht in das Fahrtenbuch erneut auftauchen.

1.1.4 Fazit aus dem Projekt

Das Projekt deckte viele sehr aktuelle Gebiete. Besonders interessant war die Arbeit mit der Software für den Workflow Management (Demac) und der Einsatz von Android in einem verteilten System.
Dabei fielen große Unterschiede zu der bisherigen Programmierung mittels J2ME auf.
Der Rest dieser Arbeit widmet sich dem Vergleich der Entwicklung für mobile Geräte unter Einsatz von JavaME und für Android, wobei es sich bei Android um ein sehr aktuelles, spannendes und noch wenig untersuchtes Gebiet handelt, das sich vom Anfang an ziemlich interessant und auch marktreif gezeigt hat.

2 JavaSE, JavaME und nun Android

Der Markt um die mobilen Geräte hat in den letzten 10 Jahren immer mehr an Bedeutung gewonnen und ist heutzutage ein hochaktuelles Gebiet. Computer stecken inzwischen in vielen Geräten (Waschmaschine, Bremsassistent im Auto, Telefon...) und teilweise ohne dass dies wahrgenommen wird. Mit Mobiltelefonen kann man eine Vielfalt von Funktionen benutzen, zusätzlich zum Telefonieren, was langsam Nebensache geworden ist; Mobiltelefone werden außerdem immer leistungsfähiger, da viel Forschung und Entwicklungsaufwand in diesen Bereich gesteckt werden.

Java hat sich als Programmiersprache für den mobilen Markt durchgesetzt.
1998 war Java nicht mehr nur eine Programmiersprache, sondern eine ganze Plattform. Auf diese Zeit, mit der Java 2 Plattform, gehen die Bezeichnungen J2EE (Enterprise Edition), J2SE (Standard Edition) und J2ME (Micro Edition) zurück[16].
Der Zusammenhang wird in Abbildung 2-1 verdeutlicht.

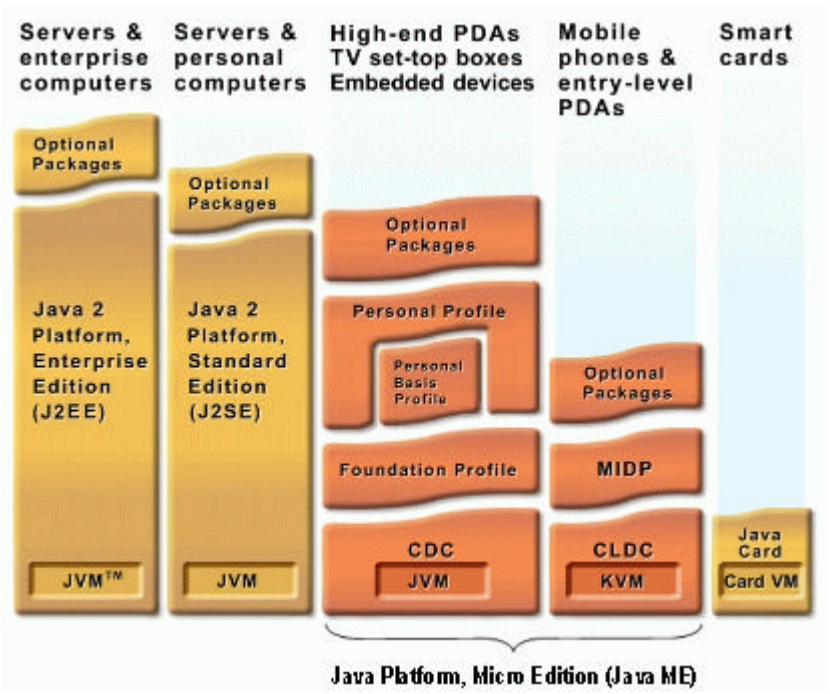

Abbildung 2-1 Java und JavaME Architektur (Quelle: Sun[17])

[16] S. u.A. [Ul09], Kap 1.5

[17] *http://java.sun.com/javame/img/javame_components.gif*

Die Standard Edition, für Client-Anwendungen gedacht, hat sich immer mehr in der Programmierwelt etabliert. Allerdings waren ihre Bibliotheken viel zu groß für den begrenzten Speicher von mobilen Geräten, auch wenn dieser zunehmend größer wurde[18]. Aus diesem Grund ist JavaME entstanden: Kurz gesagt ist JavaME eine reduzierte Version von Java für eingebettete Systeme (PDAs, Mobiltelefone..).

Eingebettete Systeme unterliegen nämlich immer noch Einschränkungen gegenüber Systemen für „normale“ Desktop-Computer: Da mobile Geräte meist klein, handlich und leicht sein müssen, ist ihr Speicher besonders begrenzt und der Bildschirm meist klein; wegen des Batteriebetriebs ist Energie nur eingeschränkt verfügbar; außerdem sind die Geräte nicht standortgebunden, was dazu führt, dass eine Verbindung zum Netzwerk nicht immer garantiert werden kann und dass die Kommunikation (z.B. mit einer Datenbank) die Beweglichkeit des Geräts berücksichtigen muss.
Aus diesen Gründen unterscheiden sich die Technologien für eingebettete Systeme und die für Desktop Computers stark voneinander und die Entstehung von JavaME war gerechtfertigt.

Inzwischen stößt die Micro Edition aber auch an ihre Grenzen: Mobile Geräte werden immer leistungsstärker und neue Bedürfnisse entstehen, dafür bietet J2ME zu wenig Möglichkeiten. Googles Android ist ein Beispiel dafür, wie man versucht hat, sich von diesen Standards zu entfernen bzw. Java an modernere Bedürfnisse anzupassen, egal ob dies eine technologisch bedingte oder eine marktstrategische Entscheidung gewesen ist[19].
Als Programmiersprache für mobile Geräte ist Android eine echte Alternative zu J2ME; als Framework ist Android starke Konkurrenz für die bisher auf den Standards von Sun basierenden Systemen.

Im Folgenden werden die Unterschiede zwischen Java SE, Java ME und Android unter die Lupe genommen.
Abschnitt 2.1 vergleicht hierzu die Architektur der Plattformen mit besonderem Augenmerk auf der jeweiligen Java Virtual Machine. Abschnitt 2.2 geht kontrastiv auf die unterstützten Sprachmerkmale der jeweiligen Java „Versionen“ und auf die Einschränkungen ein, die in der Programmiersprache von J2ME und Android vorgenommen worden sind. In Abschnitt 2.3 werden Werkzeuge für de Entwicklung vorgestellt (insbesondere die zur Verfügung stehenden Emulatoren für die Desktop-Entwicklung) und auch die Struktur eines J2ME- bzw. Android-Projektes erläutert.

[18] Nach G. Moore (1965) lässt sich die Anzahl der Transistoren auf einem Chip alle 1,5 Jahre verdoppeln, also wächst die Speicherdichte - auch bei mobilen Geräten - exponentiell.

[19] Artikel aus der Computerwoche: „Android: Google umgeht Suns virtuelle Java-Maschine“: *http://www.computerwoche.de/nachrichtenarchiv/1848313/* Stand September 2009.

Schließlich beschreibt Abschnitt 2.4 Konzept, Aufbau und Lebenszyklus der Anwendungen, welche aus Bausteinen (sogenannten Midlets bzw. Activities) zusammengesetzt sind.

2.1 Architektur der Plattformen

Die JavaSE Laufzeitumgebung (*Java Runtime Environment*, JRE) besteht bekanntlich aus einer *Java Virtual Machine* (JVM) und einer Reihe von Klassenbibliotheken. Es gibt JREs für verschiedene Betriebssysteme und dadurch ist gewährleistet, dass Java Anwendungen mit den Java Standard Bibliotheken auf allen Computern lauffähig sind, in denen eine JRE vorhanden ist.
Mobile Geräte sind – zumindest in der Entstehungszeit der Java Micro Edition – nicht in der Lage, eine JRE zu unterstützen wie für die Java Standard Edition, da diese zu viele Ressourcen verbraucht. Zusätzlich sind mobile Geräte unterschiedlich leistungsstark, so dass es schwierig ist, allgemeingültige Standards dafür festzulegen, folglich wurden sie in „Klassen“ gruppiert, für die jeweils Konfigurationen definiert wurden.
Vorwiegend haben sich zwei Klassen herauskristallisiert: schwache und weniger schwache mobile Geräte.

2.1.1 Konfigurationen und Profile in JavaME

JavaME basiert auf diesen Konfigurationen (die eine JVM und einen minimalen Satz von Klassenbibliotheken enthalten) und auf Profilen, welche die Konfiguration ergänzen.
Dazu kommen, je nach Gerät, verschiedene Packages, also optionale APIs für weitere Funktionalitäten.
Jede Konfiguration und jedes Profil basiert auf einer oder mehreren *Java Specification Request* (JSR), in denen die Anforderungen und die Schnittstellen für eine Java Erweiterung enthalten sind[20].

Es gibt zwei Konfigurationen für JavaME[21], dessen Zusammenhang in Abb. 2-2 verdeutlicht wird:

[20] JSRs wurden 1998 von Sun als Vorgehensweise für die Weiterentwicklung der Sprache Java eingeführt. Auf der Webseite des Java Community Process kann die Liste aller JSRs eingesehen werden: Java Community Process, *http://www.jcp.org/en/home/index*. Stand: September 2009.

[21] S. u.a. [Ri08], Kap. 1

- die *Connected Device Configuration* (CDC) für leistungsstärkere mobile Endgeräte (z.B. Set-Top Boxen, Smart Phones).
- und die *Connected Limited Device Configuration* (CLDC) für Geräte mit beschränkten Ressourcen, womit wir uns hier beschäftigen.

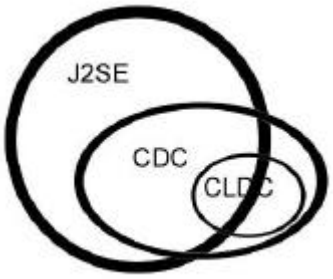

Abbildung 2-2 CDC vs CLDC

Die **CDC** enthält eine JVM und die minimal benötigten API Klassen. Für Applikationen wird zusätzlich ein Profil benötigt: Das *Foundation Profile*, welches die übrigen Klassen und APIs enthält. Weitere Profile können noch bei Bedarf eingebunden werden (zum Beispiel das *Personal Profile*, welches das vollständige Java AWT enthält, womit die Gestaltung von grafischen Benutzungsschnittstellen möglich ist). Auf CDC wird in dieser Studie nicht weiter eingegangen.

Die **CLDC**, zuerst in der minimalen Version 1.0 entstanden, ist durch die Version 1.1 um viele Funktionalitäten erweitert worden, da die Geräte leistungsstärker geworden sind.
Sie enthält eine JVM, einige Kernbibliotheken (u.a. *java.lang*, *java.util*), Ein-/Ausgabe, Sicherheit[22]. Diese Untermenge aus JavaSE ist aber noch nicht ausreichen, um Anwendungen zu entwickeln, deswegen sind zusätzlich Profile notwendig.

Auf die Konfiguration aufbauend, erweitern die Profile das System um andere für die Entwicklung erforderlichen Funktionen, zum Beispiel zur Steuerung des Lebenszyklus einer Anwendung. Das bekannteste Profil für CLDC ist **MIDP** (*Mobile Information Device Profile*). Dieses erlaubt, MIDlets zu entwickeln und definiert u.a. Schnittstellen für:

Lebenszyklus von Applikationen, Benutzer Interface (Display, Tastatur)
Event Handling, Verwaltung persistenter Daten
Netzwerk und Protokolle, Sound, Timer

[22] Im Bereich Sicherheit wird durch CLDC zweierlei gewährleistet: Die Verifikation des Codes durch die Virtual Machine (s. Abschn. 2.1.3) und das „Sandkastenprinzip“, nach dem jede Applikation in einer geschlossenen Umgebung ausgeführt wird.

2.1.2 Die Android Plattform

Android hat einen Linux Kernel und native Bibliotheken (geschrieben in C und C++), die Grundfunktionalitäten bieten (verantwortlich u.a. für die grafischen Darstellungen auf dem Display, Fenstermanagement, Video-Codecs)[23]. Vgl. Abb 2-3.

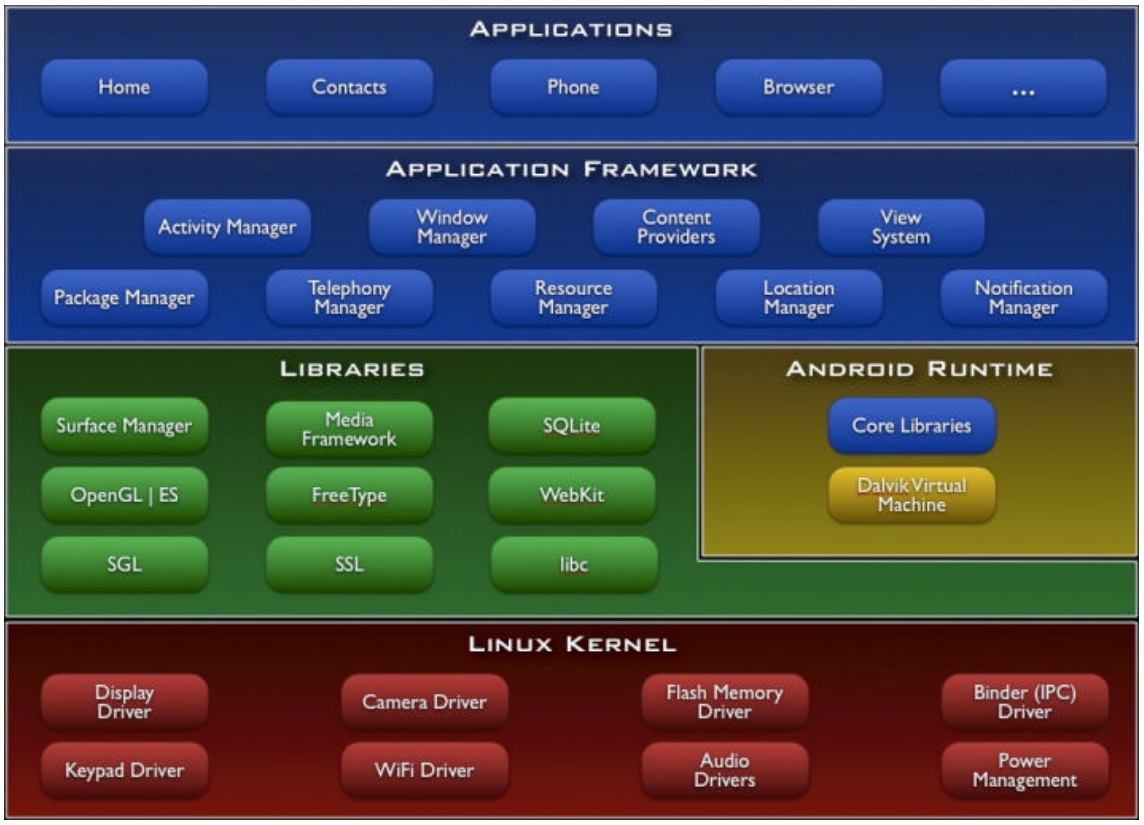

Abbildung 2-3 Android Architektur (Quelle: Android developers[24])

Darauf basierend hat Android eine eigene Runtime, mit eigener Virtual Machine (Dalvik) und Java „Core Libraries“, die verschiedene Utilities und Tools enthalten. Diese Bibliotheken stellen eine Auswahl der Java Funktionalitäten zur Verfügung, analog zur Auswahl in der Konfiguration von JavaME, sie unterscheiden sich aber davon, weil die Entwickler hier vom aktuellen Stand von JavaSE ausgegangen sind und eine neue Auswahl getroffen haben, die im Gegensatz zu JavaME fast alle Java Features enthält.
Eine Ebene höher befindet sich das „Application Framework“, eine Reihe von APIs, in Java geschrieben, die von allen Applikationen verwendet werden.

---

[23] s. Android developers, Androidology – Architectural Overview (Video), Stand Sept. 2009: *http://developer.android.com/intl/en/videos/index.html#v=QBGfUs9mQYY*

[24] *http://developer.android.com/guide/basics/what-is-android.html*

Darunter: Der *Activity Manager* ist für den Lebenszyklus der Applikationen verantwortlich (hier Activities, als Gegenstück zu MIDlets), *Content Provider* erlauben den Zugriff auf persistente Daten und regeln den Datenaustausch zwischen Applikationen, das *View System* regelt eine Reihe von Layoutmöglichkeiten (auch eine Map View und eine Browser View, um Karten und Webinhalte direkt in Applikationen einzubauen), der *Location Manager* liefert geographische Informationen für positionsbasierte Dienste.
Ein sehr kompaktes, in sich abgeschlossenes und auch spezielles System.

Die Architektur von Android ist auf der einen Seite analog zur üblichen Architektur für mobile Telefone und kann mit J2ME verglichen werden, wie in Abb 2.4. zu sehen.

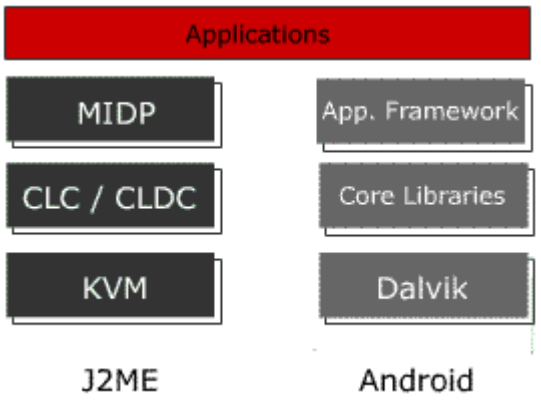

Abbildung 2-4 Strukturvergleich J2ME / Android

Abgesehen von dieser Ähnlichkeit in der Architektur ist die Umsetzung in Android andererseits komplett verschieden, angefangen vom Betriebssystem und der JVM bis hin zur Struktur der darauf basierenden Applikationen und zu den verfügbaren Java-Sprachelementen. Mit Android erfolgte also eine Entfernung von den Sun Standards, welche durch J2ME von den meisten mobilen Geräten weltweit eingesetzt werden, und ein eigenes Konzept ist entstanden.
Noch mehr: Da mobile Geräte - im Gegensatz zu PCs - sich bezüglich Hardware, Betriebssystem und Ressourcen besonders stark voneinander unterscheiden und da J2ME auf das Betriebssystem aufsetzt, hat J2ME mit diesen Unterscheiden zu kämpfen, vor allem mit Kompatibilität und Portabilität.
Android bildet ein eigenes System und bleibt von solchen Problematiken verschont, der Schwerpunkt wird auf die optimale Funktionalität innerhalb dieses Systems gesetzt.

2.1.3 JVM (und KVM) vs. Dalvik

Die ***Java Virtual Machine*** (JVM), Bestandteil der Java Laufzeitumgebung, stellt eine Zwischenschicht zwischen Anwendung und Betriebssystem / Hardware dar und macht Java plattformunabhängig.
Java Programme (.java Dateien) werden vom Compiler in .class Dateien umgewandelt, welche den von der *JVM* ausführbaren Bytecode enthalten. Die JVM ist fürs Verifizieren und Ausführen dieses Bytecodes zuständig, wobei die Ausführung eine Übersetzung in den Maschinencode der jeweiligen Plattform bedeutet. Der Java Compiler erzeugt denselben Bytecode und je nach Betriebssystem existiert eine entsprechende JVM, die in der Lage ist, das Java-Programm auf einer bestimmten Maschine auszuführen.
Java Bytecodes sind also Anweisungen für die JVM, jeweils der Länge von 1 Byte (es kann also höchstens 256 davon geben), sogenannte *opcodes*. Durch diese Anweisungen kann die stack-basierte JVM die entsprechenden Operationen durchführen.
Einige opcodes packen lokale Variablen auf den Stack (`iload`, `lload`, `fload`, and `dload` je nach Datentyp der Variable) bzw. entfernen sie und speichern den Wert in eine Variable zurück (`store`); andere Befehle konvertieren primitive Datentypen untereinander (Beispiel: `i2l` konvertiert von *int* zu *long*). Weitere opcodes führen arithmetische Operationen aus (zum Beispiel `iadd, ladd, fadd, dadd` führen für die jeweiligen Datentypen Additionen mit den 2 obersten Operanden auf dem Stack aus; die Datentypen der Operanden müssen übereinstimmen)[25].

Die **Kilobyte Virtual Machine** (KVM)[26] ist eine für kleine Geräte entworfene JVM, die für knappen Speicher (Kilobytes statt Megabytes, wie der Name sagt) ausgelegt ist, sie funktioniert aber genau nach demselben Prinzip.
Zu den wichtigsten Unterschieden zählt das abweichende Verhalten des *Bytecode Verifier*. Dieser hat in der Java Virtual Machine die Aufgabe, den auszuführenden Code direkt nach dem Laden zu prüfen und sicherzustellen, dass es u.a. keine illegalen Instruktionen oder Adressierungen enthält. Durch diesen Mechanismus will man die Maschine, auf der das Java Programm läuft, vor Angriffen schützen.
Für die Java Micro Edition ist dieser Vorgang allerdings ungeeignet, weil dadurch zur Laufzeit zu viel Speicher in Anspruch genommen wird. Aus diesem Grund ist der Vorgang in zwei Phasen aufgeteilt worden: Der Bytecode wird schon nach dem Kompilieren (also vor dem Erstellen der jar-Datei) auf dem Entwicklungssystem "pre-verifiziert". Danach werden die verifizierten .class Dateien entsprechend mit sogenannten Stack-Map-

[25] s. dazu: Sun, The Java Virtual Machine Specification
http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html
[26] s. dazu: Sun, The K virtual machine, White Paper, *http://java.sun.com/products/cldc/wp/*

Attributen markiert, die in der zweiten Phase, zur Laufzeit, nur noch überprüft werden müssen.
Dieses Verfahren kann eine Sicherheitslücke darstellen: Gelingt die Manipulation dieser Maps, lässt sich die Überwachung ausschalten[27].
Dafür bietet die KVM keine Möglichkeit, den Classloader zu erweitern oder zu ersetzen und auch keine JNI-Unterstützung (s. dazu Abschnitt 2.2.5), was garantiert, dass das Austauschen von Sicherheitspaketen zur Laufzeit nicht möglich ist.

Für Android ist eine weitere eigene Virtual Machine entworfen worden: **Dalvik** (der außergewöhnliche Name ist der Name eines isländischen Dorfes, aus dem einer der Entwickler stammt). Android Applikationen können in normalem Java geschrieben und kompiliert werden. Das im Android SDK enthaltene „dx“ Tool konvertiert dann den entstandenen Java Bytecode in ausführbaren Code für die Dalvik Virtual Machine: Das bedeutet, .class Dateien werden für Android erneut in .dex Dateien konvertiert.
Auf dem Gerät, auf dem die Applikation laufen soll, muss sich die Dalvik VM befinden (und da Android auf Linux basiert, ist auch nur eine Implementation von Dalvik vorhanden bzw. ausreichend).
Dadurch entfällt komplett die Verwendung einer Java Virtual Machine auf dem Gerät und Android löst sich von Java. Die Entwicklung erfolgt mit JavaSE, aber das Endergebnis ist kein Java Bytecode. Dies hat eine ziemlich große Debatte ausgelöst, weil die Open Set Alliance dadurch die Lizenz von Sun, sowie den Java Community Process umgeht, gleichzeitig aber starke Konkurrenz auf ähnlichem Niveau ausübt[28].
Abgesehen von den Befehlen hat Dalvik auch eine andere Architektur, da es sich hier nicht um eine Stapelmaschine sondern um eine Registermaschine handelt. Der Unterschied in der Komplexität liegt darin, dass Registermaschinen zwar längere Befehle benötigen, da diese auch die Adressen der Register enthalten müssen, in der Regel aber mit weniger Befehlen auskommen (weil sie nicht immer Daten aus und von dem Stapel bewegen müssen), folglich benötigt Dalvik weniger Schritte, um den Bytecode zu verarbeiten.
Ansonsten ist Dalvik im Prinzip - genauso wie die KVM - eine schmalere Virtual Machine für die Bedürfnisse leistungsschwächerer Geräte. Eins der Merkmale, die hier (genauso wie bei der KVM) zwecks Reduzierung des Ressourcenbedarfs weggelassen wurden, ist der *Just-InTime-Compiler* (JIT), welcher bei Java der Performanceoptimierung zur Laufzeit dient: Der JIT übersetzt Bytecode-Stücke während der Ausführung in Maschinencode, so dass dieser schneller ausgeführt werden kann; er ist nicht unwichtig aber in diesem Fall entbehrlich.

[27] s. u.A. [Ri08], Kap 1 (S. 11)

[28] Ein interessanter und sehr „direkter“ Beitrag zu diesem Thema ist hier zu lesen:
Stefano Mazzocchi, Dalvik: how Google routed around Sun’s IP-based licensing restrictions on Java ME, *http://www.betaversion.org/~stefano/linotype/news/110/* , Stand: September 2009.

Kompatibilität

Aus dem, was gerade gesagt wurde, ergeben sich natürlich Kompatibilitätsprobleme zwischen den hier besprochenen Plattformen, JavaSE, JavaME und Android.

Java ist bekanntlich eine gut portierbare Programmiersprache durch die Java Virtual Machine.
Da in der KVM viele Einschränkungen vorhanden sind, kann sie nicht unbedingt den Bytecode verarbeiten, den ein Standard Java-Compiler erzeugt hat. Viele Schlüsselwörter von Java sind nicht gültig und können bei der Entwicklung von Anwendungen nicht verwendet werden. Bestehende Java Anwendungen können nicht ohne weiteres auf ein J2ME-System portiert werden.

J2ME Applikationen können überall laufen, wo eine JVM vorhanden ist und diese CLDC/CDC und MIDP unterstützt, d.h. fast auf allen gängigen mobilen Endgeräten. Auf Android können J2ME Anwendungen allerdings nicht laufen, da Android eine andere Virtual Machine verwendet. Es entstehen jedoch immer mehr Tools, um J2ME Programme in „Android-Java“ zu konvertieren, zum Beispiel der Micro Emulator[29] oder J2Me Polish[30] (unterstützt dieses Konvertierungsfeature seit der Version 2.1, angekündigt im Februar 2009). Außerdem werden zurzeit einige Libraries entwickelt, die J2ME-Code ohne Veränderung auf Android lauffähig machen: dies geschieht durch eine Reihe von „Adapters“, die den Originalcode für Android „verständlich“ machen[31].

Android Programme, da in Java geschrieben, können – wenn sie mit einem Java Compiler kompiliert werden – auf jeder beliebigen Java Plattform laufen. Weil Android Applikationen auf mobilen Geräten laufen sollen, ist diese JavaSE-Kompatibilität allerdings wenig von Relevanz. Wichtiger wäre, dass Android Applikationen auf Geräten laufen, die bisher in der Regel JavaME unterstützen. Auf solche Geräte sind Android Applikationen nicht portierbar, da Android sich sehr von JavaME unterscheidet und da Android-Java viel mächtiger ist.
Die Richtung *Android* → *J2ME* ist also deutlich schwieriger als andersherum und hier setzen möglicherweise Kritiker an: Android bildet ein eigenes System und hat kaum Kontaktpunkte mit bestehenden Systemen.

[29] MicroEmulator: *http://www.microemu.org/.*

[30] J2MEPolish: *http://www.enough.de/*

[31] Assembla, J2ME Android Bridge: *http://www.assembla.com/wiki/show/j2ab*

2.2 Unterstützte Features der Programmiersprache

JavaSE hat eine lange Evolution durchgemacht und enthält inzwischen sehr viele Features. Die Java 2 Standard Edition besteht in der Version 5 aus 3279 Klassen in 166 Packages und in der Version 6 aus 203 Packages.
Zur Zeit der Version 1.2, woraus J2ME entstanden ist, waren es weniger als die Hälfte.
Um mit wenig Speicher auszukommen (und auch weil nicht alle diese Features für die Welt der mobilen Geräten von Relevanz sind), ist in JavaME eine Auswahl davon getroffen worden, aber es sind auch einige Bibliotheken dafür neu entworfen worden.
Die CLDC ist der kleinste gemeinsame Nenner, der für eine große Anzahl von Geräten gelten soll. Dadurch ergeben sich Einschränkungen: Einige Klassen implementieren eventuell nicht alle Methoden, die man in derselben Klasse bei JavaSE findet, damit sie auf jedem Gerät lauffähig sind. Dazu kommen neue, JavaME-spezifische Klassen.
Je nach Gerät kann diese Basisausrüstung dann durch die Profile und durch gerätespezifische Erweiterungen ergänzt werden.
Android basiert auf einem ähnlichen Prinzip, es enthält nämlich nur eine Auswahl der Java Sprachelemente plus eigene Pakete. Bei Android ist die Auswahl aber deutlich größer und komfortabler.

2.2.1 Klassen und Packages in JavaME

Wenn man in die JavaME Dokumentation schaut[32], so fällt auf, dass aus den vielen Packages der Standard Edition nur *java.lang*, *java.util* und *java.io* übrig geblieben sind, wobei diese sich inhaltlich von den Originalen zusätzlich unterscheiden und nur eine Untermenge der Klassen und Packages enthalten, um diese an die Bedingungen mobiler Geräte anzupassen; zusätzlich enthält JavaME mehrere spezifische Packages unter *java.microedition.**, die in JavaSE nicht zu finden sind.

Zusammen kommen CLDC 1.1 und MIDP 2.0 auf 12 Packages, wobei MIDP vier der fünf Packages von CLDC um einige Klassen erweitert (zum Beispiel enthält *java.util* unter MIDP dieselben Klassen wie *java.util* für CLDC 1.1 plus zwei Klassen aus der Java Standard Edition: Timer und TimerTask).

Hier ein Überblick:

[32] Sun Inc, Java ME Technology: APIs and Docs *http://java.sun.com/javame/reference/apis.jsp*

	Specification	Packages		Klassen und IF
CLDC 1.0	JSR 30	4	java.io java.lang java.util javax.microedition.io	76
CLDC 1.1	JSR 139	5	java.io java.lang java.lang.ref java.util javax.microedition.io	81
MIDP 2.0	JSR 118	11	java.io java.lang java.util javax.microedition.io javax.microedition.lcdui javax.microedition.lcdui.game javax.microedition.media javax.microedition.media.control javax.microedition.midlet javax.microedition.pki javax.microedition.rms	145

Unterschiede in java.lang

Zu den wichtigsten Unterschiede in *java.lang* (dem Package mit den grundlegenden Sprachelementen) zählen die Folgenden:

j*ava.lang.math* Fließkommaunterstützung:
In CLDC 1.0 waren keine Fließkommaoperationen möglich, d.h. man konnte die Datentypen *Float* und *Double* nicht verwenden. Der Grund war zum einen das Speicherproblem und zum anderen die Überlegung, dass die Hardware der mobilen Geräte damals ohnehin keine Fließkommaunterstützung bot. Gleichzeitig stellte dies eine große Einschränkung der Programmiermöglichkeiten dar, da solche (manchmal unentbehrlichen) Operationen per Hand nachgebaut werden mussten. Folglich wurde die Möglichkeit der Verwendung von Fließkommaoperationen in CLDC 1.1 wieder eingeführt. Dazu wurde aber der mindesterforderliche Speicher von 160 auf 192 KB erhöht.

Java.lang.ThreadGroup nicht vorhanden:
Die Parallelverarbeitung von Prozessen mittels Threads ist auch in JavaME möglich. Thread Groups werden allerdings nicht unterstützt, d.h. Threads können nur individuell gestartet oder angehalten werden.

Java.lang.Exception enthält viel weniger:
Die Fehlerbehandlung ist eingeschränkt, da nur wenige Fehlerklassen vorhanden sind. Das hat damit zu tun, dass eine ausführliche Fehlerbehandlung zu ressourcenintensiv ist. Auch ist die Fehlerbehandlung stark geräteabhängig und die einfachste Lösung in diesem Fall erscheint eher ein Neustart des Systems zu sein.

Java.lang.Object.finalize() gibt es nicht:
In den CLDC Bibliotheken existiert die Methode *Object.finalize()* nicht, die normalerweise vom Garbage Collector aufgerufen wird, um ein Objekt aus dem Speicher zu entfernen. Die Ressourcen-Freigabe muss also vom Programmierer sichergestellt werden.

2.2.2 Klassen und Packages auf Android

Android enthält mehr: Insgesamt 128 packages, wobei 40 davon Android-spezifisch (*android.**) und 47 aus JavaSE übernommen sind (mehr oder weniger stark reduziert).

Die Java Packages stammen aus JavaSE sowie auch aus *apache*, *httpclient*, *junit*..
Darunter:

java.awt.font
java.beans
java.io
java.lang
java.math
java.net
java.nio
java.security
java.sql
java.text
java.util
javax.crypto
javax.xml

Die Android-spezifischen Packages betreffen Struktur und Lebenszyklus der Anwendungen und das eigene User Interface (nichts aus Swing und so gut wie nichts aus AWT ist übernommen worden), aber auch APIs für Telefonfunktionen, Batteriebetrieb und SMS-Funktionen.

Android enthält also viel mehr „Standard-Java“ als J2ME und außerdem besonders viele Funktionen, die speziell für Android entwickelt wurden; diese bieten eine große Zahl vorgefertigter Bausteine, die die Entwicklung stark erleichtern und die sich stark von J2ME sowie auch von Standard-Java unterscheiden.

2.2.3 Neuere Java Sprachelemente

Vor allem in der Version Java 5 (aus dem Jahr 2004, Nachfolger von Java 1.4) wurden viele neue Sprachelemente und Merkmale hinzugefügt. Dadurch hat sich die Plattform stark verändert: Das war mehr als eine weitere Version, auch deswegen die Bezeichnung „Java 5“.
Diese neueren Sprachelemente fehlen in J2ME, da diese auf der API von Java 1.2 basiert.
Android dagegen ist ein jüngeres Projekt, dessen API nicht auf einer bestimmten vorherigen Java Version festgelegt ist, sondern vielmehr eine Auswahl der „wichtigsten“ Java Elemente enthält, unter anderem auch einige neuere.
Da die Einschränkung von J2ME auf Java 1.2 ein Problem darstellt, sind in den letzten Jahren verschiedene Lösungen erschienen, um einige dieser Lücken zu füllen. Darunter J2ME Polish[33], eine Reihe von Tools, die MIDP um einige fehlende Sprachelemente erweitern.
Je mehr Standard-Sprachelemente unterstützt werden, desto leichter ist zum einen die Arbeit der Programmierer, zum anderen auch die Portierung von Java-Anwendungen auf mobile Geräte.

Generics

Generics wurden 2004 in Java 2 hinzugefügt und erlauben es, Klassen, Methoden und Interfaces mit Typparametern zu parametrisieren.
Dadurch ergibt sich u.a. die Möglichkeit, Typfehler beim Compilieren und nicht erst zur Laufzeit zu erkennen.
Ein Beispiel ist folgende Deklaration:

```
List<String> myStrList = new ArrayList<String>();
```

[33] J2ME Polish, *http://www.j2mepolish.org/cms/leftsection/documentation.html* , Stand: September 09

Hier wird eine Liste mit dem Typ String parametrisiert; wenn der Programmierer nun *myStrList* mit einem anderen Datentyp füllt, würde der Compiler einen Fehler melden; außerdem erübrigt sich dadurch der Cast der Listelemente auf (String), der notwendig ist, wenn Methoden der Klasse Iterator (beispielweise *hasNext()*) auf die Liste angewandt werden sollen. Diese Vorgehensweise ist robuster, leichter zu schreiben und besser lesbar.
Generics fehlen in CLDC / MIDP. Ergänzende Tools wie J2ME Polish bieten allerdings die Möglichkeit, Generics in J2ME Anwendungen zu verwenden.
In der Android API wurden Generics übernommen und können also verwendet werden.

Annotations

Annotationen erlauben die Einbindung von Metadaten in den Quelltext. Java 5 stellt sieben vordefinierte Annotationstypen zur Verfügung (in java.lang: *@Deprecated*, *@Override*, *@SuppressWarnings* und in java.lang.annotation: *@Documented*, *@Inherited*, *@Retention*, *@Target*), weitere können vom Programmierer selbst definiert werden.
Annotationen sind ein nützliches Mittel, um aus dem Quellcode beispielsweise Beschreibungsdateien automatisch zu erzeugen.
J2ME unterstützt Annotationen nicht.
In Android dagegen können Annotationen verwendet werden.

Besondere Konstrukte

Seit Java 5 gibt es die erweiterte For-Schleife (*enhanced for loop*), um mit einer wesentlich vereinfachten Syntax über Collections iterieren zu können:

```
for(String s: Liste){
        System.out.println(s);
    }
```

Solche erweiterte For-Schleifen können in J2ME nicht verwendet werden. In Android sind diese Schleifen vorhanden, in dem Developer Guide, im Abschnitt „Designing for Performance“[34] liest man allerdings: „Use enhanced for loop syntax with caution“ und es wird davon abgeraten, diese mit ArrayList zu verwenden.
Auf mobilen Geräten spielen Speicheroptimierungen einfach eine größere Rolle als sonst.

[34] Android Developer Guide: *http://developer.android.com/guide/practices/design/performance.html*

2.2.4 Reflection

Der Java Reflection Mechanismus, also die Abfrage von Typinformationen zur Laufzeit, steht im J2ME nicht zur Verfügung. Das bedeutet, dass darauf aufbauende Mechanismen wie JavaBeans, Remote Method Invocation (RMI) oder das Serialisieren von Objekten nicht möglich sind.

Android unterstützt Reflection. Das Package *java.lang.reflect* ist in der API vorhanden, unter anderem mit der Methode: *getMethod(String name, Class... parameterTypes)*.

2.2.5 Java Native Interface

In JavaSE ist es möglich bzw. manchmal notwendig, auf Funktionen zuzugreifen, die nicht in Java implementiert sind. Dies ist zum Beispiel dann der Fall, wenn Java auf die tatsächlichen Ressourcen eines Rechners zugreifen muss, beispielhaft für die Verwendung grafischer Benutzeroberflächen des Betriebssystems. Für solche sogenannten nativen Funktionen ist die Architektur des tatsächlichen Systems zuständig und Java greift darauf zu.
Allgemeiner benötigt man die Verwendung von nativen Methoden in Java, wenn Bibliotheken eingebunden werden sollen, welche nicht in Java geschrieben sind (zum Beispiel wenn geschwindigkeitskritische Bereiche nicht in Java implementiert werden oder gekaufte .dlls von der Applikation verwendet werden müssen).
Die Schnittstelle zwischen Java und der konkreten Plattform ist das JNI (Java Native Interface).
Es können auch eigene native Methoden definiert und mit `System.loadLibrary` geladen werden, so dass sie von der JVM aus aufgerufen werden können.

In JavaME (CLDC und MIDP) fehlt das JNI absichtlich, da Handy-Hersteller Manipulationen durch direkten Zugriff auf die Geräte befürchten. Die Menge der verfügbaren nativen Funktionen ist also abgeschlossen und das bedeutet u.a., dass Bibliotheken, die auf gerätespezifische LowLevel-Funktionalitäten zugreifen, vom Hersteller des Geräts selbst bereitgestellt werden müssen.
Ein Problem dabei ist allerdings, dass der beschränkte Zugriff auf die Systemressourcen deutlich als Grenze empfunden wird, insbesondere wenn es um den Zugriff auf das File System für die Datenspeicherung geht. Viele Entwickler würden es deshalb begrüssen, wenn JNI in JavaME implementiert wäre.

Im Mai 2007 wurde von Nokia eine Umfrage zum Thema „Would you need JNI for JavaME?“ durchgeführt und viele Entwickler äußerten sich entschieden dafür[35].

Auch in Android-Java fehlte anfangs die Unterstützung von JNI für Applikationen. Nur das Framework hatte selbstverständlich die Möglichkeit, auf native Funktionen zuzugreifen, da Android native Bibliotheken (in C / C++) besitzt und verwendet. Für die Entwickler blieb aber die Möglichkeit aus, selber in ihren Applikationen native APIs zu verwenden oder welche zu schreiben. Ende 2008 wurde dies noch in verschiedenen Entwicklerforen bemängelt[36].
Die Möglichkeit, nativen Code zu schreiben und in den Applikationen zu verwenden, kam mit dem Release des NDK (*Native Development Tool*), offiziell angekündigt im Juni 2009, also hochaktuell.
Durch das NDK sind Entwickler in der Lage, selber C und C++ Sourcen zu schreiben und daraus native Bibliotheken zu erstellen, außerdem native Bibliotheken in den Application Packages Files (.apks) zu „builden“, welche dann auf Android Geräte installiert werden können.
Von der Verwendung von NDK wird trotzdem auch teilweise abgeraten, außer dies wird tatsächlich benötigt, denn die Verwendung nativer Funktionen kann sich im Allgemeinen negativ auswirken, beispielhaft auf die Portabilität der Anwendungen oder auf deren Performance. Die Möglichkeit besteht dennoch.
Gegeben, dass der Zugriff auf Systemressourcen und native Funktionen bei mobilen Geräten ein problematisches Thema bleibt im Vergleich zu sonstigen Java Anwendungen, stellt die Umsetzung von JNI einen relevanten Unterschied zwischen Android und J2ME dar. In J2ME spielen nämlich die Sicherheitsmaßnahmen eine große Rolle: Wird eine Applikation nicht von Sun signiert, so fragt das Gerät bei jedem Versuch, auf Bluetooth, GPS, Kamera oder Mikrofon zuzugreifen mehrmals beim User nach mit „Wollen Sie diese Verbindung zulassen“ bzw. „Darf das Programm auf die Kamera zugreifen?“, was nicht immer wünschenswert ist. Ein Gleichgewicht zwischen Sicherheit und Usability müsste angestrebt werden.
Bei Android ist es anders; dort ist es sogar möglich, die Funktionalität von Systemkomponentenn durch eigene Programme zu ersetzen. Dies bietet deutlich mehr Freiheit für die Entwicklung, bringt aber gewiss auch sicherheitsrelevante Probleme mit sich.

---

[35] Forum, Nokia: *http://blogs.forum.nokia.com/blog/hartti-suomelas-forum-nokia-blog/2007/05/21/who-would-need-jni-for-java-me*

[36] Forum, Android Developers: *http://groups.google.com/group/android-developers/browse_thread/thread/f87e6fce2b26db36/9c7ffe8c2b0b12e2?#9c7ffe8c2b0b12e2*

2.3 Entwicklung für die Plattformen

Wie bei jeder anderen Programmiersprache ist es auch in Java, JavaME und Android theoretisch möglich, Code in einem beliebigen Texteditor zu schreiben und anschließend über Kommandos aus der Konsole zu compilieren. Es ist aber von Vorteil, wenn beim Programmieren die Hilfsmittel einer Entwicklungsumgebung (IDE) eingesetzt werden; ohne diese sind extrem wichtige Vorgänge wie zum Beispiel das Debugging kaum möglich.

2.3.1 Werkzeuge für die Entwicklung mit JavaSE

Um Java Programme zu schreiben wird das *Java Development Kit* (JDK) benötigt.
Zusätzlich gibt es mehrere Entwicklungsumgebungen für Java, unter anderem *NetBeans*[37] und *Eclipse*[38]. Beide IDEs sind Open Source (in Java geschrieben) und können mittels PlugIns erweitert werden.

2.3.2 Werkzeuge für J2ME

J2ME Anwendungen können natürlich nicht direkt auf dem Endgerät, auf dem die Applikation später laufen soll, entwickelt werden. Die Entwicklung findet auf einem Desktop-Rechner statt.
Dafür benötigt man das Sun Java SDK (da es sich um eine Java Variante handelt) und zusätzlich das WTK (das *Java Wireless Toolkit*[39]), welches die notwendigen APIs (inklusive eines Emulators) enthält. Für JavaME existiert außerdem ein Eclipse PlugIn[40], das erlaubt, von Eclipse aus JavaME Projekte zu entwickeln.
Es gibt außerdem SDKs und Emulatoren von Geräteherstellern, wie zum Beispiel das *Nokia Developer's Suite 2.2 for J2ME*, das *Siemens Mobility Toolkit*, das *Motorola SDK v4.4 for J2ME* oder das *Sony Ericsson J2ME SDK*. Diese Emulatoren erweitern die Testmöglichkeiten der Grundfunktionalitäten, die auch beim Sun Emulator vorhanden sind, um einige gerätespezifischen Features, was bei der Vielfalt der Geräte notwendig ist.
Mit diesen Werkzeugen ausgestattet können Applikationen geschrieben und auf einem Emulator getestet werden.

---

[37] NetBeans Homepage: *http://www.netbeans.org/*

[38] Eclipse Homepage: *http://www.eclipse.org*

[39] WTK: *http://java.sun.com/products/sjwtoolkit/download-2_2.html*

[40] Zu finden auf der Eclipse Homepage: *http://www.eclipseme.org/downloads/*

Ziel der Entwicklung mit SDK und Emulatoren ist es, eine Anwendung in Form von MIDlet-Suites zu schreiben und daraus eine .jar-Datei zu erstellen, welche dann auf dem Gerät installiert werden kann.

Das .jar Archiv enthält

- die .class-Dateien,
- die Ressourcen (wie Icons, Texte, Grafiken, Audios)
- die Manifest-Datei (META-INF/MANIFEST.MF). Diese enthält in einer festgelegten Syntax (Name-Wert-Paare) Metainformationen, die u.a. beschreiben, welche Konfiguration und welches Profil verwendet werden und die zur Suite gehörenden MIDlets. Der Manifest-Datei können selbst erzeugte Merkmale hinzugefügt werden. Merkmale können durch *System.getAppProperty(String key)* abgefragt werden, wobei fürs Parameter *key* den Namen des Attributs genannt werden muss.

Neben der .jar Datei kann für die Lauffähigkeit auf bestimmten Geräten auch ein Applikationsdeskriptor benötigt werden, welcher als .jad-Datei vorliegt. Dieser enthält im Grunde dieselben Informationen wie die Manifest-Datei.

2.3.3 Werkzeuge für Android

Um Android Applikationen zu entwickeln braucht man eine ähnliche Ausstattung. Hierfür benötigt man das Android SDK[41], welches ebenso einen Emulator und verschiedene andere Tools enthält.
Mit dem SDK ist es möglich, den Source-Code mit einem beliebigen Editor zu schreiben und anschliessend über Kommandozeile zu kompilieren, aber auch hier bietet sich die Verwendung von IDEs wie Eclipse an, da es dafür einen PlugIn gibt[42].
Nach Installation des Eclipse PlugIns kann man (über den Wizard oder einfach über „Neues Projekt“) ein neues Android-Projekt anlegen.

Ein Android Projekt (beispielsweise in Eclipse) besteht grundsätzlich aus 2 Teilen[43]:

- einem **src** Ordner mit dem Java Quellcode
- und einem **res** Ordner mit den Ressourcen wie beispielsweise die xml Layout-Dateien, die Bilder und andere benötigte Dateien.

---

[41] Informationen zum Android SDK: *http://developer.android.com/intl/it/sdk/1.5_r3/index.html* Stand: Sept 2009.

[42] Das Android PlugIn kann installiert werden über Software Updates, indem man folgende neue Seite hinzufügt: *https://dl-ssl.google.com/android/eclipse/* Stand:Sept 2009, Eclipse Ganymede.

[43] Ein Beispiel des Inhalts eines Android-Projektes gibt Abb. 3-9.

Zusätzlich existiert im root-Verzeichnis die A*ndroidManifest.xml* Datei, welche die Applikationseigenschaften festlegt und in der die Komponenten der Anwendung deklariert werden. Darunter wird hier die minimale Versionsnummer der erforderlichen Android API genannt und alle Activities werden registriert. Ferner werden hier die für die Ausführung benötigte Rechte vergeben. Die Manifest-Datei ist also das, was den Zusammenhalt zwischen Applikationskomponenten gewährleistet.

Die Unterstützung durch das PlugIns wird an einigen Stellen deutlich: Es bietet einen Editor für das komfortablere Editieren der XML-Dateien (Layoutdateien, Manifest-Datei), das Erstellen von Installationspaketen wird vereinfacht und einige im Hintergrund laufende Automatismen erleichtern die Entwicklung (zum Beispiel wird die zentrale Ressourcenklasse „R“ mit dem Ordner „res“ synchron gehalten). Außedem kann der Emulator bequem von der Entwicklungsumgebung aus gestartet werden, um die Applikation zu testen (Abb. 2-5).

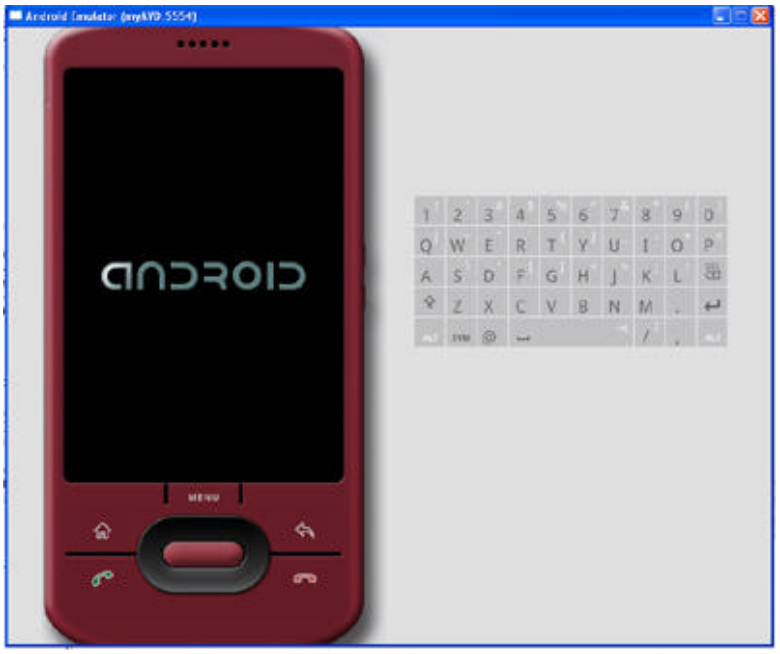

Abbildung 2-5 Der Android Emulator

Eine sehr große Unterstützung gibt das *DDMS* Tool (*Dalvik Debug Monitor Service*), welches in einer eigenen Perspektive innerhalb von Eclipse angezeigt werden kann. Das Tool erlaubt das Debugging (und unterstützt auch das Setzen von Breakpoints im Code) und einige andere Features wie z.B. das Simulieren von GPS-Koordinaten für den Emulator. Allerdings ist die Software in diesem Bereich noch nicht ganz perfekt und funktioniert noch nicht einwandfrei.

2.4 Struktur der Anwendungen

In einer Java Anwendung muss eine statische *main()* Methode in einer der Klassen enthalten sein, diese wird beim Starten des Programms gesucht. Dadurch wird eine Instanz der Java Virtual Machine gestartet.
Diese Struktur findet man auch in anderen objektorientierten Programmiersprachen wieder.
Sowohl JavaME als auch Android weisen eine komplett andere Struktur auf, was sicherlich auch durch die Tatsache bedingt ist, dass es sich um Anwendungen für mobile Geräte handelt. Dort herrschen vorwiegend andere Abläufe: meist geht es ums Anzeigen von Inhalten auf dem Bildschirm und um die Ausführung gut abgeschlossener Dienste, alle vorwiegend Tätigkeiten mit „schnellem“ Lebenszyklus.
JavaME und Android Applikationen unterscheiden sich aber auch stark voneinander. Kurz gefasst: JavaME Applikationen können ein wenig wie Java Standalone Applikationen oder Applets angesehen werden, mit den notwendigen Einschränkungen aufgrund der Ressourcenknappheit. Android Applikationen dagegen können als „Zusammenbau“ verschiedener miteinander verbundenen Bausteine angesehen werden, mit vielen Kombinationsmöglichkeiten.

2.4.1 MIDlets in JavaME

In JavaME ist es nicht möglich, ein Programm mit einer *main*-Routine zu starten. Vielmehr wird die Applikation von der Applikationsmanagementsoftware (AMS) über sogenannte *MIDlets* verwaltet. Zum Starten eines MIDlet wird die Methode *startApp()* aufgerufen. Hier wird die Initialisierung der durchgeführt. Ein MIDlet kann außerdem noch pausiert (*pauseApp()*) oder beendet (*destroyApp()*) werden (Vgl. Abb. 2-6).

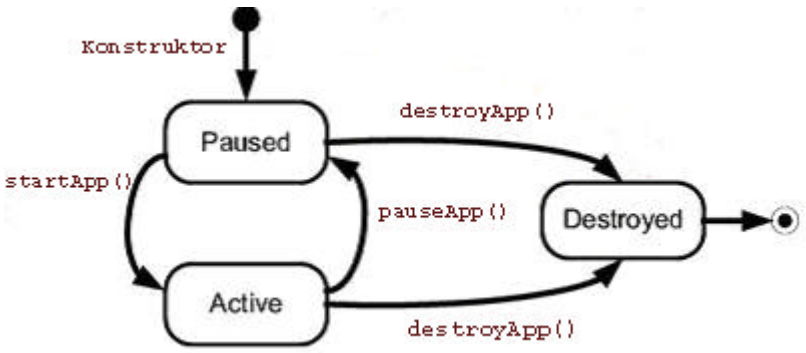

Abbildung 2-6 Lebenszyklus eines Midlets in J2ME

Bei der Erzeugung ist ein MIDlet immer im pausierten Zustand, bis *startApp()* ausgeführt wird. Verantwortlich für einen Zustandswechsel ist immer der Application Manager und nicht das Midlet selbst, diesem kann man allerdings von einem MIDlet aus mitteilen, welche Aktion unternommen werden muss (pausieren, neustarten, beenden) durch die Methoden *notifyPaused()*, *resumeRequest()* und *notifyDestroyed()*.

2.4.2 Android: Activities & Co.

Android hat auch eine eigene Struktur, die nichts mit Standard Java gemeinsam hat, andererseits auch nicht mit J2ME. Android kennt nämlich 4 Komponenten:

- Activities
- Services
- Content Providers
- Broadcast Receivers.

Eine ***Activity*** ist eine kleine Einheit für eine Tätigkeit wie z.B. ein Menu anzeigen, oder eine Funktionalität initiieren, eine Eingabe entgegennehmen oder desgleichen. Jede Activity kann Inhalte auf dem Bildschirm anzeigen (belegt in der Regel den ganzen Bildschirm) und kann mit dem Benutzer interagieren. Ausgangspunkt einer Android Applikation ist immer eine Activity, diese wird mittels der *onCreate()* Methode erzeugt.
Wie MIDlets, können Activities sich in drei Zuständen befinden: running, paused, stopped. Eine Applikation besteht aus einer oder mehreren Activities, die stapelartig geordnet sind (wenn eine Activity beendet wird, wird die unmittelbar darunter liegende sichtbar). Um von Screen zu Screen bzw. Activity zu Activity zu navigieren werden sogenannte *Intents* verwendet, also Nachrichten, die eine Beschreibung dessen enthalten, was gestartet werden soll. Beispielweise kann eine weitere Activity wie folgt aufgerufen werden:

```
Intent iStickyNotes = new Intent(this, CheckLocationAndNotes.class);
startActivity(iStickyNotes);
```

Das System sucht daraufhin die Activity, welche auf das Intent passt.
Intents sind ein zentraler Bestandteil des innovativen Konzepts von Android. Nicht nur das Starten einer Activity sondern allgemeiner die Anforderung für eine Aktion kann über Intents gestartet werden. Intents werden in der Manifest-Datei registriert. Wenn dann ein Intent abgesetzt wird, durchsucht das System die verfügbaren Komponenten und sucht das passende Gegenstück, um die Aktion durchzuführen (eine Activity starten aber auch eine Adresse im Adressbuch suchen, oder eine URL aufrufen..). Dadurch ist eine späte

Bindung zur Laufzeit von Aktion und auszuführendem Code möglich, welche die Applikation viel dynamischer gestalten lässt.

Ein ***Service*** läuft im Hintergrund - typisches Beispiel dafür ist das Abspielen einer Audiodatei – er kann also keine Inhalte auf dem Screen anzeigen und auch nicht mit dem Benutzer interagieren, er erledigt lediglich eine Aufgabe. Die Ermittlung der GPS Position ist in unserem Projekt als Service realisiert worden. Ein Service wird über die Methode *Context.startService()* gestartet.

Broadcast Receiver reagieren auf bestimmte Informationen (z.B. Batteriestand niedrig) und können darauf basierend eine Activity zur Anzeige einer Meldung starten oder den Notification Manager verwenden, um den Benutzer zu benachrichtigen.

Content Provider dienen dazu, Daten innerhalb der Applikationen auszutauschen. Daten (auch Dateien) sind immer privat innerhalb einer Applikation und können nur durch die Mechanismen eines Content Providers für andere Applikationen zur Verfügung gestellt werden[44].

Eine Applikation läuft in einem eigenen Prozess (mit einer eigenen Instanz der Dalvik VM) und alle ihre Komponenten laufen – wenn nicht anders angegeben - im selben Prozess bzw. Thread. In der manifest-Datei kann für jede Komponente (also die Elemente <activity>, <service>, <receiver>, <provider>) über das Attribut *process* ein Prozess angegeben werden, in dem die Komponente laufen soll. Dadurch können auch einzelne Komponente verschiedener Applikationen so eingestellt werden, dass sie im selben Prozess laufen.
Im Hauptthread des Prozesses werden alle dazugehörigen Komponenten instantiiert.
Komponenten, die eventuell blockierend sein könnten, weil sie lange oder kritische Berechnung ausführen, sollten besser in separatem Prozess (oder Thread innerhalb des Prozesses) verlagert werden, damit nicht die gesamte Applikation dadurch eventuell blockiert wird.

Eine Applikation hat keine Kontrolle über ihren eigenen Lebenszyklus, das bedeutet, sie kann jederzeit bei Bedarf an Ressourcen von Android beendet werden. Im Fall, dass Ressourcen freigemacht werden müssen, wertet Android aus, welche Prozesse am ehesten geschlossen werden können, zum Beispiel kommen dafür Prozesse in Frage, die in dem Moment gerade nicht angezeigt werden, also sich im Zustand *paused* befinden.

[44] S. dazu später Kap 4 über Datenpersistenz, insbesondere 4.2.2.

2.5 Zusammenfassung

Der Architekturvergleich und die Gegenüberstellung der Programmiersprachen in ihren wesentlichen Elementen haben ergeben, dass Android zweifellos in seiner Programmiersprache mächtiger ist und nach einem neueren Konzept aufgebaut ist, welches deutlich mehr Möglichkeiten eröffnet.

Man kann sagen, Android profitiere von der J2ME-Erfahrung: Es zeigt auf der einen Seite gewisse Ähnlichkeiten mit J2ME, auf der anderen Seite ist auch zu erkennen, wie Android über die Grenzen von J2ME hinauszugehen versucht, da gewisse Einschränkungen oder Nachteile von J2ME in den letzten Jahren negativ aufgefallen sind. Zum Beispiel die inzwischen zu eingeschränkte API und die Probleme der Portierbarkeit.

Android-Java ähnelt viel mehr JavaSE und ist folglich viel komfortabler für Java-Entwickler als J2ME. Hier ist ein wichtiger Punkt und vielleicht ein Schritt, den Sun mit J2ME verpasst hat: Die Programmiersprache der Micro Edition zu erweitern und an die neuen Geräte sowie an die Neuentwicklungen von JavaSE anzupassen.
Unter dieser Hinsicht ist Android sicher besser ausgerüstet.
Dazu kommt das „bessere“ Strukturkonzept von Android, das erlaubt, durch die *Intents* stark anpassbare und beliebig austauschbare Anwendungen zu erstellen (wie in Abs. 2.4.2 erläutert).

Gleichzeitig sorgt Android mit seinem speziellen Umgang mit dem Java-Bytecode und mit der Einführung vieler eigener Features immer wieder für geteilte Meinungen, vor allem bezüglich seiner Dalvik Virtual Machine. Google rechtfertigt die Entscheidung einer maßgeschneiderten Lösung mit Optimierungsgründen, was sicherlich richtig ist, stößt aber auf Widerstand seitens vieler Befürworter der Standards[45].

Weitere entscheidende Kriterien, die bei Systemen für mobile Geräte eine große Rolle spielen, sind die Möglichkeiten der Gestaltung von Benutzungsoberflächen und der Umgang mit persistenten Daten.
In den nächsten zwei Kapiteln wird auf diese Themen eingegangen, wobei die Techniken anhand von Beispielanwendungen verdeutlicht werden, als roter Faden dient auch hier ein Location Based Service; zum Schluss wird erneut ein Fazit unter Berücksichtigung auch dieser zwei wichtigen Aspekte gezogen.

[45] Ausführlicher in Kapitel 5.

3 GUIs

Graphical User Interfaces (GUIs) sind von großer Bedeutung bei Anwendungen, die für den Benutzer Informationen aufbereiten bzw. mit dem Benutzer interagieren sollen, was bei mobilen Geräten im Vordergrund steht.
Grafische Darstellungen sind ressourcenintensiv, ein kritisches Thema bei mobilen Geräten. Außerdem muss bei displayfüllenden Anwendungen (wie es bei kleinen Displays der Fall ist) berücksichtigt werden, dass je nach Größe des Displays die Anzeige stark variieren kann. Eine ähnliche Problematik kennt man aus dem Bereich der Web Anwendungen, wo das Aussehen einer Webseite stark von der Auflösung und vom verwendeten Browser abhängt.
User Interfaces für J2ME und Android funktionieren ganz anders als in den bekannten Java Lösungen. Das ist deshalb, weil das Layout für grafische Applikationen nicht so genau kontrolliert werden kann und je nach Gerät das Aussehen von Applikationen große Unterschiede aufweisen kann. Außerdem sind Zeigegeräte (wie Maus bzw. Stift) oft nicht vorhanden und bei dem kleinen Display ist es nicht sinnvoll, viele Fenster nebeneinander darzustellen. Ein anderer Ansatz also.
Die Ansätze von J2ME und Android unterscheiden sich darüber hinaus auch stark voneinander: J2ME setzt den Schwerpunkt erneut auf die Portabilität und bietet eine begrenzte Anzahl von Möglichkeiten, mit dem Hintergrund, dass diese auf den meisten Geräten funktionieren müssen; dieses Modell ist vielleicht nicht mehr ganz zeitgemäß, denn auch die ästhetischen Ansprüche sind inzwischen sehr gestiegen. Android setzt den Schwerpunkt auch hier vielmehr auf gute Resultate innerhalb des eigenen abgeschlossenen Systems und bietet dafür sehr schöne Darstellungsmöglichkeiten.

3.1.1 Benutzungsschnittstellen in Java

Java erlaubt es, grafische Benutzungsschnittstellen zu definieren; dadurch ist es möglich, Daten visuell aufzubereiten oder dem Benutzer Eingabemasken zu präsentieren, welche die Interaktion mit dem Programm erleichtern oder verschönern.
Schon in der ersten Java Version war zu diesem Zweck das *Abstract Windowing Toolkit* (***AWT***) enthalten, das bis heute in der Java API geblieben ist. AWT bietet Methoden für die wesentlichen Operationen zum Zeichnen und zur Ereignisbehandlung, sowie einen Satz von GUI-Komponenten.
AWT ist aber sehr einfach gehalten bzw. hat einen gravierenden Mangel: AWT nutzt native Komponenten des Betriebssystems (es hat eine sogenannte *Peer Architektur*), was große

Nachteile in der Portabilität aber auch in der Effizienz mit sich zieht. Zu seinem Entstehungspunkt war nicht vorauszusehen, dass Java sich so schnell entwickeln würde und dass folglich auch die Benutzerschnittstelle eine große Rolle spielen würde. Vielmehr war AWT in 6 Wochen Entwicklungszeit entstanden, da die neue Programmiersprache auch eine Möglichkeit der GUI-Entwicklung bieten musste.
Um diesen Mangel zu beheben und professionellere Oberflächen gestalten zu können wurde im Java SDK 1.2 ***Swing*** eingeführt (*javax.swing.**). Swing besteht ausschließlich aus einer in Java geschriebenen Komponentenbibliothek und ist dadurch unabhängig vom Betriebssystem und folglich viel leistungsfähiger.
Swing ist aber nicht ein Ersatz für AWT sondern setzt eher auf AWT auf:
Die Swing Klassen *JFrame*, *JWindow* und *JDialog* erweitern die entsprechenden AWT-Klassen *Frame*, *Window* und *Dialog*. Das sind drei auf der AWT-Klasse *Container* basierende Klassen (welche nicht nach der Peer Architektur realisiert sind!), wie die folgende Grafik (3-1) verdeutlicht:

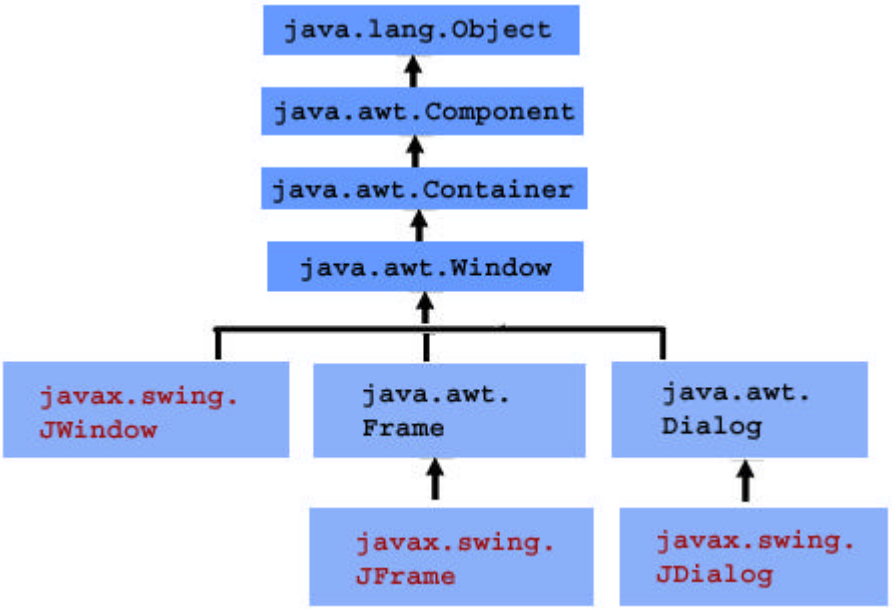

Abbildung 3-1 Hierarchie der Swing und AWT Klassen

Von diesen drei Klassen - oder direkt von *java.awt.Container* - leitet Swing die eigenen Komponenten ab (*javax.swing.JComponent*).
JFrames oder JWindows aber auch der einfachere JPanel sind also “Container”, in denen weitere Elemente platziert werden können.
Die Anordnung der Komponenten in einem Container kann durch Layouts definiert werden (FlowLayout, BorderLayout, GridLayout, CardLayout, GridBagLayout); ein

Layout wird über eine Instanz des Layout-Managers erzeugt und einem Container mit der Methode *setLayout()* zugewiesen.
Schließlich gibt es EventListener, welche den Elementen hinzugefügt werden können, um auf Ereignisse wie z.B. Benutzereingaben zu reagieren.
Zum Beispiel kann einem Button ein ActionListener hinzugefügt werden; dieser verfügt über die Methode *actionPerformed*, die ausgeführt wird, wenn der Button angeklickt wird.

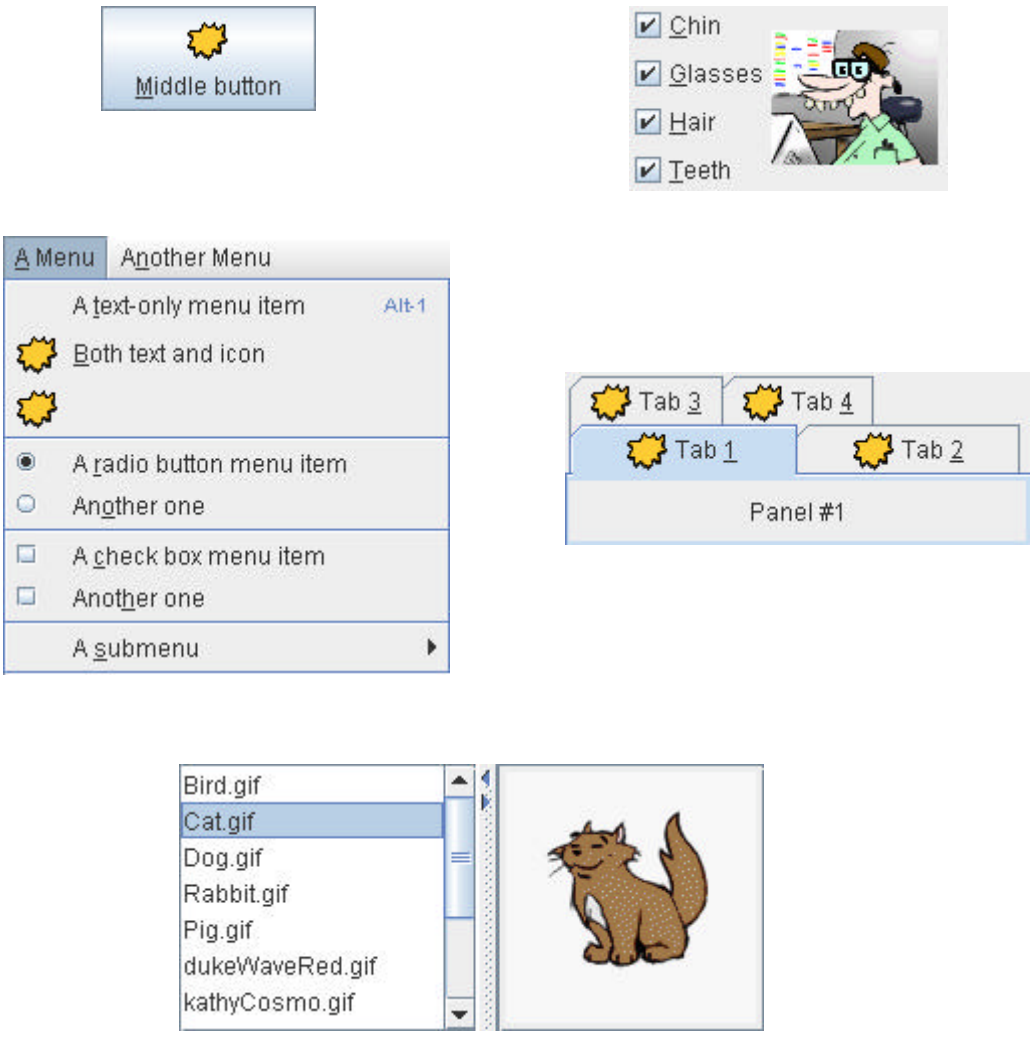

Abbildung 3-2 Look and Feel einiger Java Swing Komponenten[46]

[46] Quelle: sun, *http://java.sun.com/docs/books/tutorial/ui/features/components.html*

3.1.2 Benutzungsschnittstellen in JavaME

Die API für grafische Benutzeroberflächen in JavaME ist im Package *javax.microedition.lcdui* (Lowest Common Denominator User Interface[47]). Wie der Name sagt, stellt LCDUI einen gemeinsamen Nenner für alle Applikationen auf allen Kleingeräten dar, auch wenn diese dann sehr unterschiedlich ausfallen, aufgrund verschiedener Bildschirmgröße, Farbtiefe usw..

Grundstein ist die abstrakte Klasse ***Displayable***, die einige Eigenschaften eines anzeigbaren Bildschirms definiert (zum Beispiel das Setzen eines Titels und die Möglichkeit, einen Listener zu registrieren). Von dieser Klasse erben die ebenso abstrakten Klassen ***Screen*** und ***Canvas***, welche als Schnittstellen jeweils für die sogenannten *High Level UI* und *Low Level UI* dienen (s. Abb. 3-3).

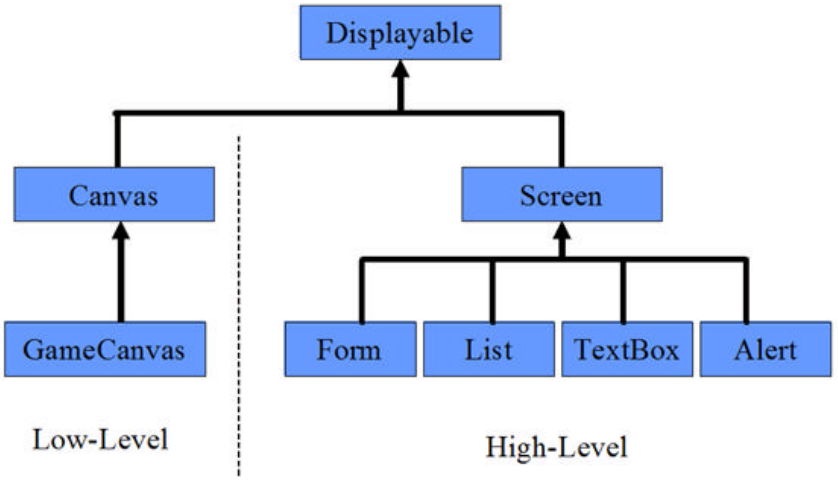

Abbildung 3-3 Beziehungen in javax.microedition.lcdui

Diese Einteilung in zwei Schichten wird angeboten, damit der Programmierer zwischen einer Minimalausrüstung und größeren Anzeigemöglichkeiten wählen kann:
Die High Level UI erlaubt, minimale grafische Oberflächen mit vorgefertigten Elementen zu gestalten (Alerts, Buttons, Textfeldern, Forms usw.); hier steht die Funktionalität im Vordergrund, das Aussehen kann meist nur wenig verändert werden; dafür sind die Anwendungen gut portierbar.
Die Low Level UI erlaubt dem Entwickler viel genauer über das Aussehen der grafischen Komponenten zu bestimmen (die grafische Ausgabe erfolgt zum Beispiel pixelweise),

[47] Aber auch: „Liquid crystal display user interface“: Java ME Reference Glossary: *http://developers.sun.com/mobility/glossary/*

wobei nicht sichergestellt ist, dass die Anwendung auf anderen Geräten genauso aussehen wird. Spieleentwicklung, ein wesentlicher Antrieb für die Entwicklung mit J2ME, verwendet die Klasse *javax.microedition.lcdui.game.GameCanvas*, welche von Canvas erbt.

Einige Beispiele:
Gewöhnliche schlichte funktionale Darstellung mit High Level UI (Abb 3-4):

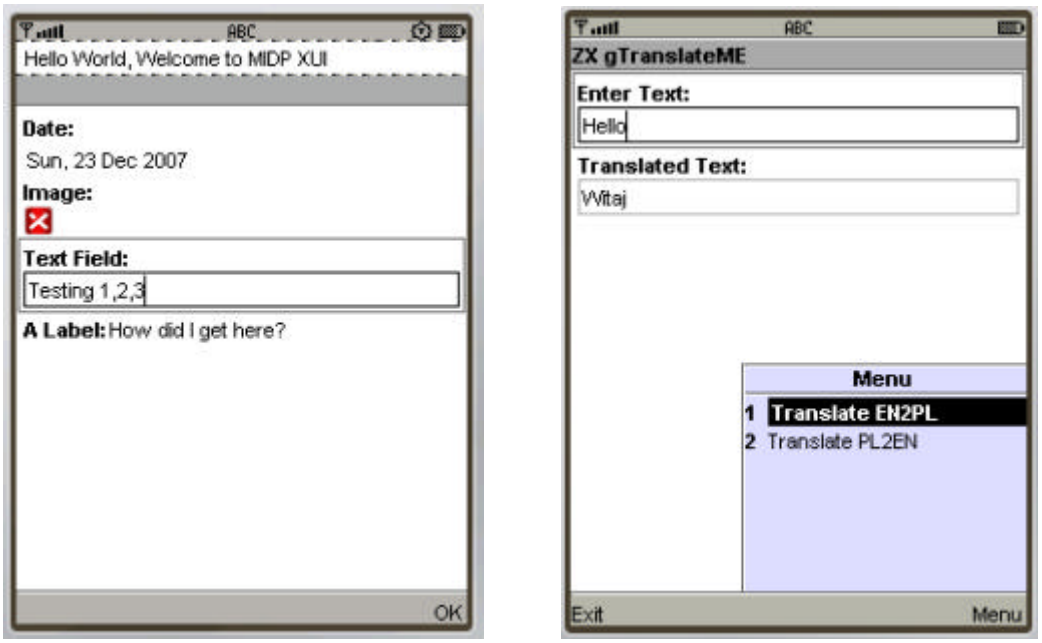

Abbildung 3-4 High Level UI (J2ME)

Wenn es auf Pixelgenauigkeit ankommt und wenn einzelne Tasten erkannt werden sollen (unentbehrlich bei Spielen), kann die Low Level UI verwendet werden, mit der sich eindrucksvolle 2D-Resultate erreichen lassen (Abb 3-5):

Abbildung 3-5 Low Level UI (J2ME), Quelle: Handango.com

3.1.3 Benutzungsschnittstellen mit Android

Activities sind bei Android diejenigen Elemente, die Anzeigemöglichkeiten besitzen. Dazu gibt es in Android vorwiegend die Klasse *android.view.View* bzw. *android.view.ViewGroup*.
Views sind rechteckige Bereiche auf dem Bildschirm, die grafische Elemente enthalten können, insofern ist *View* die Basisklasse für alle Designerelemente. Unter den Designerelementen gibt es auch solche, die mit dem Benutzer interagieren können, folglich ist View auch die Basisklasse für *Widgets* (kleine vollständige und funktionsfähige Elemente für die Benutzerinteraktion: z.B. Buttons, Pulldown-Menus, Checkboxes usw., welche mit EventListener gesteuert werden können).
ViewGroup kann mehrere Views zu einer Gruppe zusammenfassen und ist insofern die Basisklasse für *Layouts*, da Layouts mehrere Elemente zusammenfassen.

Android bietet über die Views alle gewöhnlichen Elemente an, aus denen eine grafische Applikation zusammengebaut werden kann und dazu ein breites Spektrum an Layout-Möglichkeiten.
Eine Innovation stellen die *Map View* und *Web View* dar, welche Landkarten bzw. Browseranzeigen direkt als „View“ ermöglichen. Obwohl z.B. Karten auch extern zur Applikation angezeigt werden können, bietet die MapView den Vorteil, dass die Anzeige einer Karte eng mit der Applikation zusammengehen und vom Code der Applikation kontrolliert werden kann (die Applikation selbst kann folglich beliebige Operationen auf den Karten durchführen, zum Beispiel zoomen usw.). Ähnliches gilt für die WebView, bei der eine Internetseite in der Applikation integriert werden kann.

Um eine View einer Activity anzuhängen wird die Methode *setContentView()* aufgerufen:

```
setContentView(R.layout.main);
```

oder direkt:

```
TextView tv = new TextView(this);
tv.setText(“Text to be displayed in this Activity”);
setContentView(tv);
```

Im ersten Beispiel wird als Parameter eine xml-Datei mitgegeben (hier referenziert R.layout.main die Standard xml-Datei: res/layout/main.xml), in der die Layoutelemente definiert sind. Im zweiten werden Elemente über *Views* direkt im Code definiert und angezeigt (hier eine TextView, ein Behälter für einen Text).

Im *res*-Ordner stehen die XML-Layout-Dateien, auf die im Code über die generierte R-Klasse zugegriffen werden kann, außerdem andere XML-Dateien zur Beschreibung

anderer Elemente (wie Texte für Strings, verwendete Farben..) und auch weitere Ressourcen wie Grafiken können hier abgelegt werden.
Die in den Layoutdateien einsetzbaren Parameter zur Beschreibung der Layoutelemente kommen aus den Packages *android.view.ViewGroup.LayoutParams* und *android.widget*, funktionieren aber auch recht intuitiv und erinnern ein wenig an html: Dadurch können Positionierung, Farbe und andere Eigenschaften der Layoutelemente festgelegt werden.
Wichtige Layouts sind: *FrameLayout* (einfach ein leeres Fenster, das mit einem Objekt, zum Beispiel einem Bild gefüllt werden kann) *LinearLayout* (mehrere Elemente werden horizontal oder vertikal positioniert), *TableLayout* (mit Reihen und Spalten, wie bei einer html-Tabelle), *AbsoluteLayout* (Elemente werden mit absoluten x/y-Koordinaten angegeben) *RelativeLayout* (Elemente werden relativ zu Eltern / Geschwistern positioniert) aber auch *Gallery* (Elemente werden horizontal gescrollt) und einige mehr.

Auf diese Art wird eine Trennung der grafischen Elemente vom Code erreicht. Im Code muss lediglich eine Layout-Datei eingebunden werden und grafische Inhalte können separat bearbeitet werden. Die Trennung von funktionalem Code von der grafischen Darstellung ist eine besonders effiziente Lösung, zum einen weil die Möglichkeit gegeben ist, Programm und Design von unterschiedlich spezialisierten Programmierern implementieren zu lassen, zum anderen weil Designänderungen ohne Codeänderungen möglich sind bzw. mit demselben Code verschiedene Ergebnisse realisiert werden können, indem die Layout-Datei, nicht aber die Funktionalität verändert wird.
In Java und auch in J2ME ist diese Trennung nicht möglich, grafische Inhalte werden im Programmcode definiert.

Abbildung 3-6 Einige Android Layouts und Views[48]

[48] aus: *http://www.maximyudin.com/tag/advice/page/2, http://blog.brothersoft.com/tag/gmail/, http://www.webmonkey.com/blog/Android_SDK_Update_a_Little_Late_to_the_Party*

Ein weiterer Unterschied zu J2ME ist, dass Android auch 3D-Grafiken unterstützt, dies ist möglich durch die OpenGL/ES Core Library. In einer einzelnen Applikation können sogar 2D und 3D Grafiken kombiniert werden.
Mit Android können also insgesamt anspruchsvollere grafische Schnittstellen entworfen werden, die dem modernen ästhetischen Sinn viel mehr entsprechen als die von J2ME, welche heutzutage teilweise veraltet wirken.

3.2 Beispielanwendung: Anzeige von GPS-Daten in J2ME und Android

Um die grundlegende Struktur der Anwendungen und die Anzeigemöglichkeiten zu verdeutlichen, folgt nun ein Beispiel einer einfachen Anwendung, in der ein mobiles Gerät die aktuelle GPS-Position ermittelt und sie nach Interaktion mit dem Nutzer auf dem Display anzeigt.

3.2.1 GPS-Midlet in J2ME

Die Anwendung, um Positionsdaten anzuzeigen besteht im Kern aus zwei Klassen: *LocMidlet* (das Midlet selbst, womit die Applikation startet) und *LocPosition* (eine Klasse, welche unter Verwendung des optionalen J2ME Package *javax.microedition.location* die Positionsdaten ermittelt).

Das Midlet definiert die grafischen Elemente und füllt sie mit Inhalten; dabei wird LocPosition als Thread aufgerufen und dort die Ermittlung der Koordinaten durchgeführt. Dem Benutzer wird eine kleine Auswahlliste angeboten: Wenn er die Option wählt, seine Positionsdaten einzusehen, wechselt die Darstellung und anstatt der Auswahlliste wird die mit den Koordinaten gefüllte Textbox auf dem Display angezeigt.
So sieht die Anwendung aus: Beim Zugriff auf Location Diensten wird der Benutzer aus Sicherheitsgründen gefragt, ob dies gewünscht ist (Abb. 3-7). Dies ist eine bekannte Sicherheitseigenschaft von J2ME, die nicht beeinflusst werden kann.

Abbildung 3-7 Start des Location Midlets

Nach einer positiven Antwort des Benutzers kann die Applikation beginnen (Abb 3-8):

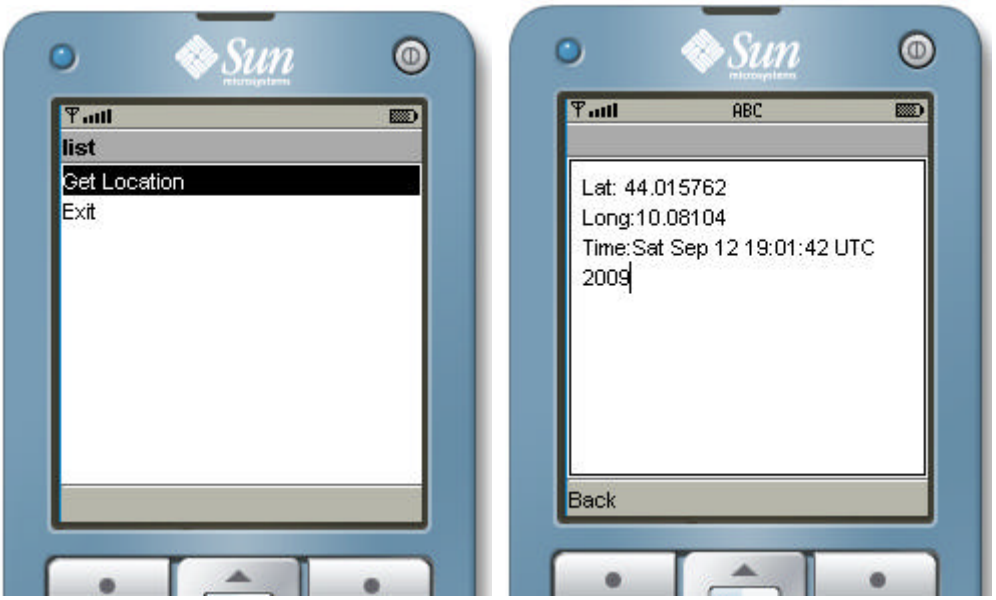

Abbildung 3-8 Ablauf des Location Midlets

Der Code von LocMidlet (Listing 3.1) besteht aus der Methode *startApp()* und aus den grafischen Elementen mit ihren Listener:

```
public class LocMidlet extends MIDlet implements CommandListener {

public void startApp() {
            disp  = Display.getDisplay(this);
            cmd1 = new Command("Back",Command.BACK, 0);
            tb1 = new  TextBox(" "," ",150, TextField.ANY );
            lp = new  LocPosition(tb1);
            tb1.addCommand(cmd1);
            tb1.setCommandListener(this);
            lpThr = new Thread(lp);
            try {
                  lpThr.start();
            }  catch ( Exception e) { //exception
            }
            disp.setCurrent(getList());
}
public List getList() {
            if (list == null) {
                  list = new List("list", Choice.IMPLICIT);
                  list.append("Get Location",null);
                  list.append("Exit",null);
                  list.setCommandListener(this);
            }
            return list;
}
```

```
void listAction() {
      String selectedString =
                  getList().getString(getList().getSelectedIndex());
      if (selectedString != null) {
            if (selectedString.equals("Get Location")) {
                        disp.setCurrent(tb1);
            }else if (selectedString.equals("Exit")) {
                        notifyDestroyed();
            }
      }
}

public void commandAction(Command command, Displayable displayable){
      if (command == cmd1) {
            disp.setCurrent(getList());
      }
      if (displayable == list) {
            if (command == List.SELECT_COMMAND) {
                  listAction();
            }
      }
 }
```

Listing 3.1 Midlet für Positionsdaten

startApp() ist die Methode, die am Anfang des Lebenszyklus eines Midlets zur Initialisierung der Komponenten aufgerufen wird.

Jedem Midlet ist ein Display-Objekt zugeordnet; dieses wird durch *Display.getDisplay(this)* ermittelt (*this* ist hier das Midlet selbst).
Um Inhalte anzuzeigen gibt es die Methode des Display-Objekts: *setCurrent(Displayable d)*. Dadurch werden aktuelle Inhalte angezeigt und nicht benötigte Anzeigen wandern in den Hintergrund.
Im Beispiel wird zunächst durch *setCurrent()* eine Liste von Auswahlmöglichkeiten auf dem Display angezeigt, die in *getList()* definiert werden. Die Liste wird mit Listener versehen.
In *listAction()* steht die Implementierung der Listener: Falls der Benutzer das Element „Get Location“ wählt soll der Inhalt der Textbox mit den Positionsdaten angezeigt werden; bei „Exit“ soll das Programm verlassen werden (und notifyDestroy soll ausgelöst werden).

Die Instanz der LocPosition Klasse wird in *startApp()* als Thread gestartet (rot markiert in Listing 3.1) und bekommt eine Textbox zugewiesen, in die das Ergebnis geschrieben werden kann. Diese Textbox wird dann sichtbar, wenn das Listenelement „Get Location“ ausgewählt worden ist (durch: `disp.setCurrent(tb1)`).

Die Methode *commandAction()* wird aufgerufen, wenn ein Befehlsereignis auf einem Displayable erkannt wird. So ein Ereignis kann entweder das Auswählen eines Befehls (wie hier im Fall von cmd1 für die Textbox) oder ein SELECT_COMMAND einer Liste. Deswegen wird für diese zwei Optionen jeweils ein Verhalten definiert: Entweder hat der Benutzer den Befehl „back“ ausgewählt, in dem Fall soll die Liste wieder angezeigt werden, oder er hat ein Listenelement ausgewählt, während die Liste auf dem Display stand.

Die Berechnung der GPS-Koordinaten erfolgt in separatem Thread und ist in der Klasse *LocPosition* implementiert (Runnable), die mit der in LocMidlet definierten Textbox als Parameter initialisiert wird.
Hier werden im Konstruktor *Location Providers* nach der Position gefragt; dafür werden sogenannte *Criteria* festgelegt (Criteria sind Einschränkungen bezüglich Genauigkeit, Antwortzeit, Kosten, Energieverbrauch usw.) und es werden die Providers durchsucht, die solche Criteria erfüllen:

```
Criteria crit = new Criteria();
crit.setCostAllowed(true);
crit.setPreferredPowerConsumption(Criteria.NO_REQUIREMENT);
LocationProvider provider = LocationProvider.getInstance(crit);
```

Wenn einmal die Providers gefunden sind, können Positionsdaten entweder einmalig (synchron) oder über einen Listener in regelmäßigen Abständen abgefragt werden.

Einmalige Abfrage:

```
Location locPos = provider.getLocation(60);
coordinates = locPos.getQualifiedCoordinates();
latitude = coordinates.getLatitude();
longitude = coordinates.getLongitude();
```

Oder über einen Listener:

```
provider.setLocationListener(this,-1,-1,-1);
```

Im letzten Fall wird durch `setLocationListener()` einen Listener registriert, der bei einem Positionswechsel neue Koordinaten ermittelt. Die Parameter „-1“ bedeuten Default-werte für Zeitintervall, Timeout und MaxAge.

Die Ergebnisse der GPS-Abfrage werden noch als String zusammengesetzt und in die Textbox geschrieben.

Der vollständige Code des LocPosition Service ist in Listing 3.2 zu finden.

```
public class LocPosition implements Runnable, LocationListener {
      Location locPos;
      StringBuffer sb = null;
      double latitude = ""; // N, S
      double longitude= ""; // E, W
      LocationProvider provider= null;
      Coordinates coordinates = null;
      TextBox loctb;  // TextBox to display the result

      public LocPosition(TextBox t1) {
        loctb =t1;
        sb = new StringBuffer("");
        try {
           Criteria crit = new Criteria();
           crit.setCostAllowed(true); //default value
           crit.setPreferredPowerConsumption(Criteria.NO_REQUIREMENT);
           provider = LocationProvider.getInstance(crit);
           locPos = provider.getLocation(60);
        } catch (LocationException ex) {
                 ex.printStackTrace();
        }
       provider.setLocationListener(this,-1,-1,-1);
      }

      /** fill the textbox with position data */
      public void run() {
            while( true ) {
                  String ss1 = new String(sb);
                  String displaylat;
                  String displaylong;

                  if (latitude != 0  && longitude != 0){
                        displaylat = Double.toString(latitude);
                        displaylong = Double.toString(longitude);
                  }
                  else{
                        displaylat = "(no data)";
                        displaylong = "(no data)";
                  }

                loctb.setString("Lat: " + displaylat + '\n' + "Long: "
                              + displaylong+ '\n' + "Time:" + ss1);
            }
      }

      /** called from the LocationListener */
      public void locationUpdated(LocationProvider provider, Location
      locPos) {
            coordinates = locPos.getQualifiedCoordinates();
            sb.delete(0,sb.length());
            sb.append(new Date(locPos.getTimestamp()).toString());
            if (coordinates!= null){
                  latitude = coordinates.getLatitude();
                  longitude = coordinates.getLongitude();
            }
      }
```

Listing 3.2 J2ME – Die Klasse: LocPosition

3.2.2 GPS-Activity auf Android

Die Android-Applikation besteht aus einer Activity, verantwortlich für die Anzeige der Inhalte auf dem Bildschirm und für die Interaktion mit dem Benutzer, und einem Service, der die Positionsdaten ermittelt.
Die Activity wird in der AndroidManifest.xml Datei deklariert, dadurch ist sie der Applikation bekannt. Die Elemente, die für das Layout benötigt werden, befinden sich im Ordner „res“ und können über die generierte R-Klasse vom Code aus erreicht werden.
Die Struktur der Applikation sieht aus wie in Abb. 3-9:

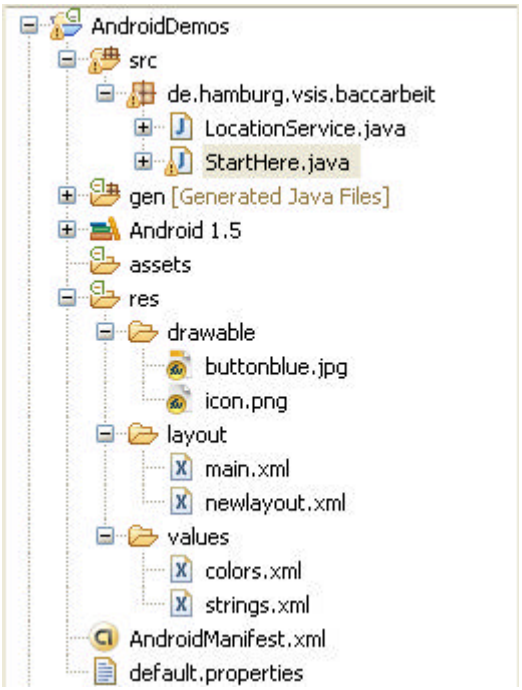

Abbildung 3-9 Verzeichnis einer Android Applikation

In der Methode *onCreate()*, aufgerufen beim Erzeugen der Activity (S. Listing 3.3), bekommt die Activity ein Layout zugewiesen, außerdem wird dem Button (welcher über seine id in der Layout-Datei gefunden werden kann) ein OnClickListener hinzugefügt, da beim Klicken eine entsprechende Aktionen ausgelöst werden soll (der Aufruf des LocationService).
Der Button wird in der Layout-Datei definiert (Farbe, Position, Text) und in der Activity referenziert; hier bekommt er auch den Listener.

```
 public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.newlayout);

       Button button = (Button)findViewById(R.id.ok);
       button.setOnClickListener(blueButtonListener);
}
private OnClickListener blueButtonListener = new OnClickListener()
{
       public void onClick(View v) {
           useLocationService();
       }
};
protected void useLocationService(){
      LocationService ls = new LocationService();
      Thread lsThr = new Thread(ls);
      lsThr.start();
      writeResult(RESULT_OK, ls.getCurrentLocation(this));
}
```

Listing 3.3 Android Activity für Positionsdaten

Die Methode *writeResult()* zeigt das Ergebnis des Location Service auf dem Display an (welcher ähnlich implementiert ist wie in der Projektaufgabe, S. Listing 1.1).

Mit `setContentView(R.layout.newlayout)` wird also für diese Activity die Layout-Datei aus dem Verzeichnis res/layout mit dem Namen „newlayout.xml“ referenziert und verwendet. In dieser Datei kann ein Layout deklariert werden sowie die darin enthaltenen Elemente: Im einfachen Fall dieser Applikation werden die Abstände, die Texte und für den Button eine entsprechende Grafik definiert.
Da es sich hier um ein *RelativeLayout* handelt, wird der Button relativ zum vorherigen Element positionert (Listing 3.4).

```
 <RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

  <TextView
        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dip"
        android:layout_marginLeft="10dip"
        android:layout_marginRight="10dip"
        android:text="@string/activityText"/>
```

```
 <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/label"
        android:layout_centerInParent="true"
        android:layout_marginTop="20dip"
        android:background="@drawable/buttonblue"
        android:layout_marginLeft="10dip"
        android:text="Check Position" />
</RelativeLayout>
```

Listing 3.4 Android Layout Datei

Der Text für die Textbox wird mittels `android:text="@string/activityText"` aus der Datei res/values/strings.xml entnommen, wo die Texte separat definiert werden:

```
<resources>
       <string name="activityText">You can check your GPS Position by pressing
the button below</string>
</resources>
```

Mit `android:background="@drawable/buttonblue"` wird eine Grafik aus dem Verzeichnis res/drawable referenziert.

Alternativ wäre es möglich, einen Standardbutton zu verwenden und dafür beispielsweise Farben zu definieren, mit `android:background="@color/green"`. In diesem Fall steht die Farbe „green“ in res/values/colors.xml:

```
<resources>
       <color name="green">#7700ff</color>..
</resources>
```

Das Ergebnis sieht so aus (Abb. 3-10):

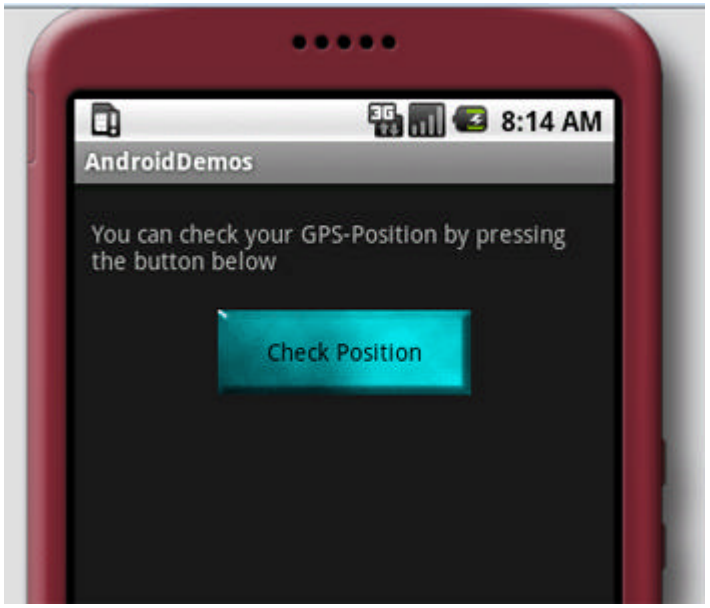

Abbildung 3-10 Android Activity mit RelativeLayout

Beim Klicken auf den Button soll nun die Position abgefragt und angezeigt werden.
Dazu ist der Listener in der Activity deklariert worden.
Beim Auftreten des Klick-Events soll die Methode aufgerufen werden (*useLocationService()* in Listing 3.3), welche den Location Service in eigenem Thread startet und das Ergebnis (einen String mit den Koordinaten) von dem Service zurückbekommt.

Um diesen Wert nun anzuzeigen kann beispielsweise ein PopUp Fenster verwendet werden (Listing 3.5):

```
Builder alert = new AlertDialog.Builder(this);
alert.setTitle("Here is your Position:");
alert.setMessage(theresult);
alert.setPositiveButton("ok", alertListener);
alert.show();
```

Listing 3.5 Android: Anzeige des Ergebnisses in Alert Fenster

wobei *theresult* den String mit den kommaseparierten Koordinaten enthält.

Im PopUp Fenster können wieder verschiedene Buttons deklariert und mit Listeners versehen werden. Das Ergebnis sieht aus wie in Abb. 3-11:

Abbildung 3-11 Anzeige eines Alert Fensters mit Android

4 Verwaltung persistenter Daten

Der Umgang mit Daten auf mobilen Geräten konfrontiert Informatiker mit vielen Schwierigkeiten und neuen Fragestellungen, die im konventionellen Bereich (stationäre ressourcenreiche Computer und klassische Datenbanken) nicht von Relevanz sind.
Zum einen muss der Begriff der Datenbanken erweitert werden, um ihn an *mobile* Anwender oder *mobile* Daten anzupassen, zum anderen müssen die einzelnen Geräte mit den aus den Abfragen erhaltenen Daten umgehen können, was wegen der beschränkten Ressourcen (wenig Haupt- und Persistenzspeicher, niedrige Prozessorleistung, kleinem Display, Batteriebetrieb) und der Sicherheitsmaßnahmen auf spezielle Art gelöst werden muss: Kritisch sind vor allem die Aufbereitung und die Speicherung der erhaltenen Daten.

Nach einer allgemeinen Einführung in die Thematik der mobilen Datenbanken wird aus Platzgründen und aus Konsistenzgründen nur der letzte Strang verfolgt, nämlich die Verwaltungs- und Speichermechanismen und generell der Umgang mit persistenten Daten auf mobilen Geräten. Dabei werden die Lösungen von J2ME und Android kontrastiv untersucht.

4.1 Mobile Datenbanken

4.1.1 Datenabfragen von mobilen Geräten aus

Merkmal des Mobile Computing ist die Tatsache, dass der Nutzer sich bewegt. Wenn dieser also Informationen von einer zentralen Datenbank abfragt, können diese eventuell von der Position des Anfragenden abhängig sein. Diese inzwischen gängige Thematik fällt in den Bereich der *Location Based Services* (LBS, s. dazu [ScVo04]).
Es wird unterschieden in

- *Nicht-positionsabhängige Anfragen*. Beispiel: Ein mobiler Nutzer fragt eine Literaturliste zu einem bestimmten Thema ab. Hier muss die Position des Nutzers nicht berücksichtigt werden. Eine Anpassung der Ergebnisse kann trotzdem notwendig sein, da ein mobiler Nutzer eine andere Präsentation der Ergebnisse erwartet als von einem Desktop Computer aus, nämlich eine eingeschränktere Anzeige, bedingt durch die eingeschränkteren Ressourcen seines Geräts.

- *Positionsbewusste Anfragen*. Beispiel: Ein mobiler Nutzer fragt nach den Abfahrtzeiten der Fähre von Blankenese nach Cranz am Tag X. Obwohl hier eine

Position berücksichtigt werden muss, ist diese expliziter Bestandteil der Anfrage, es muss also nicht zusätzlich die Position des Nutzers berücksichtigt werden, die Ergebnismenge unterscheidet sich nicht von der einer nicht-positionsabhängigen Anfrage.

- *Positionsabhängige Anfragen* (*location-dependent-queries*). Beispiel: Ein mobiler Nutzer fragt nach Sehenswürdigkeiten in seiner Umgebung. Die Anfragen enthalten nicht explizit die Position des Anfragenden, dennoch muss diese für die Beantwortung berücksichtigt werden.

Im Fall von positionsabhängigen Anfragen ist also zusätzlich zur Anpassung an den Benutzerkontext (also an die technischen Rahmenbedingungen mobiler Geräte) auch eine *inhaltliche* Anpassung der Daten erforderlich. Dieselbe Abfrage („Welche Hotels befinden sich in der Nähe“) wird völlig unterschiedliche Ergebnisse liefern, je nach Position des Anfragenden.
Die **inhaltliche Anpassung** der Ergebnisse in Abhängigkeit einer Position ist eine Operation, die von den Datenbanksystemen durchgeführt werden muss. Dazu verwenden Datenbanksysteme weitere Attribute, die nicht in der Anfrage enthalten sind, nämlich zusätzliche lokationsbezogene Attribute, die mit in die Anfrage einfließen. Dieses Thema wird hier nicht vertieft, da es den Rahmen der Arbeit sprengen würde[49].
Die **technisch bedingte Anpassung**, die sich aus der Beschränktheit der Ressourcen auf mobilen Geräten ergibt, wird im nächsten Abschnitt behandelt.

4.1.2 Technisch bedingte Anpassungen auf mobilen Geräten

Der technische Rahmen von mobilen Geräten erfordert Anpassungen im Ablauf der Transaktionen und in der Darstellung der Ergebnisse:

Neue Transaktionsmodelle

Auf der einen Seite – das wird hier nur kurz erwähnt und nicht vertieft[50] – müssen Transaktionsmodelle für mobile Transaktionen überdacht werden.
Die bewährten ACID-Eigenschaften von klassischen Transaktionen (Atomarität, Konsistenz, Isolation und Dauerhaftigkeit) sind weniger geeignet für leistungsarme Geräte

[49] Dazu sei auf Abschnitte 5.2 und 5.3 in [HöTüKR05] verwiesen.

[50] Ausführlich wird das Thema in Kap 8 von [HöTüKR05] behandelt.

mit wenig stabilen Verbindungen. Auf solchen Geräten setzt man in der Regel die Fehlertoleranz höher, weil der Preis, den man zahlt, um eine Transaktion rückgängig zu machen, oft zu hoch ist. Eine längere Transaktion, die kurz vor Ende ist, wird nicht mehr wegen Verletzung einer Integritätsbedingung abgebrochen, sondern vielmehr nach Möglichkeit zu Ende gebracht, weil ein Neuansetzen zu aufwendig ist und weil andere kürzere Transaktionen dadurch zu lange blockiert würden. Auf solchen Systemen mit beschränkten Ressourcen und längeren Transaktionen, werden die ACID Bedingungen gelockert und mobile Transaktionen entfernen sich im Konzept von den „klassischen" Transaktionen (also von solchen, wie etwa für Versicherungen, Fluggesellschaften usw., bei denen es auf Genauigkeit und Fehlerfreiheit der Daten ankommt und die über umfangreiche Ressourcen verfügen)[51].

Anpassung der Anfrageergebnisse

Zusätzlich zur Anpassung der Transaktionen an die technischen Grenzen der anfragenden Geräte, müssen auch die Anfrageergebnisse selbst angepasst werden.
Die erhaltenen Daten können dem Nutzer nur dann wirklich hilfreich sein, wenn sie die gewünschte Information so darstellen, dass sie möglichst einfach erfasst werden kann. Die optimale Art der Informationsdarstellung kann je nach Art des verwendeten Geräts und je nach Benutzerkontext stark variieren.
Ein Laptop hat ein größeres Display, eine ausführlichere Tastatur und möglicherweise eine bessere Netzverbindung als ein Handy, welches dagegen einen eingeschränkten Anzeigebereich und eine instabile Verbindung besitzt. Im Kontext eines Fahrassistenten im Auto kann es außerdem von großem Vorteil sein, wenn die Information auch auf akustischem Wege präsentiert wird, um weniger Ablenkung zu produzieren.

All dies hat zum Ansatz geführt, dass die Rahmenbedingungen (Gerätetyp, Art der Verbindung, Benutzerkontext) in Form von „Profilen" erfasst werden, die als zusätzliche Information bei einer Anfrage mitgeschickt werden könnten. Ein Beispiel sind CC/PP Profile[52] nach W3C Standard, welche aus einer XML-Beschreibung bestehen.
Die Informationsquelle kann anhand des Profils die Anfrageergebnisse so aufbereiten, dass sie für den Benutzer in der optimalen Darstellung vorliegen.
Dieser wichtige Ansatz hat in dieser statischen Form natürlich auch Nachteile (vor allem eine Redundanz in der Datenbank: die Daten müssen statisch in mehreren Ausführungen zur Verfügung stehen); eine dynamische Anpassung wäre allerdings in den meisten Fällen

[51] Zu „klassischen" Datenbanken und ACID Eigenschaften s. A.Kemper, A.Eickler, *Datenbanksysteme, Eine Einführung*, Oldenbourg 2004.

[52] W3C, Composite Capabilities/Preference Profiles**,** *http://www.w3.org/Mobile/CCPP/*

zu kompliziert, und das führt dazu, dass diese intelligente Aufbereitung der Ergebnisse in der Praxis selten verwendet wird. Das bekannteste Beispiel einer Datenanpassung dürfte die *Wireless Markup Language* (WML) [53] sein, eine XML-basierte Auszeichnungssprache zur Darstellung veränderlicher Inhalte auf mobilen Telefonen, die Teil des WAP-Protokolls[54] ist.

4.2 Datenspeicherung auf mobilen Geräten

Mit *Datenpersistenz* ist die Möglichkeit gemeint, Daten dauerhaft zu speichern, so dass sie auch nach einem Neustart des Geräts zur Verfügung stehen.
In Java und generell Desktop-Anwendungen werden Daten im Dateisystem gespeichert und alle Anwendungen haben darauf Zugriff. Auch Datenbanksysteme speichern ihren Inhalt in (eine oder mehrere) Dateien.
Manchmal bieten sich auch XML-Dateien zur Speicherung von Daten an. Zwar ist XML nicht als Datenbank anzusehen, aber die XML-Syntax bietet für die Verwaltung von Daten eine gute Alternative zu relationalen Datenbanken, vor allem bei semistrukturierten Daten, also solchen, die kein vorgegebenes Datenschema besitzen[55].
Aber ob XML oder Datenbanksystem-spezifische Dateien, ist der Zugriff auf Dateien seitens der Anwendungen in herkömmlichen Systemen und auch unter Java nicht sonderlich problematisch.

Auf mobilen Geräten und insbesondere Telefonen ist der Zugriff seitens der Anwendungen auf das Dateisystem vorwiegend aus Sicherheitsgründen problematisch: Es ist zu gefährlich, wenn jede Anwendung die Möglichkeit bekommt, persistente Daten des Telefons zu verwenden oder zu ändern. Dies, ähnlich wie die Problematik der JNI Unterstützung, ergibt sich auch aus der Tatsache, dass ein Mobiltelefon mehrere Funktionalitäten in einem einschließt und – auch wenn man das manchmal vergisst! – immer noch als Telefon funktionieren muss. Seine Grundfunktionalität muss also gewährleistet bzw. geschützt werden, was auf der anderen Seite eine Einschränkung der Freiheit in der Programmierung darstellt.

---

[53] Ausfühliche Beschreibung in R. P. Korte, WML, *http://www.wml-tutorial.de/* Stand: Okt 2009.

[54] Open Mobile Alliance, WAP Protocol, *http://www.wapforum.org/what/index.htm.*

[55] In relationalen Datenbanken sind Daten stark strukturiert: Die Tabellen bestehen aus gleichartigen Tupeln. Semistrukturierte Daten können dagegen in ihrer Struktur stark variieren, da zum Beispiel nicht alle Elemente alle Attribute besitzen. In diesem Fall ist XML (mit dem eigenen XML-Schema) die flexiblere Lösung.

Auch ist der Speicher von mobilen Geräten besonders beschränkt und folglich wenig geeignet für komplexere Datenbanken und Datenspeicherungssysteme, wie es auf „normalen“ Desktop-Anwendungen üblich ist.
Daher sind andere Lösungen erforderlich: *Die Daten müssen in möglichst einfacher Form im Speicher abgelegt werden und der Zugriff darauf seitens der Applikationen muss geregelt werden.*
Zum einen also darf die Größe der gespeicherten Daten nicht zu hoch sein, was bedeutet, dass die Komplexität und der Umfang der gespeicherten Information reduziert werden muss. Zum anderen soll nicht jede Applikation die Möglichkeit haben, auf alle Daten zuzugreifen, sondern im Idealfall darf jede Applikation nur auf die eigenen Daten zugreifen (öffnen, schreiben, lesen, speichern, schließen). Der Datenaustausch zwischen Applikationen muss allerdings auch möglich sein und dies bedarf besonderer Mechanismen.

4.2.1 Datenpersistenz in J2ME

Für die persistente Datenerhaltung, wird unter MIDP das *Record Management System* (RMS) genutzt[56].
Das RMS abstrahiert den Zugriff auf Datenbanken unabhängig von der tatsächlichen Art der darunterliegenden Speicherkonzepte eines einzelnen Geräts.
Die *javax.microedition.rms* Schnittstelle, über die MIDlets auf RecordStores zugreifen können, besteht aus nur einer Klasse: ***RecordStore***, welche die gesamte Funktionalität für die persistente Datenspeicherung, wie z.B. das Öffnen eines bestehenden oder das Anlegen eines neuen RecordStores, wie auch das Schließen und Löschen zur Verfügung stellt.
Über RMS können einzelne Record-Objekte gespeichert werden, die aus Byte-Arrays bestehen, also Datensätzen mit einem Index (das ist die ID, über die man darauf zugreifen kann) und einer Reihe von Bytes, die die Daten darstellen. Java-Datentypen werden also beim Speichern in Byte-Arrays konvertiert und beim Lesen werden die Bytes in Java-Objekte zurückkonvertiert. Der Umgang mit den Byte-Arrays ist Aufgabe des Entwicklers.
Jedes MIDlet einer Suite kann auf alle zugehörigen RecordStores lesend und schreibend zugreifen und auch neue erzeugen. Wenn das MIDlet entfernt wird, werden auch die zugehörigen RecordStores entfernt.

So wird in J2ME ein RecordStore angelegt (der Parameter *true* bedeutet, dass der RecordStore erzeugt werden soll, falls er nicht existiert):

```
RecordStore rs = RecordStore.openRecordStore( "name", true );
```

[56] S. [Ri08] Kap 6, oder [BrMo06] Kap 6.

Ein Datensatz kann wie folgt gespeichert werden:

```
byte[] input = "Name Nachname| weg 99| 22765 Stadt".getBytes();
int id = rs.addRecord(input, 0, input.length );
```

Jeder Datensatz wird mit einer ID verbunden, welche immer eindeutig ist. Mit dieser ID können die Daten wieder aus dem Speicher geholt werden:

```
byte[] output = new byte[ 100 ];
int bytesRead = rs.getRecord( id, output, 0 );
```

Die Daten stehen anschließend im neuen Buffer und können für die Verwendung wieder in den String konvertiert werden.

Da in RecordStores nur Bytes gespeichert werden und Serialisierung kein Feature von JavaME ist, liegt es am Programmierer, sowohl die Datenstrukturen festzulegen als auch die Serialisierung selbst zu implementieren, dazu können Java Streams verwendet werden (*java.io.DataInputStream* und *java.io.DataOutputStream*).

RecordStores sind eine gute, einfache und effiziente Lösung aber für komplexere Daten nicht immer ausreichend.
Weitere Möglichkeiten, Daten zu beziehen oder zu speichern, ergeben sich über **Netzwerk-Verbindung** oder über den Zugriff auf das File System.

Es gibt Erweiterungen für J2ME, um Daten ins **Dateisystem** zu speichern. Zu Beachten ist hierbei dass viele Hersteller aus Sicherheitsgründen das Schreiben ins Dateisystem nicht erlauben, zumindest nicht bei MIDlets, die nicht signiert sind und folglich nicht als vertrauenswürdig gelten.
Das Verhalten bezüglich des Zugriffs auf Dateien unterscheidet sich also je nach Gerätemodell und wird nicht auf jedem J2ME-fähigem Gerät zuverlässig unterstützt wie die RecordStores. In einigen Fällen wird zwar der Zugriff auf das Dateisystem erlaubt, dies ist aber sehr unschön, da das Gerät an den entsprechenden Stellen immer den Benutzer nach Zustimmung fragt und die Anwendung erst nach einer positiven Antwort weitergeht.

4.2.2 Datenpersistenz auf Android

Android verwendet den J2ME Mechanismus der RecordStores nicht. Es bietet stattdessen grundsätzlich zwei Arten der Datenspeicherung: Speicherung in Dateien oder Verwendung einer einfachen Datenbank (SQLite).
Zusätzlich können Daten über ein Netzwerk ausgetauscht werden, über die Packages *java.net.** und *android.net.**.

Zu Beachten ist, dass Activities, die sich im Zustand *onPaused()* befinden, als „killable“ bezeichnet werden, weil sie von Android bei Bedarf geschlossen werden können, folglich muss die Datenspeicherung spätestens in diesem Zustand erfolgen. Die Methoden *onStop()* oder *onDestroy()* könnten eventuell nicht mehr ausgeführt werden.

Speicherung in Filesystem

Um auf das FileSystem zuzugreifen bietet *android.content.Context* einige Methoden an wie *openFileInput()* und *openFileOutput()*, welche *FileInputStreams* und *FileOutputStreams* zurückgeben.
Aus Sicherheitsgründen funktioniert jede Android Applikation nach dem „Sandkasten-Prinzip“, das bedeutet, dass die Applikation eine in sich geschlossene Einheit bildet und nicht ohne weiteres mit anderen Applikationen interagieren kann; das bedeutet auch, dass eine Applikation keine Dateien schreiben oder lesen kann, die nicht ihr gehören (dies wird geprüft anhand der User-ID, die jede auf dem Gerät installierte Applikation zugewiesen bekommt).
Beim Anlegen einer neuen Datei können Zugriffsrechte dafür vergeben werden. Auch können „Permissions“ über die Manifest-Datei festgelegt werden.
Innerhalb einer Applikation können aber Dateien angelegt und gelesen werden: Die oben genannten Schreib- und Lesemethoden gehören der Klasse *Context*, sind also innerhalb eines bestimmten Kontext (einer Applikation) verwendbar.

Speicherung in die Datenbank

Sehr praktisch ist die Verwendung der Android SQLite Datenbank.
SQLite enthält ein relationales Datenbanksystem (mit Transaktionen, Unterabfragen, Views, Triggers) und ist nur wenige hundert Kilobyte groß; aus diesem Grund ist sie besonders geeignet für eingebettete Systeme wie mobile Telefone. Viele Funktionalitäten

zur Transaktionsverwaltung, die in diesem Anwendungsbereich zweitrangig sind[57], fehlen (vor allem in Bezug auf Koordination von gleichzeitigem Schreib- und Lesezugriff) dafür lässt sich die Datenbank sehr leicht und ohne zusätzliche Software in eine Anwendung integrieren.

Das Package *android.database.sqlite* bietet Klassen und Methoden für die Datenbankverwaltung, d.h. um eine Datenbank zu erzeugen, Daten zu speichern und abzufragen. Wichtige Methoden sind zum Beispiel: `openOrCreateDatabase()` um eine Datenbank anzulegen, `execSQL()` um einen SQL Befehl wie INSERT, DELETE, auszuführen, `query()` für die Datenabfragen.

Die Daten werden auf dem Gerät in: `/data/data/<package_name>/databases` abgelegt.

Das Lesen einer Datenbank erfolgt innerhalb derselben Anwendung über ein ***Cursor***-Objekt. Über einen Cursor ist es möglich, sich durch die Ergebnistabelle einer Abfrage zu bewegen. Man kann dadurch die Tabelle von Zeile zu Zeile durchgehen und man kann auf die Daten zugreifen, indem der Spaltenname bzw. Spaltenindex angegeben wird.

Austausch über Content Providers

Da gespeicherte Daten auch bei Android nur der Applikation zugänglich sind, der sie gehören, kann man für den Datenaustausch zwischen Anwendungen *Content Providers* verwenden[58]. Ein Content Provider ist eine Klasse, die ein Standard-Set an Methoden implementiert, wodurch Applikationen Daten beziehen oder speichern können. Viele Content Provider sind bei Android für den Austausch häufig verwendeter Daten (wie gespeicherte Kontaktdaten, gespeicherte Bilder, Anrufliste, Einstellungen) vordefiniert. Diese werden über die Klassen im Package *android.provider* erreicht.

Weitere Content Providers können beliebig definiert werden (und müssen in der manifest-Datei registriert werden).

Die Abfrage der Daten erfolgt indirekt über *android.content.ContentResolver* Objekte und liefert *Cursor* Objekte zurück.

[57] Vgl. Abschnitt 4.1.2

[58] s. Android Dev Guide, *http://developer.android.com/guide/topics/providers/content-providers.html*

4.3 Beispielanwendungen: Verwaltung von Positionsdaten

Um das Speichern und das Abfragen von Daten auf mobilen Geräten zu demonstrieren (jeweils mit J2ME und Android), realisieren wir eine kleine „Sticky Notes“-Anwendung.
Sticky Notes speichert die Positionsdaten einiger für den User relevanter Orte und erlaubt es, Notizen zu diesen Datensätzen anzulegen. Dann checkt die Anwendung die aktuelle Position des Telefonnutzers. Sobald die aktuelle Position sich einem der gespeicherten Orte nähert, zeigt die Anwendung eine Meldung mit der eingegebenen Notiz auf dem Display an.
Beispielsweise möchte der Nutzer erinnert werden, dass er eine Aufgabe erledigen soll, sobald er in die Firma kommt, also legt er dafür eine Notiz an und speichert sie unter „Arbeit“. Wenn er zur Arbeit kommt, merkt die Anwendung über eine Abfrage der GPS-Position dass man sich an einem Ort befindet, zu dem es eine Notiz gibt und zeigt sie dem Nutzer.

In unserem Beispiel gehen wir davon aus, dass der Benutzer folgende Daten auf dem Gerät speichern möchte:

Ortbezeichnung	**Koordinaten (String)**	**Notiz**
Uni	`53.5989764,9.9326651`	„Sprechstunde nicht vergessen“
zuHause	`53.2150244,10.4232788`	„Lisa anrufen“
Arbeit	`52.3527337,9.7318267`	-

Zum Einen müssen diese Daten persistent gespeichert werden. Zum Anderen müssen sie nach jedem Positionscheck abgefragt und falls benötigt angezeigt werden.

Dazu werden Location Services verwendet, welche die Position kontinuierlich abfragen und nach jeder Ermittlung einer neuen Position diese mit den Gespeicherten vergleichen.
Wenn der Abstand zu einem gespeicherten Koordinatenpaar kleiner ist als eine gewisse Entfernung (Radius) muss das System noch prüfen, ob in dem dazugehörigen Datensatz eine Notiz vorhanden ist.
In dem Fall sind alle Voraussetzungen erfüllt und der Datensatz (bestehend aus der Ortsbezeichnung, welche vom Nutzer für die Position eingegeben wurde und der entsprechenden Notiz) wird auf dem Display angezeigt.

4.3.1 Sticky Notes als Midlet

Unter Verwendung einer einfachen High Level UI, des Mechanismus der RecordStores und der J2ME-Location API kann die oben Beschriebene Anwendung in J2ME realisiert werden (vgl. Abb. 4-1):

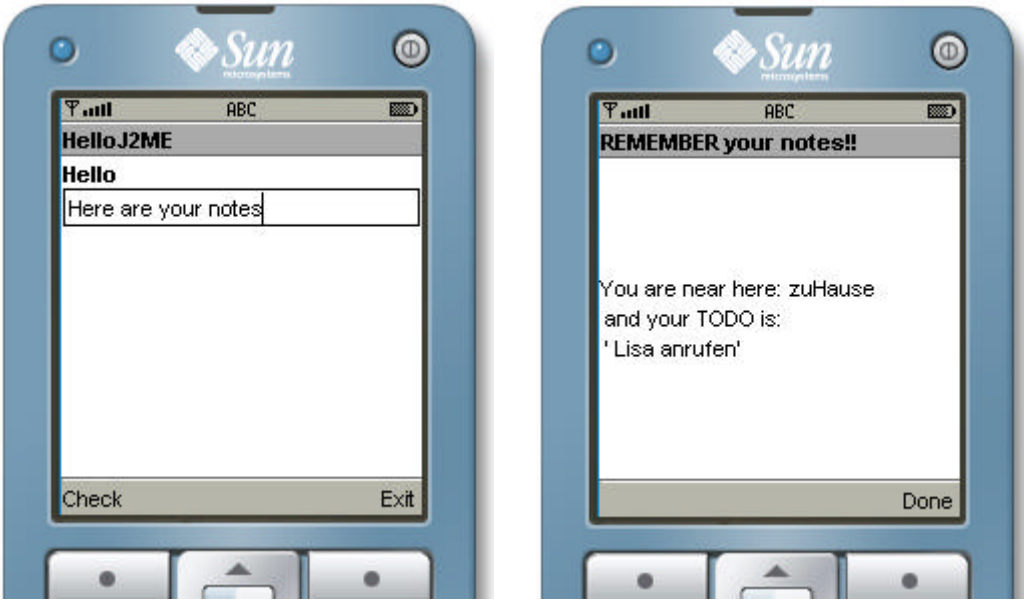

Abbildung 4-1 : J2ME Sticky Notes

Über RecordStores werden die Datensätze gespeichert, wobei hier sofort nachteilig auffällt, dass das oben beschriebene Design der Datenbank nicht erhalten werden kann. Stattdessen werden die Datensätze als String gespeichert, der Bindestrich dient der späteren Verarbeitung (Listing 4.1):

```
private RecordStore rs = null;
static final String REC_STORE = "myplaces";

rs = RecordStore.openRecordStore(REC_STORE, true );

writeRecord("Uni - 53.5989764,9.9326651 - Sprechstunde nicht
vergessen");
```

Listing 4.1 : Speicherung der Daten in RecordStore

Anstatt der Speicherung über RecordStores würde in diesem speziellen Fall die Speicherung über die Klasse *Landmarks* der J2ME Location API bequem dazu dienen. Diese Klasse besitzt Methoden zum Setzen und Auslesen von Adressinformationen, Beschreibung, Name, Koordinaten einer „Marke“, also eines Ortes; Landmarks können in einem *LandmarkStore* verwaltet werden. Diese ist eine wichtige und äußerst nützliche

Funktionalität, welche die Location API von J2ME auszeichnet. Da es hier um die Verdeuchlichung der Speicherungsmechanismen allgemein geht, wurde darauf verzichtet.

Das Midlet zum Abrufen der gespeicherten Daten und zur Anzeige der Notizen ist wie folgt aufgebaut:

Über `display.setCurrent(fmMain)` wird dem Display eine Form hinzugefügt, welche die grafischen Komponenten enthält, wie schon in der GUI Anwendung erläutert (Kap. 3). Darunter werden dem Display zwei Befehle hinzugefügt: Einen zum Beenden der Anwendung und einem zum Starten des StickyNotes Check.
Durch Klicken auf „check“ wird folgende Methode des Midlets aufgerufen (s. Listing 4.2, *checkNotes()*):

```
public void checkNotes() {

  new Thread(new Runnable() {
       public void run(){
            try {
               lp = new  LocPosition();
            } catch (InterruptedException e) {
            writelog("error in checkDB thread lp "+e.toString());
            } catch (LocationException e) {
            writelog("error in checkDB thread lp "+e.toString());
            }
      }
}).start();

readRecords();

for (int j = 0; j < this.results.length; j++)
{

      final String place =
            this.results[j].substring(0, this.results[j].indexOf("-"));
      String coords =
            this.results[j].substring(this.results[j].indexOf("-")+1,
            this.results[j].lastIndexOf('-'));
      final String note =
            this.results[j].substring(this.results[j].lastIndexOf('-
            ')+1, this.results[j].length());
      double dbllatitude =
            Double.parseDouble(coords.substring(0,
            coords.indexOf(",")).trim());
      double dbllongitude =
            Double.parseDouble(coords.substring(coords.indexOf(",")+1,
            coords.length()).trim());
      final Coordinates myCoordinates =
            new Coordinates(dbllatitude, dbllongitude, 0);
```

```
   new Thread(new Runnable() {
      public void run() {
         try
         {
           LocationProvider.addProximityListener(new ProximityListener()
           {
            public void monitoringStateChanged(boolean arg0) {}
            public void proximityEvent(Coordinates arg0, Location arg1)
                  {
                  Alert alert = new Alert("REMEMBER your notes!!");
                  alert.setTimeout(Alert.FOREVER);
                  alert.setString("You are near here: "+ place + "\n and
your TODO is: \n '"+ note +"'");
                  display.setCurrent(alert);
                  }
           }, myCoordinates, 1000.0F);
          }
          catch (LocationException e) {..}
      }
   }).start();
 }
}
```

Listing 4.2 :StickyNotes : Datencheck im Midlet

Als erstes (nachdem der Benutzer zugestimmt hat, die Location Dienste in Anspruch zu nehmen) wird eine Instanz der Klasse LocPosition erzeugt (welche die aktuelle Position liefert und einen LocationListener startet, damit die Position immer aktuell gehalten wird)[59]; dieser Aufruf wird in einem eigenen Thread getätigt, da es kritische Befehle enthält, welche die gesamte Applikation evtl. blockieren könnten.

Der vorhandene Location Objekt muss nun mit den gespeicherten Daten verglichen werden.
ReadRecords() liest die Einträge aus dem RecordStore und speichert sie - so wie sie dem RecordStore entnommen werden - in einen Array namens „results“ (der Code dazu wird weiter unten erläutert) . Durch mehrere und teilweise umständliche Stringoperationen werden die einzelne Elemente aus den Records extrahiert.
Da die RecordStores beschränkte Möglichkeiten der Datenspeicherung bieten und da in J2ME keine Generics verwendet werden können (was für den Array „Result“ äußerst praktisch wäre[60]), sind die vielen Stringoperationen notwendig; dadurch wird der Code komplexer und folglich fehleranfälliger.

[59] Diese Klasse ähnelt der schon vorgestellten *LocPosition* Klasse aus Listing 3.2 und wir hier nicht erneut gezeigt.

[60] Vgl. dazu die Verwendung von `ArrayList<String> results` in der Android Anwendung, Abschn. 4.3.2

Am Ende der Stringoperationen sind für jeden Eintrag aus dem recordStore die einzelnen Elemente (Ort, koordinaten und Notiz) einzeln vorhanden und aus den Koordinaten wird ein Coordinates-Objekt erstellt (`myCoordinates`) .
Diese Koordinaten sind mit den aktuellen Koordinaten zu vergleichen.
Dazu kann der ProximityListener verwendet werden (wieder in eigenem Thread gestartet).
Initialisiert wird der ProximityListener mit

- einem neuen Listener (Inner Class), in dem das Verhalten im *proximityEvent*-Fall mit dem gewünschten Verhalten überschrieben wird (die Meldung mit der Notiz soll ausgegeben werden)
- den aktuellen Koordinaten aus dem RecordStore-Eintrag, die mit der aktuellen Position verglichen werden müssen
- einem Radius in Metern: innerhalb dieses Radius gilt die Position als „in der Nähe des gesuchten Objektes“.

Der Listener arbeitet nun im Hintergrund und sobald sich die Koordinaten im besagten Radius um einem gespeicherten Eintrag herum befinden, erscheint die jeweilige Meldung aus Abb. 4-1.

Zum Einlesen des RecordStores enthält die Methode *readRecords()* folgenden Code (Listing 4.3):

```
byte[] recData = new byte[10];
int len;
for (int i = 1; i <= rs.getNumRecords(); i++)
{
      if (rs.getRecordSize(i) > recData.length)
      recData = new byte[rs.getRecordSize(i)];
      len = rs.getRecord(i, recData, 0);
            //record as String:
      String oneEntry = new String(recData, 0, len);
            //extract last part (only the note):
      String note = oneEntry.substring(oneEntry.lastIndexOf('-')+1);
            //insert entry in result-string only if note is not empty:
      if (note.trim()!=""){
                  this.results[i-1] = oneEntry;
      }
      System.out.println("Record#"+i+": "+new String(recData,0,len));
}
rs.closeRecordStore();
```

Listing 4.3 : StickyNotes Lesen aus dem RecordStore

Das Fazit dieser Beispielanwendung ist sehr aufschlußreich: Auf der einen Seite beeindruckt die Location API von J2ME mit ihren ausgefeilten Möglichkeiten. Auf der anderen macht die Arbeit mit den RecordStores, besonders unter dem Verzicht auf Generics, einen besonders großen Aufwand beim Verarbeiten der gespeicherten Daten, worum es in diesem Kapitel geht.

4.3.2 Sticky Notes mit Android

Eine einfache StickyNotes Anwendung für Android besteht aus 2 Activities und einem Service für die Ermittlung der Position.
Die Daten, die aus Benutzereingaben stammen, werden zunächst in die SQLite Datenbank gespeichert. Dies kann über *execSQL* mit der gewohnten SQL-Syntax geschehen (Listing 4.4).

```
SQLiteDatabase myDB = openOrCreateDatabase(MY_DATABASE_NAME,
MODE_PRIVATE, null);

myDB.execSQL("CREATE TABLE IF NOT EXISTS " + MY_DATABASE_TABLE+ "
(id integer AUTO_INCREMENT PRIMARY KEY, myname varchar(100) ,
coordinates varchar(100), annotations varchar(255))" +";");

myDB.execSQL("INSERT INTO " + MY_DATABASE_TABLE + " (myname,
coordinates, annotations)" + " VALUES ('uni',
'53.5989764,9.9326651', 'sprechstunde nicht vergessen');");
```

Listing 4.4 Android: Speicherung von Datensätzen in SQLite Datenbank

Wenn das Gerät sich später in der Nähe eines gespeicherten Koordinatenpaars befindet und für diese Koordinaten eine Notiz in der Datenbank existiert, wird ein Alert-Fenster mit der Notiz angezeigt (Abb. 4-2):

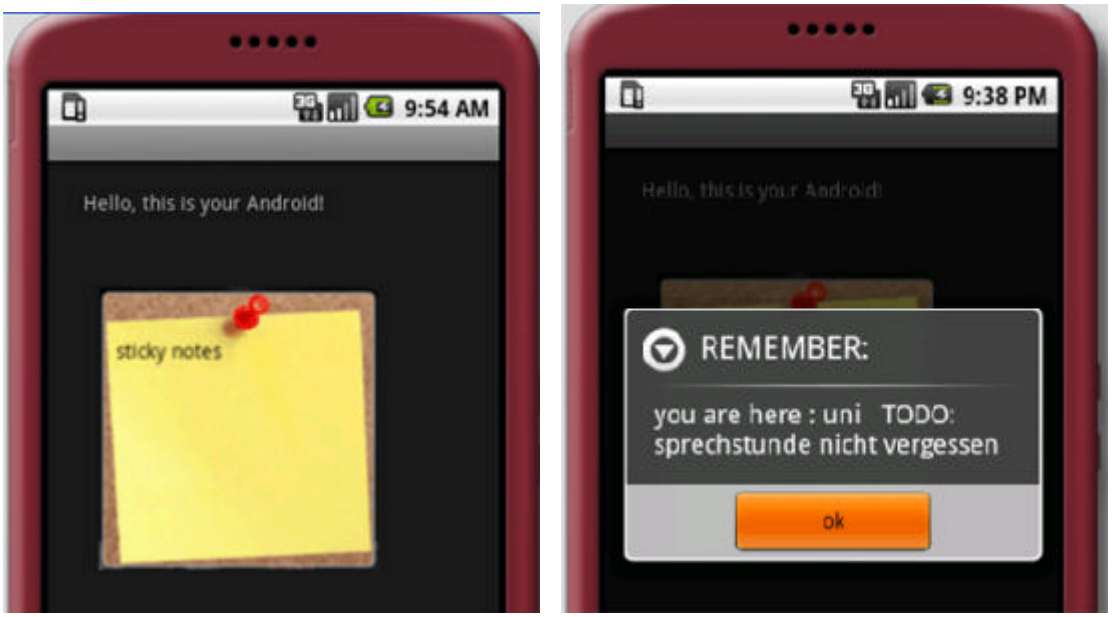

Abbildung 4-2 Android: Sticky Notes

Die **erste Activity** definiert ein Layout, welches die benötigten Elemente enthält (Text und Bild).
Dann startet diese erste Activity über einen *Intent* (s. Beispiel im Abschnitt 2.4.2) eine **zweite Activity** namens „CheckLocationAndNotes“, welche die ganze Arbeit macht: Position erfragen und Datenbank prüfen.

Der Quellcode dieser zweiten Activity wird im Folgenden schrittweise erläutert.

Bei *onCreate()* wird der Location Service gestartet, um die aktuelle Position ermitteln zu lassen, die für die weitere Berechnungen benötigt wird (Listing 4.5).

```
public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
       LocationService ls = new LocationService();
       String location = ls.getCurrentLocation(this);
       checkNotes(location);
    }
```

Listing 4.5 Android: Sticky Notes – Start der Activity

Zu beachten ist, dass Android eine eigene Location API[61] verwendet, die sich sehr von der schon ausgereiften J2ME Location API unterscheidet und deutlich eingeschränkter ist.
Untem anderem kennt Android keine *Landmarks*, deswegen muss die Speicherung der Ortsdaten in die Datenbank manuell erfolgen und deswegen kann der *ProximityListener* nicht verwendet werden.
Im diesem Gebiet hat J2ME Android noch einiges voraus, denn Landmarks und ProximityListeners sind Funktionalitäten, die auf mobilen Geräten häufig benötigt werden und es ist äußerst praktisch, wenn sie schon als vorgefertigte Bausteine zur Verfügung stehen. Aus dem Grund sind schon einige Lösungen entstanden, um Android um diese fehlende Funktionalität zu erweitern: J2ME-Polish[62] definiert zum Beispiel im Package *de.enough.polish.android.location* das fehlende aus J2ME bekannte Interface (*Proximity-Listener*) und die fehlenden Klassen: *Criteria*, *Landmark*, *LandmarkStore*, *QualifiedCoordinates* usw. In dieser Anwendung wird aber nur die Android API verwendet und Speicherung der Ortsmarken sowie die Berechnung der Entfernung zu den gespeicherten Koordinaten müssen hier manuell realisiert werden.

[61] Das Package *android.location* besteht aus 2 Interfaces und 8 Klassen. Vgl. Android Referenz: *http://developer.android.com/reference/android/location/package-summary.html*

[62] j2mepolish, Erweiterung der Location API für Android:
http://www.j2mepolish.org/javadoc/j2me/de/enough/polish/android/location/package-summary.html

Weiter geht es mit der Methode *checkNotes()* in Listing 4.6:

```
SQLiteDatabase myDB2 = null;
final static String MY_DATABASE_NAME = "myLandmarks";
final static String MY_DATABASE_TABLE = "myPlaces";
ArrayList<String> results = new ArrayList<String>();


private void checkNotes(String currentLocation) {

      myDB2 = this.openOrCreateDatabase(MY_DATABASE_NAME,
MODE_PRIVATE, null);
      Cursor c = myDB2.query(MY_DATABASE_TABLE, new
String[]{"myname", "coordinates", "annotations"}, null, null,
null, null, null);

      int dbplaceColumn  = c.getColumnIndex("myname");
      int dbcoordsColumn  = c.getColumnIndex("coordinates");
      int dbnotesColumn  = c.getColumnIndex("annotations");

      startManagingCursor(c);
      c.moveToFirst();

      if (c != null) {
        if (c.isFirst()) {
          do {
              String dbplace = c.getString(dbplaceColumn);
              String dbcoords = c.getString(dbcoordsColumn);
              String dbnotes = c.getString(dbnotesColumn);
              if (isInScope(dbcoords, currentLocation)){
                 results.add(dbplace + " - " +dbcoords+" -
"+dbnotes);
              }
           } while(c.moveToNext());
        }
      }


       for (int r=0; r<results.size(); r++){

            String oneResult = results.get(r);
            String[] details = oneResult.split("-");
            String note = details[2];
            if (!note.equals("")){
                  displayNotes(details, c);
            }
       }
 }
```

Listing 4.6 Android: Sticky Notes – Methode checkNotes

Die Methode *checkNotes()* hat die Aufgabe, die erhaltene aktuelle Position mit den Einträgen der Datenbank zu vergleichen und ggfs. die Notiz ausgeben, falls vorhanden.

Nachdem die Datenbank geöffnet worden ist, werden die Daten mittels eines Cursors gelesen. Die Abfrage benötigt den Tabellenname und die Spalten, die gelesen werden sollen.
Der Index der jeweiligen Spalte wird in eine Integer-Variable gespeichert und kann dann verwendet werden, um den Inhalt der Spalte auszulesen:
`Cursor.getString(columnIndex);`
Diese Operation wird in einer Schleife für alle Zeilen der Tabelle wiederholt, beginnend mit der ersten Zeile (`c.moveToFirst()`) und solange, bis es Nachfolgezeilen gibt (`while (c.moveToNext())`).
In der Schleife muss man sich nur die Ergebnisse der Tabellenabfrage merken, dessen Koordinaten in einer gewissen Entfernung zu den aktuellen Koordinaten liegen. Die Ergebnisse können bequem in ein ArrayList von Strings gespeichert werden.
Wenn die Datenbank komplett durchgesucht und die ArrayList gefüllt worden ist, braucht man nur noch einmal über diese Ergebnismenge zu iterieren und eine Meldung auszugeben, falls eine Notiz dafür vorhanden ist.

Zwei Methoden aus Listing 4.6 werden hier nicht dargestellt, aber der Vollständigkeit halber in Folgendem beschrieben:

Die Methode *displayNotes()* gibt das AlertFenster mit dem gewünschten Inhalt als String aus. Stattdessen hätte man die Daten auch auf dem aktuellen Display (mit beliebiger Formatierung) darstellen können, unter Verwendung eines `SimpleCursorAdapters` und der dazugehörigen `setViewBinder()`Methode, die Daten aus einer Datenbankabfrage in eine View einbindet. Hier bietet Android viele schöne Möglichkeiten.
Da in diesem Fall in der Regel nur ein Eintrag vorhanden sein wird (nur ein Ort trifft zu und wahrscheinlich nur eine Notiz ist dafür gespeichert), ist das AlertFenster die passendste Darstellungsart, auch weil es die Aufmerksamkeit des Nutzers besser auf sich lenkt.

Die Methode *isInScope()* hat die Aufgabe, die Nähe zwischen den gespeicherten Koordinaten für einen besonderen Ort mit den Koordinaten aus der Positionsabfrage zu berechnen. D.h. sie gibt *true* zurück, falls die gespeicherten Koordinaten innerhalb eines Radius (oder vereinfacht: Rechteck) um die aktuell ermittelten Koordinaten liegen. In dem Fall befindet man sich in der Nähe des gespeicherten Ortes und es kann die Notiz dafür ausgegeben werden.
Das was in J2ME der ProximityListener bequem für den Entwickler erledigt (man braucht nur einen Radius anzugeben) muss bei Android manuell programmiert werden, was in diesem Fall nicht trivial ist, weil es um das Rechnen mit Geokoordinaten geht. Das ist definitiv ein Nachteil.

5 Zukunftsperspektiven

JavaME ist zu ihrer Entstehungszeit ein wichtiger Schritt gewesen und bietet insgesamt eine gute Grundfunktionalität, wenn man die vielen Einschränkungen in Kauf nimmt. Vor allem die Location API ist besonders ausgereift und effektiv.
Die GUIs sind ein wenig überholt, obwohl durch die LowLevel UI relativ gute Grafikergebnisse (zum Beispiel für 2D-Spiele) erhalten werden können.
Das Record Management System von J2ME ist für einfache Daten ein guter, praktikabler Weg, aber in vielen Fällen ist es deutlich zu umständlich und kann mit einer Datenbank nicht mithalten.
Die Einschränkungen von J2ME werden in der Tat immer schwerwiegender mit der zunehmenden Leistungsfähigkeit der Zielgeräte und mit den wachsenden Ansprüchen der Benutzer. Das ist ein Grund, warum zurzeit alternative Lösungen auf dem Markt erscheinen, darunter Android.
Sun hat angekündigt, die Weiterentwicklung von JavaME nicht weiter verfolgen zu wollen, da die neueren Geräte bald völlig in der Lage sein werden, JavaSE zu unterstützen[63].
Es ist gut möglich, dass die Zukunft für Java auf mobilen Geräten aus JavaSE plus Zusatzbibliotheken etwa für GPS, SIP[64] und Bluetooth besteht.

Android bietet dagegen eine innovative Lösung, die neue Möglichkeiten eröffnet, für die es momentan viel Nachfrage gibt. Auch der ästhetische Aspekt spielt hier eine große Rolle: Android bietet Unterstützung für 3D-Grafiken und sehr moderne und anschauliche Layout-Möglichkeiten. Auch die angebotene SQLite Datenbank ist ein großer Vorteil, der die Programmierung deutlich erleichtert.
Hier und da wird deutlich, dass Android noch nicht einen endgültigen Stand erreicht hat und dass noch Verbesserungsbedarf besteht, gleichzeitig ist auch die Möglichkeit zur Verbesserung gegeben, zum einen aufgrund des robusten Konzeptes selbst, zum anderen aufgrund der Tatsache, dass Android ein Open Source Projekt ist, an den sich viele Entwickler weltweit interessieren und mitarbeiten. Die von Google eingeführte Android Developer Challenge[65], mit hohen Preisgeldern für die besten Anwendungen hat sicher dazu beigetragen, dass die Entwicklerwelt auf Android aufmerksam geworden ist.

---

[63] Das kündigte James Gosling (Sun's Vizepräsident) in einem Interview, wie in diesem Artikel zu lesen: cnet news, Sun starts bidding adieu *http://news.cnet.com/8301-13580_3-9800679-39.html* (Artikel von 10.2007, besucht im Oktober 2009).

[64] *Session Initiation Protocol* zur Abbau und Steuerung einer Kommunikationssitzung

[65] Google Inc., Android Developer Challenge: *http://code.google.com/intl/de-DE/android/adc/*

Auch wenn Android einen eigenen Bytecode verwendet, Tatsache bleibt, dass Android in Java programmiert wird (und zwar fast mit der vollständigen JavaSE API). Dadurch hat Android ein großes Entwicklungspotential aufgrund der Vielzahl von verfügbaren Java-Programmierern. Sein Vorgänger, der iPhone[66], erfreut sich aufgrund seiner „Verschlossenheit“ weniger Beliebtheit[67], trotz der großen Resonanz auf dem Markt[68].

Umstritten ist bei Android die Tatsache, dass es sich um ein neues System handelt, welches sich stark von den bisherigen Standards für mobile Geräte entfernt (vorwiegend J2ME). Solche Lösungen sind in der Regel problematisch, weil die Einhaltung von Standards und die Portabilität ein wichtiger Faktor gerade auf dem mobilen Markt ist, in dem man ohnehin mit den Problemen der Vielfältigkeit und der Kompatibilität zu kämpfen hat.
Viele betrachten deshalb Android kritisch, u.a. der Entwickler Hari Gottipati, der bemerkt: *„The mobile environment is already fractured. Even with J2ME, he has to alter his applications for different phones. But in that case, as a developer, I'm porting once and maybe tweaking for different phones. But with this [Android], you'll need to develop a separate application that's not standard. Unless Android becomes mainstream and kills J2ME ... why should I develop applications that are not standard which I'm not sure about? “*[69]
Oder der Telekommunikations-Experte Ofir Leitner: *“don't get me wrong, I believe that iPhone and Android are both great and promising platforms that open new possibilities for mobile developers, and we can already see its effects on the platforms market. There's nothing like a competitor "breathing on your neck" to get you finally going faster... But: Don't focus all your energy there. Pay attention to the platforms that are currently in the hands of your users, and that in spite of how things look like now, will probably stay there, at least enough to make you get used to them...”*[70]

Noch ist Android also nicht ganz etabliert, die Voraussetzungen dafür sind aber gegeben, wie diese Studie insgesamt ergeben hat. Android ist einfach komfortabel zu programmieren, überzeugt mit seinem Architekturkonzept und man kommt damit auf sehr schöne Ergebnisse.

---

[66] Apple, iPhone Homepage: *http://www.apple.com/de/iphone/*

[67] Gern wird der iPhone als verschlossen/unzugänglich für Programmierer bezeichnet, z.B. mit diesem schönen Satz: „a self-contained, jewel-like masterpiece locked in a sleek protective shell“ (*http://www.wired.com/techbiz/media/magazine/16-07/ff_android?currentPage=2*)

[68] s. handy.com, iPhone 3G bereits eine Million Mal verkauft (abgerufen im Oktober 2009): *http://www.handy.com/news/6204/iphone-3g-s-bereits-eine-million-mal-verkauft.html*

[69] Quelle: Infoworld, Google and Sun may butt heads over Android (Stand: Oktober 2009) *http://www.infoworld.com/t/applications/google-and-sun-may-butt-heads-over-android-294*

[70] Quelle: Next Generation Mobile Content: The battle of the mobile plattforms, *http://www.nextgenmoco.com/2008/07/battle-of-platforms.html* (Stand: Oktober 2009).

J2ME wirkt dagegen langsam überholt, bleibt dennoch wichtig wegen der tatsächlichen Verbreitung auf dem Markt.
Dennoch: Maßgeschneiderte Lösungen wie der iPhone oder Android, die nur für Geräte mit gewissen Voraussetzungen gedacht sind und ein komplettes eigenes System definieren, scheinen den Trend zu bestimmen, einfach weil sie mehr bieten können und zuverlässiger sind. Dazu hat Android als Open Source Projekt die besten Perspektiven. Entscheidend für sein Erfolg wird aber letztendlich die tatsächliche Unterstützung seitens der verschiedene Gerätehersteller sein. Und die sieht vielversprechend aus.

Literaturverzeichnis

Sun Inc, *The Java ME Platform* – http://java.sun.com/javame/

Ri08 R. Rischpater: *Beginning Java ME Platform*. Apress 2008

BrMo06 U. Breymann, H. Mosemann, *JavaME*, Hanser 2006

Sun Inc, *Java ME Technology: APIs and Docs* http://java.sun.com/javame/reference/apis.jsp

EclipseME Installation Guide - http://eclipseme.org/docs/installation.html

Me08 R. Meier, *Professional Android Application Development* , Wrox Press 2008

Google Inc, *Android Dev Guide* - http://developer.android.com/guide/index.html

D. Roth, *Google's Open Source Android OS will free the wireless web*, http://www.wired.com/techbiz/media/magazine/16-07/ff_android

Ul09 C. Ullenboom, *Java ist auch eine Insel*, 8. Auflage, Galileo Computing 2009

Java Community Process, http://www.jcp.org/en/home/index

HöTüKR05 Höpfner, Türker, König-Reis, *Mobile Datenbanken und Informationssysteme*, dpunkt 2005

ScVo04 J.Schiller, A.Voisard, *Location-based services*, Elsevier 2004.

Abbildungsverzeichnis

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Hamburg, den 26.10.2009

[Cristina Mossoni]