Fachhochschule Nordwestschweiz
Hochschule für Technik

Institut für Mobile und Verteilte Systeme

FOKUS REPORT

Android
Einblick in die Dalvik Virtual Machine

Seite 5

Lokalisierung
GPS-freie Lokalisierung mittels WLAN

Seite 13

Highspeed Kameras
Innovative Bildaufbereitung in Hochgeschwindigkeitskameras

Seite 34

2009

9 771662 201005 95

Editorial

Schuhe sind für Fussgänger ein fast alltägliches Hilfsmittel für Mobilität. Sie stehen damit für Fortschritt im eigentlichen Sinne. Doch Schuhe repräsentieren weit mehr. Konsequenterweise ist die Wahl der richtigen Schuhe selten nur eine funktionale Abwägung und ganz andere Fragen drängen sich auf: Welche passen und fühlen sich gut an? Womit erziele ich am meisten Aufmerksamkeit? Wie bringe ich meinen Typ am besten zur Geltung? Wie viel Paare sind notwendig und wie viel wünschenswert? Das Angebot ist riesig und es wird nicht erstaunen, dass bei unterschiedlicher Priorisierung der Anforderungen ganz verschiedene Vorschläge in den Vordergrund rücken. Kurzum: Die Qual der Wahl! Warum nur ein einziges Paar Schuhe, das kaum allen Anforderungen genügen wird, wenn man mit mehreren agiler und souveräner voranschreitet? Oder vielleicht doch lieber auf Alltagsschuhe fokussieren anstatt mehrschichtig auftreten und sich dabei gar verlaufen?

Die letzten Jahre haben gezeigt, dass unser Institut auf dem richtigen Weg ist. Diesen Weg gehen wir weiter, werden dabei aber noch deutlicher zum Ausdruck bringen, worin unsere Kompetenzen liegen und für welche Forschungs- und Dienstleistungsprojekte wir die geeigneten Ansprechpartner sind. Daher treten wir neu unter dem Motto „Effizienz und Mobilität in der Informatik“ auf. Die nachfolgenden Kurzbeschreibungen unserer vier Fachbereiche verdeutlichen dieses Motto und kreisen den Wirkungskreis ein.

Im Fachbereich *„Mobile Software-Systeme“* geht es um die Entwicklung von Applikationen, die den Zugang zu Diensten und Informationen zu jeder Zeit, an jedem Ort und in verschiedensten Formen ermöglichen. Zu den bekannten Erwartungen an ein stationäres Software-System kommen zusätzliche Ansprüche und Schwierigkeiten hinzu, wie zum Beispiel der permanente drahtlose Netzzugang oder die stark verkleinerte Benutzerschnittstelle mit reduzierten Eingabemöglichkeiten. Als interessante Informationsquelle sehen wir vor allem den Gerätekontext, welcher mit wegweisender, intelligenter Software gewinnbringend eingesetzt werden kann.

Mobile, aber auch herkömmliche Software sollte robust, flexibel, intuitiv und kostengünstig sein. Deren Entwicklung ist nach wie vor sehr aufwendig, fehleranfällig und komplex. Mit den heutigen Werkzeugen und Methoden kann der zunehmenden Komplexität und der lange Lebenszeit jedoch nur ungenügend begegnet werden. Dies führt dazu, dass die Qualität von Software über deren Lebenszeit stetig abnimmt. Daher setzen sich unsere Mitglieder des Fachbereichs *„Effiziente Soft-*

Inhalt

Editorial 3

Einblick in die Dalvik Virtual Machine 5

Lokalisierung mittels WLAN 13

Erweiterung eines Webdienstes um einen mobilen Webzugang 21

Reverse Generation and Refactoring of Fit Acceptance Tests for Legacy Code 26

ICT System und Service Management – eine Disziplin im Wandel 31

Innovative Bildaufbereitung in Hochgeschwindigkeitskameras 34

Impressum

Verlag:
Fachhochschule Nordwestschweiz FHNW
Institut für Mobile und Verteilte Systeme
Steinackerstrasse 5
CH-5210 Brugg-Windisch
www.fhnw.ch/technik/imvs
Tel +41 56 462 44 11

Kontakt:
Marc Dietrich
info-imvs@fhnw.ch
Tel +41 56 462 46 73
Fax +41 56 462 44 15

Redaktion: Prof. Dr. Christoph Stamm
Layout: Thekla Müller-Schenker
Erscheinungsweise: jährlich
Druck: jobfactory Basel
Auflage: 250

ISSN: 1662-2014

ware-Entwicklung" mit offenen Fragestellungen der Software-Architektur, der -Entwicklung und der -Qualitätskontrolle auseinander.

Die Verarbeitung von datenintensiven, multimedialen Inhalten macht auch vor mobilen Geräten nicht Halt. Spätestens wenn solche Software aus Gründen erhöhter Portabilität oder einer effizienteren Entwicklung bzw. Wartung in ihrer Ausführung ineffizient wird, ist es an der Zeit, sich mit Methoden zur Erhöhung der Effizienz zu befassen. Im Fachbereich *„Effiziente und Parallele Software"* werden Performanzkiller in Software aufgespürt und durch geeignete Massnahmen wie bessere Algorithmen oder Parallelisierung eliminiert.

Mobile und verteilte Systeme sind auf eine stabil und sicher funktionierende Vernetzung von Servern und Endgeräten angewiesen. Damit diese Informatik- und Kommunikationsinfrastruktur zuverlässig, wirtschaftlich und nachhaltig arbeitet, braucht es viel Wissen über das richtige Aufbauen, Betreiben und Pflegen von ICT. Der Fachbereich *„ICT System- and Service-Management"* befasst sich mit allen technischen, organisatorischen und prozeduralen Aktivitäten für die optimierte Verwaltung komplexer, verteilter und meist inhomogener ICT-Infrastrukturen.

Wir sind überzeugt, mit diesen vier Fachbereichen nach aussen hin Transparenz geschaffen zu haben, ohne uns intern abzugrenzen. Bitte nutzen Sie diesen direkteren Einblick und messen Sie uns am angewandten Fortschritt!

Die Qual der Wahl fordert uns nicht nur bei einem Schuhkauf, sondern auch bei der Auswahl eines passenden mobilen Gerätes: Soll es ein bewährtes Symbian-Gerät sein, oder ist der einfache Datenaustausch zwischen verschiedenen Windows-Geräten wichtig, oder steht das Design eines trendigen iPhones im Zentrum, oder will man doch lieber mit einem hippen Android-Gerät beeindrucken?

In unseren Projekten lernen wir diese Vielfalt immer wieder neu kennen. So zum Beispiel schaut Carlo Nicola den Android-Geräten etwas genauer unter die "Haube" und studiert die dort verwendete Dalvik VM unter dem Aspekt der Effizienz. Mit dem Ziel, höchstmögliche Portabilität von Anwendungen zu ermöglichen, gehört der breite Einsatz von virtuellen Maschinen spätestens seit Java zum guten Ton der Informatik. Längst ist aber bekannt, dass Java VMs nicht das wahre Allheilmittel sind. Daher haben die Entwickler von Android andere Ansätze gewählt, um die Effizienz zu erhöhen. Eine Just-in-Time-Kompilation ist jedoch nicht dabei, was den Autor zur Aussage verleitet, dass der Verzicht darauf ein Fehler gewesen ist.

Den mobilen Geräten unter die „Haube" zu schauen ist nicht immer einfach. Im Gegenteil, oft versuchen die Geräte- bzw. Betriebssystemhersteller hardwarenahe Funktionen zu verbergen. Windows Mobile ist hierbei eine löbliche Ausnahme und bietet sich daher für innovative Ideen an. Beat Walti und Christoph Stamm zeigen in ihrem Bericht auf, wie unter Einbezug von WLAN-Signalstärken und Referenzmessungen eine Positionsbestimmung ohne GPS auf wenige Meter genau möglich ist. Für viele kontextbasierte Anwendungen, z.B. aus dem Bereich des Social Networkings, ist eine solche Genauigkeit ausreichend.

Wer mit seinem sozialen Netzwerk das nächste Treffen planen will, während er bereits unterwegs ist, hat nun die Möglichkeit, den bekannten Doodle-Dienst auch auf seinem mobilen Telefon zu benutzen. Jürg Luthiger beschreibt, wie es zur mobilen Webapplikation gekommen ist und welche Anpassungen gegenüber dem Basisdienst notwendig geworden sind. Er erläutert wichtige Entscheidungen in der Systemarchitektur damit auch zukünftige Anpassungen den Benutzerbedürfnissen und den Vorgaben des Basisdienstes gerecht werden.

Die Überprüfung der Tauglichkeit einer Software erfolgt meistens mittels Testens. Vor allem bei Systemen, die im Bau oder Umbau sind, spielt das Testen eine zentrale Rolle zur Aufrechterhaltung der Codequalität. Der Umbau führt aber nicht selten auch zu einer notwendigen Anpassung des Testcodes, was schliesslich in doppelter Arbeit mündet. Daher sind Werkzeuge gefragt, die Testcode und -daten automatisch erzeugen und bei Veränderungen des Quellcodes auch anpassen können. Das von den Autoren Martin Kropp und Wolfgang Schwaiger entwickelte Werkzeug iTDD ist ein guter erster Schritt in diese Richtung.

Im Artikel „ICT System und Service Management – eine Disziplin im Wandel" geht Hannes Lubich auf die wirtschaftliche Bedeutung ständig verfügbarer, zuverlässiger und sicherer ICT-Infrastrukturen ein. Der Tatsache bewusst, dass noch einige Verbesserungen im Aufbau, Betrieb und Pflege solcher Systeme bezüglich Komplexitätsreduktion, Steigerung der Kosteneffizienz und Verbesserung der Nachhaltigkeit möglich sind, zeigt der Autor auf, wo angewandte Forschung und Lehre die richtigen Hebel in Bewegung setzen muss.

Im Schlussbeitrag dieses Heftes zeigt Martin Schindler, dass auch in sehr kurzer Zeit etwas Schönes entstehen kann. Mit seiner neuen Software entstehen schön ausgeleuchtete und von Artefakten befreite Farbbilder direkt in einer Hochgeschwindigkeitskamera. In typischen Bildsensoren mit vorgeschalteter Bayer-Maske müssen pro Bildpunkt zwei fehlende Farbanteile durch umliegende Bildpunkte interpoliert werden. Der Autor hat verschiedene Interpolationsalgorithmen getestet und einen neuen Algorithmus entwickelt, der störende Moiré-Effekte effizient reduziert.

Prof. Dr. Christoph Stamm
Forschungsleiter IMVS

Einblick in die Dalvik Virtual Machine

Die Dalvik Virtual Machine ist der eigentliche Motor des Android-Systems. Neben den Unterschieden gegenüber der Standard Java Virtual Machine werden wir drei Themen genauer betrachten: (i) Die Rolle des Programms Zygote beim Starten der Dalvik Virtual Machine; (ii) die Rolle der Registerstruktur der Dalvik Virtual Machine auf die allgemeine Performanz des Android-Programmierungsmodells und (iii) der Einfluss des neuen Class-File-Formats von Dalvik auf die Effizienz der virtuellen Maschine.

Carlo U. Nicola | carlo.nicola@fhnw.ch

Android ist ein Betriebssystem und auch eine Software-Plattform für mobile Geräte wie Smartphones, Mobiltelefone und Netbooks. Es basiert auf dem Linux-Kernel 2.6 und wird von der Open Handset Alliance entwickelt, wobei ein grosser Teil der Software frei und quelloffen ist [Wiki]. Die Dalvik Virtual Machine (DVM) ist der eigentliche Motor des Android-Systems. Sie ist von Anfang an für anspruchsvolle Java-Applikationen (Version 5.0 bzw. 1.5) konzipiert worden, mit dem Ziel, diese Anwendungen auf extrem bescheidenen Hardware-Ressourcen möglichst effizient laufen zu lassen. Die bescheidenen Ressourcen lassen sich wie folgt zusammenfassen: eine CPU mit maximaler Taktfrequenz von 500 MHz; 64 Megabyte Speicher, von denen nur 20 Megabyte für Applikationen zur Verfügung stehen; ein Linux Betriebssystem ohne *swap space* und das ganze Gerät nur mit Batterien betrieben. Die für die Entwicklergemeinde jedoch wichtigste Randbedingung ist, dass alle Android-Applikationen mit einem gewöhnlichen Java SDK 5.0 geschrieben werden dürfen. Diese letzte Forderung wurde auch in der Tat umgesetzt, obschon Java Bytecode nicht direkt auf der DVM lauffähig ist. Java Bytecode wird für die Dalvik Virtual Machine speziell kompiliert und in einer Dex-Datei (Dateinamenserweiterung .dex) abgelegt.

In den folgenden Abschnitten werden wir den Fragen nachgehen, wie sich die die Dalvik Virtual Machine von der Standard Java Virtual Machine unterscheidet, welche Rolle die Registerstruktur der DVM auf die allgemeine Performanz des Android-Programmierungsmodells hat und wie sich das neue Bytecode-Dateiformat (Dex-Format) auf die Effizienz der DVM auswirkt.

Starten von Android-Anwendungen

Bevor wir uns dem Starten von Android-Anwendungen zuwenden, werfen wir zuerst einen kurzen Blick auf das generelle Startprozedere beim Ablauf des Android *Boot*-Prozesses:

1. Der *init.rc*-Prozess startet verschiedene *daemons*.
2. Anschliessend wird der Service Manager und Zygote aufgerufen. Zygote wird im weiteren Systemablauf dafür sorgen, dass jede Android-Anwendung als normaler Linux-Prozess über den bekannten *fork*-Mechanismus gestartet wird.
3. Der System-Manager stellt die Verbindung mit dem Linux-Kernel her, damit die Dienste, welche via Kernel-Treiber auf die Hardware zugreifen, später auch den Android-Anwendungen via API zur Verfügung stehen.

Abbildung 1 zeigt diese drei Schritte schematisch und schlägt auch die Brücke zum Start der eigentlichen Android-Anwendungen. Wie bereits erwähnt, sorgt der Prozess Zygote dafür, dass eine Android-Anwendung als normaler Linux-Prozess mittels *fork()* gestartet wird. Gleichzeitig wird zu jedem Prozess eine DVM mit eigenem Heap und Stack gestartet und die notwendigen Bindungen zu den Java *Core Libraries* hergestellt. Dies hat verschiedene Vorteile, die sich direkt in der Struktur der DVM widerspiegeln. Zum einem ist das Problem der Sicherheit der Android-Java-Applikation (inbegriffen Byte Code Verifizierung) fast zu 100% durch die Linux-Prozesse garantiert. Zum andern kann Zygote vor dem Applikationsstart die wichtigsten Java-Pakete (*Core Libraries*) im speziellen Dex-Format in einen geschützten Speicherbereich laden, auf den alle Android-Applikationen über die DVM zugreifen können. Die Auswirkungen dieses Designentscheids auf die Arbeit des *Garbage Collectors* werden wir später im Abschnitt „Shared Memory und Garbage Collection" behandeln. Schliesslich soll an dieser Stelle noch bemerkt werden, dass jeder Linux-Prozess die *bionic library* (eine ganz gewöhnliche C-Standard-Library) mitschleppt.

Die Dalvik Virtual Machine

Die gewöhnliche Java Virtual Machine (JVM) ist eine Stack-Maschine, deren Instruktionen mit Bytecode dargestellt sind. Der Interpreter der JVM führt pro Bytecode den Dispatch-Prozess aus, welcher aus drei Phasen besteht:

- *fetch:* der aktuelle Bytecode wird vom Stack geholt;
- *decode:* in Abhängigkeit des Bytecode-Typs auf die notwendigen Argumente auf dem Stack zugreifen;

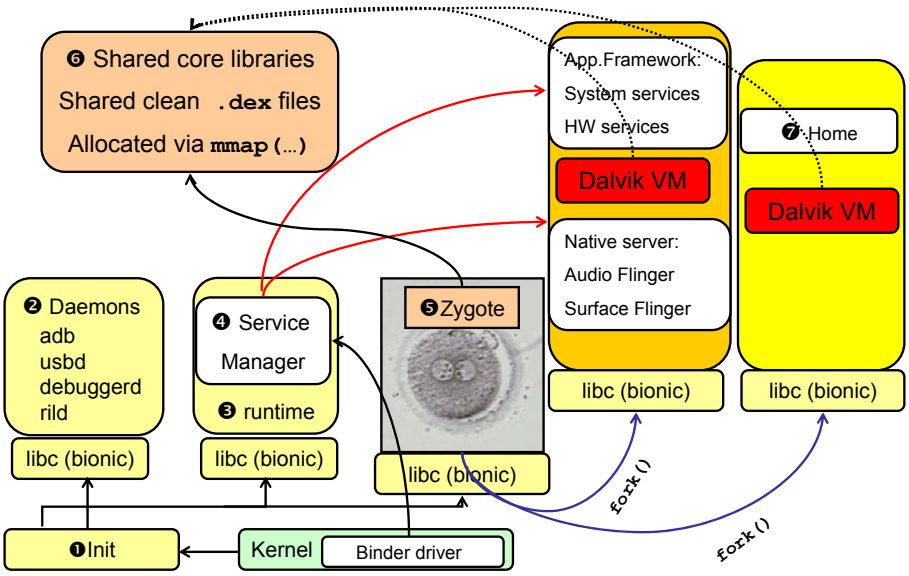

Abbildung 1: Der init.rc-Prozess in Android. Glossar der Abkürzungen: adbd: Android debugger; usbd: USB Schnittstelle; rild: radio interface layer; debugerd: debug system. Weitere daemons werden gestartet, sind aber in der Abbildung nicht dargestellt. Audio und Surface Flinger: eine Google Bezeichnung für MP3- bzw. graphische Dienste (modifiziert aus [Bern08]). Die eingekreisten Ziffern geben die zeitliche Reihenfolge der Prozesse an.

- *execute:* die durch den Bytecode dargestellte Funktion ausführen. Diese Phase beansprucht freilich am meisten Zeit.

In dieser Implementierung (siehe Listing 1) ist der Bytecode-Stack als Array dargestellt und die *Dispatch*-Stufe ist durch einen doppelten indirekten Zugriff via C-Zeiger kompakt und elegant realisiert. Diese einfache C-Funktion illustriert auch klar die drei Phasen des Interpreters, die oben erwähnt wurden. Nämlich wird *op_x* zuerst geholt (als Argument von *interp(...)*; dann decodiert via *DISPATCH()* und schliesslich ausgeführt (Funktion Aufruf nach der entsprechenden Bytecode Marke). Weiter geht mit dem wiederholten Aufruf von *DISPATCH()* am Ende der Zeile[1].

Register- vs. Stack-Maschine

Die DVM und JVM sind grob gesehen sehr ähnlich, wobei sie sich aber in zwei Hinsichten grundlegend unterscheiden: Die DVM ist nicht als Stack-Maschine, sondern als Register-Maschine realisiert, und die Längen der Opcodes betragen bei der DVM zwei anstatt nur einem Byte. Eine auf Register basierte virtuelle Maschine holt ihre Bytecodes und Operanden aus (virtuellen) Registern. Dazu ist es natürlich erforderlich, dass die Operanden in Abhängigkeit des Opcodes in bestimmte Register geschrieben und von dort gelesen werden und nicht generell vom Stack geholt werden, wie es in einer Standard JVM geschieht.

Die Vorteile einer Register- gegenüber einer Stack-Maschine sind *„prima facie"* nicht so evident, besonders wenn man zusammen mit der obigen Bemerkung bezüglich virtueller Register auch die DISPATCH()-Phase des Interpreters analysiert (siehe Listing 1). Solange der Stack und die virtuellen Register als lineare Arrays implementiert sind, unterscheiden sich die Realisierungen von DISPATCH() nur unwesentlich. Erst wenn die virtuellen Register auch wirklich in echten Prozessorregistern abgebildet werden, kann von einem Zeitgewinn bei der Ausführung der DISPATCH()-Anweisung ausgegangen werden. Mit einer Abbildung von virtuellen in reale Register handelt man sich aber auch einen gewaltigen Nachteil ein: Wenn man eine VM so modelliert, dass sie eine konkrete Prozessorarchitektur annähert, dann verliert man alle Portabilitätsvorteile einer herkömmlichen Stack-Maschine, die bekanntlich auf jeder erdenklichen Prozessorarchitektur realisierbar ist.

Auf der anderen Seite kann eine VM, die besser an die Hardware angepasst ist, direkt mit Maschinensprache umgehen und somit auf komplizierte *Just In Time* (JIT) Kompilierung verzichten. Diesen Vorteil macht sich auch die DVM zu nutzen, da sie sich primär auf die ARM-Architektur ausrichtet. Die DVM kann auf maximal 64K virtuelle Register zugreifen, die natürlich im L2- bzw. Hauptspeicher abgebildet werden müssen. Java Methoden, die mehr als 16 Argumente und Parameter benötigen, sind jedoch extrem selten, so dass üblicherweise fast alle Argumente und Parameter bei einem Methodenaufruf in den 16 16-Bit Registern des ARM-Prozessors Platz finden.

Interessanter wird, wenn der Vergleich zwischen einer Stack und einer Register basierten VM auch auf die Bytecodelänge ausgeweitet wird. Wir vergleichen dazu in Tabelle 1 die Anzahl Code-Bytes für die Funktion *tryItOut(...)* aus Listing 2, welche zwei Integer-Parameter addiert und das Resultat zurückgibt. Bei einer hypothetischen

1 Anmerkung für Hackers: Das Interpreter-Programm soll mit gcc (ab v. 4) kompiliert werden.

```
#define DISPATCH() { goto *op_table[*((s)++) - ‹a›]; }
static void interp(const char* s) {
    static void* op_table[] = {
        &&op_a, &&op_b, &&op_c, &&op_d, &&op_e
    };
    DISPATCH();
    op_a: printf(«Hell»); DISPATCH();
    op_b: printf(« or»); DISPATCH();
    op_c: printf(" Para"); DISPATCH();
    op_d: printf("dise!\n"); DISPATCH();
    op_e: return;
}
```

Listing 1: Einfache Struktur eines C-Interpreter-Programmes. Der Stack ist mit einem linearen Array realisiert. Die Bytecodes, die als Argument der Funktion interpret(…) erscheinen, werden via DISPATCH() durch doppelte Indirektion zur richtigen Marke geführt, wo schliesslich der dekodierte Bytecode ausgeführt wird.

Register VM müssen die beiden Operanden der Addition in Register kopiert werden. Bei der DVM sorgt bereits der Aufrufer der Methode für das Ablegen der Übergabeparameter in Registern. Das Zwischenspeichern des Resultats der Addition bevor es zurückgegeben wird, ist im Java Code klar ersichtlich und wichtig, weil *ireturn* das Resultat nur aus dem Operand-Stack holen kann.

Java VM 1.6	Register VM	Dalvik VM (16 Bit Opcade)
0 iload_1	move v11 -> v2	
1 iload_2	move v12 -> v3	
2 iadd	iadd v2 v3 -> v0	0000: add-int v0, v2, v3
3 istore_3		
4 iload_3		
5 ireturn	ireturn v0	
6 Bytes	4 Bytes	6 Bytes

Tabelle 1: Vergleich Anzahl generierte Bytecodes bei einer Stack-VM (Java VM 1.6) und einer hypothetischen Register VM. In der dritten Spalte die tatsächlichen Bytecodes, welche von Dalvik 1.5 generiert werden. Die add-int Instruktion benötigt mit ihren Parametern vier Bytes.

Wie das kleine Beispiel in Tabelle 1 schön zeigt, wird die theoretisch mögliche Bytecode-Einsparung einer Register VM von der DVM üblicherweise nicht erreicht. Dabei sollte man aber nicht vergessen, dass die DVM mit 16 Bit langen Opcodes arbeitet, was in Klartext bedeutet, dass die DVM pro Lesevorgang doppelt so viele Bytes laden kann wie die Standard JVM. Gerade bei heute üblichen 16-Bit, 32-Bit- oder 64-Bit-Architekturen ist die Bearbeitung von lediglich 8 Bits pro Opcode eine Verschenkung von Ressourcen. Da bei einer Register VM die Argumente und Parameter in echte Prozessorregister geladen werden, ist deren Zugriff jedoch entsprechend schneller als derjenige auf einen externen Stack.

Ein weiterer kleiner Vorteil einer Register VM ist, dass die typische Nebenbedingung einer Stack VM ersatzlos wegfällt, nämlich dass der Opcode Stack am Ende eines Methodenaufrufes im gleichen Zustand wie am Anfang sein soll.

Zusammengefasst lässt sich sagen, dass Register VM eine bessere Effizienz bei der Bearbeitung der Opcodes versprechen, und wenn Argumente und Parameter einer Java Methode in den Prozessorregistern Platz finden, ist auch die Ausführungszeit kleiner.

Wenn man auf eine portierbare Implementierung des Interpreters verzichtet (was ja Dalvik auch tut), kann man ihn auch sehr elegant und kompakt in der jeweiligen CPU-Maschinensprache programmieren. Eine besonders gelungene Implementierung ist diejenige für die ARM-CPU-Familie, die zurzeit in allen Android Smartphones eingesetzt wird [Bern08].

Dalvik Opcodes

Alle 220 Opcodes von Dalvik werden einheitlich mit 16 Bit definiert. Dies ist nicht nur ein bewusster Entscheid, um eventuelle juristische Streitigkeiten mit Sun zu vermeiden, sondern auch eine sinnvolle Anpassung der VM an die realen Prozessorarchitekturen, die schliesslich diese VM auch verwirklichen.

Die Struktur der Opcodes lässt sich gut anhand der Methode *public void tryFinally()* aus Listing 2 illustrieren.

In Listing 3 sind #6, #17 mit *Lch/fhnw/examples/FinallyInternal;.tryItOut:()V* und #5, #20 mit *Lch/fhnw/examples/FinallyInternal;.wrapItUp:()V* äquivalent. Zum Verständnis des Java 1.4 Codes ist es hilfreich zu wissen, dass jsr die Rücksprungadresse auf den Operanden Stack rettet. Somit wird in Instruktion 4 zur Instruktion 14 gesprungen, dort die gerettete Rücksprungadresse (7 return) vom Stack in die lokale Variable 2 gespeichert, dann die Methode der finally-Klausel ausgeführt und schliesslich mit ret 2 an die in Variable 2 gespeicherte Adresse zurückgesprungen. Der Opcode jsr ist somit eine Art von Java-Methodenaufruf, der aber die JVM-Spezifikation verletzt [Gos95, Ag97]. Man hat dies ab Java 1.5 korrigiert. Die DVM hält die JVM-Spezifikation ein und generiert erst noch besser lesbaren Code.

Der Aufbau eines Dalvik Opcodes lässt sich wiederum gut an einem Beispiel zeigen. Die Anweisung *invoke-virtual {v4,v0,v1,v2,v3}, Foo.method6:(||||)V* ruft eine Instanzmethode der Klasse *Foo* auf, konkret die sechste Methode in der Methodentabelle,

```
public class FinallyInternal {
    public void tryFinally() {
        try {
            tryItOut(1, 2);
        } finally {
            wrapItUp();
        }
    }
    private int tryItOut(int r1, int r2) {
        int retVal;

        retVal = r1 + r2;
        return retVal;
    }
    private void wrapItUp() {
    }
}
```

Listing 2: Das Test-Programm, das für die Analyse in Listing 3 benutzt wurde. Hier wurde speziell untersucht, wie die Klausel try{…} finally{…} von der Dalvik VM übersetzt wird.

die neben der this-Referenz noch vier Parameter *v0, v1, v2* und *v3* benötigt und keinen Rückgabewert liefert. *v4* ist dabei die *this* Referenz. Die Anweisung wird wie folgt codiert: *0x6E53 0x0600 0x0421*. *6E* codiert die eigentliche Instruktion *invoke-virtual*; in *53* steht die 5 für die Anzahl Parameter und 3 für *v3*; *0600* bezeichnet den Index in der Methodentabelle (Methode 6); *0421* bezeichnet die Parameter *v0, v4, v2, v1*. Dabei stellen die *vx* die (virtuellen) Register der DVM dar.

Dalvik spezifiziert auch Instruktionen, die erst nach der Verifizierung der Bytecode-Dateien (Dateinamenserweiterung *.dex*, siehe unten) vom Programm *dexopt* benutzt werden können. *dexopt* gestaltet die Dex-Dateien effizienter. Dabei werden grundsätzlich drei Dinge [AP08a] optimiert:

- *Alignement*: Die Opcodes und die Daten müssen nach 16 Bit ausgerichtet sein. Spezielle Ausrichtungen wie z.B. 64 Bit werden explizit in Assembler-Code vorgenommen. *dexopt* führt zusätzliche *nop* (*no operation*) Instruktionen ein, um die gewünschte Ausrichtung zu erreichen.
- *Virtuelle Methoden:* Für alle Aufrufe virtueller Methoden wird der Methodenindex durch einen Index in eine *vtable* ersetzt, die die Startadresse des Code-Teils im .dex-File beinhaltet, Dadurch wird eine Indirektion eingespart. Dies ist besonders wichtig, weil Zygote vorkompilierte Klassen herunter lädt, welche von allen zukünftigen Instanzen der DVM benutzt werden.
- *Effizientere Opcodes:* Einzelne Opcodes können durch effizientere ersetzt werden, wie zum Beispiel *invoke-virtual-quick*, welcher besonders effizient mit der *vtable* des Zielobjektes arbeitet.

Die so optimierten Dex-Dateien werden mit der Dateinamenserweiterung *.odex* bezeichnet.

Das Dalvik Executable Format

Ein so grundlegender Abschied vom klassischen Java *.class* Opcode Format, war dem DVM Team auch eine willkommene Möglichkeit, Korrekturen vorzunehmen, die den aktuellen Stand der

Java VM ≤ 1.4

```
0  aload_0
1  invokevirtual #6
4  jsr 14
7  return
8  astore_1
9  jsr 14
12 aload_1
13 athrow
14 astore_2
15 aload_0
16 invokevirtual #5
19 ret 2
```

Java VM ≥ 1.5

```
0  aload_0
1  invokespecial #17
4  goto 14 (+10)
7  astore_1
8  aload_0
9  invokespecial #20
12 aload_1
13 athrow
14 aload_0
15 invokespecial #20
18 return
```

Dalvik VM

```
0000: invoke-direct{v1}, Lch/fhnw/examples/FinallyInternal;.tryItOut:()V
0003: invoke-direct {v1}, Lch/fhnw/examples/FinallyInternal;.wrapItUp:()V
0006: return-void
0007: move-exception v0
0008: invoke-direct {v1}, Lch/fhnw/examples/FinallyInternal;.wrapItUp:()V
000b: throw v0
```

Listing 3: Opcodes für die Methode tryFinally() generiert mit drei Java Compilern: a) Java 1.4: Anzahl Bytes 20; b) Java 1.5: Anzahl Bytes 19; c) Dalvik 1.5: Anzahl Bytes 22

Segment Name	Format	Beschreibung
header	header_items	Neben der Versionsnummer, der obligaten magischen Zahl und gewisser Sicherheitsmerkmale findet man hier Angaben über die Grösse der verschiedenen Segmente und deren Startadressen.
string_ids	string_id_item[]	Enthält alle Strings, die von dieser Datei entweder als interne Bezeichner (z.B. von Typen) oder als konstante String-Objekte benutzt werden.
type_ids	type_id_item[]	Bezeichner für alle Typen (Klassen, Arrays, Elementartypen) die in dieser Datei referenziert werden, unabhängig davon, ob sie in dieser Datei definiert werden oder nicht.
proto_ids	proto_id_item[]	Signaturen aller Methoden, die in den verschiedenen Klassen referenziert sind. Beispiel: wrapItUp()V
fields_ids	fields_id_item[]	Bezeichner für alle Instanzvariablen, die in dieser Datei referenziert und benutzt werden, unabhängig davon, ob sie in dieser Datei definiert werden oder nicht.
methods_ids	methods _id_ item[]	Bezeichner für alle Methoden, die in dieser Datei referenziert und benutzt werden, unabhängig davon, ob sie in dieser Datei definiert werden oder nicht.
class_ids	class _id_item[]	Liste der Klassendefinitionen. Die Liste muss die Basisklasse und die implementierten Interfaces vor der Klasse, welche diese benutzt, auflisten.
data	ubyte[]	Datenbereich, worin die Informationen für alle oben definierten Tabellen gespeichert sind.
link_data	ubyte[]	Datenbereich für die statische Bindung von weiteren Dateien.

Tabelle 2: Die verschiedenen Segmente des Dex-Formats

Java-Erfahrungen widerspiegelten. Man sollte nur an die gewaltigen Probleme denken, die die Sun-Ingenieure bewältigen mussten, um das Konzept von *Generics* in Java zu realisieren, ohne die Struktur der Java Opcodes zu verändern!

Das Dalvik Executable Dateiformat (Dex-Format) ist in Segmente unterteilt, deren Reihenfolge zwingend vorgegeben ist. Die Tabelle 2 fasst diese Segmente zusammen und beschreibt sie kurz. In Abbildung 2 zeigen wir ein konkretes Beispiel eines Header-Segmentes: Die nummerierten Kästchen sind wie folgt zu interpretieren: (1) „dex■035■“: Magische Zahl und Version; (2) 32-Bit CRC Checksumme aller Bytes mit Ausnahme der ersten 12; (3) Länge der Datei in Bytes; (4) Länge des Headers in Bytes; (5) Little-Endian CPU (die umgekehrte Reihenfolge der Bytes würde eine Big-Endian CPU voraussetzen); (6) Startadresse des Segments *string_ids*; (7) Startadresse des ersten Bezeichners in der Tabelle *string_id_item*.

Während die einzelnen Bytecode-Dateien (.class) eines für die Standard JVM kompilierten Java-Programmes üblicherweise in einem (komprimierten) Java-Archiv (.jar) zusammengefasst werden, so braucht es für die DVM kein zusätzliches Archiv-Format, da das Dex-Format sowohl einzelne Klassen als auch eine beliebige Kollektion von Klassen definieren kann.

Das wichtigste Merkmal des Dex-Formats besteht darin, dass alle Strings zur Bezeichnung von Klassen, Methoden usw. nur einmal in der gesamten Datei gespeichert werden. Alle Wiederholungen werden konsequent gestrichen und nicht wie im jar-Format für jede neue Klasse noch einmal im *constant_pool* Bereich definiert (Abb. 3). Damit wird die Dateilänge einer Dex-Datei im Durchschnitt um 35% kürzer als diejenige einer äquivalenten, unkomprimierten jar-Datei.

Da das Dex-Format Informationen für den Android Debugger (*adb*) codiert, werden diese im *data*-Segment mit dem DBG-Präfix speziell gekennzeichnet. Die Art und Weise wie diese Debug-Information definiert wurde, ist massgeblich aus der DWARF Debugging Format Spezifikation übernommen worden [DWARF07]. Zum Beispiel wird die in dieser Spezifikation definierte LEB128-Codierung (eine variable Bit-Codierung für die Darstellung von ganzen Zahlen) stark benutzt, um so genannte *encoded-value* Elemente der Dex-Datei kompakt darzustellen.

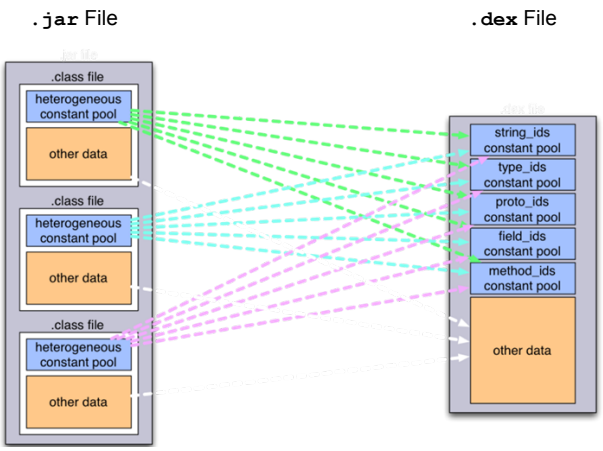

Abbildung 2: Beispiel eines konkreten Header-Segmentes einer Dex-Datei

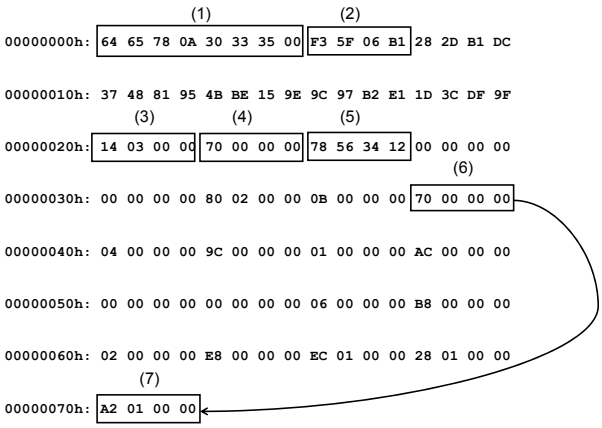

Abbildung 3: Das Dex-Format fasst alle heterogenen constant_ pool Bereiche zusammen, die eine jar-Datei für jede mitgeschleppte Klasse neu definiert (aus [Bern08]).

```
public class MemInfo extends Activity {
    private TextView displayResults;
    private Debug.MemoryInfo dMemInfo;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        dMemInfo = new Debug.MemoryInfo();

        // Set output for results
        displayResults = (TextView) findViewById(R.id.results);

        // Add button listener. Watch for button clicks.
        Button getButton = (Button) findViewById(R.id.get);
        getButton.setOnClickListener(getListener);

        Button setButton = (Button) findViewById(R.id.set);
        setButton.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View v) {
                System.getProperties().put(this.getClass().
                    getName(),"Android");
            }
        });
    }
    private OnClickListener getListener = new OnClickListener() {
        public void onClick(View v) {
            Debug.getMemoryInfo(dMemInfo);
            displayResults.setText(
                «Shared/Private dirty/clean Memory\n\n»
                + «Dalvik proportional set size [kB] «
                + Integer.toString(dMemInfo.dalvikPss) + "\n"
                + "Dalvik private dirty [kB] "
                + Integer.toString(dMemInfo.dalvikPrivateDirty) + "\n"
                + "Dalvik shared dirty [kB] "
                + Integer.toString(dMemInfo.dalvikSharedDirty) + "\n");
        }
    };
}
```

Listing 4: Android-Applikation, die laufend die Grösse ihrer verschiedenen Speicherbereiche darstellt. Dies zeigt konkret das Zusammenspiel zwischen Zygote shared Speicherbereichen und denen, die die Applikation lokal definiert hat (private dirty).

Verifizierung und Optimierung von Dex-Dateien

Die Überprüfung (*verify*) der Dex-Dateien weicht sehr stark vom Standard Java-Verfahren ab. Der Grund liegt darin, dass viele Klassen und Applikationen (so genannte *boostrap classes*) von Zygote ins *dalvik-cache* Verzeichnis ohne Beteiligung eines *Class Loaders* mittels Memory-Mapping geladen werden. Dies bedeutet, dass die komprimierten Dex-Dateien nicht nur dekomprimiert, sondern vor dem Speichern auch einer Vorverifizierung unterzogen werden. Nach der Vorverifizierung aber noch vor der Speicherung werden die Dex-Dateien mit *dexopt* optimiert und erhalten dadurch die neue Dateierweiterung *.odex.* Das Ganze dient auch dazu, die Android-Applikationen schneller starten zu können.

Zygote muss während der Vorverifizierung einige vernünftige Annahmen treffen, um nicht später mit dem *Class Loader* der DVM in Konflikt zu geraten [AP08b]. Als Beispiel gehen wir von ei ner Applikation *MyApp.apk*[2] aus, die eine eigene String-Klasse im Package *java.lang* unter dem Namen String definiert und somit in Konflikt mit der Standardklasse *java.lang.String* kommt. Grundsätzlich könnte man in der Vorverifizierungsphase annehmen, dass die echte Implementierung von *java.lang.String* bereits in den *core.jar* Klassen definiert wurde. Somit könnte Zygote unser Programm *MyApp.apk* weiter überprüfen und am Ende noch optimieren. Das ist nicht sehr klug, weil beim späteren Laden unserer Klasse in der DVM das System merken wird, dass etwas nicht in Ordnung ist. Daher wählt *dexopt* eine bessere Strategie: Sobald eine Klassendefinition gefunden wird, die die gleiche Signatur einer früher vorverifizierten Klasse beinhaltet, stoppt *dexopt* ohne jegliche Verifizierung bzw. Optimierung vorzunehmen. Es ist dann die Aufgabe der DVM diese Klassendefinition zu verifizieren. Der Verifikationsprozess in der DVM soll dabei strikt achten, dass alle Referenzen zu den Klassen entweder in unserer Applikation oder in einer früheren *bootstrap.apk* vorhanden sind, um zu vermeiden, dass ein benutzerdefinierter *Class Loader* z. B. eine neue Version der *core.jar* Klassen (vielleicht mit einem Virus-Programm bestückt!) laden kann.

2 Android Package Files verwenden die Dateierweiterung .apk. Es handelt sich dabei um ZIP-komprimierte Ordner analog zu Java Archives (.jar). Die core libraries werden im Verzeichnis dalvik-cache als system@framework@core.jar@classes.dex abgelegt.

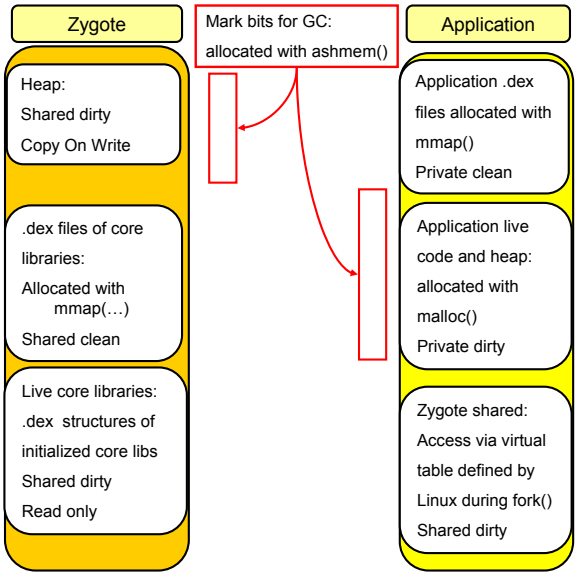

Abbildung 4: Zusammenhang zwischen den globalen (shared) Bereichen von Zygote und denjenigen einer beliebigen Android-Applikation.

Neben der Lösung des eben beschriebenen Problems soll *dexopt* weitere Checks durchführen [AP08b]. So soll sichergestellt werden, dass (i) nur legale Dalvik-Opcodes in den Methoden benutzt werden; (ii) *move-exception* als erste Instruktion in einem *exception handler* auftritt; (iii) *move-result** nur unmittelbar nach einer *invoke-** oder *filled-new-array* Instruktion vorkommt und (iv) *dexopt* nur solche Rücksprünge auf den Stack verbietet, bei der eine *new-instance* Instruktion auf eine nicht initialisierte Registerreferenz hinweist.

Viele weitere Standard-Checks einer JVM sind in der DVM ohne jegliche Bedeutung. Zum Beispiel hat die DVM keinen Operanden-Stack, daher ist eine Überprüfung desselben unnötig, und die Typen-Restriktion auf Referenzen im *constant_pool* fällt in einer DVM weg, weil kein solcher *constant_pool* in einer Dex-Datei spezifiziert ist.

Optimierungen innerhalb des Dex-Formats sind dagegen eine Besonderheit der DVM. Sie lassen sich wie folgt klassifizieren [AP08a]:

- Optimierungen, welche Instruktionen und Daten auf 16-Bit ausrichten. Oft bedeutet dies das Einführen von *nop*-Instruktionen im Code-Teil und das Auffüllen (*padding*) der Datenbereiche bis die gewünschte Ausrichtung erreicht worden ist.
- Das Ausschneiden (*prune*) von leeren Methoden.
- *Inline*-Austausch von oft verwendeten Methoden mit direkten Aufrufen auf native Implementierungen.
- Anstatt virtuellen Methodenaufrufe via *invoke-virtual** und über einen Methodenindex zu tätigen, werden Instruktionen wie *invoke-quick* verwendet, die effizienter auf absolute Adressen in einer Sprungtabelle (*vtable*) via Index zugreifen.
- Daten (z.B. Hash-Werte) im Voraus berechnen, um diese Berechnungen in der DVM zu vermeiden.

Shared Memory und Garbage Collection

Im Allgemeinen klassifiziert jedes Betriebssystem den Speicher in vier Qualitätsklassen, die sowohl nach der Art seiner Allokation als auch nach der Art seiner Zugänglichkeit definiert sind.

- *Shared clean*: Der Speicher ist allen Prozessen zugänglich (global). Da alle Android-Applikationen via *fork()* durch Zygote erzeugt werden, bedeutet dies, dass sie auf bestimmte Speicherregionen von Zygote zugreifen können. *Clean* bezeichnet die Art wie Zygote diese gemeinsamen Speicherregionen mit Informationen gefüllt hat. In diesem Fall benutzt Zygote *mmap(…)* für schnelles Laden/Löschen eines binären Speicherabbildes durch das Betriebssystem (Memory-Mapping).
- *Shared dirty*: Wie „shared clean" aber die Allokation/Freigabe erfolgt mit *malloc(…)* und *free(…)*, was langsam und von jeder Applikation abhängig ist.
- *Private clean*: Hier bedeutet *private* prozessspezifisch. Beispiel: Lokale Aufbewahrung der spezifischen Dex-Dateien jeder einzelnen Android-Applikation, die mit *mmap(…)* zugewiesen worden sind.
- *Private dirty*: Wie „private clean" aber die Allokation/Freigabe erfolgt mit *malloc(…)* und *free(…)*. Beispiel: Applikations-Heap[3].

Diese Speicherklassifizierung ist eine Grundvoraussetzung für die sparsame Speicherverwaltung, wenn Zygote mehrere Android-Applikationen starten soll. Die Devise lautet dabei: Möglichst viele initialisierte Klassen im Voraus in globale (*shared*) Speicherbereiche zu laden, um den *memory footprint* aller Applikationen so gering wie möglich zu halten.

Tatsächlich lädt Zygote eine mehr oder weniger geschickt gewählte Mischung von Java-Klassen im Voraus in einen globalen Speicherbereich, damit mehrere Applikation diese Klassen gemeinsam benutzen können, ohne selber lokal Platz dafür zu verbrauchen, und um das Starten der eigentlichen Android-Applikationen zu beschleunigen.

Die erwähnte Mischung von Java-Klassen soll nicht nur für möglichst viele Anwendungen nützlich sein, sondern auch während der Lebensdauer der Applikationen beinahe unverändert bleiben. Um dieses Ziel zu erreichen, bildet Zygote zwei Speicherbereiche: Ein *shared dirty* (aber *read only*) für das Speichern der Dex-Strukturen und ein ebenfalls *shared dirty* Heap-Bereich, worin die Klassenobjekte realisiert werden und allen anderen Applikationen zur Verfügung gestellt werden. Dieser Zygote-Heap soll aber möglichst selten verändert werden. Wenn eine Applikation ein Klassenobjekt vom Zygote-Heap referenziert,

3 Im Listing 4 findet man eine Android-Applikation, die die Zunahme bzw. die Abnahme der verschiedenen Speicherbereiche darstellt.

wird dieses Objekt in den eigenen (*private dirty*) Heap kopiert (*copy on write*). Ein dritter Bereich (*shared clean*) beinhaltet alle Dex-Dateien der sogenannten Kernbibliotheken, die in *core.jar* definiert worden sind. Wenn eine Applikation Pakete aus diesem Bereich benötigt, werden die beiden bereits diskutierten *shared dirty* Speicherbereiche entsprechend erweitert. Die Abbildung 4 zeigt noch einmal die Zusammenhänge zwischen den globalen Speicherbereichen von Zygote und denjenigen einer beliebigen Android-Anwendung.

Das Zusammenspiel zwischen *shared* Speicherbereichen von Zygote und Android-Applikationen setzt der Funktionalität eines *garbage collectors* (GC) einige Grenzen. Dazu ist wichtig zu bemerken, dass der Dalvik GC ein gewöhnliches *mark and sweep* Verfahren benutzt. Die Markierungsbits (die Information, ob Objekte noch am Leben sind) können beim *mark and sweep* Verfahren entweder zusammen mit den Objekten im Heap oder in einem vom Heap getrennten Speicherbereich aufbewahrt werden. Die Verwendung eines getrennten Speicherbereichs ist für Dalvik die einzige Lösung, welche der Heap-Verwaltung in der komplexen Symbiose zwischen Zygote und Android-Applikation gerecht werden kann. Wie bereits erwähnt, sollen Daten auf dem Zygote-Heap selten verändert werden, da sie für möglichst viele verschiedene Applikationen gelten sollen. Daher wird vorzugsweise eine externe (globale) Struktur für die Aufbewahrung der Markierungsbits eingesetzt. Sie ermöglicht auch die Unterscheidung zwischen allgemeinen *shared* Objekten und Objekten, die die Applikation selber instanziert hat. Der GC soll nur im Heap der Applikation schalten und walten; Objekte im *shared* Berich werden (wenn überhaupt) nur von einem speziellen GC von Zygote eingesammelt und eventuell gelöscht, nachdem alle GCs der Applikationen nach einem vollständigem *sweep* gestoppt wurden (siehe auch Abb. 4).

Das Android Linux Betriebssystem benutzt die Funktionen *ashmem_create_region(...)* und *ashmem_set-prot_region(...)*, um die Speicherbereiche für die Markierungsbits als anonyme *shared* Speicheregionen zwischen Prozessen zu definieren. Man beachte, dass diese Bereiche von Android Linux Kernfunktionen gelöscht werden können. Dies ist freilich notwendig, damit der GC von Zygote seine Arbeit verrichten kann.

Zusammenfassung

In diesem Übersichtsartikel haben wir einige der Hindernisse aufgezeigt, die das Google-Team überwinden musste, um die volle Kompatibilität mit dem Java SDK 1.5 zu bewahren ohne jedoch die Sun-Spezifikation der JVM einzuhalten. Dieser Tabubruch ist Google nicht nur sehr gut gelungen, sondern hat auch neue Ideen in der Java-Community ausgelöst, um die Sun 1-Byte-JVM endlich an die Realitäten der heutigen HW anzupassen. Die Hauptannahme des Google-Entwicklungsteams, dass die Dalvik VM (im Gegensatz zur Sun HotSpot JVM) keine *Just In Time* (JIT) Kompilierung benötigt, hat sich dagegen besonders im mobilen Bereich als falsch erwiesen. Zwei Entwicklungen zeugen von diesem Sinneswandel bei Google: Einerseits die Freigabe der Android Native Libraries (die eine bessere Einbindung von C-Programmen in Android Java-Programme erlaubt) und anderseits die Mitteilung, dass wirklich *really soon* die Dalvik VM mit einem JIT-Kompiler erweitert wird.

Referenzen

[Ag97] Agesen, O., Detlefs, D. Finding References in Java Stacks. OOPSLA97 Workshop on Garbage Collection and Memory Management, October 1997.

[AP08a] The Android Open Source Project, Dalvik Optimization and Verification With dexopt, 2008: http://github.com/android/platform_dalvik/blob/c1b54205471ea7824c87e53e0d9e6d4c30518007/docs/dexopt.html.

[AP08b] The Android Open Source Project, Dalvik Bytecode Verifier Notes, 2008 http://github.com/android/platform_dalvik/blob/c1b54205471ea7824c87e53e0d9e6d4c30518007/docs/verifier.html.

[Bern08] Bernstein, D. Dalvik VM Internals. IO-Google Conference, May 29th, 2008.

[DWARF07] Eager, M. J.: Introduction to the DWARF Debugging Format, (2007)

http://dwarfstd.org/

[Gos95] Gosling, J. Java Intermediate Bytecodes. ACM SIGPLAN Workshop on Intermediate Representations, 111-118, 1995.

[Wiki] Android (Betriebssysstem): http://de.wikipedia.org/wiki/Android_(Betriebssystem).

Lokalisierung mittels WLAN

Mobile Anwendungen, die auf die aktuelle Position des mobilen Gerätes zugreifen können, erlauben eine grosse Bandbreite von interessanten Anwendungen. Oftmals ist ein GPS-Empfänger verfügbar und somit können Positionsdaten über GPS bezogen werden. In Gebäuden und Strassenschluchten sind jedoch GPS-Daten entweder gar nicht verfügbar oder stark fehlerbehaftet. Daher stossen GPS-freie Lokalisierungsansätze auf grosses Interesse. Wir haben einen Ansatz basierend auf vorgängig gemessenen Referenzdaten (WLAN Fingerprints) genauer untersucht[1] und präsentieren hier Resultate, die zeigen, mit welcher Lokalisierungsgenauigkeit gerechnet werden kann und von welchen Faktoren diese abhängt. Für den Fall, dass die Referenzdaten eine Auflösung von etwa 10 m haben, liegen die mittleren Lokalisierungsfehler zwischen 5 und 15 m.

Christoph Stamm, Beat Walti, christoph.stamm@fhnw.ch

Mobile Anwendungen werden üblicherweise stark personenbezogen und ortsunabhängig betrieben. Dieser Umstand lässt sich programmtechnisch dahingehend ausschöpfen, dass Kontextinformationen mit einbezogen werden und der Benutzerin der mobilen Anwendung dadurch ein minimales Gefühl von künstlicher Intelligenz vermitteln. Eine der wichtigsten Kontextinformationen ist die geographische Position des mobilen Gerätes bzw. deren Benutzerin. Die Verfügbarkeit von Positionsdaten in mobilen Systemen ist heute im Allgemeinen recht gut, vor allem dort wo auf das *Global Positioning System* (GPS) zugegriffen werden kann, denn immer mehr mobile Geräte enthalten einen eingebauten GPS-Empfänger. Der grosse Nachteil von GPS ist hauptsächlich der ungenügende Empfang der Satellitendaten innerhalb oder entlang von Gebäuden oder in engen Strassenschluchten, die zu einer Abschattung führen (siehe Abb. 1 links). Glücklicherweise sind jedoch in solchen Situationen sehr oft drahtlose Netzwerke (WLANs) vorhanden.

Ein WLAN-Zugriffspunkt (*Access Point*, AP) sendet in regelmässigen Abständen ein Signal aus, um es allfälligen Clients zu ermöglichen, den AP zu finden, sowie eine Verbindung mit ihm aufzubauen. Dieses Signal sendet ein AP auch dann aus, wenn er sich im „versteckten" Modus befindet, das heisst, wenn der Service Set Identifier (SSID) nicht öffentlich ist. Die Identifikation eines Netzwerkadapters des APs (MAC-Adresse) ist immer zugänglich, unabhängig davon ob der AP passwortgeschützt ist oder nicht. Dies ermöglicht einem Client stets eine aktuelle Liste der erreichbaren APs zur Verfügung zu haben und sich rasch mit einem zu verbinden, falls eine Datenverbindung erwünscht wird. In den meisten Fällen kann neben der MAC-Adresse auch die vorhandene Signalstärke ermittelt werden.

In diesem Artikel beschreiben wir zuerst verschiedene Ansätze zur GPS-freien Lokalisierung, stellen dann unseren eigenen genauer vor, welcher sich stark am Wi-Fi Positioning System von Skyhook-Wireless orientiert [SHW] und präsentieren schliesslich unsere Testresultate.

Ansätze zur Lokalisierung ohne GPS

Eine sehr grobe Lokalisierung eines Mobiltelefons ist ohne GPS anhand der aktuell zugeordneten Basisstation und deren Koordinaten möglich (Cell-of-Origin-Verfahren). Üblicherweise geben die Betreiber der GSM-Netze diese Koordinaten jedoch nicht frei, sondern stellen sie der Polizei und den Rettungsdiensten zur Verfügung oder nutzen sie allenfalls für eigene Anwendungen. Trotzdem sind heute verschiedene private und öffentliche Datenbanken (z.B. Sitefinder in Grossbritannien [SF]) mit den wichtigsten Angaben zu GSM-Basistationen verfügbar und können für die Lokalisierung verwendet werden. Falls für das mobile Gerät mehrere Basisstationen „sichtbar" sind, so können die Koordinaten aller sichtbaren Basisstationen zu einer groben Triangulierung bzw. Einmittung der aktuellen Position herangezogen werden. Die typische Positionierungsgenauigkeit liegt bei diesem Ansatz zwischen 200 und 1000 m.

Eine Umsetzung und Erweiterung des Cell-of-Origin-Ansatzes wird im Place Lab Projekt verfolgt [PL]. Neben den GSM-Antennen werden WLAN-Zugriffspunkte und fixe Bluetooth-Geräte mit ihrer Geräteidentifikation, Position und Reichweite in einer frei verfügbaren Datenbank abgelegt. Zur Bestimmung der unbekannten Position eines mobilen Gerätes werden die an der aktuellen Position vorhandenen Zugriffspunkte ermittelt und deren Koordinaten in der Datenbank nachgeschlagen. Aus diesen Koordinaten wird schliesslich durch Distanzschätzung oder Einmittung auf die eigene Position geschlossen.

1 Mit freundlicher finanzieller Unterstützung des Fördervereins Fachhochschule Nordwestschweiz Solothurn FVFS.

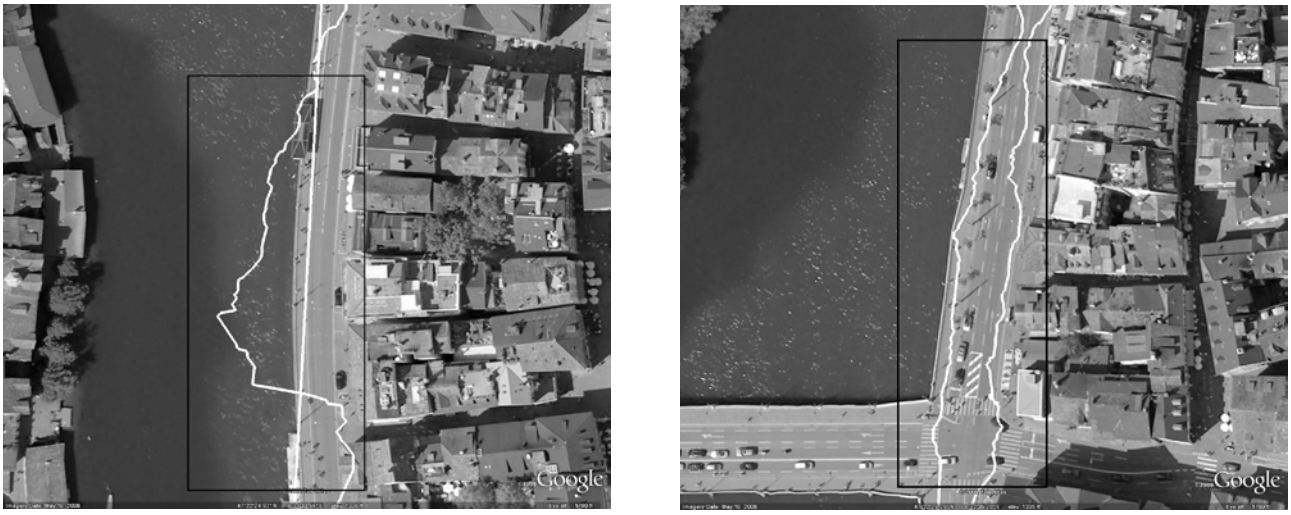

Abbildung 1: GPS-Positionen können von den wirklichen Positionen massiv abweichen (links) und nicht weit davon entfernt wieder sehr gut der Realität entsprechen (rechts)

Das Cell-of-Origin-Verfahren funktioniert nur, wenn die Positionen der Basisstationen bzw. APs bekannt sind und vom mobilen Gerät in irgendeiner Form abgefragt werden können. Während die Erfassung der Koordinaten von Basisstationen noch realistisch anmutet, kann dies bei der enormen Anzahl von WLAN APs nur schwer in Betracht gezogen werden. Daher sind Ansätze gefragt, bei denen auf die Positionen der Zugriffspunkte verzichtet werden kann. Der nachfolgende Ansatz ist ein solcher, wobei natürlich auch hier die Erfassung von gewissen Referenzkoordinaten ein wichtiger Teil des Verfahrens darstellt.

Das hybride Lokalisierungssystem XPS der Firma Skyhook-Wireless [SHW], welches in Apple's iPhone zum Einsatz kommt, verwendet neben GPS und dem Cell-of-Origin-Verfahren noch einen dritten Ansatz: das Wi-Fi Positioning System (WPS). Auch bei diesem Ansatz kommt eine (ständig zu aktualisierende) Datenbank zum Einsatz. In dieser Datenbank werden aber nicht die Positionen der Zugriffspunkte gespeichert, sondern es werden entlang der wichtigsten Strassen Referenzmessungen durchgeführt und erfasst. Eine solche Erfassung ist oft einfacher als die Bestimmung der Koordinaten der privaten Zugriffspunkte, da sie im öffentlichen Raum erfolgen kann. An einer erfassten Position können alle verfügbaren Mobilfunksignale (WLAN, GSM usw.) zusammen mit ihrer Signalstärke und einer Identifikation des Zugriffspunktes (z.B. MAC-Adresse) als „Fingerabdruck" ermittelt werden. Ein solcher „Fingerabdruck" ist beinahe eindeutig für eine Geländeposition und wird darum als charakteristisch bezeichnet. Eine Referenzmessung besteht nun aus dem charakteristischen „Fingerabdruck" und der zugehörigen Geländeposition, welche zum Beispiel mittels GPS erfasst werden kann. Zur Lokalisierung der aktuell zu bestimmenden Position wird wiederum der „Fingerabdruck" ermittelt und dieser mit denjenigen der Datenbank verglichen. Die relevantesten Referenzmessungen liefern dann eine Menge von Geländepositionen, die schliesslich zur Bestimmung der aktuellen Position verwendet werden. Somit ist eine vernünftige Bestimmung der Position nur möglich, wenn genügend Referenzdaten zum Vergleichen vorhanden sind. Die Firma Skyhook-Wireless spricht auf ihrer Website von einer Positionierungsgenauigkeit von 10 bis 20 m. Die Erfassung der Referenzdaten erfolgt vor allem in den USA und Europa. Es sind allerdings erst die grösseren Ballungszentren (z.B. Grossraum Zürich) und teilweise die wichtigsten Autobahnen erfasst worden.

In [DZ02] geben die Autoren einen guten Überblick über die verschiedenen Lokalisierungsansätze im Zusammenhang mit WLANs und beschreiben auch die zuvor skizzierten drei Techniken. Sind die Positionen der APs bekannt, so kann je nach der Anzahl der „sichtbaren" APs der rudimentäre Cell-of-Origin-Ansatz oder die genauere Triangulierung verwendet werden. Im ersten Fall liegt die Positionierungsgenauigkeit bei maximal 25 m, im zweiten ist eine Genauigkeit von 15 m erreichbar, unter der Restriktion einer hohen Zelldichte, was bedeutet, dass die APs einen ungefähren Abstand von 10 m haben. Werden jedoch analog zum WPS-Ansatz von Skyhook-Wireless vorgängig Referenzmessungen an bekannten Positionen durchgeführt, so erhöht sich die durchschnittliche Positionierungsgenauigkeit von 15 auf 10 m.

Auch im verallgemeinerten Kontext der drahtlosen (Ad-hoc) Sensornetzwerke spielt Lokalisierung eine wichtige Rolle. Sensornetzwerke können zum Beispiel zur Temperatur-, Licht-, Gas- oder Radioaktivitätsmessung verwendet werden. Solche Messungen machen oft nur dann Sinn, wenn die Position der Messung bekannt ist. Während für einzelne Sensoren die Positionen gegeben oder bekannt sind (Ankerpunkte), so müssen sie für die anderen möglichst genau bestimmt werden. Dazu

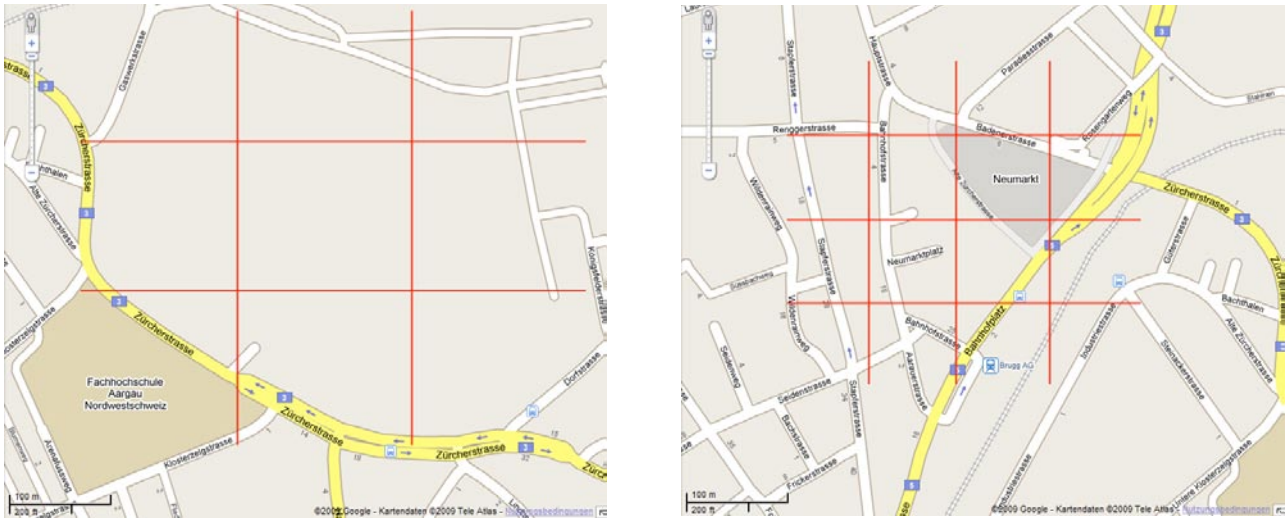

Abbildung 2: Verschiedene Kachelgrössen bei gleichem Kartenmassstab

bedarf es einer Metrik zur Approximation der Distanzen zwischen den Sensoren. Sind die Approximation zu genau drei Ankerpunkten vorhanden, so kann die Position exakt bestimmt werden, bei mehr als drei Distanzen wird üblicherweise versucht der quadratische Fehler zu minimieren [CYC06]. Die Distanzschätzungen erfolgen zum Beispiel über die Signallaufzeiten oder die Signalstärken. Im Fall von einer Line-of-Sight (LOS) Verbindung sind solche Näherungen meistens akkurat, im NLOS-Fall jedoch fast unbrauchbar. Daher sind unterschiedliche Ansätze bekannt, welche versuchen, NLOS-Verbindungen zu identifizieren, um sie für die Distanzmessung ausschliessen zu können [WH96]. Sollte dies aus ganz verschiedenen Gründen nicht möglich sein, so bleiben noch Ansätze, den Einfluss von vermeintlichen NLOS-Verbindungen auf die Positionierungsgenauigkeit zu mindern [JXZ07a, JXZ07b].

Verwaltung der Referenzdaten

Auf dem mobilen Gerät bietet sich eine API an, die alle verfügbaren Lokalisierungstechniken kapselt und jeweils die bestmögliche Lokalisierungstechnik verwendet. Ein Beispiel, wie eine solche API aussehen und realisiert werden kann, wird in [HM09] aufgezeigt.

Während bei einer Lokalisierung mittels GPS die Positionsbestimmung im mobilen Gerät ohne Zugriff auf einen Datenserver möglich ist, braucht das Wi-Fi Positioning System umfangreiche Referenzdaten. Diese Referenzdaten können auf einem zentralen Server gehalten oder zumindest teilweise lokal auf dem Gerät verwaltet werden. Im Fall des zentralen Servers würde ein Lokalisierungsservice Sinn machen, der nach der Übertragung des lokalen „Fingerabdrucks" die berechnete, aktuelle Position zurückliefert. Selbstverständlich sind Referenzdaten von Hamburg wenig hilfreich, wenn die Lokalisierung in Zürich geschehen soll, da WLANs eine eng begrenzte Reichweite haben. Das Gleiche gilt auch im grösseren Massstab: Referenzdaten vom Zürcher Limmatquai sind nur dann verwendbar, wenn man sich in der der Nähe des Limmatquais aufhält und nicht im drei Kilometer entfernten Stadtteil Oerlikon. Da sich eine Person jedoch in kürzester Zeit und mit wenig Aufwand vom Limmatquai nach Oerlikon bewegen kann, macht es Sinn, dass naheliegende Referenzdaten schnell verfügbar sind. Einschlägige effiziente Datenstrukturen für Externspeicher, welche Bereichsabfragen gut unterstützen, versuchen solche Nachbarschaften gebührend zu berücksichtigen [KNRW97]. Verschiedene dieser Datenstrukturen verwenden eine Kachelstruktur, wobei die Kachelgrösse in Abhängigkeit der Anwendung und der Datendichte gewählt werden sollte.

Eine geeignete Kachelgrösse berücksichtigt in unserem Fall sowohl den maximalen Abstrahlradius eines APs als auch die Dichte der APs im Gelände (siehe Abb. 2). Das bedeutet, dass die Kachelgrösse im Stadtzentrum kleiner ist als am Stadtrand, dass sie aber mindestens so gross ist, dass eine kleine Anzahl davon ausreicht, um die notwendige, lokale Umgebung für eine Lokalisierung zur Verfügung zu stellen.

Speicherbedarf

Um abschätzen zu können, ob eine WLAN-basierte Lokalisierung direkt auf dem mobilen Gerät durchgeführt werden kann, ist es hilfreich den Arbeitsspeicherbedarf abzuschätzen, denn nach wie vor ist der schnellere Arbeitsspeicher eines mobilen Gerätes eng begrenzt. Ein ökonomischer Umgang mit dem Arbeitsspeicher ist somit unumgänglich.

Auf die Referenzdaten bezogen heisst das, dass nur die wirklich erforderlichen Referenzdaten vom langsamen Externspeicher in den Arbeitsspeicher geladen werden sollen. Um ein Gefühl für die Datenmenge der Referenzdaten zu entwickeln, betrachten wir ein lokales Datenset von 100 km^2. Bei einer räumlichen Auflösung von 10 m in x- und

y-Richtung liegen in diesem lokalen Datenset bereits eine Million Referenzpunkte. Da ein grosser Teil davon an nicht öffentlich zugänglichen Orten liegen wird, reduzieren wir die Menge der Referenzpunkte auf 10%, d.h. auf 100'000 in unserer Beispielsrechnung. Pro Referenzpunkt brauchen wir die Geländeposition (geografische Länge und Breite, evtl. auch die Höhe über Meer) und einen WLAN „Fingerabdruck", d.h. eine Menge „sichtbarer" APs mit ihren Identifikationen und Signalstärken. Die Geländekoordinaten zusammen mit dem „Fingerabdruck" am Referenzpunkt nennen wir eine Referenzmessung. In einem städtischen Gebiet wie Zürich sind im Mittel gut zwanzig APs pro Referenzmessung zugreifbar. Jeder AP wird durch seine eindeutige, sechs Bytes lange MAC-Adresse identifiziert. Die Signalstärke eines APs ist oft nur ganzzahlig in Dezibel-Milliwatt (dBm) abrufbar und benötigt daher nur ein Byte. Unter der Annahme, dass eine geographische Position mit zwölf Bytes ausreichend genau gespeichert werden kann, kommen wir auf einen Speicherbedarf von $100000 \cdot (12 + 20 \cdot (6 + 1))$ Byte, also etwa 14.5 MiByte. Dieser Wert ist als untere Grösse zu verstehen, da in der Praxis noch Speicher für die dynamische Verwaltung der Daten notwendig ist. Für ein aktuelles mobiles Gerät sind 14.5 MiByte Arbeitsspeicher, die permanent für die Lokalisierung zur Verfügung stehen müssen und somit von anderen Anwendungen nicht genutzt werden können, nicht wenig, aber dennoch realisierbar. Dennoch wäre es eine unnötige Arbeitsspeicherverschwendung, das ganze lokale Datenset permanent im Hauptspeicher zu haben, wie das einsichtige Beispiel mit Zürich Limmatquai und Oerlikon gezeigt hat.

Positionsbestimmung

Im Zentrum steht die Positionsbestimmung, welche möglichst genau erfolgen soll und nur so viel Ressourcen verwendet, dass die Lokalisierung ohne negative Auswirkungen in anderen mobilen Anwendungen eingesetzt werden kann. Wir gehen davon aus, dass ein lokales Set von Referenzmessungen, die Referenzdaten, und der aktuelle „Fingerabdruck" verfügbar sind. Ein „Fingerabdruck" F besteht aus einer Menge von Paaren (ID_j, P_j), wobei ID_j eine eindeutige Identifikation für den Zugriffspunkt AP_j ist, z.B. seine MAC-Adresse, und P_j die am Erfassungspunkt des „Fingerabdrucks" festgestellte Signalstärke des AP_j darstellt. Eine Referenzmessung R besteht aus einer geographischen Position (x,y,z) und einem „Fingerabdruck" F.

Das Problem der Positionsbestimmung besteht nun aus zwei Teilproblemen: In einem ersten Schritt sollen n möglichst relevante Referenzmessungen $R_{i \in \{1..n\}}$ aus den Referenzdaten ausgewählt und in einem zweiten Schritt daraus dann die gesuchte Position berechnet werden. Damit die Positionsberechnung zu einem vernünftigen Resultat führt, ist es entscheidend, dass die verwendeten Referenzmessungen R_i in nächster Umgebung der zu bestimmenden Position liegen und diese wenn möglich umgeben. Je besser dieses Ziel erreicht wird, desto geringer ist der Einfluss des angewandten Verfahrens auf die nachfolgende Positionsberechnung.

Wenden wir uns nun zuerst dem ersten Teilproblem zu. Um die relevanten Referenzmessungen aus den verfügbaren Referenzdaten herauszufiltern benötigen wir ein sinnvolles Ähnlichkeitsmass. Wie auch bei echten Fingerabdrücken sind charakteristische Gemeinsamkeiten gefragt. In unserem Falle sind dies vor allem die gemeinsamen ID_j des aktuellen „Fingerabdrucks" F_0 und eines beliebig anderen „Fingerabdrucks" F_i. Ähnliche Referenzmessungen haben folglich möglichst viele gemeinsame ID_j und vorzugsweise keine, die im aktuellen Fingerabdruck F_0 nicht vorhanden sind. Des weitern sollten die Signalstärken P_j der gemeinsamen APs ähnlich gross sein.

Als geeignetes Ähnlichkeitsmass a hat sich nach mehreren Testreihen $a = 0.9\ a' + 0.1\ a''$ herausgestellt, wobei a' ein Mass für die Übereinstimmung der Zugriffspunkte und a'' ein Mass für ähnlich grosse Signalstärken der gemeinsamen P_j darstellen:

- $a' = \frac{1}{2}(k/m_0 + k/m_i)$ mit k = Anzahl gemeinsamer ID_j, m_0 = Anzahl APs von F_0 und m_i = Anzahl APs von F_i;
- $a'' = 1/(1 + d/k)$ mit d^2 = Summe der quadratischen Differenzen der Signalstärken der gemeinsamen APs.

Die Gewichtung zwischen a' und a'' zeigt deutlich, dass in erster Linie die Übereinstimmung der MAC-Adressen wichtig ist und erst in zweiter Linie auf die Signalstärken abgestellt werden soll.

Die n Referenzmessungen R_i mit dem höchsten Ähnlichkeitsmass werden nachfolgend *NearestNeighbours* genannt, der Hoffnung verleihend, dass diese Referenzmessungen wirklich nahe bei der aktuellen Position liegen. Nur wenn die *NearestNeighbours* in der Nähe der zu bestimmenden Position liegen, kann eine Interpolation der Positionen der Referenzmessungen zu einer guten Schätzung der wirklichen Position führen. In Abbildung 3 sieht man deutlich, dass durch Auswahl der beiden unteren Referenzmessungen als *NearestNeighbours* (als dunkle Kreise dargestellt) fast unmöglich die gesuchte Position (dargestellt durch das Männchen) durch Interpolation gefunden werden kann.

Die eigentliche Bestimmung der Position verwendet nun die n *NearestNeighbours*. Als praktikable Werte für n haben sich 2 und 3 gezeigt. Mehr als drei Referenzmessungen erhöhen im Allgemeinen die Unsicherheit und bringen insbesondere bei einer rasterförmigen Verteilung der Referenzmessungen wenig zusätzlichen Gewinn.

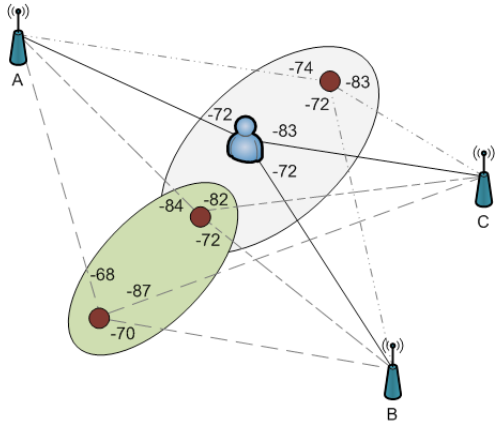

Abbildung 3: Die Auswahl der Referenzmessungen entscheidet darüber, ob eine Positionsabschätzung durch Interpolation erfolgsversprechend ist oder nicht (Signalstärken in dBm)

Ob zwei oder drei geeigneter ist, hängt stark von der Erfassungsart der Referenzdaten ab. Werden die Referenzmessungen in mehr oder weniger regelmässigen Abständen entlang einer Strasse durchgeführt, so haben diese Referenzdaten eine linienförmige Verteilung und daher führt die Interpolation von lediglich zwei Referenzmessungen zum besten Ergebnis. Haben die Referenzdaten dagegen eine flächenförmige Verteilung, so macht eine Interpolation der drei nächsten Referenzmessungen Sinn. Als Interpolationsart dient üblicherweise eine lineare Interpolation der (gewichteten) Positionen. Wird auf Gewichte verzichtet, so wird der Mittelwert der Positionen der *NearestNeighbours* berechnet.

Eine Verwendung von Gewichten kann durchaus sinnvoll sein, wenn die Gewichte erlauben, die zu bestimmende Position im richtigen Mass auf die *NearestNeighbours* auszurichten. Im besten Fall handelt es sich bei den Gewichten um grobe metrische Distanzabschätzungen, die auf Basis einer typischen Sendeleistung eines WLAN-Zugriffspunktes (z.B. 25 mW) und eines Ausbreitungsmodells basieren. Als Ausbreitungsmodelle bieten sich Freiraumdämpfung (nur für Sichtverbindungen zutreffend) oder spezielle empirische Modelle (z.B. modifizierte Version von Okumura-Hata) an [Sta01]. Die Freiraumdämpfung in Dezibel beträgt: $32.44 + 20 \log_{10} f + 20 \log_{10} d$, wobei f die Trägerfrequenz in MHz und d die Distanz in km sind. Entlang von Strassen besteht jedoch oft eine Fast-Sichtverbindung zwischen dem Empfänger und den stärksten APs in der Umgebung, d.h. eine direkte Sichtverbindung ist meistens durch genau eine Hausmauer unterbrochen, da die APs typischerweise in Gebäuden platziert werden. Die zusätzliche Dämpfung kann durch stärkere Gewichtung der Frequenz und der Distanz berücksichtigt werden. In Anlehnung an das COST231 Hata-Modell [COST] beträgt dann die Signaldämpfung bei 2400 MHz (WLAN 802.11g) ca.: $32.44 + 33.9 \log_{10} 2400 + 38.4 \log_{10} d$ dB. Da die Signaldämpfung mit der Differenz aus Sende- und Empfangsleistung (jeweils in dBm gemessen) übereinstimmen muss (etwelche Antennengewinne sind hier vernachlässigt worden), kann daraus die Distanz d grob angenähert werden.

Anstatt die Positionen der *NearestNeighbours* zu interpolieren, könnten auch die Positionen der APs mit Hilfe der angenäherten Distanzen und einer Triangulierung bestimmt werden. Die ermittelten, angenäherten Positionen der APs würden danach in einem zweiten Schritt benutzt, um wiederum mit angenäherten Distanzen und Triangulierung die gesuchte Position zu bestimmen. Wie aber bereits in der Einleitung erwähnt, sollte hierbei, um vernünftige Resultate zu erhalten, mit grosser Treffsicherheit zwischen Sichtverbindungen und Nicht-Sichtverbindungen unterschieden werden können.

Unter der Annahme einer Metrik zur Abschätzung der Distanz $d_i > 0$ zwischen der zu bestimmenden Position und dem *NearestNeighbour* R_i (Details siehe Tests) wählen wir das zur Position von R_i gehörende Gewicht $w_i = \left(d_i \sum_{i=1}^{n} \frac{1}{d_i} \right)^{-1}$. Dadurch führen kürzere Distanzen zu höheren Gewichten und gleichzeitig widerspiegeln die Gewichte die Verhältnisse zwischen den Distanzen. Für den Fall, dass $d_i = 0$ ist, setzen wir $w_i = 1$ und alle anderen Gewichte auf null. Die gesuchte Position (x,y,z) berechnen wir schliesslich als gewichtete Interpolation der n Positionen der *NearestNeighbours*: $x = \Sigma\, w_i\, x_i$, $y = \Sigma\, w_i\, y_i$, $z = \Sigma\, w_i\, z_i$.

Abschätzung der Genauigkeit

Um den erwarteten Lokalisierungsfehler abschätzen zu können, gehen wir von einer rasterförmigen Verteilung der Referenzmessungen und einer Koordinatenauflösung in ganzen Metern aus. Sei $2s$ die Gitterlänge in Metern und nehmen wir weiter an, dass an allen Gitterpunkten eine Referenzmessung vorliegt, dass die Testpunkte gleichförmig verteilt sind und dass ein perfekter Lokalisierungsalgorithmus die zu bestimmende Position aus den Positionen der Referenzmessungen durch lineare Interpolation bestimmt. Unter diesen Annahmen lassen sich folgende mittlere Lokalisierungsfehler e_n in Abhängigkeit der n bestgeeigneten Referenzpositionen bestimmen.

Bei $n = 1$ kann die zu bestimmende Position nicht linear interpoliert werden und daher liegt für jeden Testpunkt die optimal bestimmte Position genau an der Stelle der nächstliegenden Referenzmessung, welche maximal $s\sqrt{2}$ entfernt liegt. Betrachtet man eine quadratische Rasterzelle mit Seitenlänge $2s$, so resultiert der folgende mittlere Positionierungsfehler: $e_1 = \frac{1}{4s^2} \sum_{x=-s}^{s-1} \sum_{y=-s}^{s-1} \sqrt{x^2 + y^2}$.

Für ein praktisches Beispiel mit Rastergrösse 10

m (s = 5) resultiert ein erwarteter mittlerer Positionierungsfehler e_1 von 3.85 m.

Bei n = 2 muss für jeden Testpunkt das optimale Paar von Referenzmessungen berücksichtigt werden. Optimal in dem Sinne, dass die Distanz zwischen dem Testpunkt und einer linear interpolierten Position zwischen den beiden Referenzmessungen minimal ist. So können die beiden bestgeeigneten Referenzmessungen horizontal, vertikal oder diagonal zueinander liegen. Aus Symmetriegründen reicht es jedoch aus, von einer vertikalen und diagonalen Ausrichtung auszugehen und die horizontale Distanz mit der diagonalen zu vergleichen. Die minimale Distanz $d(x,y)$ an der relativen Position (x,y) berechnet sich dann wie folgt:

$$d(x,y) = \min\left(x, \sqrt{\left\lfloor \frac{s-y-x}{2} \right\rfloor^2 + \left\lceil \frac{s-y-x}{2} \right\rceil^2} \right).$$

Wiederum für eine quadratische Rasterzelle mit Seitenlänge $2s$ betrachtet, ergibt sich der folgende mittlere Positionierungsfehler:

$$e_2 = \frac{1}{4s^2}\left(\left(8\sum_{y=1}^{s}\sum_{x=0}^{s-y} d(x,y) \right) + 4\sum_{x=0}^{s} d(x,0) \right).$$

Bei einer Rastergrösse von 10 m reduziert sich der erwartete mittlere Positionierungsfehler e_2 auf 0.73 m.

Für $n \geq 3$ ist der erwartete mittlere Positionierungsfehler gleich null, weil bei optimaler Bestimmung der Referenzmessungen immer drei umliegende Referenzmessungen vorhanden sind und dadurch jede Position in der Ebene durch lineare Interpolation genau berechnet werden kann.

Metriken

Die Gewichte bei der Interpolation der Positionen der *NearestNeighbours* werden wie bereits erwähnt aus angenäherten „Distanzen" berechnet. In unseren Tests haben wir diese Distanzen d_i mit drei verschiedenen Metriken berechnet:

M_1: $d_i = a_i$, wobei a_i dem Ähnlichkeitsmass der Referenzmessung R_i entspricht, welches bereits zur Auswahl der *NearestNeighbours* verwendet worden ist;

M_2: $d_i = \max_j \left| d_{ij} - d_{0j} \right|$, wobei das Maximum über allen k gemeinsamen ID_j des aktuellen (F_0) und des i-ten „Fingerabdrucks" ermittelt wird, und die Distanzen d_{ij} den metrischen Distanzen (berechnet mit dem Hata-Modell) zwischen R_i und AP_j entsprechen;

M_3: $d_i = \sqrt{\frac{1}{k}\sum_{j=1}^{k}(d_{ij} - d_{0j})^2}$ entspricht der Wurzel der mittleren quadratischen Abweichung.

Oft wird in ähnlichen Projekten mit künstlichen Testdaten gearbeitet. Wir haben stattdessen eine mobile Applikation zur Testdatenakquisition erstellt, welche in regelmässigen Intervallen alle „sichtbaren" WLAN-Zugriffspunkte und die aktuelle GPS-Position aufzeichnen kann. In verschiedenen Städten (Zürich, Basel, Brugg) haben wir damit entlang einiger wichtigen Strassen Testdaten erfasst (siehe auch Abb. 1).

Testdaten

Aus den umfangreichen Testdaten präsentieren wir hier die Daten Zürich in zwei Datensätzen:

- ZH1 ist ein kleiner Ausschnitt und entspricht in etwa dem Gebiet aus Abbildung 1 rechts, er enthält etwa 44 verwendete Referenzmessungen;
- ZH2 ist ein wesentlich umfassenderer Datensatz mit ca. 1560 verwendeten Referenzmessungen.

Ein Testlauf besteht aus der zufälligen Datenaufteilung (80% Referenzmessungen, 20% Testdaten), der Reduktion der Referenzmessungen auf ein 10 m Raster und der Durchführung von durchschnittlich k Positionsberechnungen. Um den Realitätsgrad noch zu erhöhen, müssen bei den Testdaten einige APs zufällig ausgeschaltet, d.h. aus dem „Fingerabdruck" entfernt werden, denn bei WLAN-Zugriffspunkten sollte man nicht davon ausgehen, dass sie ständig in Betrieb sind. Für beide Datensätze werden mit den unterschiedlichen Parametrisierungen die ermittelten Positionierungsfehler und anderen Messwerte über 20 Testläufe gemittelt. Als Parametrisierung verwenden wir die Anzahl *NearestNeighbours* (n = 1..5), die Metrik (M_1 bis M_3) und den prozentualen Einschaltungsgrad (100%, 80%, 60%, 40%, 20%).

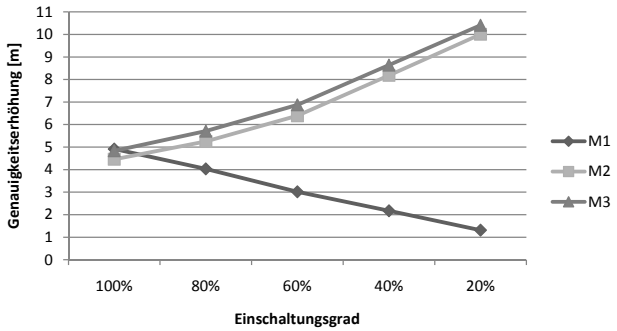

Abbildung 4: Einfluss der Metriken M1 bis M3 auf die Genauigkeitserhöhung durch gewichtete Interpolation im umfangreicheren Datensatz ZH2 für n = 3

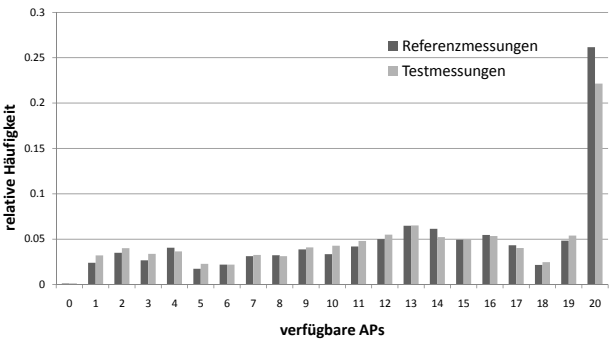

Abbildung 5: Relative Häufigkeit der Anzahl der verfügbaren WLAN-Zugriffspunkte von Referenz- und Testmessungen in der Stadt Zürich

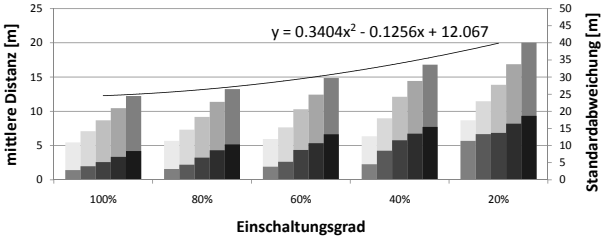

Abbildung 6: Durchschnittliche Entfernungen (und Standardabweichungen) zwischen dem Testpunkt und den n ausgewählten Referenzmessungen

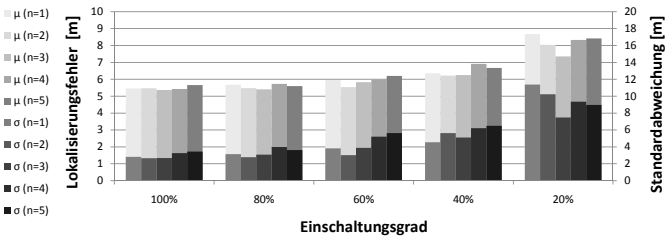

Abbildung 7: Durchschnittliche Lokalisierungsfehler (und Standardabweichungen) in Abhängigkeit der Anzahl n ausgewählter Referenzmessungen und des Einschaltungsgrads

Resultate

Die verschiedenen Tests zeigen, dass der Einfluss der Metriken auf die Lokalisierungsgenauigkeit bei der Interpolation gering ist. In Abbildung 4 ist stellvertretend für alle Messungen der Genauigkeitsunterschied zwischen der gewichteten und der ungewichteten Interpolation für den Datensatz ZH2 mit $n = 3$ in Abhängigkeit der eingeschalteten APs und der verwendeten Metrik ersichtlich. Der mittlere Lokalisierungsfehler reduziert sich bei 100% Einschaltungsgrad von 14.8 um 4.8 m auf 10 m. Die gewichtete Interpolation bringt somit eine deutliche Verbesserung gegenüber der ungewichteten Interpolation; die Wahl der Metrik ist relevant, da der positive Einfluss auf die Lokalisierungsgenauigkeit von M_1 deutlich geringer ausfällt als bei den beiden anderen Metriken. Der Unterschied zwischen M_2 und M_3 ist jedoch gering, wobei in beiden Testsets in allen Testläufen M_3 immer leicht bessere Resultate brachte. Daher werden wir die folgenden Resultate nur für Metrik M_3 zeigen.

Pro Datensatz analysieren wir zuerst die Auswahl der *NearestNeighbours*, ermitteln dann die Lokalisierungsfehler und setzen diese letztlich mit den theoretischen Untergrenzen in Beziehung.

Datensatz ZH1: In diesem kleinen Datensatz werden nur jeweils 44 Referenzmessungen verwendet und durchschnittlich 40 Testmessungen durchgeführt. Zur Beurteilung der Güte der Auswahl der *NearestNeighbours* betrachten wir die durchschnittliche Distanz (und Standardabweichung) der *n NearestNeighbours* von der Testposition (Abb. 6). Für $n = 1$ haben wir an früherer Stelle gesehen, dass in einem 10 m Raster bei perfekter Auswahl des nächsten Nachbarn die Distanzen zwischen 0 und $5\sqrt{2} \approx 7$ m liegen und die mittlere Distanz 3.85 m beträgt. Unsere Tests zeigen nun, dass die Auswahl der *n NearestNeighbours* nicht allzu schlecht funktioniert, da der nächste Nachbar im Mittel zwischen 5.5 (bei 100%) und 8.7 m (bei 20%) liegt. Die mittlere Distanz nimmt mit n linear und mit abnehmendem Einschaltungsgrad leicht quadratisch zu.

In Abbildung 7 sind schliesslich die mittleren Lokalisierungsfehler in Abhängigkeit des Einschaltungsgrads dargestellt. Die besten Resultate sind bei $n = 2$ oder $n = 3$ je nach Einschaltungsgrad gemessen worden, was nicht weiter überrascht. Dass die mittleren Lokalisierungsfehler bis zum Einschaltungsgrad 40% recht stabil bleiben, zeigt, dass die Auswahl der *NearestNeighbours* gut funktioniert. Vergleicht man die Werte aus Abbildung 7 mit denjenigen aus Abbildung 6, so sieht man für $n > 1$ deutlich, dass durch gewichtete Interpolation eine Position berechnet werden kann, deren Lokalisierungsfehler geringer ist als die mittlere Distanz vom Testpunkt zu den ausgewählten Referenzpunkten. Es ist aber auch ersichtlich, dass die Hinzunahme eines vierten oder fünften nächsten Nachbarn zu einer Verschlechterung der Lokalisierung beiträgt.

Datensatz ZH2: In diesem umfangreicheren Datensatz werden durchschnittlich 1560 Referenzmessungen verwendet und durchschnittlich 1430 Testmessungen durchgeführt. Nicht an allen gewählten Testpunkten kann eine Lokalisierung durchgeführt werden, da vereinzelt keine WLAN-Signale empfangen worden sind. Bei der Erfassung einer Referenz- bzw. Testmessung sind maximal die zwanzig stärksten Signale berücksichtigt

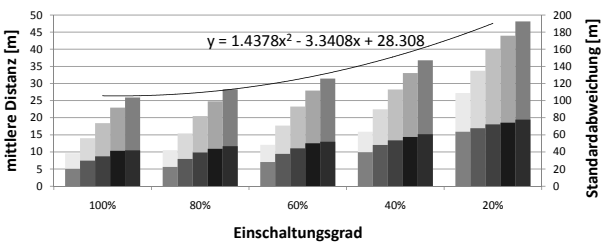

Abbildung 8: Durchschnittliche Entfernungen (und Standardabweichungen) zwischen dem Testpunkt und den n ausgewählten Referenzmessungen

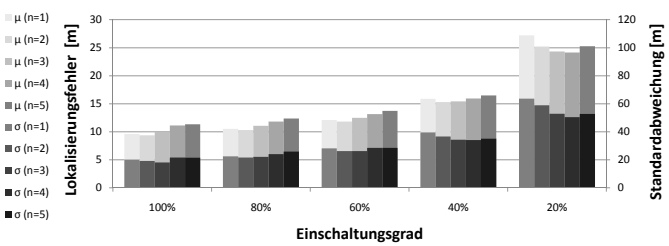

Abbildung 9: Durchschnittliche Lokalisierungsfehler (und Standardabweichungen) in Abhängigkeit der Anzahl n ausgewählter Referenzmessungen und des Einschaltungsgrads

worden (siehe Abb. 5). Zur Beurteilung der Güte der Auswahl der *NearestNeighbours* betrachten wir wiederum die durchschnittliche Distanz (und Standardabweichung) der n *NearestNeighbours* von der Testposition (Abb. 8). Die ausgewählten *NearestNeighbours* liegen nun deutlich weiter entfernt vom Optimum, was damit zusammenhängt, dass etliche Testpunkte nur über ein paar wenige weit entfernte Referenzmessungen verfügen. Der nächste Nachbar liegt im Mittel zwischen 9.6 (bei 100%) und 27.2 m (bei 20%) vom Testpunkt entfernt. Generell Die nimmt die mittlere Distanz mit n linear und mit abnehmendem Einschaltungsgrad leicht quadratisch zu.

Die dazugehörigen Lokalisierungsfehler sind in Abbildung 9 abgebildet. Wiederum zeigt sich, dass $n = 2$ und $n = 3$ zu den besten Resultaten führen, wenn alle Einschaltungsgrade berücksichtigt werden. Bei einem sehr hohen Einschaltungsgrad würde ein nächster Nachbar ausreichen; umgekehrt bei einem sehr tiefen Einschaltungsgrad hilft die gewichtete Interpolation von 3 oder sogar 4 *NearestNeighbours* deutlich zu einer Verbesserung der Lokalisierungsgenauigkeit.

Fazit und Ausblick

Das Resultat unserer Arbeit ist eine Lokalisierungs-API, die man schnell und einfach in eine bestehende Applikation einbinden und anschliessend darüber Positionsinformationen beziehen kann. Die Güte der zurückgelieferten Positionen hängt im Fall der WLAN-Lokalisierung von der Dichte, der Aktualität der Referenzmessungen, der Anzahl bekannter Zugriffspunkte, die zum Zeitpunkt der Testmessung aktiv sind, und vom Interpolationsverfahren ab. Solange mehr als 40% der bekannten APs bei einer Testmessung aktiv sind, funktioniert die Lokalisierung recht stabil. Während in Gebieten mit ausreichender Anzahl WLAN Zugriffspunkten eine Lokalisierungsgenauigkeit von 5 bis 6 m realistisch ist, sinkt die Genauigkeit in „dünn besiedelten" Gebieten auf 10 bis 15 m, je nach Einschaltungsgrad. Die geeignete Metrik zur Auswahl der Referenzpunkte, welche für die Interpolation herbeigezogen werden sollen, schätzen wir weitaus wichtiger ein als das gewählte Interpolationsverfahren, da durch gewichtete Interpolation lediglich Verbesserungen von maximal 10 m erzielt wurden.

Das Abfahren der wichtigsten Strassen und der Aufbau der Referenzdatenbank beanspruchen sehr viel Zeit und Ressourcen. Das Google Streetview-Projekt [GSV] wie auch das Skyhook-Wireless-Projekt zeigen aber, dass gewisse Firmen durchaus bereit sind, einen vergleichbaren Aufwand zu leisten. Eine andere Möglichkeit besteht darin, die Benutzer der Lokalisierungs-API, welche auch ein GPS in ihrem mobilen Gerät integriert haben, als freiwillige Datensammler zu rekrutieren.

Um unsere Lokalisierungstechnik zu testen, haben Jugendliche in der Informatik-Projektwoche von „Schweizer Jugend forscht" eine mobile Applikation entwickelt, mit der sich Wander- und Biketouren sehr einfach aufzeichnen lassen. Eine andere mobile Anwendung liefert Lokalisierungsinformationen in Echtzeit. Dabei übermittelt ein Dienst des Mobiltelefons in regelmässigen Abständen die aktuelle Position an einen Server und verknüpft sie mit zusätzlichen geografischen Informationen. Dadurch werden Anwendungen im Bereich der sozialen Netzwerke ermöglicht, wie z.B. „Wo befindet sich Paul?" oder „Mit wem könnte ich zu Mittag essen?".

Momentan portieren wir unsere Technologie nach Android und erfassen Referenzmessungen innerhalb von mehrstöckigen Gebäudekomplexen, um den Einsatz der WLAN-Lokalisierung auch in grossen Business-Centern oder Shopping-Malls zu demonstrieren.

Referenzen

[COST] COST Hata Model. http://en.wikipedia.org/wiki/COST_Hata_model

[CYC06] Chan, Y.T., Yau, C.H., Ching, P.C. Exact and approximate maximum likelihood localization algorithms. IEEE Trans. Veh. Technology. Vol. 55, 2006.

[DZ02] Dornbusch, P., Zündt, M. Realisierung von Positionsortungen in WLAN. Proc. ITG-Fachtagung „Technologie und Anwendungen für die mobile Informationsgesellschaft", 2002.

[GO07] Gruntz, D., Olloz, S. Erfahrungen mit dem Location API (JSR 179). JavaSPEKTRUM 06, 2007.

[GSV] Google Street View. http://maps.google.com/help/maps/streetview/

[HM09] Häfelfinger, M., Marraffino, F. Lokalisierung in drahtlosen Netzwerken ohne GPS. Informatik-Projektarbeit, FHNW, 2009. http://www.fhnw.ch/technik/imvs/forschung/projekte/lokalisierung-basierend-auf-wlan

[JXZ07a] Xing, J., Zhang, J., Jiao, L., Zhang, X., Zhao, C. A Robust Wireless Sensor Network Localization Algorithm in NLOS Environment. IEEE Int. Conf. Control and Automation, 2007.

[JXZ07b] Xing, J., Zhang, J., Jiao, L., Zhang, X., Zhao, C. LCC-Rwgh: A NLOS Error Mitigation Algorithm for Localization in Wireless Sensor Network. IEEE Int. Conf. Control and Automation, 2007.

[KNRW97] Kreveld, M.v., Nievergelt, J., Roos, T., Wiedmayer, P. Algorithmic Foundations of Geographic Information Systems. LNCS 1340, 1997.

[OS08] Oser, H-P., Stamm, C. HikeTracker – Eine Smartphone-Software zur Unterstützung von Freizeitaktivitäten. IMVS Fokus Report, 2008. http://www.fhnw.ch/technik/imvs/publikationen/artikel-2008/hiketracker

[PL] Place Lab, a privacy-observant location system. http://www.placelab.org

[SF] Ofcom, Mobile Phone Base Station Database. http://www.sitefinder.ofcom.org.uk/

[SHW] Skyhook Wireless, hybrid positioning system XPS. http://www.skyhookwireless.com/

[Sta01] Stamm, C. Algorithms and Software for Radio Signal Coverage Prediction in Terrains. Diss. ETH Zurich, 2001.

[WH96] Wylie, M.P., Holtzman, J. The non-light-of-sight problems in mobile location estimation. IEEE Int. Conf. Universal Personal Communications, 1996.

Erweiterung eines Webdienstes um einen mobilen Webzugang

In diesem Projekt[1] wird ein webbasierter Dienst um einen Zugang für Mobiltelefone erweitert. Die mobile Webapplikation stellt dem Benutzer eine Teilmenge der Funktionalität des Basisdienstes über ein entsprechend optimiertes Benutzerinterface zur Verfügung. Im Artikel werden wichtige Entscheidungen in der Systemarchitektur erklärt, damit die mobile Webapplikation jederzeit flexibel an die Bedürfnisse der Benutzer und an die Vorgaben des Basisdienstes angepasst werden kann. Wir wollen aber auch zeigen, dass sich die Entwicklung einer mobilen Webapplikation nicht gross von der Entwicklung einer normalen Webapplikation unterscheidet.

Jürg Luthiger | juerg.luthiger@fhnw.ch

Der mobile Markt ist ein Zukunftsmarkt; entsprechend wichtig ist es die Entwicklungen in diesem Bereich nicht zu verpassen. Ein Dienstleister verspricht sich mit einem mobilen Zugang eine Attraktivitätssteigerung seiner Plattform sowohl für die Benutzer, für die Online Werber wie auch für die Partner. Ausserdem können neue Benutzergruppen oder Geschäftsmodelle erschlossen werden.

Webbasierte Dienste erlauben einen raschen Einstieg in den mobilen Markt. In der Regel kann man einfache Dienste mit einem modernen, mobilen Webbrowser problemlos nutzen. Werden bei der Entwicklung solcher Seiten die Eigenschaften mobiler Browser aber zu wenig berücksichtigt, können unschöne Effekte bei der Darstellung der Webseiten auftreten.

In diesem Artikel wollen wir ein paar Entscheidungen aufgreifen, die wir bei einer konkreten Umsetzung eines mobilen Webzugangs zu einem bestehenden Dienst getroffen haben. Die Überlegungen betreffen die Integration in das Backend System. Es ist wichtig festzuhalten, dass diese Überlegungen im Rahmen einer Prototypen-Entwicklung entstanden sind. Mit diesem Prototyp haben wir die Benutzerbedürfnisse anhand einer konkreten mobilen Anwendung aufgenommen und getestet. Die Erkenntnisse aus diesem Projekt sind zwischenzeitlich in den mobilen Zugang für das Doodle System eingeflossen [DooMob].

Kontext

Doodle AG bietet einen Terminplaner als Gratisdienstleistung für alle Internetbenutzer an [Doodle]. Die Einstiegshürde ist besonders tief, denn Doodle erfordert keine Registrierung, keine Softwareinstallation und ist sehr einfach zu bedienen. Deshalb ist dieser Dienst sehr beliebt und in der Schweiz der klare Führer bei den Terminfindungswerkzeugen und eine der meistbesuchten Websites der Schweiz.

Der Doodle-Dienst unterstützt auf eine effiziente Art die Terminfindung in verteilten Teams. Der Initiator erstellt über ein Webformular eine Terminumfrage, indem er den Zweck und verschiedene Terminvorschläge angibt. Aus der Anfrage generiert Doodle eine Einladung in Form einer eindeutigen Webadresse. Diese URL kann der Initiator nun via E-Mail an das restliche Team senden. Die Teammitglieder wählen anschliessend ihre Präferenzen auf der entsprechenden Webseite und Doodle erstellt eine Statistik der gewählten Termine. So kann das Team sehr schnell einen passenden Termin finden.

Mobile Doodle-Benutzer meldeten jedoch, dass die Webseiten in den mobilen Webbrowsern teilweise falsch dargestellt werden und dass besonders die Termine, die standardmässig in einer grossen Tabelle aufgelistet sind, zu übermässigem Scrollen führen.

Doodle AG hat sich deshalb entschlossen diese Kundenwünsche aufzunehmen und der mobilen Benutzergruppe ein entsprechend optimiertes Webinterface anzubieten.

Problemstellung

Bei der Entwicklung dieses mobilen Dienstes sollen folgende Fragenstellungen adressiert werden:

- Welche Funktionalität soll dem Benutzer zur Verfügung stehen? Obwohl die Benutzung der Doodle-Plattform über den Webclient einfach und selbsterklärend ist, scheint es nicht sinnvoll zu sein, auch für den mobilen Webclient den vollen Funktionsumfang anzubieten. Auf einem mobilen Gerät sind zum Beispiel die Eingabemöglichkeiten beschränkter, vor allem die Eingabe von Text kann mühsam werden.
- Welche Interaktionen sind für den Nutzer mit der mobilen Anwendung sinnvoll? Die Interaktionsmöglichkeiten der Doodle-Plattform sind

1 Mit freundlicher finanzieller Unterstützung der Hasler Stiftung.

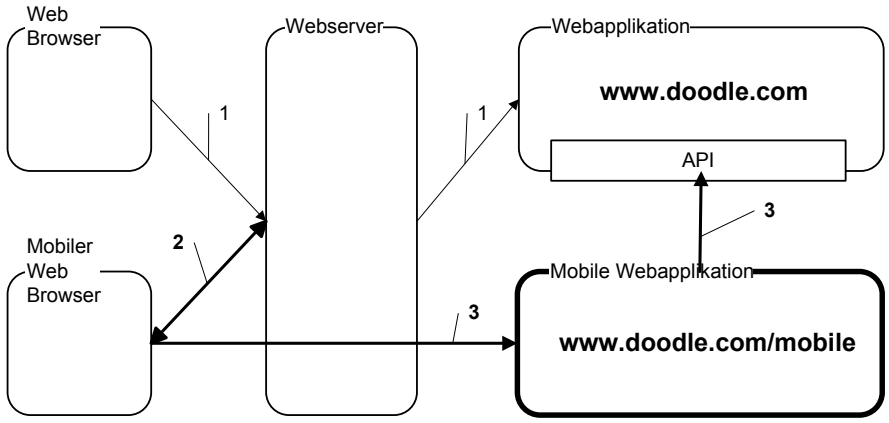

Abbildung 1: Konzept

auf die Möglichkeiten eines Desktop Systems mit Tastatur und Maus ausgelegt. Auf einem mobilen Gerät stehen diese Inputmöglichkeiten jedoch nicht zur Verfügung. Die mobile Plattform muss dies berücksichtigen.

- WiekanndasbestehendeDoodleBackendsinnvoll erweitertwerden,umweiterenClientTypeneinen sicheren, zuverlässigen Zugang zu ermöglichen? Die Systemarchitektur der Doodle-Plattform ist eng mit der Weblösung gekoppelt. Diese enge Koppelung zwischen Backend und Web-Client muss aufgelöst und durch eine lose Koppelung ersetzt werden. Erst dann wird es möglich sein, weitere Client-Typen wie die mobile Plattform in das zentrale Doodle-System einzubinden.

Funktionalität

Mobile Benutzer brauchen die Software-Dienste auch ausser Haus. Ihre Bedürfnisse an den Dienst sind in diesen Situationen in der Regel aber nicht gleich, wie wenn sie zu Hause vor dem PC sitzen und die Site von dort aus besuchen. Oft wollen sie sich aktuelle Informationen beschaffen oder kleinere, klar terminierte Arbeiten erledigen. Es ist deshalb wichtig, den mobilen Nutzer als ein neues Kundensegment zu betrachten und die mobile Website fokussiert auf diese Bedürfnisse auszurichten. Daher kann es durchaus Sinn machen, gewisse Funktionalität des Basisdienstes nicht in der mobilen Version anzubieten. Die Benutzeranalyse zum Beispiel ergab, dass die Teilnahme an einer bereits existierende Terminumfrage im Vordergrund steht, um auch ausser Haus die eigenen Präferenzen bei einer Einladung einreichen zu können.

Da die Funktionalität des mobilen Dienstes in der Regel vom Basisdienst abweicht, ist es wichtig, dass der Basisdienst komplett unabhängig vom mobilen Client bleibt. Darum wird die mobile Website in diesem Projekt als eine eigenständige Webapplikation entwickelt, die parallel zum Basisdienst betrieben werden kann (siehe Abb. 1). Gleichzeitig entsteht auf dem Basisdienst eine Schnittstelle [DooAPI], die von beliebigen Clients programmatisch genutzt werden kann und auch von der mobilen Webapplikation angesprochen wird. Diese starke Separierung ergibt eine grosse Unabhängigkeit und erlaubt das problemlose Testen verschiedener Alternativen auf der mobilen Seite. Dies ist ein grosser Vorteil vor allem in der Phase des Prototypenbaus.

Interaktionsmöglichkeiten

Die modernen, mobilen Webbrowser beherrschen die aktuellen Webstandards. Deshalb ist ein Einsatz veralteter WAP-Technologien [WAP] überflüssig. Stattdessen lassen sich die heute gängigen Webtechnologien einsetzen, wie XHTML und CSS und zur dynamischen Aufbereitung Java Server Faces (JSF). JavaScript kann ebenfalls genutzt werden.

Der Zugang auf eine mobile Website muss möglichst einfach gestaltet werden, da das Eintippen einer URL auf einem mobilen Gerät mühsam ist. Das Platzieren des entsprechenden Links auf der Startseite des Basisdiensts stellt eine erste Variante dar. Man kann jedoch auch die Header-Informationen einer Browseranfrage nutzen, um eine Umleitung auf eine eigenständige mobile Site automatisch zu aktivieren und den Benutzer komplett von einer manuellen Eingabe zu befreien. Da jeder Browsertyp eine eigene Kennung hat – sie ist unter dem Schlüsselwort „user-agent“ im Header abgelegt – ist auf dem Server eine Identifikation des Browsers problemlos möglich. In dieser zweiten Variante wird die Webapplikation als auch die mobile Webapplikation über die gleiche URL angesprochen, zum Beispiel mit www.doodle.com (siehe Abb. 1). Mit den Header-Informationen aus dem *Request* wird nun entschieden, ob die Anfrage von einem normalen oder von einem mobilen Browser stammt. Im ersten Fall wird der Webserver die Anfrage an den Basisdienst weiterleiten (siehe 1 in Abb. 1) und im zweiten Fall wird eine geeignete Komponente im Webserver ein *Redirect* auf die mobile Webapplikation initiieren (2). Die mobile Applikation kann dann über das Doodle API mit dem Basisdienst kommunizieren (3).

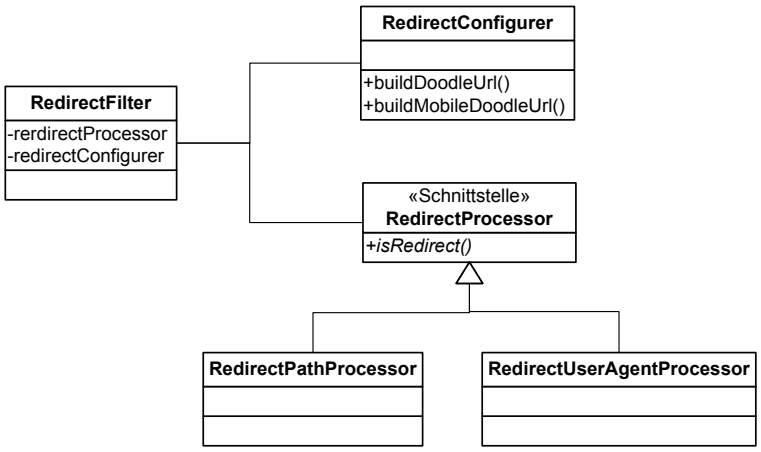

Abbildung 2: Klassendiagramm Filterkonzept

Die Erkennung des Browser-Typs auf dem Server wird in Java am einfachsten mit einer Filterkomponente umgesetzt. In Abbildung 2 ist das entsprechende Klassendiagramm aufgezeichnet, welches aus der Anforderung entstanden ist, Varianten für das Umleiten (*Redirect*) austesten zu können. Der Filter hat dabei je eine Referenz auf die Klasse *RedirectConfigurer* und das Interface *RedirectProcessor*, zu welcher momentan zwei konkrete Implementierungen existieren. Die Klasse *RedirectPathProcessor* dient lediglich zu Testzwecken; sie untersucht den Pfad der Anfrage. Die Klasse *RedirectUserAgentProcessor* hingegen verarbeitet die Header-Informationen, im Speziellen den Eintrag in der Zeile „user-agent", um eine Umleitung feststellen zu können. Dank dieser Entkoppelung der Funktionalität reduziert sich die Logik in der Filterklasse *RedirectFilter* zu den Schritten in Listing 1.

In Zeile 7 wird der aktuell konfigurierte *processor* aufgerufen, um festzustellen ob eine Umleitung ausgelöst werden muss. Falls ja, erstellt der *configurer* die entsprechende URL, welche nun über die *response* an den Browser zurückgeschickt wird. Falls nein, kann der *request* in der Filterkette weitergegeben werden (Zeile 11). In den Zeilen 2 bis 5 erkennt man den Einsatz des Spring Frameworks [Spring]. Die Konfiguration der Klassen *RedirectProcessor* und *RedirectConfigurer* können dank dieses Frameworks externalisiert werden und erlauben eine weitere Abstraktion der Klassen untereinander, was für das flexible Testen der verschiedenen Komponenten sehr nützlich ist.

Anbindung an das Doodle Backend

Wie in Abbildung 1 dargestellt, wird das Basissystem über eine Schnittstelle (API) in die mobile Webapplikation integriert. Es ist ganz wichtig, dass die Funktionalität nicht in den Doodle Clients dupliziert werden muss und deshalb hat die Gestaltung der Schnittstelle eine grosse Bedeutung. Das API soll alle Anwendungsfälle unterstützen, die über die Client-Applikation abgewickelt werden müssen. In einer Projektphase, wo eine solche Schnittstelle ständigen Anpassungen aus praktischen Gründen unterworfen ist, macht es Sinn, ein Beschreibungskonzept zu wählen, welches sehr flexibel ist und dadurch schnelle Änderungen erlaubt.

Wir haben die Doodle-Schnittstelle in der REST-Technologie gestaltet und umgesetzt. So kann gleichzeitig an der Client-Applikation entwickelt und serverseitig die Schnittstelle den Bedürfnissen des Clients angepasst werden. Die Beschreibung der Schnittstelle erfolgt mit einem

```
1   ...
2   WebApplicationContext wac =
3       WebApplicationContextUtils.getRequiredWebApplicationContext(
            getServletContext());
4   RedirectProcessor processor =
        (RedirectProcessor)wac.getBean("redirectProcessor");
5   RedirectConfigurer configurer =
        (RedirectConfigurer)wac.getBean("redirectConfigurer");
6
7   if (processor.isRedirect(request)) {
8       String redirectString =
            configurer.buildMobileDoodleUrl(pollId, processor.getDetails());
9       response.sendRedirect(response.encodeRedirectURL(redirectString));
10  } else {
11      chain.doFilter(request, response);
12  }
13  ...
```

Listing 1: Filterlogik

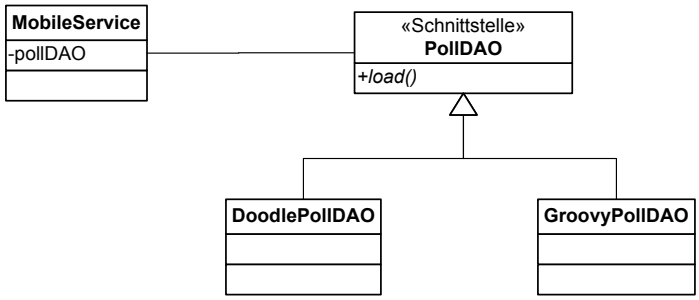

Abbildung 3: Einsatz des DAO Patterns zur Abstraktion der Doodle Systemintegration

XML-Schema, welches genutzt werden kann, um mit einem geeigneten Framework entsprechende Java-Klassen zum Lesen und Schreiben der Kommunikationsdaten automatisch zu generieren. Sie erlauben das Umwandeln von Java-Objekten in eine XML-Darstellung und umgekehrt. Bei Schemaänderungen werden diese Klassen neu generiert, was vollständig automatisiert werden kann. Weitere Anpassungen im Klassenmodell des Clients sind nur bei grösseren Änderungen in der Schnittstelle erforderlich. Jedoch sind mit der XML-Variante Anpassungen zur Laufzeit nicht möglich.

Während der Entwicklungsphase können die Änderungen aber doch gross sein. Daten müssen zum Beispiel besser strukturiert und in verschiedene Klassen aufgeteilt werden. In dieser Phase empfiehlt sich der Einsatz einer Skriptsprache, da diese Sprachen in der Regel eine kompaktere Implementierung und Anpassungen auch zur Laufzeit erlauben.

Im Java-Umfeld ist Groovy [Groovy] ein möglicher Kandidat. Mit Groovy ist eine sehr kompakte Implementierung der Anbindung möglich. Listing 2 stellt das Groovy-Skript dar, um über eine gegebene URL das XML-Dokument auf der Serverseite auszulesen (Zeile 4), den Inhalt zu parsen und das entsprechende Domänenmodell aufzubauen (Zeilen 6 bis 31). Die entsprechende Implementierung der Variante mit dem XML-Schema ist um ein Mehrfaches grösser und umfasst über 200 Zeilen Java Code.

Um eine Abstraktion zwischen der mobilen Webapplikation *MobileService* und den Server-

```
1   ...
1 class GroovyPollDAO implements PollDAO {
2       Poll load(String id) {
3           def poll = new Poll()
4           def pollSrc = new XmlParser().parse(«$id»)
5
6           poll.description = pollSrc.description.text()
7           poll.title = pollSrc.title.text()
8           poll.initiator = pollSrc.initiator.text()
9           poll.type = (pollSrc.type.text == ‹DATE›) ? poll.type = Poll.Type.DATE
10                                          : poll.type = Poll.Type.TEXT
11          poll.timeZone = pollSrc.timeZone.text()
12          poll.levels = pollSrc.levels.text()
13
14          pollSrc.participants.participant.each {participant ->
15              def vote = new Vote(participant.name.text(), new LinkedList())
16              participant.preferences.option.each { option ->
17                  vote.votes << option.text()
18              }
19              poll.votes << vote
20          }
21
22          pollSrc.options.option.each { opt ->
23              def option
24              if (poll.type == Poll.Type.DATE) {
25                  option = new DateOption(opt.text(), Locale.GERMAN)
26              } else if (poll.type == Poll.Type.TEXT) {
27                  option = new TextOption(opt.text())
28              }
29              poll.options << option
30          }
31          return poll
32      }
33  }
```

Listing 2: Groovy Skript für die flexible Anbindung des Doodle Backends

Zugriffsklassen wie das Groovy Skript zu erhalten, empfiehlt sich der Einsatz des Data Access Object (DAO) Design-Patterns (Abbildung 3). Das DAO übernimmt die Kommunikation mit dem Server und die Umwandlung der Daten in das korrekte Schnittstellenformat. Durch die strikte Trennung zwischen Nutzung und Besorgung der Daten können verschiedene Implementierungen zur Serveranbindung einfach getestet werden, da sie ohne Auswirkungen auf den Client austauschbar sind.

Zusammenfassung

Die Entwicklung einer mobilen Webapplikation unterscheidet sich nicht wesentlich von der Entwicklung einer normalen Webanwendung; es lassen sich die gleichen Technologien einsetzen. Das Spring Framework [Spring] bildet bei unserem Prototyp das Fundament. Mit diesem Framework kann man sehr einfach verschiedene Implementierungen testen. Bei der View-Technologie kam Java Server Faces (JSF) zum Einsatz, da Doodle JSF auch für ihre normale Webapplikation nutzt.

Es hat sich in unsere Analyse gezeigt, dass die modernen mobilen Webbrowser sehr mächtig sind und viele Webstandards (XHTML, CSS, JavaScript) unterstützen. Ebenfalls sind die mobilen Telefone mit den entsprechenden Webbrowsern leistungsfähig genug, um eine normale Webseite in nützlicher Zeit aufzubereiten und darzustellen. Dennoch gibt es zwischen den Browsertypen immer wieder kleinere Unterschiede, die nur durch entsprechende Tests eruiert werden können. Wir haben während der Analysephase alle Interaktionselemente, die zum Einsatz kommen sollen, mit den verschiedenen Browsern auf Darstellung und Benutzung untersucht. Grosse Unterschiede zeigen sich vor allem zwischen den Browsern auf normalen mobilen Telefonen und Smartphones. So erstaunt zum Beispiel sehr, wie unterschiedlich eine Auswahlliste mit Mehrfachauswahl dargestellt werden kann.

Bei der Portierung einer Webapplikation auf mobile Geräte muss das mobile User Interface genau analysiert werden. In vielen Fällen wird man die Benutzeroberfläche neu gestalten müssen, um den kleineren Bildschirmen, der fehlenden Tastatur und Maus Rechnung tragen zu können. Es ist zum Beispiel wichtig, die Texteingaben auf ein Minimum einzuschränken, da dies auf einem Handy viel schwerfälliger ist als bei einem normalen Computer mir Tastatur. Für einen Teil der normalen Webseiten kann man mit einer Anpassung des CSS Stylesheets die Portierung an die mobilen Geräte lösen. Muss der Informationsgehalt hingegen verdichtet werden, wird die Entwicklung einer eigenen Webseite unumgänglich.

Die Browserkennung über das Header-Kennwort „user-agent“ wird von den verschiedenen Browser sehr gut und detailliert unterstützt. Das Kennwort kann auf der Serverseite sinnvoll genutzt werden, um zum Beispiel Serverdienste oder CSS Stylesheets dem Browser entsprechend programmatisch auszuwählen.

Wir haben unseren Prototyp als eigenständige Webapplikation realisiert, um aus Gründen der unterschiedlichen Tests eine möglichst lose Kopplung an den Doodle-Dienst zu etablieren. Dabei haben wir gesehen, dass eine flexible Serveranbindung bei einem instabilen Server-API am besten mit einer Skriptsprache zu realisieren ist. Sobald sich das API aber stabilisiert hat, kann ein Wechsel zu einer anderen, engeren Serveranbindung aus Performanzgründen Sinn machen.

Doodle hat in der Zwischenzeit auf ihrem produktiven System einen mobilen Zugang aufgeschaltet [DooMob]. Dabei haben sie verschiedene Resultate aus dem vorliegenden Projekt aufgenommen. Besonders die Erkenntnisse bei der Gestaltung der Benutzeroberfläche sind sehr wichtig gewesen. Für die Implementierung der mobilen Webapplikation haben sie selber eine engere Anbindung an den Server gewählt und somit die aus einer Anbindung über das Doodle-API resultierenden Performanceverluste verhindert.

Referenzen

[Doodle] Website Doodle AG, http://doodle.com/

[DooAPI] RESTful Doodle, http://www.doodle.com/xsd1/RESTfulDoodle.pdf

[DooMob] Mobile Website von Doodle AG, http://doodle.com/mobile/main.html

[Groovy] Groovy Website, http://groovy.codehaus.org/

[Spring] Website Spring Framework, http://www.springsource.org/

[WAP] Wireless Application Protocol, Wikipedia, http://de.wikipedia.org/wiki/Wireless_Application_Protocol

[Zob01] Zobel, J. Mobile Business und M-Commerce. Hanser, 2001

Reverse Generation and Refactoring of Fit Acceptance Tests for Legacy Code

The Fit framework is a well established tool for creating early and automated acceptance tests. Available Eclipse plug-ins like FITpro support for new requirements and new code writing of test data and creation of test stubs quite well. In our project we faced the problem, that a large legacy system should undergo a major refactoring. Before this, acceptance tests had to be added to the system to ensure equivalent program behavior before and after the changes. Writing acceptance tests manually for existing code is very laborious, cumbersome and very costly. However reverse generation of fit tests based on legacy code is not foreseen in the current Fit framework, and there are no other tools available to do so. So we decided to develop a tool which allows generation of the complete Fit test code and test specification based on existing code. The tool also includes automatic refactoring of test data when refactoring production code and vice versa, when changing the Fit test specification, it also updates production code accordingly. This reduces the maintenance effort of Fit tests in general and we hope, this will help to spread the usage of Fit for acceptance and integration testing even more.

Martin Kropp, Wolfgang Schwaiger | martin.kropp@fhnw.ch

For clarification we use the following terms for the rest of the proposal[1]. Production code is the code of the application, i.e. the source code of a system under test (SUT) and of a method under test (MUT), respectively. Test code in general reflects the code that is written to execute a test. Test data covers both input and expected data for a test. Moreover, a test case includes test data and the preconditions and expected postconditions. Finally, a test fixture comprises all that is needed to run a test, especially the test code and the test case.

Use Cases

When working with legacy code, a typical scenario might be that a developer has to add new functionality in a certain module. To be able do so, he may have to refactor certain areas in this module. Due to the often encountered lack of modularization in legacy systems and the missing tests, one approach is to add tests starting from the top level (i.e. on acceptance level). Having tests for the top level methods ensures that any misbehavior of the underlying refactored code will be detected immediately. The developer can use our iTDD (integrated Test Driven Development) tool to generate the test code and the related test data, based on the given method signature. The developer can then add new, or edit existing test data in Fit tables form within an IDE (e.g. Eclipse). The tests can be executed immediately and will indicate the success or failure of the test. In case of new code, the process is very similar: following the TDD approach, where the developer first specifies the signature of the new method, he can then generate Fit tests based on the new method. In this case, however, the test will fail as long as the implementation of the new method is not complete. Moreover, during maintenance as a developer refactors a method, the tool will automatically carry along the changes to the test data and the test code. So for a complete and integrated TDD approach we want to address all relevant activities in a software construction process [SWE], which result in the following use cases:

1. Generation of test fixture for legacy code.
2. Generation of test fixture for new code.
3. Refactoring of production code.
4. Refactoring of test code.
5. Refactoring of test cases.

3. Legacy Code Testing with Fit

Fit has been designed to enable early executable acceptance testing [Cun08]. The idea is, that starting from a use case specification, or from one scenario of a use case, the user can derive a concrete test case and specify it in tabular form. This tabular form represents a kind of view mock, which can then be implemented in code as a concrete Fit test. In this case the main purpose of the Fit tests are to validate and verify the user requirements. In the case of testing legacy code however, there is no need for requirements verification. In this case the main purpose of the Fit tests is to ensure the

1 The original version of this paper of the same authors will be presented at the OOPSLA conference. OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

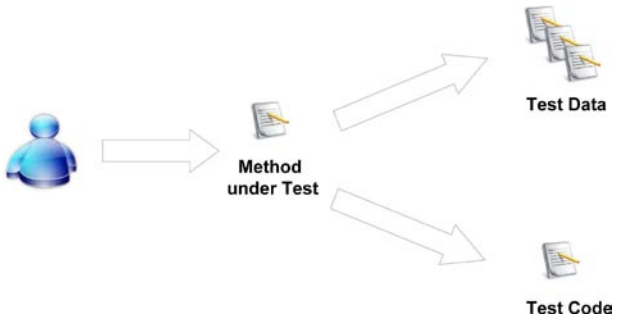

Figure 1: Separating test data from test code

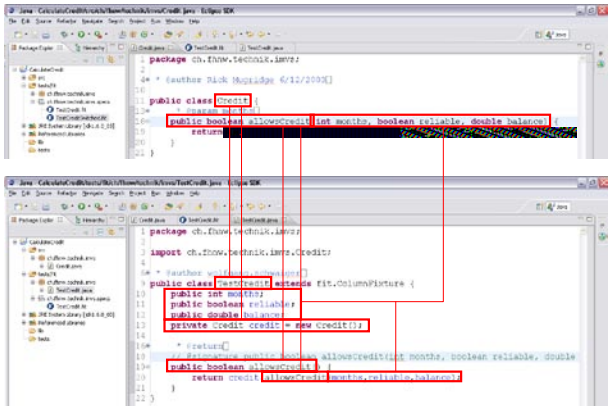

Figure 2: Mapping of production code and generated test code

equivalent behavior of the system on acceptance level when changing the underlying code base, i.e. providing a safe playground when changing the system.

To enable user centered approach for test specification the Fit framework provides a strict separation of test data and test code as indicated in Figure 1. Advantages of this concept are [MC05, Pet08]:

- Users can specify test cases in a user centered view, e.g. in HTML tables or even Excel spreadsheets.
- Test specifications can be defined by the end users and other stakeholders and not only by developers.
- There is only one test code implementation for all related test data of the same kind and thus does not increase with the number of specified test data.
- Test specifications do not change during code refactoring operations.
- Frameworks which use this separation enable better continuous integration of tests.
- Tests can be easily written on different levels of abstraction.

However, no lunch is for free, some drawbacks of this approach are [MC05, Pet08]:

- Connections between test data and test code have to be maintained outside of the test code.
- Developers have to use another, additional test framework.

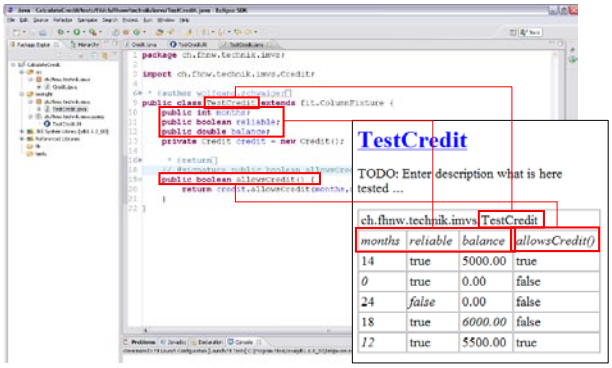

Figure 3: Mapping by convention strategy of test code and test data

Nonetheless, the advantages prevail, especially the significant test code reduction because of one common test code base for test cases of the same kind. This makes it easier to automate the test code generation. Also, the ability for the user to specify tests through the special test data viewer allows more profound test specifications.

iTDD Concepts

Fully automated test maintenance also includes the complete test code generation. Basically, the following two generation schemas can be differentiated.

- For new written applications (new code) the goal of a test suite is to map the functionality to the expected requirements. Usually, TDD starts with the implementation of a method stub – MUT – and then continuous with the specification of the expected behavior of the implementation in form of pre-written test cases. For this step, mainly the user jumps in (e.g. user stories) and leaves the developer with the implementation of the MUT. As mentioned before the tests will fail as long as the implementation is not complete.
- For legacy code the focus is more on the injection of tests into the system to assure not to break the functionality during refactorings, feature extensions, etc. Tests can be added by selecting the desired method in the SUT which is to be monitored and tested during change op-

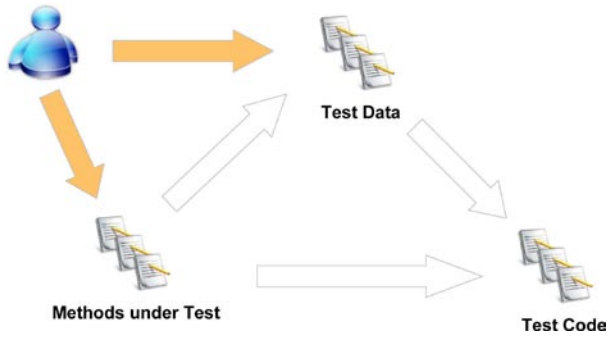

Figure 4: Tracing code changes

```
// production code
package math;
public class Calc {
   . . .
   public float divide (float n, float d) { return n/d ; }
}

// generated test code
package math;
public class CalcTest extends fit.ColumnFixture {
   public float n;
   public float d;
   private Calc calc = new Calc( );

   // test method
   public float d ivide ( ) {
      // call MUT
      return calc.divide ( n , d );
   }
}
```

Listing 1: Test code sample

erations. The tests are generated based on the given method signature and can be run immediately to prove the functionality.

For both cases the iTDD generates the complete Fit test code and the Fit HTML file containing the test specification as illustrated in Figure 3. The automated test generation process itself relies on a simple *mapping by convention strategy* which is used by Fit. This means, that, for example, the names of the column headers of the Fit test data tables correspond with the names of the parameters for the called method under test. The source information of the MUT is processed to extract all relevant information for automated test generation. Figures 2 and 3 illustrate the mapping of source code information with the generated test code and test data.

Listing 1 shows a very simple example of a method *divide* which takes two arguments and simply returns the quotient of *n* and *d*. In the lower part of Listing 1 the generated test code is stated which in this case extends the *ColumnFixture* fixture type. The test method has the same name as the MUT to enable a simple identification strategy for refactorings (see section Refactorings). Moreover, the declared field names are related to the class names and parameter names of the MUT.

Refactorings

While in general most attention is paid to the initial development of a software product, this phase makes only minor part of the overall cost of a software product. Actually, most effort on a software product is spent on the maintenance of the product through its whole product life cycle [Fea05]. Thus, any reduction in maintenance effort can bring a significant gain to reduce the overall development cost of a system. The iTDD tool addresses this issue by adding refactoring functionality also for Fit tests, i.e. for code as well as for test fixtures. The iTDD tool includes fully automated test code maintenance, making test code completely transparent for the developer. The developer and user can focus on changes that really matter to the system: improving the production code and improving test cases and test data. The test code is completely hidden and is adapted automatically through appropriate refactorings. The filled block arrows in Figure 4 indicate direct user interactions with the system, driven by external influences (e.g. requirements change). The white block arrows indicate adaptations which are carried along all impact files automatically by the tool. In software development changes may occur at various places:

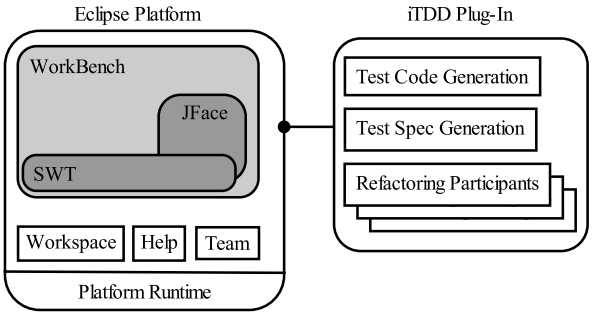

Figure 5: Architectural concept

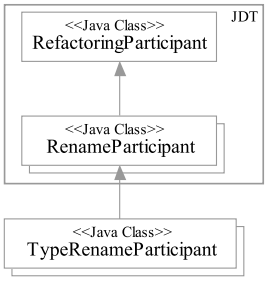

Figure 6: Extending Eclipse Refactoring

```
<extension point = ”org.eclipse.ltk.core.refactoring.rename Participants”>
<renameParticipant class =
    ”com.luxoft.fitpro.plugin.refactoring.participants.TypeRenameParticipant”
    id = ”itdd.participants.compilationunitrename”
    name = ”Compilation Unit Rename Refactoring Participant”>
  <enablement>
    <with variable = ”affectedNatures”>
      <iterate operator = ”or”>
         <equals value = ”org.eclipse.jdt.core.javanature”></equals>
      </iterate>
    </with>

  <with variable = ”element”>
     <instanceof value = ”org.eclipse.jdt.core.IType”></instanceof>
  </with>
  </enablement>
</renameParticipant>
. . .
```

Listing 2: Refactoring Configuration

1. a) The developer may change the production code due to refined requirements or to improve the code;
2. b) The user may also refine the test specification due to changed requirements or to improve readability;
3. c) The test engineer may change the test code implementation.

While a and b are mainly externally driven, namely by changing requirements, c is purely internal and usually seen as rather dry and time consuming. *Changing code* Existing toolkits (e.g JavaTM-Development Toolkit) support a wide range of code refactorings, so that they make a good foundation, on which the iTDD tool can build on. As shown in Figure 4 code changes do not only affect the production code but also the related test code and test data. The iTDD tool extends existing refactorings with additional components which propagate changes to the affected test files, e.g. column headers, class names, etc. (see section Eclipse Integration). *Changing test specification* As mentioned before the user may also change the test specification. He can enter new or modify and delete existing test data, for example. Except for the test data repository itself, this does not affect any other files. However, changes in the test specification may also concern changing of labels of input and output fields (e.g. to increase readability). These changes also have to be handled in the appropriate test code and might even propagate to the production code to keep naming and interfaces synchronized. Input fields might even be deleted or new fields be added due to changed requirements. This would finally lead to changes in the method signature of the production code. iTDD addresses this through the appropriate test specification refactoring which propagates the changes automatically to the production code.

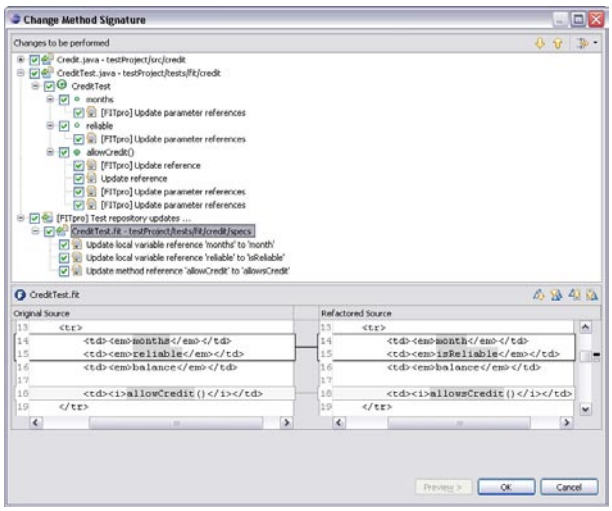

Figure 7: Refactoring UI Integration

Although the generation of test data was not in the focus of this work, the iTDD provides a simple random generator for test data. Moreover, the iTDD architecture provides a test data generator interface which allows easy integration of external test data generators. The interface allows specifying constraints for each individual parameter of a method, like max and min values, for example.

Eclipse Integration

In this chapter we will describe how we implemented the concept described above and how we integrated it into the Eclipse-IDE based on its plug-in concept and the Eclipse Abstract Syntax Tree (AST).

The iTDD tool has been implemented as an Eclipse plug-in [CR08] and consists of the two main modules for test code and test data generation, and various refactoring modules as depicted in Figure 5. The test code and data generation modules are actually integrated into one test generation module which appears as a new menu item in the Eclipse context menu. The refactoring modules are fully integrated into the Eclipse refactoring mechanism which is explained in the next paragraph.

Furthermore, to gather all relevant information for the automated test generation the Eclipse AST is used (e.g. to find class names, method calls, etc).

The Eclipse IDE already offers a wide variety of refactoring functionality in its Java Development Toolkit (JDT). The JDT also offers extension capabilities to add new refactorings for Java and for other file types and language types. The Eclipse refactoring architecture is based on the *Refactoring Participants* concept [Wid07]. This concept provides an extension mechanism, which is based on the idea to add new functionality as a new "participant" to any existing refactoring, like the Rename, Remove Parameter or Change Method Signature, for example, just as needed. To extend an existing refactoring with extra functionality, the new functionality must be implemented in a class derived from the corresponding base refactoring class as shown in Figure 6. As also indicated in Figure 6 multiple subclasses can be implemented for a given refactoring class to add a variety of new functionality to a given refactoring. This concept offers a very modular and flexible extension of existing Eclipse refactorings.

Additionally all new refactoring participants must be registered as extension points in the *plugin.xml* configuration as shown in Listing 2. So far we have implemented refactoring extensions for *Class Rename*, *Method Rename*, and *Change Method Signature* to include the refactoring of the corresponding Fit test specifications. The new refactoring extensions are fully integrated into the standard Eclipse refactoring process and the new functionality appears in the standard refactoring dialogs as shown in Figure 7.

Conclusion

The current implementation of iTDD includes the complete generation of test code and test specification for a given method signature. The automated generation is currently supporting functional tests (i.e. fit ColumnFixtures), which presumes a signature of the method under test to have input parameters and a return value. The generation process supports all basic data types as well as custom data types. For test data generation we currently use a simple random data generator for the basic data types. When generating tests for multiple methods of the same class the tool adopts the existing test code and test specification files accordingly. Refactoring operations for renaming classes, methods, and parameters are fully integrated by extending the refactoring tooling of the Eclipse IDE. Moreover, the change method signature and the modification of test specification operations are fully integrated. We are currently integrating the refactoring functionality into the open source tool FITpro [Lux] and will hopefully submit it to the community by the end of the year. Future work will focus on the extension of the iTDD tool and the operative application. Enhancements of the tool will include full support for custom data types which includes the generation of the appropriate test data. We will also implement test generation for overloaded methods and constructors. Strategies for test generation from input dialogs and workflows will be addressed in the future (see ActionFixtures and RowFixtures [MC05]). Further work could cover the development and integration of test data generation based on contract information such as given by Contract4J [ARA] to improve the quality of the test data

Acknowledgments

The authors give many thanks to the Hasler Foundation who funded this work as part of the ProMedServices project.

References

[ARA] Aspect Research Associates. Contract4J. http://www.contract4j.org (09.10.2008), 2003–2008.

[CR08] E. Clayberg and D. Rubel. Eclipse Plug-ins. Addison-Wesley Professional, Reading, Massachusetts, third edition, December 2008.

[Cun08] W. Cunningham. Framework for Integrated Test – Fit. http://fit.c2.com/ (10.10.2008), 2008.

[Fea05] M. C. Feathers. Working Effectively with legacy Code. Pearson Education, Inc., New Jersey, 2005.

[Lux] Luxoft UK Limited. FITpro – Acceptance Testing Solution. http://sourceforge.net/projects/fitpro (02.10.2008), 2008.

[MC05] R. Mugridge and W. Cunningham. Fit for Developing Software. Pearson Education, Inc., New Jersey, 2005.

[Pet08] D. Peterson. Concordion. http://www.concordion.org (01.10.2008), 2007–2008.

[PMS08] Hasler Foundation. ProMedServices – Proactive Software Service Improvement. http://web.fhnw.ch/technik/ projekte/promedservices (19.03.2009), 2008.

[SWE] IEEE Comnputer Society. SWEBOK. http://www.swebok.org/ (27.01.2007), 2007–2008.

[Wid07] T. Widmer. Unleashing the Power of Refactoring. IBM Rational Research Lab Zurich, February 2007.

ICT System und Service Management – eine Disziplin im Wandel

Gut verfügbare, zuverlässige und sichere ICT Infrastrukturen spielen in der Wirtschaft, der Verwaltung und im Privatleben eine immer wichtigere Rolle – jedoch sind die zugehörigen Aktivitäten für Aufbau, Betrieb und Pflege dieser Infrastrukturen gegenüber anderen Disziplinen der Informatik oft wenig sichtbar und prominent. Die Disziplin ICT System und Service Management bemüht sich in diesem Kontext neben angewandter Forschung bezüglich Komplexitätsreduktion, Steigerung der Kosteneffizienz und Verbesserung der Nachhaltigkeit des Ressourceneinsatzes auch um eine umfassende Abdeckung der entsprechenden Aktivitäten in der Aus- und Weiterbildung.

Hannes P. Lubich | hannes.lubich@fhnw.ch

Gemäss aktuellen Schätzungen von Marktanalysten der Gartner Group [Gartner] werden 2/3 aller Informatikmittel weltweit für betriebliche Aufgaben und betriebsgebundene ICT-Investitionen eingesetzt. Forrester Research [Forrester] schätzt, dass etwa die Hälfte aller in der Informatik Beschäftigten in betriebsrelevanten Funktionen tätig ist. Zudem werden ICT-Systemen in der weltweit vernetzten Wirtschaft immer höhere Werte anvertraut. An die Erhebung, den Transport, die Speicherung, die Bearbeitung, die Archivierung und die Entsorgung der entsprechenden Informationen sind seitens der Anwender, Betreiber und Regulatoren hohe Erwartungen bezüglich Zuverlässigkeit, Verbindlichkeit und Sicherheit der entsprechen ICT-Services geknüpft.

Gleichzeitig steht die ICT-Branche, seien es Hersteller, Betreiber oder Anwender von Hardware-, Software- und Service-Lösungen, unter enormem Kostendruck – insbesondere der im Betrieb traditionell hohe Fixkostenanteil steht unter ständiger, kritischer Beobachtung. Dies führt zu einem eigentlichen Dilemma im ICT System und Service Management – in wirtschaftlich positiven Zeiten sind genügende Mittel vorhanden, um die vorhandenen ICT-Umgebungen gemäss den Bedürfnissen der Benutzer rasch auszubauen und zu erweitern, jedoch fehlt oft die Zeit für die sorgfältige Konsolidierung und Harmonisierung der ICT-Infrastruktur. In Krisenzeiten wäre im Grundsatz genug Zeit für die Konsolidierung vorhanden, jedoch besteht dann meist wenig Bereitschaft für die nötigen Vorinvestitionen.

Dem nachhaltigen, zuverlässigen sowie den jeweiligen wirtschaftlichen und rechtlich / regulatorischen Anforderungen angemessenen Betrieb von ICT Systemen und Services kommt also eine grosse Bedeutung zu. Die erfolgreichen Anbieter entsprechender Services sind diejenigen, die agil genug sind, im Spannungsfeld zwischen Kundenanforderung, Dienstqualität, Kosten und Risiken schnell die richtigen Entscheide zu treffen und entsprechend umzusetzen.

Was ist ICT System und Service Management?

ICT System und Service Management (ICT-SSM) umfasst alle technischen, organisatorischen und prozeduralen Aktivitäten für die optimierte Bereitstellung und Verwaltung komplexer, verteilter und meist inhomogener IT-Infrastrukturen, und bildet damit ein wesentliches Element eines zuverlässigen, wirtschaftlichen und nachhaltigen IT Betriebes. In der Praxis finden sich verschiedenste Ausprägungen, je nach funktionalen Bedürfnissen und Optimierungszielen, rechtlich-/regulatorischen Vorgaben und einsetzbaren Mittel. Die Palette reicht dabei von der Diensterbringung als reines „cost center“ bis hin zur „profit and loss“ Verantwortung der ICT-Betriebsorganisation im Wettbewerb mit anderen internen oder externern Anbietern.

Die Kernelemente des ICT-SSM bilden einige wenige Grundsätze, die der geordneten, zweckmässigen und zuverlässigen Diensterbringung zugrunde liegen:

Zweckbindung

- *„fitness for purpose“:* Ist die bereitgestellte ICT-Umgebung grundsätzlich für den intendierten Zweck geeignet?
- *Inventarisierung und Lizenzierung:* Ist die bereitgestellte ICT-Umgebung in das bewirtschaftete IT-Inventar integriert und sind alle Fremdkomponenten korrekt lizenziert?
- *Aufwandszuordnung und Abrechenbarkeit:* Lässt sich der gesamte Aufwand bezüglich Betrieb, Pflege, Weiterentwicklung usw. den Verursachern zuordnen und sind die entsprechenden Kostenumlagen möglich und transparent?
- *Konfigurierbarkeit, Erweiterbarkeit, Wartbarkeit:* Passt die Lösung in die bestehende Landschaft, welche Anpassungen müssen gemacht

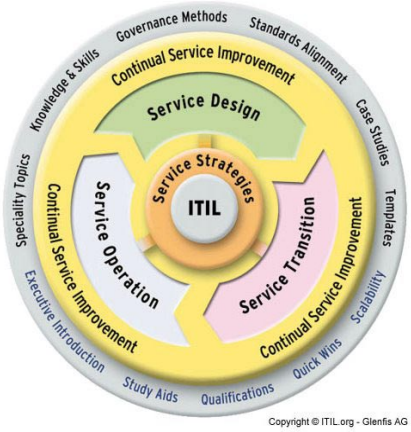

Abbildung 1: ITIL Referenzmodell. Quelle: www.itil.org

werden und können im betrieb die jeweils nötigen Erweiterungen und Wartungen vorgenommen werden?

Leistungsfähigkeit
- *„fitness for use":* Ist die bereitgestellte ICT-Umgebung leistungsfähig genug, um für den absehbaren Planungshorizont die Anforderungen der Benutzer zu erfüllen?
- *Performance, Kapazitätsplanung, Skalierbarkeit:* Ist zukünftiges Wachstum berücksichtigt und gibt es positive bzw. negative Effekte für andere ICT-Komponenten (Synergien, Netzauslastung usw.)?

Verfügbarkeit
- *Sicherheit, Wiederherstellbarkeit, Widerstandsfähigkeit:* Ist die bereitgestellte ICT-Umgebung in das Sicherheits- und Business Continuity-Konzept integriert und bestehen ausreichende Planungen für den Krisenfall?
- *Fehlererkennung und -behebung:* Wie schnell und zuverlässig können Fehlerzustände vorausgesagt, erkannt und durch Gegenmassnahmen in ihrem Schadenpotential auf ein akzeptables Mass reduziert werden?

Natürlich kann der Betrieb von ICT-Systemen und -Services (ICT-S&S) nicht auf alle diese Parameter optimiert werden. Es ist Aufgabe jeder ICT-Betreiberorganisation mit Auftraggebern, Aufsichtsorganen und Lieferanten den jeweils benötigten Service Level festzulegen und aufrecht zu erhalten.

Standardisierung

Schon seit geraumer Zeit sind Bestrebungen im Gang, den Betrieb von ICT-S&S auf eine einheitlich normierte, überprüfbare und vergleichbare Basis zu stellen. Nebst vielen herstellerspezifischen Quasi-Standards hat sich ITIL (IT Infrastructure Library), das inzwischen in der Version 3 weltweit normiert wurde, als Referenzmodell durchgesetzt (siehe Abb. 1 sowie [ITIL]).

ITIL gliedert das ICT-SSM in klar voneinander abgegrenzte Phasen, vom Entwurf der Service-Strategie über das Service Design und die Betriebsübergabe bis hin zum regulären Service Betrieb und der Weiterentwicklung, Pflege und Wartung der bereitgestellten Systeme und Services. Jede Phase enthält formal beschriebene Prozesse, Methoden und Hilfsmittel, die den gesamten Lebenszyklus der eingesetzten ICT-Komponenten und Services abdecken. Besondere Bedeutung kommt dabei dem ICT Asset Management, den Veränderungsprozessen sowie der Definition formell verbindlicher Service Level Agreements zu. Alle Phasen und Prozesse des ITIL-Modells werden durch Methoden für die kontinuierliche Messung und Verbesserung der angebotenen Services ergänzt.

Es bestehen also hinreichende Methoden und Hilfsmittel, um den Betrieb von ICT-S&S formell zu regeln, zu beschreiben und bezüglich Effizienz und Effektivität zu messen bzw. zu verbessern. Jedoch sind längst nicht alle ICT-Anwender- bzw. Anbieterorganisationen bereit und in der Lage, ihre komplexen, oft historisch gewachsenen ICT-Umgebungen entsprechend zu beschreiben und auf Basis dieser Beschreibung zu vereinheitlichen.

Schnittstellen zu anderen Disziplinen

Das ICT-SSM weist naturgemäss eine grosse Anzahl von Schnittstellen zu umgebenden Disziplinen auf. Im Bereich ICT sind dies insbesondere:
- Netzwerk Management
- Security Management und Business Continuity / Disaster Recovery Planung
- Liegenschafts-/Gebäudemanagement
- Technologiemanagement
- Qualitätsmanagement

Auch zu geschäftlichen Disziplinen hat das ICT-SSM eine enge Verbindung. Hier sind insbesondere das Information Management, das Business Process Engineering und das Requirements Engineering wichtige Beitraggeber, aber auch Abnehmer der entsprechenden Aktivitäten. ICT-SSM ist also im Grundsatz interdisziplinär und „lebt" von der guten technischen, organisatorischen und prozeduralen Vernetzung mit den umgebenden ICT- und Business-Strukturen. Die Arbeit in diesem Gebiet erfordert dementsprechend vernetztes Denken und Handeln.

Aktueller Stand und zukünftige Herausforderungen

Das ICT-SSM hat inzwischen einen durchaus akzeptablen Stand erreicht. ICT-Systeme können gut dokumentiert werden, die technischen System-Management-Schnittstellen sind bekannt und ITIL bietet eine gute Grundlage für die Normierung, Messung und Verbesserung. Kommerzielle wie auch frei verfügbare Software-Werkzeuge für das ICT System Management sind in ausreichender Menge und Qualität etabliert.

Jedoch verändert und entwickelt sich die ICT-Landschaft auch weiterhin rapide und stellt damit das ICT-SSM vor ständig neue Herausforderungen. Signifikante Treiber für die ständige Anpassung und Erweiterung der Methoden und Hilfsmittel sind insbesondere:

- ein ständig wachsendes Mass an Komplexität und Inhomogenität der ICT (Betriebssysteme, Endgeräte, Schnittstellen, Middleware-Komponenten, „gebündelte" Applikationen, proprietäre Benutzer- und Management-Schnittstellen),
- die drahtlose Vernetzung immer weiterer ICT-Komponenten sowie
- der Trend zu neuen, flexibleren Finanzierungsmodellen und Kostenbegrenzungen.

Es sind also von verschiedenster Seite neue Anforderungen bezüglich ICT-SSM zu erfüllen:

Anwender
- Transparenz, Planbarkeit und Kostenwirksamkeit der Anforderungen
- Bereitschaft zur Integration in ein vereinheitlichtes ICT Service Management
- Effizienter Einsatz, insbesondere bezüglich Umwelteffizienz („Green IT")

Hersteller
- Harmonisierung von Schnittstellen, Integrierbarkeit der angebotenen Lösungen
- Virtualisierung und „on demand" Verfügbarkeit der angebotenen Lösungen
- Abbildung von technischer und geschäftlicher Management-Funktionalität

Aus- und Weiterbildung
- Starke Vernetzung mit anderen System und Service Management Disziplinen
- Vertieftes Verständnis für nicht-technische Elemente der Diensterbringung

ICT-S&S Manager / Betreiber
- Transparenz bezüglich Aufwand, Nutzen und Risiken
- Nachhaltigkeit der getätigten ICT-Investitionen
- Reduktion des „vertikalen Denkens" in technologischen Silos

Vertieftes Wissen über die geschäftlichen Kernprozesse, die von den jeweiligen ICT-Umgebungen unterstützt werden.

Schlussfolgerungen

Das ICT-SSM bildet die Schnittstelle zwischen der Service-Definition, Spezifikation und Erstellung einerseits und der betrieblich effizienten und effektiven Nutzung dieser Dienste andererseits. Wie eingangs erwähnt werden hierfür bis zu 2/3 aller Informatikmittel weltweit eingesetzt. Es lassen sich demnach folgende Schlussfolgerungen ziehen:

- ICT-SSM ist ein Kernelement der ICT-Diensterbringung;
- ICT-SSM ist Teil eines hoch integrierten IT Service Managements;
- Ein wesentliches Ziel des ICT-SSM ist die kosteneffiziente, zuverlässige und nachhaltige Erbringung der entsprechenden Dienstleistungen;
- In komplexen ICT-Umgebungen sind betriebliche Aufgaben ebenfalls komplex – System und Service Management Spezialisten sind entsprechend gefordert;
- Das Berufsbild erfordert weitreichende technische, organisatorisch/planerische und kommerzielle Fertigkeiten und Fähigkeiten.

Vor diesem Hintergrund bildet das ICT-SSM einen wichtigen Teil des Lehr- und Forschungsangebots der FHNW. In der Ausbildung werden entsprechende Themen mit einer dedizierten Vertiefungsrichtung im Bachelor- und Master-Studiengang der Informatik angeboten. Zudem besteht neu ein entsprechendes Weiterbildungsangebot (Diploma of Advanced Studies (DAS) in IT Systems and Operations Management) für Interessenten aus der Wirtschaft und Industrie.

Im Bereich der angewandten Forschung und Entwicklung wird unsere Forschungsgruppe selektiv Themen des ICT-SSM in enger Zusammenarbeit mit der Industrie bearbeiten. Themen der angewandten Forschung und Entwicklung werden im Bereich Architektur und Design (z.B. Peer-to-Peer und Cloud-Achitekturen, Modularisierung, Selbstheilung, Sicherheit / Business Continuity), der Entwicklung von Anwendungen für ICT-Dienste und von Werkzeugen für den Betrieb der ICT-Infrastruktur, der praktischen Erprobung im Labor sowie im Bereich der Service-Orientierung (Software als Service, Service-Cockpits, sowie Definition, Verifikation und Benchmarking von Service Agreements für ICT-Infrastrukturen) angesiedelt sein. Darüber hinaus werden entsprechende Beratungsleistungen (IT Process Engineering, funktionale Anforderungsanalyse, Reifegradbeurteilung, Nachhaltigkeit und Umwelteffizienz) angeboten.

ICT-SSM ist demnach ein Gebiet mit hohem Erwartungs- und Kostendruck, aber auch mit Relevanz und Zukunft – auch und besonders in wirtschaftlich herausfordernden Zeiten.

Referenzen

[Forrester] Forrester Research: http://www.forrester.com

[Gartner] Gartner Group: http://www.gartner.com

[ITIL] IT Infrastructure Libary: http://www.itil.org

Innovative Bildaufbereitung in Hochgeschwindigkeitskameras

Digitale Bilder sind aus dem Alltag nicht mehr wegzudenken. Die meisten Hersteller von digitalen Kameras befinden sich in Asien. Im Nischensegment der Hochgeschwindigkeitskameras gehört jedoch das in Baden-Dättwil ansässige Unternehmen AOS Technologies AG zu den führenden Anbietern. Mit AOS zusammen haben wir neue, effiziente Algorithmen und entsprechende Bildbearbeitungssoftware entwickelt, um bei Bildraten von dreissig Bildern pro Sekunde hochqualitative RGB-Bilder ohne störende Farbmoiréeffekte direkt in der Kamera zu erzeugen.

Martin Schindler, Christoph Stamm | christoph.stamm@fhnw.ch

Industrielle Bildverarbeitung kommt heute in sehr vielen Industriezweigen zum Einsatz. Auch in Bereichen, wo digitale Kameras nicht gleich auf Anhieb erwartet würden, erfüllen sie unerlässliche Beobachtungs-, Überwachungs- und Analyseaufgaben. Hochentwickelte Hochgeschwindigkeitskameras werden längst nicht mehr nur zur Gewinnung von Zeitlupenaufnahmen bei Crash-Tests eingesetzt, sondern immer mehr auch zur Feinkalibrierung und Optimierung von vollautomatischen Produktionsstrassen. Mit bis zu zehntausend Bildern pro Sekunde können entscheidende Details von schnellen Abläufen erfasst werden, welche für die Effizienz und das Gelingen ausschlaggebend sein können.

Im vorliegenden Bericht beschreiben wir ein gemeinsames Projekt[1] mit dem in Baden-Dättwil ansässigen Unternehmen AOS Technologies. Gemeinsam entwickelten wir ein neues, innovatives Bildaufbereitungsverfahren für Hochgeschwindigkeitskameras. Dieses Verfahren ermöglicht ohne nachgeschalteter Bildaufbereitung in einem separaten Rechner die Erzeugung von klaren, qualitativ hochstehenden Bildern direkt in der Kamera. Die Bildaufbereitung umfasst das Entfernen von Rauschen (Fixed-Pattern- und Banding-Noise), die Generierung eines RGB-Bildes durch Bayer-Pattern-Interpolation, Helligkeits- und Kontrastanpassung und Weissabgleich. Um eine solche Bildaufbereitung in Echtzeit bei mindestens 30 Bildern pro Sekunde zu ermöglichen, sind sehr effiziente und innovative Algorithmen notwendig.

Der zeitaufwendigste Teil dieser Bildaufbereitung ist die Bayer-Pattern-Interpolation, bei der die roten, grünen und blauen Anteile eines RGB-Bildpunktes aus den örtlich separierten RGB-Anteilen der Bayer-Pattern interpoliert werden. Dabei kann es je nach Verfahren vorkommen, dass für die roten und blauen Anteile unterschiedliche Interpolationsrichtungen verwendet werden, was vor allem in fein texturierten Bildbereichen zu optisch störenden Farbmoiréeffekten führen kann.

1 Mit freundlicher finanzieller Unterstützung des Forschungsfonds des Kantons Aargau.

Aufbau digitaler Kameras

Das Kernstück einer digitalen Kamera ist der Bildsensor. Ein Bildsensor ist eine regelmässige Anordnung von mehreren Millionen einzelner Fotodioden (Lichtsensoren). Dabei bestimmt die Anzahl der Fotodioden die maximale Bildauflösung. In den verschiedenen auf dem Markt erhältlichen Digitalkameras kommen üblicherweise zwei verschiedene Arten von Bildsensoren zum Einsatz: In Kompaktkameras werden fast ausschliesslich CCD-Sensoren verbaut, während in Profikameras und Camcordern mehrheitlich CMOS-Sensoren anzutreffen sind. Die Vorteile der CMOS-Sensoren sind vor allem der geringere Stromverbrauch, die höhere Verarbeitungsgeschwindigkeit, sowie die Möglichkeit nur bestimmte Regionen auszulesen. Der Stromverbrauch beträgt etwa 1/10 eines CCD-Sensors. Bei CMOS-Sensoren besitzt zudem jede einzelne Fotodiode einen Verstärker, was das direkte Auslesen einzelner Bildpunkte (Pixel) grundsätzlich ermöglicht [Sen].

Eine Fotodiode wie auch ein gesamter Bildsensor kann nur Helligkeitsinformationen (Graustufenbilder) verarbeiten. Um zu den Farbinformationen zu gelangen werden den Sensoren auf verschiedene Arten Farbfilter vorgeschaltet. Die am häufigsten verwendete Art ist die nach ihrem Erfinder benannte Bayer-Pattern [Bay]. Dieses schachbrettähnliche Muster besteht zu je 25% aus Rot und Blau und zu 50% aus Grün (siehe Abb. 4). Grün kommt deshalb doppelt so oft vor, weil das menschliche Auge auf Grün empfindlicher reagiert als auf andere Farben. Dieser Farbfilter hat aber zur Folge, dass für jeden Pixel jeweils nur eine von drei notwendigen Farbinformation bekannt ist. Die beiden fehlenden Farbinformationen müssen aus den angrenzenden interpoliert werden.

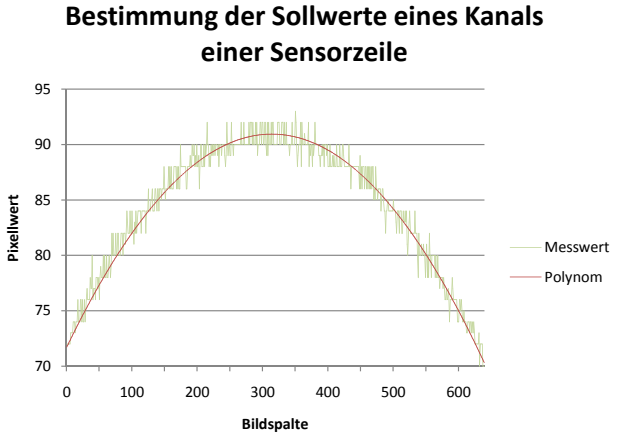

Abbildung 2: Bestimmung der Sollwerte eines Farbkanals einer Sensorzelle

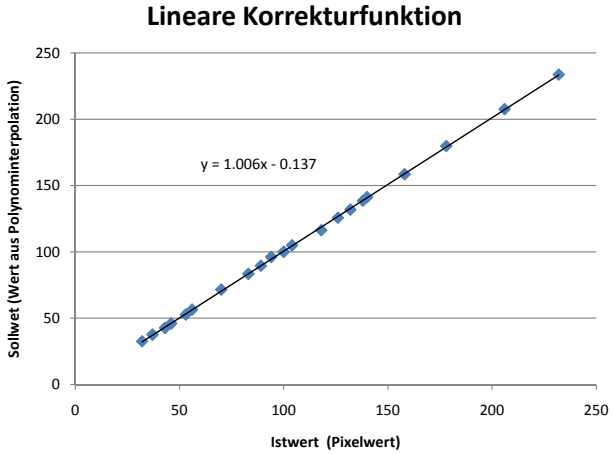

Abbildung 3: Beispiel einer linearen Korrekturfunktion pro Farbkanal und Pixel

Ein zweiter Ansatz stammt von der Firma Foveon bzw. Sigma. In deren Ansatz [Fov] wird der Effekt ausgenutzt, dass Photonen unterschiedlicher Wellenlängen (Farbe) unterschiedlich tief ins Silizium eindringen. Langwelliges rotes Licht dringt tiefer ein als kurzwelliges blaues Licht. Daher werden pro Pixel drei Lichtsensoren übereinander angeordnet, womit alle drei Farbinformationen parallel erfasst und auch ausgelesen werden können. Eine dritte Möglichkeit, welche vor allem in teuren Kameras eingesetzt wird, ist die Aufteilung des einfallenden Licht in dessen drei Grundfarben mittels eines Strahlteilerprismenblocks. Durch geschickte und kompakte Anordnung von teildurchlässigen und teilreflektierenden Prismen lassen sich drei separate Bildsensoren (je einen für Rot, Grün und Blau) verwenden. Dadurch sind für jeden Pixel alle drei Farbinformationen separat abrufbar.

Je nach Aufbau des Bildsensors kommt es beim Auslesen der Helligkeits- bzw. Farbinformationen zu kleinen Ungenauigkeiten. Diese Ungenauigkeiten werden vom Betrachter als Rauschen wahrgenommen. CMOS-Sensoren rauschen üblicherweise stärker als CCD-Sensoren. Der Grund dafür ist im Aufbau des CMOS-Sensors zu finden. Das Licht wird von Fotodioden eingefangen, verstärkt und mit AD-Wandlern in ein digitales Signal gewandelt. Dabei entstehen, bedingt durch die Toleranzen der einzelnen Komponenten, kleine Differenzen zwischen den Pixeln. Da der CMOS-Sensor spaltenweise ausgelesen, verstärkt und quantifiziert wird, entsteht ein streifenartiges Muster, das sogenannte Banding-Noise (siehe Abb. 1 auf der Umschlagrückseite). Als Fixed-Pattern-Noise wird das pro Pixel individuelle Rauschen bezeichnet, welches durch die Toleranzen der einzelnen Pixelverstärker verursacht wird [Rau]. Sowohl bei Banding- als auch bei Fixed-Pattern-Noise handelt es sich um fast konstante Muster, welche weniger vom Bildinhalt als von der Betriebstemperatur des Sensors abhängen. Beide Arten von Rauschen sind auf allen Digitalbildern mehr oder weniger stark zu sehen.

Reduktion des Rauschens

Im vorliegenden Projekt arbeiten wir mit einem CMOS-Farbsensor von Micron mit einer Auflösung von 1.3 Megapixel [MT9M413]. Dieser Bildsensor erzeugt Farbbilder mithilfe einer Bayer-Pattern-Interpolation und ermöglicht Hochgeschwindigkeitsaufnahmen bei voller Sensorauflösung mit 500 Bildern pro Sekunde. Mit einer Halbierung der Bildauflösung lässt sich jeweils die Bildrate verdoppeln. Wie oben bereits erwähnt, ist die Bildqualität von CMOS-Farbsensoren infolge verschiedener Arten des Rauschens vermindert. Mit entsprechenden Massnahmen sollen vor allem die konstanten Muster (Fixed-Pattern- und Banding-Noise) möglichst gut eliminiert werden. Eine typische Eigenschaft der Farbsensoren mit Bayer-Pattern sind die beiden unterschiedlichen Helligkeiten der grünen Pixel auf den geraden und ungeraden Zeilen. Diese müssen auf eine gemeinsame Helligkeit korrigiert werden, da sonst ein horizontales Streifenmuster auf dem Grünkanal entstehen würde.

Der von AOS eigens entwickelte Algorithmus berechnet für jeden Pixel anhand von unterschiedlich belichteten Helligkeitsbildern (Referenzbildern) eine Korrekturfunktion, die den einzelnen Pixel auf den Mittelwert des Gesamtbildes korrigiert. Die Helligkeitsbilder sollten dabei möglichst homogen ausgeleuchtet sein und das gesamte Farbspektrum abdecken, also möglichst gleichmässig grau sein. Eine solche gleichmässige Ausleuchtung lässt sich in der Praxis allerdings kaum mit vernünftigem Aufwand erzielen. Oft sind die Bilder im Zentrum heller und werden gegen den Rand hin dunkler.

Der von uns neu entwickelte Korrekturalgorithmus geht von n inhomogen ausgeleuchteten Referenzbildern aus. Pro Zeile und Farbkanal (Rot, Grün, Blau) der n Referenzbilder werden die

vorhandenen Pixelintensitäten durch ein Polynom interpoliert. Die Werte dieser Polynominterpolation bilden dann die Sollwerte zur Berechnung der Korrekturfunktionen (Abb. 2). Pro Kanal und Pixel erhält man so *n* Mess- und Sollwerte. Daraus wird pro Kanal und Pixel eine lineare Korrekturfunktion interpoliert (Abb. 3). Die zwei Koeffizienten der linearen Korrekturfunktion werden pro Farbkanal und Pixel gespeichert und bei jeder Aufnahme mit dem Sensor abgerufen und entsprechend angewandt. Da eine solche Korrekturfunktion individuell auf eine Digitalkamera samt ihrem zugehörigen Bildsensor zugeschnitten ist, muss sie für jede Kamera separat ermittelt werden.

Bayer-Pattern-Interpolation

Die digitalisierten Rohdaten eines Farbsensors mit Bayer-Pattern umfassen jeweils nur eine von drei notwendigen Farbinformationen pro Pixel. Die beiden fehlenden Farbinformationen für ein vollständiges RGB-Bild müssen aus den vorhandenen Werten mit Hilfe eines Algorithmus interpoliert werden. Auf den ersten Blick scheint dies einfach zu sein. So könnte man zum Beispiel die beiden (bei Rot und Blau), resp. vier (bei Grün) bekannten gleichfarbigen Nachbarn nehmen und davon den Mittelwert berechnen. Dieser Ansatz kann allerdings bei starken Kanten und regelmässigen Texturen zu störenden Farbfehlern, sogenannten Farbmoirés, führen (siehe Abb. 4 auf der hinteren Umschlaginnenseite). Zu Moirés kommt es im Allgemeinen, wenn zwei regelmässige Raster in unterschiedlichem Winkel übereinander gelegt werden. Die Frequenz des Moirés von zwei Rastern mit gleicher Gitterweite ist proportional zum Sinus des Zwischenwinkels. Das heisst, vor allem bei spitzem Zwischenwinkel entstehen tieffrequente, stark störende Moiréeffekte. Bei digitalen Aufnahmen, die zu Moiréeffekten führen, ist der Bildsensor das eine Raster und das regelmässige Muster, welches fotografiert wird, das zweite Raster.

Eine erste wesentliche Qualitätsverbesserung gegenüber dem oben erwähnten einfachen Interpolationsalgorithmus wird erreicht, indem nicht nur die gleichfarbigen Pixel zur Interpolation eines fehlenden Pixels herbeigezogen werden, sondern auch die direkten Nachbarn, welche nicht von derselben Farbe sind. Eine zweite Qualitätsverbesserung kann durch die Bestimmung einer dominanten Interpolationsrichtung erreicht werden. Die dominante Interpolationsrichtung wird dann für alle Farbkanäle verwendet und führt zu einer besseren Erhaltung von Kanten. Um jeweils die dominante Interpolationsrichtung ausfindig machen zu können, muss pro Pixel eine lokale Bildregion analysiert werden. Im aktuellen „state-of-the-art"-Algorithmus was die Bildqualität betrifft, wird mit sehr grossem Aufwand die dominante Interpolationsrichtung ausfindig gemacht [CC06]. Dazu werden die Varianzen der Farbdifferenzen über eine Nachbarschaft von neun Pixeln berechnet.

All diese zusätzlichen Qualitätsverbesserungen erfordern zusätzlichen Rechenaufwand und reduzieren somit die Anzahl RGB-Bilder, welche pro Sekunde generiert werden können.

DaVinci Digital Media Processor

In unserem Projekt arbeiten wir mit einem DaVinci Digital Media Processor TMS320DM6437 von Texas Instruments [DMP]. Dieser Prozessor bietet grundsätzlich die Möglichkeit, eine Bayerkonvertierung in einem speziellen Hardwareblock, der sogenannten Preview-Engine, effizient zu berechnen. Die Preview-Engine selber ist eine mehrstufige, parametrisierbare Bildverarbeitungskette innerhalb des Video Processing Front End, welche z.B. Weissabgleich, Gammakorrektur, Helligkeits- und Kontrastanpassung, RGB-YC_bC_r-Konvertierung umfasst. Als Eingabedaten werden Rohdaten im Bayer-Format verarbeitet, die entweder direkt aus einem Bildsensor oder aus dem Speicher stammen.

Die übliche Hauptaufgabe der Preview-Engine ist die Bayer-Pattern-Interpolation zur Erzeugung eines RGB- resp. YUV-Bildes. Der Interpolationsalgorithmus ist jedoch fix und lässt sich nicht unseren individuellen Wünschen anpassen. Hingegen lässt er sich als Ganzes ausschalten und dadurch können die restlichen Funktionen der zuvor beschriebenen Bildverarbeitungskette zur Farbanpassung verwendet werden. Unsere Implementierung des Interpolationsalgorithmus ist somit unabhängig von der Preview-Engine, nutzt aber natürlich die Möglichkeiten des Prozessors, Daten parallel verarbeiten zu können. So kann der Prozessor beispielsweise innerhalb eines Taktzyklus vier 8-Bit Multiplikationen gleichzeitig berechnen. Vor allem ist aber der parallele Zugriff auf den Bildspeicher entscheidend für eine schnelle Verarbeitung der Daten. Deshalb ist es wichtig, dass pro CPU-Takt immer 8 Bytes gelesen oder geschrieben werden.

Eine erste, einfache Implementierung ohne Optimierungen des originalen Interpolationsalgorithmus von Chung und Chan führt auf dem Prozessor zu einer Bildrate von weniger als einem Bild pro Sekunde bei einer Bildauflösung von 800 mal 600 Pixel, womit die Interpolation mehr als einen Faktor dreissig zu langsam für unsere Anwendung ist. Werden im gleichen Algorithmus alle wesentlichen Teile durch auf Assemblerebene optimierte Funktionen ersetzt, so erhöht dies zwar wesentlich die Performanz, aber die geforderte Bildrate von 30 Bildern pro Sekunde wird mit ca. 4 Bildern pro Sekunde nicht annähernd erreicht. Deswegen muss ein Kompromiss zwischen Interpolationsqualität und Verarbeitungsgeschwindigkeit angestrebt werden. In unserem

Fall besteht der Kompromiss darin, dass wir vollständig auf die Bestimmung der dominanten Interpolationsrichtung verzichten.

Interpolationsalgorithmus

Zur Illustration der folgenden Interpolationen dient Abbildung 5 auf der hinteren Umschlaginnenseite.

- *Interpolation der grünen Pixel:* Da der Grünkanal bereits zur Hälfte vorhanden ist, werden zuerst die fehlenden grünen Pixel interpoliert. Die Interpolation unterscheidet zwei Fälle, je nachdem ob an der gesuchten Stelle bereits ein roter oder blauer Pixel bereits vorhanden ist (siehe Abb. 5a und 5b). Die nachfolgende Interpolation gilt für den Fall a mit vorhandenen roten Pixeln. Für den Fall b werden einfach die roten Pixel durch blaue ersetzt:

$$\widehat{G}_{x,y} = \left(\frac{G_{x-1,y} + G_{x+1,y}}{2}\right) + \left(\frac{2R_{x,y} - R_{x-2,y} - R_{x+2,y}}{4}\right)$$

- *Interpolation der roten und blauen Pixel:* Nachdem alle fehlenden grünen Pixel interpoliert worden sind, werden die roten und blauen Pixel interpoliert. Da sich die blauen Pixel nach demselben Verfahren wie die roten Pixel interpolieren lassen, beschreiben wir hier nur das Verfahren für die roten Pixel. Wir unterscheiden dabei drei Fälle (Abb. 5b, 5c, 5d). Im Fall b gibt es vier gleichfarbene Nachbarn:

$$\widehat{R}_{x,y} = \widehat{G}_{x,y} + \frac{\left(R_{x-1,y-1} - \widehat{G}_{x-1,y-1}\right) + \left(R_{x-1,y+1} - \widehat{G}_{x-1,y+1}\right)}{4} + \frac{\left(R_{x+1,y-1} - \widehat{G}_{x+1,y-1}\right) + \left(R_{x+1,y+1} - \widehat{G}_{x+1,y+1}\right)}{4}$$

  Im Fall c sind die beiden nächsten roten Pixel oberhalb und unterhalb des zu interpolierenden Pixels vorhanden. Zum grünen Pixel an der Stelle werden die Differenzen zwischen dem roten und dem grünen Nachbarpixel addiert:

$$\widehat{R}_{x,y} = G_{x,y} + \frac{R_{x,y-1} - \widehat{G}_{x,y-1} + R_{x,y+1} - \widehat{G}_{x,y+1}}{2}$$

  Der Fall d entspricht dem Fall c mit dem Unterschied, dass die beiden nächsten roten Pixel links und rechts des zu interpolierenden Pixels liegen:

$$\widehat{R}_{x,y} = G_{x,y} + \frac{R_{x-1,y} - \widehat{G}_{x-1,y} + R_{x+1,y} - \widehat{G}_{x+1,y}}{2}$$

Farbanpassungen

Nebst einem rausch- und artefaktfreien Bild ist die natürliche Farbwiedergabe ein wichtiges Merkmal eines „schönen" Bildes. Die natürliche Farbwiedergabe wird durch Kontrast, Helligkeit, Farbsättigung und Gamma beeinflusst. Veränderungen dieser vier Bildeigenschaften sollen direkt in der Kamera und in Echtzeit erfolgen.

Der Kontrast, die Helligkeit sowie das Gamma können wie bereits erwähnt von der Preview-Engine des DMPs auf Basis der Rohdaten verändert werden. Die Farbsättigung des RGB-Zielbildes kann hingegen erst nach der Bayer-Pattern-Interpolation verändert werden, da die Sättigung durch die Interpolation beeinflusst wird.

Die Farbsättigung S entspricht im Wesentlichen dem Verhältnis von der kleinsten Farbkomponente zum Durchschnitt über die drei Farbkomponenten: $S = 1 - \min(R_{in},G_{in},B_{in})/avg$, wobei $avg = (R_{in} + G_{in} + B_{in})/3$ und $R_{in},G_{in},B_{in} \in [0,1]$. Bei einem reinen Grauton sind $R_{in} = G_{in} = B_{in}$ und somit ist das Minimum gleich dem Durchschnitt, was zu einer minimalen Sättigung von 0 führt. Primär- und Sekundärfarben haben mindestens eine Farbkomponente gleich 0 und die restlichen Farbkomponenten gleich 1. Daher ist das Minimum gleich 0 und der Durchschnitt grösser als 0, was zu einer maximalen Sättigung von 1 führt. Zwischen einem Grauton und einer voll gesättigten Farbe kann dann mit dem Parameter $s \in [0,1]$ linear interpoliert werden:

$$\begin{pmatrix} R_{out} \\ G_{out} \\ B_{out} \end{pmatrix} = avg + s \begin{pmatrix} R_{in} - \text{avg} \\ G_{in} - \text{avg} \\ B_{in} - \text{avg} \end{pmatrix}$$

Resultate

Zur Reduktion des Sensorrauschens entwickelten wir ein einfach zu bedienendes Hilfsprogramm für den Kamerahersteller, welches aus einer Serie von Referenzbildern die Korrekturkoeffizienten für die jeweilige Kamera berechnet. Diese Korrekturkoeffizienten werden anschliessend an die Kamera übertragen und sorgen dafür, dass bei jeder Aufnahme das kameraspezifische Rauschen reduziert wird. Die dadurch erreichte Erhöhung der Bildqualität lässt sich mit technischen Differenzmassen nur ungenügend messen, weil hierbei unklar ist, was genau als Referenz gelten soll. Daher beruft man sich auf die optische Kontrolle durch geschulte Experten.

Ähnlich ist es mit der Feststellung der Reduktion des Moiréeffekts. Wiederum lässt sich mit den üblichen Qualitätsmassen für Bildvergleiche, wie z.B. Color-Peak-Signal-to-Noise-Ratio (CPSNR) [CC06, PSNR], die Reduktion nicht adäquat erfassen und es wird vorwiegend auf eine optische Qualitätsmessung durch eine Fachperson zurückgegriffen. Bilder mit reduzierten Moiréeffekten können durchaus eine verminderte CPSNR gegenüber anderen mit Moiréeffekt haben. Das heisst, die Reduktion des Moiréeffekts führt nicht automatisch zu einer Erhöhung der CPSNR. Dies ist auch mit ein Grund dafür, warum die CPSNR zur Beurteilung der Bildqualität von verlustbehafteten Bildverarbeitungsmethoden umstritten ist.

Mit der Reduktion der Moiréeffekte durch unseren schnellen Bayer-Pattern-Interpolationsalgorithmus kann aber zweifelsfrei eine markante

Verbesserung in der optischen Bildqualität erreicht werden (vergleiche Abb. 6 auf der hinteren Umschlaginnenseite). Für den Fall, dass für eine nachgeschaltete Bildverarbeitung auf einem Rechner genügend Zeit vorhanden ist, so können auch die zeitaufwendigeren Varianten unseres Interpolationsalgorithmus eingesetzt werden, die eine noch stärkere Reduktion des Moiréeffekts gestatten.

Die zur Bayer-Pattern-Interpolation vor- und nachgeschalteten Farbanpassungen (Kontrast, Helligkeit, Gamma usw.) in der Kamera, können durch eine Software auf einem mit der Kamera verbundenen Rechner gesteuert werden. Dabei handelt es sich jedoch nicht um eine externe Bildnachbearbeitung, sondern lediglich um eine Parametrisierung der Kamera und ein Monitoring der resultierenden Bildqualität.

Zusammengefasst lässt sich sagen, dass die von uns neu entwickelten Algorithmen und Software einerseits sehr effizient sind und anderseits den optischen Ansprüchen der Fachleute von AOS entsprechen. Das Projekt ist eine Erfolgsgeschichte für alle Beteiligten. Daher hat sich AOS dazu entschieden, diese Algorithmen in zukünftigen Kameras einzubauen.

Referenzen

[Bay] Bayer-Sensor: http://de.wikipedia.org/wiki/Bayer-Sensor

[CC06] Color Demosaicing Using Variance of Color Differences, King-Hong Chung and Yuk-Hee Chan, IEEE, Oktober 2006. http://www.eie.polyu.edu.hk/~enyhchan/J-TIP-CDemosaicking_using_VarCD.pdf

[DMP] DaVinci Digital Media Processor TMS320DM6437, Texas Instruments: http://focus.ti.com/docs/prod/folders/print/tms320dm6437.html

[Fov] Foveon X3-Technologie: http://www.foveon.com/article.php?a=67

[MT9M413] Micron 1.3 Megapixel CMOS active-pixel digital image sensor (MT9M413): http://pdf1.alldatasheet.com/datasheet-pdf/view/97416/MICRON/MT9M413C36STC.html

[Moi] Moiréeffekt: http://de.wikipedia.org/wiki/Moir%C3%A9-Effekt

[PSNR] Peak signal-to-noise ratio, Wikipedia: http://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

[Rau] Was sind die Ursachen von Bildrauschen: http://www.fotos.docoer-dig.de/Bildrauschen.htm

[Sen] Allgemeine Informationen zu Bildsensoren: http://www.fotos.docoer-dig.de/Bildsensoren.htm

Abbildung 4: Entstehung eines Farbmoirés durch lineare Interpolation (Durchschnitt der 2 resp. 4 gleichfarbigen Nachbarpixel)

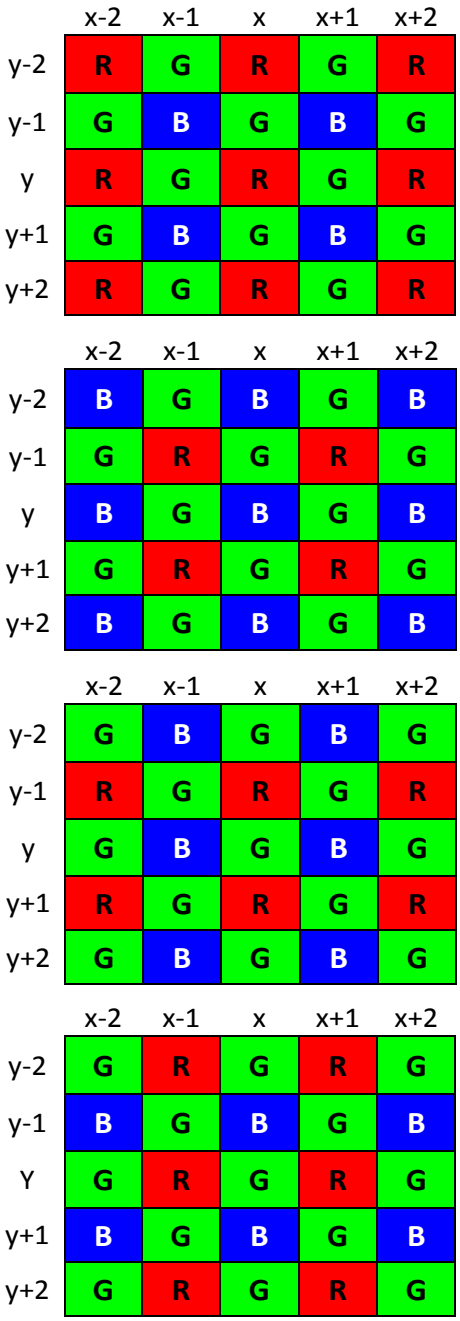

Abbildung 5: Vier unterschiedliche Interpolationsfälle a, b, c und d (von oben nach unten)

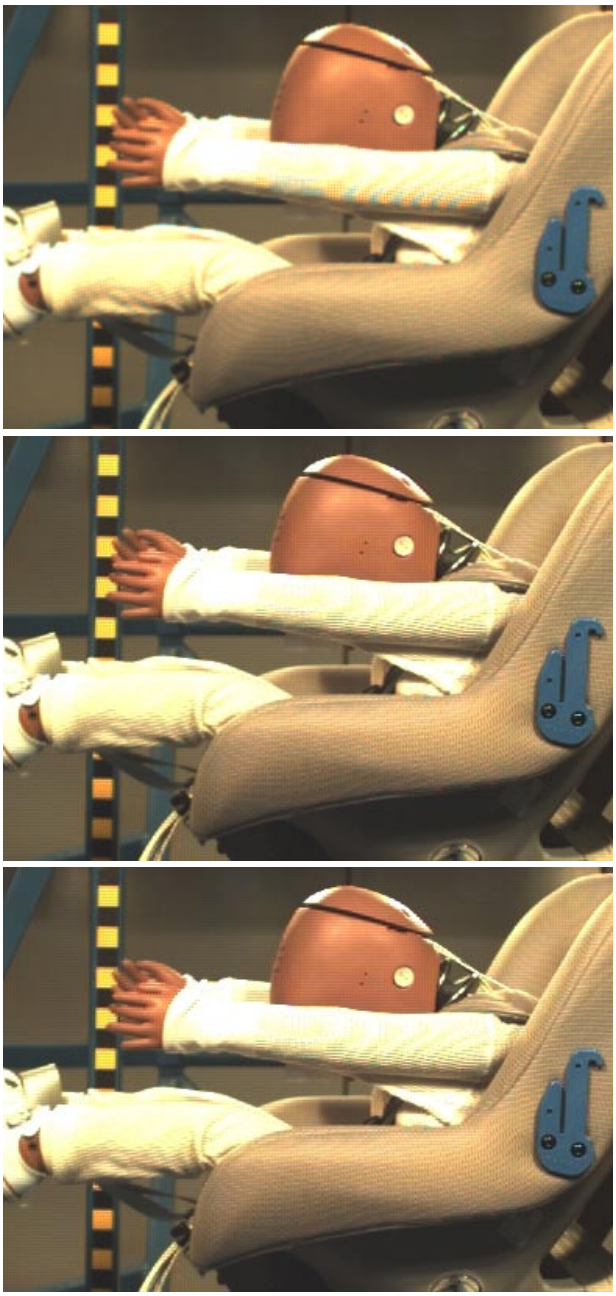

Abbildung 6: Ergebnisse drei verschiedener Bayer-Pattern-Interpolationen: ursprünglich verwendete Interpolation (oben), von uns entwickelte schnelle Interpolation direkt in der Kamera (Mitte), zeitaufwendige Post-Processing-Variante (unten)

Abbildung 1 auf Umschlagrückseite: Banding-Noise in Form von vertikalen Linien im Hintergrund gut ersichtlich

Fachhochschule Nordwestschweiz
Institut für Mobile und Verteilte Systeme
Steinackerstrasse 5
CH-5210 Brugg-Windisch

www.fhnw.ch/technik/imvs
Tel. +41 56 462 44 11