FH Technikum Wien

SEMINARARBEIT

Debugging in Android

ausgeführt von Manfred Fettinger, BSc
Ing. Florian Nikl, BSc
Jürgen Müllner, BSc

BegutachterIn: DI Stefan Schnidt

Wien, März 2010

FACHHOCHSCHULE
TECHNIKUM WIEN

Ausgeführt an der FH Technikum Wien

Studiengang MWI

Lehrveranstaltung Telekom&Mobile Computing

Inhaltsverzeichnis

1 ADB – Android Debug Bridge

Die ADB ist standardmäßig Teil der Tools des Android Software Development Kit (SDK). Aufzufinden ist es im Verzeichnis `/tools` des SDK. Es ist von der Kommandozeile aufzurufen. Einige Mögliche Kommandos:

Kommando	Bedeutung
`adb devices`	Liste aller Verfügbaren Geräte
`adb install <path_to_apk>`	Installiert eine Anwendung
`adb shell`	Ruft eine Shell auf

Tabelle 1: Einige ADB Kommandos

Eine Vollständige Auflistung aller Kommandos ist unter http://developer.android.com/intl/zh-CN/guide/developing/tools/adb.html zu finden.

1.1 Debugging mit ADB

ADB kann auch beim Debuggen hilfreich sein, hierfür stehen folgende Kommandos zur Verfügung.

Kommando	Bedeutung
`adb logcat`	Zeit logging Daten am Screen an
`adb bugreport`	Ausgabe von dumpsys, dumpstate und logcat Daten
`adb jdwp`	Liste von verfügbaren JDWP Prozessen eines Gerätes

Tabelle 2: ADB Debugging Kommandos

1.1.1 ADB Logcat

Nach der Eingabe von ADB Logcat kann mann mit etwa folgender Ausgabe rechnen (Ausschnitt):

```
V/HeadToHeadRacing( 8698): Waiting for draw loop to terminate ...
V/HeadToHeadRacing( 8698): Waiting for draw loop to terminate ...
I/HeadToHeadRacing( 8698): *** Draw Thread Stopped ***
D/HeadToHeadRacing( 8698): Saving success.
D/Sensors (  141): sensors=00000000, real=00000000
W/InputManagerService(  141): Starting input on non-focused client
com.android.internal.view.IInputMethodClient$Stub$Proxy@43ab1e70 (uid=10086 pid=8698)
D/AKMD    (  130): Compass CLOSE
I/WindowManager(  141): Setting rotation to 0, animFlags=0
I/WindowManager(  141): Config changed: { scale=1.0 imsi=0/0 loc=de_AT touch=3 keys=2/1/2
nav=3 orien=1 layout=18}
```

Die ersten beiden Einträge, welche durch ein ‚/' getrennt sind, beschreiben die Priorität und den Tag z.B. `V/HeadToHeadRacing`.

Die Priorität kann folgende Werte annehmen:

- `V` — Verbose (lowest priority)
- `D` — Debug
- `I` — Info
- `W` — Warning
- `E` — Error
- `F` — Fatal
- `S` — Silent (highest priority, on which nothing is ever printed)

Um jetzt gezielt Informationen zu bekommen, gibt es die Möglichkeit die Ausgabe zu filtern. Der Ausdruck um die Ausgabe zu filtern lautet `tag:priorität`. Wobei die Angabe der Priorität den minimum-Level bedeutet. Wird z.B. für die Priorität E gewählt, werden alle Einträge größer gleich E ausgegeben. Also E, F und S. Es ist auch möglich mehrere Tag-Priorität Kombinationen anzugeben.

- `adb logcat HeadToHeadRacing:E`
- `adb logcat HeadToHeadRacing:E *:S`

Das erste Beispiel würde alle Einträge des Tags HeadToHeadRacing ausgeben, bei denen die Priorität >= E ist. Beim zweiten Beispiel werden die gleichen Einträge von HeadToHeadRacing ausgegeben wie vorher und alle Einträge mit Priorität S aller anderen Tags. Somit ist es möglich sich auf die gewünschten Einträge zu konzentrieren.

Da die Bildschrimausgabe eventuell zu unübersichtlich ist, kann mit dem Umleitungszeichen ‚>' die Ausgabe in eine Datei umgeleitet werden. Um das Logging zu beenden ist die Tastaturkombination `<STRG+C>` zu verwenden. Eine Umleitung kann so aussehen `<adb logcat > log.txt>`.

Weiters ist es möglich einen Standard-Filter in einer Umgebungsvariable abzulegen. Der Befehl hierzu: `export ANDROID_LOG_TAGS="HeadToHeadRacing:E *:S"`

Es ist weiters noch möglich z.B. das Ausgabeformat zu ändern worauf hier nicht weiter eingegangen wird. Alle Informationen sind unter http://developer.android.com/intl/zh-CN/guide/developing/tools/adb.html zu finden.

Verwendung als Entwickler

Nun gut, es ist also möglich sämtliche Logging-Ausgaben diverser Prozesse einzusehen. Wie benutzte ich das jetzt als Entwickler für meine eigene Applikation? Das `Paket android.util` stellt hiefür die Klasse Log bereit. Die wichtigsten Methoden sind in folgender Tabelle dargestellt.

Methode	Bedeutung
`v(String tag, String msg)`	`Sendet Verbose Nachricht`
`d(String tag, String msg)`	`Sendet Debug Nachricht`
`e(String tag, String msg)`	`Sendet Error Nachricht`

Tabelle 3: Methoden der Klasse Log

Natürlich ist für jede Priorität eine solche Methode vorhanden. Der Aufruf gestaltet sich sehr einfach: `Log.v(„MeinTag“, „Hier steht die Nachricht“);`

Der folgende Logging-Ausschnitt zeigt folgenden logcat Aufruf: `adb logcat LoggingDemo:*`

```
I/ActivityManager(  141): Start proc com.fettinger.ld for activity
com.fettinger.ld/.LoggingDemo: pid=9323 uid=10089 gids={1015}
D/dalvikvm( 9302): LinearAlloc 0x0 used 647412 of 5242880 (12%)
V/LoggingDemo( 9323): index=0
I/ActivityManager(  141): Process com.android.vending (pid 8982) has died.
V/LoggingDemo( 9323): index=1
V/LoggingDemo( 9323): index=2
V/LoggingDemo( 9323): index=3
W/ActivityManager(  141): Launch timeout has expired, giving up wake lock!
W/ActivityManager(  141): Activity idle timeout for HistoryRecord{43c40f10
com.fettinger.ld/.LoggingDemo}
V/LoggingDemo( 9323): index=4
V/LoggingDemo( 9323): index=5
V/LoggingDemo( 9323): index=6
D/dalvikvm(  184): GC freed 8326 objects / 439096 bytes in 266ms
V/LoggingDemo( 9323): index=7
V/LoggingDemo( 9323): index=8
V/LoggingDemo( 9323): index=9
```

Zu Beginn erhält man noch allgemeine Informationen des Loggings, sobald die Applikation startet sieht man, dass vor allem `V/LoggingDemo` Einträge vorhanden sind. Die Virual Machine *DALVIK* und der ActivityManager senden jedoch auch weiterhin Informationen.

Diese Art des debuggings ist vor allem bei Applikationen die als Service im Hintergrund laufen sehr hilfreich. Der Einsatz des Debuggers von Entwicklungsumgebungen wie z.B. Eclipse sind da nicht zu Empfehlen.

Mit den Methoden des genannten Pakets ist es natürlich möglich über den Parameter msg jegliche Variablen die in einer Klasse verwendet werden auszugeben, wie im gezeigten Auszug z.B. der Variableninhalt von index.

1.1.2 ADB bugreport

Einen noch tieferen Einblick in das System bietet die Option `bugreport`. Mittels logcat bugreport werden einem sämtliche Systemparameter ausgegeben. Dies inkludiert dumpsys, dumpstate und logcat. All diese Informationen werden zusammen ausgegeben. Nachstehend ein kleiner Auszug:

```
============================================================
== dumpstate
============================================================
------ MEMORY INFO ------
MemTotal:          97924 kB
MemFree:            2984 kB
Buffers:             464 kB
Cached:            23236 kB
SwapCached:            0 kB
..
..
============================================================
== dumpsys
============================================================
Currently running services:
  SurfaceFlinger
  accessibility
```

```
  activity
  activity.broadcasts
  activity.providers
..
..
------------------------------------------------------------------------------
DUMP OF SERVICE activity.broadcasts:
Broadcasts in Current Activity Manager State:
  Registered Receivers:
  * ReceiverList{43affc20 141 system/1000 local:43afe4d0}
    app=ProcessRecord{43c90f78 141:system/1000} pid=141 uid=1000
    Filter #0: BroadcastFilter{43b7d910}
      Action: "android.intent.action.BOOT_COMPLETED"
```

Dieser Auszug enthält alle Informationen die für einen Bug-Report notwendig sind, er enthält wie schon erwähnt jede Einzelheit des Systems. Für das Debuggen einer Anwendung würde ich jedoch empfehlen es zuerst mit logcat zu Versuchen, da bugreport aufgrund der Vielzahl der Informationen etwas schwierig zu lesen ist.

1.1.3 ADB jdwp

Das Java Debug Write Protocol (jdwp) ist das Protokoll, welches zwischen der JavaVirtualMachine (JVM) und dem Debugger verwendet wird.

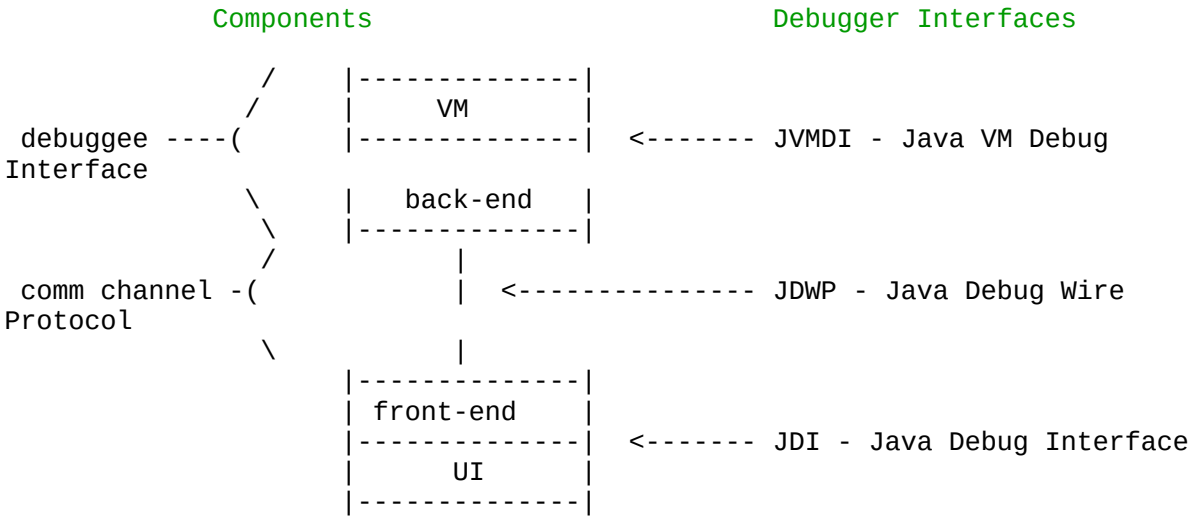

Abbildung 1: Debugger Architektur [1]

Detailliertere Informationen sind aus [1] zu entnehmen. Der Aufrufeparameter jdwp liefert eine Übersicht aller Prozesse (PIDs) welche einen JDWP Transport hosten. Das sind dann die Prozesse, welche gerade laufen und debuggt werden können. Jeder debugfähige Prozess wird in einer eigenen VM ausgeführt, die Ausgabe von jdwb liefert den Port der VM für jeden Prozess..

2 Dalvik Debug Monitor Service (DDMS)

DDMS ist wie adb Bestandteil des Android SDK und im Verzeichnis `/tools` zu finden. Ein kleiner Auszug von dem, was DDMS bietet:

- Port-Forwarding
- Screenshots des Mobiltelefons
- Thread und Heap informationen
- Logcat
- Prozessinformationen
- Funkinformationen
- Spoofing von eingehenden SMS und Anrufen
- Standort-Daten spoofing

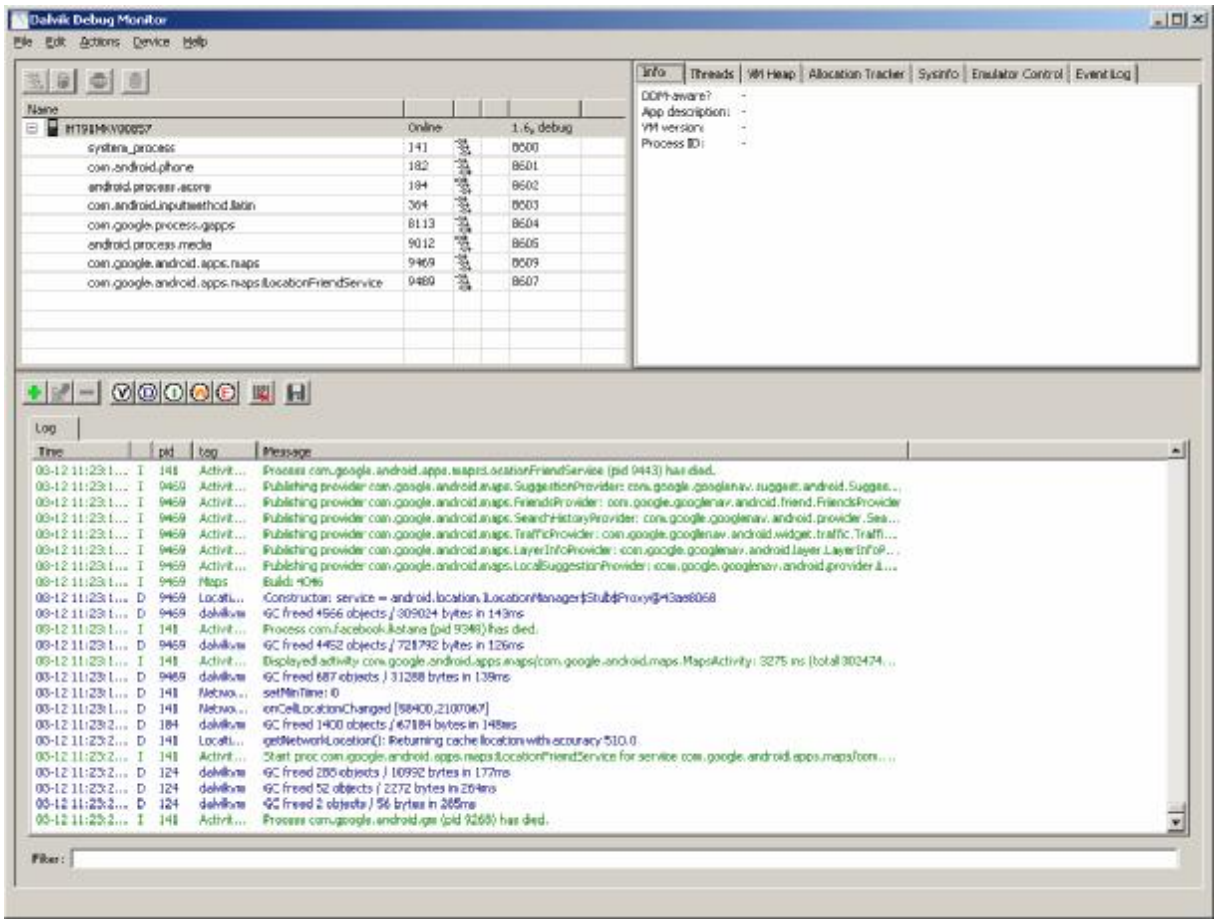

Abbildung 2: Hauptfenster des DDMS

Beim Start von DDMS verbindet sich dies mit adb und startet ein Geräte-Monitoring-Service zwischen diesen beiden Tools. Verbindet sich nun ein Mobilgerät, wird weiters ein VirtualMachine Monitoring Service gestartet

Abbildung 2 zeigt das Hauptfenster von DDMS. Freunde des GUI werden jetzt aufatmen, dass es neben adb ein Nicht-Kommandozeilen-Tool gibt.

Im Teilfenster links oben sieht man die aktuelle Prozessliste, welche man auch mittels adb jdwp bekommt. Also jene Prozesse, die gerade Laufen und debuggt werden können. Startet man eine Applikation am Mobiltelefon, kommt ein neuer Eintrag hinzu.

Im Teilfenster rechts daneben gibt es mehrere auswählbare Reiter:

- Info
  - Zeigt Informationen wie die VM-Version und die Prozess ID
- Threads
  - Zeigt die Threads des ausgewählten Prozesses an mit Informationen wie die Thread ID, den Status und Laufzeit
- VM Heap
  - Informationen über den Heap der VirtualMachine
- Allocation Tracker
  - Kann für jeden Prozess gestartet werden und zeigt an, welche Speicherbereiche vom Prozess angefordert wurden.
- Sysinfo
  - Allgemeine grafische Information über z.B. CPU load, Alarms, Wakelocks
- Emulator Control
  - Bei verwendung mit einem Emulator, können hier einige Einstellungen vorgenommen werden. Z.B. kann ein Anruf simuliert werden.
- Event Log
  - Ermöglicht das Anlegen eines Event Logs

Das untere Fenster entspricht dem Aufruf von `adb logcat`. Hier kann man in Echtzeit das Logging beobachten und mit Buttons die Filterung nach Prioritäten vornehmen. Weiters ist es möglich mehrere TAG-Priority Filter-Kombinationen zu definieren und abzuspeichern.

3 Traceview

Traceview wird ebenfalls mit dem SDK mitgeliefert. Es ist ein grafisches Tool um die Performance einer Applikation zu überwachen. Hierzu wird während der Laufzeit ein Tracefile erzeugt, dass mittels traceview dann grafisch aufbereitet wird. Die Klasse Debug stellt 2 Methoden zur Verfügung, mit denen der Trace gestartet und beendet wird. Folgendes Beispiel verdeutlicht den Gebrauch:

```
// start tracing to "/sdcard/calc.trace"
Debug.startMethodTracing("calc");
// ...
// stop tracing
Debug.stopMethodTracing();
```

Das dann erstellte File muss anschließend vom Gerät auf den Computer kopiert werden. Dies wird mit `adb pull` erreicht. Z.B.: `adb pull /sdcard/calc.trace /tmp`. Dies würde das File in den lokalen Ordner tmp kopieren. Jetzt kann traceview mittels `traceview calc.trace` aufgerufen werden.

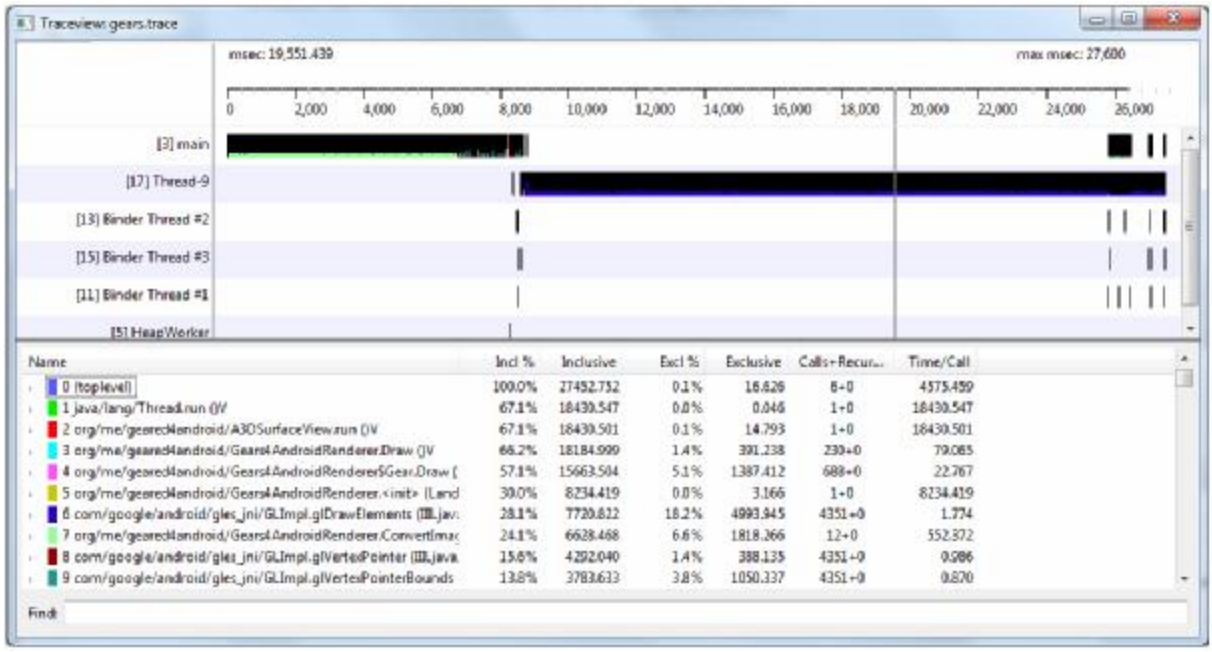

Abbildung 3: Traceview

Wie in Abbildung 3 ersichtlich wird, ist im oberen Bereich das Timeline Panel und im unteren Bereich das Profile Panel zu finden. Das Timeline Panel zeigt, wann jeder einzelne Thread gestartet und gestoppt wurde. Das Profile Panel zeigt eine Zusammenfassung von dem, was innerhalb einer Methode passiert ist.

4 Eclipse

Wählt man als Entwickler Eclipse als Entwicklungsumgebung, kann man dort das Debugging wie bei herkömlichen Java-Entwicklungen verwenden. Man setzt Breakpoints auf die gewünschten Zeilen im Code und führt das Programm unter „Debug as Android Application“ aus.

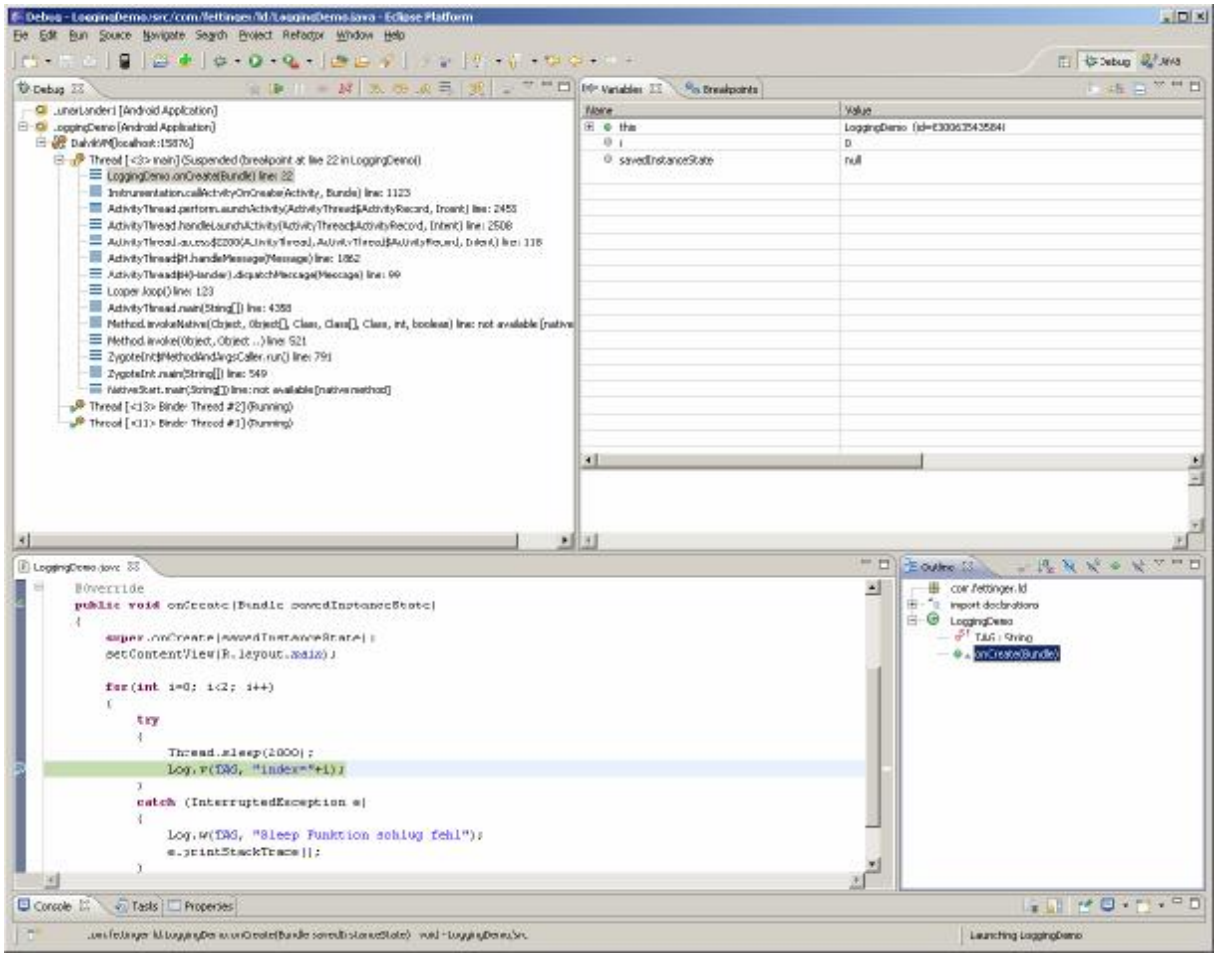

Abbildung 4: Screenshot Debugging in Eclipse

Während die Applikation auf dem Mobiltelefon oder Emulator ausgeführt wird, kann man ich Echtzeit durch den Code debuggen. Dabei kann man wie gewohnt den Aufrufstack und den Inhalt von Variablen ansehen.

DDMS in Eclipse

Es ist weiters auch möglich eine DDMS-Perspektive in Eclipse aufzurufen. Hierzu wählt man im Menü unter Window – Open Perspective – DDMS. Anschließend wird die DDMS Perspektive geöffnet. Von dort kann man fast alle Features wie im DDMS benutzen. Eine Liste mit den aktiven Prozessen steht zur Auswahl, aus denen man einen Prozess zum Debuggen auswählen kann. Das logcat wird am unteren Bildschirm ausgegeben. Auch hier ist es möglich Filter für diverse Prioritäten zu setzen.

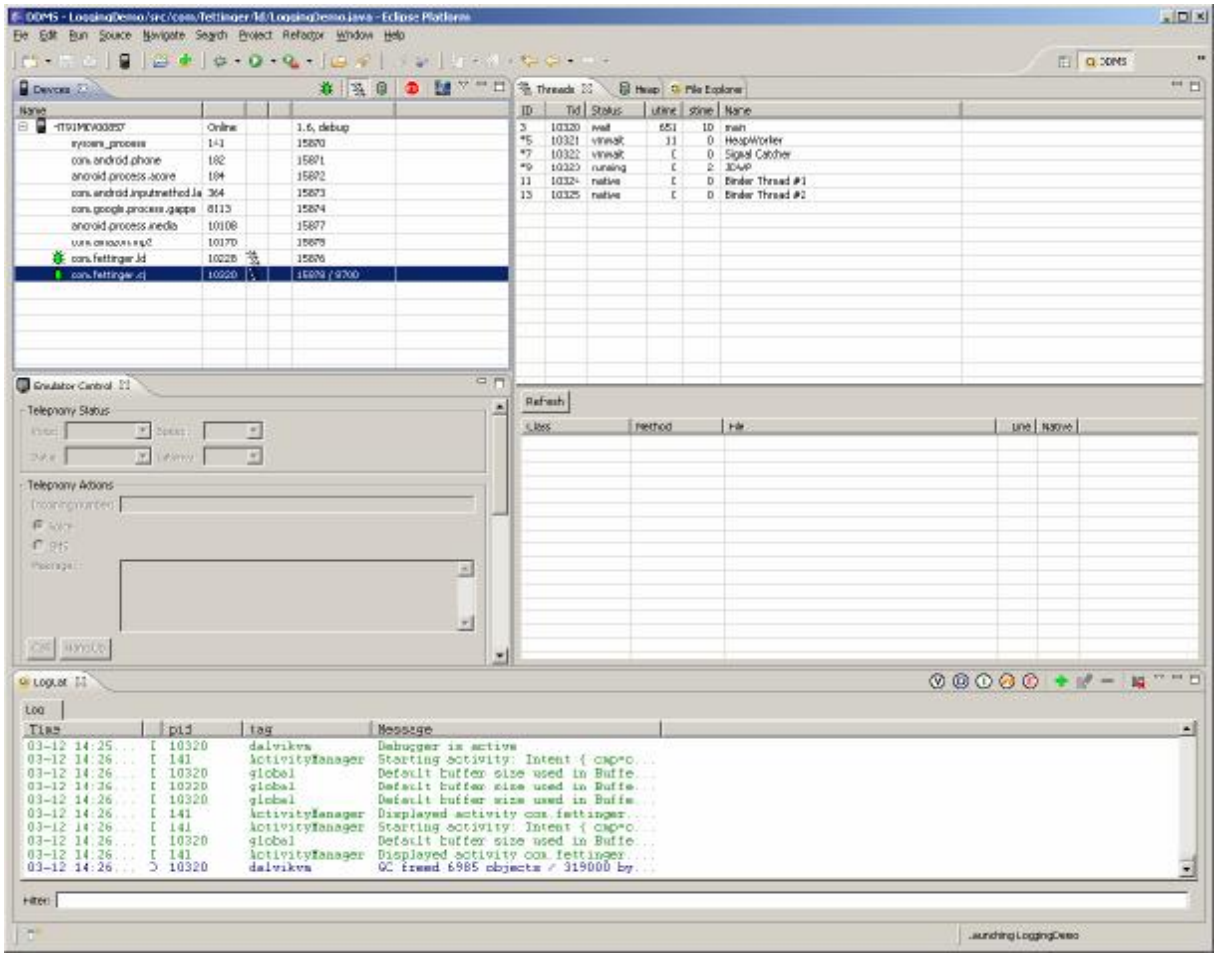

Abbildung 5: DDMS Perspektive in Eclipse

Somit braucht man DDMS nicht extra aufrufen und kann gleich in Eclipse seine Arbeit fortsetzen.

5 Zusammenfassung

Mit den Tools adb, DDMS und Traceview bietet Android den Entwicklern ordentliche Werkzeuge zum Debuggen. Wobei DDMS wegen der grafischen Benutzeroberfläche dem Entwickler etwas mehr entgegenkommt. Weiters ist es von dort auch einfach möglich Screenshots vom Mobilen Endgerät zu erzeugen. ADB bietet hingegen den debugreport an, welcher einen umfassenden Systemauszug ausgibt. Den nutzen dieser Funktion sehe ich eher im Entwickeln von eigenen ROMs, was ja unter Android durchaus möglich ist. Traceview ergänzt diese Tools und ist hervorragend für Performance Analysen geeignet.

Der geschulte Java Programmierer wird natürlich gerne zur normalen debugging Möglichkeit unter Eclipse greifen. Diese Methode ist die einzige, in der der Ablauf des Programms angehalten werden kann und eine Schritt für Schritt Ausführung ermöglicht.

Daher sehe ich den Einsatz von adb bzw. DDMS vor allem darin, fertige Programme auf deren Zustände abzufragen. Hier kann jederzeit durch den Aufruf einer Methode der Klasse Log z.B. die aktuelle GPS Position, das Ergebnis einer Berechnung oder ähliches Ausgegeben werden. Vor allem könnte man einen Enduser bitten im Fall eines Problems ein Protokoll mit adb zu erstellen, da man die Log-Einträge durchaus im fertigen Programm belassen könnte.

Das Debuggen mit Eclipse dient am Besten während dem Entwickeln eines neuen Produkts. Hier ist es möglich den Programmablauf zu stoppen und die Applikation Schritt für Schritt auf Fehler zu untersuchen.

Literaturverzeichnis

[1] Java Platform Debugger Archtitecture
http://java.sun.com/j2se/1.4.2/docs/guide/jpda/architecture.html

Abbildungsverzeichnis

Tabellenverzeichnis