Nepomuk

- digitales Metronom mit Sprache

Projekt/ Tutorium (20556)

Thomas Uhrig

Schwabstraße 86

70193 Stuttgart

Inhaltsverzeichnis

1. Einleitung und Projektidee

Für viele Menschen ist heutzutage ein Mobiltelefon ein alltäglicher Wegbegleiter. Es garantiert nicht mehr nur die ständige Erreichbarkeit, sondern verwaltet Termine, pflegt Aufgaben und fungiert als Kamera oder Spielkonsole. Mobiltelefone sind dabei längst mehr als eine in Hardware gegossene Firmware - sie sind eine Programmierplattform.

Seit dem großen Erfolg des iPhones sind sogenannte *Apps* in aller Munde. Dies sind kleine, teils mehr oder weniger nützliche Programme, die für eben eine solche mobile Plattform ausgelegt sind. Sie können Kalender implementieren, kleine Spiele oder nützliche Helfer für Alltagsprobleme. Für den Nutzer sind sie dabei meist kostenlos bzw. günstig zu beziehen und durch das Handy überall verfügbar. Dieser Vorteil sollte auch für das hier vorgestellte Projekt genutzt werden.

Michiel Oldenkamp, seines Zeichens langjähriger und professioneller Musiker, gab den Anstoß zu diesem Softwareprojekt. Er stellte in seinen Unterrichtsstunden häufiger fest, dass gerade für unerfahrene Musiker das "Klicken" eines Metronoms alleine nicht ausreicht. Sie sollten nicht nur wissen *ob* und *wie* sie im Takt liegen, sondern vor allem auch wo. Seine Idee war es, dass das Metronom nicht nur stupide „klicken“ müsste, sondern auch die Takte mit zählen und ansagen sollte. Ideal wäre folglich ein digitales Metronom mit einer Sprachausgabe, möglichst auf einem mobilen und tragbaren Endgerät wie etwa einem Android-Handy.

Das digitale Metronom mit Sprache sollte hierfür auf verschiedene Sprachaufnahmen in unterschiedlichen Landessprachen zurückgreifen. Es sollte einfach und unkompliziert zu handhaben sein und über ein klassisches Metronom hinaus die gewünschte Funktionalität bieten.

2. Spezifikationen

Michiel Oldenkamp lieferte neben der eigentlichen Grundidee, auch einige detaillierte Spezifikationen für das Projekt. Diese sind im folgenden aufgelistet, wobei Änderungen an den ursprünglichen Spezifikationen kommentiert sind. Die Spezifikation, sind unter anderem auch in Rücksprache mit Michiel Oldenkamp entstanden, bzw. konkretisiert worden.

Tempo in BPM (Beats per Minute)

> *Das Tempo soll in der Einheit Beats per Minute variabel mit einer Nachkommastelle einstellbar sein. Es soll von ca. 48 bis etwa 170 BPM reichen.*

Anmerkung: Ursprünglich sollte das Tempo in ganzzahligen zweier Schritten eingestellt werden können. Da die Variante mit einer Nachkommastelle jedoch wesentlich flexibler ist, wurde diese bevorzugt. Die Bedienbarkeit ist durch eine `SeekBar` unproblematisch und intuitiv. Außerdem macht diese variable Tempoeingabe, die Trigger-Funktion (siehe unten) erst möglich. Sie ist programmiertechnisch zudem leicht umzusetzen. Die Geschwindigkeit ist zwischen 0.1 BPM und 175 BPM beschränkt (die Gründe für die Beschränkung sind im Kapitel „5.4. Audioverarbeitung“ nachzulesen).

Verschiedene Taktarten

> *Es sollten verschiedene Taktarten, z.B. 2/4-Takt, 3/4-Takt etc, einstellbar sein. Je nach Taktart, werden unterschiedliche viele „Klicks“ betont bzw. nicht betont.*

Anmerkung: Die Taktarten sind auf einen Wertebereich von je 1 bis 9 also (1/1 bis 9/9) beschränkt. Dieser Wertebereich ist jedoch mehr als ausreichend, da schon ein 5/4-Takt oder ähnliches, mehr als selten vorkommt. Es wird jeweils der erste Schlag im Takt betont, wobei sich diese Funktion auch abschalten lässt.

Unäre oder ternäre Unterteilung des Schlages

> *Es sollte möglich sein, wahlweise eine binäre, eine ternäre oder gar keine Unterteilung einzuschalten. Diese sollte dann durch ein weiteres „Klicken“ bzw. ein „und“ angesagt werden.*

Anmerkung: Diese Funktion wurde als einzige wegen den Problemen mit der Audioverarbeitung der Android-Plattform nicht implementiert. Es wäre nötig gewesen zwischen den eigentlichen „Klicks“ noch einen (*bei unär*) bzw. zwei (*bei ternär*) weitere Töne unterzubringen. Dies hätte die Frequenz in der die Dateien abgespielt werden müssten erhöht und damit ihre maximale Länge derart beschränkt, dass dies nur bei sehr langsamen Tempo möglich wäre.

Verschiedene Sprachen

> *Es sollten verschiedene Sprachen, wie etwa Deutsch und Englisch, sowohl für Text als auch für Ton angeboten werden.*

Anmerkung: Es wurden in der Implementierung eine Reihe von Sprachen verwendet, die schlicht im Rahmen eines Projekts möglich waren zu realisieren. Die verschiedenen Sprachen wurden im Absprache mit der Sprecherin gewählt. Es sind die Sprachen Deutsch, Englisch, Spanisch, Französisch, Russisch, Italienisch und Niederländisch umgesetzt.

Einzähler für die Stimmaufnahme

> *Es sollte für Stimmaufnahmen eine Einzählfunktion umgesetzt werden, die vor dem Eigentlichen „Ticken“ vorzählt (1, 2, 3, 4...).*

Anmerkung*:* Diese Funktion wurde so umgesetzt, dass bei eingeschalteter Funktion, ein Takt mit Sprache vorgezählt wird. Ist die Sprachausgabe dabei ausgeschaltet, ist nach dem ersten Takt nur noch ein „Klicken“ zu hören. Ist sie hingegen eingeschaltet, so hat diese Funktion keine Wirkung.

Trigger-Funktion

> *Es sollte möglich sein, dass Tempo mit dem Finger ein zu tippen. Je nach Tippgeschwindigkeit, sollte das Tempo automatisch berechnet und eingestellt werden.*

Anmerkung*:* Neben den Vorgaben von Herrn Oldenkamp, wollte ich zusätzlich eine Trigger-Funktion implementieren. Diese sollte das Eintippen des Tempos für das Metronom ermöglichen.

Animation & Zeitangabe

Neben der reinen Tonausgabe, sollte auch eine visuelle Anzeige umgesetzt werden. Dies gibt dem User das Gefühl, dass wirklich etwas „abgespielt“ wird und etwas passiert. Außerdem kann so leicht gezeigt werden wo im Takt man sich befindet. Zudem wurde eine Zeitanzeige umgesetzt und ein Zähler der gespielten Takte.

3. Die Android-Plattform

3.1. Überblick

Android bezeichnet eine Softwareplattform für Handys, Smartphones und mobile Endgeräte. Es wird als quelloffenes Softwareprojekt von der Open Handset Alliance (OHA) entwickelt und verbreitet. Diese ist ein Konsortium aus rund 65 Firmen, welche im Jahre 2007 von der Firma Google ins Leben gerufen wurde. Seitdem liegt ihr Fokus auf der Entwicklung und Vermarktung der Android-Plattform. Mit dabei sind bekannte Namen der IT- und Softwarebranche wie Acer, Texas Instruments und eBay.

Die Android-Plattform findet sich auf momentan 26 Endgeräten wieder, von Herstellern wie z.B. Samsung oder Sony Ericsson. Der Markt an Applikationen umfasst ca. 45.000 Apps (April 2010), von denen ca. 60% frei verfügbar sind. Jeden Monat kommen ca. 9.000 neue hinzu.

Android umfasst ein auf Linux basierendes Betriebssystem, Middleware und einige standardmäßige Schlüsselapplikationen. Es stellt eine Programmierschnittstelle, sowie ein Java-SDK, für die eigens entwickelte Java-Virtual-Maschien Dalvik zur Verfügung. Für diese muss der Java-Code explizit compiliert werden, da sie grundlegend anders arbeitet, als die JVM von Sun.

Die Klassenbibliothek umfasst mehr als 1500 Klassen und rund 400 Schnittstellen. Hierüber werden zahlreiche Features, wie etwa Media-Support, GPS oder SQLite für den Entwickler zugänglich gemacht.

Die Programmierung von Android-Applikationen ist ohne Gebühren, Verträge oder Anmeldungen von diversen Konten möglich. Hiermit stellt das quelloffene System eine echte Alternative zum Konkurrenzprodukt iPhone dar.

3.2. Grundlegender Aufbau einer Android-Applikation

Applikationen für die Android-Plattform werden ausnahmslos in der Programmiersprache Java entwickelt. Sämtlicher Quelltext, sowie alle benötigten Dateien, werden als Archiv mit der Endung *.apk exportiert. Hierfür steht das Android Asset Packaging Tool (aapt) zur

Verfügung. Das entstehende Archiv "ist die entwickelte Applikation" und enthält alle Dateien und erstellten Klassen. Es kann auf verschiedenen Geräten installiert und ausgeführt werden. Eine jede Android-Applikation besteht aus vier verschiedenen Komponenten.

Activities

Aktivities sind einzelne und unabhängige Teile des GUI. So könnte das Metronom etwa eine Aktivity besitzen, in der der Benutzer einen Start- und Stop-Button findet. Eine andere Aktivity wird ein Auswahlmenü bieten und wieder eine andere könnte einen Dialog zur Programmeinstellung implementieren. Alle Aktivities sind unabhängig von einander, bilden aber zusammen das GUI. In ihnen werden verschiedene Views, wie z.B. Buttons oder Images, platziert. Eine Aktivity erbt dabei von der Klasse `Activity`, ein View wiederum von `View`.

Services

Ein Service ist eine Funktion die im Hintergrund ausgeführt wird. Dieser wird auch dann weiter ausgeführt, wenn der eigentliche Programmbildschirm verlassen wird. Alle Services erben von `Service`.

Broadcast Receivers

Broadcast Receivers reagieren auf "Events" die global auf der Android-Plattform geworfen werden. Dies kann z.B. eine Meldung über einen niedrigen Batterie-Status oder einen eingehenden Anruf sein. Sie können dann verschiedene Aktionen einleiten.

Content Providers

Content Providers dienen dem Speichern und Abrufen von Daten (diese können so auch anderen Applikationen zugänglich gemacht werden). Sie erben alle von der Klasse `ContentProvider`.

Neben diesen vier Komponenten, die spezifisch für die Android-Plattform sind, können auch gewöhnliche Java-Klassen geschrieben und genutzt werden. So greift Nepomuk z.B. lediglich auf das Mittel von Activities zurück, der Rest ist herkömmlicher Java-Code.

Eine jede Android-Applikation legt wichtige Meta-Informationen in einem XML-File namens `AndroidManifest.xml` fest. Hier werden z.B. benötigte Bibliotheken oder Icons festgehalten. Jedoch muss auch *jeder* Service und *jede* Aktivity hier dem System bekannt gemacht werden.

Ebenfalls wird hier der Einstiegspunkt in die Anwendung festgelegt. Denn anderes als in "normalen" Java-Programmen, gibt es hier keine `main`-Funktion. Jeder Service und jede Artivity kann einzeln aufgerufen werden, z.B. auch von anderen Programmen. Sie sind unabhängig. Daher kann es auch nötig sein den State einer Aktivity zu speichern, um etwa ein Einstellungsmenü mit den selben "Reglereinstellungen" erneut aufrufen zu könne.

4. Planung und Ablauf

Im Folgenden möchte ich einen groben Überblick über den zeitlichen Ablauf des Projektes geben. Die einzelnen hier aufgelisteten Phasen und Arbeitsschritte, sind jedoch weder streng chronologisch, noch so strikt getrennt, wie die Auflistung vielleicht glauben macht. So erstreckte sich etwa die "Einarbeitung" über das gesamte Projekt hinweg - nämlich immer dann wenn ein neues Problemfeld zu lösen war. Ebenfalls ging der GUI-Prototyp fließend in die Entwicklung der endgültigen Oberfläche über, da diese aus den ersten Versuchen nach und nach entstand. Die Tests und Versuch fanden sich ebenfalls während der gesamten Entwicklungszeit wieder.

Die folgende Auflistung soll damit mehr einen kleine Anhaltspunkt darüber geben, was überhaupt gemacht wurde und wie viel Zeit (relativ zu den anderen Arbeitsschritten) hierfür aufgewendet wurde. Die "Wochenangabe" ist also eine Art Größenverhältnis.

Team und Projektfindung (3 Wochen)

Bei einem Projekt wird häufig, die für Planung und Organisation benötigte Zeit unterschätzt. So vergingen auch bei diesem Projekt rund drei Wochen zu Beginn des Semesters, für Team- und Projektfindung. Beispielsweise gab es einige Treffen und Besprechungen mit Kommilitonen und Professoren bevor das eigentliche Projekt überhaupt beginnen konnte. Zudem war anfangs noch unklar, ob das Projekt alleine oder zu zweit angegangen werden sollte. Auch die Plattform (Android oder iPhone) war unklar.

Einarbeitung (2 Wochen)

Zu Beginn eines jeden neuen Projekts, steht zumeist eine Einarbeitungsphase - so auch bei diesem. Da für mich die Android-Plattform bzw. die Entwicklung mobiler Anwendungen Neuland war, musste ich bei den Grundlagen beginnen. Die Tatsache, dass auf dem Android-Handy die Programmiersprache Java verwendet wird, erleichterte mir jedoch den Einstieg.

Typische Probleme wie etwa "streikende" Eclipse-Plugins und nicht funktionierende Emulatoren nahmen jedoch wiederum mehr Zeit als erwartet in Anspruch. Insgesamt dauert die Planungs- und Einarbeitungszeit dadurch deutlich länger als gedacht. Es verging

mehr als ein Monat, bis endlich Code geschrieben werden konnte. Dies ist besonders im Sommersemester, welches ohnehin kurz ist, problematisch.

GUI-Prototyp (2 Wochen)

Da die Aufgaben des Metronoms schon zu Beginn relativ klar waren, war auch eine grobe Vorstellung der Oberfläche schon gegeben. Sie sollte neben einem Start- und Stop-Button, auch Möglichkeiten zur Manipulation des Tempos und der Taktart bieten. Daher wurden in den ersten Schritten eine provisorische Oberfläche für das Metronom mit einigen Knöpfen erstellt. Mit dieser konnte dann in die Implementierung der Audioverarbeitung eingestiegen werden (dem sicherlich wichtigsten Part).

Erste Audioverarbeitung (1 Wochen)

Der wichtigste, aber am Anfang noch "unklarste" Teil der Entwicklung war die Audioverarbeitung. Wie können Dateien abgespielt werden? Wie können sie in einem konstanten und regelmäßigem Takt abgespielt werden? - Da dieser Teil essentiell für das Metronom ist, wurde schon früh versucht hier Lösungsansätze zu finden.

GUI (1 Wochen)

Als die Audioverarbeitung zunehmend Gestalt an nahm, wurde die Oberfläche fertig gestellt. Hierbei wurden zunächst alle nötigen Buttons ohne Funktion implementiert. Nach und nach wurden diese dann mit Methoden verknüpft und so die Funktionalität implementiert. Außerdem wurden freie Icons für die Buttons gesucht und mit in die Anwendung eingebunden.

Einstellungen (2 Wochen)

Als das Metronom alle grundlegenden Eigenschaften implementiert hatte, wurden die Einstellungsmöglichkeiten umgesetzt. Diese waren nicht von Anfang an klar. So wurden z.B. die Einstellungen einer unären/ ternären Unterteilung mehrfach von Checkbuttons auf Listen geändert und letztendlich grundsätzlich verworfen. Außerdem bereitete der Umgang mit Propertie-Files auf dem Android-Handy unerwartete Probleme. Die Möglichkeit in das eigene Programm Nutzereinstellungen einzubinden, wird hier grundlegend anders umgesetzt als in einer herkömmlichen Java-Anwendung, da es nicht ohne weiteres möglich

ist, beliebig Dateien zu schreiben. So können Propertie-Files zwar gelesen, nicht aber geändert werden.

Desing for Performance (3 Wochen)

Die Ausgabe der Sprachdateien zum eigentlichen Ticken machte zu Beginn große Probleme. Es war unklar, ob und wie genau dies realisiert werden könnte. Als dann ein erster Weg gefunden war (über eine `SoundPool` Klasse), machte das Timing Probleme. Es war schwer die Audiodateien im immer gleichen Takt abzuspielen. Daher wurden verschiedene Lösungen ausprobiert, wie etwa der `MediaPlayer` bzw. der `SoundPool`, `Handler` oder `Thread`, Ticken und Stimme getrennt oder in einer Datei, Wave- oder OGG-Format.

Dieses Problem ging unerwartet Tief in die Materie ein, da teilweise auf den Garbage Collector und ähnliches Rücksicht genommen werden musste.

Spracheinstellungen für Schrift und Ton (1 Wochen)

Die Aufnahme der Sprachdateien, bzw. das verfassen der Texte in verschiedenen Sprachen, machte ebenfalls unerwartet viel Arbeit. Diese mussten so gestaltet werden, dass im Nachhinein keine Änderungen mehr nötig waren, da es zum Teil sehr viele Audiodateien wurden. Sie mussten gewissenhaft geschnitten werden (gleiche Länge etc.), was schlicht einige Zeit in Anspruch nahm. Außerdem musste sich mit der Sprecherin für einen Termin im HoRadS-Studio abgestimmt werden.

Tests und Versuche (1 Wochen)

Während den Arbeiten am Metronom, wurden verschiedene Methoden und Klassen für Problemstellungen ausprobiert. So wurden z.B. Handler-Klassen und klassische Threads ausprobiert, MediaPlayer bzw. Soundpools. Dies wurde im Programmcode immer wieder umgestellt und nach Tests bzw. der weiteren Entwicklung zum Teil wieder verworfen. Außerdem wurden teilweise Klassen erstellt, welche im Projektverlauf wieder umorganisiert wurden bzw. unnötig wurden.

Sonstiges (1 Woche)

Es fielen zudem typische Arbeiten wie Organisation, die Suche kleinerer Fehler oder dem Testen auf einem Android-Telefon an. Auch diese nahmen über das gesamte Semester hin viel Zeit in Anspruch.

Gegen Ende des Projekts benötigt außerdem die Dokumentation einiges an Zeit, wenn auch durch das ständige Mitschreiben während des Semesters, weniger als gedacht. Außerdem stand zu diesem Zeitpunkt noch die Media Night an, wo ebenfalls Arbeiten wie Logo- und Posterdesign, neben der eigentlichen Entwicklung anfielen.

Fazit

Bei der Entwicklung des Projekts, floss viel Zeit in Probleme, die zu Beginn noch unklar waren. Hierbei machten zum Teil Kleinigkeiten und Eigenheiten der Android-Plattform einige Probleme. So gab es z.B. Buggs, die erst spät, beim Teste auf dem eigentlichen Endgerät erkannt wurden. So stürzte Anfangs die gesamte Applikation z.B. ab, wenn die Tastatur des Handy aufgeklappt wurde. Dies startete nämlich die Einstiegsklasse neu, welche wiederum alle Objekte neu erzeugte. Dieses Problem wurde zwar gelöst, war aber bei der Entwicklung auf dem Emulator, lange Zeit nicht aufgefallen.

Als positiv empfand ich persönlich die Projektentwicklung mit Redmine. Dieses nutze ich sowohl als Notizbuch für Ideen, als Zeitkonto für einen groben Überblick, sowie auch als Wiki zur Dokumentation. Besonders dieses ständige parallele Mitschreiben, hat im Nachhinein viel Arbeit erspart.

5. Implementierung

5.1. Architektur und Designentscheidungen

Bei der Architektur wurde auf einen sauberen und gut strukturierten Code geachtet. Es liegt eine strenge Aufgabenteilung vor, bei der das Model-View-Controller Pattern (MVC-Pattern) als Vorbild diente. Außerdem wurde darauf geachtet, dass die Implementierung schnell ist und wenig „Müll“ erzeugt. Im folgenden soll ein Überblick über die einzelnen Klassen und deren Aufgaben gegeben werden.

Package `nepomuk.animation`

`Class Balken`: Diese Klasse enthält die Informationen über den weißen Balken, welcher die Position im Takt anzeigt. Hier werden Größe, Farbe und Aussehen des Balkens bestimmt. Außerdem stellt diese Klasse grundlegende Methoden zur Manipulation des Balken (etwa der Position) zur Verfügung.

`Class Grid`: Diese Klasse enthält die Informationen über den Grid, der unterhalb des Positionsbalken angezeigt wird. Auch hier werden Aussehen und Größe festgelegt. Außerdem werden hier die *.JPG-Dateien mit den Zahlen geladen und an der korrekten Stelle gezeichnet. Diese Klasse enthält ebenfalls grundlegende Methoden zur Manipulation des Grids. So muss dieses z.B. geändert werden können, falls der Takt geändert wird.

Package `nepomuk.interfaces`

`Interface Updateable`: Dieses Interface sieht eine einzige Methode `public void update()` vor. Das Interface soll von allen Klassen implementiert werden, welche sich aktualisieren sollen, falls eine Nutzereinstellung (z.B. die Sprache) geändert wurde. Die `update()`-Methode soll dabei mit Hilfe der Klasse `PUtil`, die für die Klasse nötigen Nutzereinstellungen in lokalen Variablen speichern. Hierzu ruft `PUtil` alle bei ihr registrierten Klassen auf, die dieses Interface implementieren. Diese rufen von `PUtil` wiederum ihre benötigten Attribute ab und speichern sie in Variablen. Dies erhöht die Performance, da z.B. nicht in einem Thread permanent diese Information abgefragt werden, sondern einmalig bei Einstellungsänderungen.

Package `nepomuk.index`

`Class Start`: Dies ist die Einstiegsklasse der Applikation. Hier wird die Benutzeroberfläche zusammengesetzt und Listener-Objekte erzeugt. Außerdem werden hier alle nötigen Objekte zentral instantiiert und verknüpft. Zudem löst diese Klasse in ihrem Konstruktor das erstmalige Laden der Nutzereinstellungen aus.

Sie implementiert das Interface `Updateable`, da sie die Beschriftung ihrer Buttons bei Änderung der Spracheinstellung aktualisieren soll.

`Class Settings`: Dies ist die zweite Activity der Applikation. Sie stellt das Einstellungsmenü für den Nutzer dar. Auch sie implementiert das Interface `Updatable` und aktualisiert die Beschriftung ihrer Komponenten. Außerdem ist diese Klasse dafür verantwortlich, dass bei einer Änderung der Nutzereinstellungen alle anderen Klassen informiert werden. Hierzu ruft sie die Methode `update()` der Klasse `PUtil` auf. Hier haben sich alle Klassen registriert, welche das Interface `Updateble` implementieren. Diese werden wir oben beschrieben aufgerufen.

Package `nepomuk.controllers`

`Class PlayController`: Hier findet die Audioverarbeitung statt. Diese Klasse verwaltet die Audio-Objekte und spielt diese im Takt ab. Ihr wird ein Objekt vom Typ `TaktController` übergeben, da dieses zentral alle Informationen über den Takt hält. Es kennt Geschwindigkeit, Position und Taktart.

`Class TaktController`: Diese Klasse hält zwei Objekte vom Typ `BPM` und `BAR`, die Modelle der Applikation. Außen stehende Objekte können die Modelle nur über den Controller ändern. So weiß der Controller, dass eine Änderung vorliegt und kann seine Views aktualisieren - die `TextView`-Objekte der Oberfläche, die Tempo und Taktart anzeigen.

`Class TimeController`: Dieser Controller aktualisiert ständig ein `TextView` in welchem die aktuelle Abspielzeit angezeigt wird. Hierzu implementiert diese Klasse einen eigenen Handler, dem zeitlich versetzt eine Nachricht zugeschickt werden kann. Dieser kann daraufhin eine beliebige Methode ausführen.

Die Klasse `TimeController` ist somit nicht vom `PlayController` abhängig. Falls die

Zeitanzeige und die Abgespielten Töne leicht auseinander laufen ist dies nicht weiter schlimm.

Diese Klasse ist jedoch relativ teuer. Sie erzeugt eine Vielzahl von Objekten, da permanent die Zeit abgefragt und ausgerechnet werden muss. Außerdem werden ständig neue Strings erzeugt, die dann in den View geschrieben werden. Daher wurde ein Kompromiss zwischen einer detaillierten Zeitanzeige und wenigen erzeugten Objekten eingegangen.

`Class LineController`: Dieser Controller steuert die Animation und die Textausgabe der Taktposition. Er wird vom `PlayController` nach jedem Abspielen einer Audiodatei alarmiert und setzt die Animation entsprechend um eine Stelle weiter. Außerdem aktualisiert er ebenfalls eine Textanzeige über die bislang gespielten Takte.

Alle Controller implementieren das Interface `Updatable` und aktualisieren bei Änderungen ihre GUI-Komponenten für die sie zuständig sind. Außerdem sind alle Controller Singeltons. Wird nämlich der Bildschirm des Android-Handys gekippt oder die Tastatur ausgefahren, so startet das Gerät automatisch die momentane Activity neu. Da aber in eben dieser alle Objekte zentral instantiiert werden, führt dies zu Problemen. Es würden theoretisch zwei Audio-Loops entstehen, wobei der eine Thread nicht mehr gestoppt werden könnte. Daher dürfen die Controller nur einmal im System auftauchen.

Package `nepomuk.modells`

`Class BPM`: Eines der beiden Modelle. Hier werden alle Informationen zum Tempo gehalten. Sie können komfortable abgefragt werden (verschiedene Formate, z.B. `double` oder `long`).

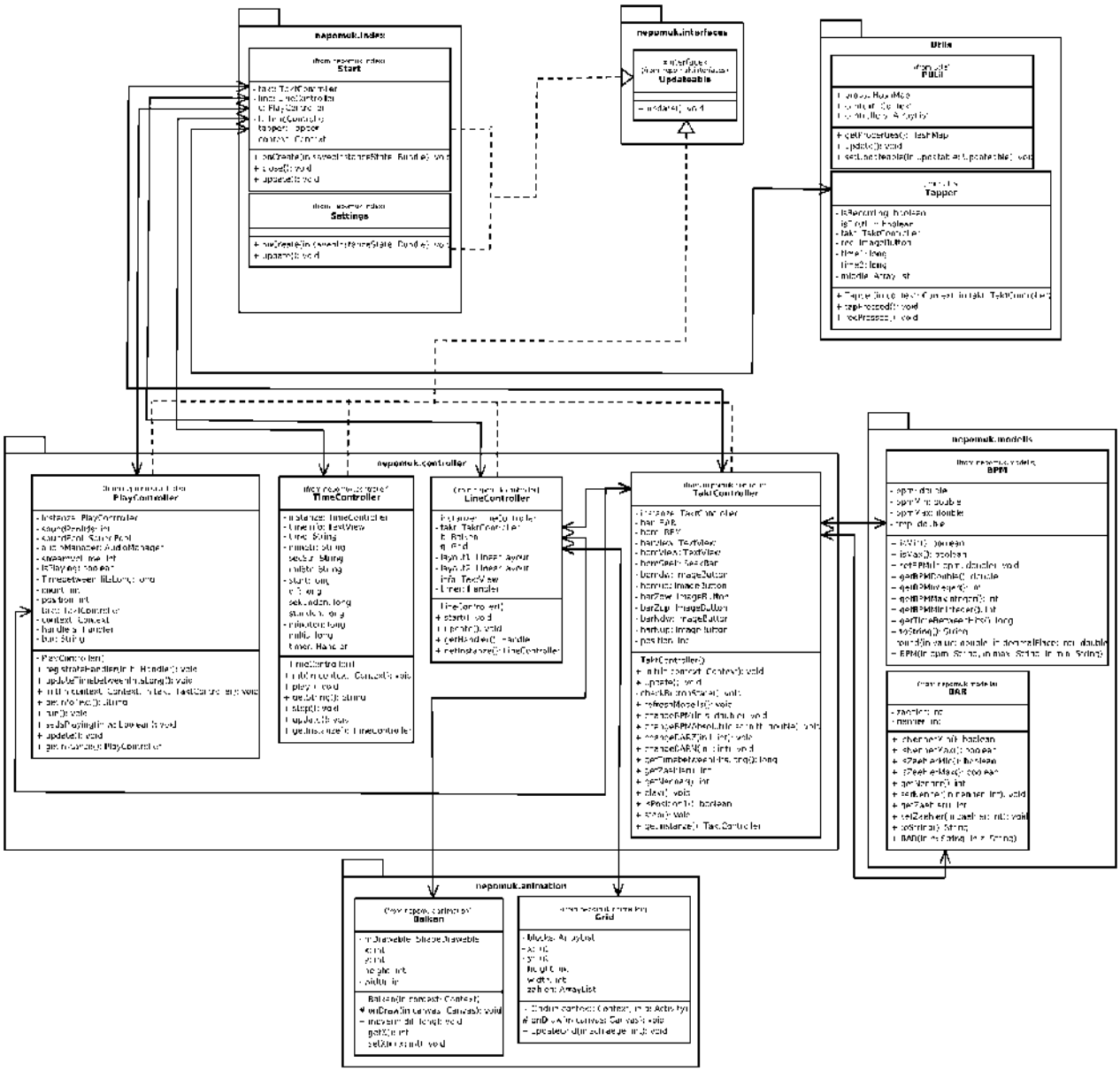
`Class BAR`: Eines der beiden Modelle. Hier werden alle Informationen zur Taktart gehalten. Sie können ebenfalls über verschiedene Methoden abgefragt werden. Außerdem wird kontrolliert, dass keine unzulässigen Werte eingestellt werden.

Package `nepomuk.utils`

`Class Tapper`: Die Klasse `Tapper` berechnet ein live eingetipptes Tempo. Sie hält einen Verweis auf das Objekt der Klasse `TaktController`, da sie dieses ändern muss. Außerdem hält sie einen Verweis auf den `Rec`-Button, da sie dessen Anzeige ändern soll.

`Class PUtil`: Diese Klasse stellt einige `static`-Funktionen bereit um die Properties und Einstellungen zentral zugänglich zu machen. Startet die Applikation, so liest diese Klasse erstmalig alle Einstellungen aus. Sie macht diese dann als `public static`-Attribut systemweit verfügbar. Alle Klassen greifen dynamisch auf dieses Attribut zu. Werden Nutzereinstellungen geändert, so wird zuerst in dieser Klasse ein Update angestoßen. Sie informiert dann alle bei ihr registrierten Objekte des Typs `Updatable`. Diese ziehen sich dann die von ihnen benötigten Einstellungen aus dieser Klasse und speichern sie in Variablen.

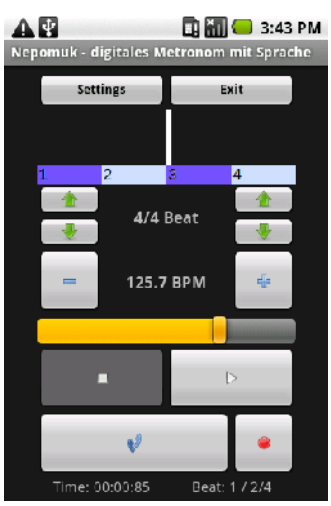
UML



5.2. GUI

Anforderungen

Die Oberfläche einer Mobilen-Applikation muss verschiedenen Ansprüchen genügen. Diese unterscheiden sich zum Teil von denen einer herkömmlichen Anwendung. Zum Einen, muss das GUI eine hinreichende Möglichkeit bieten die Applikation und ihre Funktionen zu bedienen. Dieser Anspruch kollidiert jedoch häufig mit einem anderen. Denn zum Zweiten muss das GUI eine übersichtliche und schlichte Bedienung ermöglichen. Gerade für Mobile-Applikationen, welche häufig mit kleinen Bildschirmen und schlechten Tastaturen zu kämpfen haben, ist das Stichwort Usability ein zentrales Thema. Alle Buttons und Regler müssen eine ausreichende Größe aufweisen, um auch über einen Touchscreen bedient werden zu können.



Alle Bedienelemente für das Metronom sollten auf einer Activity sichtbar sein. Es sollte ohne umschalten in ein anderes Menü möglich sein, Takt und Tempo zu ändern und das Metronom zu starten bzw. zu stoppen. Lediglich die Einstellungen wie z.B. die Sprache, sollten sich in einer zweiten Activity wieder finden. Jedoch ist auch diese nicht verschachtelt und zeigt alle Einstellungsmöglichkeiten in einer Ansicht. Somit ist die Menütiefe sehr gering und intuitiv zu überschauen.

Implementierung

Die GUI einer Android-Anwendung wird durch ein oder mehrere XML-Files festgelegt. Diese können komfortable mit verschiedenen Tools, wie DroidDraw oder den Android-Eclipse-Tools, erstellt und editiert werden. Die XML-Dateien finden sich im Android-Projekt in `res/layout/main.xml` wieder. Dort werden sämtliche GUI-Komponenten, wie Buttons oder Textfelder beschrieben (Größe, Lage, Farbe, ID etc.).

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical" android...>

      <LinearLayout android:orientation="horizontal"...>
                  <Button android:id="@+id/settings"... />
                  <Button android:id="@+id/exit"... />
      </LinearLayout>

      <LinearLayout android:id="@+id/timelinelay".../>
      <LinearLayout android:id="@+id/timegridlay".../>
...
```

Jede dort erstelle Komponente wird dann im Java-Code geladen. Ihr kann so ein Listener-Objekt hinzugefügt werden, welches z.B. bei einem "click" auf einen Button reagiert. Die erstellten Komponenten, werden dabei über die automatisch generierte Klasse `R` referenziert:

```
((ImageButton) this.findViewById(R.id.play)).setOnClickListener(
new OnClickListener(){
      public void onClick(View v) {
                  System.gc();
                  t.play();
                  ...
}});
```

Die Android-Plattform bietet zur Entwicklung ein Set eigener GUI-Elemente an. Hierzu zählen unter anderem `Button`, `SeekBar`, `RadioButton` und `CheckButton`. Jedoch ist die Auswahl an Komponenten relativ stark eingeschränkt. So findet sich z.B. kein Drop-Down-Menu oder Schieberegler mit integrierten Buttons. Dies schränkt die Entwicklung zum Teil ein.

Für die Usebility der Applikation, wurden jedoch auch einige Bewusste Einschränkungen in puncto Bedienbarkeit eingegangen. Der Nutzer soll etwa Takt und Geschwindigkeit lediglich über Buttons bzw. eine `SeekBar` bedienen können. Ein Eingabefeld für Freitext

macht die Handhabung für den Nutzer unnötig kompliziert. Das Display sowie die Tastatur sind relativ klein und ein Eingabefeld würde auf manchen Geräten eine aufpoppende digitale Tastatur zur folge haben. Außerdem müssten falsche Eingaben, wie etwa negative Zahlen oder Buchstaben gefiltert werden. Daher wurde auf diese *freihand* Möglichkeit bewusst verzichtet.

5.3. Konfiguration

Das Metronom soll verschiedene Einstellungsmöglichkeiten bieten. Angefangen von Kernaufgaben, wie der Sprachauswahl für Text und Ton, bis hin zur Möglichkeit ein Anzählen ein- bzw. auszuschalten. Diese Einstellungen sollen persistent gespeichert werden, um bei jeder Ausführung der Applikation verfügbar zu sein.

Der "klassische Weg"

Der "klassische Weg" dies zu realisieren führt über den Einsatz von Java-Propertie-Fiels, welche auch auf der Android-Plattform verfügbar sind. Sie können im Ordner `res/raw` abgelegt werden und dann im Programm genutzt werden. Jedoch führt dieses Vorgehen, für sich allein genommen in eine Sackgasse. Die Android-Plattform bietet anders als herkömmliche Betriebssysteme, keine uneingeschränkte Möglichkeit an, Dateien zu lesen, als auch zu schreiben. Daher lassen sich die Propertie-Files zwar einlesen, nicht jedoch mit veränderten Einstellungen wieder abspeichern. Dies ist für die Applikation jedoch zwingend erforderlich. Daher muss neben den Properties-Files, eine zweite Android-Technologie genutzt werden: der Preferences-Mechanismus.

Preferences

Der "Lösungsweg" führt auf der Android-Plattform über den sogenannten Preferences-Mechanismus. In einem XML-File können sogenannte Preferences, also Einstellungen wie etwa ein CheckButton mit einem Namen/ ID, festgelegt werden, aus denen heraus dann automatisch ein Einstellungsfenster mit GUI-Elementen generiert wird. Dieses wird anfänglich mit Default-Werten belegt. Jedoch wird jede Änderung intern persistent gespeichert. Hierzu unterhält die Plattform eine SQLite-Datenbank, deren Benutzung für den Entwickler jedoch völlig transparent bleibt.

```xml
<?xml version="1.0" encoding="utf-8"?>

<PreferenceScreen xmlns:android="http:...">

      <ListPreference android:key="lang" .../>
      <ListPreference android:key="schema"... />

      <CheckBoxPreference android:key="stimme"... />
      <CheckBoxPreference android:key="ak"... " />

</PreferenceScreen>
```

Die Einstellungen für die Applikation setzten sich somit aus zwei Bestandteilen zusammen: Der Preferences-Mechanismus speichert individuelle User-Einstellungen. Anhand von diesen, werden dann aus dem hinterlegten Propertie-File, die jeweils benötigten Attribute ausgelesen. Diese werden anhand von Namenskonventionen gefunden (in der Form xxx_SPRACHE, z.B. settings_Deutsch bzw. settings_English). Diese werden dann zentral in der Applikation verfügbar gemacht (ein `public static` Attribut der Klasse `PUtil`). Die jeweiligen Klassen bedienen sich aus diesem Pool, der dynamisch bei Einstellungsänderungen aktualisiert wird.

Alle Klassen welche für GUI-Elemente zuständig sind (also Controller und Activitys) implementieren das Interface `Updateable`. Wird eine Nutzereinstellung geändert, wird zuerst der Konfigurationspool `PUtil` aktualisiert. Dieser informiert dann alle bei ihm registrierten Objekte vom Typ `Updatebale`.

```
public static void update() {
            getProperties();
            for(int i = 0; i < controllers.size(); i++)
                  controllers.get(i).update();
      }
```

5.4. Audioverarbeitung

Die Audioverarbeitung ist der wichtigste Teil der Applikation. Sie umfasst sowohl das zeitlich genaue Abspielen der Audiodateien, als auch die Ausgabe bzw. Auswahl der richtigen Töne. Sie ist somit für die Performance des Metronoms sehr wichtig.

Vor der eigentlichen Audioverarbeitung, steht jedoch die Aufbereitung der Audiodateien, für die es zwei prinzipielle Möglichkeiten gibt:

Möglichkeit 1: Ticken und Stimme getrennt

Die sauberste Methode zur Erstellung der Audiodateien, wäre es Ticken und Stimmausgabe in unterschiedlichen Audiodateien zu halten. Diese müssten dann lediglich zeitgleich abgespielt werden. Der Aufwand für die Erstellung der Dateien wäre damit minimal. Es wäre für jede Sprache nur eine Zahlenreihe von 1 bis 9 notwendig und für jedes Klangschema ein betontes, sowie ein unbetontes Ticken.

Dieser Ansatz führt jedoch zu einigen Problemen. Zum einen gibt es keine Garantie darüber, dass die beiden Audiodateien wirklich gleichzeitig zu hören sind. So kann etwa die Sprachausgabe, obwohl im Programmcode in der Zeile danach ausgegeben, leicht verzögert zu hören sein. Diese Verzögerung ist dabei willkürlich und durch die VM bedingt, nur schwer in den Griff zu bekommen. Das zweite Problem ist zwar unscheinbarer, aber dafür schwerwiegender. Denn selbst wenn es gelingen würde beide Audiodateien gleichzeitig abzuspielen, wäre das Problem nicht gelöst, im Gegenteil. Es muss nämlich nicht das Wort und das Ticken parallel zu hören sein, sondern die Betonung des Wortes und das Ticken. So klingt etwa das Wort "zwei" erst langsam mit einem "zsss"-Laut ein, bevor die Betonung auf die Silbe "ei" bzw. "wei" kommt. Daher müsste das Wort kurz vor dem Ticken angespielt werden, damit beim Ticken gerade die Silbe "ei" erklingt. Beim Wort "acht" wäre es genau andersherum (die Betonung liegt hier sicherlich auf dem Wortanfang, "a") und beim Wort "drei" wäre es ähnlich wie beim Wort "zwei", jedoch sicherlich um einige Millisekunden verschoben.

Dieses Problem müsste also für jedes Wort individuell gelöst und programmiert werden (in jeder Sprache!). Da zudem das wirklich gleichzeitige Abspielen der Dateien ohnehin schon Schwierigkeiten bereitet, wäre ein gleichzeitiges Abspiele unter Berücksichtigung der Betonungen kaum umsetzbar. Daher führt dies Methode in eine Sackgasse.

Möglichkeit 2: Ticken und Stimme in vorgefertigten Dateien

Diese Methode klingt zwar anfänglich nach unnötigem Aufwand und keineswegs nach sauberer Programmierung, sie löst aber die oberen beiden Probleme. Zum Einen ist sie für die Performance des Metronoms wesentlich besser. Es müssen nicht mehr zwei Dateien zeitgleich abgespielt werden, sondern nur noch eine vorgefertigte. Das Problem mit der parallelen Verarbeitung entfällt also. Außerdem muss die Betonung des Wortes nicht mehr individuell berücksichtigt und mit dem Ticken abgestimmt werden. Das Wort und das Ticken befindet sich schließlich schon korrekt positioniert in einer Datei. Ist zudem jede Datei mit einer kurzen Pause zu Beginn geschnitten, die entweder zum Einklingen eines Wortes (z.B. "zsss" bei "zwei") genutzt werden kann, oder auch nicht, können alle Dateien gleichermaßen abgespielt werden. Daher entfällt auch dieses Problem.

Anmerkung: *Jedoch könnte mit derart vorbereiteten Audiodateien auch die vorherige Variante benutzt werden, vorausgesetzt die Dateien werden wirklich zeitgleich abgespielt. Ist dies nicht der Fall, so fällt die Verzögerung doppelt ins Gewicht, da das Ticken und die Stimme nicht im Takt sind und zudem die Betonungen nicht stimmen.*

Nachteilig an dieser Lösung, ist die große Zahl an Audiodateien die im Vorfeld erstellt werden müssen. Bei drei Klangschemata, sieben Sprachen mit je 9 Zahlen und je zwei Tick-Sounds (betont und unbetont), wären ca. 195 Dateien.

Diese Vorarbeit zahl sich jedoch in der Performance und einfachen Programmierung aus. Da die Dateien außerdem im Wave-Format gespeichert werden und so ein aufwändiges Encoding entfällt ist dies sehr performant. Da zudem alle Dateien sehr kurz sind (< 0.5 sec.), ist der Speicherbedarf noch relativ klein.

Unschön an dieser Menge von Dateien ist jedoch deren Einbindung in den Programmcode. Da alle Audiodateien über Attribute der automatisch generierten Klasse `R` referenziert werden (z.B. `R.raw.sound_1`), können sie nicht anhand von Einstellungen und zusammen gesetzten Strings geladen werden. Daher beinhaltet der `PlayController` eine relativ große, wenn auch sehr redundante `Update` Methode. In ihr sind alle Audiodateien für die jeweiligen Spracheinstellungen fest codiert und werden anhand von if-Konstrukten ausgewählt und geladen.

Implementierung

Die Android-Plattform bietet die Möglichkeit sowohl Audio- als auch Videodateien zu verarbeiten. Es ist möglich verschiedene Formate aus dem Dateisystem, dem Sourcefolder oder einer über Internet erreichbaren Quelle zu nutzen. Eine detaillierte Übersicht der verfügbaren Formate, ist unter http://developer.android.com/guide/appendix/media-formats.html einzusehen.

Möglichkeit 1: Die Klasse `MediaPlayer`

Die Android-Klassenbibliothek bietet für die Einbindung von Audio- sowie Videodateien verschiedene Klassen an. Hierbei bietet die Klasse `MediaPlayer` die einfachste und intuitivste Art der Nutzung. Sie ermöglicht es eine beliebige Audiodatei in einer Instanz zu halten und sie mit `play()` abzuspielen. Jedoch ist sie in ihren Möglichkeiten relativ stark beschränkt. So bietet der `MediaPlayer` keine Möglichkeit mehrere Audiodateien gleichzeitig abzuspielen oder sie in Geschwindigkeit und Lautstärke zu manipulieren.

Um die Dateien in der Anwendung nutzen zu können, werden diese im Ordner `res/raw/` abgelegt und können dann über die automatisch generierte Klasse `R` referenziert werden: `R.raw.sound_1`. So können sie einem Objekt der Klasse `MediaPlayer` bekannt gemacht werden:

```
MediaPlayer mp1 = MediaPlayer.create(context, R.raw.file_2);
…
if (takt.isPosition1())
      mp2.start();
else
      mp1.start();
```

Der Parameter `context` ist dabei die Activity in der die Datei abgespielt werden soll.

Der `MediaPlayer` bietet zwar eine Methode `public void setLooping (boolean looping)` an, mit der ein Loop der Audiodaten ein- bzw. ausgeschalten werden kann. Jedoch ist es nicht möglich mit dieser Funktion die Länge des Loops zu bestimmen. Es wird automatisch die Länge der Audiodatei verwendet. Des weiteren ist es nicht möglich mehrere Audio-Dateien abwechselnd zu loopen.

Möglichkeit 2: Die Klasse SoundPool

Ähnlich wie der MediaPlayer kann die Klasse SoundPool genutzt werden um Audiodateien zu laden und abzuspielen. Jedoch können in einem SoundPool mehrere Dateien, wesentlich komplexer verwaltet werden. So ist es etwa möglich, für jede Datei einen Lautstärkeparameter mit zu übergeben, bzw. die Abspielgeschwindigkeit (inklusive der Tonhöhe) zu beeinflussen. Dabei erwies sich diese Variante als nicht weniger schnell in der Praxis. Daher wird in der Applikation ein SoundPool verwendet, was spätere Erweiterungen flexibler macht.

Die Klasse PlayController

Die eigene Klasse PlayController soll die Taktgebung kontrollieren. Hierfür leitet sie die Klasse Thread ab und führt in zeitlichen Abständen die Audiodatei aus. Sie wird zu Beginn des Programmstartes instantiiert und als Thread gestartet. Der Button "Play" bzw. "Stop" setzten eine Variable boolean isPlaying auf true bzw. false. Entsprechend wird ein Audiofile gespielt oder nicht. Hierzu hält der PlayController eine Instanz eines SoundPools zur Wiedergabe der Audiodateien. Die Update Methode lädt hierbei je nach Nutzereinstellungen die nötigen Audiodateien. Diese sind komplett vorgefertigt.

Die Informationen über den Takt (Taktart und Tempo) sind über die Klasse TaktController zugänglich. Sie hält wiederum zwei Objekte der Klasse BAR und BPM (Modelle). Diese kapseln die Datenstruktur der Takt- bzw. Tempoinformation und können sie in verschiedenen Formaten (String, Long, Zähler und Nenner) zurückgeben.

Über die Methode sleep(...) wird der Thread schlafen gelegt und läuft erst nach der entsprechender Zeit wieder an. Dies kann letztendlich zur Vorgabe des Tempos dienen. Wichtig ist hierbei, dass der Thread punktgenau aufgeweckt wird.

```
public void run(){
            while(true){
                    position = takt.getPosition();
                    if(isPlaying){
                            takt.play();
                            if ( position == 1 ){
                                    soundPool.play(soundPoolIds[0]...);
                            else if ( position == 2 )
                                    soundPool.play(soundPoolIds[1]...);
```

5.5. Animation

Das Metronom soll neben der akustischen Ausgabe, auch eine optische Information über die Position im Takt enthalten. Hierzu soll eine Art Animation, ähnlich einem Audio-Sequencer, während des Abspielens mitlaufen. Sie gibt zum Einen Informationen über den Takt wieder (Position, Anzahl der Schläge) und zum Anderen vermittelt sie dem User auch den Eindruck einer "laufenden" Applikation.

Möglichkeiten der Android-Plattform

Die Android-Plattform bietet die Möglichkeit einfache Animationen in eine Anwendung einzubauen. Hierfür stehen verschiedene Mechanismen bzw. Bibliothek zur Verfügung.

Die `Tween Animation` erlaubt einfach Animationen von Texten. Sie ermöglicht Rotation, Skalierung und Translation. Diese Techniken lassen sich dabei beliebig verknüpfen (simultan und sequentiell) und können bequem in einer XML-Datei definiert werden.

Die `Frame Animation` erlaubt es einzelne Bilder als Sequenz hintereinander zu animieren. Sie ist also vergleichbar mit einem Filmstreifen, welcher abgefahren wird. Die Animation kann dabei beliebig gestartet und angehalten werden. Außerdem ist es möglich sie als Loop abzuspielen.

Die Klasse `ShapeDrawable` erlaubt es ähnlich der Java 2D-Bibliothek einfache geometrische Formen frei zu zeichnen. Mit dieser Methode kann am feinsten auf den Ablauf der Animation Einfluss genommen werden. Jedoch muss diese "von Hand" programmiert werden (inklusive Threads).

Implementierung

Für die "Animation" der Zeitlinie in der Applikation wurde auf die Klasse `ShapeDrawable` zurückgegriffen. Diese bietet die meisten Möglichkeiten und ist am leichtesten zu manipulieren (z.B. um sich im Takt zu bewegen). Die Animation ist zudem relativ einfach und ist somit auch von Hand umzusetzen. Im ersten Schritt wird zunächst eine Animation programmiert, welche den Balken bei jedem Schlag, eine Taktstelle weiter setzt. Dies ist vorerst ausreichend und gibt einen optischen Eindruck von der Position wieder. Später ist eine flüssige Bewegung denkbar.

6. Design for Performance

Die wohl wichtigste Eigenschaft eine Metronoms klingt zunächst relativ banal: es soll den Takt halten. Von der Softwareseite her betrachtet ist diese Aufgabenstellung jedoch relativ komplex - *insbesondere in Java auf einem Mobiltelefon*.

Das Problem ist, dass Java keinerlei Garantien darüber gibt, wann ein Thread ausgeführt wird. Hat die VM andere Arbeiten zu erledigen (z.B. den Garbage-Collector zu starten) oder ist aufgrund von User-Eingaben mit neuen Rechenaufgaben konfrontiert, so kommt sie sprichwörtlich "aus dem Takt". Es kann daher keine sichere Aussage darüber gemacht werden, wann ein Thread des Entwicklers abgearbeitet wird. Außerdem bietet ein Mobiltelefon nur eine relativ beschränkte Rechenleistung an. Daher sind auch Aufgaben wie das Laden, Decodieren und Abspielen eines Audiofiles zeitaufwändig und komplex. Es muss somit auf eine schlanke und performante Programmierung geachtet werden. Folgende Punkte wurden hierbei in der Applikation umgesetzt.

Vorgefertigte Audiodateien

Es werden nicht zwei Audiodateien parallel abgespielt (Ticken und Stimme) sondern je nach Einstellungen eine schon vorgefertigte Audiodatei geladen. Dies sorgt zwar für eine große Zahl an vorgefertigten Audiodateien, ist jedoch wesentlich performanter. Die Arbeit, die in der Entwicklung in die Erstellung der Audiodateien geflossen ist, erspart während der Laufzeit Rechenzeit. Diese Lösung ist zwar nicht so flexibel und aus programmiertechnischer Sicht eher unschön, läuft jedoch problemloser.

Außerdem wird ein zweites Problem hiermit gelöst: Die Sprachausgabe muss nicht bzw. darf nicht gleichzeitig mit dem Ticken abgespielt werden. Vielmehr muss die Betonung im Wort gleichzeitig mit dem Ticken zu hören sein. Da jedes Wort unterschiedlich lang ist und an einer anderen Stelle die Betonung enthält, wäre diese Aufgabe programmiertechnisch zu erledigen sehr aufwändig. Sind die Audiodateien jedoch schon vorgefertigt, so entfällt dieses Problem. Alle Audiodateien beginnen zeitgleich und enthalten Wort und Ticken im richtigen Zeitpunkt "gelayert". Sie können daher im Programm alle gleich behandelt und abgespielt werden.

Primitive Datentypen

In Java besteht anders als in rein objektorientierten Sprachen, die Möglichkeit primitive Datentypen wie `int`, `long` und `double` zu verwenden. Es stehen jedoch auch sogenannte Wrapper-Klassen wie Integer, `Long` und `Double` zur Verfügung. Diese sind jedoch "echte" Objekte und somit teurer als die primitiven Datentypen. Bei der Entwicklung wurde daher darauf geachtet, dass möglichst oft diese einfachen Strukturen verwendet wurden.

Das Interface `Updatable`

Alle Einstellungen und Informationen liegen zentral in der Klasse `PUtil` (Properties-Util). Jede Klasse die diese Informationen benötigt, könnte theoretisch hier nach den aktuellen Informationen sehen und sie über `get`- und `set`-Methoden beziehen. Dies wäre jedoch zum Teil aufwändig, da etwa die Zeitausgabe die Informationen über die eingestellte Sprache alle paar Millisekunden benötigt. Daher implementieren alle Controller das Interface `Updateble` welches die methode `update()` vorsieht. Werden die Nutzereinstellungen geändert, so wird von jedem Controller die Methode `update()` aufgerufen. Sie zieht die nötigen Einstellungen aus `PUtil` und speichert sie in lokalen Variablen. Auf diese kann dann schnell zugegriffen werden. Jedoch werden so Informationen zentral gehalten und der Programmierer wird dafür verantwortlich, dass ein Update auch wirklich ausgeführt wird. Sonst resultiert ein inkonsistenter Zustand (in den Controllern würden andere Einstellungen stehen als in PUtil eigentlich schon vorhanden sind). Da jedoch nur in einer Aktivity Nutzereinstellungen geändert werden können, kann der Update-Aufruf zentral implementiert werden.

High-Priority

Das Metronom soll in jedem Fall den Takt halten. Dieser "Takt" wird durch den `PlayController` realisiert, welcher die Klasse `Thread` ableitet. Dieser Thread ist eine Art Rückrad der Applikation. Er muss zwingend im Takt laufen. Daher wird seine Priorität aus den maximalen Wert gesetzt. Dies ist daher unproblematisch, da alle anderen Threads in ihrer Priorität unverändert bleiben. Lediglich dieser Thread soll bevorzugt werden. In welchen Fällen er um viel bevorzugt wird, ist dabei zweitrangig, es wird schlicht "das Möglichste" getan.

Garbage-Collector

Neben den vom Programmierer initiierten Threads, laufen in einem Java-Programm auch solche der VM ab. Hierzu zählt etwa der Garbage-Collector. Damit dieser nicht zu einem für das Programm "ungünstigen" Zeitpunkt angestoßen wird, kann er vom Programmierer selbst vorgeschlagen werden. Mit `System.gc()` kann er der VM einen Hinweis geben, jetzt den Garbage-Collector zu starten, da dies vielleicht ein passender Zeitpunkt ist. Dies wird auch für das Metronom genutzt, in dem z.B. beim drücken des Start-Buttons diese Methode aufgerufen wird. Somit ist die Chance geringer, dass er während des folgenden Tickens dazwischen funkt.

Es ist jedoch nicht sinnvoll den GC etwa nach jedem Ticken aufzurufen. Ruft man ihn nach jedem Ticken auf, so löscht er ca. 500 bis 600 Objekte in etwa 80 ms. Ruft man ihn hingegen nicht auf, so kann man im Log erkennen, dass er ca. 15000 Objekte in knapp 95 ms löscht. Daher entsteht bei dauerndem Aufrufen viel Over-Head. Es ist wohl performanter ihn seltener anlaufen zu lassen, da jedes Anlaufen auch für wenige Objekte viel Zeit kostet.

Vermeidung von neu erzeugten Objekten

Die Eclipse-Android-Tools bieten die Möglichkeit die Objekte die im Heap liegen bzw. erzeugt werden zu beobachten. Es lässt sich sehen durch welchen Methodenaufruf sie zustande kamen und welchem Thread sie angehören. So kann relativ einfach bestimmt werden wo "zuviel" oder unnötige Objekte erzeugt wurden. Es wurde (mir persönlich) deutlich, dass nicht nur ein explizites `new` ein neues Objekt erzeugen kann. Es wurde daher darauf geachtet, dass in keiner Schleife oder Methode neue Objekte angelegt werden, sondern lediglich Instanz-Variablen genutzt werden.

Zudem wurde durch das beobachten des Heaps deutlich wo im Programm „Schwachstellen“ liegen. So ist die Ausgabe der Zeit auf dem Bildschirm durch das häufige erzeugen neuer Strings relativ teuer. Es fallen viele Objekte an, die im späteren Verlauf vom GC gelöscht werden müssen. Dies ist wohl die so gesehen teuerste Methode.

"Umgekehrte" for-Schleifen

Eine "klassische" for-Schleife, erhöht ihren Zähler je Durchlauf um eins und prüft dann ob die Größe des Array schon erreicht ist. Dies ist unnötig, da die Größe des Arrays eigentlich von Beginn an fest steht, aber in jedem Durchlauf (eventuell sehr viele) abgefragt wird. Daher ist es performanter die Schleife um zudrehen:

```
for(int i = handlers.size()-1; i >= 0; i--)
      handlers.get(i).sendMessage(handlers.get(i).obtainMessage());
```

Hier wird nur zu Beginn der Schleife die Größe des Arrays einmal abgefragt. Die Zählvaribale wird dann solange um eins verringert bis sie kleiner als null wird. Dies erspart das ständige abfragen der Arraygröße (siehe Vorlesung Design Patterns).

Keine internen Getter- und Setter-Methoden

Getter- und Setter-Methoden sind in der objektorientierten Entwicklung eigentlich Standard. Sie dienen dem Zugriff auf Variablen und kapseln diese vor der Außenwelt. Durch sie kann etwa eine Domänenprüfung durchgeführt werden oder ein Wert in verschiedenen Formaten verfügbar gemacht werden.

Getter- und Setter-Methoden stellen jedoch auch eine weitere Abstraktionsebene bzw. Indirektion dar. Anstatt direkt auf die Variable zuzugreifen, wird der Umweg über eine dieser Methoden gegangen. Bei Anwendungen die möglichst performant laufen müssen, kann u.U. auf diesen Umweg bewusst verzichtet werden. Zwar entsteht so ein „unschöner“ Code, welcher auf das eigentlich zentrale Kapselungsprinzip verzichtet, aber es stellt sich in aller Regel ein Performance-Gewinn ein.

Daher kann und wurde zumindest auf interne Getter- und Setter-Methoden verzichtet. Da die Klasse selbst, ohnehin wissen muss wie ihre Attribute beschaffen sind, ist dies vertretbar.

7. Tests

Bei der Entwicklung kam es zu einigen Problemen, insbesondere in Verbindung mit der Audioverarbeitung der Android-Plattform. Dabei kamen verschiedene Fragen auf, die im Folgenden beantwortet werden sollen.

Welche Objekte werden erzeugt (Stichpunkt Heap-Dump)?

Die Andoid-Eclipse-Tools bieten eine komfortable Möglichkeit die Objekte im Heap des Handys zu beobachten. In der Ansicht "Allocation Tracker" werden alle erstellten Objekte, samt zugehörigem Thread und Methoden-Aufruf dargestellt. Über diesen Mechanismus wurden folgende "ungünstigen Stellen" im Code verbessert:

- Die Klasse `TimeController` schreibt alle 10 ms (ursprünglich sogar 10 ms) eine Zeitangabe auf den Bildschirm. Hierfür ermittelt sie die Zeit seit dem drücken des Start-Buttons und stellt diese dar:

```
private String getString() {

          diff = System.currentTimeMillis() - start;
          sekunden = (diff / 1000);
          stunden = (long)Math.floor(sekunden / 3600);
          sekunden -= (stunden*3600);
          minuten = (long)Math.floor(sekunden / 60);
          sekunden -= (minuten*60);
          millis = (diff % 1000)/10;

          if(minuten < 10)      minStr = "0" + minuten;
          else                  minStr = "" + minuten;

          if(sekunden < 10)     secStr = "0" + sekunden;
          else                  secStr = "" + sekunden;

          if(millis < 10)       milStr = "0" + millis;
          else                  milStr = "" + millis;

          return time + ": " + minStr + ":" + secStr + ":" + milStr;
   }
```

Ursprünglich wurden dabei bei jedem Methodenaufruf die Objekte neu erstellt (also `diff`, `sekunden` etc. sowie die Strings `minStr`, `secStr` und `milStr`). Dies wurde verbessert, dahingehend, dass die Objekte nur einmal als Instanz-Variablen gehalten

werden. Jedoch werden auch durch die Konvertierung der Zahlen in Strings viele Objekte erzeugt (z.B. `char[]` und `StringBuilder`). Daher werden erheblich bessere Ergebnisse erzielt, wenn die gesamte Methode weggelassen wird bzw. deren Aufruf seltener stattfindet.

- Die Klasse `PlayController` fragt in der Thread-Schleife permanent die Zeit bis zum nächsten Ticken ab:

```
this.sleep( takt.getTimebetweenHitsLong() );
```

Dies führt über die Klasse `Takt` im Modell `BPM` zu folgendem Aufruf:

```
double tmp = bpm * 10;
return 600000/(long) tmp;
```

Dieser Aufruf ist dahingehend ungünstig, dass permanent ein neues Long-Objekt (`tmp.longValue()`) sowie ein neues Double-Objekt (`double tmp`) erzeugt wird. Der gesamte Abfrage Vorgang wurde somit so abgeändert, dass die Klasse `PlayController` nur noch einmal bei Geschwindigkeitsänderungen diesen Wert abfragt und selbst speichert.

```
public void updateTimebetweenHitsLong(){
this.TimebetweenHitsLong = takt.getTimebetweenHitsLong();
}
```

Durch diese Maßnahmen läuft der Garbage Collector nun nur noch äußerst selten an. Ist das Tempo konstant auf 120 BPM eingestellt und wird der GC initial beim drücken des Play-Buttons gestartet, so startet er erst 1 min 5 Sekunden später erneut, er löscht dann ca. 12000 Objekt. Daraufhin startet er einige Sekunden später wieder, löscht aber nur knapp 30 Objekte. Danach startet er >2 min nicht mehr.

Vor der Verbesserung, ist er ca. alle 15 Sekunden gestartet und hat jeweils 15000 Objekte gelöscht.

Als Fazit kann man festhalten, dass Objekte oft unbewusste erzeugt werden, sie entstehen nicht nur an Stellen an den denen explizit `new` aufgerufen wird. Es ist äußerst sinnvoll den Heap und die Herkunft der Objekte im Auge zu behalten.

Ist es sinnvoll den GC regelmäßig selbst zu starten?

Läuft der Garbage-Collector von selbst an, so löscht er ca. 15000 Objekte in etwa 120 ms (auf der echten Device). Wird er jedoch von Hand etwa nach jedem Klicken angeworfen, so löscht er ca. 600 Objekte in 80 ms. Daher ist dieses Vorgehen wenig effizient, da er jedes mal eine lange Anlaufzeit braucht.

Ist der Debugging-Modus auf dem Handy (der echten Device) aktiv?

Der Debugging-Modus lässt sich auf der "echten Device" wahlweise ein- oder ausschalten. Hierzu enthält die `AndroidManifest.xml` ein Attribut `Debuggable` und das Android-Handy die Einstellung `MENU > Applications > Development > USB debugging` mit welcher das Debugging ein- bzw. ausgeschaltet werden kann. Ist es aktiv, so kann man die Log-Ausgaben auf der Eclipse-Console sehen sofern das Handy über USB verbunden ist, andernfalls nicht.

Ist die Log-Ausgabe auf dem Handy ausgeschaltet, so ist es gefühlt schneller. Leider lässt sich dies schwer messen, die Log-Ausgabe ist schließlich abgeschaltet.

Ist die Länge der Audio-Dateien von Bedeutung?

Das Metronom soll in (unter Umständen) sehr kurzen Abständen Audiodateien abspielen. Bei hohen Geschwindigkeiten, kann dabei die Länge eines Audiofiles, welches nur grob geschnitten ist, länger sein als die Pause bis zum nächsten Abgespielten. Ein Beispiel: Ist das Tempo auf 120 BPM eingestellt, so sind 120 Schläge pro Minute zu hören. Somit pro Sekunde 2, was jedem Audiofile eine Zeitspanne von 500 ms einräumt. Dies ist meistens kein Problem. Ist die Geschwindigkeit jedoch auf 180 BPM eingestellt, so hat jedes Audiofile nur noch ca. 333 ms Zeit abgespielt zu werden. Dies wird zum Problem wenn die Dateien selbst länger sind.

In der ersten Version wurden Dateien einer einheitlichen Länge von ca. 370 ms verwendet. Dies ermöglichte ein problemloses Abspielen bis etwa 160 BPM bzw. knapp darüber. Wurde die Geschwindigkeit jedoch so hoch eingestellt, dass die Dateien länger waren als der gegebene Zeitraum in dem sie gespielt werden sollten, kam das Metronom hörbar aus dem Takt. Es wurde die Warnung `AudioFlinger write blocked for 100 ms` auf dem Android-Log geworfen. Wurden die Audiofiles wiederum kürzer geschnitten, so wurde bei

der selben Geschwindigkeit keine Warnung geworfen. Daher ist es von Bedeutung welche Länge die Dateien besitzen.

Bei kurzen Audiodateien kam es jedoch zu einem Problem auf dem Emulator. Waren die Dateien sehr kurz geschnitten (etwa < 220 ms), so spielte der Emulator den ersten Ton nicht ab, bzw. erst zusammen mit dem zweiten. Dieses Problem tauchte jedoch nicht auf dem Android-Handy selbst auf, weshalb es als Bug des Emulators einfach ignoriert wurde.

Wann wacht ein Thread tatsächlich auf?

Da der subjektive Eindruck über die Pünktlichkeit der Threads täuschen kann, soll dieses Verhalten getestet werden. Hierzu wird der Zeitpunkt vor dem Aufruf der `Sleep` Funktion festgehalten (`oldDate`) und wiederum mit der aktuellen Zeit verglichen, sobald der Thread wieder aufwacht:

```
/*if(oldDate == null)
    oldDate = new Date();

newDate = new Date();
System.out.println("Vorgabe: " + takt.getTimebetweenHitsLong());
System.out.println("Ist:     " + (newDate.getTime()-oldDate.getTime()));
oldDate = new Date();*/
```

Im Log lässt sich so folgendes Ergebnis ermitteln: Liegt die Soll-Ausgabe bei 500 ms (also 120 BPM), so schwankt der Thread ohne dass ein Sound abgespielt wird um ca. 501 ms (+/- 2). Dieses Ergebnis ist gut. Wird ein Sound abgespielt, so schwankt der Thread für gewöhnlich zwischen 501 ms und 525 ms. Setzt jedoch der Garbage-Collector zu einem ungünstigen Zeitpunkt ein, so kommt der Thread erst nach bis zu 630 ms zurück.

8. Probleme bei der Entwicklung

Nepomuk war eines meiner ersten und für mich persönlich größeren Softwareprojekte, in welches ich über ein gesamtes Semester hinweg Zeit investiert habe. Daher bin ich anfänglich recht unbefangen in die Entwicklung gestartet und dachte mich voll auf das Android-SDK verlassen zu können. Ein Metronom ist schließlich keine zu komplexe Aufgabe und wird mit rund 1500 Klassen wohl schon zu machen sein - aber was wenn nicht?

Das Android-SDK bietet mit einer Klassenbibliothek von mehr als 1500 Klassen, Lösungswege für eine Großzahl von Problemen. Jedoch ist es schwer von diesen vorgefertigten Lösungswegen abzuweichen und eigene zu entwickeln. Denn diese Lösungen sind weit mehr als nur eine Art "Best Practice", sie sind "so vorgesehen".

Beispiel Einstellungen

Auf der Android-Plattform gib es einen vorgesehen Weg mit Nutzereinstellungen umzugehen und diese persistent zu speichern. Dieser geht über den sogenannten Preferences-Mechanismus und die interne SQLite Datenbank. Möchte man als Entwickler von diesem Weg abweichen (aus welchem Grund auch immer), so ist dies nicht immer möglich. Möchte man seine Einstellungen etwa in einer eigenen Datenbank halten, in einer Propertie- oder XML-Datei, oder aber in einem eigenen Format, so ist dies etwa kaum umsetzbar.

Beispiel Klasse R

Das Einbinden, Lesen und Schreiben von Dateien ist auf der Android-Plattform ebenfalls nicht uneingeschränkt möglich. So können etwa keine Dateien im Projektordner geschrieben werden, sondern lediglich auf einer SD-Karte. Außerdem werden einzubindende Dateien grundsätzlich über die automatisch generierte Klasse R referenziert. Dies macht es etwa unmöglich Dateien über ihren Namen aus zusammengesetzten Strings zu laden. Dies hatte bei Nepomuk etwa eine sehr lange fest codierte Update-Methode zur Folge.

Beispiel Real-Time-Audio

Die Android-Plattform ist nicht für Real-Time vorgesehen. Der hauseigene GC hält bei jedem Durchlauf die VM komplett an und gibt sie erst anschließend wieder frei. Daher ist auch die Aufgabe Audiodateien in Real-Time zu verarbeiten, so nicht vorgesehen. Diese Art der Anwendung wäre sogar in einer normalen Java-Umgebung schon kritisch. Bei dieser Art der Anwendung, besteht außerdem ein Unterschied darin, ob man die Applikation auf dem eigentlichen Gerät oder einem Emulator ausführt, da dieser Emulator, aus Eclipse heraus wiederum auf einer anderen Java-VM läuft.

Am Ende der Entwicklung überwiegt der Eindruck, dass auf der Android-Plattform zwar schon relativ viel möglich ist, aber längst nicht alles - und das was möglich ist, ist relativ streng vorgegeben. Die Klassenbibliothek ist nicht beliebig durch andere JAR-Archive erweiterbar, da Java-Klassen speziell für die Android-VM kompiliert sein müssen. Daher hat der Entwickler weit weniger Freiheiten als gewohnt. So halte ich persönlich die Android-Plattform mehr für ein Framework, in dessen Rahmen man bestimmte Java-Programme entwickeln kann, als für eine echte Java-Umgebung, da SDK, Möglichkeiten und Art der Programmierung teils völlig anders sind.

9. Vertrieb

Android Market

Ähnlich wie bei dem von Apple bekannten AppStore, gibt es auch auf der Android-Plattform einen einfachen und kostengünstigen Vertriebsweg für Applikation, den Android Market. Dieser ist ein virtueller Marktplatz welcher über eine Software, die standardmäßig auf den Android-Handys vorinstalliert ist, zugänglich ist. Über ihnen lassen sich sowohl kostenlose als auch kostenpflichtige Applikation beziehen und direkt herunterladen. Die einzige Voraussetzung hierfür ist ein Google-Konto.

Derzeit befinden sich rund 70000 Applikation in dem Angebot, wobei die Zahl um monatlich ca. 14000 Anwendungen steigt. Rund zwei Drittel der Angebote sind dabei kostenfrei.

Für den Entwickler kostet die Anmeldung bei Google einmalig ca. 25€, danach ist die Mitgliedschaft kostenfrei. Jedoch wird für kostenpflichtige Angebote eine Transaktionsgebühr von 30% verlangt.

Eigener Vertrieb

Android-Applikationen können jedoch auch ohne den Android Market verbreitet werden. Es ist möglich sie als herkömmliche Downloads auf einer Webseite anzubieten. Sie müssen dann lediglich vom PC zum Handy übertragen werden. Diese Art des Vertriebs ist aber sehr selten. Gebräuchlicher ist der Android Market, der sich als Verkaufs- und Donwloadplattform hierfür durchgesetzt hat.

10. Persönliches Fazit und Ausblicke

Für mich persönlich war das Projekt insgesamt ein Erfolg. Zwar bin ich mit der entwickelten Applikation nicht vollends zufrieden, da eine garantierte Real-Time-Verarbeitung fehlt und bedingt durch die Android-Plattform nicht alle Wünsche von Herrn Oldenkamp umgesetzt werden konnten, aber es ist definitiv mehr als nur ein "erster Prototyp" entstanden.

Positiv an diesem Projekt fand ich die verschiedenen und abwechslungsreichen Teilgebiete. Angefangen von der Einarbeitung in eine mir neue Umgebung, über die Entwicklung von GUI und Audioverarbeitung, Audioaufnahmen, Dokumentation, Design von Postern und Präsentation hinweg. Dies vermittelte mir den Eindruck an einer abgeschlossenen Sache arbeiten zu können, was mir sehr gefallen hat. Außerdem hat es mir gezeigt, dass zu einem Projekt mehr gehört als reines "coden".

Negativ, bzw. etwas schade fand ich an dem Projekt, am Ende nicht ganz zufrieden sein zu können. Ich hätte die Anwendung gerne fehlerlos und komplett umgesetzt, was aufgrund der Umgebung leider nicht möglich war. Dies ist für mich ein kleiner Wermutstropfen. Außerdem hätte ich gerne in einem Team gearbeitet, was durch die Unklarheiten zu Semesterbeginn leider nicht möglich wurde.

Zwar ist das Metronom an sich in diesem Semester „fertig“ geworden, jedoch könnten noch einige Kleinigkeiten und Features verbessert und ergänzt werden. Dies ist insbesondere mit der Weiterentwicklung der Android-Plattform und der Hardware interessant. So könnte die Tapper-Funktion z.B. dahingehen erweitert werden, dass nicht nur ein Tempo, sondern ein kompletter Rhythmus eingetippt werden könnte. Dies wäre sicherlich interessant, bedingt durch die momentanen Probleme viele Audio-Dateien kurz hintereinander abzuspielen, aber nicht zufriedenstellend umsetzbar.

11. Quellen und Referenzen

Zu Kapitel 3. Die Android-Plattform

- **Lars Vogel: Android Development Tutorial**
  http://www.vogella.de/articles/Android/article.html
- **Android Dev Guide: Application Fundamentals**
  http://developer.android.com/guide/topics/fundamentals.html

Zu Kapitel 5. Implementierung

- **Droid Draw: GUI-Designer für Android**
  http://www.droiddraw.org
- **Tango Desktop Proeject: Open Source Icons**
  http://tango.freedesktop.org

Zu Kapitel 6. Design for Performance

- **Android Dev Guide: Design for Performance**
  http://developer.android.com/guide/practices/design/performance.html

Zu Kapitel 7. Tests

- **Android Dev Guide: Developing on a Device**
  http://developer.android.com/guide/developing/device.html
- **Android Dev Guide: Using the Dalvik Debug Monitor**
  http://developer.android.com/guide/developing/tools/ddms.html

Zu Kapitel 9. Vertrieb

- **Android Dev Guide: Publishing Your Applications**
  http://developer.android.com/guide/publishing/publishing.html
- **Wikipedia: Android Market**
  http://de.wikipedia.org/wiki/Android_Market