Fachhochschule Köln
Cologne University of Applied Sciences
Campus Gummersbach
Fakultät für Informatik und Ingenieurwissenschaften

Verbundstudiengang Wirtschaftsinformatik

Masterthesis

Konzepte der Nebenläufigkeit unter Android

Prüfer:	Prof. Dr. Erich Ehses
Zeitprüfer:	Prof. Dr. Frank Victor
vorgelegt am:	28. Oktober 2015
von cand.:	Stephan Wagner
aus:	Overath
Telefon-Nr.:	+49-176-80007570
Matrikel-Nr.:	1106011828
E-Mail-Adresse:	stephan.wagner.mi738@gmail.com

Zusammenfassung

Diese Thesis behandelt das Thema „Konzepte der Nebenläufigkeit unter Android“. Darin wird zunächst als Einführung in die Thematik, die Nebenläufigkeit allgemein mit ihren unterschiedlichen Ausprägungen (Prozess-/Threadebene) erläutert und die Risiken die mit Nebenläufigkeit einhergeht skizziert. Dabei ist ein besonders wichtiger Punkt der Botschaftenaustausch, der mittels unterschiedlicher Techniken realisierbar ist. Auch hier birgt jede Technologie ihre individuellen Vor- und Nachteile. Die Nebenläufigkeit unter Android unterliegt einigen Besonderheiten, die sich teilweise aus den Restriktionen des Betriebssystems ergeben, aber auch aus den Anforderungen für die Android Applikationsentwicklung, die der StyleGuide vorgibt. So ist die grundsätzliche Anforderung, dass Applikationen ansprechbar bleiben. Eine Applikation, insbesondere eine Gui Applikation sollte somit beim Start von Operationen weder blockieren oder nicht mehr auf Benutzereingaben reagieren. Technisch bedeutet dies, dass zeitaufwändige Operationen nie auf dem Main Thread der Applikation stattfinden dürfen. Stattdessen müssen derartige Operationen in Hintergrundthreads ausgelagert werden. Im Kapitel 2 werden hierzu drei unterschiedliche Konzepte an Hand konkreten Beispielimplementierungen vorgestellt und in der jeweiligen Funktionsweise analysiert.

- Java Concurrency nach Java SE
- Android Concurrency aus dem Android SDK
- RXJava Framework

Die aus der technischen Analyse gewonnenen Erkenntnisse werden in Kapitel 3 genutzt um Chancen und Risiken der einzelnen Konzepte zu diskutieren. Um die Ergebnisse aus dem Diskurs für zukünftige Entscheidungsfindungen zu Rate ziehen zu können fließen die aus der Detailanalyse gewonnenen Ergebnisse in eine Szenarien basierte Analyse ein, um daraus Anhaltspunkte für den sinnvollen Einsatz der Konzepte abzuleiten.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

Mobile Endgeräte begleiten immer mehr Menschen in ihrem Alltag. Damit einher geht die intensive Nutzung von sog. Apps., womit Applikationen auf den mobilen Endgeräten bezeichnet werden. Mit der zunehmenden Leistungsfähigkeit der Geräte werden auch immer komplexere Applikationen realisierbar. Wurde zu den Anfängen der Applikationsentwicklung für mobile Endgeräte lediglich einfache Funktionalität in Applikationen integriert, werden heute mit unter teilweise sehr rechenintensive und komplexe Funktionalitäten entwickelt. Eine optimale Konzeption der Aufgabenverarbeitung innerhalb der Applikation kann dabei einen entscheidenden Faktor für die Performance und damit auch die Akzeptanz beim Nutzer darstellen. Damit gewinnt die Nebenläufigkeit auch in der Applikationsentwicklung für mobile Endgeräte an Wichtigkeit. Nebenläufigkeit oder auch Parallelverarbeitung bezeichnet in der Informatik die Eigenschaft eines Programms oder eines Systems verschiedene Aufgaben zeitgleich, also parallel zu bearbeiten. Die jeweilige Verarbeitung kann dabei in sich abgeschlossen sein, d.h. die zu verarbeitenden Aufgaben sind voneinander unabhängig, oder die Verarbeitung hängt von den Ergebnissen aus anderen Aufgaben ab. Je nach Art und Weise der Parallelverarbeitung sind verschiedene Problematiken und Risiken zu beachten. Für Nebenläufigkeit unter dem Android Betriebssystem sind zusätzliche Besonderheiten zu beachten. Dieses Betriebssystem ist auf mobile Endgeräte zugeschnitten und hat diesbezüglich spezielle Anforderungen an Applikationen die darauf laufen sollen. Die Firma Google als Hersteller vom Betriebssystem Android legt hierbei großen Wert auf die Einhaltung eines StyleGuides der die Benutzbarkeit applikationsübergreifend in einem einheitlichen Standard definiert. Darin wird die grundlegende Anforderung nach der kontinuierlichen Ansprechbarkeit von Applikationen gefordert. Die Frage ist wie kann den Anforderungen an Android Applikationen mittels unterschiedlicher Konzepte der Nebenläufigkeit begegnet werden, sodass das von Google geforderte Ziel der Ansprechbarkeit erreicht werden kann.

Welche Problemstellungen, Restriktionen oder Risiken gehen mit der Verwendung bestimmter Konzepte einher und wie praktikabel sind diese für den konkreten Praxiseinsatz?

1.2. Zielsetzung und Vorgehen

In dieser Arbeit soll untersucht werden, wie konkurrierende Parallelverarbeitung in mobilen Anwendungen realisiert werden kann. Dabei besteht das Ziel, die Entwicklung von Nebenläufigkeit durch Verwendung unterschiedlicher Techniken zu vereinfachen und ggf. auf einem höheren Abstraktionsniveau ab zu bilden. Zunächst gilt es in einer kurzen Einführung in die Thematik, die grundsätzlichen Definitionen kurz zu erläutern und auf Besonderheiten der Parallelverarbeitung unter Android einzugehen. Weiter wird ein Überblick über eine Auswahl von unterschiedlichen Konzepten der Nebenläufigkeit unter Android erarbeitet. Diese werden mittels einfacher Beispiele vorgestellt und analysiert. Den Abschluss bildet ein kritischer Diskurs, um in Abhängigkeit vom Einsatzkontext eine Differenzierte Sicht auf die Anwendung der einzelnen Konzepte zu erhalten. Die Ergebnisse des Diskurses werden in einer szenarienbasierten Analyse aufgegriffen um diese greifbarer zu machen.

1.3. Begriffsdefinition und Grundlagen der Nebenläufigkeit

Um sich den Konzepten der Nebenläufigkeit anzunähern, werden zunächst einige Begriffsdefinitionen benötigt. Die Nebenläufigkeit meint dabei konkret die parallele Verarbeitung von Aufgaben. Hierzu wird eine Aufgabe in Unteraufgaben aufgeteilt, um diese weitestgehend von einander unabhängig abzuarbeiten. Die Definition wie diese Verarbeitung ablaufen soll, ist in einem Programm hinterlegt. Die Ausführung von Programmen wird von Prozessen und Threads geregelt. Diese werden im folgenden Abschnitt definiert und ein tieferes Verständnis von der Parallelverarbeitung auf Betriebssystemebene erarbeitet.

1.4. Prozesse und Threads

Die genauen Eigenschaften von Prozessen und Threads sind abhängig vom Betriebssystem auf dem sie laufen. Da in dieser Arbeit der Fokus auf Nebenläufigkeit unter Android liegt, beziehen sich die folgenden Erläuterungen zu Prozessen und Threads auf das allgemeine Unix/Linux Betriebssystem auf dem Android basiert.

1.4.1. Prozess

Wird eine Anwendung gestartet, so erzeugt das Betriebssystem zunächst einen Prozess, der den Adressraum für sämtliche Programmdaten und Komponenten reserviert. Für Prozesse kann folgende Definition getroffen werden. Sie gilt betriebssystemübergreifend:

> Ein Prozess stellt ein Programm in Ausführung dar und ist für die Kontrolle(Sicherung) der damit verbundenen Betriebsmittel verantwortlich.

Die Prozesse sind (in der Regel) an einen Benutzer gebunden, welcher wiederum über bestimme Rechte u.a. im Dateisystem verfügt. Dabei sind für Linux Betriebssysteme folgende Prinzipien zu beachten:

- Hierarschische Prinzip
- Sandbox Prinzip

Das hierarchische Prinzip schreibt die Abhängigkeit von Prozessen gegenüber ihren Erzeugern vor. Mit Ausnahme des Root Prozesses des Betriebssystems, werden alle Anwendungen durch einen Vater Prozess erzeugt. Die damit verbundene Vater-Kind Abhängigkeit bildet eine Baumstruktur, in der jeder Prozess seinen erzeugenden Prozess kennt. Ein Prozess kann nur aus anderen Prozessen heraus erzeugt werden. Stirbt ein Prozess, so werden die Kind Prozesse in der Regel vom Root Prozess des Betriebsystems adoptiert.
Das Sandbox Prinzip ist ein Sicherheitskonzept aus dem Kern eines Linux/Unix Betriebssystems. Darin wird sichergestellt, dass jede Anwendung nur die eigenen Daten sehen darf. So wird bei der Installation

für jede Anwendung ein eigener Betriebssystem- User erzeugt, der über spezielle Rechte zu Prozessen und Dateien verfügt. Damit wird zum Ausführungszeitpunkt verhindert, dass Programmdaten für andere Programme sichtbar werden. Die Sicht jedes Prozesses einer Anwendung ist begrenzt auf die Ressourcen die dem jeweiligen Betriebssystem User zugeordnet sind.

1.4.2. Thread

Die Begriffe Prozesse und Threads dürfen nicht synonym verwendet werden. So kann ein Thread wie folgend Definiert werden:

> Ein Thread stellt einen Ausführungsfaden eines Programmes dar.

Dieser besteht aus aus einem aktuellen Befehlszeiger, einem eigenen Stack und dem Inhalt der Prozessorregister. Zum Start einer Anwendung wird der sog. Main Thread erzeugt. Aus diesem lassen sich beliebig weitere Threads erzeugen. Dabei besteht keine hierarchische Bindung wie bei der Vater-Sohn Prozesshierarchie. Innerhalb eines Prozesses erzeugte Threads erhalten Zugriff auf den hier reservierten Speicher des Prozesses. Alle in einem Prozess erzeugte Threads sind von diesem abhängig. Wird demnach ein Prozess terminiert, so werden auch alle darin erzeugte Threads terminiert.

1.5. Botschaftenaustausch und Kommunikation

Der Botschaftenaustausch zwischen Prozessen unterscheidet sich vom Botschaftenaustausch zwischen Threads. Während bei der Inter Prozess Kommunikation maßgeblich das Betriebssystem am Austausch von Nachrichten zwischen Prozessen beteiligt ist, können bei der Inter Thread Kommunikation unterschiedliche Techniken unabhängig vom Betriebssystem angewandt werden. Für die Kommunikation zwischen Threads eignen sich z.B. Dateien aber auch sogenannte MessageQueues mit denen

Produzenten und Konsumenten Konstrukte erzeugt werden können. Die Inter Thread Kommunikation bleibt begrenzt auf die Threads innerhalb einer Anwendung. Die Inter Prozess Kommunikation dagegen, definiert die Kommunikation über Programm- und Systemgrenzen hinaus. Innerhalb eines Betriebssystems wird in der Regel der Speicherbereich jedes Prozesses in sich gekapselt und vor anderen Prozessen verborgen (s.o. Sandbox-Prinzip). Daher werden Mechanismen seitens des Betriebssystems benötigt (Botschaftenaustausch über Socket, etc..) um die Kommunikation zu gewährleisten. Diese Mechanismen sind aufwendig und eignen sich dadurch weniger für eine effiziente Parallelverarbeitung. Daher konzentriert sich diese Arbeit auf die Kommunikation auf Thread Ebene, und die Inter Prozess Kommunikation wird nicht weiter thematisiert. Die folgenden Abschnitte geben einen Einblick auf gängige Techniken zur Realisierung von Inter Thread Kommunikation.

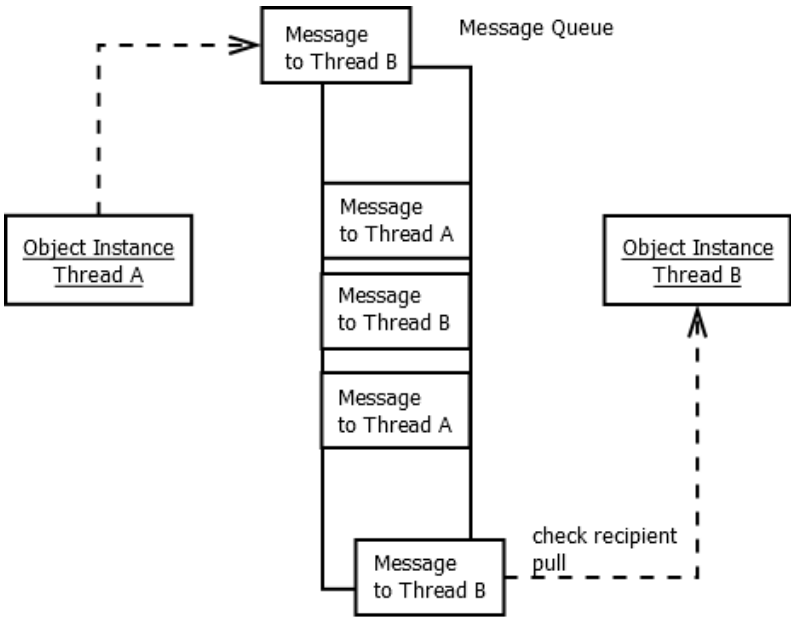
1.5.1. Geteilte Datei/ Speicher

Einer der einfachsten technischen Mittel für den Daten- /Botschaftenaustauch ist die Nutzung einer gemeinsamen Datei im Dateisystem des Betriebssystems. Dadurch, dass das Betriebssystem den exklusiven Zugriff auf Dateien gewährleisten kann, ist es möglich ohne großen Aufwand eine synchronisierte Kommunikation zu realisieren. Etwas komplexer ist es für die Kommunikation einen Speicherbereich zu allokieren und die Referenz darauf den jeweiligen Kommunikationspartern für den Datenaustausch zur Verfügung zu stellen. In diesem Fall muss der exklusive Ausschluss selbst realisiert werden, falls er gewünscht ist. Hierzu dienen einfache Primitive aus dem java.lang.concurrency Packet.

1.5.2. MessageQueues

Die folgende Abbildung gibt einen schematischen Überblick über die Nutzung einer MessageQueue für die Kommunikation zwischen Objektinstanzen aus unterschiedlichen Threads. Beide Objekt Instanzen müssen eine Referenz auf das MessageQueue Objekt halten um Nachrichten (z.B. Message to Thread B) in diese Queue einzustellen oder herauszuholen. Das

Konzept des nachrichtengetriebenen Datenaustausches hat den Vorteil, dass jede Nachricht eine atomare (in sich geschlossene) Einheit darstellt. Dadurch lassen sich einzelne Arbeitsaufträge unterscheiden. Je nach Implementierung der MessageQueue sind auch keine weiteren Synchronisationen mehr nötig.



Die Kommunikation mittels einer MessageQueue kann in zwei Formen realisiert werden:

- Unidirektional
- Bidirektional

Bei der unidirektionalen Kommunikationsform darf ein Kommunikationspartner nur entweder Nachrichten aus der Queue entnehmen oder hinein geben. In unserem Beispiel dürfte demnach nur die Objekt Instanz des Thread A Nachrichten in die Queue geben und die Objekt Instanz des Thread B darf lediglich aus dieser lesen.
Bei der bidirektionalen Kommunikationsform dürfen beide Kommunikationspartner je Nachrichten in die MessageQueue einstellen wie auch herausnehmen.

Die letzten beiden Abschnitte haben einen Überblick über Technologieen gegeben, mit denen der Austausch von Informationen innerhalb einer nebenläufigen Verarbeitung über Thread Grenzen hinaus realisiert werden kann. Dabei stellen MessageQueues eine Abstraktionsebene zur Kommunikation über fest definierten Speicher dar. Die Entwicklung von Nebenläufigkeit kann jedoch auch zu schwerwiegenden Problemen führen.

1.6. Risiken von Nebenläufigkeit

Der Botschaftenaustausch zwischen Threads, sowie deren Synchronisation kann zu schwerwiegenden Problemen im Zuge der Prallelverarbeitung führen. Folgende Szenarien sind eher allgemein gehalten, jedoch gilt es, besonders in dem Kapitel 2 zu den konkreten Implementierungen von Nebenläufigkeit, diese Problematiken zu beachten. In Kapitel 3 werden die in Kapitel 2 zu diskutierenden Beispielimplementierungen u.a. an Hand der hier aufgeführten Risikoszenarien und dem damit verbundenen Fehlerpotential bewertet.

1.6.1. Philosophenproblem

Ein zentrales Problem der theoretischen Informatik ist das Philosophenproblem das erstmals beschrieben wurde durch Edsger W. Dijkstra. Darin wird ein Szenario berschrieben, in dem eine bestimmte Anzahl von Philosophen auf begrenzte Ressourcen zugreifen und bei gleichzeitigem Zugriff sich gegenseitig blockieren können.

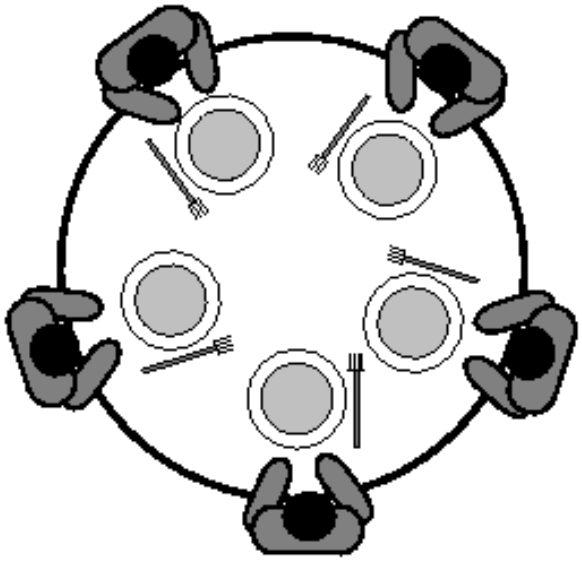


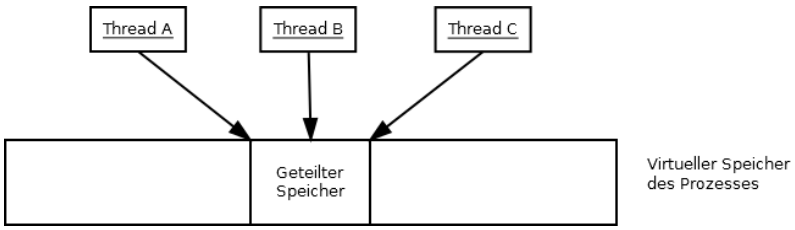
Abbildung 1.: Philosophenproblem [**?**,]

Die Abbildung zeigt wie eine Gruppe von Philosophen an einem Runden Tisch sitzen, vor ihnen etwas zu Essen. Um zu essen, benötigen nach diesem theoretischen Aufbau die Philosophen die rechte und die linke Gabel neben dem jeweiligen Teller. Dabei versuchen die Philosophen zu nächst die Gabel zu ihrer Linken zu nehmen. Ist die Gabel frei, so behälten sie diese in der Hand bis auch die Gabel auf der rechten aufgenommen werden kann. Kann ein Philosoph eine Gabel zur Zeit nicht nehmen, da sie in Verwendung ist, verweilt er denkend bis die benötigte Gabel frei ist. Versuchen jedoch alle Philosophen gleichzeitig die Gabeln aufzunehmen, so besteht die Gefahr einer Verklemmung (engl. Deadlock). Der Ablauf stockt und die Philosophen verharren denkend bis sie verhungern. In Bezug auf die Kommunikation über geteilten Speicher oder Dateien kann dieses Problem auftreten wenn parallele Zugriffe auf exklusive Ressourcen nicht sauber synchronisiert werden.

1.6.2. Race Conditions

Ein weiteres Problem bei der Parallelverarbeitung tritt bei geteilten Speicher bzw. Daten auf. Folgende Abbildung illustriert das Szenario, dass drei Threads auf einen gemeinsamen Speicherbereich zugreifen. Die Threads eins bis drei greifen konkurrierend lesend, wie schreibend auf den

Speicherbereich zu und tauschen darüber Informationen untereinander aus. Der Zugriff erfolgt nach dem Prinzip „Wer zuerst kommt mahlt zuerst“(= Race Condition).



Ist der Zugriff der Threads auf den Speicher nicht synchronisiert, so kommt es zu dem Shared Memory Effekt, (geteilter Speicher Effekt) nach dem eine Datenstruktur die, die Grundlage von Berechnungen darstellt, durch einen anderen Thread verändert wird, ohne dass die Änderung dem ersten Thread bekannt gemacht wird. Da solche Probleme von der jeweils in diesem Moment vorliegenden Prozessauslastung im System abhängen (tatsächlich gleichzeitig laufende Threads im Multi Core System), sind derartige Effekte schwer reproduzierbar und somit auch die Ursachen schwer zu finden.

1.6.3. Resistive Touchbildschirm Technologie

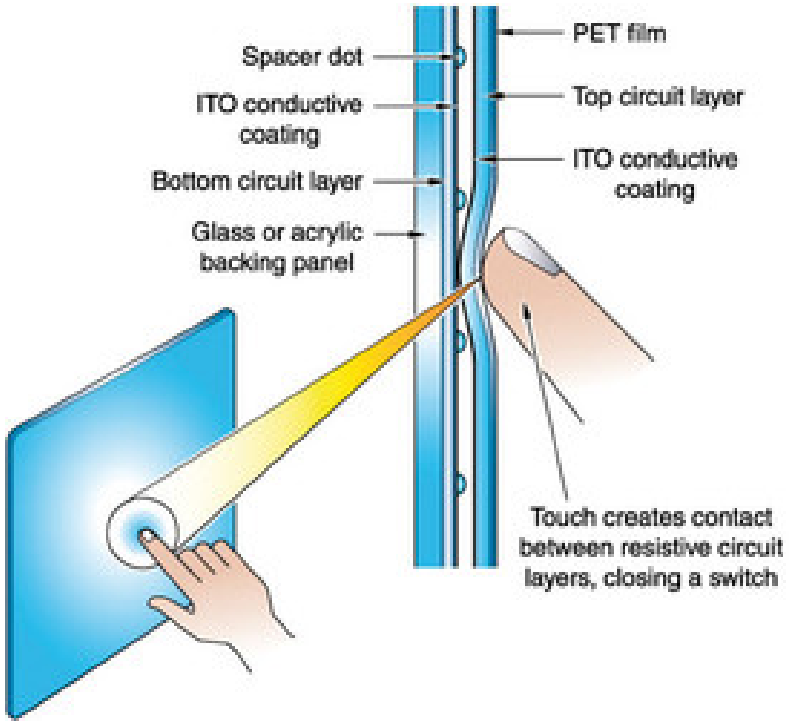


Abbildung 2.: Resistive Technologie [**?**,]

In der resistiven Touchscreen-Technologie wird ein Touchbildschirm in Deckschicht und Grundschicht unterteilt. An den Innenkanten beider Schichten ist ein leitender Film angebracht, der je mit unterschiedlicher Spannung geladen ist. Wie in der Abbildung gezeigt, schließt eine Berührung den Stromkreis und mit Hilfe der an den Bildschirmkanten angebrachten Sensoren die genauen Koordinaten des Berührungspunktes ermittelt. Prinzipiell ist bei dieser Technologie die Erkennung von Mehrfachberührungen möglich, jedoch hängt dies von den verbauten Sensoren ab.

1.6.4. Surface Accustic Wave (SAW)

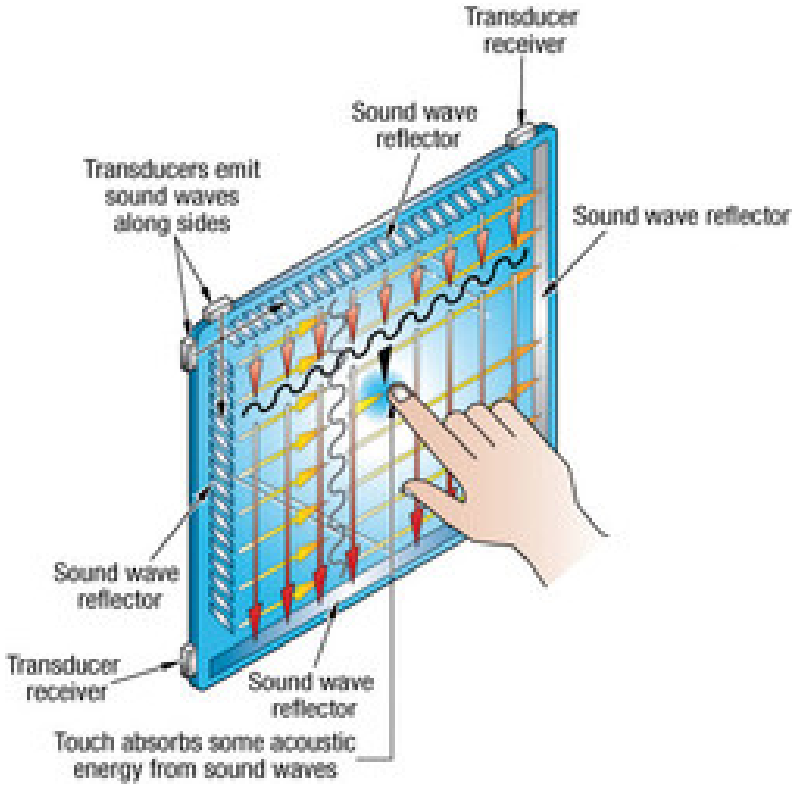


Abbildung 3.: Surface Accustic Wave Technologie [**?**,]

Das technische Prinzip dieser Technologie ist vergleichbar mit dem des Infrarot-Gitters. Schallwellen werden an bestimmten Punkten des Touchbildschirms emittiert und an zwei Kanten (x;y) so reflektiert, dass ein Raster durch die Schallwellen im Berührungsbereich aufgespannt wird. An den zwei übrigen Kanten leiten Reflektoren die einzelnen Schallwellen zu einem Sensor. Die Berührung hat eine Dämpfung der Schallwellen (in x- und y-Richtung) zur Folge, die zum Zeitpunkt der Berührung den Berührungspunkt passieren. Da die Schallwellen zeitverzögert erzeugt werden, also nur immer eine Schallwelle in eine X-Richtung und nur eine in Y-Richtung, ist somit der Berührungspunkt genau zu ermitteln. Dies bedeutet jedoch auch, dass zum selben Zeitpunkt nur ein Berührungspunkt ermittelbar ist, womit Multi-Touch-Gesten durch diese Technologie nicht unterstützt werden.

1.6.5. Oberflächen-kapazitive Touchscreen Technologie

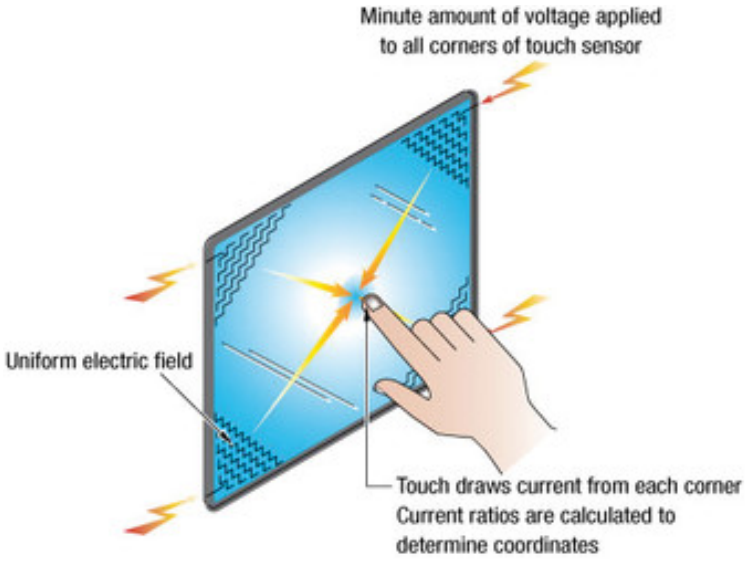


Abbildung 4.: Oberflächen Kapazitive Technologie

Die kapazitive Touchscreen-Technologie registriert Berührungen durch die Ablenkung innerhalb eines elektronischen Feldes, welches durch Elektroden an den Bildschirmkanten erzeugt wird. Sensoren an den Bildschirmecken messen dabei das elektronische Feld und damit jegliche durch Berührung ausgelöste Veränderung des Feldes. Der Finger fungiert dabei als Erdung, über den Elektronen aus dem Feld abfließen und somit für einen Spannungsabfall während der Berührung sorgt. Die genaue X- und Y- Koordinate errechnet sich aus den Strömen, die an den Sensoren registriert werden. Auch hier beschränkt sich die Technologie auf die Registrierung von lediglich einer Geste.

1.6.6. Projiziert Kapazitiver Touchscreen Technologie

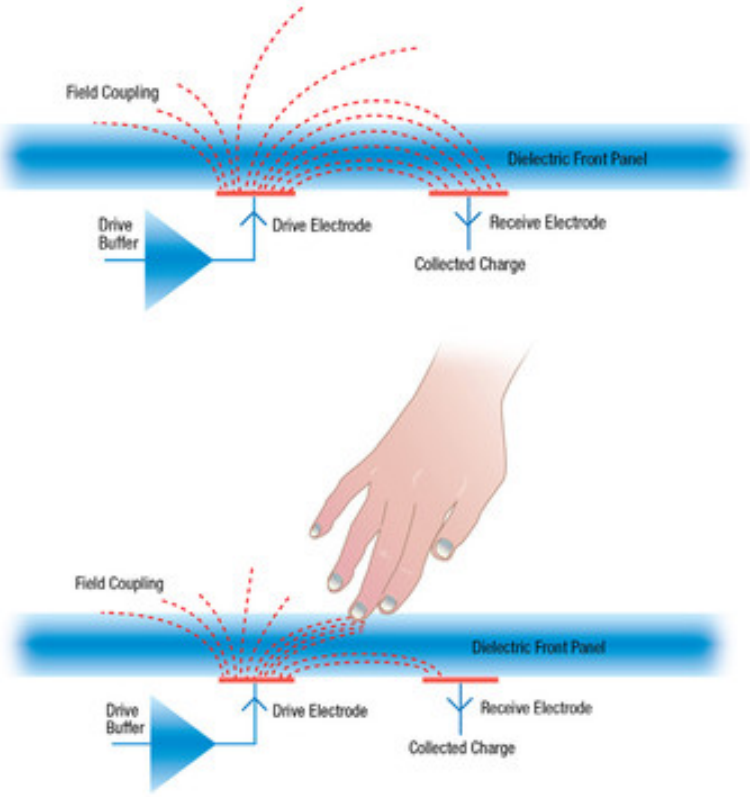


Abbildung 5.: Projiziert Kapazitive Technologie [**?**,]

Die projiziert- kapazitive Touchscreen-Technologie erweitert die Oberflächen-Kapazitiv-Technologie um einzelne kleinere Felder auf dem Touchbildschirm und ist die aktuellste der genannten Technologien. Der wesentliche Vorteil gegenüber der Oberflächen-Kapazitiven-Technologie ist, dass hier schon die Näherung eines Fingers ausreicht, um eine Berührung zu registrieren. Dadurch ist es möglich selbst durch dünne Handschuhe eine Berührung zu registrieren, was einen klassischen Einsatzkontext für den medizinischen Bereich darstellt. Die Abbildung zeigt, wie allein durch die Annäherung eines Fingers Teilbereiche des Feldes durch natürliche Ladung in den Fingern abgelenkt werden.

2. Gestenerkennung und Steuerung von Funktionalität durch Gesten

2.1. Signalverarbeitung und erste Gesteninterpretation durch das Android OS

Die Auflistung der einzelnen Technologien zum Touchbildschirm hat gezeigt, wie unterschiedlich die Umsetzungen der Berührungserkennung und deren Leistungsfähigkeit in Bezug auf die Erkennung von einfachen und mehrfachen Berührungen zur selben Zeit sind. Das Android Betriebssystem ist darauf ausgelegt, die unterschiedlichen Formen von Touchbildschirmen zu unterstützen. Je nach verbauter Technologie ist dabei die Erkennung von mehreren Berührungspunkten freigeschaltet oder nicht. Für den Entwickler einer Android Applikation ist dies jedoch völlig transparent. Im Falle einer registrierten Berührung wirft die API ein sogenanntes „MotionEvent“. Die Android API leistet hier eine erste Interpretation der Gesten, denn je nach Art der Berührung wird ein bestimmtes „MotionEvent“ gefeuert. Diese Events können abgefangen werden und liefern Berührungsparameter (Koordinaten) für eine weitere, detailliertere Interpretation der Gesten. Der folgende Abschnitt beschreibt die prototypische Implementierung exemplarischer Gesten und geht neben den Besonderheiten und Schwierigkeiten der Gesteninterpretation auch auf die durch die Android API angebotenen MotionEvents ein.

2.2. Erkennung und Interpretation von Gesten sowie Steuerung jeweiliger Funktionalität durch Gesten

Für die Erkennung von Gesten bietet das Android SDK die Schnittstelle OnTouchListener an. Darin lässt sich die Methode

```
boolean onTouch(View zoomView, MotionEvent anEvent){...}
```

überschreiben. Über den Parameter anEvent vom Typ MotionEvent können unterschiedliche Berührungsgesten erkannt werden. Die Methode OnTouchListener wird in Abhängigkeit von bestimmten Ereignissen aufgerufen. Hierzu ist zu einer View-Instanz die konkrete Implementierung des OnTouchListener Interfaces zu adressieren:

```
viewInstance.setOnTouchListener(touchListenerImpl);
```

Wenn der Touchsensor eine Berührung registriert, wird für das jeweilige Layout geprüft, welche darin enthaltenen Views die Methoden des OnTouchListeners überschreiben und das Event an die jeweilige Implementierung weitergeleitet. Des Weiteren ist dem Objekt vom Typ MotionEvent zu entnehmen, um welche Art von Berührung es sich handelt. Je nach Art und Weise der Berührung liefert die Maskierung der zur Event Aktion

```
anEvent.getActionMasked()
```

ein Ergebnis, welches durch ein Element der Menge aus den folgenden Konstanten beschrieben wird:

Menge M

{

```
int ACTION_DOWN = 0;
int ACTION_UP = 1;
int ACTION_MOVE = 2;
int ACTION_CANCEL = 3;
int ACTION_OUTSIDE = 4;
int ACTION_POINTER_DOWN = 5;
int ACTION_POINTER_UP = 6;
int ACTION_HOVER_MOVE = 7;
int ACTION_SCROLL = 8;
```

```
int ACTION_HOVER_ENTER = 9;
int ACTION_HOVER_EXIT = 10;
}
```

Für die Interpretation der Gesten und weiter die Steuerung exemplarischer Funktionalitäten wie Zoom oder Drag gilt dann:

ACTION_DOWN: Der Touchsensor registriert eine Berührung in einem in sich geschlossenen Bereich.
Beispiel: Die Fingerkuppe eines Nutzers berührt den Touchscreen. Die Koordinaten erhält der Entwickler durch das Auslesen des Event-Parameters. Über die Methoden

`anEvent.getX()` und
`anEvent.getY()`

können die Koordinaten ausgelesen werden, auf denen sich ein Finger auf dem Touchbildschirm befindet. Diese Methoden geben im Fall von mehrerer Berührungen (z.B. mehrere Finger) immer die Koordinaten des ersten Berührungspunktes an.

ACTION_UP: Der Touchsensor registriert das Aufheben einer ursprünglich registrierten Berührung. Dies setzt zuvor die Registrierung einer ACTION_DOWN voraus.

ACTION_POINTER_DOWN: Der Touchsensor registriert eine zweite Berührung (zweiter Finger). Die Koordinaten der zweiten Berührung, sowie auch im Falle weiterer Berührungspunkte, lassen sich aus dem Event entsprechend der Reihenfolge der Berührung ermitteln:

```
anEvent.getX(0..n);
anEvent.getY(0..n);
```

Dies entspricht der Berührung mit dem ersten bis N-ten Finger. Für die Funktionalität Zoom ist es notwendig, die initiale Distanz zwischen den zwei Berührungspunkten zu ermitteln und temporär vorzuhalten.

ACTION_MOVE: Der Touchsensor registriert eine Bewegung weg vom ursprünglich registrierten Berührungspunkt, dessen Ermittlung diesem Event vorausgeht. Um die o.g. Funktionalitäten zu realisieren,

wird für die Ein-Finger-Geste "Drag"der Richtungsvektor bei einer Bewegung auf dem Touchbildschirm ermittelt, indem vom initialen Berührungsspunkt bis zur aktuellen Position des Fingers die Differenz in x wie in y Richtung errechnet wird.

2.3. Gestengesteuertes Vergrößern und Verkleinern sowie Veränderung des Fokus für Darstellungsobjekte vom Typ ImageView

Ian F. Darwin liefert mit seinem Kochbuch für die Android Entwicklung zu der Zoomfunktionalität folgenden Ansatz:

```
// Remember some things for zooming
PointF start = new PointF();

ImageView view = (ImageView) v;
// make the image scalable as a matrix
view.setScaleType(ImageView.ScaleType.MATRIX);
...
switch (event.getAction() & MotionEvent.ACTION_MASK) {
case MotionEvent.ACTION_DOWN: //first finger down only
start.set(event.getX(), event.getY());
break;
...
case MotionEvent.ACTION_MOVE:
if (mode == DRAG)
matrix.postTranslate(event.getX() - start.x, event.getY() -
start.y);

else if (mode == ZOOM)
matrix.postScale(scale, scale, mid.x, mid.y);

// Perform the transformation
view.setImageMatrix(matrix);
```

}
[?,]
Diese Auszüge aus Darwins Kochbuch zur Andorid Entwicklung zeigen schematisch die dynamische Skalierung einer ImageView in Abhängigkeit der erkannten Gesten mit Hilfe einer speziell für diesen Objekttyp anpassbaren Matrix. Die Matrix lässt sich direkt aus einer Objektinstanz vom Typ ImageView holen und mittels der Methoden

- `postTranslate()` und
- `postScale()`

verarbeiten. Dabei ist zu beachten, dass die Matrix nicht direkt verändert werden darf, sondern nur deren Kopie. Der Richtungsvektor für die neue Fokussierung und damit Transformation der Matix errechnet sich aus der Differenz der aktuellen Koordinaten zu den Anfangskoordinaten und wird zur Transformation der Matrix entsprechend übergeben:

```
matrix.postTranslate(event.getX() - start.x,
        event.getY() - start.y)
```

Im Fall von mehreren registrierten Berührungen, wie z.B. bei der Pinch-Open-Geste für die Ansteuerung einer Zoom-Funktionalität, muss zunächst die Anfangsdistanz zwischen den Berührungspunkten auf dem Touchbildschirm ermittelt werden. In einem Koordinatensystem aus zwei Dimensionen (x|y) wird die Länge einer Strecke von P1(xp1|yp1) und P2(xp2|yp2) wie folgt berechnet: $\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$

In Darwins Implementierung ist dies in der Methode `spacing()` realisiert. Weiter ist über das gesamte ACTION_MOVE Event hinweg, die neue Distanz der Berührungspunkte zu berechnen und mit der initialen Distanz in ein Verhältnis zu setzen. Der Quotient aus

$$ScaleFaktor = \frac{dist_{new}}{dist_{old}}$$

ergibt den Faktor für die Skalierung der ImageView. Die ImageView wird daraufhin entsprechend relativ zu der Positionsänderung der Berührungspunkte neu skaliert.

```
matrix.postScale(scale, scale, mid.x, mid.y);
```

Darwins Vorgehensweise stellt eine komfortable und einfache Möglichkeit dar, eine ImageView Instanz dynamisch zu skalieren oder den Fokus der Darstellung zu verändern. Jedoch bezieht sich die Lösung allein auf Darstellungselemente vom Typ ImageView, wohingegen die Superklasse View, die durch ImageView erweitert wird, es nicht vorsieht die Matrix zur Skalierung und Positionierung der View direkt zu verändern. Damit ist Darwins Lösung allein für Darstellungsobjekte vom Typ ImageView geeignet.

2.4. Abstraktion gestengesteuerter Funktionalitäten zum Skalieren und Fokussieren von Darstellungselementen

Dieser Abschnitt beschäftigt sich mit der Frage, inwieweit sich der Ansatz von Darwin generalisieren lässt und somit unabhängiger vom Typ des Darstellungsobjektes wird. Kann hierzu eine Lösung gefunden werden, so ist weitergehend zu diskutieren, inwieweit diese für andere Android App Projekte verfügbar gemacht werden kann.

2.4.1. Generalisierung der Funktionalitäten zum Skalieren und Fokussieren von Darstellungselementen

Die skizzierte Implementierung nach I.F. Darwin im letzten Abschnitt zeigt zwar, wie ein bestimmtes Darstellungselement in Abhängigkeit von bestimmten Gesten skaliert, jedoch gilt diese Lösung nicht für andere Darstellungselemente. Ein generischer Ansatz müsste die Skalierung und Fokussierung für alle Darstellungselemente leisten, mindestens jedoch für alle Elemente, die vom Typ android.view.View ableiten.
Der im Folgenden skizzierte Prototyp soll Darwins Ansatz um eine dynamische Anzahl von Darstellungselementen mit der Kompatibilität zu unterschiedlichen Darstellungstypen erweitern. Dieser Projektarbeit liegt

eine prototypische Implementierung bei (Klasse `Touch.java`), in der das Skalieren und Fokussieren von Darstellungselementen für allgemein alle Objekte ermöglicht wird, die vom Typ `android.view.View` ableiten (z.B. `TextView extends View`). Ein entsprechender Algorithmus könnte damit die Funktionalität zur Skalierung und Fokussierung applikationsübergreifend generalisieren. Für die Skalierung eines Darstellungsobjektes vom Typ View bietet die Android API folgende Methoden an:

- `setScaleX()`
- `setScaleY()`

und für die Veränderung des Fokus auf die View:

- `setTranslationX()`
- `setTranslationY()`

Ein dabei auftretendes Problem sind unerwünschte Seiteneffekte bei der Verwendung des Listeners zur Registrierung von Berührungen und Bewegungen. Dieser Listener kann einer View zugeordnet werden:

```
viewInstance.setOnTouchListener(touchListenerInstance)
```

Wird jedoch dieselbe View-Instanz innerhalb der TouchListener-Implementierung neu skaliert, kommt es auf Grund der angestoßenen Zoom-Funktionalität zu einem flackernden Übergang von der ursprünglichen Skalierung in die neue Skalierung. Dies liegt daran, dass das Koordinatensystem der View durch die Skalierung unmittelbar selbst verändert wird. Stellt diese View jedoch auch das Koordinatensystem für die Gestenerkennung, (`viewInstance.setOnTouchListener(...)`), so wird die Berechnung des Zoomfaktors gestört.
Die View, welche durch den Zoom neu skaliert wird, darf also im Falle einer gestengetriebenen Zoom-Funktionalität nie die Bemessungsgrundlage (Koordinaten) für die Berechnung des Skalierungsfaktors sein. Alternativ kann die sogenannte Parent-View der zu skalierenden View-Instanz als Bemessungsgrundlage herangezogen werden, wie es die folgende Abbildung zeigt:

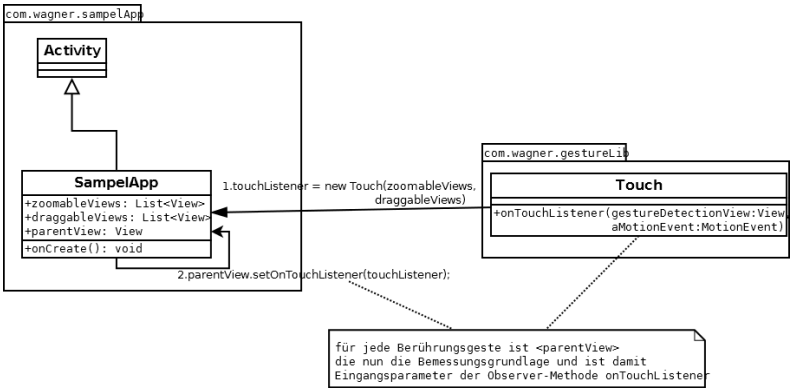


Abbildung 6.: Konzeption eines generischen Touchlisteners

Die Parent-View ist in der Baumstruktur des Layouts einer View-Instanz übergeordnet.

2.4.2. Ausblick zur Integration der generischen Implementierung zu gestengesteuerten Funktionalitäten

Der letzte Abschnitt hat gezeigt, wie gestengesteuerte Funktionalitäten zum Skalieren und Fokussieren von Darstellungselementen, allgemein für unterschiedliche Darstellungstypen definiert werden können. Dieser Abschnitt gibt einen Ausblick, wie der entworfene Ansatz aus dem letzten Kapitel allgemein für Android Applikationsprojekte angeboten werden kann.
In Android Projekten lassen sich wie auch in anderen Java Projekten zusätzliche Bibliotheken in Form von .jar - Archiv Dateien einbinden. Grundsätzlich ist es möglich diese Bibliotheken manuell in die jeweilige Entwicklungsumgebung einzufügen. Sollten jedoch unterschiedliche Versionen zu den jeweiligen Android APIs unterschieden werden sind bestimmte Werkzeuge zur Versionverwaltung und dem übersichtlichen Management von Abhängigkeiten zu empfehlen. Auch wenn Tests zu Funktionalitäten der Bibliothek vor dem Erzeugen des .jar - Archives durchlaufen werden müssen, sollte dies automatisiert sein. Hierzu gibt es unterschiedliche Werkzeuge, die sich für das Auflösen von Abhängigkeiten sowie das finale Bauen der Applikation anbieten. Exemplarisch bietet sich hier das Build

Werkzeug „Maven“mit dem Android Plugin an. Der im letzten Kapitel entworfene Ansatz lässt sich damit in eine eigenständige Bibliothek auslagern, um diesen in unterschiedlichen Android-Applikationsprojekten zu nutzen.

2.5. Fazit

Eingangs stand die Frage im Raum, wie den Einstiegshürden in der Android-Entwicklung sowie dem Problem des Code-Copy entgegen gewirkt werden kann. Anhand der Entwicklung von gestengesteuerter Funktionalität (speziell hier Zoom und Drag) ist verdeutlicht worden, welche Schwierigkeiten auf Einsteiger in der Android Entwicklung warten. Gleichzeitig wird mit der Implementierung nach Darwin (Kapitel 3.3) deutlich, welche Mengen an komplexem Sourcecode zu gestengesteuerter Funktionalität (siehe Implementierung nach F.Darwin Kapitel 3.3) in Applikationsprojekte integriert werden müsste, um grundlegendes Verhalten zu definieren. Der in diesem Projekt geschaffene Ansatz zur Generalisierung von gestengesteuerter Funktionalität bietet die Möglichkeit den dahinterstehenden Programmcode zu zentralisieren, also Code Copy vorzubeugen und gleichzeitig die Android App-Entwicklung zu vereinfachen. Für weitere gestengesteuerte Funktionalitäten kann ähnlich verfahren werden, vorausgesetzt es kann ein geeigneter Abstraktionsgrad gefunden werden, wie er hier für die Skalierung und Fokussierung definiert ist.

Anhang

Listing 1: Die Klasse Touch.java

```
package com.wagner.android.gesturelib.gestureutils;

import android.graphics.PointF;
import android.util.FloatMath;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;

import java.util.List;

/**
 * This Class offers a generic Zoom and Drag
 * functionality that is controlled by using
 * Touch Gestures
 *
 * @author Stephan Wagner
 * @version 1.1.1.5
 *          Time: 22:06
 */
public class Touch implements View.OnTouchListener {

/**
 * The List of views that will be scaled
 * by using touch gestures.
 */
private List<View> transformableViews;

/**
 * The Tag for logging to this Listener.
 */
private final String TAG = getClass().getName();

/**
 * The initial coordinates of the first touch.
 */
private PointF start;

/**
```

```
 * The flag for defining the touch mode.
 */
private int mode;

/**
 *  Touch mode as functionality selector.
 */
private enum Mode{NONE, DRAG, ZOOM}

/**
 * The initial Distance for the
 * calculation of zoom.
 */
private float initDist;

/**
 * The initial Distance for the
 * calculation of zoom.
 */
private float newDist;

/**
 * The current scale factor of the zoom.
 */
private float scaleFactor;

/**
 * The maximal scale factor;
 */
private float maxScale;

/**
 * The minimal scale factor;
 */
private float minScale;

/**
 * The latency of the scale. Makes the zoom slower.
 */
private float latency;

/**
 * The Constructor of the touchListener
 * Implementation sets the initial not null parameters.
```

```
     *
     * @param aTransformableViewList The list of View
     *                               Instances that should
     *                               be transformed.
     * @param aMaxScaleFactor        the maximal scale factor.
     * @param aMinScaleFactor        the minimal scale factor.
     * @param aLatency               the latency for the
     *                               transformation, that is
     *                               used to make it slower.
     */
    public Touch(final List<View> aTransformableViewList,
                 final float aMaxScaleFactor,
                 final float aMinScaleFactor,
                 final float aLatency) {

        transformableViews = aTransformableViewList;
        start = new PointF();

        //Setting initial Values
        mode = Mode.NONE.ordinal();
        initDist = 1;
        newDist = 1;
        scaleFactor = 1;

        //setting scale
        maxScale = aMaxScaleFactor;
        minScale = aMinScaleFactor;

        //setting latency
        latency = aLatency;
    }

    /**
     * The implementation of TouchListener Interface,
     * that realizes drag and zoom functionality controlled
     * by multi-touch-gestures
     * @param touchView gives the base for the
     *                  registration of each touch.
     * @param anEvent holds the type and the
     *                coordinates of each touch.
     * @return returns true if listener has consumed the event.
     */
    @Override
    public boolean onTouch(final View touchView,
```

```
                        final MotionEvent anEvent) {

switch (anEvent.getActionMasked()) {
//first finger down only
case MotionEvent.ACTION_DOWN:
    //getting position
    start.set(anEvent.getX(), anEvent.getY());

    Log.d(TAG, "mode=DRAG");
    mode = Mode.DRAG.ordinal();
    break;

//first finger lifted
case MotionEvent.ACTION_UP:
    break;

//second finger lifted
case MotionEvent.ACTION_POINTER_UP:
    //new initial Distance is the last calculated distance
    //e.g. for more than one following pinch open gestures
    //and the connected zoom in function.
    initDist = newDist;

    mode = Mode.NONE.ordinal();
    Log.d(TAG, "mode=NONE");
    break;

//second finger down
case MotionEvent.ACTION_POINTER_DOWN:
    initDist = spacing(anEvent);
    mode = Mode.ZOOM.ordinal();
    Log.d(TAG, "mode=ZOOM");
    break;

case MotionEvent.ACTION_MOVE:
    if (mode == Mode.DRAG.ordinal()) {
    //movement of first finger

    float newX = anEvent.getX();
    float newY = anEvent.getY();

    float distanceX = (start.x - newX) / latency;
    float distanceY = (start.y - newY) / latency;
```

```
    for (final View draggableView : transformableViews) {
        draggableView.setTranslationX(
                draggableView.getTranslationX() - distanceX);
        draggableView.setTranslationY(
                draggableView.getTranslationY() - distanceY);
    }

    } else if (mode == Mode.ZOOM.ordinal()) { //pinch zooming
    float newDist = spacing(anEvent);

    if (newDist > initDist || newDist < initDist) {
        float factor = newDist / initDist;

        if (newDist > initDist && scaleFactor < maxScale)
        //pinch open --> zoom in
        {
            factor = factor / latency; //latency
            scaleFactor = scaleFactor + factor;
        }
        if (newDist < initDist &&
                scaleFactor > minScale &&
                factor < scaleFactor)
        //pinch close --> zoom out
        {
            factor = factor / latency; //latency
            scaleFactor = scaleFactor - factor;
        }

        for (final View scalableView : transformableViews) {
            scalableView.setScaleX(scaleFactor);
            scalableView.setScaleY(scaleFactor);
        }

        }
    }
    break;
}


return true;
}


/**
```

```
     * Calculates the distance between first
     * two touch points on touch screen.
     *
     * @param anEvent that contains the coordinates
     *                of the touch points.
     * @return the distance as float.
     */
    private float spacing(final MotionEvent anEvent) {
        final float x = anEvent.getX(0) - anEvent.getX(1);
        final float y = anEvent.getY(0) - anEvent.getY(1);
        return FloatMath.sqrt(x * x + y * y);
    }


}
```

A. Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, den 28. Oktober 2015

(Unterschrift)