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:	30. 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.

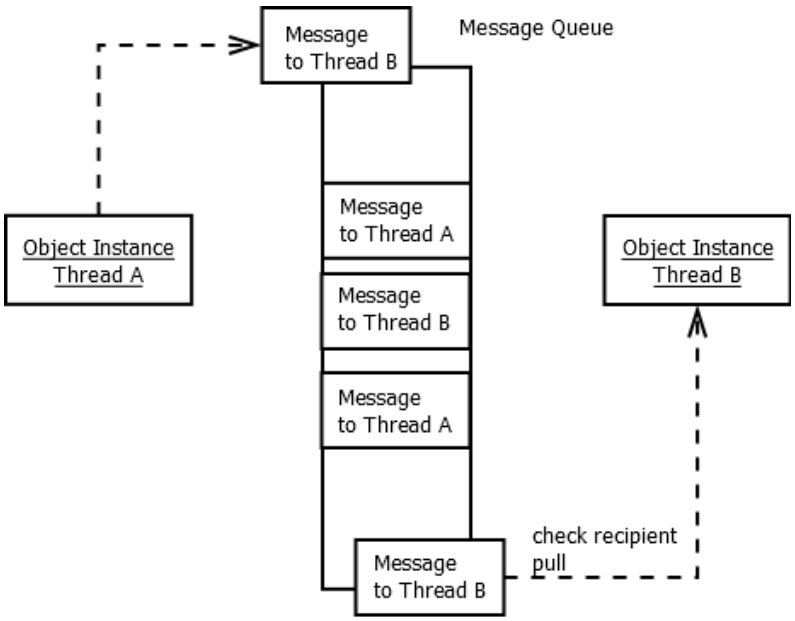
1.5.1. Geteilte Datei/ Speicher

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.

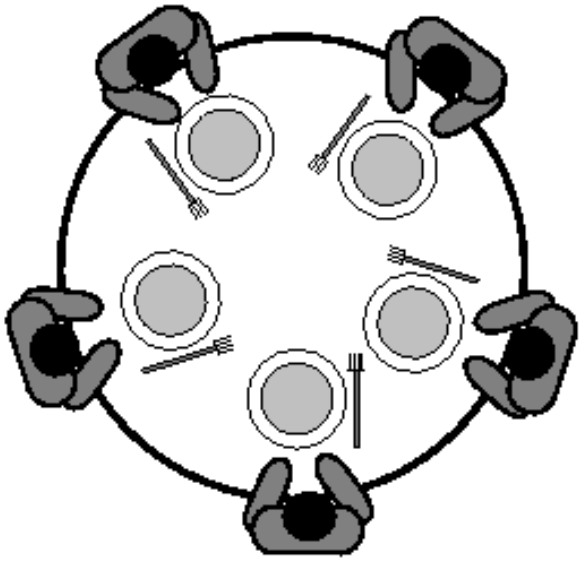


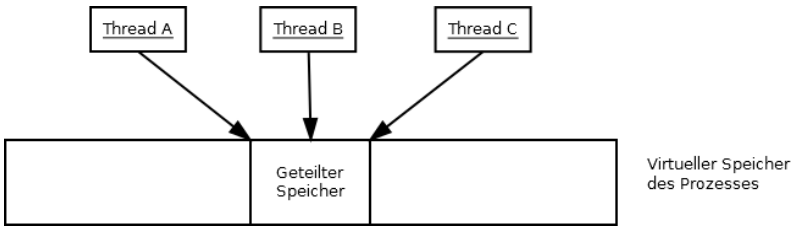
Abbildung 1.: Philosophenproblem [**?**,]

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. Speicherleck

Speicherlecks (eng. Memory Leaks) entstehen häufig aus Programmierfehler heraus, in denen Speicher reserviert, aber dieser nicht mehr freigegeben wird. Geschieht dies ausreichend oft während einer Laufzeit innerhalb des Addressraums des Programmes, so gerät im schlimmsten Fall der seitens der Hardware begrenzte Speicher an seine Grenzen. Das Betriebssystem registriert dieses Verhalten und terminiert sofort die Ausführung des Programmes. Bei der Technik des geteilten Speichers, ist das Risiko eines Speicherlecks z.B. dann präsent, wenn sich der Entwickler um korrekte Dereferenzierung des Speichers und dessen Freigabe explizit kümmern muss. Wird Speicher n-Fach allokiert und nicht wieder freigegeben, so handelt es sich um ein ernstes Speicherleck. Auch bei der MessageQueue sind Speicherlecks möglich, z.B. wenn kontinuierlich Botschaften in die Queue gepackt werden, diese aber nicht aus der Queue wieder herausgenommen werden.

1.6.4. Reihenfolgeproblem

Im Falle einer bidirektionalen Kommunikation mittels einer MessageQueue muss beachtet werden, dass es nicht vorhersehbar ist, in welcher Reihenfolge die Nachrichten in die MessageQueue gelegt werden, oder wann sich welcher Thread eine Message aus der Queue holt. Dies kann zu unerwarteten Verhalten führen. So kann ein Thread B ungewollt blockieren, wenn er für seine weitere Verarbeitung eine bestimmte Nachricht benötigt und diese jedoch sich in der Reihenfolge hinter einer Nachricht für den anderen Thread A befindet. So muss Thread B solange warten bis Thread A seine Nachricht aus der Queue nimmt.
Welche Konzepte es gibt, um sich den allgemeinen Problematiken und Risiken von Nebenläufigkeit speziell für Android Applikationen zu nähern ist zentraler Forschungsschwerpunkt dieser Arbeit. Diese Konzepte werden im Kapitel 2 im einzelnen vorgestellt.

1.7. Parallelverarbeitung und Besonderheiten unter Android

Im vorangegangenen Abschnitt wurde allgemein auf die Terminologie, die Eigenschaften, sowie die Kommunikation innerhalb von Nebenläufigkeit eingegangen. Hier gilt es nun einen Fokus auf Nebenläufigkeit im mobilen Umfeld zu setzen, insbesondere unter dem Betriebsystem Android. Weiter wird ein erster Einblick in das Komponentenmodell, sowie in den Lebenszyklus von Android Applikationen erarbeitet.

1.7.1. Prozesse und Threads unter Android

Prozess Charakteristika Das Android Betriebssystem basiert auf dem Linux/Unix System. Daher gelten die o.g. Eigenschaften zu Prozessen unter dem Linux/Unix Betriebssystem auch für Android. Sie werden lediglich um folgende Charakteristika erweitert.

1. Fordergrundprozess → Fordergrundprozesse sind verantwortlich für alle Komponenten einer Anwendung, die unmittelbar im Vordergrund, also für den Nutzer sichtbar sind.

2. sichtbarer Prozess → Sichtbare Prozesse fassen alle Komponenten zusammen, die zwar nicht unmittelbar sichtbar sein müssen, jedoch Komponenten aus dem Vordergrund beeinflussen.

3. Service Prozess → Service Prozesse laufen losgelöst von anderen Prozessen. Einmal gestartet laufen diese selbstständig (z.B. das Abspielen von Musik). Sie lassen sich jedoch weiterhin von anderen Prozessen steuern.

4. Hindergrundprozess → Der Hintergrundprozess hält gestoppte Anwendungskomponenten. So werden darin Activity Instanzen abgelegt, für welche die onStop() Methode (siehe Activity Lifecyle) aufgerufen wurde. Dies hat die Funktion, die Anwendung möglichst schnell wieder zu reaktivieren, wenn der Nutzer dies wünscht. Der Hintergrundprozess darf nicht mit der im folgenden Kapitel thematisierten Hintergrundverarbeitung zusammen in einen Kontext gebracht werden.

5. leerer Prozess → Leere Prozesse werden bei ausreichend Hardware Ressourcen durch das Android Betriebssystem erzeugt und als Ressourcen für einen schnelleren Start von Applikationen gehalten. Wünscht ein Nutzer eine Anwendung zu starten, so existiert bereits hierfür ein Prozess mit entsprechender Laufzeitumgebung.

Rein aus der Sicht des Betriebsystems handelt es sich stets um den gleichen Prozess, jedoch kann er unterschiedliche Charakteristika innerhalb seines Lebenszyklus annehmen.

Sandbox Prinzip Eine weitere Besonderheit in Android erweitert das Sandbox Prinzip von Unix/Linux Systemen zur Kapselung von Prozessen und deren Ressourcen (siehe Abschnitt 1.2.1 “Prozesse und Threads”). Dabei erzeugt Android bereits zum Installationszeitpunkt pro Anwendung einen speziellen User. Wird die Anwendung gestartet, so richtet das Betriebssystem einen in sich geschlossenen Speicherbereich ein und ordnet

diesen dem jeweiligen Appliction-User zu. Somit wird der exklusive Zugriff auf den Speicherbereich allein durch diese Applikation realisiert.

UI-Thread als Main Thread Bei der Entwicklung von Android Applikationen steht die Benutzerschnittstelle im Vordergrund. So erscheint es konsequent, dass das User Interface durch den Main Thread verarbeitet wird (hier gilt also Main-Thread = UI-Thread). Dieses Ausführungsmodell steht z.B. dem von Java.Swing entgegen, in welchem die Benutzerinteraktion in einem sekundären Thread, ungleich dem Hauptthread gesteuert wird. Dies hat unweigerlich Einfluss auf die Konzeption von Applikationen unter Android, welche u.a. zeitintensive Operationen durchführen müssen. Denn in diesem Fall darf der UI Thread nicht blockieren und sich nach Ende der Verarbeitung erst wieder zurück melden. Die zeitintensive Operation muss in einen sekundär Thread ausgelagert werden, sodass der UI Thread weiter Benutzereingaben verarbeiten kann.

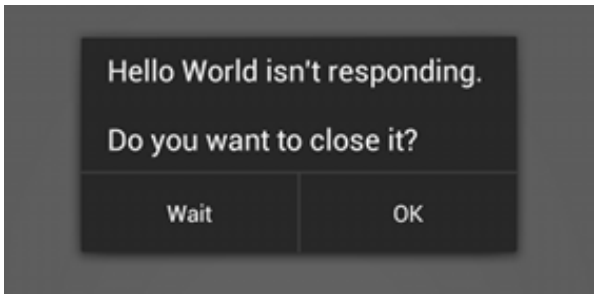
1.7.2. Besonderheiten im Android Umfeld bezüglich des Thread Managements

Das Android Betriebssystem formuliert einige spezielle Regeln für die Ausführung von Threads, die sich an einen hohen Anspruch seitens Google an der Benutzbarkeit der Applikationen orientieren. So ist der Thread für die Steuerung der Benutzeroberfläche (User Interface kurz UI) unter besonderer Beobachtung. Wird dieser Blockiert oder ist ausgelastet mit zeitintensiven Operationen, wird dieser vom Betriebssystem nach einer bestimmen Zeit angehalten. Je nach Konfiguration des Android Betriebssystems wird es dem Nutzer angeboten entweder weiter zu warten oder die Applikation zu terminieren.

1.7.2.1. ANR Dialog

Ein grundsätzlicher Anspruch, den Google an mobile Anwendungen unter seinem Android Betriebssystem stellt, ist die Benutzbarkeit durch den alltäglichen Anwender. Dieser muss über keine technischen Kenntnisse verfügen um die Applikation entsprechend einfach und intuitiv benutzen kön-

nen. So ist es unerwünscht, dass eine Applikation nach dem Start irgend eines Vorganges blockiert, also auf Interaktion des Nutzers nicht reagiert. Google formuliert in seiner Andorid Developper Dokumentation daher die grundsätzliche Anforderung der kontinuierlichen Ansprechbarkeit von Appliationen bzw. der nie zu unterbrechenden Interaktionsfähigkeit zwischen Benutzer und Applikation. So ist es gemäß der Designvorgaben für Android Applikationen unerwünscht, das Applikationen blockieren. Um jedoch für den Fall einer blockierenden Anwendung gerüstet zu sein, bietet Android die Möglichkeit, mittels eines Applikation Not Responding Dialogs, kurz ANR Dialog die Verarbeitung innerhalb einer Anwendung abzubrechen oder weiter auf deren Ergebnis zu warten. Wird im Dialog auf „Warten“geklickt, gibt das Betriebssystem den UI Thread wieder zur Ausführung frei. Folgende Abblildung zeigt einen exemplarischen ANR Dialog:

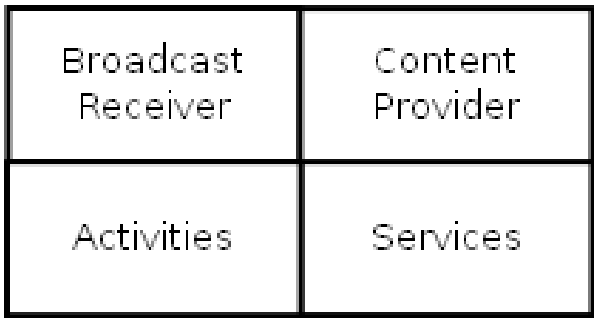


In diesem Beispiel greift Android in den Lebenszyklus der Aktivity ein, von der die zeitintensive Berechnung ausgeht und pausiert diese. Je nach Wunsch des Nutzers wird die Applikaiton dann in den Status „Pausiert“überführt, oder terminiert (siehe: onStoped() bzw. onDestroyed() im Lifecycle Diagramm) und vom Garbage Collector entsorgt (siehe hierzu Lebenszyklus einer Activity im nächsten Abschnitt). Die Zeit T innerhalb derer eine Applikation nicht auf Benutzereingaben reagiert und deren Ablauf der genannte Dialog erscheint ist für jedes Android Gerät konfigurierbar. Obwohl diese Eingreifmöglichkeit eine wichtige Funktion für die Kontrolle von Applikation durch den Nutzer darstellt, haben sich diverse Gerätehersteller mittlerweile entschieden die Zeit T sehr groß zu wählen, oder sogar den ANR-Dialog generell zu deaktivieren. In den Android Versionen ab Honeycomb führt das Blockieren einer Applikation bereits zur sofortigen Terminierung durch das Betriebssystem. Doch unab-

hängig vom ANR-Dialog bleibt der direkte Zusammenhang zwischen der Ansprechbarkeit einer Applikation und der Benutzbarkeit der Anwendung und dem Nutzererlebnis. Blockiert über zu lange Zeit eine Anwendung, so steigt damit auch die Unzufriedenheit des Nutzers über die Anwendung, besonders da er häufig die technischen Zusammenhänge, welche eine zeitintensive Verarbeitung durch aus rechtfertigen können, nicht kennt und gemäß der angesprochenen Designvorgaben auch nicht kennen muss.

1.7.2.2. Komponentenmodell

Bisher wurde die technische Ausführung von Android Applikationen auf der Betriebssystemebene erläutert. Für die einzelnen Konzepte der Nebenläufigkeit unter Android gilt es nun jedoch näher auf die Konzeption von Applikationen mittels klar definierten Komponenten selbst ein zu gehen, bevor im Kapitel 2 die einzelnen Konzepte an konkreten Implementierungsbeispielen verdeutlicht werden. Die folgende Abbildung zeigt die Basiskomponenten aus denen Anwendungen unter dem Android Betriebssystem bestehen:



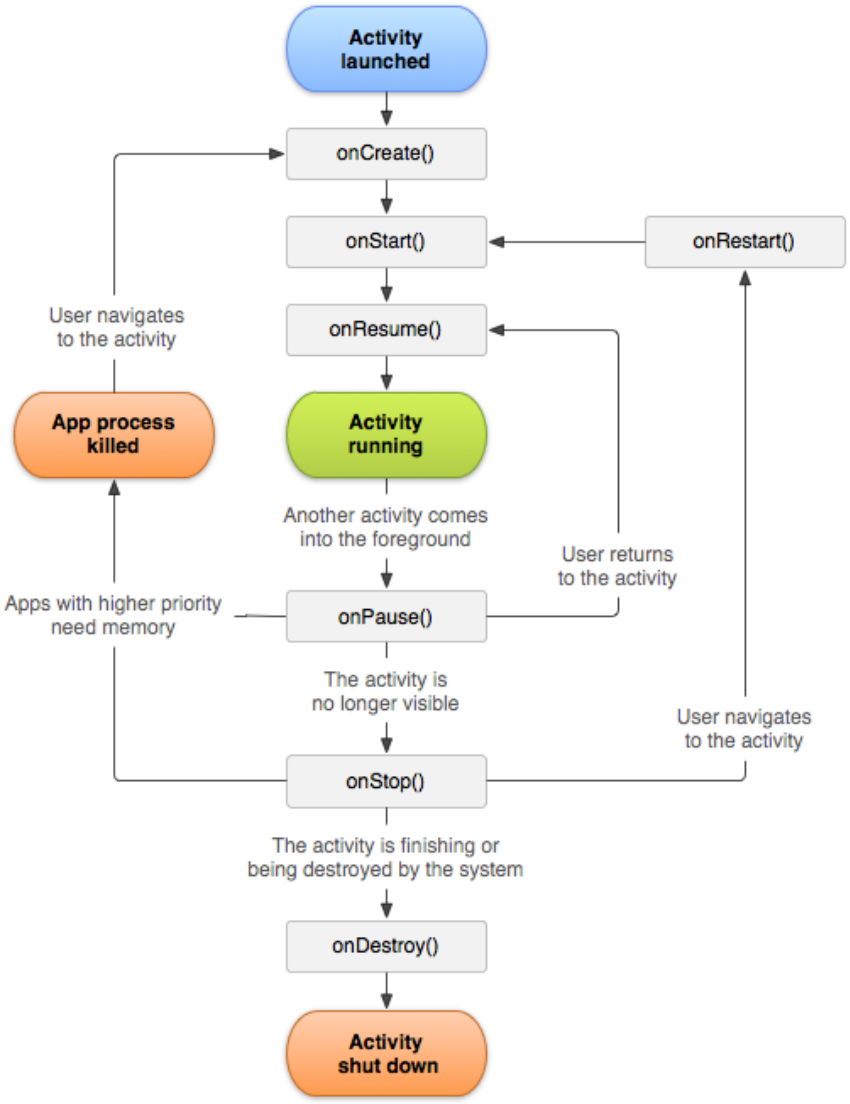
- Broadcast Receiver erlauben die Registrierung von systemweiten oder applikationsinternen Events.
- Content Provider repräsentieren Daten als relationalen Datensatz und ermöglichen den Zugriff über Applikationsgrenzen hinaus auf diese Daten.
- Activities definieren das Verhalten von graphischen Benutzerschnittstellen
- Services können zeitintensive Hintergrundberechnungen abbilden.

Bei Services ist zu beachten, dass diese, wenn nicht anders definiert in dem Main Thread laufen, was jedoch für zeitintensive Verarbeitungen zu Problemen mit dem Benutzerinterface führen kann, wie in den folgenden Abschnitten näher beschrieben wird. Services dienen lediglich als mögliche Kapselung um Hintergrundverarbeitungen klarer vom Rest der Applikation zu trennen, sowie Verarbeitungen ohne aktives Benutzerinterface durchzuführen. Die Android Developper Dokumentation weist explizit darauf hin, keine zeitaufwändigen Verarbeitungen hier zu definieren wenn der Service im Main Thread läuft, da ansonsten die Gefahr besteht, dass der Main Thread und somit auch das Benutzerinterface blockiert. Dies gilt neben Services auch für die anderen Komponenten.

1.7.2.3. Lebenszyklus einer Android Anwendung (Activity)

Da sich diese Arbeit primär auf Applikationen mit Benutzerinterface konzentriert, macht es Sinn kurz auf den Lebenszyklus von Activities einzugehen der bereits in Kapitel „Besonderheiten im Android Umfeld bezüglich des Thread Managements“kurz aufgegriffen wurde. Applikationen unter dem Android Betriebssystem unterscheiden sich in einigen Details deutlich von normalen Java Applikationen. Im Hinblick auf die nebenläufige Verarbeitung ist es daher von Bedeutung einen genaueren Blick auf den Lebenszyklus von Android Applikationen zu werfen. Die folgende Abbildung gibt einen Überblick über die Stati einer gestarteten Activity. Eine Objekt Instanz vom Typ android.os.Activity ist dabei der Einstiegspunkt in eine GUI-Applikation und wird vom Betriebssystem beim Start der Anwendung aufgerufen. Die einzelnen Statusübergänge, die in der Abbildung durch die jeweiligen Rückrufmethoden (onCreate(), onStart, onResume())symbolisiert werden, sind für diese Arbeit weniger von Bedeutung. Interessanter sind die Abhängigkeiten zum Lebenszyklus des Prozesses, welcher für die Anwendung gestartet wurde, und damit die Frage nach der Lebensdauer von Threads. Die Abbildung zeigt deutlich zwei Szenarien in denen jeweils die Anwendung aus dem Sichtbarkeitsbereich des Nutzers entfernt wird (Übergang von onPaused() $\rightarrow$ onStop()). In dem ersten Szenario bleibt der Prozess nach dem Aufruf der onStop() Rückrufmethode bestehen. Das Android Betriebssystem behält damit die Reservierung des Adressraums im Speicher für diesen Prozess und ermög-

licht ein schnelles Wiederaufrufen der Applikation, wenn der Nutzer dies wünscht (siehe in Abbildung Übergang von onStop() → onRestart()). Innerhalb des Prozesses definierte Nebenläufigkeit wird angehalten, d.h. die Ausführung aller Threads (auch des UI-Threads) wird unterbrochen. Die Threads selbst werden in den Status „Wartend“oder „Schlafend“überführt. Kehrt der Nutzer zu der Anwendung zurück, wird zunächst der UI-Thread wieder gestartet und in der Activity Instanz die Methode onResume() aufgerufen. Soll die nebenläufige Verarbeitung wieder gestartet werden, so empfiehlt es sich dies in der onResume() Methode zu implementieren.



Im zweiten Szenario wird der Adressraum dieses Prozesses für andere Anwendungen benötigt (siehe Abbildung onStop() → onCreate()). Das

Betriebssystem terminiert in Folge dessen den Prozess und damit auch alle darin laufenden Threads. Für die Nebenläufigkeit auf Thread Ebene würde dies bedeuten, dass alle Threads zusammen mit dem Prozess mit terminiert werden. Kommt die Anwendung durch den Aufruf des Nutzers wieder in den Vordergrund, so wird zunächst wieder ein Prozess mit entsprechendem Adressraum vom Betriebssystem eingerichtet, der UI Thread neu erzeugt und nun in dem UI Thread die Activity neu instanziiert (dabei Aufruf onCreate()). Alle zuvor erzeugen sekundären Threads existieren nicht mehr. Die Verarbeitung im Hintergrund wurde unterbrochen und terminiert. Die Hintergrundverarbeitung lässt sich ggf. neu aus der onCreate Methode heraus neu starten.

1.8. Anforderungen an Applikationen

Abschließend zu dieser Einführung in die grundlegenden Konzepte zur Ausführung und den Bestandteilen von Android Applikationen, werden nun allgemeine Anforderungen an die Applikationen zusammengefasst, wie sie auch der Application Style Guide von Google vor gibt. Applikationen für das Android Betriebssystem verfügen zum großen Teil über eine graphische Benutzeroberfläche, die wenigsten sind reine Hintergrund Programme. Darum liegt auch der Fokus der Applikationen auf deren Benutzbarkeit. Entsprechend ergeben sich auch besondere Anforderungen für die Entwicklung von Nebenläufigkeit. In Android Applikationen wird die Interaktion mit dem Benutzer über den Hauptthread (= UI- Thread), abgehandelt. Sind nun innerhalb einer Applikation zeitintensive Berechnungen oder andere Vorgänge definiert, so ist darauf zu achten, dass dadurch nicht der UI-Thread blockiert wird. Dieser soll stets bereitgehalten werden um mit dem Benutzer zu interagieren. Eine nebenläufige Verarbeitung ist also so zu definieren, dass sekundäre Threads vom UI Thread aus initialisiert und gestartet werden können und im Weiteren völlig losgelöst vom UI Thread operieren. Kommt es zum Nachrichtenaustausch zwischen dem UI Thread und den sekundären Threads (Datenaustausch) darf dies für den UI Thread keine zeitaufwändige Operation darstellen, diese Nachrichten zu verarbeiten. Weiter benötigt der Nutzer auch im Einzelfall Rückmel-

dung über den Bearbeitungsstand aus den sekundären Threads, sowie auch im Fehlerfall entsprechende Meldungen.

1.9. Fokus und Eingrenzung

In dieser Arbeit soll sich primär darauf konzentriert werden, wie asynchrone Parallelverarbeitung realisiert werden kann und welche Gefahren sich aus der Komplexität dieser Aufgabe ergeben können. Weiter ist die Fragestellung im Fokus, wie eine Fehlerbehandlung innerhalb asynchroner Parallelverarbeitung realisiert werden kann bzw. wie unterschiedliche Meldungen an den UI -Thread übertragen werden können. Um die Problemstellung der asynchronen Parallelverarbeitung für Andorid Plattformen greifbarer zu machen werden zwei Szenarien beschrieben in denen man bei der Implementierung die asynchronen Parallelverarbeitung sinnvoll demonstrieren kann:

Zugriff auf Web Ressourcen In Android Applikationen ist es häufig nötig auf Web Ressourcen zu zu greifen. Bis jedoch die gesamte Ressource geladen ist, kann es u.U. je nach Verfügbarkeit des Netzwerks zu längeren Wartezeiten kommen. Arbeitet die Applikation nun streng sequentiell, so würde sie blockieren, bis die Ressourcen geladen sind und sich dann damit zurückmelden.

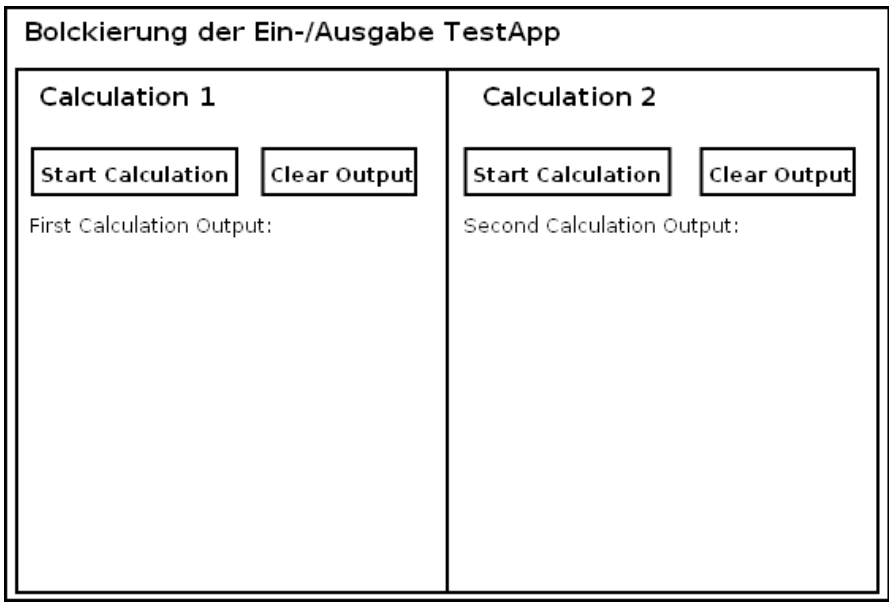
Zeitintensive Berechnungen Neben dem Zugriff auf Netzwerkressourcen können auch einzelne Berechnungen oder Suchfunktionen längere Zeit in Anspruch nehmen. Ebenso ist hier auf eine asynchrone Parallelverarbeitung zu achten, denn genauso wie in o.g. Szenario kann hier die Anwendung bei sequentieller Abarbeitung blockieren (also keine Benutzereingaben verarbeiten) und in Folge dessen durch das Betriebssystem terminiert werden.
Die im Kapitel 2 aufgeführten Implementierungsbeispiele beziehen sich primär auf das zweite Szenario, um für Präsentationszwecke unabhängiger von Netzwerkeigenschaften zu sein.

2. Asynchrone Parallelverarbeitung unter Android

Für die Realisierung von asynchroner Parallelverarbeitung werden in diesem Kapitel drei Lösungsansätze entwickelt. Jeder dieser Ansätze versucht die asynchrone Verarbeitung im Hintergrund, losgelöst vom Hauptthread der Applikation zu realisieren. Als Beispiel für die im Hintergrund zu tätigende Verarbeitung steht eine rechenintensive und damit zeitaufwändige mathematische Operation. Ziel ist mit dieser Operation ausreichend Rechenauslastung zu erzeugen, sodass im Falle einer synchronen Abarbeitung die Applikation blockieren kann.

2.1. Bolckierung der Ein-/Ausgabe durch zeitintensive Verarbeitung

Dieser Abschnitt soll einen Einblick in das Verhalten von Applikationen unter dem Android Betriebssystem geben, wenn zeitintensive Berechnungen durchgeführt werden. Das Implementierungsbeispiel ist an das in Kapitel 1.9 beschriebene Szenario zu zeitintensiven lokalen Berechnungen angelehnt. Die folgende Abbildung zeigt zwei Fenster einer Android Applikation mit je einem Knopf zum Start einer bestimmten zeitintensiven mathematischen Berechnung, deren Ergebnisse auf im jeweiligen Fenster angezeigt wird.



Die mathematische Operation ist so gestaltet, dass sie im Komplexitätsgrad variabel einstellbar ist und somit auch die Verarbeitungszeit verlängert werden kann. Dabei handelt es sich um die Berechnung einer Quersumme aus einer sehr großen Primzahl. Die Ermittlung der Primzahl verlängert sich in Abhängigkeit zu der im Die mathematische Operation ist so gestaltet, dass sie im Komplexitätsgrad variabel einstellbar ist und somit auch die Verarbeitungszeit verlängert werden kann. Dabei handelt es sich um die Berechnung einer Quersumme aus einer sehr großen Primzahl. Die Ermittlung der Primzahl verlängert sich in Abhängigkeit zu der im Vorhinein definierten Mindestgröße der Primzahl (hier 1500Byte BigInteger):

```
BigInteger veryBig = new BigInteger(1500, new Random());
BigInteger randomPrimeNumber = veryBig.nextProbablePrime();
int summe = 0;

while (0 != randomPrimeNumber.compareTo(BigInteger.ZERO))
{
// addiere die letzte ziffer der uebergebenen zahl zur summe
summe = summe + (randomPrimeNumber.mod(BigInteger.TEN)).intValue();
```

```
// entferne die letzte ziffer der uebergebenen zahl
randomPrimeNumber = randomPrimeNumber.divide(BigInteger.TEN);
}
targetString.append(summe);
```

Die Berechnung wird dabei angestoßen aus der Activity und allein innerhalb des UI Thread durchgeführt. In einem ersten Versuch wird die Komplexität nach und nach erhöht um zunehmend die Interaktion der Applikation mit dem Nutzer über die Benutzerschnittstelle zu blockieren. Nach einer bestimmten Blockierungszeit T ist zu beobachten, das der ANR Dialog des Betriebssystems erscheint und anbietet die Applikation zu beenden oder weiter zu warten. An dieser Stelle registriert das Betriebssystem, dass der UI-Thread als Haupt-Thread der Applikation für die Zeit T keine weitere Interaktion des Benutzers verarbeiten kann. Der Benutzer ist bis zu diesem Zeitpunkt nach Aktivierung der Berechnung nicht in der Lage die Applikation zu beenden oder zu wechseln. Daher schreitet nun das Betriebssystem ein, pausiert die Applikation und bietet dem Nutzer in einem ANR-Dialog an, die Applikation zu terminieren, oder weiter auf die Rückmeldung der Applikation zu warten.

2.2. Parallelverarbeitung mit der Java Standard Edition (Java Concurrency)

In diesem Beispiel wird nun versucht die zeitintensive Berechnung aus dem letzten Abschnitt, mittels der allgemeinen Mechanismen der Java Standard Edition in eine parallele Verarbeitung auszulagern. Damit soll das Blockieren der Anwendung verhindert werden. Hierzu wird die Berechnung in eine Klasse ausgelagert, die das Interface `java.lang.Runnable` implementiert. Diese kann darauf hin einem neuen Thread zur Ausführung übergeben werden. Der folgende Auszug aus dem Quellcode zeigt dabei den Konstruktor, sowie die `run()`- Methode der Klasse.

```
/**
* The constructor.
* @param aView a view element.
* @param aCallbackHandler the Callback Handler for sending
messages to ui-thread */ public RandomPrimeNumGenerator(final
View aView, final Handler aCallbackHandler)
{
Log.d(TAG,"Call Constructor");
targetView = aView;
handler = aCallbackHandler;
}
@Override
public void run()
{
Log.d(TAG, "Call run");
String result = startCalculation();
Message message = new Message();
Bundle bundle = new Bundle();
bundle.putCharArray(String.valueOf(targetView.getId()),
result.toCharArray());
message.setData(bundle);
Log.d(TAG,"Call handler");
handler.sendMessage(message);
}
```

Das hier verwendete Handler Objekt wird dem Konstruktor aus der Activity übergeben. Wird die durch den Konstruktor `RandomPrimeNumGenerator` erzeugte Runnableinstanz nun einem Thread Konstruktor übergeben und auf dieser Threadinstanz die `start()`-Methode Aufgerufen, so startet die Java Virtuelle Maschiene einen neuen Thread, der die zeitintensive Berechnungen ausführt. Der folgende Codeabschnitt ist in der Activity defininiert. Es wird somit aus dem Main Thread heraus ein neuer Thread erzeugt.

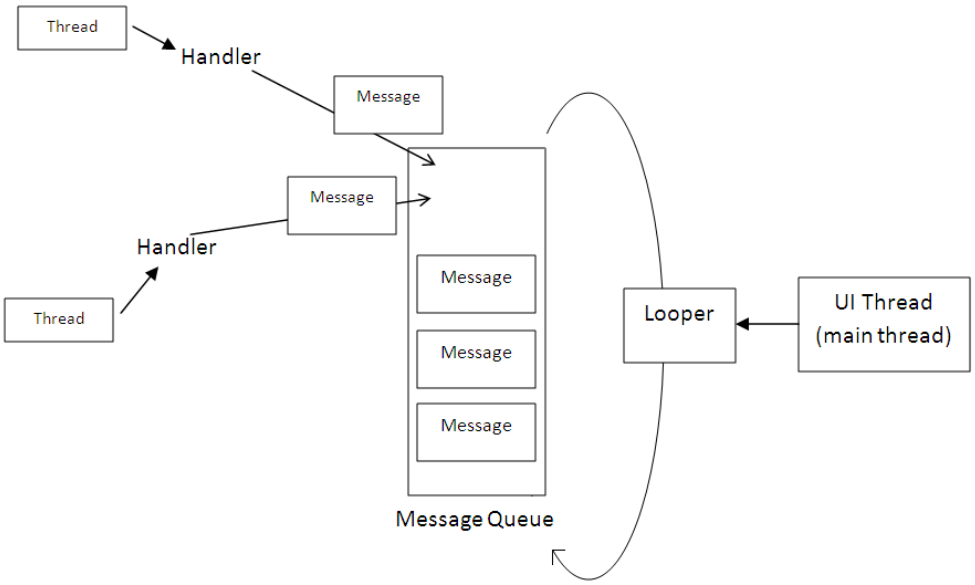
```
private final Handler handler = new Handler();

public void initCalculation(View aView)
{
RandomPrimeNumGenerator runnable = new RandomPrimeNumGenerator(aView,
handler);

Thread newThread = new Thread(runnable);
newThread.start();
}
```

Die Händler Instanz, welche dem Konstruktor der Klasse `RadomPrimeNumGenerator` mitgegeben wurde, dient dabei der Kommunikation zwischen der Activity im UI-Thread und dem neuen Thread. So ist es möglich mittels einer Rückruf Methode, Informationen zurück an den UI-Thread zu spielen. Die Handler Implementierung stammt aus dem Packet android.os und ist demnach keine Funktionalität aus der Java Standard Edition. Es gibt alternativ auch andere Möglichkeiten mittels der Mechanismen aus der Java Standard Etition die Kommunikation zwischen den Threads zu realisieren. So kann eine eigene Nachrichteninfrastruktur geschaffen werden (vgl. Kapitel 1.x MessageQueue) oder es können auch Java Futures genutzt werden. Dennoch ist es für dieses Beispiel sinnvoll sich der Android-spezifischen Funktionalität zur Kommunikation über Thread Grenzen zu bedienen, denn diese wird bereits durch das Android Betriebssystem für jede Applikation bereitgestellt. Eine eigene Implementierung ist daher zwar jeder zeit möglich, aber stellt eine unnötige Redundanz dar. Diese Funktionalität wird im nächsten Abschnitt kurz erläutert um weiter auf deren korrekte Anwendung eingehen zu können.

2.2.1. Handler-Looper Mechanismus zur Inter-Thread-Kommunikation

Das Android SDK bietet für die Kommunikation zwischen Threads, die der selben Applikation angehören den Handler Looper Mechnismus an.

Damit ist es möglich einen Thread an die applikationsinterne Nachrichteninfrastruktur einzuhängen und mit dem Haupt Thread, also hier dem UI Thread kommunizieren. Die Kommunikation ist dabei unidirektional. Hier gelangen Nachrichten nur von den sekundären Threads über die Nachrichtenstruktur zum UI-Thread und nicht umgekehrt. Die Nachrichtenstruktur wird in Form einer Nachrichtenwarteschlange umgesetzt (einer Instanz vom Typ MessageQueue), die nach dem First In First Out Prinzip durch ein Objekt vom Typ android.os.Looper abgearbeitet wird. Für die Instantiierung des Handlers gilt, dass er stets nur aus dem Thread erzeugt werden darf, der auch über eine valide Looper Instanz und somit über eine MessageQueue Instanz verfügt. Dies ist per Default immer der UI-Thread.



Die o.g. MessageQueue und die Looper-Instanz wird zum Initialisierungszeitpunkt der Applikation erzeugt. Sie bildet über die Kommunikation zwischen sekundären Threads und dem UI Thread hinaus, auch die Kommunikation zwischen den Basiskomponenten der Applikation ab (siehe Kapitel 1.3.3 Komponentenmodell). Darunter zählen u.a. Activities, Broadcast Receivers, etc.

2.2.2. Verwendung des Handler Looper Mechnismus

In unserem Beispiel wird ein Handler aus der Activity erzeugt, die in dem UI-Thread verarbeitet wird. Der Hander kann somit in seinem Konstruktor auf die Looper Instanz des UI Threads zugreifen und damit auch die Referenz der MessageQueue abrufen, in die er Nachrichten ablegen soll. In der run()-Methode der Klasse RandomPrimeNumGenerator wird das Ergebnis der Berechnung als gebundenes Schlüssel-Wert-Paar in einem Nachrichtenobjekt vom Typ anroid.os.Message abgelegt und der Handlerinstanz als zu versendende Nachricht übergeben.

```
@Override
public void run()
{
Log.d(TAG, "Call run");
String result = startCalculation();
Message message = new Message();
Bundle bundle = new Bundle();
bundle.putCharArray(String.valueOf(targetView.getId()),
result.toCharArray()); message.setData(bundle);
Log.d(TAG,"Call handler");
handler.sendMessage(message);
}
```

Die Handlerimplementierung in der Activity liefert die Rückruf Methode `handleMessage()`, die der Looper aufruft, wenn er die Nachricht unserer Handlerinstanz aus der Nachrichtenschlange nimmt.

```
private static Handler HANDLER = new Handler()
{
@Override
public void handleMessage(Message msg)
{
updateView(msg);
}
};
```

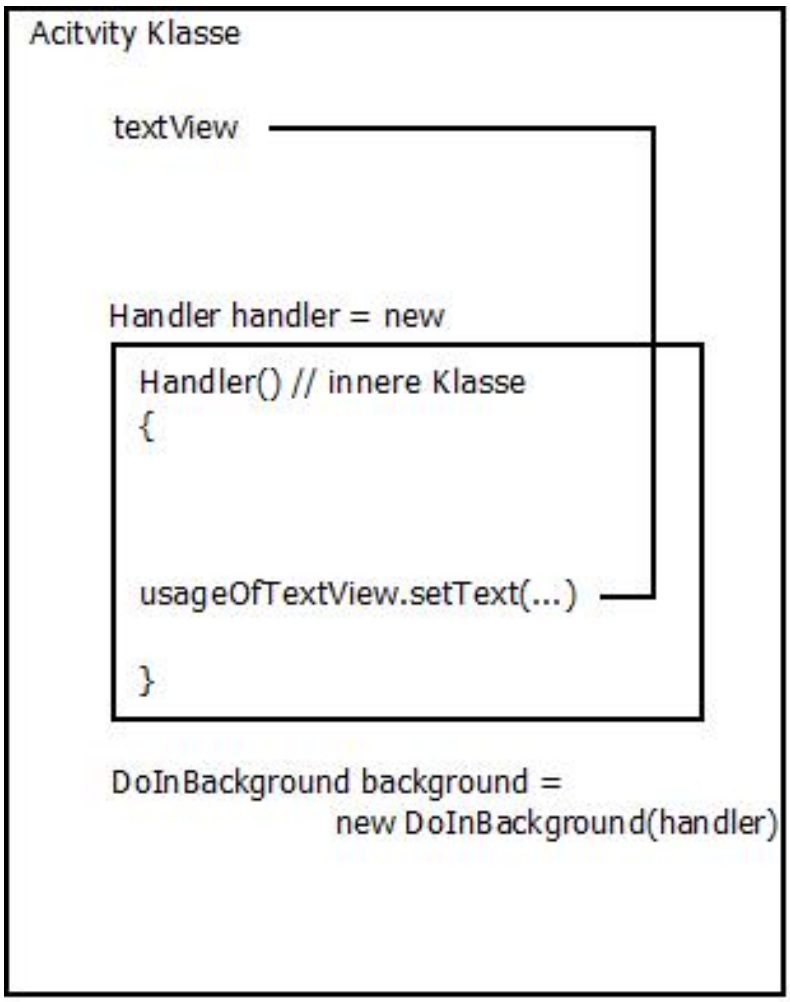
Im Falle von mehreren Handlerinstanzen, die Nachrichten in die MessageQueue ablegen, gibt es keine Kontrolle über die Reihenfolge. Die Nachrichten werden durch den Looper in der Regel (abhängig von der MessageQueue Implementierung) nach dem First In First Out Prinzip abgearbeitet, welches für komplexere Handler – Thread Konstrukte (also mehr als ein sekundärer Thread) zu beachten ist. Die Methode updateView(msg) der Activity wird im weiteren Programmverlauf nun asynchron aufgerufen und extrahiert die Nutzdaten aus der übermittelten Nachricht, um damit die jeweiligen Interaktionselemente der Benutzerschnittstelle zu aktualisieren.

```
public static void updateView(Message aMessage)
{
Log.d("RandomPrimeNumGenerator", "Callback handleMessage");
Bundle bundle = aMessage.getData();

if(bundle.containsKey(String.valueOf(R.id.startCalculation1)))
{
char[] firstResult = (char[])bundle.get(String.valueOf(R.id.startCalculati
Log.d("RandomPrimeNumGenerator", "Callback view string: "+
String.valueOf(firstResult));

firstCalculationOutput.setText(String.valueOf(firstResult));
firstCalculationOutput.invalidate();
}

if(bundle.containsKey(String.valueOf(R.id.startCalculation2)))
{
char[] secondResult = (char[])bundle.get(String.valueOf(R.id.startCalculat
Log.d("RandomPrimeNumGenerator", "Callback view string: "+
String.valueOf(secondResult));

secondCalculationOutput.setText(String.valueOf(secondResult));
secondCalculationOutput.invalidate();
}
```

```
}
```

2.2.3. Probleme bei der Nutzung des Handler Looper Mechanismus

Das o.g. Implementierungsbeispiel zeigt wie Parallelverarbeitung mit der Standard Thread Erzeugung aus dem `java.util.concurrency` Paket realisiert werden kann. Jedoch besteht so wie der Handler Looper Mechnismus hier verwendet wird ein Risiko. Dadurch, dass durch den UI-Thread eine Handler Instanz erzeugt wird und diese dann im sekundären Thread genutzt wird, können Memory Leaks (Speicherlecks) entstehen die, die Stabilität des Systems gefährden können. Innerhalb von Applikationen unter Android werden in der Regel für bestimmte Ereignisse wie z.B. Konfigurationsänderungen (Drehen des Bildschirm = Veränderung des Darstellungsformates) die betreffenden Activities neu instantiiert und die alten Instanzen dem Garbage Collector übergeben. Die Folgende Graphik skizziert die Abhängigkeiten bei einem möglichen Implementierungsszenario:



In diesem Fall kann die alte Activity Instanz nicht entsorgt werden, da diese immer noch eine Referenz auf das aktuell in einem zweiten Thread genutzte Handler Objekt hält. Zusätzlich wird innerhalb der Handler Klasse auf ein Element der Activity zugegriffen. Beide Referenzen hindern den Garbage Collector daran die Activity abzuräumen, solange die Verarbeitung im sekundär Thread noch läuft. Dauert die Hintergrundverarbeitung nun sehr lange, so ist es durch häufiges Drehen des Displays schnell möglich, weitere Activity Instanzen zu erzeugen und somit das System zu destabilisieren (der Hauptspeicher läuft voll). Dieser Effekt wird noch dadurch verstärkt, dass gerade die Activity alle GUI Elemente referenziert, also sehr speicherintensiv ist. Der Konstruktor der Klasse android.os.Handler versucht bereits zur Instanziierung den Entwickler vor diesem Risiko zu warnen:

```
/**
* Default constructor associates this handler with the
queue for the current thread.
* If there isn't one, this handler won't be able to receive
messages.
*/
public Handler() {
if (FIND_POTENTIAL_LEAKS) {
final Class<? extends Handler> klass = getClass();
if ((klass.isAnonymousClass() || klass.isMemberClass() ||
klass.isLocalClass()) && (klass.getModifiers() & Modifier.STATIC)
== 0) {
Log.w(TAG, "The following Handler class should be static or
leaks might occur: "+ klass.getCanonicalName());
}
}

mLooper = Looper.myLooper();
if (mLooper == null) {
throw new RuntimeException( "Can't create handler inside
thread that has not called Looper.prepare()");
}
mQueue = mLooper.mQueue;
mCallback = null;
}
```

Hier wird deutlich, dass explizit geprüft wird, ob das Handler Objekt als statisches Objekt aus einer inneren Klasse heraus deklariert wurde. Falls nicht wird schon hier eine Warnung ausgegeben, dass an dieser Stelle ein Speicherleck droht:
`Log.w(TAG, "The following Handler class should be static or leaks might occur: "+ class.getCanonicalName());`
Für die Handler Referenz würde somit schon die Definition als statische Instanz ausreichen. Werden jedoch dem sekundär Thread weitere Referenzen übergeben sind diese als WeakReferences zu kapseln. Diese zeigen dem Garbage Collector an, dass er bei der Evaluation z.B. einer Activity

Instanz nicht die aktuelle Nutzung von WeakReferenzes untersuchen muss.

2.2.4. Vorsicht im Umgang mit Java Futures

Als Nachtrag zu dem Implementierungsbeispiel aus dem letzten Abschnitt, wird auf die berechtigte Fragestellung eingegangen, ob nicht auch Java Futures für den Informationsaustausch, bzw. die Ergebnisübergabe genutzt werden können anstatt sich dem Handler Looper Mechanismus zu bedienen. Java Futures bieten zwar grundsätzlich die Möglichkeit asynchrone Verarbeitung zu realisieren, jedoch eignen sie sich in diesem Anwendungszenario eher weniger für den Austausch von Nachrichten bzw. Rückgabe des Ergebnisses aus der Hintergrundberechnung. Denn wird für den Erhalt eines Ergebnisses vom UI Thread Future.get() aufgerufen so, wird der UI Thread durch die get()-Methode blockiert, solange angeforderte Ergebnis noch nicht vorliegt. Dies ist gerade das Verhalten was verhindert werden muss. Alternativ kann ein Konstrukt entworfen werden, in der kontinuierlich oder in bestimmten Zeitabständen versucht wird das Ergebnis abzufragen, doch diese Lösung ist weder elegant noch performant. Entsprechend eignet sich die Java Standart Implementierung des Future eher weniger für die in dieser Arbeit fokussierten Problemstellung.

2.3. Parallelverarbeitung mit AndroidAsyncTask (Anroid Concurrency)

Im Vorangegangenen Abschnitt wurde gezeigt, das die Realisierung mittels Java Concurrency und dem Handler Looper Konstrukt durchaus einen gewissen Komplexitätsgrad und damit auch einige Gefahren für eine stabile Anwendung birgt. In diesem Abschnitt wird ein Konzept von Google vorgestellt, welches die Realisierung von nebenläufiger Verarbeitung deutlich vereinfachen soll. Der Schlüssel hierzu ist die abstrakte Klasse android.os.AsyncTask. Sie stellt eine Hilfsklasse zu dem oben beschriebenen Handler Looper Mechanismus dar und muss für die Verwendung abgeleitet werden. Dabei sind drei generische Primärparametertypen zu

definieren, welche die Nutzdatentypen zur Initialisierung, Durchführung, und Ergebnisrückgabe der Hintergrundberechnung konkretisieren.

`private class AsyncTaskImpl extends AsyncTask<Params, Progress, Result>`
Für Parametertypen gilt:

- Params → betrifft alle Parameter, die Nutzdaten für die Hintergrundverarbeitung zu dessen Ausführung mittels der Methode `doInBackground(Params... params)` übertragen.
- Progress → betrifft alle Parameter, die während der Ausführung mittels der Rückruf Methode `onPublishProgress(Progress... progress)` den Fortschritt, bzw den Status der Hintergrundverarbeitung an den UI-Thread transportieren.
- Result → betrifft alle Parameter, die das Ergebnis aus der Hintergrundverarbeitung mittels der Rückruf Methode `onPostExecute(Result ...result)` an den UI-Thread übergeben.

Die Android Developper Dokumentation nennt weiter die vier wesentlichen Methoden zur Steuerung der Hintergrundverarbeitung:

- `onPreExecute()`
- `doInBackground(Params...params)`
- `onProgessUpdate(Progress... progress)`
- `onPostExecute(Result... result)`

Die Methoden teilen das Ausführungsmodell der Verarbeitung im AsynchTask in klar voneinander abgetrennte Abschnitte (siehe auch Sequenzdiagramm in Abbildung xx). Die Methodennamen geben einen ersten Hinweis auf die jeweiligen Aufrufzeitpunkte, die in der Klasse AsyncTask fest definiert sind. So wird die Methode

`onPreExecute()`
vor der Hintergrundverarbeitung bereits im Konstruktor der Klasse AsyncTask aufgerufen. Hier kann entsprechend alle Logik integriert

werden, die noch vor der eigentlichen Hintergrundverarbeitung stattfinden soll. Der hier definierte Code wird noch auf dem UI-Thread ausgeführt. Dies bietet sich an für Initialisierungen wie z.B. für eine Prozessanzeige. Die Methode

`doInBackground()`

ist als abstrakte Methode gekennzeichnet und muss bei der Vererbung von AsyncTask überschrieben werden. Sie kapselt die aufzurufende Logik um diese in einem separaten Thread aufrufen zu können. In unserem Beispiel wird hier die zeitintensive Berechnung definiert. Die `doInBackground()`-Methode ist eine Rückruf-Methode und wird indirekt angestoßen durch den Aufruf

`asyncTaskInstance.execute(Params...params)`
Die Klasse `AndroidAsyncRandomPrimeGen` überschreibt die `doInBackground()`-Methode und integriert hier die zeitintensive Verarbeitung: `@Override protected AsyncTaskResult<String> doInBackground(Integer... params)`

```
{
if(params == null || params.length != 1)
{
return new AsyncTaskResult<String>(new IllegalArgumentException("Not
the rights params:"+ params));
}
triggerViewId = params[0].intValue();

StringBuilder targetString = new StringBuilder(10);
for(int i = 0; i < 10; i++)
{

BigInteger veryBig = new BigInteger(1500, new Random());
//Zeitintensive Berechnung
//siehe Bsp BlockingEingabeAusgabeBsp...
return new AsyncTaskResult<String>(targetString.toString());
}
```

Die oben beschriebenen Übergabeparametertypen werden in diesem Codebeispiel zur Methode `doInBackground()` als Integer konkretisiert und spezifizieren in der Implementierung die ID der View, welche die Berechnung angestoßen hat. An Hand dessen wird später die Ziel-View ermittelt, die das Ergebnis darstellen soll. Das Ergebnis der Berechnung wird hier als konkreter `AsyncTasResult<String>` Typ zurückgegeben, damit dieser String an die `onPostExecute(Result ... result)` Methode weitergegeben werden kann. Analog zur Hintergrundverarbeitung bietet Google mit der Methode `onProgressUpdate()` die Möglichkeit aus der laufenden Berechnung im sekundären Thread, Informationen oder Nachrichten wie z.B. Statusmeldungen an den UI Thread zu senden. Die Methode `onPostExecute()` wird nach Abschluss der Hintergrundoperationen dann wieder auf dem UI-Thread ausgeführt. Sie bietet sich an, um finale Aktualisierungen auf Basis der Ergebnisse aus der Hintergrundverarbeitung durchzuführen. In unserem Beispiel könnte das Ergebnis in die jeweiligen View Objekte der Activity übertragen werden:

```
@Override
protected void onPostExecute(AsyncTaskResult<String> result)
{
...
//update user interface
String realResult = result.getResult();

if (triggerViewId == R.id.startCalculation1) {
firstOutputView.setText(realResult);
firstOutputView.invalidate();
}
if (triggerViewId == R.id.startCalculation2) {
secondOutputView.setText(realResult);
secondOutputView.invalidate();
}
}
```

Da einzig und allein von den vorgestellten Methoden, die `doInBackground()` Methode in dem sekundären Thread ausgeführt

wird, muss darauf geachtet werden, keine zeitintensive Logik in den anderen Methoden zu integrieren. Ansonsten besteht wieder die Gefahr der blockierenden Anwendung, analog zu dem o.g. Beispiel zur Blockierenden Anwendung. Weiter ist gemäß der Android Developer Dokumentation darauf zu achten, dass nur gering zeitaufwändige (wenige Sekunden) Operationen mittels des AsyncTask in eine Hintergrundverarbeitung ausgelagert werden sollten. Woran das liegt und welche Konsequenzen sich aus längeren Operationen ergeben wird hier nicht genannt. Die Android Developper Dokumentation gibt hierzu lediglich einen groben Überblick wie dieser optimal zu nutzen ist und für welche Problemstellungen sich die Verwendung der Klasse AsyncTask eignet. Es bleiben also weitere Fragen offen:

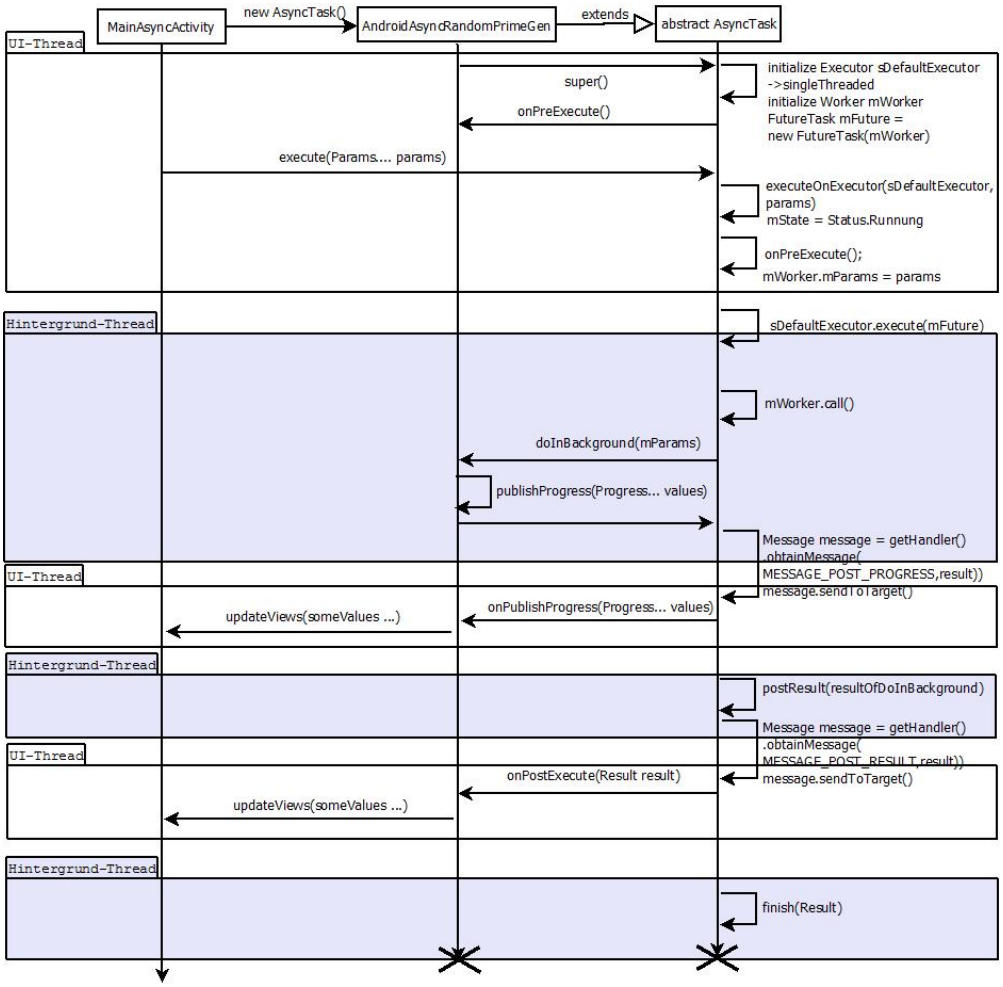
- Wie funktioniert nun der AsyncTask Mechanismus konkret?
- Gemäß der Dokumentation wird die Inter-Thread Kommunikation mittels des oben vorgestellten Handler-Looper Mechanismus (siehe auch Parallelverarbeitung mit Java SE) realisiert. Doch wie wird diese Kommunikation im Detail für die genannten Rückrufmethoden über Thread Grenzen hinaus umgesetzt?
- Wie und wann wird der Hintergrund Thread erstellt?
- Wie funktionieren die Executor Instanzen insbesondere der `THREAD-POOL-EXECUTOR` der in der Android Developper Dokumentation lediglich kurz genannt wird?

Diesen Fragen widmet sich der Folgende Abschnitt in dem genauer auf die konkrete Implementierung der Klasse andorid.os.AsyncTask eingegangen wird.

2.3.1. Ausführungsmodell von Android AsyncTask in Bezug auf Multi Threading

Im letzten Abschnitt wurde auf die allgemeine Funktionsweise der Klasse android.os.AsyncTask eingegangen, ohne genauer zu hinterfragen wie die einzelnen Mechanismen im Hintergrund funktionieren. Dabei blieben noch einige Fragen offen, deren Klärung nun ein genauerer Blick in die

Implementierung der abstrakten Klasse AndroidAsyncTask erfordert. In einem ersten Schritt wird zunächst das Ausführungsmodell gemäß der im letzten Abschnitt vorgestellten Beispielimplementierung im Detail vorgestellt. Hierzu dient das folgende Sequenzdiagramm:



Das dargestellte Sequenzdiagramm gibt einen ersten Einblick in die interne Funktionsweise der Klasse AsyncTask. Es ordnet die einzelnen Vorgänge zur Initialisierung, Durchführung und Beendigung der Hintergrundberechung im Android AsycnTask dem jeweiligen Thread zu. Es ist zu entnehmen, dass die Initialisierungen durch den Aufruf des Konstruktors angestoßen werden und im UI-Thead stattfinden. Zu diesen Initialisierungen gehören:

- Erzeugung der Standart Executor Instanz, die die Ausführung in Thread steuert
- Erzeugung einer Worker Instanz, welche die Hintergrundverarbeitung kapselt
- Erzeugung einer Queue Instanz welche die später zu verarbeitenden Aufgaben hintereinander anreiht
- Erzeugung eines Handlers der die asynchrone Kommunikation zwischen dem Hintergrund Thread und dem UI Thread gemäß des Handler Looper Mechanismus (siehe Abschnitt zu Handler Looper Mechanismus Kapitel 2.x) regelt.

„Wartend“

2.4. 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.

2.5. 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 Berührungspunk-

ten 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.

2.6. 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.

2.6.1. 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 ursprüng-

lichen 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:

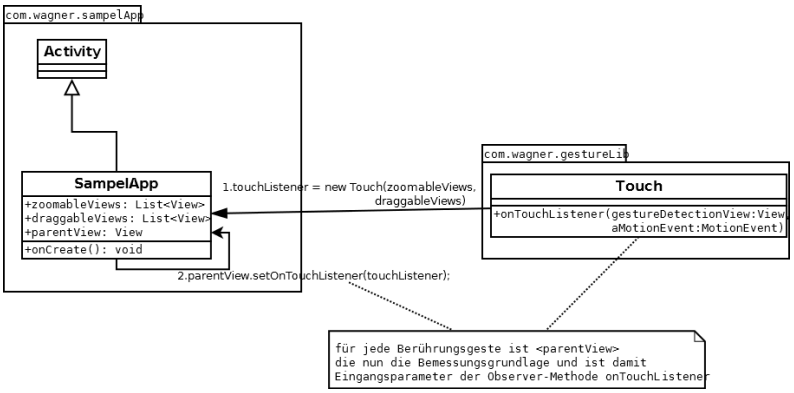


Abbildung 2.: Konzeption eines generischen Touchlisteners

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

2.6.2. 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.

2.7. 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 30. Oktober 2015

(Unterschrift)