Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Fachstudie Nr. 164

Automatisierte, quantitative Analyse von Android-Applikation-GUIs

Robin Goldberg, Hansjörg Schmauder, Benjamin Schmidt

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Albrecht Schmidt
Betreuer/in:	Dipl.-Inf. Niels Henze, M. Sc. Alireza Sahami
Beginn am:	2012-08-15
Beendet am:	2013-02-14
CR-Nummer:	D.2.2

Kurzfassung

Nach der rasanten Verbreitung klassischer Mobiltelefone zeichnet sich in den letzten Jahren ein neuer Trend ab: Immer mehr Mobiltelefone werden durch sogenannte „Smartphones“ ersetzt. Diese Smartphones bieten dem Nutzer über die klassischen Funktionen wie Telefon und SMS hinaus Zugriff auf ein breites Feld von Funktionen und Diensten, von denen viele direkt mit dem mobilen Internet verbunden sind. Der große Unterschied zwischen klassischen Mobiltelefonen und Smartphones liegt dabei vor allem im Konzept der Applications (Apps). Nutzer können eine Vielzahl unterschiedlicher Apps aus großen Apps-Stores herunterladen und installieren, um den Funktionsumfang ihres Smartphones erheblich zu erweitern. Die große Zahl der Apps auf der einen Seite und die steigende Bedeutung von Smartphones und Apps auf der anderen Seite machen dieses Feld auch für die Forschung sehr interessant. Dazu wurden Techniken entwickelt, um automatisch gefährliche Apps zu identifizieren oder Fehler in einer großen Anzahl von Apps zu finden. Im Gegensatz dazu beschäftigt sich diese Arbeit mit der automatisierten Analyse von Apps aus der Sicht der Mensch-Rechner-Interaktion. Dazu haben wir 400 beliebte Android Apps untersucht. Die Ergebnisse legen nahe, dass sich die Komplexität der Benutzungsschnittstelle je nach App-Kategorie unterscheidet. Des Weiteren haben wir die zur Gestaltung verwendeten Layout-Dateien mit dem Ziel analysiert, Elemente und Muster von Elementen zu finden, die häufig verwendet werden.

Das häufigste Muster, das wir identifizieren konnten, besteht aus drei Elementen und macht insgesamt 5.43 % aller Elemente der Layout-Dateien aus. Damit tritt es häufiger auf als einige Standard-Elemente wie beispielweise Fortschrittsbalken und Kontrollkästchen. Die zehn häufigsten Muster machen zusammen 21.13 % aller Elemente aus. Sie treten alle häufiger auf als bekannte Elemente wie Optionsfelder. Mithilfe dieser Muster lassen sich nicht nur Erkenntnisse über die derzeitige Gestaltung von App-Oberflächen gewinnen, sondern auch Ideen für neue, sinnvolle Elemente finden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Der Trend zum Smartphone ist ungebrochen, viele Menschen sind inzwischen permanent mit mobilem Internet versorgt und knnen von berall auf E-Mails, Nachrichten und andere Dienste zurckgreifen. Als zentrales Bedienelement hat sich der Touchscreen herauskristallisiert, der Anzeige und Bedienung der Smartphones zugleich darstellt. Erweitert wird die Funktionaliter Smartphones durch eine Vielzahl sogenannter Apps, kleiner Programme, welche einen bestimmten Dienst anbieten, Kommunikation ermglichen oder einfach nur unterhalten sollen. Bei tausenden von Apps, die auf den verschiedenen Marktplen erhlich sind und sich hig mit lichen Themen beschigen, spielt die Gestaltung der GUI und Interaktion eine entscheidende Rolle. Sie kann ber den Erfolg oder Misserfolg einer App entscheiden, da es im Normalfall keine Anleitungen gibt und die Nutzer auch keine Motivation haben, Hilfefunktionen zu befragen. Die App muss aber nicht nur eine gute Usability haben, also einfach und intuitiv zu bedienen sein, sondern auch Likeability ist ein wichtiges Thema. Die Nutzung der App soll Spaachen und fr den Nutzer ein Erlebnis darstellen.

Damit verbunden ist die visuelle Prntation, da direkt mit den Elementen auf dem Bildschirm interagiert wird. Je nach Art der App knnen dabei ganz verschiedene Konzepte und Elemente zum Einsatz kommen. Wend mobile Spiele versuchen, eine mglichst gute Immersion zu erreichen, mssen Nachrichten-Apps bersichtlich sein und in Kommunikationsprogrammen wird hig versucht, den Eindruck einer direkten Konversation aufrecht zu erhalten. Dies alles macht die Betrachtung der Benutzungsoberflen von Apps zu einem interessanten und lohnenswerten Feld.

Diese Arbeit stellt folgende Aspekte vor: Zunst werden in Kapitel 2 verwandte Arbeiten beschrieben, die helfen sollen, den Kontext dieser Arbeit richtig einzuordnen und einen erblick ber das Themengebiet zu bekommen. Im dritten Kapitel folgt eine Beschreibung des Android-Systems und der Funktionsweise von Apps und dem fr diese Arbeit genutzten App-Store. Kapitel 4 erkl, wie die einzelnen Apps analysiert wurden und welche Daten dabei gewonnen werden konnten. Die Vorstellung der Ergebnisse folgt in Kapitel 5, im letzten Kapitel stellen wir Folgerungen vor und geben Ideen, welche Herausforderungen auf diesem Feld noch offen sind.

2 Verwandte Arbeiten

In diesem Kapitel werden zunächst einige Arbeiten vorgestellt, die sich ebenfalls mit der Analyse von Apps beschäftigt haben. Im Wesentlichen haben wir zwei Kategorien bestehender Arbeiten zu diesem Thema identifiziert. Zum einen wurden schon einige automatisierte Untersuchungen von (Android-)Apps durchgeführt, die sich jedoch zum Großteil auf Aspekte der Sicherheit und der Korrektheit der Apps konzentrierten. Zum anderen gibt es bereits Arbeiten, die sich mit der Gestaltung und Analyse der Android-GUI beschäftigen sowie zahlreiche Richtlinien und Hinweise für die Oberflächengestaltung von Apps. Diese Arbeiten konzentrieren sich aber zumeist auf Beispielanwendungen oder wenige, ausgesuchte Apps.

2.1 Automatisierte (Android-)App-Analyse

Die ersten Arbeiten, in denen größere Mengen von Android-Apps untersucht wurden, beschäftitgen sich zum Großteil mit Aspekten der Sicherheit und dem Rechte-System der Android-Plattform. Einen einfachen Ansatz wählen dabei Barrera et al [BKOS10]. Sie extrahieren die Manifest-Dateien von 1.100 Apps, um daraus alle von den Apps benutzten Rechte auszulesen. Diese werden anschließend als Bit-Vektoren gespeichert, um darauf Analysen zu den verwendeten Rechte auszuführen. Die Apps werden dabei in Kategorien entsprechend denen im Android-Market gruppiert. Die meisten Rechte finden sich bei Apps aus dem Bereich Kommunikation, die wenigstens beim Bereich der Themes. Insgesamt wurde festgestellt, dass einige wenige Rechte wie zum Beispiel der Zugriff auf das Internet sehr häufig verwendet werden, während der Großteil nur selten zum Einsatz kommt. Zudem wurden einige Fehler wie der doppelte Aufruf von Rechten oder Aufrufe auf nicht existierende Rechte gefunden. Im Weiteren werden auch noch verschiedene Kategorien über die Ähnlichkeit ihrer Rechte verglichen, um die Nutzung weiter aufzuschlüsseln und Verbesserungsvorschläge für das Rechte-System von Android geben zu können.

Der Fokus bei der Arbeit von Felt et al. [FCH+11] liegt darin, herauszufinden, wie viele Android Apps mehr Rechte aufrufen als sie tatsächlich brauchen. Ihr Tool Stowaway untersucht, welche API-Funktionen in der App tatsächlich aufgerufen werden. Dazu liegen die Apps zunächst in Form von Dalvik-Executables vor, werden dann per Dedexer Tool dissassembled und von Stowaway mittels statischer Codeanalyse untersucht. Diese findet alle API-Aufrufe und vergleicht sie mit einer vorher erstellten Karte, auf der verzeichnet ist, welcher Aufruf zu welcher Permission gehört. Des Weiteren wurden auch Content-Providers und Intents untersucht, um hier ebenfalls die verwendeten Rechte herauszufinden. Bei der Untersuchung von 940 Apps wurde etwa ein Drittel als überpriveligiert eingestuft. Als Ursachen identifizieren die Autoren verschiedene Fehler, von der Nutzung veralteter Rechte über

Copy & Paste-Fehler bis hin zu Artefakten durch Testcode oder schlechte API-Information über die für einen Aufruf tatsächlich benötigten Rechte.

Einen Schritt weiter geht die Arbeit von Chin et al. [CFGW11]. Dort werden die Kommunikation zwischen einzelnen Android-Apps und die mit dem Android-Modell einhergehenden Risiken untersucht. Android-Apps kommunizieren über sogenannte Intents. Dabei handelt es sich um Nachrichten, die innerhalb einer App, zwischen verschiedenen Apps oder auch systemweit gesendet werden können. Intents enthalten immer einen Empfänger und optional Daten, welche zum Empfänger gesendet werden. Der Empfänger kann dabei entweder explizit angegeben oder implizit vom Android-System bestimmt werden. Zur Analyse des Intent-Verhaltens von Apps nutzt ihr Tool ComDroid statische Codeanalyse auf Basis von dissassembleten Dalvik-Executables. Es verfolgt die Intents von der Erzeugung an der Quelle bis zur Konsumierung bei einer Senke und untersucht dabei die Möglichkeit, das Intent durch unzureichende Definition von Rechten oder ungenaue Angabe von Sender oder Empfänger zu manipulieren oder Intents einer App abzufangen und die Daten darin auszulesen. Außerdem werden die einzelnen Komponenten von Apps auf die Möglichkeit hin untersucht, sie mithilfe von Intents zu beeinflussen, weil die empfangenen Nachrichten nicht sorgfältig genug geprüft und verarbeitet werden. Bei der Untersuchung von 20 Apps mit Hilfe von ComDroid wurden 34 mögliche Angriffspunkte identifiziert, 12 der untersuchten Apps wiesen wenigstens eine Verwundbarkeit auf.

Im Bereich von automatisierter App-Analyse wurden bisher hauptsächlich Aspekte wie Sicherheit und Datenschutz untersucht. In der Arbeit von Hu und Neamtiu [HN11] findet sich allerdings ein erster Ansatz zur GUI-Analyse. Sie zielt ab auf das Auffinden von typischen Fehlern in Android-Apps mittels automatisch generierter Testfälle. Dies erfordert aufgrund der Android-GUI-Architektur eine neue Vorgehensweise verglichen mit etablierten Testmethoden. Die Aktivitäts- und Ereignis-Struktur von Android-Apps lässt sich nicht mit bisherigen GUI-Analyse-Tools abbilden, dazu ist eine Kombination von Tools nötig. Zum Test der GUI wurden hierbei JUnit zur automatischen Testfallgenerierung und Monkey, ein Tool zur Generierung von Android-UI-Events, genutzt. Die Daten der Tests wurden in das System-Log geschrieben, welches anschließend ausgewertet wurde. Es ließen sich mit dieser Methode sowohl bekannte Fehler entdecken als auch bisher unbekannte Fehler auffinden.

Zusätzliche dynamische Codeanalyse zur Einschätzung der Gefahr durch eine App stellen Bläsing et al. [BBS+10] in ihrer Arbeit vor. Neben der statischen Codeanalyse führen sie die untersuchte App in einer Sandbox aus, um ihr tatsächliches Verhalten zu beobachten. Dabei geht es vor allem darum, Apps zu identifizieren, welche die Privatsphäre des Nutzers verletzen könnten oder Malware enthalten. Bei der statischen Analyse wird nach verdächtigen Mustern im Code gesucht. Als verdächtige Muster werden vor allem Versuche angesehen, den normalen Ablauf im Android-System durch direkte Aufrufe von nativem Code in Bibliotheken oder direkter Codeausführung zu beeinflussen oder per Reflection Beschränkungen der API zu umgehen. Während der dynamischen Analyse werden alle Aktionen der App aufzeichnet. Das dabei entstandene Log kann dann entweder manuell oder automatisiert untersucht werden. Den Einsatzbereich sehen die Autoren bei Prüfungen im Marktplatz ebenso wie für Anti-Virus-Tools auf dem Android-Gerät.

Szydlowski et al. [SEKV12] beschäftigen sich in ihrer Arbeit auch mit dem Problem der Sicherheit, sie betrachten allerdings iOS-Apps. Im Fokus ihrer Arbeit liegt die Herausforderung der dynamischen Analyse von Apps, die hauptsächlich über ihre GUI gesteuert werden. Das Ziel ihrer Analyse ist es, potentiell gefährliche Apps zu identifizieren. Dass die Einbeziehung der GUI nötig ist, zeigen ihre Angaben zur Codeüberdeckung: Ohne Einbeziehung der GUI liegt sie bei 16 %, mit GUI-Analyse bei 69 %. Im Gegensatz zu den vorher vorgestellten statischen Analyseverfahren wird bei dynamischen Analysen die App direkt bei ihrer Ausführung beobachtet. Da es sich um eine grundlegende Arbeit handelt, wurden keine richtigen Apps untersucht, sondern nur die Tools anhand einer Beispiel-App getestet. Dabei wurde gezeigt, dass die Verwendung der GUI absolut notwendig ist, um einen Großteil der Funktionsaufrufe einer App tatsächlich zu finden und damit auf eine ähnliche Abdeckung wie bei statischer Codeanalyse zu kommen. Der Vorteil der dynamischen Analyse besteht in dem Fall darin, auch Code untersuchen zu können, bei dem die statische Analyse scheitert, weil der Code maskiert wurde.

2.2 Mobile Interaktion

Fortwährend beschäftigen sich Forscher mit der Ergründung, Datensammlung und Beobachtung über das Nutzungsverhalten von Anwendern einer App. Cui und Roto betrachteten, wie Anwender das mobile Internet nutzen [CR08] und fanden dabei heraus, dass die Dauer der Web-Nutzung zwar kurz ist, aber mehr Zeit im Browser verbracht wird, wenn die Benutzer per WLAN verbunden sind. Böhmer et al. führten eine groß angelegte Studie mit einem genauen Protokoll über die Anwendungsnutzung bei Android-Apps durch [BHS+11]. Basierend auf grundlegenden und kontextuellen Statistiken entwickelten sie das Vorschlagsystem „Appazaar“ [BBK10]. Möller et al. untersuchten das Update-Verhalten und die Auswirkungen auf die Sicherheit im Google Play Market [MDR+12]. Sie berichten, dass Nutzer dazu neigen, Updates auch eine Woche nach Erscheinung der Aktualisierung noch nicht zu installieren. Rahmati et al. führten eine Longitudinalstudie durch und verglichen, wie Benutzer aus unterschiedlichen sozialen und wirtschaftlichen Verhältnissen („SES“) sich an neue Smartphone-Technologien und deren Installation bzw. Nutzung anpassen [RTS+12]. Sie zeigen, dass Nutzer aus einem Umfeld von niedrigerem Status mehr Geld für Anwendungen ausgeben und auch mehr Anwendungen installieren. Die Gruppe mit dem niedrigsten Status schätzt die Usability ihrer iPhones verglichen mit den anderen Gruppen am schlechtesten ein. Die Nutzung der Smartphone-Dienste wird von 14 jugendlichen Nutzern überprüft [RZ12]. In der Feldstudie wird berichtet, dass sie ihre Smartphones zu verschiedenen sozialen Zwecken einsetzen und sehr von der Mobilität profitieren, sodass sie es auch unterschiedlich einsetzen, je nachdem, wo sie sich befinden. Verkasalo [Ver09] zeigt außerdem, dass Nutzer bestimmte Typen mobiler Dienste in bestimmten Kontexten verwenden. So werden Browser- und Multimedia-Anwendungen hauptsächlich eingesetzt, wenn die Nutzer unterwegs sind, Spiele hingegen öfters dann, wenn sie zu Hause sind. Balagtas et al. bewerten verschiedene UI-Designs und Eingabetechniken für Touchscreen-Mobiltelefone [BFFH09].

Einige Veröffentlichungen beschäftigen sich auch mit dem Nutzerverhalten bei der Verwendung und Interaktion mit Apps auf mobilen Geräten. Henze et al. evaulierten die

Genauigkeit von Berührungsinteraktion bei mobilen Apps [HRB11] und leiteten eine Kompensationsmethode ab, die die Nutzereingaben auf dem Display verschiebt, um Fehler zu reduzieren. Außerdem untersuchten sie das Eingabeverhalten auf virtuellen Tastaturen, wie sie bei Smartphones häufig Verwendung finden [HRB12] und schlossen dort, dass die Markierung der angeklickten Stelle mit einem einfachen Punkt die Fehlerrate bei Benutzern auf der Android-Tastatur senkt. Leiva et al. beschäftigten sich mit der Unterbrechung der mobilen Anwendung sowohl durch bewusstes Vor- und Zurückwechseln zwischen den geöffneten Anwendungen als auch durch jähe Unterbrechungen wie eingehende Telefonanrufe [LBG+12]. Sie stellen fest, dass derartige Unterbrechungen zwar tendenziell selten vorkommen, im Falle des Eintretens allerdings häufig einen signifikanten Overhead zur Folge haben.

3 Android-Architektur

Dieses Kapitel gibt einen Überblick über die Android-Architektur. Dazu wird zunächst das Android-System selbst beschrieben. Dann folgt die Beschreibung der Architektur von Android-Apps, bevor zuletzt noch der Marktplatz für Android-Apps, Google Play, vorgestellt wird. Dieses Kapitel führt damit in die wesentlichen Konzepte ein, welche nachher in der Beschreibung der Analyse verwendet werden und stellt diese in den korrekten Kontext. Es werden hauptsächlich die für diese Arbeit wesentlichen Aspekte vorgestellt[1]. Die aktuelle Version von Android ist 4.1, da aber viele ältere Geräte keine Updates auf aktuelle Versionen bekommen, sind noch viele Android 2.x-Geräte in Nutzung, die 3.x-Serie war rein für Tablets ausgelegt.

3.1 Android-System

Die Grundlage des Android-Systems ist eine spezielle Form des Linux-Kernels. Dieser enthält die Treiber für die Hardwarekomponenten des jeweiligen Gerätes wie z.B. Display, Kamera oder WiFi und kümmert sich auch um das ganze Energiemanagement des Geräts. Darauf aufbauend gibt es eine Reihe von Bibliotheken, ähnlich denen des Standard-Java, die in C oder C++ geschrieben sind und sich um Grundfunktionen der Software kümmern, beispielsweise den Zugang zu Datenbanken, Mediendarstellung, Grafiken, Verschlüsselungen und Ähnliches. Auch spezielle Versionen von Standardbibliotheken wie der libc sind hier zu finden, welche auf die Bedürfnisse der Hardware mit wenig Speicher, langsameren Prozessoren und geringem Energieverbrauch optimiert sind.

Die eigentlich Funktionalität für Entwickler und Benutzer ist im Application Framework und den Applications enthalten. Diese sind in Java geschrieben und werden von einem eigens entwickelten Tool auf den Bytecode für die Dalvik Virtual Machine (VM) von Android angepasst. Dies ermöglicht die Nutzung von normalen Entwicklungsumgebungen aus dem Java-Umfeld, der Code muss lediglich am Ende noch in das DEX-Format umgewandelt werden. Bei der Dalvik-VM handelt es sich um eine eigens für Android entwickelte Java-Ausführungsumgebung. Das Application Framework bietet Zugriff auf grundlegende Funktionen über verschiedene Manager und abstrahiert den Zugriff auf Funktionen der Standardbibliotheken und der Dalvik-VM für die Entwickler von Applications. Zu den

[1] Eine vollständige Beschreibung der Android-Architektur findet sich beipielweise auf http://developer.android.com.

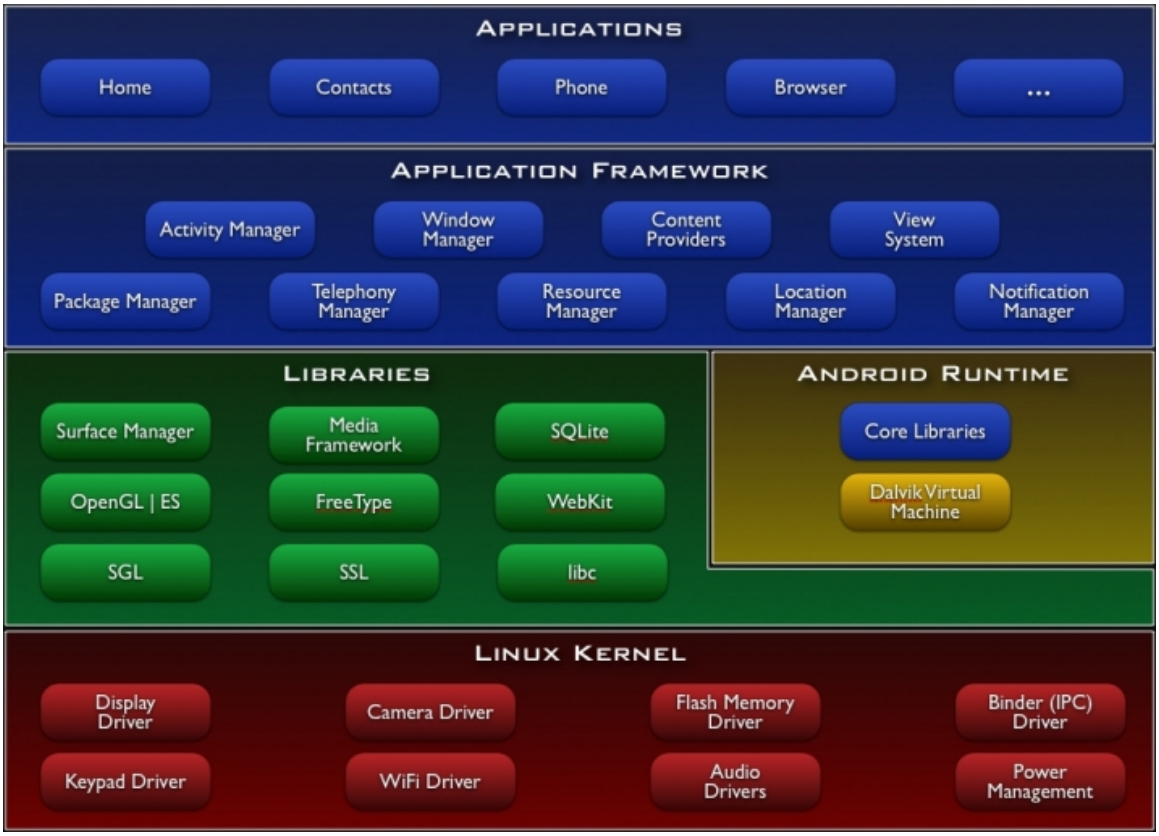

Abbildung 3.1: Android-Systemarchitektur

vorhandenen Managern zählen unter anderem der Telephone Manager, der Location Manager und der Resource Manager, aber auch der Window Manager und Activity Manager zur Verwaltung von Oberflächen und Funktionen. Einen vollständigen Überblick über die einzelnen Komponenten des Android-Betriebssystems bietet Abbildung 3.1[2].

3.2 Apps

Applications für Android werden in Java geschrieben, für die Gestaltung der Benutzeroberfläche werden XML-Dateien verwendet. Zusätzlich können noch Ressourcen in Form von Sprachdateien oder Bildern eingebunden werden. Der Ausgangspunkt jeder Application ist die Manifest-Datei. Sie ist in XML geschrieben und enthält grundsätzliche Informationen zur App. Hier wird definiert, wo der Code der App zu finden ist, welche Aktivitäten in einer App zur Verfügung stehen und welche davon als Startaktivitäten gekennzeichnet sind. Des Weiteren können in der Manifest-Datei auch Angaben zu Rechten, öffentlich Verfügbaren

[2]Quelle: http://developer.android.com/images/system-architecture.jpg

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1" android:versionName="1.0" package="com.example.androidhelloworld"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="15" />
    <application android:theme="@style/AppTheme" android:label="@string/app_name" android:icon="@drawable/ic_launcher">
        <activity android:label="@string/title_activity_main" android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Abbildung 3.2: Android-Manifest-Datei

Schnittstellen und weiteren Konfigurationen der App eingetragen werden, eine Beispieldatei zeigt Abbildung 3.2.

Die Grundlage für die Programmierung einer App stellt in Android die sogenannte Activity dar. Eine Activity stellt genau eine Bildschirmansicht dar, der Quellcode zur Funktionalität wird in einer Java-Klasse implementiert, das Layout sollte über eine XML-Datei definiert werden, es können jedoch auch Veränderungen am Layout im Quellcode der Klasse vorgenommen werden. Die verschiedenen Activitys können sich gegenseitig aufrufen und damit die gesamte Funktionalität der Application abbilden. Jede Activity ist eine Unterklasse von Activity und muss bestimmte Funktionen enthalten, die dem Android-System sagen, was zu tun ist, wenn eine App gestartet, gestoppt oder beendet wird. Auf dem Bildschirm nimmt eine Activity im Normalfall den gesamten Platz ein. Da jeweils nur eine Activity aktiv sein kann, muss jede Activity Mechanismen vorsehen, ihren Zustand zu speichern und wiederherzustellen, wenn in der Zwischenzeit eine andere Application verwendet wurde.

Häufig existieren von einer Anwendung Versionen in verschiedenen Sprachen und es gibt unterschiedliche Layouts für unterschiedliche Bildschirmgrößen, Ausrichtungen und Auflösungen. Damit nicht für jede Version eigene APKs veröffentlicht werden müssen, können unterschiedliche Sprach- und Layoutdateien in speziell benannten Ordner abgelegt werden, wo sie automatisch vom Android-System in der jeweiligen Situation ausgewählt werden. Alle diese Elemente werden schließlich zu einem komprimierten Ordner, dem Android Application Package (APK), zusammengepackt. Das APK enthält alle Ressourcen, die Manifest-Datei und den in Bytecode kompilierten Code der Activitys. Ein solches APK kann dann zum Beispiel vom Play Market heruntergeladen und installiert werden.

Die Gestaltung der Oberfläche erfolgt primär durch XML-Dateien. Dort werden die visuellen Elemente und deren Eigenschaften angegeben, die in einem Elementbaum verschachtelt sind. Jedes dieser Elemente ist von der Klasse View abgeleitet und stellt einen Bereich auf dem Bildschirm dar, mit dem der Nutzer interagieren kann. Für alle üblichen visuellen Elemente stehen bereits fertige Klassen in Android zur Verfügung. Auf diese Weise können Buttons, Textfelder und Labels einfach in das Layout eingefügt werden. Um die Anordnung kümmert sich der angegebene Layout-Manager. Im Beispiel in Abbildung 3.3 füllt dazu ein relatives Layout zunächst den gesamten Bildschirm, während die einzelnen Elemente in einem linearen Layout untereinander angeordnet werden.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout android:layout_width="fill_parent" android:layout_height="fill_parent"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <LinearLayout android:orientation="vertical" android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:layout_alignParentLeft="true" android:layout_alignParentTop="true"
    android:layout_alignParentRight="true" android:layout_alignParentBottom="true">
        <TextView android:id="@id/textView1" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="@string/hello_world" />
        <EditText android:id="@id/editText1" android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:hint="@string/enter_name" android:ems="10" android:inputType="text">
            <requestFocus />
        </EditText>
        <LinearLayout android:layout_width="fill_parent" android:layout_height="wrap_content">
            <Button android:id="@id/button1" android:layout_width="fill_parent" android:layout_height="wrap_content"
            android:text="@string/button_text" android:hint="@string/button_hint" />
        </LinearLayout>
    </LinearLayout>
</RelativeLayout>
```

Abbildung 3.3: Android-Layout-Datei

3.3 Play Store

Über den Google Play Store können Apps verbreitet werden. Die meisten Android-Smartphones haben den Zugang zum Play Store direkt integriert, es gibt aber auch Android-Geräte, die andere Stores ansprechen (z.B. Amazons Kindle). Im Play Store werden die Apps in verschiedensten Kategorien wie beipsielsweise Spiele, Büro, Kommunikation, Sport, Unterhaltung usw. eingeteilt. Zusätzlich gibt es Übersichten über aktuelle, beliebte und gut bewertete Apps. Google untersucht Apps, die im Play Store hochgeladen werden, um Schadprogramme und unzulässige Inhalte zu vermeiden. Apps können sowohl kostenlos angeboten werden wie auch über das integrierte Bezahlsystem gekauft werden. Im Play Store wird auch die Version des Android-Systems des Benutzer geprüft und es werden ihm nur Apps angeboten, die unter seiner Version lauffähig sind.

Zum jetzigen Zeitpunkt (Ende Oktober 2012) sind im Play Store etwa 700.000 App vorhanden[3]. Zugriff auf kostenlose Inhalte im Play Store besteht aus 190 Ländern; in 132 Ländern können Apps nicht nur erworben werden, sondern es können auch innerhalb der Apps Käufe getätigt werden. Etwa die Hälfte der Apps bietet solche In-App-Käufe an. Die Entwicklergemeinde rund um Android ist sehr aktiv, pro Monat werden rund 40.000 neue Apps erstellt[4].

[3]Quelle: http://www.businessweek.com/news/2012-10-29/google-says-700-000-applications-available-for-android-devices

[4]Quelle: http://web4tech.com/2012/06/27/google-play-store-statistics-updates/

4 Reverse-Engineering und Analyse

Nachdem die Komponenten und die Struktur von Android-Apps vorgestellt wurden, soll in diesem Kapitel darauf eingegangen werden, wie die Apps automatisiert heruntergeladen und analysiert werden können. Im Zuge dieser Arbeit kommen dabei zwei Werkzeuge zum Einsatz. Zum einen eine angepasste Version des Programms APKFetcher, welches in Java geschrieben ist und auf der inoffiziellen Google Market API[1] basiert. Mit diesem Tool werden die APK-Archive von Apps heruntergeladen und zusammen mit einer Textdatei mit zusätzlichen Informationen wie dem Namen, der Kategorie oder dem Zeitpunkt des Downloads abgespeichert. Das zweite Werkzeug, der APKAnalyzer, liest dann diesen Ordner aus, erstellt eine Liste der gefundenen Apps und führt die Analyse der einzelnen Apps aus. Der APKAnalyzer wurde eigens für diese Arbeit in C# implementiert. Die Analyseergebnisse werden in einer SQLite-Datenbank abgespeichert. Der Schwerpunkt liegt bei Daten, die einen direkten Bezug zur Benutzeroberfläche aufweisen, und natürlich der Benutzeroberfläche selbst. Die einzelnen Teile, ihre Funktion und die Daten, welche sie generieren, werden in den folgenden Unterkapiteln im einzelnen beschrieben.

4.1 Download

Der erste Schritt der Datengewinnung bestand darin, die Apps vom Google-Market herunterzuladen. Hierzu wurde die Google Market API eingesetzt, die in dem von Niels Henze entwickelten APKFetcher-Werkzeug verwendet wird. Mithilfe des Tools wird eine Verbindung zum Market-Server hergestellt unter Verwendung einer Android-Authentifizierung. Das beinhaltet eine Device-ID, eine E-Mail-Adresse und das zugehörige Passwort. Als Device-ID nutzten wir die Identifikationsnummer eines unserer eigenen Geräte (ein HTC Wildfire S), bei der E-Mail-Adresse und dem Passwort das mit dem Gerät verknüpfte Konto.

Der APKFetcher baut mit der Google Market API Anfragen an den Market-Server zusammen und setzt diese dann ab. Da wird die populärsten Apps analysieren wollten, erzeugten wir diesen Query dementsprechend mit verschiedenen Parametern: Gesucht wurden

- ausschließlich kostenlose Apps
- geordnet nach absteigender Popularität
- sowie weitere Informationen zu den Apps wie über das APK-Archiv hinausgehende Statistiken etc.

[1] Homepage: http://code.google.com/p/android-market-api/

Teile dieser zusätzlichen Informationen wurden zur heruntergeladenen APK-Datei geschrieben. Dabei wurde von der Google Market API keine Eigenschaft angeboten, die die Popularität der App im Google Market einordnet. Wir haben uns deshalb auf die Reihenfolge der vom Store empfangenen APKs verlassen, sodass das Ranking der Apps über einen fortlaufenden Downloadindex realisiert wurde. Die sonstigen Daten, die in die Textdatei geschrieben wurden, enthalten

- den Namen der App,
- die Kategorie, der die App zugeordnet ist,
- die Bewertung der App durch die Nutzer, die 1–5 Sterne vergeben konnten,
- die aus der Downloadreihenfolge gewonnene Platzierung im Ranking sowie
- den Zeitstempel des Downloadzeitpunkts.

Im nächsten Schritt müssen die App-Archive dekomprimiert werden.

4.2 APK entpacken

Beim Entpacken der APK-Dateien kommt das APKTool[2] zum Einsatz. Das APK-Format ist ein gepacktes Containerformat, welches allerdings eigene Kodierungen verwendet, was die Nutzung eines speziellen Werkzeugs zum Entpacken notwendig macht. Das APKTool ist in Java geschrieben und wird im Kontext des APKAnalyzer über eine Batch-Datei aufgerufen. Es entpackt die Ordner mit den verwendeten Ressourcen und wandelt die Java-Bytecode-Dateien in .smali-Dateien um. Bei diesem SMALI-Format handelt es sich um eine teilweise menschenlesbare Sprache, die benutzt wird, um aus dem Bytecode nicht den ursprünglichen Javacode erzeugen zu müssen.

4.3 Resourcen analysieren

Für die Benutzeroberfläche interessant ist im Wesentlichen der Ordner 'res', welcher alle von der Benutzeroberfläche verwendeten Ressourcen enthält. Dazu zählen Bilder und Animationen ebenso wie die XML-Layout-Dateien, welche die Oberfläche beschreiben, und Dateien, welche den Text der Oberfläche in unterschiedlichen Sprachen enthalten können. Für jede Kategorie möglicher Ordner gibt es die Standardordner „drawable“ für Bilder und Animationen, „values“ für Sprachen und „layout“ für XML-Dateien zur Beschreibung der Benutzungsoberfläche.

Es können zusätzliche Ordner angelegt werden, indem an den Ordnernamen bestimmte Suffixe angehängt werden. Für Sprachen sind das beispielsweise Ländercodes aus zwei

[2]Homepage: http://code.google.com/p/android-apktool/

Buchstaben. Ein Ordner für Text in deutscher Sprache hätte den Namen „values-de“. Zusätzlich zu Sprachen können auch bestimmte Auflösungen oder Bildschirmgrößen definiert werden, sowie Definitionen für gedrehte Bildschirme mittels der Erweiterung „-land“ codiert werden. All diese Suffixe können auch kombiniert werden, sodass zum Beispiel Ordner wie „layout-ar-xhdpi-land“ für ein Layout in arabischer Sprache für hochauflösende Geräte im Landscape-Modus stünde. Das Android-System sucht dann abhängig von der Konfiguration des Gerätes automatisch die passenden Ressourcen. Sind Ressourcen in einem speziellen Ordner nicht vorhanden, werden die Ressourcen aus dem Standardordner geladen.

Der APKAnalyzer liest zunächst alle Ordner ein, die im „res“-Verzeichnis liegen. Dann zählt er, wie viele „drawable“-, „layout“- und „values“-Ordner vorhanden sind. Drawables und Layouts werden dabei typischerweise für verschiedene Auflösungen, Größen oder Ausrichtungen definiert, wohingegen Values im Normalfall für verschiedene Sprachen definiert sind. Der APKAnalyzer sucht nach den verwendeten Auflösungen und Größen. Diese sind für Android-Apps in vier vorgegebenen Stufen definiert. Für die Größe kann small, medium, large und xlarge angegeben werden, für die Auflösung ldpi, mdpi, hdpi und xhdpi; diese fassen jeweils Gruppen möglicher Auflösungen und Größen zusammen[3]. Die gefundenen Definitionen werden in der Datenbank abgespeichert. Bei den Sprachen wird nur die Zahl der verwendeten Sprachen ermittelt, indem nach Suffixen, bestehend aus zwei Buchstaben, gesucht wird sowie dem Bindestrich-Zeichen als Begrenzer.

4.4 Manifest analysieren

Die Manifest-Datei[4] einer Anwendung beschreibt im XML-Format die Komponenten einer Anwendung. Das Manifest wurde schrittweise mithilfe von XPath analysiert, wobei die einzelnen Befunde in einer Datenbank festgehalten wurden. Der Wurzelknoten enthält bereits Informationen zum Paketpfad, unter dem später die SMALI-Dateien zu finden sind, die den Code der Anwendung enthalten und Rückschlüsse auf die Layout-Dateien erlauben. Der darunterliegende Knoten „application“ enthält neben Informationen, ob Hardwarebeschleunigung aktiv ist oder spezielle UI-Optionen gesetzt sind, auch alle in der Anwendung zur Verfügung stehenden Aktivitäten mitsamt den Informationen, wo ihre korrespondierenden SMALI-Dateien im Paket zu finden sind. Ob es sich bei einer Aktivitäten um ein Hauptaktivität, also einen Einstiegspunkt, handelt, kann man anhand des *android.intent.action.MAIN*-Attributes herrausfinden. Ist dieses gesetzt, kann die Anwendung mit dieser Aktivität starten. Zusätzlich enthält die Manifest-Datei Informationen über die verwendeten Berechtigungen, Hardwarefeatures, Bibliotheken und viele weitere.

Auf die folgenden Informationen haben wir uns innerhalb unserer Analyse konzentriert:

- Aktivitäten (activity)

[3]Eine genau Auflistung findet sich auf http://developer.android.com/guide/practices/screens_support.html.
[4]Für Informationen über die Manifest-Datei siehe http://developer.android.com/guide/topics/manifest/manifest-intro.html

- Verwendete Bibliotheken (uses-library)
- Erforderte Berechtigungen (uses-permission)
- Verwendete Features wie Beschleunigungssensor (uses-feature)
- Die verwendete SDK-Version (uses-sdk)
- Hardwarebeschleunigung (hardwareAccelerated)
- Theme (theme)
- UI-Optionen (uiOptions)

4.5 Code und Layout analysieren

Für die gefunden Referenzen auf Aktivitäten wird im nächsten Schritt versucht, die zugehörigen Layout-Dateien zu ermitteln. Um die Layout-Datei für eine Aktivität zu finden, mussten wir drei verschiedene Informationsquellen auswerten. Zum einen die Manifest-Datei ansich, da an der jeweiligen Aktivität ein Attribut hängt, das darüber Auskunft gibt, wie die SMALI-Codedatei im Paket heißt. Dann wurde die Datei „public.xml“ geparst und gespeichert. Diese Datei stellt eine Abbildung von einer abstrakten ID auf einen Pfad zu einer Datei innerhalb des Pakets im Ressourcen-Ordner dar. Innherhalb dieses Ordners liegen die Layout-Dateien. Nun folgt die Analyse der SMALI-Datei. Innerhalb dieser findet man eventuell einen Abschnitt wie:

```
const/high16 v4,32446
...
invoke-virtual {v7,v4},com/android/Test/setContentView ; setContentView(I)V
```

setContentView zeigt an, dass hier eine Sicht gezeigt wird, wofür eine Layout-Datei geladen werden muss. Um welche Layout-Datei es sich an dieser Stelle dabei handelt, findet man anhand des zweiten Parameters nach *invoke-virtual* heraus. Nachdem man in der SMALI-Datei eine Zeile mit *setContentView* gefunden hat, geht man wieder einige Zeilen zurück und versucht eine Initialisierung der Variable (in diesem Fall *v4*) zu finden. Hier wurde der Variable zuvor der Wert 32446 zugewiesen. Diesen Wert findet man in der Datei „public.xml“ wieder. Hier ist er allerdings auf 10 Zeichen mit 0 aufgefüllt. Es kann natürlich diverse Konstrukte im Code geben, die dazu führen, dass man nicht mehr ohne Weiteres *setContentView* oder die Variableninitialisierung findet. Diese Tatsache haben wir in unserer Analyse nicht weiter betrachtet, sondern nur festgehalten, wann keine Layout-Datei zu einer Aktivität gefunden werden konnte.

Sind die Referenzen auf alle Layout-Dateien in der Datenbank abgespeichert, untersucht der APKAnalyzer die gefundenen Layout-Dateien und speichert die dort gefundenen XML Elemente ebenfalls in der Datenbank ab. Dazu werden die Dateien als XML-Dokumente eingelesen und die gefundenen Elemente vom Wurzelknoten aus bis zu den Blattknoten

jeweils mit Namen und der Referenz auf den Vaterknoten und die Layout-Datei abgespeichert. Damit lässt sich für die Analyse die komplette Baumstruktur jedes Layout-Dokuments für spätere Auswertungen wiederherstellen. Gespeichert werden nur XML-Elemente, keine Attribute, da dies für Analysen bezüglich der Struktur der einzelnen Bildschirmseiten ausreichend ist. Elemente können dabei sowohl von Android definierte Standartelemente wie Layouts (LinearLayout, RelativeLayout,...) oder interaktive Elemente (Schaltflächen, Eingabefelder,...) sowie selbst definierte Elemente sein. Die selbst definierten Elemente lassen sich anhand ihres Namens erkennen. Die Bezeichnungen sind nach dem Schema von Java-Paketen benannt (z.B. „com.rgoldberg.CustomView“), daher reicht die Suche nach einem Punkt, um sie zu identifizieren. Die in Android definierten *include*-Elemente, mit denen sich eine XML-Datei in eine andere einbetten lässt, werden nicht aufgelöst sondern, nur als normale Elemente erfasst.

4.6 Datensatz

Unter Einsatz der oben beschriebenen Downloadmethode luden wir am 20. August 2012 die 400 populärsten Apps vom Google Play Store herunter und erzeugten die Informationen dafür. Für den Download gaben wir keine sprachlichen Präferenzen an. Daher waren die erhaltenenen Daten – wo durch die App unterstützt – lokalisiert in deutscher Sprache, vermutlich aufgrund der Gerätesprache, der Verbindung zum Market-Server aus bundesdeutschem Gebiet, des verwendeten Google-Accounts oder der deutschen SIM-Karte. Enthielt die App keine deutsche Übersetzung, war stets ein englischer Name angegeben. Wir nehmen an, dass dies keine Einschränkung der heruntergeladenen Apps mit sich brachte, da sich unter den Downloads auch Apps befinden, die ganz offensichtlich nicht für den deutschen Markt zugeschnitten sind, beispielsweise „FOX News“ oder „Domino's Pizza USA“.

Die Analyse der Statistiken aus den Informations-Dateien zu den APKs ergab, dass die 400 Apps in 21 Kategorien gegliedert waren. Die Kategorien mit den meisten Apps waren „Werkzeuge“ mit 14,5 % und „Kommunikation“ mit 9,2 % Anteil an der Gesamtheit der heruntergeladenen Apps. Tabelle 4.1 zeigt die weitere Aufteilung in die anderen Kategorien. Wir haben Spiele im Rahmen dieser Studie bewusst außer Betracht gelassen, da deren Benutzeroberflächen konzeptionell sehr stark von den GUIs anderer Arten von Anwendungen abweichen.

Benutzer können die heruntergeladenen Apps im Google Play Store bewerten und hierzu ihren Gesamteindruck auf einer Skala von 1–5 abgeben. Die durchschnittliche Bewertung der 400 Apps lag bei 4,25, der Median betrug 4,36 und die Standardabweichung lag bei 0,45. 80 % der Apps wurden mit mindestens 4 Punkten bewertet. Der offensichtliche Zusammenhang ist, dass populäre Apps in der Regel auch von den Benutzern hoch bewertet werden. Interessanterweise gab es einige Ausreißer unter den Apps, die besonders schlechte Bewertungen erhalten haben, beispielsweise „More for me“, eine App aus der Kategorie „Shopping“, die als 131. App in der Downloadreihenfolge heruntergeladen wurde, jedoch nur eine Bewertung von 1,88 erhielt. Die durchschnittlich höchsten Bewertungen wurden in den Kategorien „Personalisierung“, „Effizienz“ und „Werkzeuge“ vergeben; dort lagen die

Kategorie	N	Bewertung	Activitys	Layouts	Bilder
Werkzeuge	58	4.37	14.00	29.64	34.47
Kommunikation	37	4.29	36.65	88.16	65.22
Unterhaltung	34	4.17	21.41	49.68	23.56
Effizienz	34	4.38	25.32	65.38	60.29
Soziale Netzwerke	34	4.11	44.44	118.88	74.71
Musik & Audio	31	4.19	24.97	66.35	59.77
Fotografie	21	4.34	24.57	60.76	61.81
Shopping	19	4.03	36.89	106.53	56.89
Bücher & Nachschlagewerke	16	4.25	18.94	54.81	50.50
Reisen & Lokales	15	4.21	40.73	131.47	73.73
Lifestyle	14	4.30	37.57	89.43	41.43
Gesundheit & Fitness	13	4.28	50.77	93.92	55.38
Medien & Videos	12	4.34	22.92	53.08	37.75
Personalisierung	11	4.44	15.45	55.82	29.55
Nachrichten & Magazine	11	4.10	24.73	66.09	40.09
Finanzen	10	4.26	61.10	118.40	44.20
Büro	8	4.12	25.25	99.38	69.38
Wetter	8	4.24	18.13	36.25	166.50
Sport	7	3.98	40.00	136.29	46.86
Software & Demos	4	4.28	1.25	1.75	0.00
Lernen	3	3.72	39.33	88.00	78.67

Tabelle 4.1: Die Verteilung der heruntergeladenen Apps auf die verschiedenen Kategorien. Die letzten drei Spalten zeigen die durchschnittliche Anzahl der Activitys, Layouts und Bildern der Apps in der jeweiligen Kategorie.

Bewertungsdurchschnitte zwischen 4,37 und 4,44. Am anderen Ende standen die Kategorien „Shopping“, „Sport“ und „Lernen“ mit durchschnittlichen Bewertungen zwischen 3,72 und 4,03.

5 Ergebnisse

5.1 Statistiken beliebter Android-Anwendungen

Die heruntergeladenen und wie in Kapitel 4 beschriebenen analysierten APKs resultierten in einer Gesamtanzahl von 778.071 Dateien, die in insgesamt 47.706 Ordnern organisiert waren.

5.1.1 Sprachen

Wir nutzten die Ressourcen der Anwendungen, die zur Internationalisierung verwendet werden können, um zu bestimmen, welche Sprachen eine Anwendung explizit unterstützt. Bei allen Anwendungen war Englisch die Voreinstellung. Die Zahl der von den Anwendungen verwendeten Sprachen beläuft sich auf 235 unterschiedliche Sprachen, wenn man die regionalen Variationen (z.B. en_us und en_gb) mit einschließt. Im Durchschnitt unterstützt eine Anwendung 12,74 unterschiedliche Sprachen (SD = 16.42). 47 der Anwendungen unterstützen nur die voreingestellte Sprache und 56 unterstützten eine zusätzliche Sprache außer Englisch. Mehr als die Hälte der Anwendungen (216) unterstützen fünf oder mehr Sprachen. Abbildung 5.1 zeigt die zwölf am häufigsten unterstützten Sprachen. Die am häufigsten unterstützten Sprachen neben Englisch sind Chinesisch (63,8 %), Spanisch (56,6 %) und Französisch (47,6 %). Es gibt über 50 Anwendungen (12,5 %), die 30 verschiedene Sprachen unterstützen.

5.1.2 Unterstützte Anzeigen

Indem die Suffixe der Ressourcen-Dateien analysiert wurden, konnten wir herrausfinden, welche unterschiedlichen Punktdichten (dots per inch, DPI) und Bildschirmgrößen von einer Anwendung unterstützt werden. Für die Punktdichte gibt es 4 Suffixe: LDPI (Low DPI), MDPI (Medium DPI), HDIP (High DPI) und XHDPI (Extra High DPI). Die Analyse zeigt, dass nur 173 Anwendungen explizit alle vier Varianten unterstützen und 26 Anwendungen gar keine speziellen Punktdichten per Ressourcen anbieten. 93 % der Anwendungen unterstützen HDPI, 75 % MDPI, 70 % LDPI und 50 % XHDPI. Vier weitere Suffixe wurden für die Bildschirmgrößen verwendet: small, normal, large und xlarge. Die Ergebnisse zeigen, dass neun Anwendungen alle vier Größen explizit unterstützen und 215 Anwendungen keine spezielle Bildschirmgröße definieren. Die Größe large wurde am häufigsten, nämlich von 170 Anwendungen, unterstützt.

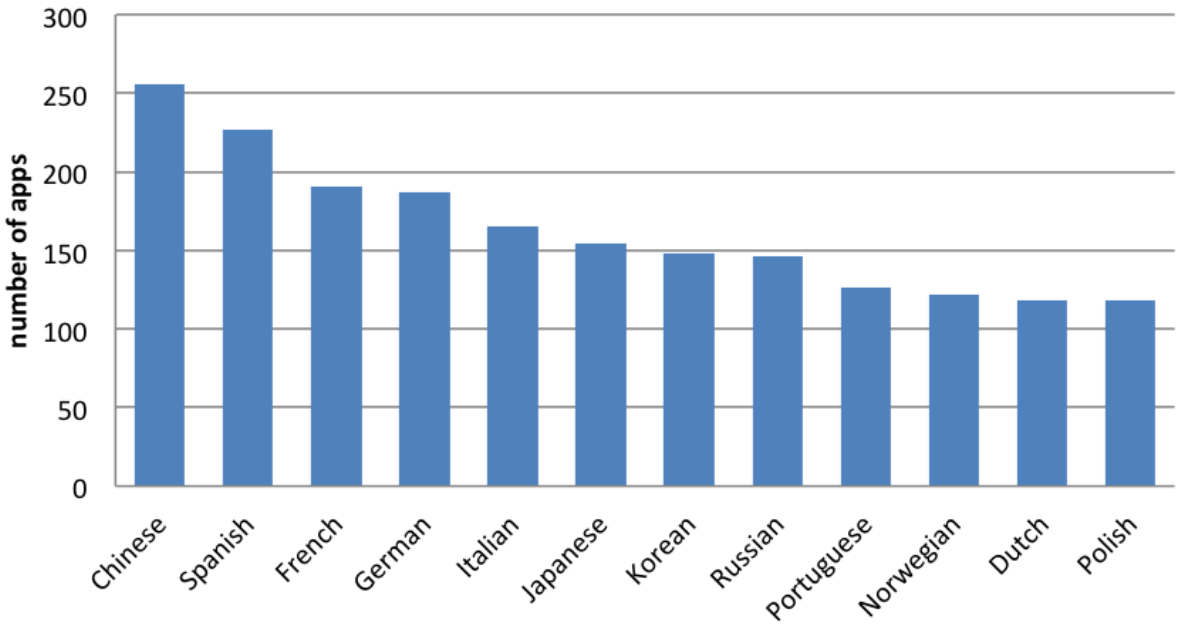

Abbildung 5.1: Die zwölf meistverwendeten Sprachen neben Englisch.

Es sollte erwähnt werden, dass ein Bildschirmgrößen- oder Punktdichten-Suffix nicht unbedingt impliziert, dass die Ressourcen nur für diese spezielle Bildschirmgröße in dieser Punktdichte geeignet sind. Falls Ressourcen für das verwendete Gerät nicht direkt zur Verfügung stehen, kann das System die Ressource, die am besten passt, selbst auswählen und verwenden. Dies könnte ein Grund dafür sein, weshalb die meisten Anwendungen nicht jede Ressource direkt anbieten, sondern die Ressourcen auf eine Art spezifizieren, die es erlaubt, sie auf einer Vielzahl von Geräten zu verwenden.

5.1.3 Einstiegspunkte

Wie bereits zuvor beschrieben, können Anwendungen mehr als einen Einstiegspunkt (Main Activity) besitzen. Die Untersuchung der Manifest-Datei zeigte, dass zehn Anwendungen keine Main-Activitys haben und 90 mehr als eine. Die „Kayak“-Anwendung hat mit 64 die höchste Anzahl an Haupteinstiegspunkten.

5.1.4 Analyse der Berechtigungen

Wir untersuchten die Metadaten, die von jeder Anwendung bereitgestellt werden, indem wir die Manifest-Datei analysierten. Insgesamt konnten wir 355 unterschiedliche Berechtigungen ($M = 11{,}2$ Berechtigungen pro App, $SD = 7{,}83$) feststellen. Insbesondere haben wir die Standardberechtigungen von Android näher untersucht, die die Anwendungen benötigen.

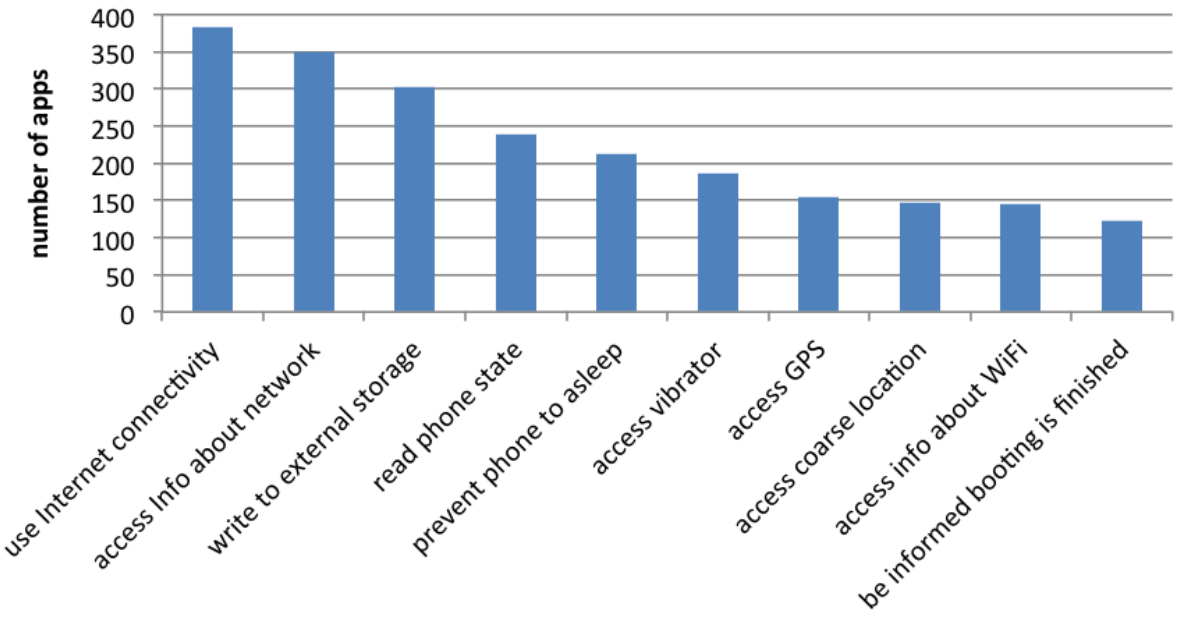

Abbildung 5.2: Die häufigsten Android-Standardberechtigungen.

Von den 355 Berechtigungen sind 121 Android-Standardberechtigungen ($M = 9{,}6$ Berechtigungen pro App, $SD = 6{,}6$). Abbildung 5.2 zeigt die zehn am häufigsten vorkommenden Android-Standardberechtigungen. Die drei häufigsten sind Internetzugang (8,7 % aller extrahierten Berechtigungen), die Berechtigung, um festzustellen, ob eine Netzwerkverbindung vorhanden ist (7,9 %), und die Berechtigung, Daten auf den externen Datenträger des Gerätes zu schreiben (6,8 %).

Taktiles Feedback

Die meisten Android-Geräte sind mit einem Vibrationsmotor ausgestattet, der es erlaubt, fühlbare Rückmeldung zu geben. Die Anwendung benötigt allerdings eine entsprechende Berechtigung, um den Vibrationsmotor zu aktivieren. Die Ergebnisse zeigten, dass 47,25 % der Anwendungen diese Berechtigung verlangen und somit taktiles Feedback geben können.

Standortinformationen

Wir waren außerdem interessiert an der Verwendung von kontextbezogenen Informationen, insbesondere der Standortinformationen, die von einer Anwendung ausgewertet werden können. Insgesamt können 190 Anwendungen den Standort des Gerätes abrufen. 154 Anwendungen nutzen genaue Standortinformationen durch die Verwendung des GPS-Sensors und 147 Anwendungen verwendeten eine grobe Standorterkennung (z.B. über die Mobilfunkzellen oder über sichtbare WiFi-Netzwerke). 111 Anwendungen können beide Informationen abrufen, den genauen und den groben Stadort des Gerätes.

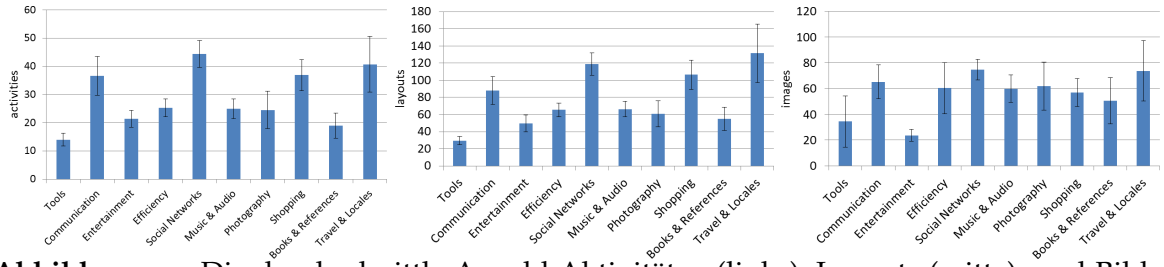

Abbildung 5.3: Die durchschnittle Anzahl Aktivitäten (links), Layouts (mitte), und Bilder (rechts) für die zehn Kategorien mit den meisten Anwendungen in unserem Datensatz. Die Fehlerbalken zeigen die Standardabweichung.

Vernetzung

96,25 % der Anwendungen verlangen Internetzugang. Weitere Analysen zeigten, dass 10,25 % der Anwendungen Bluetooth nutzen, 8,25 % haben die Berechtigung, SMS zu versenden, und 2,5 % verwenden die Nahfeldkommunikations-Möglichkeiten des Gerätes (near field communication, NFC). Es muss erwähnt werden, dass nicht alle Geräte die NFC-Technologie unterstützen.

5.1.5 Anzahl der Benutzungsschnittstellen

Um festzustellen, ob sich die Benutzungsschnittstellen von Anwendungen aus unterschiedlichen Kategorien unterscheiden, haben wir eine statistische Analyse der am häufigsten vorkommenden Kategorien unseres Datensatzes gemacht. Wegen des geringen Probenumfanges für einige Kategorien haben wir uns auf die zehn häufigsten Kategorien fokussiert ($N \geq 15$). Wir verwenden die Anzahl der Aktivitäten, die Anzahl der Layouts und die Abzahl der verwendeten Bilder als Indikator für die Komplexität einer Bunutzungsschnittstelle. Tabelle 4.1 bietet eine Übersicht über die Anzahl der Aktivitäten, Layouts und Bilder jeder Kategorie. Um festzustellen, ob sich Kategorien stark unterscheiden, haben wir eine ANOVA-Varianzanalyse durchgeführt. Im Anschluss daran haben wir ein Games-Howell-Test für den paarweisen Vergleich von Kategorien gleicher Varianz durchgeführt.

Aktivitäten

Nachdem wir die Anzahl der Aktivitäten jeder Anwendung extrahiert hatten, untersuchten wir, ob die durchschnittliche Anzahl an Aktivitäten sich zwischen den zehn häufigsten Kategorien signifikant unterscheidet. Levene's Test zeigt, dass die Homgenitätsannahme verletzt wurde ($F(9{,}289) = 3{,}89$ mit $p < 0{,}001$). Der ANOVA-Test deckte einen signifikanten Unterschied zwischen den Kategorien auf. Ein im Nachhinein ausgeführter Games-Howell-Test zeigt sechs signifikante paarweise Unterschiede zwischen den Kategorien. Anwendungen der Kategorie „Werkzeuge“ ($M = 14{,}00$; SD $= 17{,}34$) haben weniger Aktivitäten als Anwendungen der Kategorie „Soziale Netzwerke“ ($M = 44{,}44$; SD $= 27{,}64$; $p < 0{,}001$) und „Shopping“

($M = 36{,}89$; SD $= 24{,}07$; $p < 0{,}05$). Unterhaltungsanwendungen ($M = 21{,}41$; SD $= 17{,}34$) haben signifikant weniger Aktivitäten als Anwendung aus der Kategorie „Soziale Netzwerke“ ($p < 0{,}01$). Weiterhin haben Anwendung dr Kategorie „Soziale Netzwerke“ signifikant mehr Aktivitäten als Anwendungen aus „Musik & Audio“ ($M = 24{,}97$; SD $= 19{,}41$; $p < 0{,}05$) und aus „Bücher & Nachschlagewerke“ ($M = 18{,}97$; SD $= 17{,}83$; $p < 0{,}01$).

Layouts

Weiterhin haben wir die Anzahl der Layouts pro Anwendung untersucht, um statistische Differenzen zwischen den Kategorien festzustellen. Wieder einaml zeigte der Levene-Test, dass die Homgenitätsannahme verletzt wurde ($F(9{,}289) = 4{,}44$; $p < 0{,}001$). Eine ANOVA deckte signifikante Unterschiede zwischen den Kategorien auf ($F(9{,}289) = 6{,}87$; $p < 0{,}001$). Ein Games-Howell-post-hoc-Test lieferte acht signifikante paarweise Unterschiede zwischen den Kategorien. Anwendungen der Kategorie „Werkzeuge“ ($M = 29{,}64$; SD $= 35{,}52$) haben weniger Layouts als Anwendungen der Kategorien „Kommunikation“ ($M = 88{,}16$; SD $= 98{,}72$; $p < 0{,}05$), „Effizienz“ ($M = 65{,}38$; SD $= 45{,}99$; $p < 0{,}01$), „Soziale Netzwerke“ ($M = 118{,}88$; SD $= 76{,}21$; $p < 0{,}001$), „Musik & Audio“ ($M = 66{,}35$; SD $= 51{,}13$; $p < 0{,}05$) und „Shopping“ ($M = 106{,}53$; SD $= 74{,}39$; $p < 0{,}01$). Anwendungen der Kategorie „Unterhaltung“ ($M = 49{,}68$; SD $= 57{,}74$) haben weniger Layouts verglichen zu Anwendungen der „Sozialen Netzwerke“ ($p < 0{,}01$). Effizienz-Anwendungen haben ebenso weniger Layouts als die der Kategorie „Soziale Netzwerke“ ($p < 0{,}05$).

Bilder

Weiterhin haben wir die durchschnittliche Anzahl der Bilder verglichen. Levenes Test zeigt, dass die Homogenitätsannahme nicht verletzt wurde ($F(9{,}289) = 0{,}64$; $p = 0{,}77$). Auch eine ANOVA zeigte keine signifikanten Unterschiede zwischen den Kategorien. Folglich verzichteten wir auf eine post-hoc-Analyse.

Zusammenhänge

Wenn man sich die Diagramme aus Abbildung 5.3 genauer anschaut, kann man daraus schließen, dass es eine eventuelle Korrelation zwischen der Anzahl der Aktivitäten, Layouts und Bilder der Anwendungen gibt. Also untersuchten wird die Zusammenhänge zwischen der Anzahl der Aktivitäten, Layouts und Bilder. Der Pearson-Korrelationskoeffizient zeigt, dass signifikante paarweise Zusammenhänge zwischen allen drei Parametern existieren. Es gibt eine starke Korrelation zwischen der Anzahl der Aktivitäten und der Anzahl der Layouts ($r = 0{,}79$; $p < 0{,}0001$). Weiterhin gibt es eine Korrelation zwischen der Anzahl der Aktivitäten und der Anzahl der Bilder ($r = 0{,}29$; $p < 0{,}0001$) sowie zwischen der Anzahl der Layouts und Anzahl der Bilder ($r = 0{,}39$; $p < 0{,}0001$). Es ist nicht überraschend, dass eine Anwendung mit einer großen Anzahl an Aktivitäten auch über eine große Anzahl an Layouts verfügt. Diese starke Korrelation weist auf ein verbreitetes Muster hin.

5.1.6 Diskussion der Befunde

Wir fanden heraus, dass unter allen analysierten Android-Anwendungen 88,25 % weitere Sprachen außer Englisch unterstützen. Es wird außerdem eine große Bandbreite and Sprachen untersützt, die Mehrheit der Anwendungen bietet sogar fünf oder mehr Sprachen. Die Ergebnisse zeigen, dass die beliebtesten Android-Anwendungen sehr facettenreich sind in Bezug auf die Sprachunterstützung. Man kann daraus schließen, dass die Chancen, erfolgreich zu sein, höher sind, wenn eine Anwendung in vielen Sprachen lokalisiert wurde.

Wir haben die Anzahl der Aktivitäten, Layouts und Bilder einer Anwendung analysiert. Es wurde gezeigt, dass Anwendungen aus verschiedenen Kategorien signifikant unterschiedliche Anzahlen an Aktivitäten und Layouts aufweisen. Wir haben gezeigt, dass die Werkzeuge und Anwendungen aus den Bereichen „Unterhaltung“, „Effizienz“, „Musik & Audio“, „Fotografie“ und „Bücher & Nachschlagewerke“ weniger Sichten und Layouts haben als Anwendungen aus den Kategorien „Kommunikation“, „Soziale Netzwerke“, „Shopping“ und „Reisen“. Die stark lineare Korrelation zwischen der Anzahl der Aktivitäten und der Anzahl der Layouts weisen auf einen linearen Faktor hin. Außerdem ist es ungewöhnlich, für Anwendungen mehr als einen Einstiegspunkt anzubieten. Nur 20 % der Anwendungen haben mehr als eine Hauptaktivität. 96,25 %, ein überwältiger Großteil der untersuchten Anwendungen, wollen auf das Internet zugreifen und fast die Hälfte der Anwendungen (47,50 %) greifen auf Lokationsinformationen zu. Auch wenn es unterschiedliche Gründe gibt, warum eine Anwendung Zugriff zum Internet verlangt (z.B. um Werbung anzuzeigen), so kann man ebenfalls von der hohen Anzahl an Anwendungen darauf schließen, dass viele Anwendungen auf dynamischen Inhalten basieren. Anzumerken ist außerdem, dass fast die Hälte der Anwendungen (47,25 %) in der Lage ist, taktiles Feedback über den Vibrationsmotor des Mobilgerätes zu geben. Weiterhin untersützen die Anwendungen verschiedene Geräte basierend auf der Anzeigepixeldichte und weniger nach Bildschirmgrößen.

Anwendungen von verschiedenen Kategorien unterscheiden sich in Bezug auf die Benutzungsschnittstellenkomplexität. Beispielsweise haben Werkzeuge weniger Sichten und Layouts als Anwendungen der „Sozialen Netzwerke“. Werkzeuge, wie der Name vermuten lässt, befassen sich mit spezifischen Anwendungsfällen. Ein typisches Beispiel dafür ist die Anwendung *Spirit Level Plus*, die es ermöglicht, das Gerät als Wasserwage zu verwenden. Die ebenfalls niedrige Anzahl an Aktivitäten und Layouts von Anwendungen anderer Kategorien wie „Unterhaltung“, „Effizienz“, „Musik & Audio“, „Fotografie“ und „Bücher & Nachschlagewerke“ lässt darauf schließen, dass diese Anwendungen auch nur für spezifische Anwendungsfälle gedacht sind.

5.2 User-Interface-Elemente und -Muster

Wir interessieren uns für häufig genutzte Steuerelemente der Benutzungsschnittstelle und potentielle Muster für das von den Anwendungen verwendete Design. Üblicherweise wird das User Interface von Android-Apps in XML-Dateien verwaltet, die das Layout beschreiben. Diese Dateien geben an, welche Elemente an welcher Stelle stehen und welche Struktur

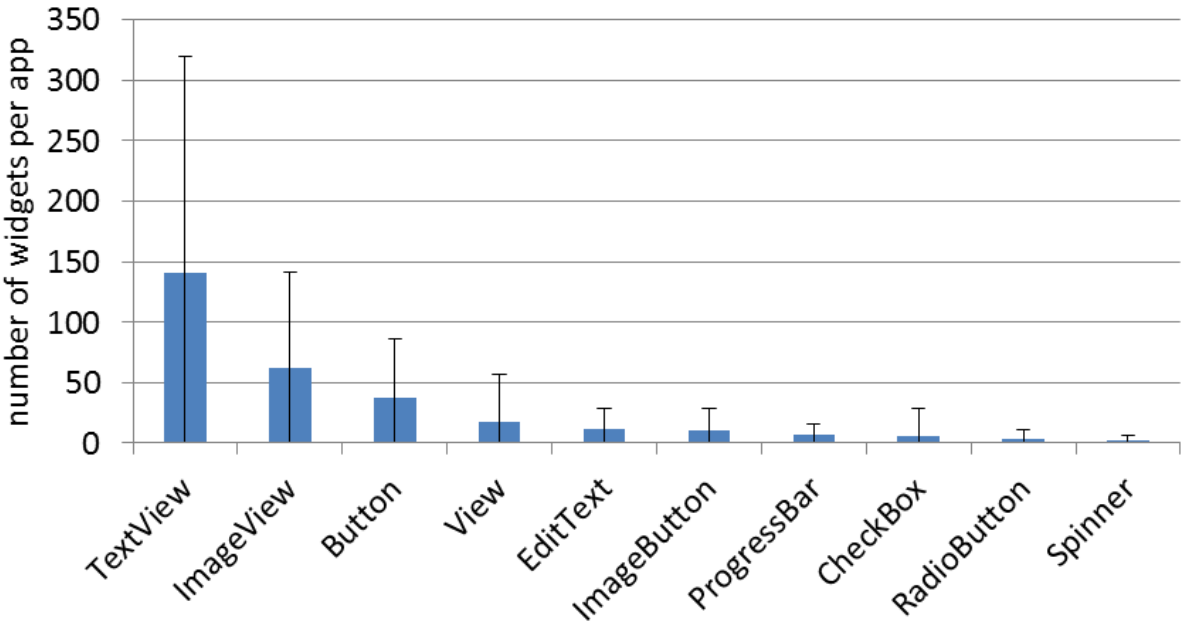

Abbildung 5.4: Die durchschnittliche Anzahl von Widgets pro App, aufgetragen für die zehn häufigsten Standard-Widgets von Android in unserem Datensatz. Die Fehlerbalken zeigen die Standardabweichung.

sie aufweisen, beispielsweise ob sie noch weitere Elemente enthalten. Im Folgenden wird ein grober Überblick über Android-GUIs gegeben. Danach werden die Ergebnisse der Studie präsentiert, die die meistgenutzten Elemente und die häufigsten Verknüpfungen von Elementen bestimmt hat. In letzterem Schritt wurden Muster gesucht, die sich aus der hierarchischen Schachtelung der Steuerelemente ergeben.

5.2.1 Layouts

Die Benutzungsschnittstelle einer Android-App besteht grundlegend aus einer Menge von Activitys; davon entspricht jede Activity einem einzelnen Bildschirm mit einem User Interface. Dieses wiederum enthält sogenannte „Widgets“, z.B. Textfelder, Kontrollkästchen oder Schaltflächen. Widgets werden innerhalb Layout-Containern verwendet, die angeben, wie die Struktur des User-Interfaces der Activity aussieht. Layout-Container können wiederum weitere Layout-Container als Elemente enthalten, wodurch für die Beschreibung der GUI eine hierarchische Struktur entsteht.

Prinzipiell ist es auch möglich, Widgets und andere Layout-Elemente direkt im Quellcode der App zu definieren, die Android Developer Guidelines empfehlen aber die Deklaration der UI-Struktur in den Layout-Dateien im XML-Format.

Die Android-API stellt einige verschiedene Layout-Container bereit, um die Zusammenstellung des User-Interfaces und den darin enthaltenen Elementen zu ermöglichen. Außerdem

Widget	Apps	Anteil	Gesamt
TextView	383	35,50 %	56467
ImageView	380	15,59 %	24794
Button	355	9,37 %	14912
View	271	4,35 %	6917
EditText	318	2,91 %	4628
ImageButton	294	2,71 %	4308
ProgressBar	300	1,67 %	2662
CheckBox	285	1,54 %	2443
RadioButton	176	0,76 %	1213
Spinner	178	0,48 %	759

Tabelle 5.1: Die zehn am häufigsten verwendeten Widgets in unserem Datensatz. Die Spalten zeigen den Name des Widgets, die Anzahl der Apps, in denen es verwendet wurde, den prozentualen Anteil gegenüber der Gesamtzahl aller gefundenen Widgets und die Gesamtzahl, wie oft das Widget gefunden wurde.

könenn Entwickler zusätzlich eigene Widgets und Layout-Container programmieren. Die fünf meistgenutzten Layout-Container werden nachfolgend kurz erläutert.

- *LinearLayout*: Die Elemente werden in einer einzelnen Spalte oder Zeile nach-/nebeneinander angeordnet.
- *RelativeLayout*: Ermöglicht die Positionierung der untergeordneten Elemente relativ zueinander oder zum beinhaltenden Layout-Container.
- *FrameLayout*: Stellt für ein einzelnes Steuerelement den im Layout-Container angegebenen Platz zur Verfügung.
- *TableLayout*: Ordnet die Elemente ähnlich dem linearen Layout-Container an, erlaubt aber die Platzierung in Zeilen und Spalten und nicht nur in einer Richtung.
- *AbsoluteLayout*: Erlaubt die absolute Positionierung der untergeordneten Elemente. Dieses unflexible Layout wird in besonderen Situationen eingesetzt, da seine Verwendung die Wartung der App erschwert.

Die Layout-Container beschreiben also die Lage der Widgets; der Benutzer der App interagiert jedoch ausschließlich mit den eingebetteten Widgets. Von der Android-API werden bereits einige gebräuchliche Widgets bereitgestellt, wie man sie aus klassischen Desktop-Anwendungen oder von Webseiten kennt. Typische Beispiele sind TextViews, die beschreibenden Text darstellen, ImageViews für Bilder, Buttons (Schaltflächen), EditTexts für die Eingabe von Text oder ProgressBars zur visuellen Fortschrittsanzeige.

5.2.2 User-Interface-Elemente

Insgesamt konnten wir aus den APKs der 400 heruntergeladenen Apps 29.086 Layout-Dateien im XML-Format extrahieren. Wir analysierten diese Dateien, um die am häufigsten

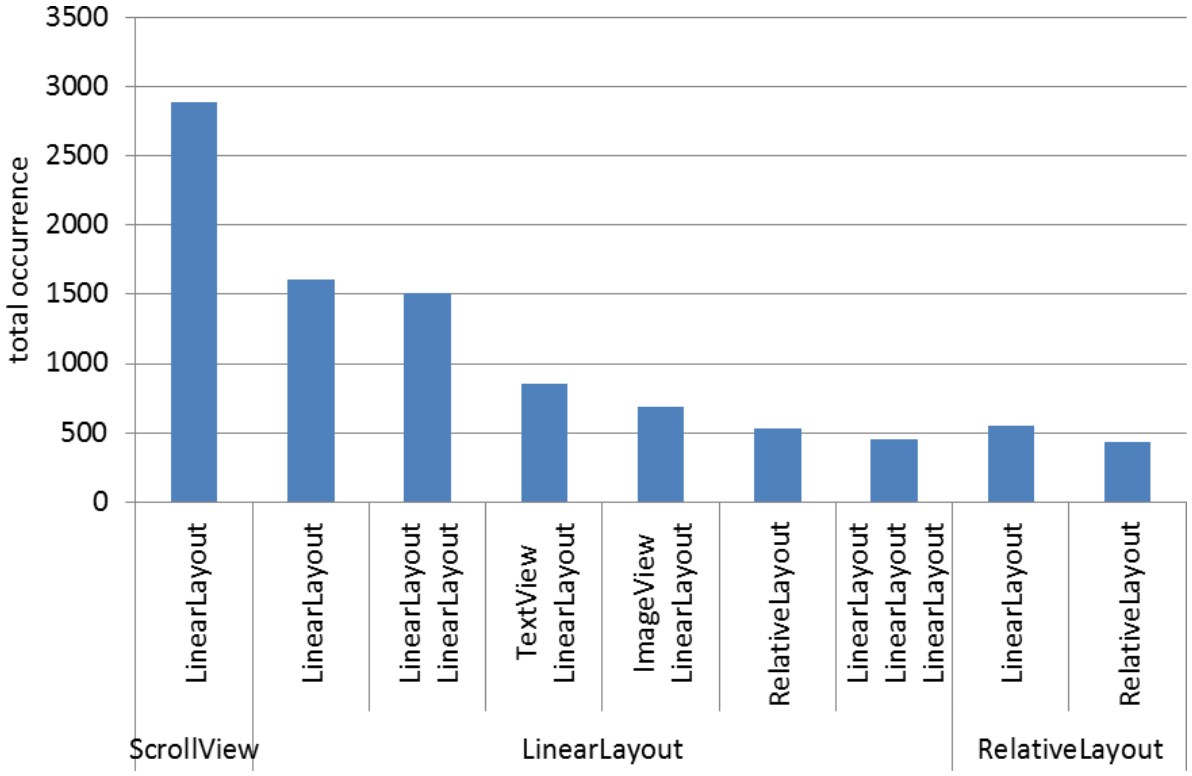

Abbildung 5.5: Die häufigsten Layout-Muster.

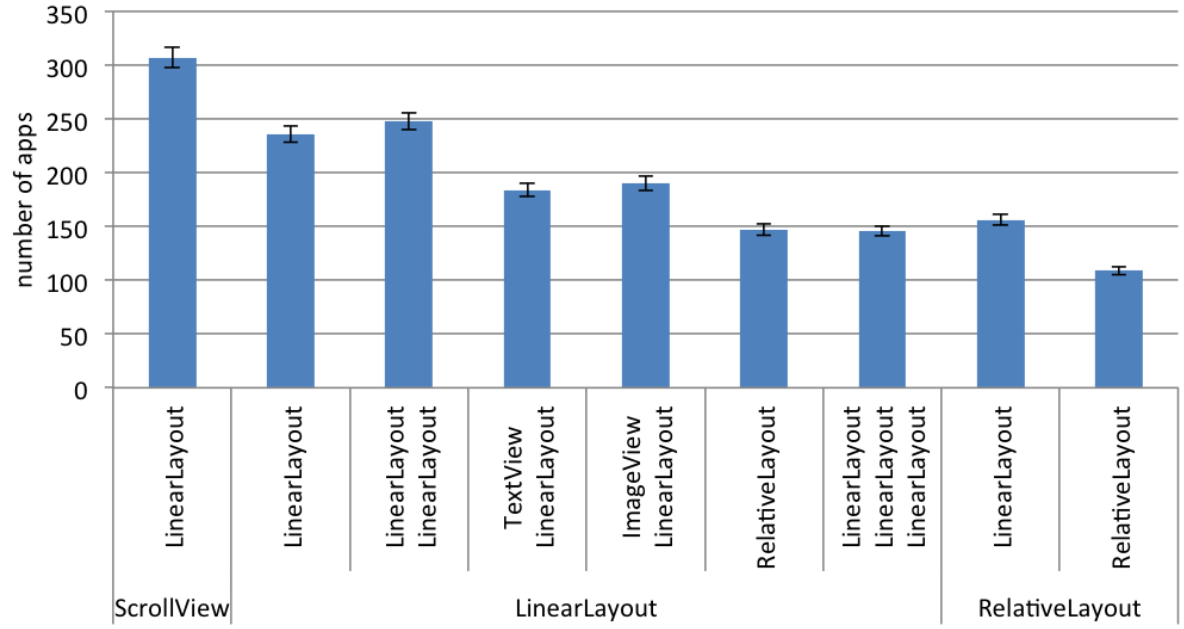

Abbildung 5.6: Die Verwendung häufiger Layout-Muster durch die Apps im Datensatz.

eingesetzten Layout-Container und Widgets herauszufinden. Insgesamt waren in in allen Layouts zusammen 77.343 Standard-Layout-Container aus der Android-API und 159.072 Widgets enthalten, was naiv im Schnitt zwei Widgets pro Layout-Container entspräche.

Abbildung 5.5 zeigt die Verwendung der Standard-Layout-Container in unserem Datensatz. Das LinearLayout hat einen Anteil von 66,95 % an der Gesamtzahl der Layout-Container und wird von 390 Apps verwendet. Der Anteil des RelativeLayouts (verwendet in 365 Apps) beträgt 24,20 %. Auch FrameLayout und ScrollView werden vom Großteil der Apps verwendet (in 307 bzw. 332 Apps), haben aber nur 7,82 % und 2,35 % Gesamtanteil. Mit jeweils unter einem Prozent an der Gesamtheit der Layouts auf den hinteren Plätzen befinden sich das TableLayout (167 Apps) und das AbsoluteLayout (35 Apps). Abbildung 5.6 zeigt die durchschnittliche Verwendung der Layouts in den Apps.

Eine ähnliche Betrachtung führten wir für die Standard-Widgets der Android-API durch. Das unter den 159.072 Elementen mit Abstand am häufigsten auftretende Element war der TextView (35,5 %), gefolgt von ImageView (15,6 %) und Button (9,4 %). Tabelle 5.1 zeigt die zehn meistverwendeten Widgets. Zusammen machen diese zehn Widget-Typen 74,87 % der Gesamtzahl der Widgets aus. Sie werden von über der Hälfte der 400 analysierten Apps verwendet. Abbildung 5.7 gibt einen weiteren Einblick in die durchschnittliche Verwendung der Widgets in den verschiedenen Apps.

Zusätzlich zu den Standard-Widgets aus der Android-API fanden wir 4.022 eigenentwickelte Layouts und Widgets in den Layout-Dateien. Dabei handelt es sich häufig um leichte Abwandlungen von Schaltflächen oder Layouts, manche Widgets stellten aber auch größere Komponenten wie Galerie-Ansichten dar oder ermöglichten die Auswahl eines Datums.

5.2.3 User-Interface-Muster

Nachdem wir die Widgets und Layout-Container aus den Layout-Dateien analysiert hatten, versuchten wir, potentielle Muster für den Entwurf einer Benutzungsschnittstelle zu finden. Hierfür betrachteten wir, wie Elemente kombiniert wurden, d.h. welche Elemente zusammen verwendet wurden und durch welchen Typ von Layout-Container sie üblicherweise gruppiert wurden.

Hierfür untersuchten wir die Layout-Container und die darin enthaltenen Elemente, die durch die Inklusionsbeziehung hierarchisch strukturiert sind. Daher bildeten wir aus den XML-Dateien die Eltern-Kind-Beziehung als Datenstruktur. Hierüber konnten wir für jedes Element herausfinden, in welchem Layout-Container es organisiert war und welche Geschwisterelemente es auf derselben Ebene besitzt. Dadurch fanden wir Strukturen und Kombinationen heraus, die wir anhand der Häufigkeit, mit der sie auftraten, zählten, um gebräuchliche Muster in der Hierarchie zu entdecken.

Insgesamt fanden wir 22.870 unterschiedliche Kombinationen von Elementen. Hiervon wurden jedoch 75,8 % lediglich einmal verwendet. Die Analyse erlaubte die Klassifizierung der Muster in zwei verschiedene Arten.

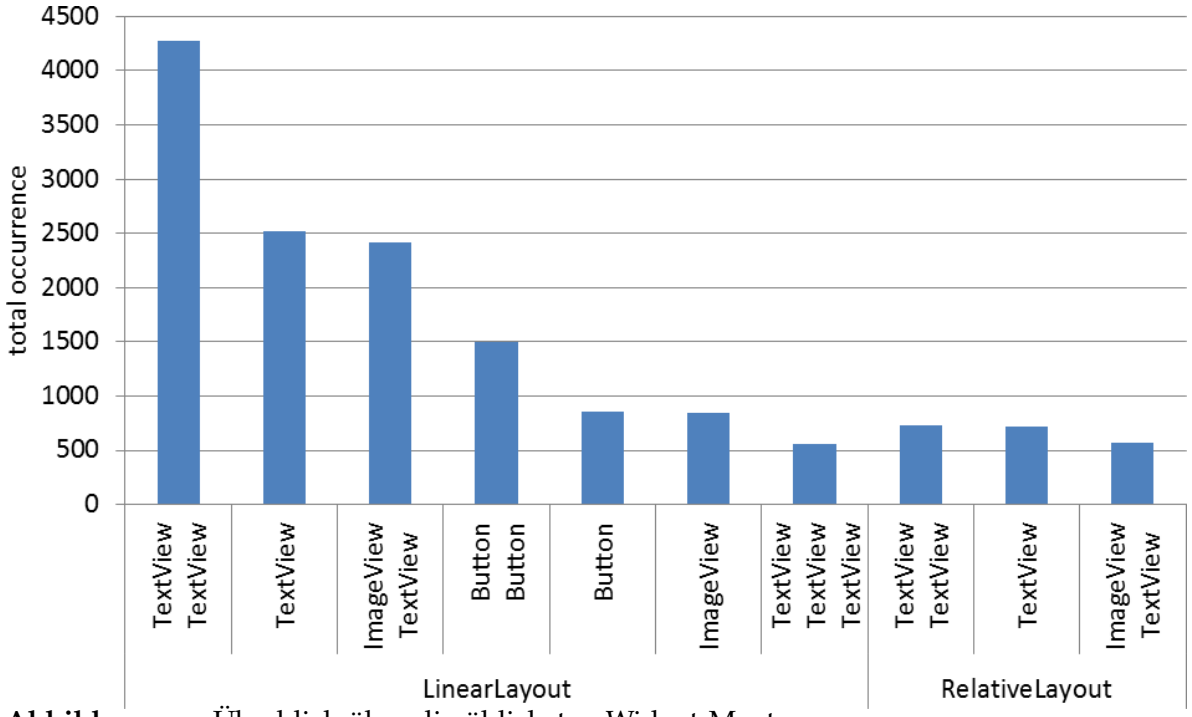

Abbildung 5.7: Überblick über die üblichsten Widget-Muster.

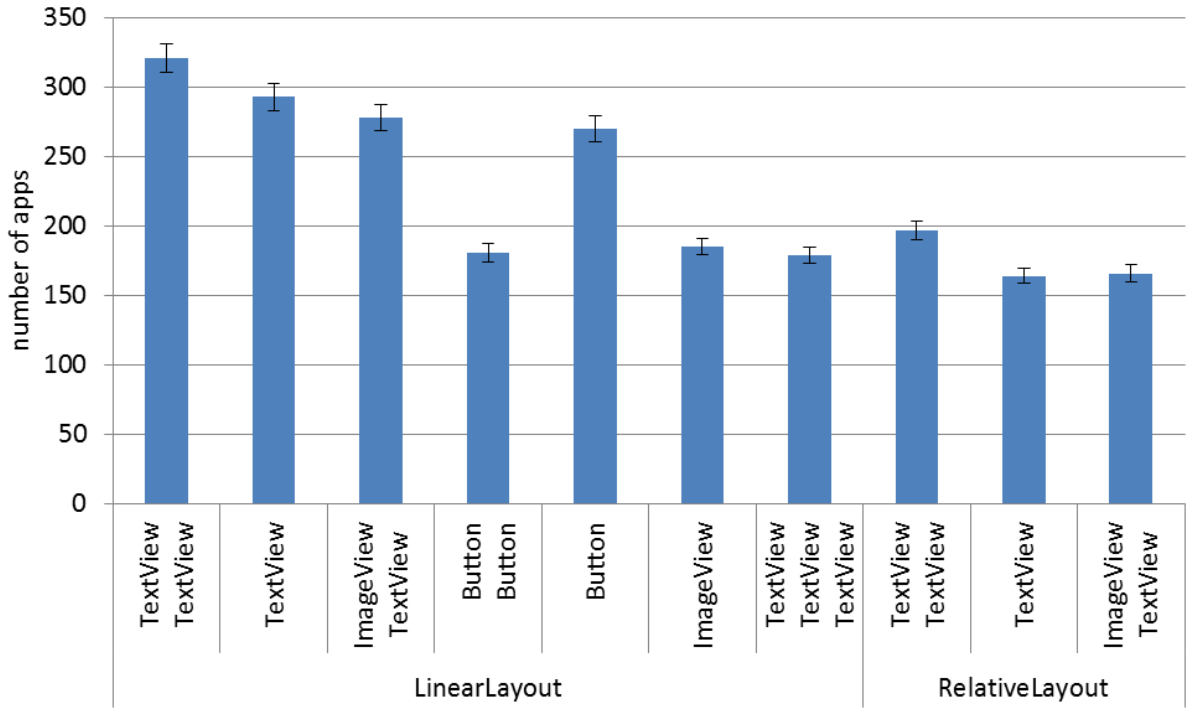

Abbildung 5.8: Verwendung der häufigen Widget-Muster in den Apps.

Der erste Typ besteht aus einem Layout-Container, der neben beliebig vielen Widgets noch andere Layout-Container enthält. Aus dieser Kategorie ist das am häufigsten anzutreffende Muster ein ScrollView als Layout-Container, der ein LinearLayout enthält. Dieses Muster ermöglicht die Verwendung von mehr Elementen, als der Bildschirm aufgrund seiner Auflösung gleichzeitig darstellen könnte. Abbildung 5.5 zeigt die neun am häufigsten gesehenen Muster dieser Kategorie. Wir betrachteten außerdem die Anzahl der Apps, in denen das jeweilige Mustern verwendet wurde; das eben beschriebene wurde beispielsweise in 307 Apps genutzt. Interessanterweise wird das zweithäufigste Muster in weniger Apps (insgesamt 236) verwendet als das dritthäufigste (248 Apps). Die Verwendung in den Apps aus dem Datensatz wird in Abbildung 5.6 veranschaulicht. Eine zusätzliche Betrachtung der Kategorien der Apps ergab, dass Apps aus der Kategorie „Lernen“ durchschnittlich häufiger auf solche Muster zurückgriffen als Apps anderer Kategorien.

Der zweite Typ von Mustern besteht aus einem Layout-Container, der ausschließlich Widgets als Kind-Elemente beinhaltet. Abbildung 5.7 demonstriert die zehn Kombinationen dieser Art, die am öftesten auftraten. Spitzenreiter dieser Art von Muster ist die Kombination zweier TextViews in einem LinearLayout, gefolgt von einem LinearLayout mit nur einem TextView, die Kombination aus ImageView und TextView in einem LinearLayout belegt den dritten Platz. Wir untersuchten auch die Verwendung von ButtonViews in verschiedenen Pattern. Die Verwendung der Muster durch die verschiedenen Apps zeigt unter diesen ersten drei Plätzen ähnliche Trends. Ähnlich dem obigen Phänomen aus der ersten Art von Muster wird die vierthäufigste Kombination (zwei Buttons in einem LinearLayout) in weniger Apps verwendet als die fünfthäufigste (lediglich ein Button in einem LinearLayout). Abbildung 5.8 zeigt die Verwendung der Muster in den Apps. Diese Art von Entwurfsmuster wurde durchschnittlich besonders häufig in Apps der Kategorie „Soziale Netzwerke“ genutzt.

5.2.4 Diskussion der Befunde

Wir haben die Benutzungsschnittstellen der 400 populärsten Apps vom Google Market analysiert und dabei bestimmt, welche Widgets und Layout-Container am häufigsten auftreten. Wir fanden dadurch heraus, dass über die Hälfte aller verwendeten Widgets die Elemente TextView und ImageView ausmachen, die nur dazu dienen, Text bzw. Bilder anzuzeigen. Obwohl wir damit rechneten, dass Elemente zur Anzeige von Informationen offenkundig einen größeren Anteil ausmachen müssen als interaktive Elemente, überrascht das Verhältnis. Beispielsweise fanden wir etwa 47-mal so viele TextViews als RadioButtons zur Auswahl einer aus mehreren Optionen. Einige Standard-Widgets der Android-API können beinahe als esoterisch betrachtet werden, beispielsweise die Elemente ToggleButton oder SeekBar, die nur jeweils 0,37 % bzw. 0,25 % der Gesamtzahl der Widgets ausmachen. Die häufigsten interaktiven Widgets (Button- und EditText-Elemente) legen die Vermutung nahe, dass die Layouts der untersuchten Apps hauptsächlich dazu dienen, Text einzugeben und virtuelle Schaltflächen zu drücken.

Außer den einzelnen Widgets identifizierten wir Muster von Elementkombinationen. Die Resultate zeigen, dass die Muster allgemein häufig verwendet werden, also nicht nur in speziellen Apps zum Einsatz kommen. Einige dieser Kombinationen werden deutlich

häufiger verwendet als manche Standard-Widgets. Zusammengerechnet sind 21,13 % aller Widgets und Layout-Container unseres Datensatzes Teil mindestens eines der zehn häufigsten Muster von Elementen. 77,28 % aller ScrollViews enthalten ein LinearLayout. Gäbe es diese Kombination als eigenes Layout-Container-Element, wäre es der fünfthäufigste Layout-Container. Interessanterweise ist das zweithäufigste Muster ein LinearLayout, das in einem LinearLayout enthalten ist – eine Kombination, die als „nutzlos“ erachtet wird[1].

Das häufigste Widget-Muster in unserem Datensatz besteht aus zwei TextViews in einem LinearLayout. Diese Kombination stellt 5,43 % aller User-Interface-Elemente dar und kommt damit alleine etwa so häufig vor wie alle CheckBox-, RadioButton-, ToggleButton- und SeekBar-Elemente zusammen. Die Entwicklung dieser Kombination als neues Widget würde sich an sechster Stelle in die Rangfolge der am häufigsten genutzten Widgets einordnen. Die Entdeckung solcher empirischen Entwurfsmuster könnte also zu optimierten Widgets führen.

5.3 Einschränkungen

Die 400 in dieser Arbeit untersuchten Anwendungen wurden mittels einer nicht repräsentativen Stichprobenauswahl gezogen. Dazu kam der Ranking-Algorithmus des Google-Play-Markets zum Einsatz, indem die 400 populärsten Anwendungen heruntergeladen wurden. Auf welcher Basis Google dieses Ranking erstellt, ist nicht bekannt. Es ist anzunehmen, dass diese 400 Anwendungen auch tatsächlich weit verbreitet sind, da sie den Nutzern des Google-Play-Markets ebenfalls als erstes in den Listen angezeigt werden, aber der Einfluss anderer Faktoren wie das Rating der Nutzer kann nicht ausgeschlossen werden.

Für den Download von Anwendungen muss die Geräte-ID eines Android-Gerätes angegeben werden. Die Einstellungen und Eigenschaften dieses Gerätes beeinflussen ebenfalls die angezeigten Anwendungen, da Entwickler die Möglichkeit, haben die Nutzung ihrer Anwendungen auf bestimmte Gerätearten, Sprachen oder Konfigurationen zu beschränken; diese Einstellungen werden von Google genutzt, um nur passende Anwendungen anzubieten. Alle heruntergeladenen Anwendungen waren kostenlos, wir gehen jedoch davon aus, dass die populärsten Anwendungen für viele Sprachen und Plattformen verfügbar sind und wir damit viel genutzte Anwendungen untersucht haben.

Eine weitere Einschränkung bildet die Art, mit der Layout-Informationen in Android-Anwendungen angegeben werden können. Obwohl Google empfiehlt, alle Layout-Informationen in XML-Dateien abzulegen, können sie auch direkt im Code der Anwendung Elemente für die Benutzeroberfläche definiert werden. Diese werden bei unserer Analyse nicht abgedeckt, allerdings glauben wir, dass die grundsätzlichen Layouts in XML-Dateien

[1] Die Vorgehensweise vieler Entwickler, ein LinearLayout in einem LinearLayout zu platzieren, kommentiert das Android SDK: „This LinearLayout layout or its LinearLayout parent is useless“.

definiert werden und direkt im Code eher interaktive Elemente nach Bedarf erzeugt werden. Da unser Interesse häufig wiederkehrenden Mustern in den Layouts galt, scheint dies akzeptabel.

6 Folgerung und Ausblick

6.1 Zusammenfassung und zukünftige Arbeiten

Ziel dieser Arbeit war es, die Struktur der Benutzungsoberfläche einer großen Anzahl von Android-Anwendungen zu untersuchen, um sich wiederholende Muster und auffällige Kombinationen und Eigenschaften zu finden. Zu diesem Zwecke haben wir zunächst die 400 populärsten Anwendungen aus dem Google Play Market herunter geladen. Diese wurden anschließend entpackt und dekompiliert, um an Informationen über die Benutzungsoberfläche zu gelangen. Mit den eingesetzten Reverse-Engineering-Techniken konnten Teile des ursprünglichen Quellcodes und alle verwendeten Ressourcen zugänglich gemacht werden. Mittels Analysen der XML-Layout-Dateien, der verwendeten Ressourcen und des Quellcodes konnten wir eine Reihe von Kennzahlen und Zusammenhängen extrahieren und in einer Datenbank speichern. Diese reichen von den verwendeten Sprachen über die von einer Anwendung benötigten Rechte bis hin zu kompletten Layout-Informationen mit Hierarchien und Mustern der einzelnen Aktivitäten.

Bei Betrachtung der Sprachen fiel eine breite Unterstützung für mehrere Sprachen auf, 88,25 % der Anwendungen unterstützen mehr als eine Sprache. Bei mehr als der Hälfte der Anwendungen waren es noch 5 verschiedene Sprachen, wobei Chinesisch, Spanisch und Französisch am häufigsten auftraten. Die Möglichkeit, Ressourcen für verschiedene Bildschirmauflösungen und Größen zu hinterlegen, wurde insgesamt nur wenig genutzt. Das Android-System ist in der Lage, die Ressourcen selbstständig auszuwählen und anzupassen. Dazu passt die Beobachtung, dass eher Resourcen für hohe Auflösungen und größere Bildschirme hinterlegt werden und die Umwandlung für einfachere bzw. geringer auflösende Formate dem Android-System überlassen wird. Weniger als die Hälfte der untersuchten Anwendungen unterstützen alle vier möglichen Auflösungen und hatten Definitionen für verschiedene Bildschirmgrößen.

Während mit 96,25 % annähernd alle Anwendungen die Berechtigung für den Zugriff auf das Internet verlangen, sind nur 47,25 % in der Lage, taktiles Feedback über die Vibrationsmotoren des Gerätes zu geben. Auch die Abfrage des aktuellen Standorts wird nur in etwa der Hälfte der untersuchten Anwendungen verlangt. Weitere häufig vorhandene Berechtigungen sind die Prüfung auf eine bestehende Netzwerkverbindung und das Schreiben auf den Datenträger des Android-Gerätes.

Bei der Betrachtung der Aktivitäten einer Anwendung fällt auf, dass lediglich 20 % mehr als eine Hauptaktivität und damit mehr als einen Einstiegspunkt anbieten. Insgesamt zeigen sich signifikanten Unterschiede in der Anzahl der Aktivitäten bei Anwendungen unterschiedlicher Kategorien. Während Anwendungen aus der Kategorie „Soziale Netzwerke“

über sehr viele Aktivitäten verfügen, ist die Anzahl bei Anwendungen aus den Kategorien „Werkzeuge“ und „Unterhaltung“ wesentlich geringer.

Im Hinblick darauf, ob die Anzahl der Layout-Dateien stark variieren, ließen sich bei den Anwendungen – gruppiert nach den am häufigsten auftretenden Kategorien – zwei Gruppen ausmachen: Sehr umfangreich sind Anwendungen der Kategorien „Kommunikation“, „Soziale Netzwerke“, „Shopping“ und „Reisen“ gestaltet. Im Kontrast dazu stehen Anwendungen aus den Kategorien „Unterhaltung“, „Tools“, „Musik & Audio“ und „Effizienz“, die entweder nur einen kleinen, speziellen Aufgabenbereich oder sehr dynamische Benutzeroberflächen besitzen, die mittels der statischen Analyse in dieser Arbeit nur unzureichend abgebildet werden können. Hier findet sich auch ein erster Ansatz zu möglichen weiteren Arbeiten. Emulatoren bieten die Möglichkeit, Anwendungen während der Nutzung zu beobachten und Nutzereingaben zu simulieren. Damit können bessere Erkentnisse über die Beziehungen der einzelnen Aktivitäten innerhalb der Anwendung und auch der Interaktion mit anderen Anwendungen gewonnen werden. Zudem könnten so auch Benutzungsoberflächen erfasst werden, die sich dynamisch ändern und damit neue Einsichten über Interaktionskonzepte gefunden werden.

Bei der Analyse der Layout-Dateien konnten wir häufig genutzte Android-Widgets identifizieren, die Beliebtheit der unterschiedlichen Layout-Möglichkeiten feststellen und häufig auftretende Muster aus mehreren Widgets und Layouts finden. Auffällig war hier die große Anzahl einfacher Text-, Bild- und Button-Widgets, komplexere, interaktive Widgets wurden nur selten verwendet. Einige der in vielen Anwendungen auftreten Muster aus einfachen Widgets kommen deutlich häufiger vor als komplexere Standard-Android-Widgets, so beispielweise die Kombination von mehreren Textelementen oder Text- und Bildelementen.

Bei den Layout-Containern wird hauptsächlich das LinearLayout innerhalb eines ScrollViews verwendet, um einfach Inhalte untereinander auf dem Bildschirm anzubieten. Überraschenderweise findet sich auch die Kombination aus ineinander verschachtelten LinearLayouts sehr häufig, obwohl diese Art der Strukturierung auf die Gestaltung der Aktivitäten keinen Einfluss hat. Eine nähere Untersuchung dieses Gebietes könnte nach komplexeren Mustern suchen, in dieser Arbeit wurden nur Muster betrachtet, die eine Tiefe von maximal zwei Ebenen haben. Eine genauere Analyse der Struktur der Benutzeroberflächen könnte Einsichten über Gestaltungregeln für erfolgreiche Anwendungen liefern. Hier sind auch praktische Anwendungen der gewonnenen Erkenntnisse denkbar. Vor allem in Entwicklungsumgebungen und Gestaltungswerkzeugen könnten häufig vorkommende Muster genutzt werden, um Layouts schneller und einfacher zu gestalten und dem Nutzer dienliche Hinweise zur Strukturierung der Benutzeroberfläche seiner Anwendung zu geben.

Literaturverzeichnis

[BBK10] M. Böhmer, G. Bauer, A. Krüger. Exploring the design space of context-aware recommender systems that suggest mobile applications. In *Proceedings of CARS*. 2010. (Zitiert auf Seite 11)

[BBS+10] T. Blaesing, L. Batyuk, A. Schmidt, S. A. Camtepe, S. Albayrak. An Android Application Sandbox System for Suspicious Software Detection. In *Proceedings of 5th International Conference on Malicious and Unwanted Software (MALWARE)*, S. 55–62. 2010. (Zitiert auf Seite 10)

[BFFH09] F. Balagtas-Fernandez, J. Forrai, H. Hussmann. Evaluation of user interface design and input methods for applications on mobile touch screen devices. In *Proc. of Interact*, S. 243–246. Springer, 2009. (Zitiert auf Seite 11)

[BHS+11] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, G. Bauer. Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage. In *Proceedings of MobileHCI*, S. 47–56. 2011. (Zitiert auf Seite 11)

[BKOS10] D. Barrera, H. G. Kayacik, P. C. van Oorschot, A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, S. 73–84. ACM, New York, NY, USA, 2010. doi:10.1145/1866307.1866317. URL http://doi.acm.org/10.1145/1866307.1866317. (Zitiert auf Seite 9)

[CFGW11] E. Chin, A. P. Felt, K. Greenwood, D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, S. 239–252. ACM, New York, NY, USA, 2011. doi:10.1145/1999995.2000018. URL http://doi.acm.org/10.1145/1999995.2000018. (Zitiert auf Seite 10)

[CR08] Y. Cui, V. Roto. How people use the web on mobile devices. In *Proceedings of WWW*, S. 905–914. 2008. (Zitiert auf Seite 11)

[FCH+11] A. P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, S. 627–638. ACM, New York, NY, USA, 2011. doi:10.1145/2046707.2046779. URL http://doi.acm.org/10.1145/2046707.2046779. (Zitiert auf Seite 9)

[HN11] C. Hu, I. Neamtiu. A GUI bug finding framework for Android applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, S. 1490–1491. ACM, New York, NY, USA, 2011. doi:10.1145/1982185.1982504. URL http://doi.acm.org/10.1145/1982185.1982504. (Zitiert auf Seite 10)

[HRB11] N. Henze, E. Rukzio, S. Boll. 100,000,000 taps: analysis and improvement of touch performance in the large. In *Proc. of MobileHCI*, S. 133–142. 2011. (Zitiert auf Seite 12)

[HRB12] N. Henze, E. Rukzio, S. Boll. Observational and Experimental Investigation of Typing Behaviour using Virtual Keyboards on Mobile Devices. In *Proc. of CHI*. 2012. (Zitiert auf Seite 12)

[LBG+12] L. A. Leiva, M. Böhmer, S. Gehring, et al. Back to the App: The Costs of Mobile Application Interruptions. In *Proc. of MobileHCI*. 2012. (Zitiert auf Seite 12)

[MDR+12] A. Möller, S. Diewald, L. Roalter, F. Michahelles, M. Kranz. Update behavior in app markets and security implications: A case study in Google Play. In *Proceedings of MobileHCI*. 2012. (Zitiert auf Seite 11)

[RTS+12] A. Rahmati, C. Tossell, C. Shepard, P. Kortum, L. Zhong. Exploring iPhone usage: The influence of socioeconomic differences on smartphone adoption, usage and usability. In *Proceedings of MobileHCI*. 2012. (Zitiert auf Seite 11)

[RZ12] A. Rahmati, L. Zhong. Studying smartphone usage: Lessons from a four-month field study. In *IEEE Transactions on Mobile Computing*. 2012. (Zitiert auf Seite 11)

[SEKV12] M. Szydlowski, M. Egele, C. Kruegel, G. Vigna. Challenges for Dynamic Analysis of iOS Applications. In J. Camenisch, D. Kesdogan, Herausgeber, *Open Problems in Network Security*, Band 7039 von *Lecture Notes in Computer Science*, S. 65–77. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-27585-2_6. URL http://dx.doi.org/10.1007/978-3-642-27585-2_6. (Zitiert auf Seite 11)

[Ver09] H. Verkasalo. Contextual patterns in mobile service usage. *Personal and Ubiquitous Computing*, 13(5):331–342, 2009. (Zitiert auf Seite 11)

Alle URLs wurden zuletzt am 11. 02. 2013 geprüft.