Bachelor-Thesis

Fachhochschule Düsseldorf

vorgelegt am: 12. April 2012

Konzeption und Implementierung eines Android-Services für das OSGi-Framework

Design and implementation of an Android service for the OSGi Framework

Name:	Niels Wientzek
Matrikelnummer:	468045
1. Prüfer:	Prof. Dr. Lux
2. Prüfer:	Prof. Dr. Schaarschmidt

ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorgelegte Bachelor-Arbeit selbständig angefertigt und keine andere als im Schrifttumsverzeichnis angegebene Literatur benutzt habe.

Ort, Datum: .....

Unterschrift: .....

Zusammenfassung

In diesem Projekt wird ein Softwaresystem entwickelt, mit dem es möglich ist ein OSGi Framework durchgängig im Hintergrund von Android zu betreiben. Für die Demonstration der Kommunikation zwischen Framework und weiteren Android Anwendungen wird eine Beispielanwendung entworfen. Im ersten Teil dieser Arbeit wird auf die technischen Grundlagen von Android sowie auf die OSGi Service Platform eingegangen. Anschließend folgt die Anforderungsanalyse, sowie die State-of-the-art-Analyse. Das Konzept wird erstellt und anschließend die Implementierung erläutert. Zum Schluss wird das Softwaresystem evaluiert und eine Zusammenfassung aller Kapitel beendet die Arbeit.

Abstract

In this project, a software system will be developed, which make it possible to operate an OSGi framework consistently in the background of Android. For communication between the Framework and other Android applications, a sample application is designed to demonstrate this. The first part of this thesis discusses the technology behind Android and the OSGi Service Platform. After that the requirements analysis and state-of-the-art analysis follows. The concept is created and the implementation is explained. Finally, the software system is evaluated and a summary of each chapter is at the end of this thesis.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Diese Bachelorarbeit entstand in Anlehnung an die von Thomas Schmitz durchgeführte Bachelorarbeit „Entwicklung einer OSGi-Service-Komponente zum dynamischen Laden von Benutzeroberflächen für Android im Umfeld von Ambient Assisted Living“ (1). Die dort entwickelte Software sollte es ermöglichen, verschiedene Geräte dynamisch auf einem Android Smartphone zu verwalten. Dazu zählte auch das Reagieren auf Alarmmeldungen verschiedener Sensoren. Die entwickelte Android Anwendung ist dabei nur in der Lage, die Geräte zu verwalten bzw. auf Alarmmeldungen zu reagieren, wenn Sie auf dem Smartphone aktiviert ist. Befindet sich eine andere Anwendung im Vordergrund, oder ist das Smartphone inaktiv, kann nicht auf die Alarmmeldungen reagiert werden. Da die Benachrichtigung des Benutzers im Alarmfall eine tragende Rolle im Konzept der Software spielt, ist es notwendig diese Funktionalität herzustellen. Der Schwerpunkt dieser Arbeit liegt dabei darin, das OSGi Framework stabil und durchgängig auf einem Android Smartphone zu betreiben, um ständig auf Meldungen verschiedenster Geräte eingehen zu können.

1.2 Überblick über die Arbeit

In dieser Arbeit werden zunächst die zwei Schlüsseltechnologien, Android als Basissystem sowie OSGi als Modulsystem im Allgemeinen beschrieben. In Unterpunkt 2.3 wird darauf, eingegangen, welche Vorteile die Verwendung von OSGi unter Android mit sich bringt. Dazu zählt vor allem die Verwendbarkeit von bestehendem OSGi Programmen auf Android und das flexiblere Installieren, Laden und Updaten von Anwendungen, die als OSGi Bundles auf einem Smartphone laufen, im Vergleich zu nativen Android Anwendungen.

Kapitel 3 beschäftigt sich mit der Anforderungsanalyse des Projektes, indem die Anforderungen an die zu entwickelnde Software aufgeführt werden. Darunter fällt vor allem die durchgängige Verfügbarkeit des OSGi Framewoks, sowie die Kommunikation zwischen OSGi Kontext und Android Anwendungen.

Im Abschnitt State-of-the-art-Analyse wird auf die aktuellen Implementierungen der OSGi Plattform eingegangen. Dabei werden die OSGi Frameworks Equinox, das Prosyst OSGi Framework sowie, Apache Felix vorgestellt. Dabei wird deutlich, dass nur das von Prosyst entwickelte „mBS Mobile OSGi for Android“ Framework, sowie Apache Felix unter Android lauffähig sind. Die freie Verfügbarkeit und die aktive Entwicklergemeinde von Apache Felix

führte dazu, dass es bereits Projekte gibt, die Apache Felix als OSGi Framework unter Android einsetzen. Einige dieser Projekte werden am Ende des Kapitels beschrieben.

Den Hauptteil der Arbeit bilden die Kapitel 5 Konzeption und 6 Implementierung. Es werden verschiedene Konzepte für die Bereiche Bereitstellung eines OSGi Frameworks unter Android, Implementierungskonzepte für die GUI der Geräte, sowie der Kommunikationsmöglichkeiten zwischen OSGi und Android beschrieben. Besonders die verschiedenen Kommunikationsmöglichkeiten die Android für die Interaktion zwischen Client und Service anbietet, werden hier beschrieben. Neben den in diesem Zusammenhang oft beschriebenen Möglichkeiten werden auch Android Intents vorgestellt, die sich wegen Ihrer losen Bindung für eine einfache Nachrichtenübertragung, auch mit Nutzdaten, sehr gut für die Kommunikation eignen. Abschließend wird ein detailliertes Gesamtkonzept beschrieben.

Das Kapitel der Implementierung beschäftigt sich mit der, vom Konzept ausgehenden, prototypischen Implementierung. Es wird eine Android Anwendung entwickelt, die das Apache Felix OSGi Framework als Android Service startet. Innerhalb des Frameworks laufen Bundles, welche die Kommunikation mit den Geräten übernehmen. Außerdem wurde eine weitere Android Anwendung entwickelt, welche die GUI und die Auswertung der Daten eines Wassermelders übernimmt. Im Alarmfall ist das OSGi Wassermelder Bundle in der Lage die Wassermelder Android Anwendung zu starten und eine Alarmierung in Form von audio- , visueller- und haptischer Rückmeldung auszugeben.

Die Evaluation erfolgt im entsprechenden Kapitel. Es werden die verschiedenen Szenarien durchgespielt und geprüft, ob sich die Software wie gewünscht verhält. Außerdem wird ein Benchmark durchgeführt, welcher zeigen soll, das der Hintergrundprozess des Frameworks nicht merkbar die Leistung des Smartphones beeinträchtigt.

Im letzten Kapitel dieser Arbeit werden alle Ergebnisse zusammengefasst und in einem Fazit aufgeführt. Dabei wird auch auf die möglichen Weiterentwicklungen der Software eingegangen.

2 Grundlagen

2.1 Android

Android ist ein Betriebssystem für mobile Geräte wie Smartphones und Tablet Computer. Ursprünglich stammt das System von der gleichnamigen Firma, die Mitte 2005 von Google aufgekauft wurde. Android wird von der Open Handset Alliance (2) als freie und quelloffene Software weiterentwickelt. Die Open Handset Alliance ist ein Zusammenschluss aus Netzbetreibern, Software-Firmen, Marketing Unternehmen, Halbleiterindustrie sowie Geräteherstellern der von Google am 5. November 2007 initiiert wurde. Durch die Möglichkeit für Gerätehersteller Android kostenlos zu nutzen, wird es gerne eingesetzt. Auf dem Mobile World Congress, der im Februar 2012 stattfand, teilte Google mit, dass jeden Tag 850.000 Android Geräte aktiviert werden. (3)

2.1.1 Systemaufbau

Im folgenden Kapitel soll auf die Android Architektur eingegangen werden, auf die jede Version aufbaut. Abbildung 1 gibt eine Übersicht über die 5 Hauptteile der Architektur.

- Anwendungsschicht
- Anwendungsrahmen
- Bibliotheken
- Android-Laufzeitumgebung
- Linux-Kernel

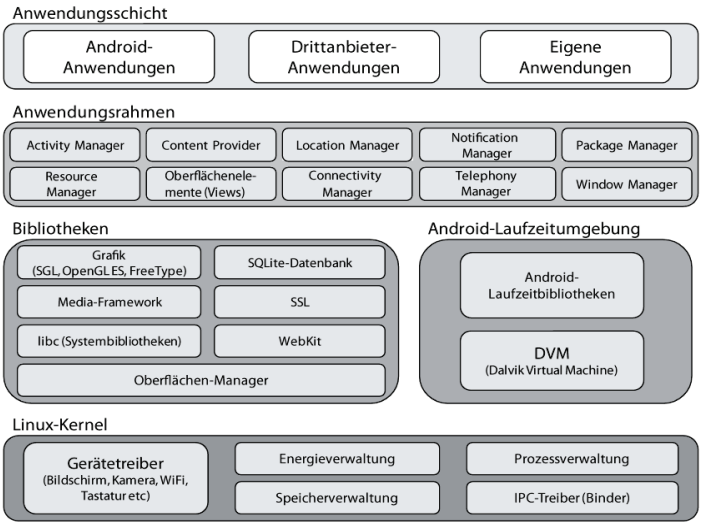


Abbildung 1: Android Systemaufbau (4)

Android basiert zwar auf einem Linux Kernel, es unterscheidet sich aber in vielen Einzelheiten stark von anderen Linux-Versionen. Auf die einzelnen Teile der Architektur wird im Folgenden genauer eingegangen.

2.1.1.1 Linux Kernel

Kern des Betriebssystem ist ein angepasster Linux Kernel in der Version 2.6.x (Ab Android 4.0 Kernel 3.0.x). Ein Standard Kernel würde die verhältnismäßigen schwachen CPUs, im Vergleich zu Desktop PCs, zu stark belasten und die Akkulaufzeit deutlich verringern. Deshalb wurden zahlreiche Treiber und Bibliotheken auf die Bedürfnisse mobiler Geräte angepasst oder vollständig ersetzt. Eine der Besonderheiten des Kernels besteht in der dort implementierten Methode der Interprozesskommunikation (IPC). Diese Art der Kommunikation ist in der Regel sehr Ressourcen aufwändig. Die auszutauschenden Daten müssen von ihrer internen Form in ein für die Übertragung geeignetes Format und wieder zurück gewandelt werden. Dieser Prozess des Marshalling und Unmarshaling ist speicher- und rechenintensiv. Um dies zu vermeiden, implementiert Android einen speziellen Treiber namens Binder. Dieser realisiert die IPC durch Shared Memory Zugriffe. Programme tauschen per IPC lediglich Referenzen auf Objekte aus, die im Shared Memory abgelegt sind. Das Marshalling und Unmarshaling entfällt hier komplett. Die Kernelmodule Kernel-Debugger und Logger ermöglichen zudem ein komfortables Debuggen von Android Anwendungen. Per logcat lassen sich die Ausgaben in Echtzeit verfolgen.

2.1.1.2 Bibliotheken

Unter den Android Bibliotheken finden sich viele bekannte Standard C/C++ Bibliotheken wie sie auch bei anderen Linux Distributionen vorkommen. Zu diesen zählen unter anderem das WebKit als Rendering-Engine für den Browser, die für eingebettete Systeme optimierte 3D-Grafikbibliothek OpenGL ES und SQLite für Datenbankfunktionen. Als libc Bibliothek verwendet Android nicht die weit verbreitete Glibc Implementierung, sondern setzt hier auf die C-Bibliothek Bionic. Diese ist Glibc kompatibel, ist aber mit rund 200 KByte deutlich kleiner und auf Geschwindigkeit optimiert. Des Weiteren ist sie um einige Android spezifische Funktionen erweitert. Binoic enthält beispielsweise eine besondere Pthread-Implementierung. Diese berücksichtigt, dass auf einem mobilen Gerät nicht hunderte Threads parallel laufen müssen, sondern vor allem jeder Thread schnell gestartet werden muss und wenig Speicher verbraucht.

Des Weiteren gibt es auch speziell für Android entwickelte System Bibliotheken, zu diesen zählen Beispielsweise:

- Surface Manager - Zugriffssteuerung des Anzeigesystem
- SGL - Rendering Bibliotheken für Vektorgrafiken
- Media Framework - Codecs für Multimedia Inhalte

2.1.1.3 Android Laufzeitumgebung

Die Android Laufzeitumgebung besteht aus zwei Komponenten. Die Core Bibliotheken, die aus der Java Standard Edition stammen und für Android portiert wurden und der Dalvik Virtual Machine, die die Ausführung der einzelnen Anwendungen übernimmt.

Zu den Core Bibliotheken, die für Android portiert wurden, zählen:

• Sprach und Unterstützungsbibliotheken
  - java.lang - stellt Elementare Grundtypen von Java bereit
  - java.util - Collections, Event-Handling, Datum
• Basisbibliotheken
  - java.io - Eingabe und Ausgabe von Dateien/Streams
  - java.math - mathematische Funktionen und Konstanten
  - java.net - Basis Netzwerkfunktionen
  - java.nio - Erweiterung des I/O Paket
• Integrationsbibliotheken
  - java.sql - Datenbank Schnittstelle
  - javax.sql - Erweiterungen des SQL Paket
• Sicherheitsbibliotheken
  - javax.crypto - stellt Verschlüsselungstechnologien bereit
  - java.security - (veraltet) Java Sicherheits Framework
  - javax.security - (aktuelle) Java Sicherheits Framework
• weitere Bibliotheken
  - javax.net - erweiterte Netzwerkfunktionalität
  - javax.sound - Paket für die Audioverarbeitung
  - javax.xml - XML Verarbeitung
  - org.w3c.dom - Schnittstelle für Document Object Model
  - org.xml.sax - SAX-Parser

(1)

Die DalvikVM ähnelt einer Java Virtual Machine, allerdings gibt es auch hier spezifische Android Anpassungen. Dalvik basiert auf der quelloffenen Java-VM Apache Harmony (5), arbeitet aber mit einem eigenen Bytecode. Android Programme werden in Java geschrieben und anschließend mit dem Java-SDK Compiler zu normalem Java-Bytecode kompiliert. In dieser Form ist das Programm aber nicht auf Android Geräten ausführbar. Der Java-Bytecode muss zunächst mit dem Android Developer Tool namens dx in Dalvik-Executable Bytecode, kurz dex-Bytecode, umgewandelt werden. Im Laufe des

Buildprozesses werden einige Optimierungen bei der Zusammenstellung des fertigen Android Programmes durchgeführt. Die nötigen Schritte vom Java Quellcode bis hin zur Signierten Android Anwendung sind in Abbildung 2 aufgeführt. In der Regel sind diese Schritte für den Entwickler mithilfe einer geeigneten IDE und den integrierten Android Developer Tools mit wenigen Klicks durchgeführt.

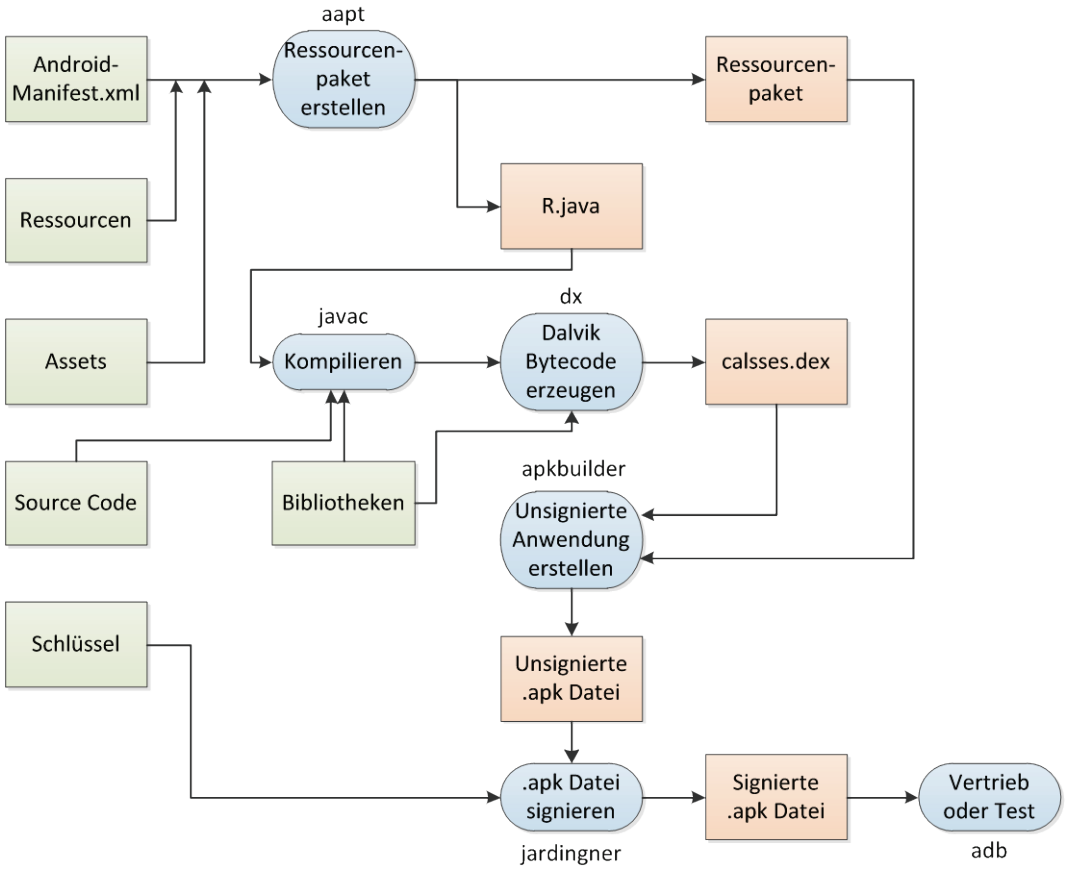


Abbildung 2: Buildprozess einer Android Anwendung

Die grün gekennzeichneten Felder, der Abbildung 2, stellen die vom Entwickler bereitgestellten Ressourcen dar. Die Manifest Datei enthält die Beschreibung der Anwendung. In den Verzeichnissen Ressourcen und Assets liegen die Anwendungs-ressourcen wie Bilder und Musikdateien. Dabei unterscheiden sich beide Ordner kaum, nur der Zugriff auf die im Ressourcen Ordner liegenden Dateien gestaltet sich einfacher. Aus diesen drei Quellen wird, mithilfe der Android Developer Tools, die R.java und das Ressourcen Paket erstellt. Die R.java verwaltet den Zugriff auf die Ressourcen, während im Ressourcen Paket die eigentlichen Dateien liegen. Bibliotheken, der Sourcecode der Anwendung, sowie die R.java werden kompiliert. Anschließend wird der entstandene Java-Bytecode in dex-Bytecode umgewandelt. Die entstandene classes.dex Datei wird zusammen mit den Ressourcen zu einer unsignierten Android Anwendung zusammengefasst. Die entstandene .apk Datei ist in dieser Form bereits unter Android installierbar. Möchte der Entwickler die Anwendung offiziell im Google Play Store vertreiben, muss sie mit dem Schlüssel des Entwicklers noch signiert werden.

Jede Android Anwendung läuft in einem eigenen Prozess und in einer eigenen DalvikVM. Jede Anwendung erhält außerdem ihren eigenen Ordner im Dateisystem. Die Zugriffsrechte der Ordner liegen jeweils bei dem, für jede Anwendung einzeln generierten, Linux Benutzer. Zusammen hat dies den Vorteil, dass einerseits eine Anwendung abstürzen kann, ohne eine andere zu beeinflussen. Andererseits wird die Sicherheit deutlich erhöht, da eine Anwendung nicht ohne weiteres auf eine andere zugreifen kann.

2.1.1.4 Anwendungsrahmen

Der Anwendungsrahmen enthält die Android-spezifischen Klassen und abstrahiert die zugrunde liegende Hardware. In ihm befindet sich beispielsweise der Activity Manager, der den Lebenszyklus einer Anwendung verwaltet, damit bildet er die Grundlage für die darüber liegende Anwendungsschicht. Zu den weiteren Diensten gehören beispielsweise:

- Package Manager: Verwaltet die heruntergeladenen und installierten Anwendungen
- Ressource Manager: Verwaltet die Ressourcen, wie Grafiken, Layouts und Strings der Anwendungen
- Location Manager: Stellt die Daten des GPS Empfängers zu Verfügung
- Notification Manager: Ermöglicht Anwendungen das Anzeigen von Nachrichten in der Statuszeile

2.1.1.5 Anwendungsschicht

Den Abschluss bildet die Anwendungsschicht, hier befinden sich die eigentlichen Anwendungen, auch Apps genannt. Je nach Android Version bzw. durch Anpassung der Geräte Hersteller und Vertreiber sind hier verschiedene Anwendungen vorinstalliert. Auch nachinstallierte und selbst programmierte Anwendungen befinden sich in dieser Schicht. Zu den üblichen vorinstallierten Anwendungen zählen beispielsweise:

- Google Maps – Online und Offline Karten Anzeige mit Positionsbestimmung und Navigation
- Google Play Store (früher Android Market) – Der Google Anwendungs Markt, hier können neue Anwendungen bezogen werden.
- Browser - Internet Browser
- Kontakte – Einfache Kontaktverwaltung
- Kalender – Einfache Kalenderanwendung

Die Interaktion zwischen Android Anwendungen durch die Freigabe von Komponenten oder Daten erhöht die Wiederverwendbarkeit. Zu den Anwendungskomponenten zählt Android:

- Activitys
- Content Provider
- Broadcast Receivers
- Services

Ein Beispiel für die Wiederverwendung stellt die Anwendung Kontakte dar. Sie gehört zu den meist referenzierten Programmen. Die Telefon Anwendung, Voip Anwendungen sowie Instant Messanger greifen beispielsweise auf die Kontaktdaten der Kontakte Verwaltung zurück.

2.1.2 Android Anwendungslebenszyklus

Als Lebenszyklus bezeichnet man verschiedene Zustände von Komponenten. Der Lebenszyklus einer Komponente beginnt mit deren Erzeugung und endet, wenn die Komponente beendet wird. Android kann in den Lebenszyklus von Anwendungen eingreifen, dies wird nötig wenn die Systemressourcen zur Ausführung eines Programmes, durch Belegung von nicht verwendeten Programmen, erschöpft sind. Das Eingreifen von Android in den Lebenszyklus wird in Abschnitt 2.1.3 genauer beschrieben. Die verschiedenen Stationen und Wege im Lebenszyklus einer Activitiy sind in Abbildung 3 aufgeführt. Eine Activity ist eine Klasse, welche für das Darstellen der Benutzeroberfläche benutzt wird.

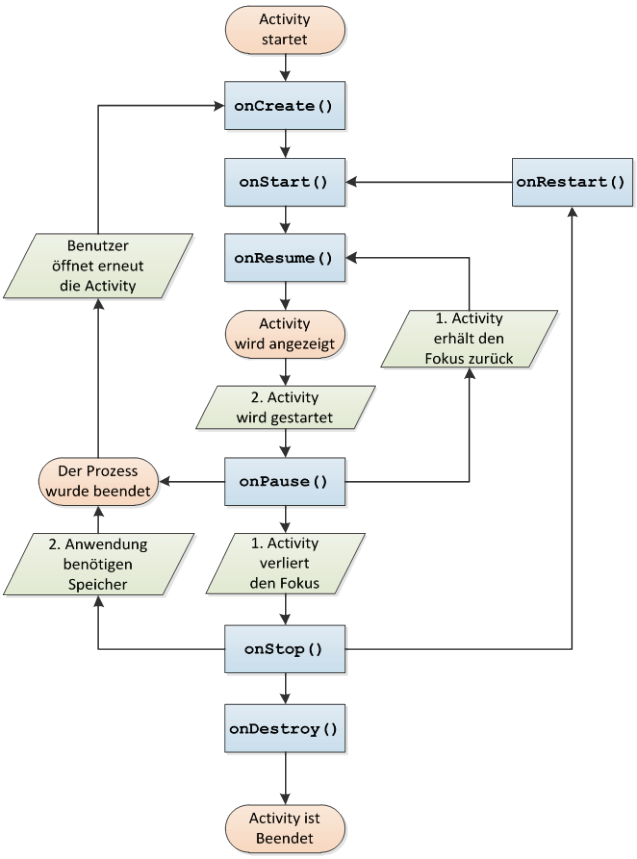


Abbildung 3: Lebenszyklus einer Android Activity

Ausgehend von dem Starten einer Anwendung, wird zunächst die `onCreate` Methode der Activity ausgeführt. Wie alle weiteren Methoden ist der Inhalt dieser frei wählbar. Nach dem Abarbeiten der `onCreate` Methode folgt die `onStart` Methode. Jetzt befindet sich die Anwendung im Vordergrund und ist damit aktiv. Versetzt sich das Smartphone nun beispielsweise in den Standby Modus oder wird eine andere Anwendung aufgerufen, wird die Methode `onPause` aufgerufen. In diesem Zustand bleibt die Anwendung bis sie entweder aufgrund von Ressourcenmangel von Android geschlossen wird, oder vom Nutzer wieder in den Vordergrund gerufen wird. Wird Sie vom Nutzer wieder aufgerufen, erfolgt das Abarbeiten der `onResume` Methode. Die Methode `onStop` verhält sich ähnlich wie `onPause`. Bei einem erneuten Aufruf ist hier der Wiedereinstiegspunkt eine Ebene höher als bei `onStart`, zusätzlich wird die Methode `onRestart` durchlaufen. Auch das gezielte Beenden einer Anwendung, durch den Benutzer, hat einen eigene Methode bekommen, `onDestroy`.

2.1.3 Android Prozess- und Komponenten-Management

Android Anwendungen werden in den seltensten Fällen vom Anwender explizit geschlossen, vielmehr wechselt der Anwender zwischen verschiedenen Anwendungen, wobei die verlassene Anwendung zunächst im Hintergrund verschwindet.

Wie bereits in Abschnitt 2.1.2 angesprochen hat Android die Möglichkeit, in den Lebenszyklus von Anwendungen einzugreifen. Bei knappen Ressourcen beendet Android Prozesse, um Speicher wieder freizugeben. Es wird dabei zwischen verschiedenen Prozesstypen und deren Zustände unterschieden. Eine Activity, die gerade angezeigt wird, hat eine hohe Priorität. Ihr Prozess wird in der Regel nie von Android aufgrund mangelnder Ressourcen beendet. Anwendungen, deren Activity schon länger nicht angezeigt wurden und keine Hintergrund Prozesse wie beispielsweise einen Broadcast Receiver implementieren, werden hingegen sehr schnell vom Betriebssystem beendet.

Im Folgenden werden die verschiedenen Prozesse, beginnend mit der niedrigsten Priorität, aufgelistet und beschrieben.

Leere Prozesse

Prozesse, deren Komponenten bereits beendet wurden, werden nicht zwangsläufig zeitgleich beendet. Diese leeren Prozesse werden, solange ausreichend Ressourcen zur Verfügung stehen, am Leben gehalten. Die Komponenten werden in dieser Zeit in einem Cache gespeichert. Dadurch können diese Anwendungen wesentlich schneller erneut gestartet werden. Leere Prozesse werden bei knappen Ressourcen als erste beendet.

Hintergrundprozesse

Hintergrundprozesse enthalten weder Activities, die gerade angezeigt werden, noch laufende Services. Die beinhalteten Komponenten sind nicht aktiv und nicht für den Benutzer sichtbar. Hintergrundprozesse werden in der Reihenfolge dem Alter entsprechend beendet. Alte, schon lange inaktive Prozesse werden zuerst beendet.

Serviceprozesse

Serviceprozesse beinhalten Remote Services. Sie haben keine Verbindung zu anderen Komponenten und erledigen im Hintergrund ihre Aufgabe. Ein Beispiel für einen Hintergrundprozess ist das Abspielen von Musikdateien, als Dienst einer Musik Anwendung.

Sichtbare Prozesse

Sichtbare Prozesse sind Activitys die sichtbar sind, aber nicht aktiv. Dies ist beispielsweise der Fall, wenn eine Activity teilweise durch eine zweite verdeckt wird. Nur die obenliegende Activity kann zunächst vom Benutzer gesteuert werden. Die dahinterliegende Activity wird meist ausgegraut.

Aktive Prozesse

Die höchste Priorität haben aktive Prozesse. Sie enthalten aktive Komponenten wie beispielsweise eine Activity, die im Vordergrund angezeigt wird, oder einen Broadcast-Receiver, dessen `onReceive` Methode ausgeführt wird.

2.1.4 Android Beispiele

Android wird nicht nur auf Smartphones und Tablet Computer als Betriebssystem verwendet. Es kommen immer weitere Produkte auf den Markt, die während der Anfangszeit von Android sicherlich nicht geplant waren. Neben Autoradios kamen in der vergangenen Zeit vor allem Android TV Boxen auf den Markt. Diese embedded Computer sollen in erster Linie an vorhandene Fernseher angeschlossen werden, um auf Ihnen Videos abzuspielen und im Internet zu surfen. Durch die zahlreichen Android Anwendungen und die in der Regel vorhandenen Schnittstellen wie LAN, WLAN und USB lassen sich die Anwendungsgebiete gut ausbauen.

In folgender Tabelle werden zwei Smartphones und ein Tablet Computer anhand einiger Leistungskenndaten verglichen:

	Huawei U8510 Ideos X3 (6)	LG P990 Optimus Speed (7)	Asus Transformer Prime (8)
Bild			
Marktstart	3. Quartal 2011	1. Quartal 2011	1. Quartal 2012
Android Version	Android 2.3 Gingerbread	Android 2.2 Froyo	Android 4.0 Ice Cream Sandwich
Prozessor	Qualcomm (1x 600MHz)	Nvidia Tegra 2 (2x 1Ghz)	NVIDIA Tegra 3 (4x 1,4GHz)
RAM	265 MB	512 MB	1 GByte
Aktueller Preis	100€	300€	600€

Anhand der Tabelle ist erkennbar, dass es Endgeräte mit Android in den verschiedensten Preisklassen gibt. Das Huawei U8510 richtet sich an Smartphone Einsteiger, mit rund 100€ gehört es zu den preiswertesten Smartphones. Dessen Leistung reicht für alle zur Zeit gängigen Anwendungen, auch grafisch aufwändigere Spiele können mit ihm ohne große Einschränkungen betrieben werden. Das LG P990 ist etwas älter als das Gerät von Huawei, zu Marktstart war es das erste Smartphone mit Dual-Core Prozessor. Zur Oberklasse der mit Android verfügbaren Hardware gehört der Tablet PC Asus Transformer.

2.1.5 Vor- und Nachteile

Die kostenlose Verfügbarkeit ist einer der Vorteile von Android, dies macht es für Smartphone-Hersteller besonders interessant, da sie die grundlegende Entwicklung nicht finanzieren müssen. Auch die Anpassbarkeit und der große Kreis an unterstützer Hardware machen Android interessant.

In der Anpassbarkeit und der Hardware Toleranz liegen gleichzeitig auch Nachteile. Nahezu hat jeder Endgeräte-Hersteller ein eigenes Oberflächen-Design, außerdem gibt es eine Vielzahl von verschiedensten Geräten. Neue Android Versionen sind bei weitem nicht für jedes Gerät verfügbar, zusätzlich gibt es sogenannte Custom ROMs (9), die von Einzelpersonen oder Communities entwickelt werden. Diese sollen die neuen Android Versionen auf nicht offiziell von den Herstellern unterstützen Geräte bringen. Diese enorme Vielfalt an Hardware und Android Versionen führt zur Verwirrung bei Benutzern. Auch Entwickler haben es dadurch schwer sicherzustellen, dass eine Anwendung problemlos auf den verschieden Geräten und Android Versionen läuft.

2.2 OSGi Service Platform

Die OSGi Service Platform ist eine Java-basierte Softwareplattform, die ein dynamisches Modulsystem für Java bereitstellt. Die Softwarekomponenten (Bundles) und Dienste (Services) können zur Laufzeit gestartet, gestoppt, installiert und deinstalliert werden, ohne dass das darunterliegende Framework neugestartet werden muss. Die Spezifizierung erfolgt durch die OSGi Alliance (10). Diese wurde im März 1999 unter dem damaligen Namen „Open Service Gateway Initiative“ von mehreren Software Herstellern, wie z.B. IBM und Oracle gegründet. Anfang 2004 wechselte ihr Name in OSGi Alliance.

2.2.1 Das OSGi Framework

Der Hauptbestandteil der Serviceplattform, ist das OSGi Framework, welches einen Container für die Bundles und Services bereitstellt. Ein Bundle kann eigenständig im Framework installiert werden. Es beinhaltet Klassen und Ressourcen, die fachlich oder technisch eine zusammenhängende Einheit ergeben. Die Ressourcen eines Bundles sind zunächst nicht von außen zugreifbar, hierfür müssen die Packages, in denen die Klassen und Ressourcen liegen, explizit exportiert werden.

Es gibt verschiedene Implementierungen, die dem, von der OSGi Alliance festgelegten, OSGi Standards entsprechen. Einige Beispiele finden sich in Punkt 4.1 wieder. Jede dieser Implementierungen folgt aber der Grundarchitektur, welche sich in fünf Bereiche unterteilen lässt. Der grundlegende Aufbau des Frameworks ist in Abbildung 4 sichtbar. Auf die einzelnen Schichten wird im Folgenden eingegangen.

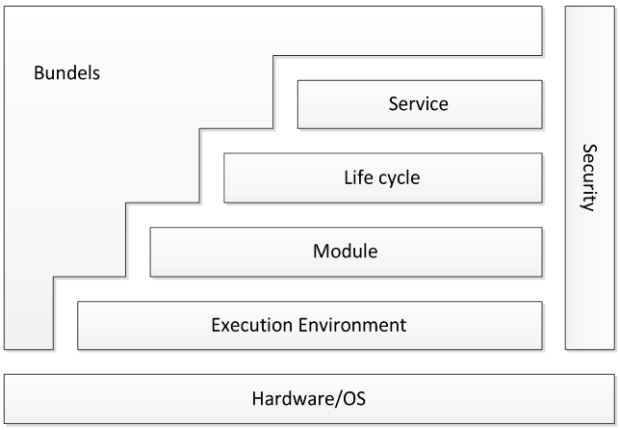


Abbildung 4: OSGi Schichten

2.2.1.1 Execution Environment

Das Execution Environment stellt eine Java Laufzeitumgebung für das Framework bereit. Die Anforderungen an diese sind in Kapitel 999 Execution Environment Specification des OSGi Service compendium (11) spezifiziert.

Innerhalb von Java werden eine ganze Reihe von Execution Environments unterschieden. Für den Betrieb eines OSGi Frameworks können beispielsweise die Voraussetzungen des „OSGi/Minumum-1.1“ Environment angegeben werden. Dies ist eine Ableitung des J2ME Foundation Profils (12) die durch die OSGi Alliance spezifiziert wurde.

2.2.1.2 Module Schicht

Die Module Schicht beinhaltet das Grundkonzept für die Modularisierung. Während das Java Entwicklungsmodel nur eine begrenzte Unterstützung für die Modularisierung bietet, füllt OSGi diese mithilfe generischer und standardisierten Lösungen. In dieser Schicht sind eindeutige und strikte Regeln für das Teilen und Verstecken von Java Paketen unter den verschiedenen Bundles definiert. Die Module Sicht kann ohne die LifeCycle und Service Schicht benutzt werden.

2.2.1.3 Lifecycle Management Schicht

Während die Module Schicht nur das statische Verhalten definiert, erweitert die Lifecycle Management Schicht diese um die für OSGi typischen dynamischen Verwaltungs-möglichkeiten der Bundles. Sie stellt die Lifecycle API für die Bundles bereit, in der das Laufzeitmodell definiert ist. Es beschreibt, wie die Bundles gestartet, gestoppt, installiert, geupdatet sowie deinstalliert werden. Diese Ebene benötigt zwingend die Module Schicht, während die Security Schicht optional ist.

Das Lifecycle Diagramm eines Bundles ist in Abbildung 5 zu sehen. Die einzelnen Zustände der Bundles werden im Folgenden beschrieben.

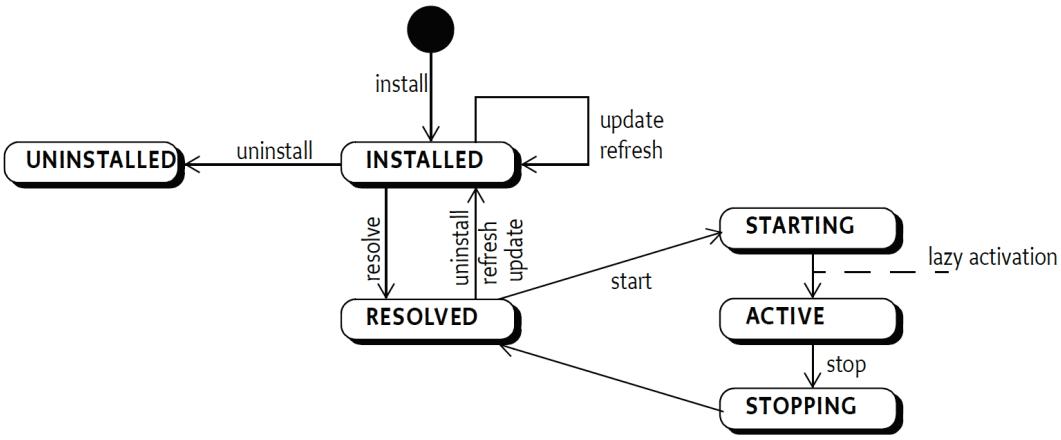


Abbildung 5: Lebenszyklus eines OSGi Bundles (13)

INSTALLED:

Direkt nach der erfolgreichen Installation befindet sich ein Bundle im INSTALLED Status.

RESOLVED:

Nach erfolgreicher Installation versuchen die meisten Frameworks die Package-Abhängigkeiten aufzulösen. Ist dies erfolgreich, wird der Status des Bundles auf RESOLVED gesetzt. Bei manchen Frameworks erfolgt das Auflösen der Abhängigkeiten erst beim ersten Start.

STARTING:

Der Status STARTING kann zwei verschiedene Zustände signalisieren. In der Regel wird hiermit kenntlich gemacht, dass das Bundle seine `start` Methode abarbeitet, aber noch nicht damit fertig ist. Durch das Setzen der Bundle-ActivationPolicy, kann der Start in dieser Phase aber auch verzögert werden, bis das Bundle zum ersten Mal benötigt wird (Lazy starting). Nach erfolgreichem Start, wird das Bundle mit dem Status ACTIVE versehen. Wirft die `start` Methode eine Exception aus, wird der Status zurück auf RESOLVED gesetzt.

ACTIVE:

Nach erfolgreichem Start, befindet sich das Bundle im Status ACTIVE. Auf die von ihm bereitgestellten Dienste kann zugegriffen werden.

STOPPING:

Während der Status STOPPING für das Bundle gesetzt ist, wird dessen `stop` Methode ausgeführt. Anschließend wird dem Bundle der Status RESOLVED zugewiesen.

UNINSTALLED:

Ein Bundle kann aus dem Framework deinstalliert werden. Ist das Bundle entfernt, erhält es den Status UNINSTALLED. In diesem Zustand kann es nicht weiter verwendet werden. Möglicherweise zeigen aber noch Referenzen auf dieses Bundle.

2.2.1.4 Service Schicht

Die Service Schicht bietet ein Programmiermodell für Bundle-Entwickler, welches das Entwickeln von Service Bundles vereinfacht. Dies geschieht durch das Entkoppeln der Service Spezifikation von seiner Implementation. Damit ist es möglich, Services nur unter Berücksichtigung ihrer Interface Spezifikation zu binden, ohne auf deren Implementierung zu achten. Dadurch ist es möglich, bestimmte Implementierungen, die für eine spezielle Anforderung wie beispielsweise eine Hardwareoptimierung, für die gleiche Aufgabe einfach auszutauschen.

Das Framework bietet mithilfe des Framework service registry den Bundles die Möglichkeit verfügbare Implementierungen auszuwählen. Bundles können neue Services registrieren, Benachrichtigungen über den Zustand von Services empfangen und nach existierenden Services suchen, um sich den gegebenen Möglichkeiten des Gerätes anzupassen. Diese Funktionen erhöhen die Flexibilität von OSGi Anwendungen, neue Bundles können installiert werden, um weitere Features bereitzustellen. Bereits installierte Bundles können modifiziert und geupdatet werden, ohne dass das Framework neugestartet werden muss. Mithilfe der Service Schicht lassen sich Serviceorientierte Architekturen (SOA) (14) realisieren.

Die OSGi Spezifikation beinhaltet bereits verschiedene Standard Services, einige von ihnen sind im Abschnitt 2.2.2 aufgelistet und beschrieben.

2.2.1.5 Security Schicht

Die Security Schicht basiert auf der Java 2 security, erweitert diese aber durch einige OSGi spezifische Komponenten. Sie regelt die Berechtigungen der einzelnen Bundles. Die Berechtigungen sind in folgende Rechte eingeteilt:

Package Permission

Für das Importieren und Exportieren von Paketen ist die Package Permission verantwortlich.

Bundle Permission

Die Bundle Permission regelt zusätzlich zur Package Permission die Abhängigkeiten von Paketen. Bundles mit dieser Berechtigung dürfen Pakete von anderen Bundles importieren

und davon abhängig sein. Das jeweils exportierende Bundle benötigt dann auch diese Berechtigung, andernfalls können die Abhängigkeiten nicht aufgelöst werden.

Service Permission

Diese Berechtigung regelt das Anfordern von Services eines Bundles, von der Service Registry. Es besteht die Möglichkeit für jeden Service diese Berechtigung einzeln zu vergeben. Dies schränkt die verfügbaren Services für das Bundle auf ein Minimum ein und gewährleistet das für die Funktion nicht relevante Services nicht zugreifbar sind.

Admin Permission

Mit dieser Berechtigung ist es einem Bundle möglich, administrative Aktionen im Framework auszuführen. Dazu zählen zum Beispiel die Installation von weiteren Bundles und das Starten und Stoppen. Über Filter können diese Rechte eingeschränkt werden. Diese Berechtigung sollte nur wenigen, für die Administration nötigen, Bundles gegeben werden, um die Sicherheit der Anwendung zu erhöhen.

2.2.2 Standard Services

OSGi Standard Services sind in dem OSGi Service Platform Service Compendium bzw. für den Mobilen Anwendungsbereich in der OSGi Service Platform Mobile Specification beschrieben. Sie bieten Lösungen für häufig vorkommende Anforderungen. Einige der spezifizierten Services werden im Folgenden aufgezählt und beschrieben:

Log Service:

Der Log Service bietet die Möglichkeit innerhalb der OSGi Service Platform zentral Nachrichten zu loggen sowie auszulesen.

Configuration Admin Service:

Mit diesem Service können Bundles sowie Services zentral und zur Laufzeit konfiguriert werden.

Declarative Services:

Declarative Services beschreiben ein Programmiermodell, welches das Arbeiten mit dynamischen Services vereinfacht. Hierbei muss sich das Bundle selbst, um die von ihm benötigten Services kümmern. Innerhalb des OSGi Frameworks Apache Felix ist dieser Service unter dem Namen Service Runtime Component (SCR) bekannt.

Event Admin Service

OSGi Bundles müssen in der Regel auf Events von anderen Bundles reagieren oder selbst Events erzeugen. Um der Flexibilität und Dynamik von OSGi gerecht zu werden, wurde der Event Admin Service spezifiziert. Dieser basiert auf dem Publish and Subscribe-Modell (14). Alle Empfänger registrieren sich am Service. Der Sender einer Nachricht sendet diese an den Service, ohne zu wissen von wem, oder ob sie überhaupt empfangen wird. Der Service leitet anschließen die Nachricht an die, passend zur Art bzw. Typ der Nachricht, entsprechenden Empfänger.

2.2.3 OSGi Bundles

Bei einem Bundle handelt es sich technisch gesehen um ein Java-Archiv, welches zusätzliche Informationen in einer Manifest Datei enthält. Dabei beinhalten ein Bundle fachlich oder technisch zusammenhängende Klassen und Ressourcen.

Eine Java Klasse, in der Regel Activator.java benannt, implementiert das Interface der BundleActivator Klasse. Dies erfordert die Implementierung der `start` und `stop` Methoden. Diese werden, falls die Activator Klasse in der Manifest Datei angegeben ist, beim Starten bzw. beim Stoppen des Bundles aufgerufen. Ein Beispiel einer Activator Klasse findet sich im Listing 1. Hier wird beim Start des Bundles der Text „Hello World“ in der Konsole ausgegeben. Beim stoppen des Bundles erfolgt automatisch das Aufrufen der `stop` Methode, welche die Ausgabe „Goodby World“ generiert.

```
public class Activator implements BundleActivator {
	private static BundleContext context;

	public void start(BundleContext bundleContext) throws
			Exception {
		Activator.context = bundleContext;
		System.out.println("Hello World");
		}

	public void stop(BundleContext bundleContext) throws
			Exception {
		Activator.context = null;
		System.out.println("Goodbye World");
		}

	public static BundleContext getContext(){
		return Activator.context;
	}
}
```

Listing 1: OSGi Bundle Beispiel Activator.java

Listing 2 zeigt die zur Activator.java passende Manifest Datei. Hier sind einige Versions Informationen angegeben. Der Bundle Name kann dabei frei gewählt werden. Zur eindeutigen Identifikation dient der Symbolic-Name. Zu importierende sowie zu exportierende Bundles können genauso wie die Mindestanforderungen an die Umgebung in der Manifest Datei festgelegt werden.

```
ManifestVersion: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Hello World
Bundle-SymbolicName: de.example.osgi.helloworld
Bundle-Version: 1.0.0.qualifier
Bundle-Description: Processes android events
Bundle-Activator: de.example.osgi.helloworld.Activator
Import-Package: org.osgi.framework;version="1.3.0"
Export-Package: de.example.osgi.helloworld
Bundle-RequieredExecutionEnvironment: JavaSE-1.6
```

Listing 2: OSGi Bundle Beispiel Manifest.MF

2.2.4 Verschiedene OSGi Frameworks und Anwendungsgebiete

Die offene Spezifikation der OSGi Service Platform führt zu verschiedenen Implementierungen der Plattform. Firmen und Communities entwickeln Open Source sowie kommerzielle Implementierungen der Plattform. Im Folgenden werden einige der Implementierungen aufgezählt:

- **OpenSource**
  - Eclipse Equinox
  - ProSyst mBedded Server Equinox Edition
  - Apache Felix
  - Makewave Knopflerfish
- **Kommerzielle** R4 zertifizierte OSGi Service Platforms:
  - ProSyst Software mBedded Server
  - Makewave Knopflerfish Pro

Anwendung findet die OSGi Service Platform in verschiedenen Bereichen. OSGi wurde ursprünglich für sogenannte Residential Gateways entwickelt. Sie stellen die Schnittstelle zwischen einer Wohnungsumgebung und der Außenwelt dar. Die Funktion ist mit einem Router vergleichbar. Auf der externen Seite steht das Internet zu Verfügung. Im internen Netz können Geräte über verschiedene Techniken über das Residential Gateway ins Internet gelangen, oder in umgekehrter Richtung über das Internet gesteuert werden. Heute wird OSGi in vielen weiteren Anwendungsbereichen eingesetzt, darunter beispielsweise:

- Automotive-Bereich – z.B. Car Infotainmentsysteme (Navigation, Kommunikation)
- Gebäude Automatisierung
- Ambient Assisted Living (AAL)

2.3 OSGi auf Android

Ein OSGi Framework auf einem Android Smartphone zu betreiben bringt grundsätzlich die Vorteile, die überwiegend in der Dynamik und Wiederverwendbarkeit der OSGi Bundles liegen, mit sich. Durch die verschiedenen Architekturmodelle, wird durch den Einsatz von OSGi, eines der Kernelemente des Android Konzeptes außerkraftgesetzt. Ein OSGi Framework läuft, wie die in ihm laufenden Bundles, innerhalb einer Java Virtual Machine. Dadurch ist ein einfacher Zugriff unter den Anwendungen sowie auf gemeinsame Bibliotheken möglich. Im Gegensatz dazu läuft jede Android Anwendung in einer Dalvik Virtual Machine. Dadurch erschwert sich die Interaktion zwischen den Anwendungen leicht, es bringt aber den Vorteil das falls eine Anwendung abstürzt nur die dazugehörige Dalvik VM betroffen ist. Die verschiedenen Konzepte sind in Abbildung 6 zu sehen.

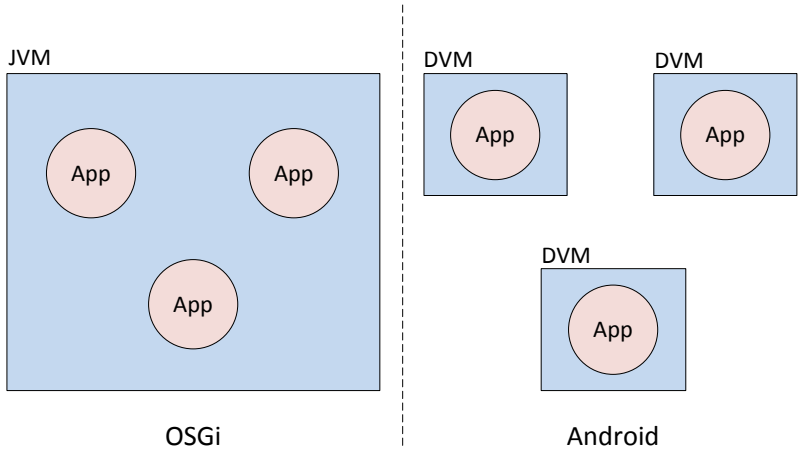


Abbildung 6: OSGi und Android Konzept im Vergleich

2.3.1 Allgemeine Probleme bei dem Betreib von OSGi auf Android

OSGi Frameworks sind in der Regel nicht ohne Anpassungen innerhalb des Android Betriebssystem startbar. Die auftretenden Probleme haben in der Regel den gleichen Ursprung, die Dalvik VM. Die VM unterstützt nicht das dynamische Laden von Klassen, dies ist aber ein Hauptbestandteil von OSGi. Dadurch ist es nicht möglich, auch mit umgewandelten Byte Code ein OSGi Framework ohne vorherige Anpassung zu starten.

Weitere Probleme können durch die Beschränkung der Klassen Bibliotheken hervorgerufen werden. Android unterstütz kein anerkanntes VM Profil, welches eine Standardisierung wesentlich vereinfachen würde.

Die Beschränkungen durch die Dalvik VM gelten als signifikante Hindernisse bei der Entwicklung von „echten“ Java Programmen innerhalb eines Android Systems.

3 Anforderungsanalyse

In diesem Kapitel sollen die Anforderungen des Beispielszenarios analysiert werden. Von diesem werden die Anforderungen für ein möglichst breites Anwendungsfeld erfasst. Dazu wird das Beispielszenario beschrieben und verallgemeinert. Von dieser Grundlage werden schließlich die funktionalen und nicht funktionalen Anforderungen identifiziert. Die für die Beispielanwendung spezifischen Anforderungen werden abschließend in Pflicht- und Wunschkriterien unterteilt und aufgelistet.

3.1 Beschreibung des Beispielszenarios

Im Rahmen der Ambient Assisted Living Forschung an der Fachhochschule Düsseldorf (15) entstand die Bachelor Thesis von Thomas Schmitz. „In diesem Projekt wird ein Softwaresystem entwickelt, mit dem es möglich ist, verschiedene Geräte über ein Android Smartphone zu überwachen und zu steuern.“ (1) Ein noch offenes Problem der Implementierung besteht darin, dass die entwickelte Android Anwendung bzw. vor allem das dort eingebundene OSGi Framework nur arbeitet, wenn die Activity der Anwendung aktiv ist. Öffnete der Benutzer eine andere Anwendung, oder geht das Smartphone in den Standbybetrieb arbeitet das Framework nicht mehr weiter. Deshalb kann nur im aktiven Zustand auf Meldungen von Geräten reagiert werden. Die von den Geräten gesendeten Broadcasts gehen dabei verloren. Das Empfangen der Broadcasts soll durch die Neuentwicklung ständig möglich sein. Die Alarmierung des Benutzers muss ebenfalls ständig ausgeführt werden können.

Das Smartphone soll ständig in der Lage sein, Meldungen von den Sensoren zu empfangen und auszuwerten. In Abhängigkeit von den empfangenen Daten sollen verschiedene Aktionen auf dem Smartphone ausgeführt werden. Gibt der Wassermelder einen Alarm, soll das Smartphone eine Warnmeldung als Benachrichtigung ausgeben.

3.2 Verallgemeinerung des Beispielszenarios

Die Implementierung des OSGi Frameworks innerhalb eines Android Services bietet bereits viele mögliche Anwendungsszenarien. Es könnten bestehende OSGi Programme im Hintergrund von Android laufen und somit die Neuentwicklung von diesen für Android auf einige Anpassungen beschränken. Um eine komfortable Integration zu entwickeln, und diese auch für andere Anwendungsszenarien einfacher zugänglich zu gestalten, ist eine Art

„Framework Manager“ denkbar. In diesem Framework Manager sollte es möglich sein, das Framework zu steuern. Das heißt im genauen

- Framework starten / stoppen
- Status anzeigen gestartet / gestoppt
- Grundlegende Konfiguration vornehmen
- Bundles auflisten, installieren und deinstallieren
- Bundles starten und stoppen

Des Weiteren muss die Kommunikation zwischen den OSGi Bundles und anderen Android Anwendungen ermöglicht werden.

3.3 Funktionale Anforderungen

In diesem Kapitel sollen die funktionalen Anforderungen, die sich aus dem Beispielszenario sowie aus dem verallgemeinerten Szenario ergeben aufgelistet werden. In den anschließend folgenden Unterpunkten werden die jeweiligen Anforderungen genauer beschrieben.

F_1 Bereitstellung eines OSGi Frameworks

F_2 Durchgängige Verfügbarkeit des Frameworks in jedem Telefon Status (ausgenommen Abgeschaltet)

F_3 Spezifische Funktionen des Beispielszenarios

F_4 Benutzerinterface für die Verwaltung des Frameworks

F_5 Bidirektionale Kommunikation zwischen OSGi Bundles und Android Anwendungen

3.3.1 F_1 Bereitstellung eines OSGi Frameworks

F_1.1 Die Software soll ein OSGi Framework enthalten, dieses wird mit Hilfe einer Android Anwendung auf dem Smartphone installiert.

F_1.2 Die Kommunikation zwischen den im Framework laufenden Bundles und dem Android Betriebssystem bzw. weiteren Android Anwendungen soll ermöglicht werden.

3.3.3 F_3 Spezifische Funktionen des Beispielszenarios

F_3.1 Aus dem Framework bzw. den Bundles heraus sollen Benachrichtigungen an den Benutzer geschickt werden können.

F_3.2 Für eine wichtige Benachrichtigung, oder um genaue Daten anzeigen zu können, soll es ermöglicht werden, dass aus einem Bundle heraus Android Anwendungen gestartet werden und mit Daten versorgt werden.

F_3.3 Aus einer Android Anwendung können, durch geeignete Schnittstellen, Informationen aus den Bundles abgerufen werden.

3.3.4 F_4 Benutzerinterface für die Verwaltung des Frameworks

F_4.1 Die Installation von Bundles soll mithilfe des Benutzerinterfaces ermöglicht werden. Das Auswählen der Bundles kann mit einer Texteingabe, in der der Pfad zum Bundle eingegeben werden kann, erfolgen. Denkbar ist auch eine Implementierung eines Datei Browsers für die Auswahl lokal abgespeicherter Bundles.

F_4.2 Der Benutzer soll mithilfe einfacher Start und Stopp Buttons, das Framework manuell starten und stoppen können.

F_4.3 Der Status des Frameworks (gestartet oder gestoppt) sowie eventuelle Log Ausgaben des Frameworks sollen dem Benutzer innerhalb des Benutzerinterfaces angezeigt werden können.

F_4.4 Allgemeine Konfigurationen des Frameworks bzw. dessen Verhalten sollen im Userinterface vorgenommen werden können. Dazu zählt beispielsweise das Autostartverhalten.

3.4 Nicht-funktionale Anforderungen

Die nicht funktionalen Anforderungen wie Zuverlässigkeit und Benutzbarkeit stellen weitere wichtige Kriterien, zu den bereits aufgeführten, dar. Sie werden im Folgenden aufgelistet und anschließend genauer beschrieben.

NF_1 Durchgängige Verfügbarkeit des Frameworks in jedem Telefon Status

NF_2 Geringer Ressourcenbedarf

NF_3 Einfache und benutzerfreundliche Handhabung

3.4.1 NF_1 Durchgängige Verfügbarkeit des Frameworks in jedem Telefon Status

NF_1.1 Nach dem Start des Frameworks läuft dieses durchgehend im Hintergrund weiter.

NF_1.2 Das Framework soll vor einer Beendigung durch Android aufgrund von knapper Ressourcen, weitestgehend geschützt werden.

NF_1.3 Während der verschiedenen Gerätezustände (Aktiv, Standby) ist das Framework durchgehend aktiv und kann auf Ereignisse reagieren.

3.4.2 NF_2 Geringer Ressourcenbedarf

NF_2.1 Der Framework Service sollte auf möglichst leistungsschwacher Hardware lauffähig sein.

NF_2.2 Die für den Betrieb der Software nötige Android Version sollte möglichst niedrig sein, um einen Betrieb auf möglichst vielen Geräten zu ermöglichen.

NF_2.3 Der Energieverbrauch sollte durch das Ausführen des Frameworks nicht maßgeblich erhöht werden.

3.4.3 NF_3 Einfache und benutzerfreundliche Handhabung

NF_3.1 Dem Anwender soll eindeutig angezeigt werden, ob das Framework aktiv ist.

NF_3.2 Die Android Anwendung zum Steuern des Frameworks, zeichnet sich durch hohe Benutzerfreundlichkeit aus. Der grafische Aufbau der Anwendung soll einfach und intuitiv gestaltet werden.

NF_3.3 Die Grundfunktionen wie das Starten und Stoppen des Frameworks, sind einfach zugänglich.

NF_3.4 Der Anwender muss bei Meldungen deutlich benachrichtigt werden können. Benachrichtigungen wie in den Vordergrund tretende Anwendungen, sowie Klingel- und Lichtzeichen sind denkbar.

NF_3.5 Es ist darauf zu achten, dass der Anwender durch das Framework nicht unangebracht gestört wird. Automatisch in den Vordergrund tretende Activitys, sind nur in Ausnahmefällen erwünscht.

3.5 Funktionen des Beispielszenarios

3.5.1 Pflichtenkriterien

P_1 Bereitstellung eines OSGi Frameworks

P_2 Durchgängige Verfügbarkeit des Frameworks

P_3 Senden von Benachrichtigungen an den Benutzer aus OSGi Bundles.

P_4 Automatischer Start des Frameworks

P_5 Manuelles starten und stoppen des Frameworks

P_6 Kennzeichnung, wenn das Framework gestartet ist

P_7 Bidirektionale Kommunikation zwischen OSGi Bundles und Android Anwendungen

3.5.2 Wunschkriterien

W_1 Installation und Deinstallation von Bundles über das Benutzerinterface

W_2 Geringer Ressourcenbedarf

W_3 Log Informationen

3.6 Fazit

In diesem Kapitel wurde das Beispielszenario, welches bereits kurz in der Einleitung beschrieben wurde, genauer ausgearbeitet. Daraus wurde das verallgemeinerte Beispielszenario vorgestellt, welches eine größere Wiederverwendbarkeit sicherstellen soll. Die funktionalen, sowie die nicht funktionalen Anforderungen wurden aufgestellt und anschließend nach Pflicht- und Wunschkriterien sortiert. Die Anforderungen werden in den folgenden Kapiteln genutzt, um Konzepte und Implementierungen zu bewerten.

4 State-of-the-art-Analyse

In diesem Kapitel sollen verschiedene bereits verfügbare Technologien und Ansätze vorgestellt werden. Eine Bewertung hinsichtlich der Verwendbarkeit, sowie der Anpassbarkeit für das Beispielszenario wird im jeweiligen Abschnitt vorgenommen.

4.1 OSGi Frameworks

In diesem Abschnitt werden verschiedene OSGi Frameworks vorgestellt und auf ihre Verwendbarkeit für das Beispielszenario hin überprüft. Durch die Spezifizierung und die teilweise zertifizierten OSGi Frameworks wird in der Regel sichergestellt, dass sich die Frameworks in ihrer Grundfunktion kaum unterscheiden und damit in der Regel einfach austauschbar sind. Die Lauffähigkeit eines Frameworks innerhalb einer DalvikVM ist damit allerdings noch nicht gegeben. Im Folgenden werden einige bekannte Frameworks vorgestellt und auf ihre Android Kompatibilität eingegangen.

4.1.1 Eclipse Equinox

Equinox (16) zählt zu den bekanntesten OSGi Frameworks. Equinox ist eine Implementation des OSGi R4 Core Frameworks und bietet zusätzlich einige optionale OSGi Services. Die Entwicklung wird von der gemeinnützigen Gesellschaft Eclipse Foundation geleitet. Die hohe Bekanntheit verdankt Equinox vor allem der integrierte Entwicklungsumgebung Eclipse. Equinox bildet das Grundgerüst für Eclipse und ermöglicht das dynamische Installieren, Updaten und Starten von Plugins (Bundles). Auf der EclipseCon, der Entwickler Konferenz der Eclipse Foundation, wurde im Jahr 2008 die erfolgreiche Portierung des Equinox Frameworks auf Android vorgestellt (17). Seitdem wurden keine weiteren offiziellen Informationen über die Lauffähigkeit oder Eignung von Equinox bekannt. Alle sonstigen Einträge in diversen Foren beziehen sich auf die 2008 vorgestellte Präsentation.

4.1.2 ProSyst

Die Firma ProSyst vertreibt Ihre eigene Implementation der OSGi Service Platform nach der Spezifikation 4.2. ProSyst bietet für verschiedene Bereiche die eigene OSGi Umgebung sowie passende Entwicklungstools. Dabei liegt der Schwerpunkt bei Embedded Devices. Zu den Bereichen zählen unter anderem:

- General Purpose OSGi
    - mBS OSGi Runtime: OSGi Umgebung optimert für Embedded Devices
- Smart Home
    - mBS Smart Home OSGi Runtime: OSGi Umgebung und Middlewares z.B. für Breitband Router
- Mobile
    - mBS Mobile OSGi for Android: OSGi Umgebung und Middleware für Android Geräte
- Telematics
    - mBS Telematics OSGi Runtime: OSGi Umgebung und Middleware für in-vehicle systems

(18)

Das ProSyst Produkt mBS Mobile for Android stellt die zur Zeit einzige, auch kommerziell vertriebene OSGi Lösung unter Android dar. Die Anwendung kann frei heruntergeladen werden, ist aber nicht quelloffen und in dieser Form nur für nicht kommerzielle Anwendung. Dadurch, dass es sich hierbei nicht um eine Entwicklergemeinde handelt und die Produkte in der Regel von Resellern genutzt werden, ist der freie Support stark beschränkt.

Die versprochenen Funktionen sind allerdings vielversprechend. Die implementierte OSGi Spezifikation der Version 4.1 aus dem Jahr 2007 ist nicht mehr aktuell. ProSyst wirbt vor allem mit der guten Integration des Framewoks innerhalb des Android Betriebssystems. Einige der Funktionen, entnommen und übersetzt aus der Produktbeschreibung (19) sind im Folgenden aufgeführt.

- OSGi Core 4.1 und OSGi Mobile 4.1 konform
- Integration von Android Intents mithilfe des OSGi Event Admins (Bi-direktionaler Austausch von Events/Intents)
- Zugriff von Android Anwendungen auf OSGi Services durch Android IDL, Intents oder Lokale Webservices
- Installation von OSGi Content mithilfe des Android Web Browsers

Sucht man in der Android Anwendung Google Play Store (früher Android Market) nach dem Suchbegriff „OSGi“ so erhält man genau einen Treffer, die ProSyst mBS Mobile OSGi Eval Anwendung. Dabei handelt es sich um die beschriebene Android OSGi Middleware von Prosyst. Bei der Installation fällt auf, dass die Anwendung bei den Android Rechten aus dem Vollen schöpft. Dies ist zwar nötig um installierte Bundles mit möglicherweise

benötigten Rechten zu versorgen, wirkt aber für viele Nutzer abschreckend. Das Gewähren von vielen Rechten kann zu Sicherheitslücken führen. Dadurch, dass beinahe alle Rechte der Middleware eingeräumt werden, sollte die Middleware dafür sorgen, das installierte Bundles nicht zwangsläufig alle Android Dienste die mit den Rechten freigeschaltet sind verwenden kann. Diese Funktionalität ist aber weder in der Dokumentation beschrieben, noch in der Anwendung selbst ersichtlich. Bei einer quelloffenen Middleware bestünde zumindest die Möglichkeit für ein bestimmtes Projekt die Rechte auf die eigenen Anforderungen abzustimmen.

4.1.3 Apache Felix

Apache Felix ist eine Open Source Toplevel-Project der Apache Software Foundation. Auch Felix ist eine OSGi Release 4.2 Implementierung und wird, wie Equinox, als Grundgerüst für eine integrierte Programmierumgebung verwendet. Die IDE NetBeans basiert auf dem Felix Framework, ein weiteres Beispiel für den Einsatz von Felix der ist Applikationsserver GlassFish.

Apache Felix ist seit Version 2.0.0 ab der Android Version 1.5 ohne Einschränkungen auf Android lauffähig. Die offizielle Felix Webseite bietet eine kurze Anleitung wie Felix innerhalb von Android gestartet werden kann (20). Zusätzlich kann das komplette Framework in Form eines jar-Archives als Bibliothek in einer Android Anwendung eingefügt werden. Eine Instanz von Felix kann anschließend erzeugt und gestartet werden.

Apache Felix bietet zudem eine aktive Entwicklergemeinde. Während der Anfertigung dieser Bachelorthesis habe ich die Emails der Felix User Group mitverfolgt, erste Antworten auf Fragen folgten in der Regel innerhalb von wenigen Stunden. In einem Zeitraum von knapp zwei Monaten entstanden so rund 270 Nachrichten, allerdings enthielt dabei keine ein Thema, das mit Android in Zusammenhang stand.

Während in der User Group, in den vergangenen zwei Monaten, keine mit Android im Zusammenhang stehenden Nachrichten auftraten, wird Felix durchaus unter Android verwendet. Einige dieser Projekte werden im folgenden Kapitel vorgestellt.

4.2. Vorstellung von bestehenden OSGi/Android Anwendungen

Wie bereits in Kapitel 4.1.3 angesprochen, gibt es bereits Android Anwendungen die OSGi verwenden. Dabei spielt das OSGi Framework Apache Felix eine große Rolle, da es in allen Projekten als OSGi Framework Verwendung findet.

4.2.1 IST-MUSIC

Das Akronym MUSIC steht für „Self-Adapting Applications for Mobile USers In Ubiquitous Computing Environments" (21). Dabei handelt es sich um eine OSGi Middleware die auf verschiedenen Betriebssystemen lauffähig ist. Zu den unterstützen Betriebssystemen zählen: Android, Windows, Windows Mobile, und Linux. Das MUSIC Projekt wurde von einem Konsortium aus verschiedenen Firmen und Hochschulen, unter anderem von Hewlett Packard und der Uni Kassel, entwickelt. Mithilfe des EU Fördergeldes in Höhe von 14,5 Millionen Euro eine Middleware deren Funktionen zwar vielversprechend aussehen, aber bislang keine Verwendung findet. Auch die Bekanntheit ist sehr gering. Während der Recherche zu dieser Bachelor Thesis, wurde in den durchsuchten Quellen das MUSIC Projekt nie erwähnt, obwohl das Projekt bereits im Oktober 2006 startete und im Mai 2010 beendet wurde.

MUSIC wurde mit Felix, Equinox sowie Knopflerfish getestet und ist auf diesen OSGi Plattformen lauffähig. Die Android Version verwendet das Felix Framework.

Eine aktive Entwicklergemeinde gibt es nicht. Im offiziellen Forum des Projektes (22) gibt es nur wenige Einträge, obwohl es hier nicht nur um die Android Version geht. Eine von mir erstellte Anfrage auf weitere Informationen zur Interaktion zwischen OSGi Bundles und Android Anwendungen wurde bis heute (zwei Monate nach Anfrage) im Forum nicht beantwortet.

4.2.2 OSAmI-FIN-Demo

Das Projekt OSAMI-FIN-Demo (23) verwendet unteranderem OSGi auf einem Android Smartphone. Bei dem Projekt handelt es sich um eine Demonstrationsumgebung zu dem „Open Source Ambient Intelligence Commons for an Open and Sustainable Internet“ (24) Projekt. Das Ziel des OSAmI Projekts ist eine einheitliche Plattform für Anwendungen in intelligenten Umgebungen.

Die OSAmI-FIN-Demo verfügt über keine Dokumentation, lediglich einige Programmzeilen sind innerhalb des Quellcodes beschrieben. Aus dem Code ist ersichtlich, dass die dort vorhandenen OSGi Bundles mithilfe eines ProSyst Framewoks auf Android laufen sollen.

4.3 Fazit

In diesem Kapitel wurden drei OSGi Frameworks vorgestellt. Die Eignung von diesen für den Betrieb unter Android ist nur bei Apache Felix und der ProSyst Lösung gegeben. Durch die Quelloffenheit, die im Verhältnis hohe Verbreitung, sowie die aktive Entwicklergemeinde von Apache Felix qualifizieren dieses OSGi Framework im Besonderen für den Einsatz unter Android.

5 Konzeption

In diesem Kapitel sollen zunächst verschiedene Konzepte für die Realisierung der Anwendung beschrieben werden. Anschließend wird das Konzept mit der besten Eignung ausgewählt und als Detailkonzept ausgearbeitet. Im Kapitel Implementierung wird anschließend dieses prototypisch als proof-of-concept umgesetzt, um die Erfüllbarkeit der Anforderungen durch das Konzept zu validieren.

Das Konzept gliedert sich in drei Teile, die jeweils einzeln analysiert werden.

- Bereitstellung des OSGi Frameworks
- Implementierung des Benutzerinterfaces
- Kommunikation zwischen dem Framework und anderen Android Anwendungen

5.1 Grundkonzepte

An dieser Stelle soll auf Basis der Erkenntnisse der vorhergehenden Kapitel für die drei Unterpunkte der Konzeption jeweils verschiedene Realisierungsmöglichkeiten vorgestellt werden.

5.1.1 Bereitstellung des OSGi Frameworks

Durch die Anforderung, dass das Framework durchgängig zu Verfügung stehen soll, bleibt nur die Möglichkeit das Framework innerhalb eines Android Services zu starten.

Android Services sind dafür vorgesehen im Hintergrund Aufgaben zu erledigen. Ein weit verbreitetes Beispiel ist hierfür eine Audioplayer Anwendung, die das Abspielen von Musik im Hintergrund innerhalb eines Services verrichtet.

Ein Service ist dabei standardmäßig kein separater Prozess, solange dies nicht anders angegeben wurde, läuft er im gleichen Prozess wie die Anwendung selbst. Ein Service ist zudem auch kein Thread. Rechenintensive Funktionen können deshalb zu „Application Not Responding errors“ (ANR) führen. Der Lebenszyklus eines Services hängt in der Regel von seiner momentanen Verwendung ab. Grundsätzlich versucht Android den Prozess, indem ein Service läuft, nicht zu beenden, solange der Service gestartet ist oder Clients mit ihm verbunden sind.

Der Service kann manuell in einen Status gebracht werden, indem er von Android als wichtiger und deshalb nur in äußersten Extremfällen durch Android beendet wird. Mit der seit Android API Version 5 vorhanden Methode `startForeground` kann dies erreicht werden. Für ältere Android Versionen steht die Methode `setForeground` zu Verfügung.

5.1.2 Implementierung der GUI

Für die Implementierung des Benutzerinterfaces der im Framework abgebildeten Geräte gibt es grundsätzlich zwei Varianten. Eine ist die native Android Anwendung, die nicht innerhalb des Frameworks läuft. Es handelt sich dabei um eine Android Anwendung, die nur über geeignete Schnittstellen für die Kommunikation zum entsprechenden Bundle verfügt. Die andere Möglichkeit ist eine GUI aus einem Bundle heraus zu erstellen. Dabei würde das Bundle mithilfe geeigneter Schnittstellen auf eine Activity zugreifen und Objekte gegebenenfalls ändern.

Die native Android Anwendung hat die Vorteile, dass sie durch die bereitgestellten Tools komfortabel mit einem grafischen Editor erstellt werden kann. Außerdem kann so weitestgehend der OSGi Code von Android Code getrennt werden. Der Nachteil liegt in der dadurch leicht begrenzten Dynamik. Durch das Sicherheitskonzept von Android ist es nicht möglich eine Anwendung durch eine Anwendung zu installieren, ohne das der Nutzer dies bestätigt.

Das Integrieren der GUI innerhalb eines Bundles hätte den Nachteil der nativen Android Anwendung nicht. Das Bundle könnte durch das Framework, genauso wie andere Bundles auch, heruntergeladen und im Framework installiert werden. Die GUI innerhalb eines Bundles zu platzieren bringt aber deutliche Schwierigkeiten mit sich. Nur der Hauptthread kann auf die GUI Elemente einer Activity zugreifen. Dies kann aber auch nur dann erfolgen wenn die Elemente zuvor in der Layout.xml Datei festgelegt wurden. Ein echtes dynamisches Erzeugen einer Oberfläche ist unter Android nicht vorgesehen. Des Weiteren würde bei einem großen Projekt mit vielen verschiedenen Geräten das Framework möglicherweise durch die Vielzahl von GUIs überladen.

5.1.3 Kommunikation zwischen dem Framework und anderen Android Anwendungen

Unabhängig von der Auswahl der Implementierungsmethode des Benutzerinterfaces sollte eine Möglichkeit zur Interaktion zwischen OSGi Bundles und Android Anwendungen gefunden werden. Android bietet hierfür im Wesentlichen vier Möglichkeiten.

- Binder
- Messenger
- Android Interface Definition Language (AIDL)
- Android Intents

Hiervon sind die ersten drei für die Verwendung mit Service Klassen vorgesehen. Die Auswahl der geeigneten Methode hängt von der vorliegenden Struktur ab.

Falls der Service nicht von außerhalb der Anwendung erreichbar sein muss und zudem im gleichen Prozess wie der Client läuft, ist die einfachste Art der Kommunikation das Erweitern der Binder Klasse. Der Client kann direkt auf die public Methoden des Services, sowie der Methoden des Binders selbst zugreifen. Ist es notwendig, dass der Service auch von anderen Anwendungen oder über Prozessgrenzen hinweg genutzt werden soll, scheidet diese Art der Interface Beschreibung aus.

Für den Fall, das die Prozessgrenzen überschritten werden müssen, bietet Android die Möglichkeit einen Messanger zu nutzen. Hierfür wird ein Messanger sowie ein Handler innerhalb des Services definiert. Der Handler kann anschließend auf die vom Client gesendeten Nachrichten reagieren. Alle Nachrichten werden zunächst in eine Warteschlange geschrieben, die innerhalb eines Threads bearbeitet werden. Dadurch wird einerseits der Service automatisch Threadsafe, andererseits geht dabei die Möglichkeit zur parallelen Verarbeitung mehrere Nachrichten verloren.

Durch das Nutzen der Android Interface Definition Language kann, im Vergleich zum Messanger, der Nachtteil der nicht unterstützten parallelen Kommunikation beseitigt werden. Mit einer vom Entwickler erstellten .aidl Datei, die das Interface beschreibt, wird mithilfe der Android SDK Tools eine abstrakte Klasse erzeugt, die das Interface implementiert und die IPC verarbeitet. Der Nachteil von AIDL liegt dabei in der verhältnismäßigen aufwändigen Implementierung, außerdem muss sichergestellt sein, dass der Service Threadsafe ist.

Eine weitere Möglichkeit der Kommunikation sind Android Intents. Sie sind dabei nicht an Service Klassen gebunden und vielseitig einsetzbar. Bei ihnen handelt es sich um Nachrichten, die entweder an einen bestimmten Empfänger, wie eine Activity, oder systemweit per Publish and Subscribe-Modell versendet werden. Intents können dabei alle serialisierbaren Objekte angehängt bekommen.

Die vier Kommunikationsarten werden jeweils im Folgenden genauer beschrieben.

Binder

Die einfachste Art Methoden eines Services aufzurufen ist das instanziieren der Binder Klasse innerhalb des Services. Die Implementierung erfolgt in drei Schritten:

- Innerhalb des Services eine Instanz der Binder Klasse erzeugen die entweder:
    - Öffentliche Methoden beinhaltet die der Client aufrufen kann
    - Die die aktuelle Serive Instanz zurückgibt, welche öffentliche Methoden enthält
    - Oder eine Instanz einer anderen Klasse, die sich innerhalb des Services befindet und öffentliche Methoden bereitstellt.
- Diese Instanz des Binders wird durch die `onBind` callback Methode zurückgegeben
- Der Client erhält die Instanz durch die `onServiceConnected` callback Methode und kann somit die übergebenen öffentlichen Methoden nutzen

Messenger

Bei einem Messenger handelt es sich um einen Verweis auf einen Handler. Dies ermöglicht die Implementierung einer auf Nachrichten basierter Kommunikation über Prozessgrenzen hinweg. Es wird ein Messanger generiert der auf einen Handler innerhalb eines Prozesses erzeugt, anschließend wird der Messanger anderen Prozessen zur Verfügung gestellt. Im Folgenden wird der Ablauf der Kommunikation stichpunktartig beschrieben:

- Der Service implementiert einen Handler, welcher einen Rückruf für jeden Aufruf eines Clients erhält
- Der Handler erzeugt ein Messenger Objekt, welches eine Referenz zum Handler ist
- Der Messanger erzeugt einen IBinder, welcher vom Service den Clients zurückgegeben wird – ausgehend von der `onBind` Methode
- Clients erstellen mithilfe des IBinders eine Instanz des Messangers, welcher den Service Handler referenziert. Der Messanger wird vom Client für das Senden des Message Objekts an den Service genutzt.
- Der Service empfängt jede Nachricht in seinem Handler, genauer gesagt innerhalb seiner `handleMessage` Methode.

Bei dieser Vorgehensweise gibt es keine Methoden des Services, die der Client aufruft. Er sendet lediglich Nachrichten (Message Objekte) die vom Handler des Services empfangen werden. Die Auswertung der Nachricht erfolgt im Handler des Services, je nach Inhalt der Nachricht können anschließend Methoden aufgerufen werden.

Im Vergleich zu AIDL ist diese Art der Kommunikation nicht Multi-Thread fähig. Der Service kann so nur eine Nachricht nach der anderen verarbeiten, da die Messenger Warteschlange auf den Service zeigt.

Android Interface Definition Language (AIDL)

Die Android Interface Definition Language ähnelt anderen IDL's. Mit ihr ist es möglich, die Schnittstellen für die Kommunikation zwischen Client und Service zu beschreiben, welche auf der Interprozess Kommunikation aufbaut. Der hierfür übliche benötigte Code für das Marshalling ist laut dem Android Developer Guide „tedius to write" (25), welches mit „mühsam oder langweilig zu schreiben ist" übersetzt werden kann. AIDL vereinfacht die Implementierung, indem der Großteil des Codes automatisch erzeugt wird. Für die Verwendung dieser Art der IPC muss der Entwickler zunächst eine .aidl Datei erstellen. In dieser wird mithilfe von Javacode ein Interface mit einer oder mehreren Methoden, welche Parameter aufnehmen und zurückgeben definiert. Das Android Developer SDK erstellt aus der .aidl Datei ein Interface mit einer abstrakten Klasse Stub. Die Klasse Stub wird durch die Methoden der Binder Klasse und die Methoden der .aidl Datei erweitert. Anschließend kann das Interface durch die Implementierung des Services und das Überschreiben dessen `onBind` Methode genutzt werden.

Im Android Developer Guide steht der Hinweis das AIDL nur genutzt werden sollte, falls es nötig ist das Clients von verschiedenen Programmen auf den Service zugreifen und der Service Mutlithrading unterstützen soll. Für alle anderen Fälle ist eine andere Art von Kommunikation vorzuziehen.

Neben dem zuvor aufgeführten Hinweis ist zusätzlich die Problematik, dass Android Anwendungen die einen Alarm eines OSGi Bundles verarbeiten sollen, ständig aktiv und mit dem Service gebunden sein müssen. Es ist zudem darauf zu achten, dass bei Änderungen innerhalb der .aidl Datei sich das Interface entsprechend ändert. Client und Service benötigen daher für eine uneingeschränkte Interaktion zwingend die gleiche .aidl Datei mit dessen generierten Klassen. Der Vorteil von AIDL besteht in dem prinzipiellen größten Anwendungsbereich. Auch C-Programme, die innerhalb von Android für möglichst gute Performance eingesetzt werden, können das per AIDL definierte Interface nutzen.

Android Intents

Intents sind betriebssystemweite Aufrufe, die für die späte Laufzeitbindung zwischen Komponenten innerhalb einer oder verschiedener Anwendungen gedacht sind. Sie sind einfach zu implementieren und benötigen wenige Systemressourcen.

Unterschieden werden Android Intents in Implizierte und Explizierte Intents. Explizierte Intents richten sich an bestimmte Java Klassen, von denen Sie empfangen werden. Implizierte Intents werden anhand von drei Eigenschaften einem oder mehreren Empfängern zugewiesen. Diese Eigenschaften sind:

- Action
- Type
- Catagory

Die Auflösung des Empfängers der implizierten Intents erfolgt dabei durch Android. Alle installierten Anwendungen, in denen ein Intent Receiver in der Manifest.xml definiert ist, sowie alle registrierten Broadcast Receiver, sind dem System bekannt. Falls ein oder mehrere Filter auf den aufzulösenden Intent zutreffen, wird dieser an den entsprechenden Empfänger weitergeleitet.

Dabei ist es nicht zwingend erforderlich, dass der Empfänger, beispielsweise eine Activity, zum Zeitpunkt der Aufrufes ausgeführt wird. Activitys können mithilfe von Intents gestartet werden.

Das Anhängen von Nutzdaten an Intents ist mit der `putExtra` Methode möglich. Es können alle serialisierbaren Objekte mitgeschickt werden.

5.1.4 Auswahl des jeweils am besten geeigneten Konzepts

Die Bereitstellung des Frameworks soll durch einen Services, der den Status „Foreground“ erhält, erfolgen. Die GUI zur Steuerung des Frameworks wird innerhalb der Android Anwendung, die auch das Framework bereitstellt, realisiert. Für die Anzeige der Wassermelder Daten wird eine weitere Android Anwendung entwickelt. Diese nutzt als Kommunikationsmöglichkeit zu den OSGi Bundles Intents. Sie erlauben eine dynamische Kopplung zwischen den zwei Anwendungen. Außerdem ist es mit ihnen möglich aus einem OSGi Bundle eine Android Anwendung zu starten.

5.2 Detail Konzept

Das zu entwickelnde Software System soll der in Abbildung 7 zu sehenden Struktur entsprechen.

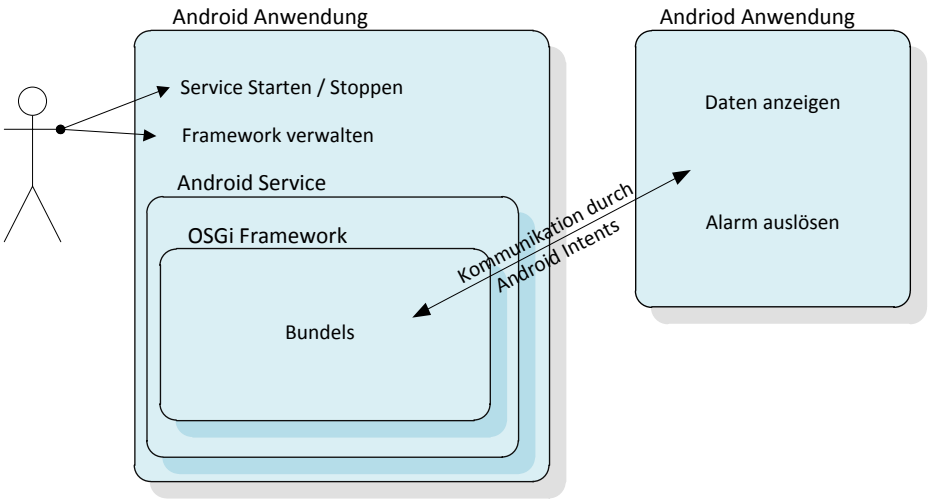


Abbildung 7: Detail Konzept

Die linke Android Anwendung beinhaltet das Apache Felix OSGi Framework. Dies läuft innerhalb eines Services welcher den Status „Foreground“ erhält, um eine Beendigung durch Android vorzubeugen. Die Android Anwendung bietet dem Nutzer die Möglichkeit den Service manuell zu starten und zu stoppen.

Eine zweite Android Anwendung ist für die Darstellung der Geräte Daten und die Alarmierung zuständig. Die Kommunikation zwischen den OSGi Bundles und der zweiten Android Anwendung erfolgt mithilfe von Intents.

5.5 Fazit

In diesem Kapitel wurden verschiedene Möglichkeiten zur Realisierung des gewünschten Softwaresystems beschrieben. Das entwickelte Konzept ist nicht auf das Beispielszenario beschränkt. Die Bereitstellung des OSGi Frameworks als Android Service sowie die Entwicklung einer Android Anwendung, die in der Lage ist mit OSGi Bundles zu kommunizieren, stellt ein grundlegendes Konzept dar. Mit diesem Konzept können prinzipiell alle vorhandenen OSGi Anwendungen, mit entsprechender GUI Anpassung auf Geräten mit dem Android Betriebssystem verwendet werden.

6 Implementierung

In diesem Kapitel wird im Detail darauf eingegangen, wie die prototypische Umsetzung des ausgewählten Konzeptes ausgeführt wurde. Dabei werden zunächst Einschränkungen und Besonderheiten dargestellt, die während der Implementierung eine Rolle spielten. Anschließend wird auf den Gesamtaufbau der Software eingegangen, um einen Überblick über die gesamte Anwendung und deren Umgebung zu erlangen.

6.1 Einschränkungen und Besonderheiten

Das Apache Felix Framework erbt von der umhüllenden Android Anwendung die Berechtigungen. Um in Bundles bestimmte Funktionen wie das Schreiben auf die Speicherkarte nutzen zu können, muss die Anwendung zudem das Framework über die entsprechenden Android Permissions verfügen.

Die Android Context Klasse hat sich während der Implementierung als eine Besonderheit zu den anderen Android Klassen herausgestellt. Bei ihr handelt es sich um eine abstrakte Klasse, welche das Interface zur Anwendungsumgebung darstellt. Sie wird unter anderem für das Senden und Empfangen von Intents verwendet. Während es für alle verwendeten Android Klassen reichte, diese als externe Klassen zu definieren, musste für die Kontext Klasse ein OSGi Service implementiert werden, der eine Referenz auf diese Klasse bereitstellt.

Eine weitere Besonderheit besteht bei der Auswahl des Threadtyps indem das Framework läuft. Mit dem ersten Ansatz das Framework in einem einfachen Thread zu starten war es nicht möglich, die zuvor erwähnte Andorid Context Klasse zu nutzen, um beispielsweise eine Benachrichtigung anzeigen zu lassen. Erst durch die Verwendung eines HandlerThreads und dem Aufrufen eines Loopers, konnten die auftretenden Exeptions beseitigt und die gewünschte Funktionalität erreicht werden.

Der OSGi Service zur Übergabe der Context Klasse sowie der HandlerThread werden in Kapitel 6.3.1.2 genauer beschrieben.

6.2 Aufbau der Anwendung

Während der Implementierung wurden drei Programme entwickelt:

- FHD Felix Manager: Android Anwendung, die das Apache Felix Framework beinhaltet.
- Wassermelder: Android Anwendung, die die GUI für den Wassermelder bereitstellt, sowie im Alarmfall die Alarmierung des Nutzers durchführt.
- Wassersimbundle: OSGi Bundle, das auf die bestehende Struktur der zuvor entwickelten Software zugreift, die Wassermelder Sensordaten empfängt und diese an die Wassermelder Anwendung weitergibt.

Der Aufbau des kompletten Systems ist in Abbildung 8 abgebildet.

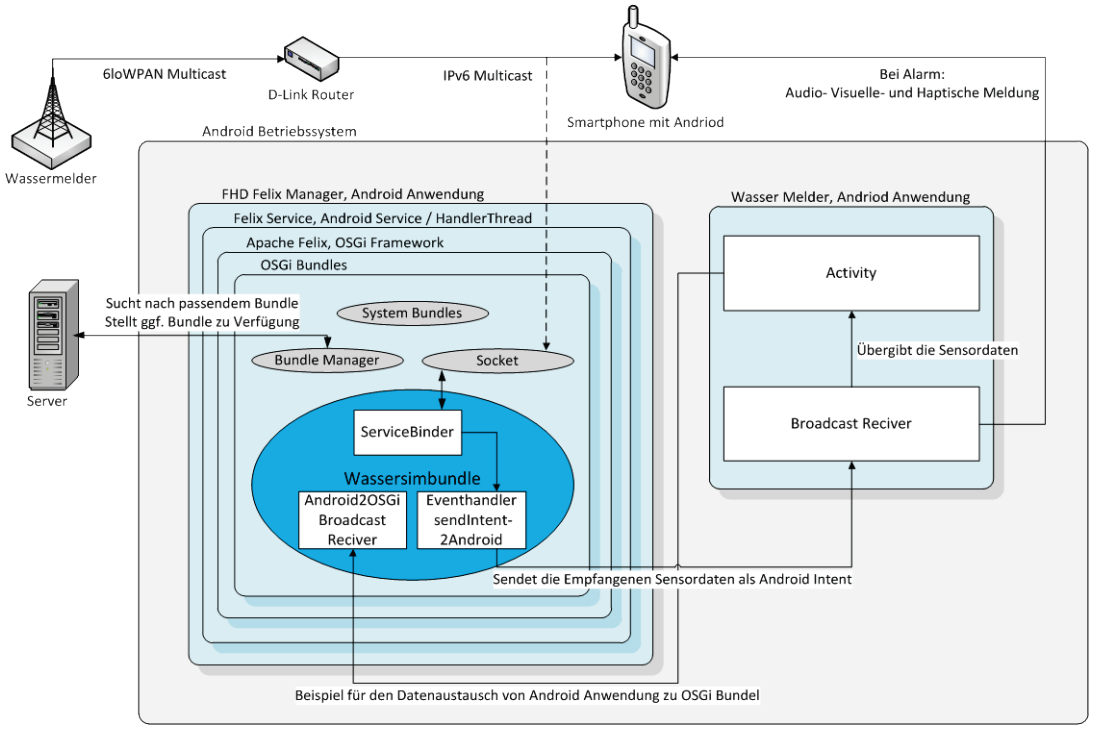


Abbildung 8: Gesamtaufbau der Software

Die Android Anwendung FHD Felix Manager stellt die erforderliche Umgebung für das Apache Felix Framework bereit. Das Framework befindet sich innerhalb einer Service Klasse, die einen Thread startet. In diesem Thread wird eine Instanz des Framework gestartet.

Innerhalb des Frameworks laufen die OSGi Bundles, darunter auch das Wassersimbundle. Die Grau eingefärbten OSGi Bundles, Bundle Manager, Socket und die System Bundles wurde unverändert aus der Bachelor Thesis von Thomas Schmitz (1) übernommen. Das Wassersimbundle greift auf der einen Seite auf das vorhandene Socket Bundle zu, um die

dort ankommenden Sensor Daten per OSGi Event zu erhalten. Die Klasse sendIntet2Android innerhalb des Bundles nimmt die erhaltenen Daten und schickt diese per implizierte Intents an Android. Der Intent mit den angehängten Daten wird anschließend von der Wassermelder Anwendung empfangen. Das Empfangen von Intents innerhalb des Wassersimbundle wird mit der Android2OSGi Klasse erreicht. In ihr ist ein Broadcast Receiver implementiert, der auf bestimmte Intents reagieren kann.

Neben dem FHD Felix Manager ist eine zweite Android Anwendung entwickelt worden. Sie empfängt die vom Wassersimbundle gesendeten Intents, wertet diese aus, zeigt diese in der Activity an und startet die Alarmbenachrichtigungen.

Durch die Bereitstellung eines OSGi Services welcher die Android Context Klasse zur Verfügung stellt, ist es möglich innerhalb eines Bundles den Android Context zu nutzen. Dies wird im entwickelten Wassersimbundle verwendet um Android Intents zu senden und zu empfangen.

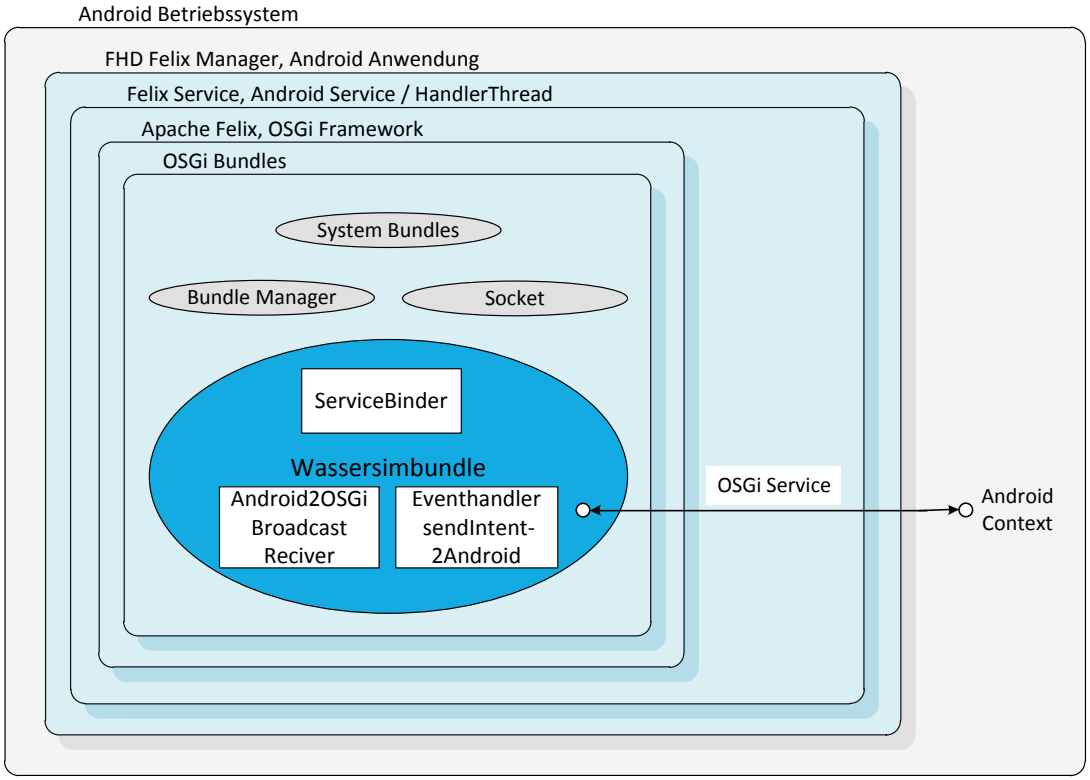


Abbildung 9: Aufbau des FHD Felix Manager

Abbildung 10 zeigt dem Verlauf der vom Wassermelder gesendeten Alarmdaten. Der Wassermelder sendet zunächst per 6loWPAN seine Daten, welche vom D-Link Router auf WLAN bzw. IPv6 Broadcast umgesetzt werden. Vom Smartphone empfangen, gelangen die Daten zum OSGi Bundle Socket. Die Nachricht des Wassermelders wird anschließend als OSGi Event an das Wassersimbundle weitergeleitet. Hier wird aus dem OSGi Event ein Android Intent erstellt und in das Betriebssystem gesendet. Der Broadcast Receiver der

Android Wassermelder Anwendung nimmt dieses entgegen und wertet die Daten aus. Die Informationen werden der Activity als Strings übergeben und, da es sich um einen Alarm handelt, die Methoden zur Benachrichtigung des Nutzers aufgerufen.

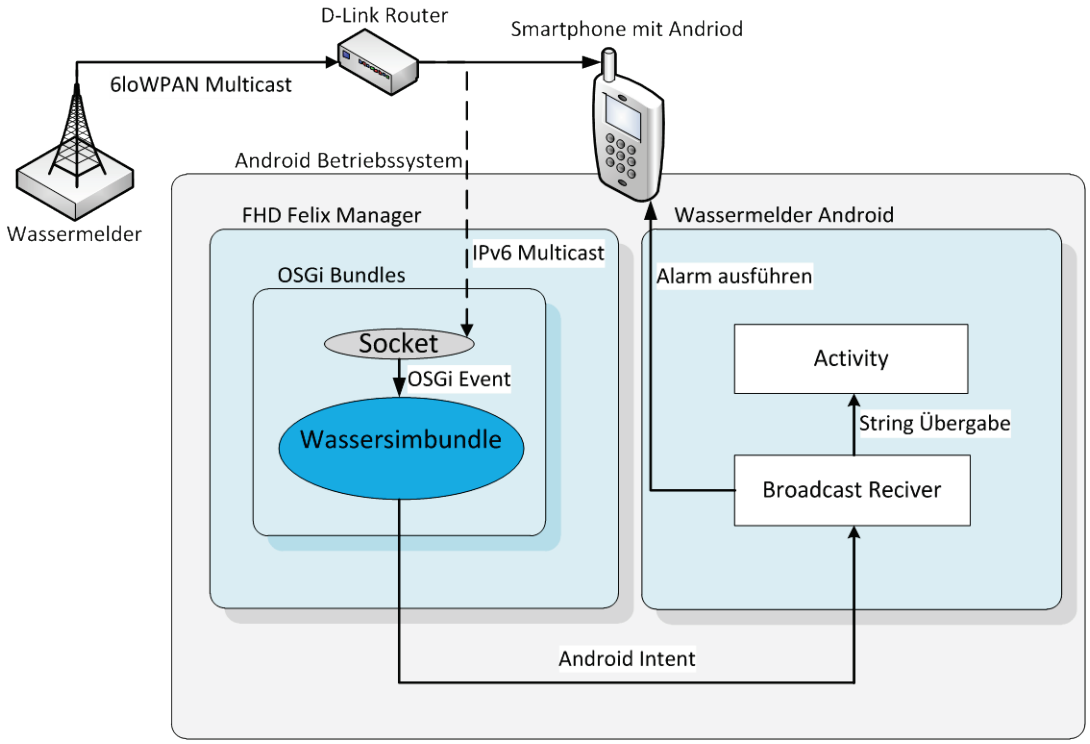


Abbildung 10: Verlauf der Alarmdaten

6.3 Die prototypische Umsetzung

6.3.1 FHD Felix Manager

Die Android Anwendung FHD Felix Manager stellt die benötigte Umgebung für das Apache Felix OSGi Framework bereit. Die grundlegende Konfiguration des Frameworks wird in ihr vorgenommen.

Mit dem Felix Manager ist es möglich das Framework zu starten und zu stoppen. Durch einen geeigneten Broadcast Receiver wird das Framework automatisch nach dem Hochfahren des Smartphones gestartet.

Die Anwendung gliedert sich in drei Java Klassen, die in der folgenden Abbildung aufgeführt sind.

Abbildung 11: FHD Felix Manager Klassendiagramm

6.3.1.1 FelixStart Klasse

Die Benutzeroberfläche des FHD Felix Managers ist sehr einfach gestaltet. Es befinden sich lediglich zwei Buttons in der Acitvity. Diese dienen zum Starten und Stoppen des Felix Services. Beide Buttons sind in der Datei main.xml definiert, diese liegt im Ressourcen Ordner der Anwendung im Unterverzeichnis layout. Listing 3 und Listing 4 zeigen die Implementierung des Stopp Buttons beispielhaft in der main.xml sowie in der FelixStart Activity Klasse innerhalb der `OnCreate` Methode. Der Start Button ist dementsprechend mit den Start Funktionen ähnlich implementiert.

```
<Button
        android:id="@+id/FelixStop"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/FelixStop" />
```

Listing 3: Button Definition in der main.xml

In der Methode `onClick` wird ein Intent, der für das Stoppen von Services bestimmt ist, gesendet. Dieser richtet sich an die FelixService Klasse.

```
FelixStop = (Button)this.findViewById(R.id.FelixStop);
FelixStop.setOnClickListener(new OnClickListener() {
public void onClick(View v) {
       ComponentName comp = new ComponentName(getPackageName(),
                                        FelixService.class.getName());
       stopService(new Intent().setComponent(comp));
       }
});
```

Listing 4: onClick Listener mit Stopp Intent für den Felix Service

In der `onCreate` Methode der Activity, wird ebenfalls ein Intent erzeugt um den Service zu starten, damit der Service auch ohne explizites Drücken auf Start gestartet wird, wenn die Anwendung aufgerufen wird und sich vorher nicht im Hintergrund befand. Die Methoden `onStart` und `onPause` sind bis auf jeweils eine Debug Konsolenausgabe ohne Funktion.

6.3.1.2 FelixService Klasse

Die Methode `getLocalIPAdress` liefert als Rückgabewert die IP Adresse des zuletzt aufgelistetem Netzwerk Interface des Smartphones. Bei bestehender WLAN Verbindung wird die IP Adresse des WLANs ausgegeben. Ist keine WLAN Verbindung hergestellt, liefert die Methode entweder die IP Adresse der Internetverbindung über das Telefonnetz oder den Wert des LocalLoop Interfaces (127.0.0.1) zurück. Tritt eine Exception auf, wird diese in der Konsole ausgegeben. Um Folgefehler zu vermeiden wird in diesem Fall die im Code eingetragene IP Adresse 127.0.0.1 zurückgegeben. Der dazugehörige Code ist im folgenden Listing aufgeführt.

```
public String getLocalIpAddress() {
       try {
       for (Enumeration<NetworkInterface> en =
              NetworkInterface.getNetworkInterfaces();
                     en.hasMoreElements();) {
              NetworkInterface intf = en.nextElement();
              for (Enumeration<InetAddress> enumIpAddr =
                                                 intf.getInetAddresses();
                     enumIpAddr.hasMoreElements();) {
                     InetAddress inetAddress = enumIpAddr.nextElement();
                     msg("IP: " +inetAddress.getHostAddress().toString());
                     return inetAddress.getHostAddress().toString();
                     }
              }
       } catch (SocketException ex) {
              System.out.println("Fehler bei getIPAddress" + ex);
       }
return "127.0.0.1";
       }
```

Listing 5: FelixService.java getLocalIPAddress Methode

Innerhalb der `onCreate` Methode werden einige Framework bzw. Bundle Einstellungen vorgenommen, eine Felix Instanz erzeugt und diese schließlich innerhalb eines HandlerThreads gestartet. Anschließend wird der Service als Vordergrundservice definiert um das Beenden durch Android bei geringem Speicher weitestgehend auszuschließen.

In Listing 6 ist der erste Teil der `onCreate` Methode aufgeführt. Hier werden die für den Start des Frameworks benötigten Einstellungen in eine Properties Instanz geladen und anschließend, beim Erstellen der Framework Instanz, übergeben. An dieser Stelle wird die zuvor beschriebene `getLocalIpAddress` Methode verwendet um die aktuelle IP Adresse des Smartphones für die OSGi Remote Shell einzustellen.

```
public void onCreate() {
	super.onCreate();

	Properties configProbs = new Properties();
	File chacheDir = this.getFilesDir();
	configProbs.setProperty(Constants.FRAMEWORK_STORAGE,
		chacheDir.getAbsolutePath());
	configProbs.setProperty(Constants.FRAMEWORK_SYSTEMPACKAGES_EXTRA,
		ANDROID_FRAMEWORK_PACKAGES);
	configProbs.setProperty("osgi.shell.telnet.ip",getLocalIpAddress()
);
	felix = new Felix(configProbs);
```

Listing 6: FelixService.java onCreate Methode Teil 1

Der zweite Teil der `onCreate` Methode ist Auszugsweise in Listing 7 aufgeführt. Hier wird zunächst ein HandlerThread erstellt. Der HandlerThread ist eine spezielle Android Form eines Threads, der zusätzlich einen eigenen Looper beinhaltet. Damit ist es möglich, Handler Klassen zu erzeugen. In diesem Fall ist er für die Übergabe des Android Contextes zwischen einem OSGi Bundle und Android zwingend nötig, andernfalls erzeugt ein Bundle, welches versucht auf den Android Context zuzugreifen, eine „java.lang.RuntimeException: Can't create handler inside thread that has not called Looper.prepare()“ Exception. Um dieses ungewollte Ausnahmeverhalten erfolgreich zu umgehen, sind zwei weitere Zeilen Code nötig. Der benötigte Looper, der die Nachrichten nacheinander abarbeitet, muss innerhalb der `run` Methode des Threads vorbereitet werden und am Ende gestartet bzw. in seinen eigene Schleife geschickt werden.

Nach dem Vorbereiten des Loopers wird innerhalb eines Try-Catch Blockes das Felix Framework gestartet. Einige erforderliche Bundles werden installiert und gestartet, dazu zählen:

- shellBundle — Kommandozeilen Shell von Felix
- remoteShellBundle — Ermöglicht den Fernzugriff auf die Felix Shell
- eventAdminBundle — Felix Event Admin Service
- scrBundle — Felix Service Runtime Component (Declarative Services)
- bundleManager — Ermöglicht das automatische Nachinstallieren von Bundles
- socketBundle — Bundle für die Kommunikation zu den Geräten
- wassersimbundle — Wassermelder Geräte Bundle, Kommunikation mit Android

Die Installation und das Starten des shellBundles sind innerhalb des Listing 7 beispielhaft aufgeführt. Die übrigen Anweisungen für die fehlenden Bundles wurden für die Übersichtlichkeit des Listings entfernt. Nach der Installation der Bundles wird ein OSGi Service registriert, der Zugriff auf den Context von Android ermöglicht. Der Looper wird nun außerhalb der beiden Try-Catch Blöcke in seine Schleife geschickt und der Thread gestartet. Anschließend wird der Service mithilfe der `setServiceAsForeground` Methode als Vordergrunddienst definiert.

```
...
new HandlerThread("OSGi"){

	@Override
	public void run() {
		Looper.prepare();

	try {
		felix.start();

		Bundle shellBundle =
				felix.getBundleContext().installBundle
		("file:///mnt/sdcard/bundles/org.apache.felix.shell-
		1.4.2.jar");
...
		try {

	felix.getBundleContext().registerService(Context.class.getName(),
					getApplicationContext(), null);
		} catch (Exception e1) {
			e1.printStackTrace();
		}
...
		shellBundle.start();
...

	catch (BundleException e) {
			e.printStackTrace();
		}
	catch (Throwable t) {
	t.printStackTrace();
	}
	Looper.loop();
	}
	}
.start();
setServiceAsForeground();
}
```

Listing 7: FelixService.java onCreate Methode Teil 2

Bei der Methode `setServiceAsForeground` handelt es sich weitestgehend um die Referenzimplementierung für einen Foreground Service aus dem Android Developer Guide (25). Der Aufruf dieser Methode sorgt dafür, dass der Service den Foreground Status erhält. Es wird zudem eine Notification erzeugt, die den aktiven Service anzeigt.

```
private void setServiceAsForeground() {
      try {

Class[] startForegroundMethodSignature = new Class[]{int.class,
                                                Notification.class};
Method startForegroundMethod = getClass().getMethod("startForeground",
                                          startForegroundMethodSignature);
NotificationManager notificationManager = (NotificationManager)
                    getSystemService(Context.NOTIFICATION_SERVICE);
Notification notification = new Notification(R.drawable.icon,
             getString(R.string.app_name), System.currentTimeMillis());
PendingIntent contentIntent = PendingIntent.
             getActivity(getApplicationContext() , 0, new Intent(), 0);
notification.setLatestEventInfo(getApplicationContext(),
      getString(R.string.app_name),"Felix Service up and running!",
                                                contentIntent);

Object[] startForegroundMethodArgs = new Object[]{Integer.valueOf(1),
                                                notification};
try {
      startForegroundMethod.invoke(this, startForegroundMethodArgs);
                    notificationManager.notify(1, notification);
      } catch (Exception e) {
      }
      } catch (Exception e) {
            setForeground(true);
      }
}
```

Listing 8: FelixService.java setForeground Methode

Die Methode `msg` ist für das Erstellen einer Android Notification zuständig. Sie wird innerhalb des Felix Managers zur Anzeige der zugewiesenen IP Adresse verwendet. Eine beinahe identische Methode wird in der Android Anwendung Wassermelder verwendet. Die Unterschiede werden im Kapitel 6.3.3.2 aufgeführt.

```
public void msg(String ThreadMsg) {

        String ns = Context.NOTIFICATION_SERVICE;
        NotificationManager mNotificationManager = (NotificationManager)
                                                getSystemService(ns);

        int icon = R.drawable.icon;
        CharSequence tickerText = "FHD FM Nachricht...";
        long when = System.currentTimeMillis();

        Notification notification = new Notification(icon,tickerText,
                                                                when);

        Context context = getApplicationContext();
        CharSequence contentTitle = ThreadMsg;
        CharSequence contentText = ThreadMsg;
        Intent notificationIntent = new Intent(this, FelixStart.class);
        PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
                                                notificationIntent, 0);

        notification.setLatestEventInfo(context, contentTitle,
                                        contentText, contentIntent);

        mNotificationManager.notify(2, notification);
}
```

Listing 9: FelixService.java msg Methode

Als erstes wird innerhalb dieser Methode auf den Notification Manager zugegriffen. Einige Werte, wie das anzuzeigende Icon und der eingeblendete Text beim Erscheinen der Benachrichtigung, werden gesetzt. Anschließend wird eine neue Notification mit den Werten erzeugt. Die eigentliche Nachricht wird der Methode in der Variablen ThreadMsg übergeben und als Überschrift sowie als Text für die Benachrichtigung verwendet. Abschließend wird die Notification mithilfe des Notification Managers angezeigt.

6.3.1.3 StartAtBoot Klasse

Die Klasse StartAtBoot ist für den Start des Felix Services nach dem Hochfahren des Smartphones zuständig. In der AndroidManifest.xml ist der Broadcast Receiver der StartAtBoot Klasse eingetragen. Der Intent Filter mit der Action „*android.intent.action.BOOT_COMPLETED*" sorgt dafür, dass der Receiver auf den von Android Intent reagiert, der nach dem Start des Betriebssystem verschickt wird. Im Listing 10 ist der entsprechende Auszug aus der AndroidManifest.xml dargestellt.

```
<receiver android:name="de.fhd.inflab.android.felixmanager.StartAtBoot">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
        <category android:name="android.intent.category.HOME" />
    </intent-filter>
</receiver>
```

Listing 10: Broadcast Receiver mit Intent Filter in der AndroidMaifest.xml

Erhält der Broadcast Receiver den BOOT_COMPLETETD Intent, wird dessen `onReceive` Methode ausgeführt. Diese startet zunächst einen neuen Thread. Dieser sorgt für eine Startverzögerung des Services. Der Intent wird zwar erst nach booten von Android verschickt, in der Regel starten auf diesem Intent noch weitere Programme. Vor allem ist diese Verzögerung wichtig damit das Smartphone mehr Zeit hat sich mit dem genutzten WLAN zu verbinden. Dadurch, dass die bereits beschriebene Remoteshell eine IP Adresse benötigt, ist es für den ordentlichen Betrieb zwingend notwendig, dass zum Zeitpunkt des Start des Services die korrekte IP Addresse des Smartphones im WLAN vorliegt. Der Thread wird mit der Thread `run` Methode gestartet, da sie im Gegensatz zur Thread `start` Methode erst die folgenden Zeilen bearbeitet, wenn die `run` Methode des Threads vollständig abgearbeitet wurde. Mit der Thread `start` Methode wird der zu startende Thread parallel zu dem folgenden Code abgearbeitet. Nach der Wartezeit von zweimal fünf Sekunden wird der Service mithilfe eines Intents gestartet. Um Android Service Klassen zu starten, bietet Android die spezielle `startService` Methode an.

```
public class StartAtBoot extends BroadcastReceiver{

	private Thread thread;
	@Override
	public void onReceive(Context context, Intent intent) {

			thread=  new Thread(){
			@Override
			public void run(){
				try {
					synchronized(this){
						wait(5000);
						wait(5000);
					}
				}
				catch(InterruptedException ex){
				}
			}
		};

		thread.run();
		Intent serviceIntent = new Intent();
	serviceIntent.setAction("FelixService");
	context.startService(serviceIntent);
	}
}
```

Listing 11: StartAtBoot Klasse

6.3.2 Wassersimbundle

Das OSGi Bundle Wassersimbundle stellt zwei Funktionen bereit. Es kann die per OSGi Event zu ihm kommenden Wassermelder Daten als Android Intent weitersenden. Außerdem kann es Android Intents empfangen und prinzipiell auf diese reagieren.

Das OSGi Bundle besteht aus drei Klassen welche in Abbildung 12 zu sehen sind.

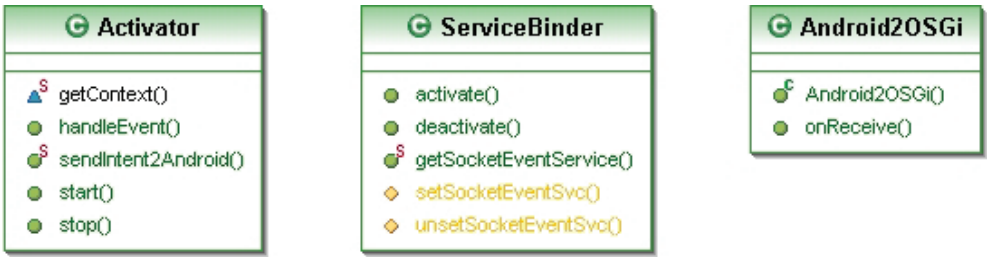


Abbildung 12: Klassendiagramm des OSGi Wassersimbundle

6.3.2.1 Android2OSGi Klasse

Die Klasse Android2OSGi implementiert den Broadcast Receiver. Schickt eine Anwendung innerhalb von Android einen Intent, der für diesen Receiver bestimmt ist, wird dessen `onReceive` Methode ausgeführt. Diese dient hier lediglich zur Demonstration der Bi-Direktionalen Kommunikation mithilfe von Intents. Bei Aufruf gibt sie in der Konsole die Nachricht „OSGi Bundle Intent empfangen!“ aus. Die Registrierung des Receivers sowie die Filterung der Intents wird in der Activator Klasse durchgeführt die in 6.3.2.2 beschrieben wird.

```
public class Android2OSGi extends BroadcastReceiver {

	public Android2OSGi(Context androidContext) {
		super();
		}

	@Override
	public void onReceive(Context context, Intent intent) {
		System.out.println("OSGi Bundle Intent empfangen!");
		}
}
```

Listing 12: Android2OSGi Klasse

6.3.2.2 Activator Klasse

In der Activator Klasse wird zunächst eine Instanz der zuvor beschriebenen Android2OSGi Klasse erzeugt. In der `start` Methode wird anschließend der Android Context über den bereits in 6.3.1.2 beschriebenen Service referenziert. Es wird ein neuer IntentFilter erstellt, der auf die Action „WasserUpdate“ reagiert, dieser wird dann mithilfe des Android Contextes am Betriebssystem registriert. Die letzte Anweisung in der `start` Methode ruft die Methode `sendIntetn2Android` auf, diese wird im Folgenden beschrieben. Übergeben werden hier einmalig die Startwerte 0;0;0.

```
public Android2OSGi Android2OSGi = new Android2OSGi(androidContext);
...
public void start(BundleContext aBundleContext) throws Exception {

       serviceRef =
              BundleContext.getServiceReference(Context.class.getName());
       androidContext = (Context) aBundleContext.getService(serviceRef);

       IntentFilter osgiWasserFilter = new IntentFilter();
       osgiWasserFilter.addAction("WasserUpdate");
       androidContext.registerReceiver(Android2OSGi, osgiWasserFilter);

       sendIntent2Android("0","0","0");
       }
```

Listing 13: Activator.java start Methode

Die Methode `sendIntent2Android` erwartet 3 Strings als Übergabe Werte. Diese sind auf die Daten des Wassermelders angepasst. Innerhalb der Methode wird ein Intent erzeugt, dass mithilfe der `setAction` Funktion, die Action „WassermelderBroadcast“ setzt. Anschließend werden die drei Daten des Wassermelders, deviceID, address und data per `putExtra` als Datenanhang zum Intent hinzugefügt. Der Intent wird nun gesendet, zur besseren Verfolgbarkeit wurde noch eine Konsolenausgabe mit den gesendeten Werten hinzugefügt.

```
public static void sendIntent2Android(String deviceID, String address,
                                      String data){

       Intent intent = new Intent();
       intent.setAction("WassermelderBroadcast");
       intent.putExtra("deviceID", deviceID);
       intent.putExtra("address", address);
       intent.putExtra("data", data);

       androidContext.sendBroadcast(intent);
       System.out.println("Intent gesendet mit: "+ deviceID + " " +
                           address + " "    + data);
       }
```

Listing 14: Activator.java sendIntent2Android Methode

Diese Methode wird von dem OSGi Event Handler aufgerufen um die Sensordaten, die bis dahin nur als OSGi Event innerhalb des Frameworks benutzt werden konnten, in das

Android System zu senden. Der Event Handler ist dabei wie die sendIntent2Android sehr einfach aufgebaut. Er nimmt die im OSGi Event enthaltenen Eigenschaften auf und übergibt diese an die `sendIntent2Android` Methode. Auffällig ist hier die Ähnlichkeit von OSGi Events, die mit Eigenschaften die per Strings zugewiesen werden und den Android Intents deren Extras ebenfalls per Strings benannt werden. Die `handleEvent` Methode ist in Listing 15 aufgeführt.

```
public void handleEvent(Event event){

	System.out.println("Wassermelder OSGi Event empfangen");

	String deviceID = (String) event.getProperty("id");
	String address = (String) event.getProperty("address");
	String data = (String) event.getProperty("data");

	sendIntent2Android(deviceID,address,data);
	}
```

Listing 15: Activator.java handleEvent Methode

Auf die Klasse ServiceBinder die für die Kommunikation zwischen dem Wassersimbundle und dem Com Socket Bundle wird an dieser Stelle nicht weiter eingegangen, da dieser Teil bereits in der Bachelorarbeit von Thomas Schmitz (1) beschrieben ist.

Listing 16 zeigt die Manifest.MF Datei des Bundles. In ihr werden einige Bundle Informationen wie Name und Version definiert. Auch die zu importierenden Bundles, bzw. die Android Klassen, die über das Framework bereitgestellt werden, sind hier aufgelistet.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Wassersimbundle
Bundle-SymbolicName: de.fhd.inflab.android.wassersimbundle
Bundle-Version: 1.0.0
Bundle-Activator: de.fhd.inflab.android.wassersimbundle.Activator
Import-Package: android.app,
 android.content,
 android.net,
 android.os,
 android.widget,
 de.fhd.inflab.android.com.socket.intf,
 org.osgi.framework;version="1.3.0",
 org.osgi.service.component,
 org.osgi.service.event
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Service-Component: OSGI-INF/component.xml,
 OSGI-INF/componentIntf.xml
DeviceID: wm
```

Listing 16: Manifest.MF Datei des Wassersimbundle

6.3.3 Wassermelder Android Anwendung

Die Android Anwendung ist für die Ausgabe der Wassermelder Daten zuständig. Die Daten werden über die zuvor beschriebenen Intents empfangen, ausgewertet und dargestellt. Empfängt die Anwendung einen Alarm des Wassermelders, werden folgende Benachrichtigungen für den Nutzer ausgeführt:

- Die Activity der Anwendung wird mit dem Alarm Status aufgerufen
- Audio Datei wird abgespielt
- Smartphone vibriert mehrmals
- Eine Android Notification wird angezeigt
- Ist das Telefon inaktiv, blinkt die Status LED bis die Notification gelöscht wurde

Ist die Activity der Anwendung aktiv, wird diese mithilfe eines Handlers alle fünf Sekunden aktualisiert. Der genaue Aufbau der Anwendung wird im folgenden Beschreiben.

Die Wassermelder Anwendung besteht aus zwei Klassen, der WassersimappActivity und der WasserReceiver Klasse. Die Activity ist für die Darstellung der Sensordaten verantwortlich. Die WasserReceiver Klasse nimmt die Daten entgegen und wertet diese aus. Bei einem Alarm werden in ihr die zuvor aufgeführten Aktionen zur Benachrichtigung ausgeführt. In Abbildung 13 ist das UML Klassendiagramm der Android Wassermelder Anwendung zu sehen.

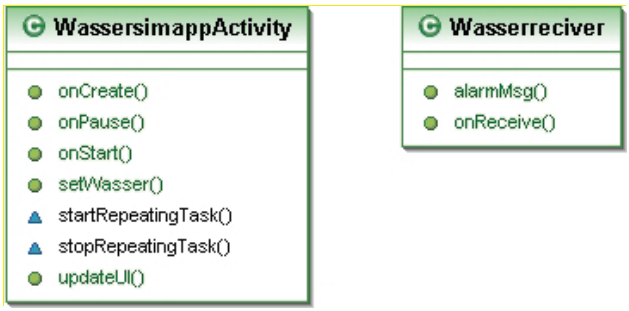


Abbildung 13: Klassendiagramm der Wassermelder Anwendung

6.3.3.1 WassersimappActivity Klasse

Die Methode `updateUI` aktualisiert die grafische Oberfläche der Activity. Zunächst wird die TextView und die ImageView mithilfe der `findViewById` Methode und den in der main.xml definierten Namen den Variablen tv bzw. status zugewiesen. Anschließend werden in die TextView die aktuellen Werte des Wassermelders übergeben, sofern diese durch die WasserReceiver Klasse gesetzt wurden. Danach wird in die ImageView standardmäßig die gelbe Grafik geladen. Ist der Status des Wassermelders „alive“ oder „alarm“ wird das grüne bzw. das rote Statusbild geladen.

```
public void updateUI(){

       tv = (TextView) findViewById(R.id.wasserstandTxt);
       status = (ImageView) findViewById(R.id.status);
       tv.setText("Info"+" "+deciveIDA+" "+addressA+" "+dataA);
      status.setImageResource(R.drawable.status_gelb);
       if (dataA.equals("alive"))
              status.setImageResource(R.drawable.status_gruen);
      if (dataA.equals("alarm"))
              status.setImageResource(R.drawable.status_rot);
   }
```

Listing 17: UpdateUI Methode

Die `updateUI` Methode wird beim Start der Activity einmalig ausgeführt und anschließend mithilfe eines Handlers alle fünf Sekunden ausgeführt, solange bis die Activity verlassen wird.

Das automatische Aktualisieren von Oberflächen kann durch verschiedene Ansätze realisiert werden. Der oft verwendete Ansatz einen neuen Thread zu erstellen und diesen in der Wartezeit schlafen zu lassen, hätte in diesem Fall zwei Schwierigkeiten zur Folge. Einerseits ist es nicht ohne weiteres möglich aus einem Thread heraus die Activity Komponenten zu ändern, anderseits ist auch ein Thread, der ständig zwischen Schlaf und Aktivität wechselt, eine gewisse Belastung für die Hardware. Mit einem zuvor instanziierten Android Handler, der unter dem Namen m_handler bekannt ist, ist es möglich, eine sehr Ressourcen sparende zeitgesteuerte Schleife zu erzeugen. Der Hauptteil dieser Lösung ist in Listing 18 aufgeführt.

```
Runnable m_statusChecker = new Runnable(){
    @Override
    public void run() {
              updateUI();
             m_handler.postDelayed(m_statusChecker, m_interval);
    }
    };
```

Listing 18: Zeitgesteuerter Aufruf der UpdateUI Methode

Es wird eine neue Runnable Instanz erzeugt, in der die `run` Methode die `updateUI` Methode ausführt und anschließend mithilfe des Handlers und der `postDelayed` Anweisung einen Eintrag in der Android Nachrichten Warteschlange erzeugt. Dieser ruft die angegebene Runnable Instanz nach einer bestimmten Zeit erneut auf. Die Runnable Instanz läuft anschließend in dem Thread an dem der Handler gebunden ist. In diesem Fall also dem Activity Thread, damit ist das Ausführen von `updateUI` kein Problem. Durch den Aufruf des `postDelayed` innerhalb der `run` Methode entsteht zunächst eine Endlosschleife. Es ist darauf zu achten das diese gestoppt wird, sobald die Activity nicht mehr aktiv ist. Das stoppen erfolgt mithilfe des Befehls „`m_handler.removeCallbacks (m_statusChecker)`“ damit werden alle Einträge der Runnable m_statusChecker aus der Nachrichten Warteschlange entfernt.

6.3.3.2 WasserReceiver Klasse

Bei der Klasse WasserReceiver handelt es sich um einen Broadcastreceiver dessen `onReceive` Methode die Daten aus dem OSGi Bundle empfängt, an die Activity Klasse weiterleitet und bei Alarm verschiedene Meldungen an den Benutzer gibt.

Im ersten Teil der `onReceive` Methode werden die angehängten Sensordaten aus dem Intent in jeweils eine String Variable geschrieben. Anschließend werden diese, durch eine `set` Methode der Activity, übergeben.

Mithilfe einer einfachen if-Abfrage wird geprüft, ob im Feld Data des Wassermelders ein Alarm gesetzt ist. Ist dies der Fall, werden verschiedene Alarmierungen ausgelöst. Die Vibration des Smartphones wird mit den folgenden Zeilen erreicht.

```
Vibrator v =
(Vibrator)context.getSystemService(context.VIBRATOR_SERVICE);
long vibrateCode[] = {200, 200, 200, 200, 200, 200, 200};
v.vibrate(vibrateCode, -1);
```

Listing 19: Haptische Meldung erzeugen

Das Smartphone vibriert mithilfe des System Services Vibrator nach dem in dem long Array definierten Muster. In diesem Fall vibriert es 200ms, anschließend erfolgt eine Pause von 200ms, insgesamt vibriert es vier Mal.

Die Audio-Benachrichtigung erfolgt durch das Abspielen einer Audiodatei. Diese OGG Datei liegt im Ressourcen Verzeichnis der Android Anwendung im Unterordner /raw der für alle Datentypen geeignet ist. Das Abspielen erfolgt durch den Android eigenen Media Player. Zur Sicherstellung, dass die aktuelle Lautstärke nicht zu leise ist, wird mithilfe des AudioManager zuerst geprüft wie hoch die maximale Lautstärke des Gerätes ist. Diese kann von Smartphone zu Smartphone und jeweils auch nach der Android Version variieren. Anschließend wird geprüft, ob die maximale Lautstärke größer ist als die aktuelle. Ist dies der Fall, wird die aktuelle auf den Wert der maximalen Lautstärke gesetzt und die Audiodatei abgespielt. Nachdem die Audiodatei abgespielt wurde, wird der OnCompletionListner ausgelöst, der im Quelltext zugewiesen wurde. In seiner Methode `onCompletion` wird zunächst die Instanz des Mediaplayers freigegeben und die Audiolautstärke auf den vorherigen aktuellen Wert gesetzt. Der für das Abspielen der Audiodateien verantwortliche Code ist in Listing 20 aufgeführt.

```
final AudioManager audioManager = (AudioManager)

       context.getSystemService(context.AUDIO_SERVICE);
int MusicMaxVol =
       audioManager.getStreamMaxVolume(audioManager.STREAM_MUSIC);
final int MusicVol =
        audioManager.getStreamVolume(audioManager.STREAM_MUSIC);
if(MusicMaxVol > MusicVol){
       audioManager.setStreamVolume(audioManager.STREAM_MUSIC,
                           MusicMaxVol, audioManager.STREAM_MUSIC);
       }
MediaPlayer mp = MediaPlayer.create(context, R.raw.alarm);
mp.start();
mp.setOnCompletionListener(new OnCompletionListener() {

       @Override
       public void onCompletion(MediaPlayer mp) {
       mp.release();
       audioManager.setStreamVolume(audioManager.STREAM_MUSIC, MusicVol,
                                       audioManager.STREAM_MUSIC);
       }
       });
```

Listing 20: Abspielen einer Audio Datei

Die Notification, die bei einem Alarm ausgelöst wird, unterscheidet sich von ihrer Implementierung kaum von der bereits vorgestellten Notification in 6.3.1.2. Zusätzlich zu den bereits beschriebenen Optionen wird zur besseren Sichtbarkeit für den Nutzer die Status LED des Smartphone zum Blinken gebracht. Dies geschieht durch vier Zeilen Code welche in Listing 21 aufgeführt ist. Die Hardwareausführung dieser Status LED kann sehr verschieden sein, manche Smartphones verfügen über keine, manche Status LEDs können in verschiedenen Farben leuchten. Dadurch ist eine exakte Aussage über die Auswirkungen der einzelnen Optionswerte nicht möglich. Die Hardware versucht die Angaben so gut wie möglich wiederzugeben. Der Farbwert der LED wird gesetzt, die Flag für die Notification die Status LED zu verwenden wird aktiviert und die Zeit für LED an und LED aus wird in Millisekunden übergeben. Dies ist in Listing 21 zu sehen.

```
notification.ledARGB = 0x00000001;
notification.flags = Notification.FLAG_SHOW_LIGHTS;
notification.ledOnMS = 2000;
notification.ledOffMS = 1000;
```

Listing 21: Parameter für die Status LED

Das Aufrufen der Activity erfolgt durch einen Intent. Jede Android Anwendung definiert eine Activity die beim Start des Programms aufgerufen wird in der Manifest.xml. Der Aufruf kann einfach durch die Zeilen, die in Listing 22 aufgeführt sind, aufgerufen werden.

```
PackageManager pm = context.getPackageManager();
Intent appStartIntent =
pm.getLaunchIntentForPackage("de.fhd.inflab.android.wassersimapp");
     if (null != appStartIntent){
     context.startActivity(appStartIntent);
     }
```

Listing 22: Intent zum Aufrufen der WassermelderActivity

Der Packet Manager liefert auf Anfrage den passenden Start Intent für die Wassermelder Anwendung. Um eine Null Pointer Exception zu vermeiden wird der Intent nur gesendet falls dieser ungleich Null ist.

Damit das Smartphone vibrieren darf, ist in der AndroidManifest.xml Datei die entsprechende Permisson freigegeben. In Listing 23 ist zudem auch der Broadcast Receiver definiert. Dieser reagiert auf Intents mit dem Action String „*WassermelderBroadcast*“.

```
...
<uses-permission android:name="android.permission.VIBRATE"/>
...
<receiver android:name=".WasserReceiver">
      <intent-filter>
            <action android:name="WassermelderBroadcast" />
            <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
</receiver>
...
```

Listing 23: Auszug der AndroidManifest.xml der Wassermelder Anwendung

6.4 Fazit

In diesem Kapitel wurde im Detail die Implementierung der einzelnen Programmteile beschrieben. Der beschriebene Quellcode kann als Grundlage für weitere Geräte, die auf einem Smartphone verwaltet werden sollen, genutzt werden. Für weitere Alarmsensoren wäre es ausreichend die Namen der Anwendungen und die Bezeichnungen der Intents anzupassen. Geräte, die vom Smartphone aus, gesteuert werden sollen, können ebenfalls realisiert werden.

7 Evaluation

In diesem Kapitel wird die Funktionsweise der entwickelten Software vorgestellt. Anschließend wird eine Bewertung anhand der bereits aufgestellten Anforderungen vorgenommen.

Die Tests wurden auf einem LG P990 Optimus Speed durchgeführt. Dessen genauen Android Versions Spezifikationen wie folgt lauten:

Android Version:	2.3.7
Baseband-Version:	1035.21_20110725
Kernel-Version:	2.6.32.55-canogenmod
Mod-Version:	CyanogenMod-7-20120313-NIGHTLY-p990

Kurz getestet wurden die Anwendungen auch auf einem Samsung I9001 mit den Android Spezifikationen:

Android Version:	2.3.3
Baseband-Version:	I9001XXKI
Kernel-Version:	2.6.35.7-perf

Dabei wurden keine Unterschiede im Verhalten der Software festgestellt.

7.1 Funktionsweise des Prototypen

7.1.1 FHD Felix Manager

Der Felix Manager startet nach dem Booten des Telefons mit 10 Sekunden Verzögerung automatisch. Nach dem Start wird in der Benachrichtigungszeile die zugewiesene IP Adresse angezeigt. Unter dem Punkt „Aktuell“ erscheint die Meldung „Felix Service up and running“. Diese Benachrichtigung kann nicht wie die Benachrichtigung der IP Adresse vom Benutzer entfernt werden. Sie signalisiert, dass der Foreground Service aktiv ist. Stoppt man den Felix Service durch den Felix Manager manuell, wird automatisch auch die „Felix Service up and running“ Benachrichtigung entfernt. Beide Benachrichtigungen sind in Abbildung 14 zu sehen.

Abbildung 14: Android Benachrichtigungen nach dem Start des Frameworks

Abbildung 15 zeigt die Oberfläche des Felix Managers. Sie ist schlicht gehalten und zeigt lediglich zwei Buttons, die bei Betätigung den Framework Service starten bzw. stoppen.

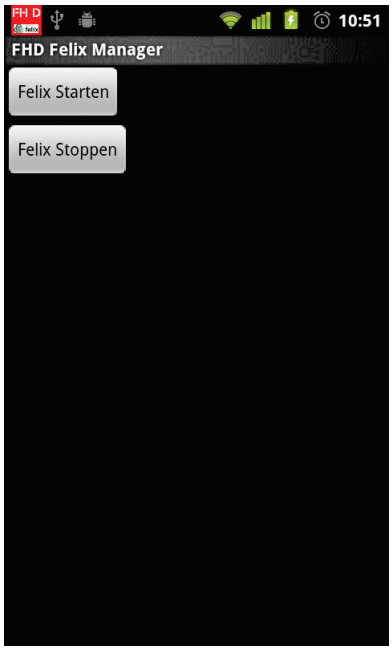
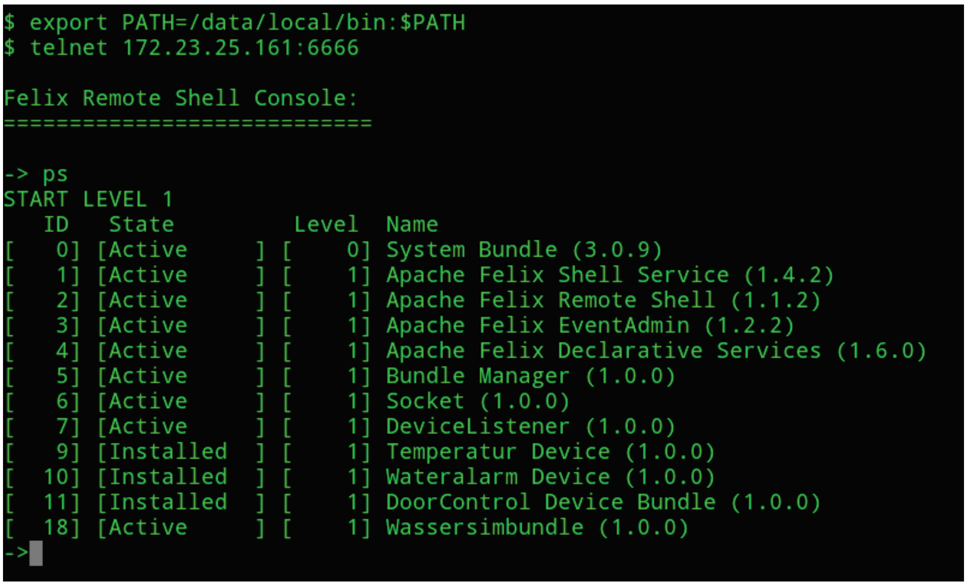


Abbildung 15: FHD Felix Manager

Durch einen Telnet Clients kann auf die Remote Shell des Frameworks zugegriffen werden. Dies kann beispielsweise vom PC mithilfe des Tools „Putty“ (26) erfolgen, oder direkt vom Smartphone aus unter Zuhilfenahme einer geeigneten Anwendung. Android beinhaltet

bereits einen Telnet Client, dieser ist aber nicht direkt durch das Smartphone startbar. Durch eine geeignete Anwendung, welche die Android Konsole öffnet, ist es möglich mit dem Befehl „telnet IPADRESSE:6666“ eine Verbindung mit der Remote Shell aufzubauen. Anschließend kann man beispielsweise mit dem Befehl „ps“ die installierten Bundles und ihren jeweiligen Status anzeigen lassen. In Abbildung 16 ist der Aufruf der Remoteshell direkt vom Smartphone aus mithilfe der Android Anwendung „Terminal Emulator“ (27) und dem anschließenden Aufruf von „ps“ zu sehen.



```
$ export PATH=/data/local/bin:$PATH
$ telnet 172.23.25.161:6666

Felix Remote Shell Console:
============================

-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (3.0.9)
[   1] [Active     ] [    1] Apache Felix Shell Service (1.4.2)
[   2] [Active     ] [    1] Apache Felix Remote Shell (1.1.2)
[   3] [Active     ] [    1] Apache Felix EventAdmin (1.2.2)
[   4] [Active     ] [    1] Apache Felix Declarative Services (1.6.0)
[   5] [Active     ] [    1] Bundle Manager (1.0.0)
[   6] [Active     ] [    1] Socket (1.0.0)
[   7] [Active     ] [    1] DeviceListener (1.0.0)
[   9] [Installed  ] [    1] Temperatur Device (1.0.0)
[  10] [Installed  ] [    1] Wateralarm Device (1.0.0)
[  11] [Installed  ] [    1] DoorControl Device Bundle (1.0.0)
[  18] [Active     ] [    1] Wassersimbundle (1.0.0)
->
```

Abbildung 16: Telnet Konsolenausgabe der Remoteshell

Der Ressourcenbedarf des Felix Managers ist verhältnismäßig gering. Der Service belegt mit dem in Abbildung 16 installierten bzw. gestarteten Bundles 8,5 MB des RAMs. Die Anwendung belegt dabei 2,37 MB auf dem internen Speicher. Die gesamt Performance des Smartphones hat sich durch den aktiven Service subjektiv nicht verändert. Um diesen rein subjektiven Eindruck zu belegen, wurden jeweils zwei Durchläufe mit dem Benchmark Tool AnTuTu Benchmark (28) in der Version 2.7.3 durchgeführt. Das Programm testet dabei die verschiedenen Komponenten wie RAM, CPU und die Grafikleistung.

Die Gesamtpunktzahl belief sich bei den Durchläufen auf 5618 und 5615 mit gestartetem Service, sowie 5641 und 5661 Punkte ohne den Service. In Abbildung 17 ist die Auswertung jeweils von einem Durchlauf abgebildet. Dabei bestätigt der Benchmark den subjektiven Eindruck, dass es keine nennenswerten Performance-Einbußen durch den Betrieb des Services gibt. Die geringeren Punktzahlen bei gestartetem Service, liegen mit weniger als 1% Unterschied zu den Ergebnissen ohne Service, innerhalb der Messtoleranz.

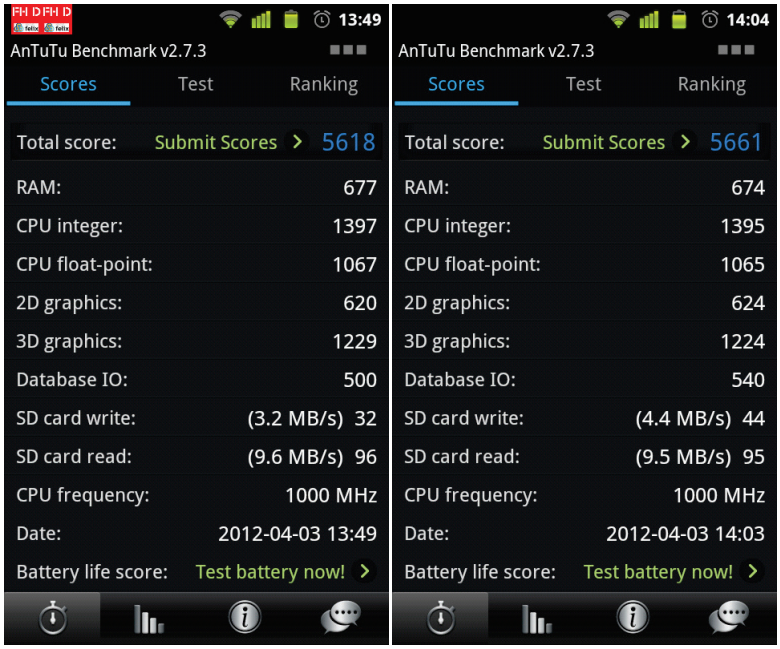


Abbildung 17: Benchmark Ergebnisse, mit gestartetem Service (l) und ohne (r)

7.1.2 Wassermelder Anwendung

Die Wassermelder Anwendung kann unabhängig vom Felix Manager gestartet werden. Nach dem Start erscheint die Activity, auf der zwei Textfelder und der Status des Wassermelders als Bild angezeigt werden. Nach dem ersten Start, bedingt durch die `onCreate` Methode, werden zunächst die Info Default Werte „0 0 0“ und der gelbe Status angezeigt. Ist das Framework mit dem Wassersimbundle nun aktiv und der Wassermelder sendet ein „alive“ ändert sich der Status entsprechend. Im Infofeld steht nun das Kürzel des Devices, hier „wm“, die IP Adresse des Senders, hier „172.23.25.160“ und der vom Gerät gemeldete Status „alive“. Der Status wurde hier mithilfe eines UDP Senders von einem PC aus gesendet. Der echte Wassermelder würde anstelle der hier angezeigten IPv4 Adresse seine IPv6 Adresse ausgeben. Das Verhalten nach dem Start, sowie der nach dem Empfangen eines „alive“ sind in Abbildung 18 zusammengefasst.

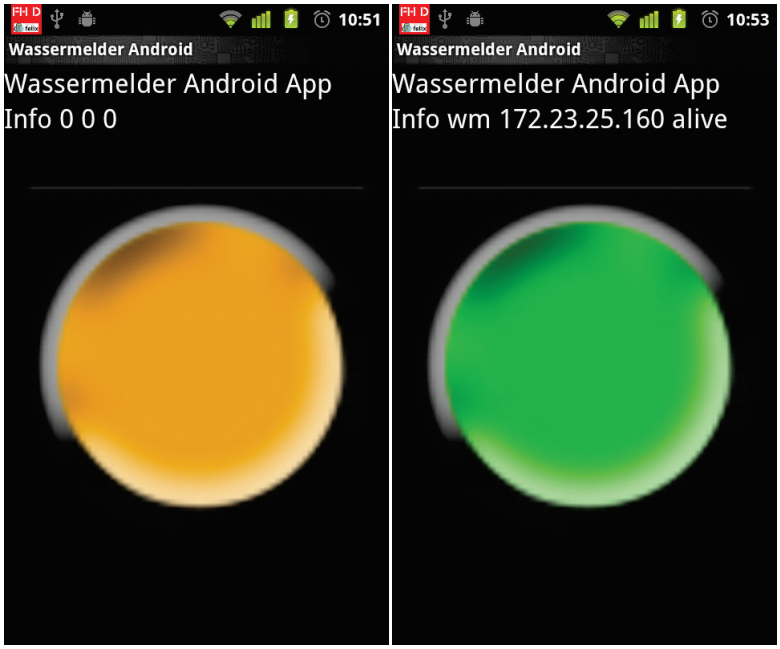


Abbildung 18: Wassermelder nach dem Start (l), nach dem empfangen eines "alives" (r)

Sendet der Wassermelder einen Alarm, wird die Activity der Anwendung automatisch in den Vordergrund gerufen. Ein Alarmton wird durch den Mediaplayer abgespielt. Hierfür wird zunächst die Medienlautstärke auf das Maximum gestellt. Das Smartphone vibriert zudem. Des Weiteren wird eine Benachrichtigung erzeugt, die dafür sorgt, dass die Status LED des Telefons blinkt, falls das Telefon inaktiv ist (Display aus). Die Activity kurz nach dem Alarm ist in Abbildung 19 (links) zu sehen. Dort ist zudem die automatische Lautstärkenanpassung zu sehen, welche die Lautstärke am Anfang des Alarms auf den Maximalwert setzt. Auch die Benachrichtigung die zunächst die ganze Statuszeile mit der Nachricht „Alarm ausgelöst: Wassermelder!“ einnimmt, ist in dieser Abbildung dargestellt.

Abbildung 19: Wassermelder bei Alarm

Dreht der Anwender das Smartphone, wird das Layout, wie in Abbildung 20, entsprechend angepasst. Oben links ist in dieser Abbildung auch das Benachrichtigungssymbol des Alarms zu sehen.

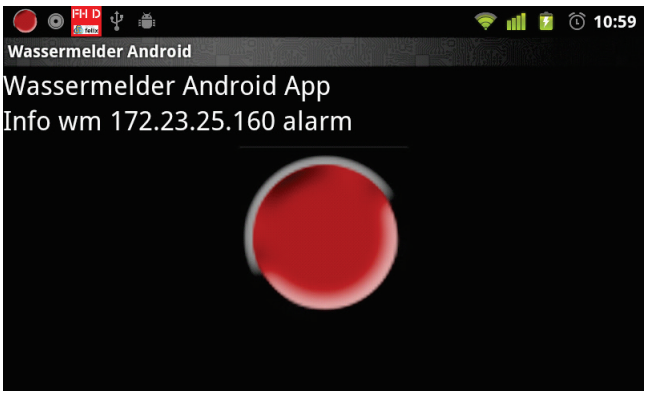


Abbildung 20: Wassermelder im Querformat

7.2 Erfüllung der Anforderungen

Für die Auswertung der Anforderungen werden zur Übersicht die aus Abschnitt 3.5 stammenden Pflicht- und Wunschkriterien nochmals aufgeführt. Die Kriterien sind jeweils mit einem Haken (Kriterium ist erfüllt) oder mit einem Kreis (Kriterium ist nicht erfüllt) gekennzeichnet.

- ✓ P_1 Bereitstellung eines OSGi Framework
- ✓ P_2 Durchgängige Verfügbarkeit des Frameworks
- ✓ P_3 Senden von Benachrichtigungen an den Benutzer aus OSGi Bundles.
- ✓ P_4 Automatischer Start des Frameworks
- ✓ P_5 Manuelles starten und stoppen des Frameworks
- ✓ P_6 Kennzeichnung wenn das Framework gestartet ist
- ✓ P_7 Bidirektionale Kommunikation zwischen OSGi Bundles und Android Anwendungen
- ○ W_1 Installation und Deinstallation von Bundles über das Benutzerinterface
- ✓ W_2 Geringer Ressourcenbedarf
- ○ W_3 Log Informationen

7.3 Probleme und Verbesserungsmöglichkeiten

Die Grundfunktionalität der Software ist sichergestellt, es treten aber gelegentlich Verzögerungen bei der Alarmierung auf.

Während des Standby Betriebs eines Smartphones, schläft nach einer bestimmten Zeit der Prozessor. Während diesen Schlafphasen konnten die Alarm Intents nicht verarbeitet werden. Auffällig ist, dass die Nachrichten des Sensors ohne Einschränkungen empfangen werden konnten, auch die Verarbeitung innerhalb des Frameworks und die daraus resultierenden Intents wurden gesendet. Der Aufruf des Broadcast Receivers erfolgte jedoch während einer Schlafphase nicht. Dies führt dazu, dass ein Alarm gelegentlich innerhalb von Android „stecken bleibt“. Der Alarm verzögerte sich dadurch meist um wenige Minuten, die längste Wartezeit während der Testphase belief sich auf 17 Minuten. Aktiviert der Nutzer das Smarphone, oder es wacht durch einen anderes Ereignis wie einen ankommenden Anruf von selbst auf, beginnt der zuvor zwischengespeicherte Alarm sofort.

Dieses Problem könnte durch verschiedene Ansätze gelöst werden. Mithilfe des Android Power Managers ist es möglich, den Prozessor daran zu hindern, in die Schlafphase zu gehen. Dadurch wird aber die Batterie des Smartphones deutlich stärker belastet. Eine weitere Möglichkeit wäre es, die Alarmmeldung in das OSGi Bundle zu verlagern, da dieses scheinbar nicht von der Schlafphase beeinflusst wird.

7.3 Fazit

Die entwickelten Komponenten funktionieren weitestgehend sehr gut. Für das einzige auftretende Problem, der Verzögerung der Alarmierung wurden zwei mögliche Lösungen angesprochen. Der Benutzer wird durch die Benachrichtigung „Felix Service up and running“ deutlich davon in Kenntnis gesetzt, das der Service gestartet ist. Durch die Start und Stopp Buttons innerhalb des Felix Managers ist es ihm möglich den Service nach Bedarf manuell zu starten und zu stoppen. Die Alarmierung durch das Abspielen eines Audio Alarms, der in den Vordergrund tretenden Activity, das Vibrieren, sowie das Blinken der Status LED ist auffällig, sodass der Benutzer diese bemerken sollte. Die gute Performance sorgt dafür, dass der Benutzer durch den Hintergrundservice nicht eingeschränkt ist.

8 Zusammenfassung

8.1 Überblick über alle Kapitel

Der Einsatz von OSGi unter Android zeichnet sich durch die deutlich erhöhte Dynamik bei der Verwendung von Programmen aus. Durch die Standardisierung zwischen den verschiedenen OSGi Frameworks ist es prinzipiell möglich, alle bestehenden Bundles mit entsprechender Anpassung der GUI, auf einem Android Smartphone laufen zu lassen.

Durch den FHD Felix Manager ist der Betrieb des OSGi Frameworks Apache Felix auf einem Android Smartphone sichergestellt. Der Einsatz eines Services ermöglicht das Ausführen des Frameworks im Hintergrund, ohne das es spürbare Performance Verluste gibt.

Die Wassermelder Anwendung verdeutlichte die Möglichkeiten der Kommunikation zwischen OSGi Bundles und Android Anwendungen mithilfe von Intents. Die Art der Implementierung kann für weitere Geräte wie Temperatursensor, CO_2 Sensor oder Notfallknöpfe übernommen werden.

8.2 Ausblick

Für die Zukunft ist es denkbar, ein Bundle mit einer standardisierten Alarmschnittstelle zu nutzen. Dies hätte den Vorteil, dass auch Alarme von neuen Geräten dem Nutzer ohne weitere Android Anwendung mitgeteilt werden können.

Der FHD Felix Manager könnte um die noch offenen Wunschkriterien, Installation und Deinstallation von Bundles sowie der Möglichkeit zur Anzeige von Loginformationen erweitert werden.

Bedingt durch das Sicherheitskonzept von Android ist es nicht möglich Android Anwendungen automatisch ohne Bestätigung des Nutzers zu installieren. Für die Installation von neuen Geräten, die über das Smartphone verwaltet werden, könnte ein weiteres Bundle entwickelt werden. Ähnlich wie bei dem System von Thomas Schmitz, bei dem die passenden Bundles von einem Server angefordert werden, könnten die Android Anwendungen auch auf das Telefon gelangen. Die Installation kann anschließend aus dem Bundle per Intent gestartet werden. Der Benutzer muss allerdings die Installation bestätigen.

Die Entwicklung einer universellen Android Anwendung, die eine Vielzahl von Geräten unterstützt, ist auch denkbar. Dies erfordert aber eine genaue Planung und schränkt gegebenenfalls die Darstellung oder Nutzung bestimmter Geräte ein.

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

1. **Schmitz, Thomas.** *Entwicklung einer OSGi-Service-Komponente zum dynamischen Laden von Benutzeroberflächen für Android im Umfeld von Ambient Assisted Living.* Düsseldorf : s.n., 2011.

2. **Open Handset Alliance.** Open Handset Alliance. [Online] Februar 2012. http://www.openhandsetalliance.com/index.html.

3. **Brintrup, Saskia.** Onlinekosten.de. *450.000 Apps im Android Market.* [Online] http://www.onlinekosten.de/news/artikel/47119/0/450-000-Apps-im-Android-Market.

4. **Becker, Anno und Pant, Markus.** *Android Grundlagen der Programierung.* Heidelberg : dpunkt.verlag GmbH, 2009. 978-3-89864-574-4.

5. **Apache Software Foundation.** Apache Harmony - Open Source Java Platform. [Online] November 2011. http://harmony.apache.org/.

6. **Huawei Device Co., Ltd.** HUAWEI IDEOS X3. [Online] April 2012. http://www.huaweidevice.com/worldwide/productFeatures.do?pinfoId=2995&treeId=3745&directoryId=6001&tab=0.

7. **LG Electronics Deutschland GmbH.** LG P990 OPTIMUS SPEED Smartphone. [Online] April 2012. http://www.lg.com/de/mobiltelefone/alle-lg-mobiltelefone/LG-P990-OPTIMUS-SPEED.jsp?s_kwcid=TC|8755|LG%20%2Boptimus%20%2Bspeed||S||7958957330.

8. **ASUSTeK Computer Inc.** Eee Pad | Transformer Prime | Spec. [Online] April 2012. http://eee.asus.com/en/eeepad/transformer-prime/specification/.

9. *Androiden-Vielfalt.* **Müller, Andrea.** c't 2011, Heft 4, s.l. : Heise, 2011.

10. **OSGi Alliance.** OSGi Alliance | Main / OSGi Alliance. [Online] April 2012. http://www.osgi.org/Main/HomePage.

11. **The OSGi Alliance.** *OSGi Service Platform Service Compendium Release 4, Version 4.2.* 2009.

12. **Oracle Corporation.** Foundation Profile. [Online] August 2009. http://java.sun.com/products/foundation.

13. **The OSGi Alliance.** *OSGi Service Platform Core Specification, Release 4, Version 4.3.* 2011.

14. **Schatten, Alexander, et al., et al.** *Best Practice Software-Engineering.* s.l. : Spektrum Akademischer Verlag Heidelberg, 2010. 78-3-8274-2486-0.

15. **Fachhochschule Düsseldorf.** Fachhochschule Düsseldorf. [Online] April 2012. http://www.fh-duesseldorf.de/a_fh.

16. **The Eclipse Software Foundation.** Equinox. [Online] April 2012. http://eclipse.org/equinox/.

17. **Hargrave, BJ und Bartlett, Neil.** Android and OSGi: Can they Work Together? [Online] März 2008. http://www.eclipsecon.org/2008/sub/attachments/Android_and_OSGi_Can_they_work_together.pdf.

18. **ProSyst Software GmbH.** Products List | ProSyst - OSGi services and embedded Java within home, vehicle and mobile devices. [Online] Juli 2010. http://prosyst.com/index.php/de/html/content/132/Products-List/.

19. **ProSyst Software GmhH.** Mobile | Products | mBS Mobile for Android | ProSyst. *OSGi services and embedded Java within home, vehicle and mobile devices.* [Online] Februar 2010. http://prosyst.com/index.php/de/html/content/49/mBS-Mobile-for-Android/.

20. **Apache Software Foundation.** Apache Felix - Apache Felix Framework and Google Android. [Online] Mai 2011. http://felix.apache.org/site/apache-felix-framework-and-google-android.html.

21. **IST-MUSIC Consortium.** IST-MUSIC. [Online] Oktober 2010. [Zitat vom:] http://ist-music.berlios.de/site/index.html.

22. **BerliOS Developer.** BerliOS Developer: Project Summary - IST-MUSIC. [Online] April 2012. http://developer.berlios.de/projects/ist-music/.

23. **OSAMI-Commons.** osami-fin-demo - osami-fin-demo - Google Project Hosting. [Online] April 2012. http://code.google.com/p/osami-fin-demo/.

24. **OSAmI-Commons.** OSAmI, Open Source Ambient Intelligence. [Online] Oktober 2011. http://www.osami-commons.org/.

25. **Google Inc.** Android Developer Guide. [Online] 2012. http://developer.android.com/guide/index.html.

26. **Thatham, Simon.** Download PuTTY - a free SSH and telnet client for Windows. [Online] Januar 2008. http://www.putty.org/.

27. **Palevich, Jack.** Android Terminal Emulator - Android Apps auf Google Play. [Online] April 2012. https://play.google.com/store/apps/details?id=jackpal.androidterm&hl=de.

28. **AnTuTu.** AnTuTu Benchmark - Android Apps auf Google Play. [Online] April 2012. https://play.google.com/store/apps/details?id=com.antutu.ABenchMark&hl=de.

29. **Wütherich, Gerd, et al., et al.** *Die OSGi Service Platform.* Heidelberg : dpunkt.verlag, 2009. 978-3-89864-457.

30. *Smart Wars.* **Müller, Florian.** c't, 2010, Bd. 24/10.

31. *Innenansichten.* **Becker, Arno.** c't 2011, Heft 4, s.l. : Heise, 2011.

32. **Cüpper, Alexander.** Einführung in das mobile Betriebssystem Android. [Online] Dezember 2010. https://www.matse.rz.rwth-aachen.de/dienste/public/show_document.php?id=7143.