Diplomarbeit

Konzeption und Umsetzung eines mobilen, kartenbasierten Informationssystems

Matthias Neubert

Oktober 2009

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Rechnernetze

Betreuender Hochschullehrer: Prof. Dr. rer. nat. habil. Dr. h.c. Alexander Schill

Betreuender Mitarbeiter: Dipl.-Medieninf. Christian Liebing

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 30. Oktober 2009

Matthias Neubert

Danksagung

Bei Herrn Prof. Dr. rer. nat. habil. Dr. h.c. Alexander Schill bedanke ich mich für die Vergabe und Betreuung der Diplomarbeit.

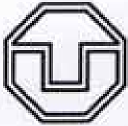
Besonderen Dank schulde ich Herrn Dipl.-Medieninf. Christian Liebing, der mich durch seine engagierte Betreuung und stete Diskussionsbereitschaft mit vielseitigen Denkanstößen bereicherte und bei der Erstellung dieser Arbeit unterstützt hat.

Für die Unterstützung bei der Lösung konkreter Probleme der Zusammenarbeit von Android bzw. der DalvikVM mit Apache Felix und auch die unermüdliche Fortentwicklung der Felix OSGi-Framework-Implementierung und von Apache Felix iPOJO möchte ich an dieser Stelle Karl Pauls, Richard S. Hall und Clement Escoffier meinen Dank aussprechen.

Des weiteren danke ich Peter Kriens für die Unterstützung bei der Anpassung des iPOJO Eclipse PlugIns.

Ich danke meiner Verlobten Elisabeth Quittenbaum und meiner Schwester Anja Röpke für ihre große Geduld und ihren Beistand beim Entstehen dieser Arbeit.

Nicht zuletzt möchte ich meinen Eltern danken, die mir durch ihre fortwährende Unterstützung das Studium und diese Arbeit ermöglichten und sie mit Anteilnahme verfolgt haben.



TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Systemarchitektur, Lehrstuhl Rechnernetze

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name, Vorname: Neubert, Matthias
Studiengang: Informatik
Matr.-Nr.: 3056055
Thema: Konzeption und Umsetzung eines mobilen, kartenbasierten Informationssystems

ZIELSTELLUNG

In den letzten Jahren haben Geo-Mashups sehr stark an Bedeutung gewonnen, wodurch eine Vielzahl personalisierter Informationssysteme auf Basis von webbasierten Kartendiensten (Google Maps, OpenStreetMap, etc.) entstanden sind. Beispielsweise lassen sich beliebige *Points-of-Interest* (POI) in das Kartenmaterial integrieren.
Im Rahmen der Arbeit soll untersucht werden, ob Ansätze für die flexible Integration und Austauschbarkeit von verschiedenen ortsbezogenen, webbasierten Informationsdiensten in diese Karten existieren und wie deren Visualisierung und Interaktion realisiert werden kann. Das Ziel dieser Arbeit ist der Entwurf eines Konzeptes, mit dem beliebige Informationsdienste über eine generische Schnittstelle aufgerufen und in eine webbasierte Karte integriert werden können. Des Weiteren ist die Unabhängigkeit von einem konkreten Client wünschenswert, so dass analysiert werden soll, ob vorhandene Informationsdienste dynamisch eingebunden werden können. Schließlich ist das erarbeitete Konzept mit Hilfe einer prototypischen Implementierung basierend auf Google Android anhand eines praxisnahen Szenarios im Bereich des ÖPNV zu validieren.

SCHWERPUNKTE

- State-of-the-Art-Analyse von Ansätzen zur flexiblen Integration und Austauschbarkeit von Informationsdiensten in webbasierte Karten
- Anforderungsanalyse und Konzeption einer Lösung, die einen einheitlichen Aufruf der Informationsdienste ermöglicht und somit deren Austauschbarkeit garantiert
- Evaluierung mit Hilfe einer prototypischen Implementierung ausgewählter Aspekte im Bereich des ÖPNV (z.B. Darstellung der Haltestellen, Abfahrtsmonitor einer Haltestelle, etc.)

Betreuer:	Dipl. Medieninf. Christian Liebing
Verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Institut:	Institut für Systemarchitektur
Beginn am:	01.03.2009
Einzureichen am:	31.08.2009

Unterschrift des verantwortlichen Hochschullehrers

1 Inhalt

1 Einführung

1.1 Motivation 1
1.2 Überblick über die Arbeit 2

2 Grundlagen

2.1 Mobile Plattformen 5
2.2 Begriffe der Geoinformatik 14
2.3 Fazit 21

3 Anforderungsanalyse

3.1 Beschreibung des Beispielszenarios 23
3.2 Verallgemeinerung des Beispielszenarios 24
3.3 Funktionale Anforderungen 25
3.4 Nicht-funktionale Anforderungen 28
3.5 Funktionen des Beispielszenarios 30
3.6 Fazit 31

4 State-of-the-art-Analyse

4.1 MapProvider 33
4.2 Darstellung von Geodaten in WebGIS-Karten 42
4.3 Vorhandene Anwendungen 51
4.4 Dynamische Softwarekomponenten 57
4.5 Fazit 81

5 Konzeption

5.1 Grundkonzepte 86
5.2 Architekturkonzepte 89
5.3 Detailiertes Konzept 97
5.4 Erfüllung der Anforderungen an das Konzept 103
5.5 Fazit 104

6 Implementierung

6.1 Einschränkungen und Besonderheiten 105
6.2 Aufbau der Anwendung 108
6.3 Beispiele der prototypischen Umsetzung 116
6.4 Weitere Ergebnisse der prototypischen Umsetzung 124
6.5 Evaluation der Implementierung 128
6.6 Fazit 130

7 Evaluation

7.1 Erfüllung der Anforderungen 131
7.2 Funktionsweise des Prototypen 137
7.3 Fazit 140

8 Zusammenfassung

8.1 Überblick über alle Kapitel 145
8.2 Ausblick 147

Quellenverzeichnis 149

Anhang

Anhang A1 .... A-3

Anhang A2 .... A-11

Anhang A3 - Listings A-15

Anhang A4 - Klassendiagramme A-19

Abbildungsverzeichnis

Abbildung 4.1: Schematische Darstellung der Android Google Maps AP 39
Abbildung 4.2: Schematische Darstellung der Android OpenStreetMap API .. 40
Abbildung 4.3: Aufbau GeoMashUp-Seite auf Basis von Google Maps 43
Abbildung 4.4: Mapplet - Architektur 47
Abbildung 4.5: Aufbau GeoMashUp-Seite auf OSM-Basis 49
Abbildung 4.6: Struktur der OSGi Spezifikation 63
Abbildung 4.7: Schichten des OSGi Framework 64
Abbildung 4.8: Lebenzyklus der OSGi Bundles 68
Abbildung 4.9: schematischer Ablauf OSGi Service Registry 69
Abbildung 4.10: OSGi Filter Syntax 71
Abbildung 4.11: Einsatz des Service Tracker 73
Abbildung 4.12: Lebenszyklus einer Service Component 75
Abbildung 4.13: Zusammenhänge bei Declarative Services 77
Abbildung 5.1: Mapplet-Webanwendung 90
Abbildung 5.2: Webanwendung auf OSGi Basis 91
Abbildung 5.3: OSGi-basierter Adapterserver 92
Abbildung 5.4: D-OSGi-basierter Ansatz 93
Abbildung 5.5: OSGi-basierte Android-Anwendung 95
Abbildung 5.6: OSGi-basierte Android-Anwendung (Detailiertes Konzept) 98

Abbildung 6.1: Schematische Darstellung des Gesamtsystems 108

Abbildung 6.2: Schematische Darstellung der Host-Anwendung 110

Abbildung 6.3: Schematische Darstellung der MapServices 113

Abbildung 6.4: Klassendiagramm MapProvider-Implementierung für Google Maps A-19

Abbildung 6.5: Klassendiagramm MapServiceLayer1-Implementation - Qype.com 119

Abbildung 6.6: Klassendiagramm MapServiceLayer3-Implementierung VVO-Abfahrtsmonitor 123

Abbildung 7.1: Auswahl der OSGi-Android-Anwendung in der AppSelectionActivity 137

Abbildung 7.2: Installation eines Bundles im Bundle Management 137

Abbildung 7.3: Deinstallation eines Bundles im Bundle Management 138

Abbildung 7.4: MapService-Viewer Startbildschirm 138

Abbildung 7.5: Darstellung der MapService-MapItems in der Karte 139

Abbildung 7.6: Der VVO Abfahrtsmonitor 139

Abbildung 7.7: Die Qype Informationsanzeige 139

Abbildung 7.8: Die Qype Informationen als Toast (ohne installierten MSL3) ... 139

Abbildung 7.9: Separate Android-Anwendungen pro MapService 140

Abbildung 7.10: MapService-Viewer: Generische Plattform für MapServices . 141

Abbildung A1.1: Webservice - Schematischer Aufbau A-7

Abbildung A2.1 Klassendiagramm Google Maps Bibliothek für Android A-12

Tabellenverzeichnis

Tabelle 2.1 Übersicht über mobile Plattformen 13

Tabelle 5.1 Zusätzliche Anforderungen an das Konzept 103

Listing-Verzeichnis

Listing 4.1: Beispiel einer Bundle Manifest-Datei 66

Listing 4.2: Möglichkeiten der Versionsangabe in der Manifest-Datei 66

Listing 4.3: Auszug aus dem BundleActivator des anbietenden Bundles 70

Listing 4.4: Auszug aus dem BundleActivator des nutzenden Bundles 71

Listing 4.5: service-component.xml von Bundle org.mnsoft.mybundl mit Event-Strategie 76

Listing 4.6: iPOJO Service Verwendung mit metadata.xml-Beschreibung ... 78

Listing 4.7: Beispiel für eine iPOJO metadata.xml - Datei 79

Listing 4.8: iPOJO Service Verwendung mit iPOJO-Annotations A-15

Listing 6.1: Bnd - Datei des Qype MSL1 121

Listing 6.2: metadata.xml des Qype MSL1 122

Listing 6.3: Bnd-Datei erweitert für BIndex repository.xml-Generierung A-16

Listing 6.4: Beispiel für die repository.xml - Datei A-16

Listing 6.5: Shell-Befehle für das Signieren und Optimieren 127

1 Einführung

1.1 Motivation

Zum aktuellen Zeitpunkt findet eine Revolution des Internets hin zum mobilen Internet statt. Angetrieben von erfolgreichen Mobilgeräte-Plattformen wie dem Apple iPhone oder Google Android wird die mobile Nutzung des Internets für den Endnutzer interessanter. Auch die Netzanbieter begünstigen diese Entwicklung durch ihren Netzausbau und attraktivere Preismodelle für Breitbandnutzung. Der mobile Kontext verstärkt auch die Nachfrage nach Location Based Services (LBS), bei der die Nutzung von Webdiensten im Vordergrund steht, welche in Abhängigkeit zum aktuellen Ort des Nutzers einen konkreten Mehrwert bereitstellen. Damit der Nutzer diese Informationen leichter auf die reale Welt beziehen kann, ist die Darstellung in einer digitalen Karte ein nützliches Hilfsmittel. Die Verfügbarkeit von GPS auf vielen dieser mobilen Endgeräte ermöglicht darüber hinaus die einfache Positionierung des Nutzers auf dieser Karte.

Mit der Einführung von Google Maps hat die Nutzung der digitalen Online-Kartendienste einen Boom erlebt, nicht zuletzt aufgrund der ausgereiften Geocoding-Funktion, die einer eingegebenen Adresse ein Koordinatenpaar zuordnet und in der Karte darstellt. Durch Google Maps wurde dem Nutzer über eine zentrale Stelle digitales Kartenmaterial kostenlos und frei verfügbar gemacht, was in diesem Umfang, dieser Qualität und Zugänglichkeit bis dahin unbekannt war. Zahlreiche andere Anbieter sind dem Trend bereits gefolgt und stellen eigene Online-Kartendienste bereit.

Aufgrund der Verfügbarkeit dieser Online-Kartendienste entstand bei Nutzern und Dienstanbietern der Wunsch, ortsbezogene Webservices in diesem freien Kartenmaterial darzustellen. Für dieses Anliegen existieren mannigfaltige Ansätze, die unter anderem in dieser Arbeit vorgestellt werden sollen.

Ein Beispiel hierfür ist der Wunsch der Verkehrsverbund Oberelbe GmbH (VVO) ihre Webservice-Informationsdienste georeferenziert zu visualisieren und auch mobil nutzbar zu machen. Aus dem Wunsch der VVO wurde ein Beispielszenario für diese Arbeit generiert, welches die Darstellung der VVO-Haltestellen in einer digitalen Karte und die Verknüpfung dieser mit einem Abfahrtsmonitor auf einem mobilen Endgerät beschreibt.

Neben der üblichen Darstellung in einem Webbrowser ermöglichen einige in dieser Arbeit betrachteten Hersteller mobiler Plattformen und Dritthersteller durch Bibliotheken eine browserunabhängige Kartendarstellung. Daraufhin entstand eine Vielzahl separater Anwendungen für die jeweiligen verschiedenen Plattformen, die jeweils einen separaten Webservice benutzen um die über ihn erhaltenen Informationen in der Karte anzuzeigen.

Obwohl sich die meisten dieser Anwendungen stark ähneln und sie viele identische Programmteile haben, existiert bisher kein einheitliches Konzept zur Integration ortsbezogener Webservices in digitales Online-Kartenmaterial in einer Art, wie sie für mobile Endgeräte geeignet ist. Ebenso existiert kein Ansatz diese verschiedenen Webservices zusammenzuführen und gleichzeitig in einer Karte darzustellen und so von Synergieeffekten zu profitieren. In diesem Kontext wäre auch ein Ansatz zur dynamischen Integration neuer Webservices sinnvoll.

Die wissenschaftliche Herrausforderung dieser Arbeit besteht daher in der Konzeption und Entwicklung eines Ansatzes, der diese Mängel behebt, in dem er dynamisch, flexibel und generisch ist, mehrere beliebige Dienste vereinen kann, Unabhängigkeit von konkreten Online-Kartendiensten ermöglicht und diese Eigenschaften für mobile Endgeräte bereitstellen kann. Das Hauptaugenmerk dieser Aufgabe liegt daher vorwiegend in dem Dynamikaspekt der flexiblen Integration beliebiger ortsbezogener Webservices in beliebiges Online-Kartenmaterial und dessen Harmonisierung bezüglich der Gegebenheiten der mobilen Plattformen.

1.2 Überblick über die Arbeit

Diese Arbeit ist in drei große Teile untergliedert. Der einführende Teil beginnt mit den Grundlagen in Kapitel 2, in welchem fachfremde Grundbegriffe geklärt werden, aber auch die verfügbaren mobilen Plattformen untersucht werden. In Kapitel 2.2 wird auch die Entscheidung für eine Plattform getroffen, eine notwendige Einschränkung, welche die darauf folgenden Kapitel entsprechend beeinflusst.

In der Anforderungsanalyse (Kap. 3) wird das in der Motivation erwähnte Beispielszenario ausgebaut und verallgemeinert. Darauf basierend werden dann funktionale und nicht-funktionale Anforderungen indentifiziert. Die Ergebnisse der darauf folgenden Kapitel werden dann stets an diesen Anforderungen bewertet.

Der einführende Teil endet mit Kapitel 4, einer ausführlichen State-of-the-art-Analyse, die den aktuellen Stand der Technik bezüglich der verschiedenen Aspekte der Aufgabenstellung untersucht.

Dabei werden Kartenanbieter verglichen und unter anderem anhand ihrer Möglichkeiten zur Integration von Informationen in die Karte bewertet. Abschließend werden softwaretechnologische Möglichkeiten zur Realisierung eines dynamischen Ansatzes untersucht.

Alle untersuchten Technologien werden auch bezüglich ihrer Eignung für den Einsatz auf der mobilen Plattform beleuchtet, die im Kapitel 2 ausgewählt wurde.

Das in diesem Kapitel erarbeitete Wissen wird zum Verständnis von Konzeption und Implementierung benötigt und in den Kapiteln 5 und 6 als bekannt vorrausgesetzt. Daher wurden Informationen, die üblicherweise in diesen Kapiteln anzusiedeln wären, in eine umfangreiche und tiefgreifende State-of-the-Art-Analyse vorgelagert. Somit soll eine thematisch strukturierte Gliederung dieser Arbeit erreicht werden.

Den Hauptteil der Arbeit bilden die Kapitel 5 (Konzeption) und 6 (Implementierung). In der Konzeption werden auf Basis des im einführenden Teil erarbeiteten Wissens verschiedene Lösungsansätze diskutiert. Abschließend wird ein Ansatz ausgewählt und detailliert. Dieses Detailkonzept soll prototypisch umgesetzt werden, was im Kapitel 6 ausführlich beschrieben wird.

Der Schlußteil der Arbeit beginnt mit Kapitel 7, der Evaluation von Konzept und Implementierung. Dort wird betrachtet, inwieweit der Ansatz die an ihn gestellten Anforderungen der Anforderungsanalyse erfüllt. Abschließend wird betrachtet was mit der prototypischen Umsetzung erreicht werden konnte und an welchen Stellen noch weiter entwickelt werden könnte.

Im Anschluss an die Evaluation werden im Kapitel 8 die Erkenntnisse und Ergebnisse der einzelnen Kapitel und der Arbeit insgesamt zusammengefasst. Im Ausblick werden dann Punkte benannt, an denen die Arbeit fortgesetzt werden könnte.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe eingeführt und erklärt, die im weiteren Verlauf der Arbeit verwendet werden. Die einführenden Kapitel „Webanwendung“ und „Webservice“ sind im Anhang A1 nachzulesen. Das dort vermittelte Grundwissen soll es auch dem themenfernen Leser ermöglichen, diese Arbeit zu verstehen. Der fachnahe Leser kann diese Kapitel überspringen.

Das Kapitel 2.1 soll einen Überblick über die vorhandenen, mobilen Zielplattformen geben und am Ende vorab eine Auswahl treffen. Im Kapitel 2.2 werden fachfremde Termini der Geoinformatik eingeführt, die zum Verständnis der Thematik und der Arbeit selbst von Nutzen sind.

2.1 Mobile Plattformen

Es gibt auf dem Markt zahlreiche, unterschiedliche mobile Plattformen, die in diesem Kapitel untersucht und verglichen werden sollen. Im Rahmen dieser Arbeit kann nicht auf alle eingegangen werden. Die vorhandene Auswahl ist nur ein Auszug der möglichen Plattformen, welche anhand der folgenden Kriterien vorgenommen wurde:

1. Zugang zum Internet über beliebige Kommunikationstechnologien

2. Relevante Marktverbreitung

3. Entwicklung von eigener Software ist für die Plattform möglich, SDKs sind mit vertretbarem Aufwand zugänglich

4. Unterstützung geeigneter, aktueller Hardware (z.B. Farbdisplay mit Touch-Screen-Funktionalität)

Punkt 4 ist vor allem in Bezug auf das Paradigma der direkten Karten-Interaktion relevant. Während die Darstellung auch auf normalen Bildschirmen möglich ist,

wäre die direkte Interaktion mit Objekten auf der Karte ohne TouchScreen-Funktion mühsam.

Die genannten Kriterien beschränken die Auswahl auf die Plattformen Google Android, Apple iPhone, Microsoft Windows Mobile und Nokia Symbian OS. Trotz aktuell 15,9% Marktanteil wird die Blackberry Plattform von Research in Motion (RIM) nicht betrachtet, da die Entwicklung eigener Software für das Blackberry nur mit erheblichen Umständen möglich ist.(Punkt 3) Als Business-Gerät setzt RIM eher auf hermetische Sicherheit denn auf Erweiterbarkeit.

Die ausgewählten Plattformen werden in den folgenden Abschnitten vorgestellt und auf Eignung für diese Arbeit untersucht. Am Ende dieses Kapitels wird anhand einer Reihe von Auswahlkriterien die am besten geeignete Plattform bestimmt. In den weiteren Kapiteln wird dann nur noch auf Konzepte und Technologien Bezug genommen, die auf dieser Plattform möglich sind. Da hierdurch Einschränkungen vorgenommen werden, muss dieser Vergleich vielfältige, auch nicht-technische Kriterien beleuchten.

Die Einschränkung wird unter anderem vorgenommen, um sich auf eine exemplarische Umsetzung konzentrieren zu können. Durch die hohe Heterogenität der mobilen Plattformen ist es, mit Ausnahme einer Webanwendung (und ihren bereits angeführten Nachteilen im Bereich mobiler Endgeräte), nicht möglich, eine einheitliche Lösung für alle Plattformen zu finden. Allein schon wegen der großen hardwareseitigen und softwaretechnischen Unterschiede, wäre der kleinste gemeinsame Nenner mit so vielen Abstrichen verbunden, dass interessante Features der Endgeräte ungenutzt blieben. Dem Wunsch der Unabhängigkeit von einer bestimmten Plattform kann also aus diesen Gründen heraus nicht im vollen Umfang genüge getan werden. Es wird allerdings im Kapitel 5 darauf eingegangen, ob die vorgesehene Architektur und das detaillierte Konzept so auch auf andere Plattformen übertragbar wäre. So wird dann die Frage nach der Allgemeinheit des zu erforschenden Ansatzes beantwortet.

2.1.1 Google Android

Android ist ein freies, quelloffenes Betriebssystem für mobile Endgeräte wie Smartphones, PDAs oder Netbooks. Seine Entwicklung wird von Google und der Open Handset Alliance vorangetrieben [1]. Zu den Mitgliedern zählen namhafte Mobilfunkanbieter wie T-Mobile und Vodafone, Hardwarehersteller wie Samsung, Sony, HTC, AsusTek, Acer und Halbleiterhersteller wie ARM, Broadcom, Intel, Atheros und nVidia, außerdem noch Softwarefirmen wie Google und Ebay.

Die unterste Android-Schicht basiert auf einem Linux-Kernel der Version 2.6 und wird mit einem, in mehreren darüberliegenen Schichten verteilten, kompletten Software-Stack ausgeliefert, der aus C- und Java-Bibliotheken besteht. Der

Software-Stack stellt die Funktionen der Hardware des Endgerätes einem für Android entwickeltem Java-Programm zur Verfügung.

Die für Android entwickelte registerbasierte virtuelle Maschine DalvikVM führt den mittels des „dx-Tools“ (Teil des Android SDK) „gedexten“ Java-bytecode effizient auf dem Android-Gerät aus. Durch cross compiling wird hierbei aus dem Java-Bytecode eines normalen Desktop-Java-Compilers wie beispielsweise javac von Suns J2SE ein für die DalvikVM optimierter Maschinencode generiert. Dadurch werden die üblichen Bedenken bei Java bezüglich der Performance teilweise ausgeräumt.

Java wird somit für normalerweise leistungsschwache mobile Endgeräte interessant und zwar auch ohne die gewaltigen Einschränkungen, die JavaME mit CDC und CLDC seinerzeit mit sich brachte [2]. Die Entscheidung für Java wird auch eine strategische sein, denn die Sprache ist vergleichsweise schnell zu lernen und weit verbreitet, sodass sich sogar vor dem Erscheinen des ersten Android-Handys bereits eine große Zahl freiwilliger Entwickler gefunden hatte, um erste Anwendungen für Android zu schreiben. Forciert wurde dies ebenfalls durch den hoch dotierten Wettbewerb „Android Developer Challenge I“, der 2008 stattfand [3].

Dadurch, dass Android frei und quelloffen ist, ist es für verschiedenste Hardwarehersteller interessant, die sich so die aufwändige und teure Entwicklung eigener Betriebssysteme sparen können [4, 5]. Dies ermöglicht eine Konzentration auf die Hardware und verursacht niedrigere Produktionskosten, die sich auch im Endverbraucherpreis wiederspiegeln können. Letzteres könnte eine massive Verbreitung von Android-basierten Geräten zur Folge haben, was Android als Zielplattform für Software-Entwicklungen noch interessanter werden lässt. Während Hauptkonkurrent Apple iPhone mit seiner Preisgestaltung das eher gehobene Marktsegment ansprechen will, könnten Android-Geräte schon bald für die breite Masse zur Verfügung stehen [6].

Da Android auf verschiedenster Hardware mit sehr unterschiedlichen Ausstattungsmerkmalen läuft, ist zu erwarten, dass die Endgeräte sehr heterogen ausfallen. Prinzipiell unterstützt Android von Haus aus ein großes Spektrum von Ausstattungsmerkmalen, beispielsweise bezüglich Netzwerkanbindung, Interaktionsmodalitäten, Bildschirmtypen und Sensoren. Alle vom iPhone verwendeten Sensoren (vgl. Kap. 2.1.2) werden auch von Android unterstützt [7].

Das erste Gerät auf Android-Basis war das von T-Mobile vertriebene HTC Magic, auch bekannt als T-Mobile G1. Die user experience beim Surfen im Internet oder auch die Einbindung von Google Maps wurden viel gelobt [8]. Zahlreiche weitere Geräte sind angekündigt. Ebenfalls wollen verschiedene Hersteller von Netbooks Android als Alternative für Windows einsetzen [9].

Die Softwareentwicklung für Android gestaltet sich leicht zugänglich. Das Android SDK [10] ist für Windows, Linux und MacOS X ohne Anmeldung frei zum Download

verfügbar. Darüber hinaus existiert eine große Zahl an Tutorials und eine recht ausführliche Dokumentation [11].

Für die Entwicklungsumgebung Eclipse bietet Google zudem das Plugin „Android Developer Toolkit“ (ADT) an, welches den Entwickler beispielsweise durch den Debugger DDMS, automatisches Erzeugen einer Android-Anwendungsdatei (APK) und durch automatische Installation in den Emulator unterstützt [12]. Auch andere Entwicklungsumgebungen werden zunehmend unterstützt. Diese Eigenschaften führten zu einer großen, sehr aktiven Entwickler-Community im Internet, die ihre teilweise sehr interessanten Ergebnisse unter Open Source-Lizenzen anderen Entwicklern frei zur Verfügung stellen. Dies macht die Plattform für den Entwickler besonders attraktiv.

Das Deployment von Software für die Android Plattform erfolgt für den Endkunden vorwiegend über den Android Market, einem Web-Katalog voller kostenloser und kostenpflichtiger Anwendungen, die Firmen, aber auch viele private Entwickler erstellt haben. Das Konzept ist ein Klon von Apple‘s erfolgreichem App Store, in welchem Software für das iPhone beziehbar ist [13]. Dieses Vorgehen ist für Smartphones gängig und findet sich beispielsweise auch bei Nokia wieder [14].

2.1.2 Apple iPhone OS

Das Apple iPhone hat mit seiner Einführung im Bereich der Smartphones viel bewegt. So wurde die Touchscreen-Interaktion zum Quasi-Standard moderner Smartphones. Apples Gerät hat aber, durch Patente geschützt, als Alleinstellungsmerkmal ein Multi-Touch-Display [15]. Das bedeutet, dass man mit zwei oder mehr Fingern mit dem iPhone interagieren kann. Hierdurch sind zahlreiche auch von den MacBooks bekannte Touchpad-Gesten auf dem iPhone möglich. Diese Technologie ermöglicht die bequeme und vor allem intuitive Bedienung, die sich über das gesamte Gerät erstreckt. Diese Bedienungsfreundlichkeit kombiniert mit der von Apple gewohnten, ausgeklügelten und nutzerfreundlichen Bedienerführung und GUI-Gestaltung haben dem iPhone seinen hohen Beliebtheitsgrad gebracht.

Dies wurde erreicht, obwohl weder die Hardware (ausgenommen die aktuelle Version 3GS) noch die Software in punkto Leistung und Funktionsumfang konkurrenzlos überlegen sind. Ganz im Gegenteil: heute übliche Leistungsmerkmale wie Videoaufnahme und MMS Versand waren lange Zeit nicht vorhanden. Auch bezüglich Rechenleistung oder Displaygröße kann sich das iPhone nicht abheben. Die Konkurrenzdisplays verfügen fast ausnahmslos über die Multi-Touch-Fähigkeit, welche zur Zeit zumindest aber aufgrund des Patentschutzes noch von keinem Gerät genutzt werden darf.

In vielen Quellen wird die intuitive Bedienung und Nutzerfreundlichkeit des iPhones als Grund für den Erfolg sowie Ausgleich für die objektiven Schwächen gewertet [16].

Die Google Maps Applikation des iPhones unterstützt beispielsweise das bequeme Zoomen mit 2 Fingern, wie es auch aus manchen Anwendungen unter MacOS X bekannt ist. Diese „Kleinigkeit" hat deshalb gerade dann so hohe Bedeutung, wenn ein Gerät wie z.B. das iPhone fast auschließlich über den TouchScreen bedient wird. Das Interaktionsparadigma TouchScreen wird im Vergleich zu Konkurrenzprodukten wesentlich stärker ausgeschöpft. Darüber hinaus verfügt das Gerät unter anderem noch über 3D-Bewegungssensoren, Lage-Sensoren für den Wechsel zwischen Hoch- und Querformat, Nähesensor zum Abschalten des Monitors während des Telefonierens, einen Helligkeitssensor, mit dem die Bildschirmhelligkeit der Umgebung angepasst wird, und GPS.

Das Betriebssystem des iPhone ist das von Apple entwickelte iPhone OS. Es ist ein auf die ARM-Architektur portiertes MacOS X. Herauszuheben ist, dass iPhone OS die Sprache Java nicht unterstützt.

Prinzipiell ist es für private Entwickler und Firmen möglich, Software für das iPhone zu entwickeln. Dies ist allerdings mit einigen Hürden verbunden, was schon zu einiger Frustration in der Entwicklergemeinde gesorgt hat [17]. Zunächst muss man sich bei Apple als iPhone Developer anmelden und muss, will man seine Applikation später veröffentlichen, eine jährliche Gebühr entrichten [18]. Nach der Anmeldung kann das iPhone SDK heruntergeladen werden.

Bislang kann man nur unter MacOS X auf einem Intel-basierten Mac mit XCode in der Sprache Objective C entwickeln [19]. Objective C ist eine in der Apple-Welt verbreitete Sprache, die wie C++ versucht Objektorientierung in ANSI C zu bringen. Durch die relative Beschränkung auf die Apple-Welt ist die Sprache jedoch nicht so verbreitet, sodass die Enwicklergemeinde zunächst nicht so groß ist, wie vergleichsweise bei Java und auch die höhere Lernkurve für einen nicht-C-affinen Entwickler ist nicht völlig zu vernachlässigen [20].

Das Deployment findet ausschließlich über Apples App Store statt [13]. Mit dem iPhone wird auf den Onlineshop zugegriffen, das gewünschte Programm ausgewählt, direkt heruntergeladen und installiert. Dass der App Store äußerst erfolgreich ist, verdeutlicht ein Komentar von Apple-Chef Steve Jobs: „So etwas wie den App Store hat es vorher in der Industrie noch nie gegeben - weder in Quantität noch in Qualität, über 1,5 Milliarden heruntergeladene Anwendungen werden es anderen sehr schwer machen aufzuholen." [21].

Um im App Store vertreten zu sein, muss die Anwendung zuvor an Apple geschickt werden. Dort wird sie bezüglich Stabilität getestet und auf Verstoß gegen Apples Regeln untersucht [22].

Eine dieser Regeln ist, das die Software nicht mit eigenen Entwicklungsplänen von Apple konkurrieren darf. Dies ist einem Entwickler eines Spiels zum Verhängnis geworden. Als dieser innerhalb von zwei Monaten 250.000 USD mit seinem

Programm verdient hatte, wurde es von Apple aus dem App Store entfernt [23]. Dies zeigt, dass die Entwicklung von Anwendungen für das iPhone von unvorhersehbaren, nicht-technischen Schwierigkeiten begleitet werden kann.

2.1.3 Microsoft Windows Mobile

Auch Microsoft beteiligt sich am Thema Smartphones. Nachdem im PDA Bereich seit 1996 mit Windows CE 1.0 reichlich Erfahrung gesammelt werden konnte, war der Schritt 2001 zum Smartphone nur logisch [24]. Die aktuelle Version „Windows Mobile 6.1 Smartphone“ ist auf zahlreichen Geräten, z.B. von HTC, Toshiba oder Samsung zu finden. Wie bei Microsoft üblich, sind die Betriebssysteme closed-source und werden über Microsofts EULA (End User License Agreement) lizensiert [25].

Der Nachfolger „Windows Mobile 7.0“, der 2010 erscheinen soll, wird sich vor allem in den Punkten gestenbasierte Steuerung und intuitive Bedienerführung der Konkurrenz wie iPhone und Android anpassen. Insgesamt hinkt Mircosoft damit der aktuellen Entwicklung etwas hinterher. Vor allem die zur Zeit noch eingeschränkte TouchScreen Funktionalität ist verglichen mitdden Möglichkeiten bei iPhone und Android ein echtes Defizit. Die meisten Geräte mit Windows Mobile werden in PDA Manier mit einem Stift (Stylus Pen) bedient, was im Allgemeinen kein großer Nachteil ist, aber zumindest in puncto Intuitivität und Ergonomie schlechter abschneidet. Die Bedienung Stylus-basierter Geräte mit den Fingern ist teilweise möglich, jedoch sind die UI-Elemente von ihren Abmessungen her dafür eher ungeeignet.

Die Software-Entwicklung findet auf Microsoft Windows ab XP und mit „Microsoft Visual Studio“ statt. Es sind die .NET Sprachen C# und Visual Basic (VB.NET) möglich, wobei allerdings nur das .NET Compact Framework, einer auf ca. 1/3 des Umfangs reduzierten Version des .NET Frameworks, verwendet werden kann. Das .NET Compact Framework und das SDK („Windows Mobile Developer Toolkit“) ist ist bei Microsoft frei herunterladbar [26]. Letzteres enthält verschiedene Emulatoren und APIs [27].

Als Java- oder auch C++ - Entwickler lässt sich relativ leicht auf C# umsteigen, sodass hier die Entwicklergemeinde großes Potential hat. Dies ist hier wie auch bei den anderen Plattformen relevant, da heutzutage ein Plattform-Hersteller nur noch seine Plattform und evtl. eine Verbreitungsinfrastruktur bereitstellt. Wie gut diese vom Kunden angenommen wird und wieviel Erfolg und Verbreitung eine Plattform finden wird, hängt mittlerweile vor allem von den zusätzlichen Programmen ab, die für diese Plattform verfügbar sind.

Das Verbreiten der Anwendungen findet ab Windwos Mobile 6.5 im „Windows Marketplace for Mobile“ statt, Microsofts Pendant zum Apple App Store und Google Market. Eine gleichnamige, auf dem Gerät installierte Software ermöglicht auch hier das bequeme Downloaden und Installieren direkt auf dem mobilen Endgerät

[28]. Wie beim iPhone muss die Software an eingeschickt werden und wird von Microsoft getestet und anhand von eigenen Kriterien begutachtet und schlussendlich zertifiziert [29].

Bezüglich der Entwicklerfreundlichkeit positioniert sich Windows Mobile zwischen iPhone und Android. Der Zugang ist so frei und leicht wie beim Android, die Verbreitung ist allerdings teurer, wenn auch für den kommerziellen Anbieter nicht so teuer wie beim iPhone.

2.1.4 Nokia Symbian OS

Der Weltmarktführer bei den Mobiltelefonen ist Nokia.[30] Das von Symbian entwickelte Symbian OS ist das grundlegene Betriebssystem von Nokias Smartphones. Das Betriebssystem wird zur Zeit von Nokia (Eigentümer von Symbian) schrittweise zu einer Open Source-Softwareplattform und steht dann unter Eclipse Public License 1.0 (EPL) [31]. Auf Symbian OS bauen verschiedene Benutzeroberflächen auf, wie zum Beispiel S60, S80 oder Sony-Ericsons UIQ (User Interface Quartz). Vor allem die S60 Plattform fand enorme Verbreitung, nicht nur durch die Nokia-Handys, sondern auch durch Lizenznehmer wie Samsung und LG.

Gleichzeitig mit dem Schritt Richtung Open Source hat Nokia angekündigt, dass Symbian OS und die darauf basierenden Benutzeroberflächen wie Nokias S60, Sony-Ericssons UIQ und NTT DoCoMo‘s MOAP zu einer gemeinsamen Plattform mit einheitlichem User-Interface-Framework zusammengelegt werden [31].

Ein weiterer aus Software-Entwicklersicht interessanter Schritt war Nokias Übernahme von Trolltec mit seinem plattformübergreifendem Anwendungs- und Benutzeroberflächen- C++Framework „Qt“. Qt wird derzeit für die S60 Benutzeroberfläche der Symbian OS Plattform portiert und wird ab S60 Edition 3 mit Feature Pack 1 lauffähig sein und unter LGPL veröffentlicht werden.

Aus Entwicklersicht ist es zur Zeit ungewiss, ob für das aktuelle Symbian OS geschriebene Software auch nach den großen Änderungen noch lauffähig sein werden. Auch „Qt for S60“ hat wahrscheinlich noch einige Änderungen vor sich, um kompatibel zur vereinheitlichten Symbian Plattform zu sein.

Da die Verwendung von Python die Installation von Python auf dem Engerät voraussetzt und die JavaME VM verglichen mit heutigen Möglichkeiten unangenehme Einschränkungen hat, bleibt zur Zeit nur noch die Entwicklung in Symbian C++, was aber selbst für erfahrene C++ Entwickler einige Monate an Einarbeitungszeit bedeuten kann. Dafür stehen dann alle Symbian APIs voll zur Verfügung.

Nokia bietet jeweils für ihre aktuellen Plattformversionen separate SDK-Pakete

zum freien Download an. Eine Anwendung wird immer gegen eine bestimmte Plattformversion geschrieben. Eine Abwärtskompatibilität besteht leider nicht, was Entwicklungsmehraufwand bedeutet, will man alle relevanten im Markt befindlichen Versionen abdecken. Dies ist zur Zeit ein nennenswertes Argument gegen die Symbian Plattform.

Die Verteilung der Anwendungen findet über eine Marktplatz-Webseite statt, die „Nokia Ovi“ heißt [32]. Die Anwendungen müssen hierfür kostenpflichtig getestet, zertifiziert und signiert werden.

Da die Vereinheitlichung der Plattform und die interessante Einbindung von Qt zum aktuellen Zeitpunkt noch in einiger Ferne stehen, ist diese Plattform für diese Diplomarbeit zunächst nicht interessant. Wenn die Änderungen abgeschlossen sind, kann sie zukünftig sehr interessant werden und ein Gegenpol zu iPhone, Android und Windows Mobile sein.

2.1.5 Auswahl der Zielplattform

Tabelle 2.1 gibt eine Übersicht über die Merkmale der vorgestellten Plattformen.

Die Symbian-Plattform wird durch die aktuellen Entwicklungen ab ca. 2011 für Entwickler sehr interessant. Bis dahin schreckt die Heterogenität der auf dem Markt befindlichen Symbian-Versionen etwas ab. Die wichtigste Entwicklungssprache für Symbian OS (Symbian - C++) bietet dem Plattformeinsteiger einen schwierigeren Anfang als vergleichsweise sauber objektorientierte Sprachen wie C# oder Java. Das Zulassungsverfahren für die Verteilung über Nokia Ovi ist sogar restriktiver als bei Apple.

Microsofts Windows Mobile ist zwar bezüglich der Programmiersprachen zugänglicher, hinkt aber zum aktuellen Zeitpunkt der Konkurrenz bezüglich echter TouchScreen-Unterstützung hinterher. Erwartungsgemäß ist die Softwareentwicklung nur unter Windows möglich. Generell scheint Microsoft im Fortschritt etwas behäbig zu sein und reagiert am spätesten auf die Trendwende. Die Stiftinteraktion gibt es zwar schon lange, aber eine wirklich intuitive und angenehme Bedienung ist damit nur eingeschränkt möglich.

Das iPhone hat alle Eigenschaften, die für diese Diplomarbeit nötig sind. Es müsste mit Objective C eine Sprache genutzt werden, die immerhin konsequent objektorientiert ist. Das SDK und damit die Softwareentwicklung ist allerdings nur für Mac OS X verfügbar. Der TouchScreen mit Multi-Touch Unterstützung ist zum Zeitpunkt noch ein Alleinstellungsmerkmal. Gegen das iPhone sprechen die restriktive Behandlung von Entwicklern im App Store und der Preis des Gerätes. Das iPhone hat zwar trotz seines hohen Preises einen guten Absatz gefunden, jedoch ist nicht zu erwarten, dass es in naher Zukunft iPhone-Ableger für niedrigere

Merkmale	Android	iPhone	Win. Mobile	Symbian
Verbreitung [33,34]	gering*	12,9% (mittel)	11,1% (mittel)	49,8% (hoch)
Prognose	seht gut	gut	stagniert	fallend**
Lizenz	Apache2 / GPLv2	GPL,APSL, Apple EULA	MS EULA	bald EPL 1.0
Quelloffen	Ja	Nein	Nein	ab 2011
Verfügbarkeit SDK	frei	nach kostenpfl. Regist.	frei	frei,nach Anmeldung
Programmier-sprachen	Java SE	Objective C	C#, VB.NET	C++, Python, JavaME
Deployment	Market	App Store	Marketplace	Nokia Ovi
Architektur	beliebig	ARM	ARM	ARM
TouchScreen	Ja	Ja	Stylus	Ja (ab S60v5)
Multi-Touch	möglich***	Ja	später	später

Tabelle 2.1 Übersicht über mobile Plattformen

* Verbreitung gering, da erst seit Ende 2008 insg. 2 Endgeräte verfügbar
** 2007 noch 65%, 2008 49,8%, Marktführerschaft bleibt erhalten [35]
*** Technisch/Hardwareseitig möglich, aber Einsatz patentrechtlich geschützt [36]

Quellen für Entwicklungspotential / Prognosen: [37, 38, 35, 38, 39, 40]

Preissegmente geben wird.

Googles Android erfüllt ebenfalls alle Kriterien, da es alles kann, was das iPhone auch kann. Hinzu kommen eine leicht erlernbare Enwicklungssprache (Java), gute Entwicklungsumgebung für unterschiedliche Betriebsysteme (Windows, Linux, MacOS X) und eine wenig restriktive Verteilungsinfrastruktur. Darüber hinaus steht es den Geräteherstellern frei zur Verfügung, was wiederum hilft, Produktionskosten zu sparen und damit auch niedrigere Preissegmente ansprechen zu können [41, 42]. Dies erweckt Hoffnungen auf ein hohes Verbreitungspotential, dank Linux-Kern

nicht nur im Smartphone-Bereich, sondern beispielsweise auch im Netbookbereich [35, 43]. Die Verteilungsplattform Android Market ist verglichen mit den Verteilungsinfrastrukturen der anderen drei Plattformen leicht zugänglich, kostengünstig und wenig restriktiv.
Durch seine vollständige Eignung für das Einsatzszenario, die genannten zusätzlichen Vorzüge und das hohe Potential ist Google Android die interessanteste Plattform in diesem Vergleich und wird daher als Zielplattform für den Prototypen bestimmt. Die in den folgenden Kapiteln vorgenommenen Betrachtungen werden sich daher an ihrer Tauglichkeit für den Einsatz mit Google Android messen lassen müssen.

2.2 Begriffe der Geoinformatik

Da bei dieser Arbeit digitales Kartenmaterial, Objekte der realen Welt mit ihrem Raumbezug, welche in Webservices bereitgestellt werden, häufig verwendete Begriffe sind, wird deutlich, dass der Fachbereich der Geoinformatik bzw. Kartographie involviert ist. Um möglichst konform mit der Begriffswelt dieser Fachbereiche zu gehen, ist es notwendig, die für diese Arbeit relevanten Begriffe ausreichend einzuführen. Es geht jedoch nicht um die einzelnen Begriffe, sondern viel mehr um ein Gesamtverständnis der Zusammenhänge.

2.2.1 Geographische Koordinaten

Geographische Koordinaten dienen der Beschreibung der Position eines Ortes auf der Erdoberfläche. Somit dienen sie in einem Online-Kartensystem zur Positionierung eines Objektes in der Karte. Da es zahlreiche Systeme für geographische Koordinaten gibt und man dadurch beim Umgang mit ortsbezogenen Webservices auf die unterschiedlichsten Formate stoßen kann, ist es notwendig, einige Hintergrundinformationen zu diesem Thema zu behandeln.

Geographische Koordinaten setzen sich aus der geographischen Länge und Breite zusammen, die auf Grund der kugelähnlichen Form der Erde in Längengrad und Breitengrad angegeben werden. Die Breitengrade werden hierbei vom Äquator in nördliche und südliche Breite getrennt und nehmen vom Äquator ausgehend zu bis 90 Grad Nord bzw. Süd. Da es für die Längengrade keine solche natürliche Grenze gibt, hat man sich im zwanzigsten Jahrhundert auf den Meridian von Greenwich in London als Nullmeridian geeinigt.

Da die Erde keine perfekte Kugel, sondern ein abgeflachter Elipsoid ist, benötigen geographische Koordinatensysteme ein Kartenbezugssystem (engl. map datum)

[44]. Ein Referenzellipsoid wird von einem Kartenbezugssystem als mathematische Grundlage zur Annäherung der tatsächlichen Form der Erde verwendet. Im Laufe der Zeit wurden verschiedene Ellipsoide entwickelt, wie zum Beispiel von Bessel (1841) und Krassowski (1940). Diese unterscheiden sich wiederum in ihrer Eignung für die verschiedenen Gebiete der Erde. So ist beispielsweise der Bessel-Ellipsoid für Mitteleuropa am besten geeignet. Beim Global Positioning System (GPS) und auch bei den Anbietern von Online-Kartenmaterial hat sich das Bezugssystem „World Geodetic System 1984“ (WGS84) mit dem Elipsoiden WGS84 mittlerweile durchgesetzt. Dies ermöglicht die einfache Darstellung von GPS-Koordinaten in Online-Karten, aber auch in Kartenmaterial von Navigationssystemen.

Da im Rahmen dieser Arbeit ein mobiles, dynamisches GIS-basiertes System entstehen soll, welches auch lokale oder nationale Dienste integriert, ist es wichtig, die Unterschiede zu kennen. So unterscheiden sich die einzelnen Kartenbezugssysteme (in Deutschland z.B. das Potsdam-Datum) untereinander und die Ellipsoide untereinander in vielen Punkten. Dies bezieht sich zum Beispiel auf die Werte für Äquatorlänge und die Position des Mittelpunkts des Ellipsoiden und des Koordinatensystems. Das WGS84-System beispielsweise vereinfacht die Frage nach den Mittepunkten, in dem für beide Mittelpunkte der Schwerpunkt der Erde angenommen wird. Dies genügt bei einem weltweiten, einheitlichen Kartenbezugssystem, führt aber durch die „Delligkeit“ der Erde in manchen Gebieten zu Ungenauigkeiten. Die nationalen Vermessungsbehörden verwenden vor allem wegen der Genauigkeit bis in den Millimeterbereich Bezugssysteme, die für die jeweilige Region bzw. das jeweilige Land entwickelt wurden.

Diese zahlreichen Unterschiede zwischen den Systemen machen es zum Teil sehr komplex und rechenaufwendig, Transformationen zwischen zwei Systemen vorzunehmen. Mit dem Wunsch nach Genauigkeit nimmt auch der Rechenaufwand zu, da es sich zum Teil um rekursive Näherungsverfahren handelt. Daher ist die Koordinatentransformation auf mobilen Endgeräten nicht üblich.

2.2.2 Global Positioning System

Das Global Positioning System (GPS) ist ein Satelliten-gestütztes System zur Ermittlung der Geokoordinaten eines GPS-Empfangsgerätes und wurde 1993 zur weltweiten, zivilen, kostenlosen Nutzung freigegeben, damals noch mit reduzierter Genauigkeit (Selective availability (SA)) [45].

Die Satelliten umkreisen die Erde und senden dabei Signale kostenlos an GPS-Empfänger auf der Erde. Diese Signale können zur Bestimmung des Ortes des GPS-Empfängers auf der Erdoberfläche oder innerhalb der Atmosphäre genutzt werden. Daher kann das GPS zur Navigation an Land, zur See und in der Luftfahrt und mittlerweile auch für GPS gestützte Landesvermessung verwendet werden.

Mit der Abschaltung der Selective Availability im Jahr 2000 verbesserte sie die durchschnittliche Genauigkeit von ca. 100m auf ca. 20m. In Abhängigkeit von der Anzahl empfangbarer Satelliten steigt die Genauigkeit mit einem normalen zivilen Empfänger auf bis zu ca. 3m. Professionelle Geräte, wie in der Landesvermessung üblich, kommen auf eine Genauigkeit von wenigen Zentimetern [46]. Zur weiteren Steigerung der Genauigkeit werden Bodenkontrollstationen genutzt, welche die Bahnen der Satelliten überwachen und die gesammelten Informationen an die zentrale Master control station senden [47]. Dort werden die Daten aufbereitet, sodass sie als Korrekturinformation an die Satelliten gesendet werden können. Außerdem können die Satelliten noch untereinander kommunizieren, um ihre Bahninformationen auszutauschen. Zusätzliche Maßnahmen wie Differentielles GPS (DGPS) oder Wide Area Augmentation System (WAAS) sollen dazu beitragen, die Genauigkeit von normalen zivilen GPS-Empfängern auf bis zu unter einem Meter zu bringen.

Für die Positionsbestimmung senden die Satelliten an die GPS-Empfänger, wer sie sind, wo sie sind, die genaue Uhrzeit des Sendens und Informationen zur Umlaufbahn [48]. Der Empfänger berechnet aus diesen Daten und der Empfangszeit die Entfernung zu jedem empfangbaren Satelliten. Um das Relief der Erde zu berücksichtigen, wird die 3D-Position (3D position fix) mittels vier oder mehr Satelliten ermittelt, da hierdurch zusätzlich zur Position auf der Erdoberfläche noch die Höhe berechenbar ist. Durch Auswertung der Positionsinformationen wird die Uhrzeit im Empfänger mit den Atomuhren der Satelliten synchronisiert. Auch dies steigert die Genauigkeit.

Trotz aller Bemühungen zur Genauigkeitssteigerung für normale GPS-Geräte wurde die für Vermessungsangelegenheiten notwendige Zentimeter-Genauigkeit bisher noch nicht erreicht. Eine Anwendung von Android im Vermessungsbreich scheint daher zur Zeit ausgeschlossen. Es ist aber nicht auszuschließen, dass zukünftige Weiterentwicklungen des GPS-Systems oder der Empfängertechnik die Situation ändern könnten. Für das Beispielszenario und für die meisten Einsatzszenarien im Endverbraucherbereich reicht hingegen die aktuell durch Smartphones mit GPS Empfänger bereitgestellte Genauigkeit bei weitem aus.

2.2.3 Geodaten

Geodaten sind die „(...) computergeeignete (transportierbare, verkaufbare) Form von Geoinformation“ (nach ISO 19107) [49], die in Geo-Informationssystemen (GIS) verarbeitet werden. Geoinformationen sind „Informationen zu Erscheinungen, die direkt oder indirekt mit einer auf die Erde bezogenen Position verbunden sind.“ (nach ISO 19107) [50] Es sind „(...) Daten über Gegenstände, Geländeformen und Infrastrukturen an der Erdoberfläche, wobei als wesentliches Element ein Raumbezug vorliegen muss. (...) Geodaten beschreiben Objekte, die durch eine Position im Raum (räumliche Lage auf der Erde, Anm. d. Verf.) direkt (z.B. durch

Koordinaten) oder indirekt (z.B. durch Beziehungen) referenzierbar sind.“ [51].

2.2.4 Geodateninfrastruktur

„Als Geodateninfrastruktur (GDI) werden die technologischen und organisatorischen Maßnahmen und Einrichtungen sowie die begleitenden politischen Entscheidungen verstanden, die sicherstellen, dass Methoden, Daten, Technologien, Standards, finanzielle und personelle Ressourcen zur Gewinnung und Anwendung von Geoinformationen zur Verfügung stehen.“ [52].

Besonderer Wert wird hierbei auf den standardisierten Austausch von Geodaten zwischen Geodaten-Produzenten, Geo-Dienstleistern und -Nutzern gelegt. Dieser erfolgt über ein Datennetzwerk wie zum Beispiel dem Internet. So können auch Geofachdaten der einzelnen Fachbereiche und Institutionen auf diese Weise zusammengeführt werden.

Die Geodaten werden in Geodatenbanken abgelegt, wobei die standardisierten Geodienste den Zugriff auf die Daten gewährleisten. Auf diese Weise können Geoportale wie Google Maps, aber auch GIS-Client-Programme standardisiert auf die Geodaten zugreifen. Die zugreifenden Clients stellen wiederum dem Endnutzer eine graphische Benutzeroberfläche zur Verfügung. Es liegt also eine serviceorientierte Architektur vor.

Die Standardisierungen finden sich in den Normen der Serie ISO 19100 und in den Implementierungsspezifikationen des Open Geospatial Consortium (OGC)(siehe Kap. 2.2.5) [53]. Die technologische Umsetzung einer Geodateninfrastruktur ist ein Geoinformationssystem (GIS).

2.2.5 Geodienste

Geodienste sind standardisierte Dienste zur Generierung von digitalem Kartenmaterial und Integration von georeferenzierten Informationen in diese Karten durch ein Netz von Diensten. Dieser vorwiegend serverseitig orientierte Ansatz soll im Kapitel 5 einem clientseitigen Ansatz und der Anforderung, sowohl standardkonforme als auch nicht-standardkonforme Dienste zu integrieren, gegenübergestellt werden. Für diese Untersuchung ist ein tieferes Verständnis von Geodiensten und ihrer Standardisierung notwendig, welches in diesem Abschnitt vermittelt werden soll.

Seit ihrer Gründung 1994 ist es Ziel des Open Geospatial Consortium (OGC) den Wandel weg von monolithischen GIS hin zu verteilten, interoperablen Geodiensten (Webservices) durch Standardisierung und Schnittstellenspezifikationen auf Basis der Normen der Serie ISO 19100 zu unterstützen [54].

Die OGC ist ein „Zusammenschluss aller relevanten GIS Anbieter, GIS Nutzer (Behörden, Firmen) und Verbände“ [55].

Die von der OGC definierten Standards umfassen beispielsweise ein XML-basiertes Austauschformat für Geodaten (OpenGIS Geography Markup Language (GML)) und einen Kartenausschnitte generierenden Dienst (Web Map Service (WMS)) [56, 57].

Dieser Standard wird neben dem Web Feature Service (WFS - für Featuredaten) und dem Web Coverage Service (WCS - für Rasterdaten) von der Software Geoserver implementiert und frei zur Verfügung gestellt [58]. Mit Openlayers existiert eine freie JavaScript Bibliothek vergleichbar zur Google Maps API, die auf WMS wie z.B. Geoserver zugreifen kann und dadurch beim Einbetten von dynamischen, WMS-konformen Karten in Webseiten behilflich ist [59]. Andere OGC-konforme Implementierungen sind Deegree und UMN Mapserver [60, 61].

Die Zusammenarbeit von raumbezogenen Webservices (Geodienste, Geodatendienste) beschreibt die Open GIS Service Architecture. Geodienste sind in einer Geodateninfrastruktur die Vermittler zwischen Client (z.B. ein Geo-WebPortal wie Google Maps) und den Geodaten in den Geodatenbanken, welche in OGC-spezifizierten Datenmodellen (Open Geodata Model) und -formaten (GML) vorliegen. Ein Geodienst macht also Geodaten strukturiert verfügbar.

Ein Beispiel für einen Geodienst ist der bereits erwähnte Web Map Service (WMS) [57]. Er dient der Erstellung von geographischen Karten aus Raster- , Vektor- und Sachdaten in einem verteilten GIS und ihrer Visualisierung in Form von Grafikformaten (JPG, PNG, etc.) meist in einem Webbrowser. Diese Bilddaten können in ihrer Transparenz variiert werden, um mehrere Kartenlayer darstellen zu können. Ein WMS ist Teil eines WebGIS, eine Geo-Portal-Webseite ist ein Client von einem WMS.

Ein anderes Beispiel für einen Geodienst ist der OpenGIS Location Service oder auch OpenLS genannt [62]. Es handelt sich um eine OGC-Schnittstellenstandardisierung für Anbieter von Location Based Services (LBS) wie zum Beispiel Routendarstellung, Verkehrsinformationen oder Restaurant-Finder. Eine Implementierung eines solchen Dienstes ist beispielsweise der Directory Service von openrouteservice.org, welcher dort für den Abruf von Points of Interest (POI) genutzt wird.

Die Spezifikationen der OGC sind sehr weitreichend und so sinnvoll diese Standardisierung auch sein mag, ist die Integration eines vorhandenen, nicht-OGC-Standard-konformen Webservice mit Raumbezug meist mit so großen Umstrukturierungen verbunden, dass sich nur eine völlige Neuentwicklung lohnt. Es ist denkbar, dass daher viele Webservice-Anbieter diesen Aufwand und die damit verbundenen Kosten scheuen. Im Rahmen dieser Arbeit soll versucht

werden einen vergleichsweise einfachen Mechanismus einzuführen, um neben den OpenLS-Diensten auch nicht-OGC-Standard-konforme, raumbezogene Webservices in das mobile, GIS-basierte System zu integrieren.

2.2.6 WebGIS

Ein WebGIS ist eine verteilte GIS Anwendung, die über ein Netzwerk mittels Webbrowser verwendet wird [63]. Ist das WebGIS OGC-konform, so ist es modular in einzelne Geodienste geteilt. Dabei ist mindestens ein Web Map Service, ein Web Feature Service und ein Web Coverage Service beteiligt. Die Kommunikation findet in standardisierten Geodatenformaten statt.

Man unterscheidet statische und interaktive WebGIS. Statische WebGIS stellen nur statisches Kartenmaterial, wie z.B. Anfahrtsskizzen dar, welche keine oder nur geringe Interaktionsmöglichkeiten bieten (z.B. Google Static Maps-API [64]). Interaktive WebGIS hingegen bieten Interaktionsmechanismen wie Pan (Bewegung in der Karte) und Zoom (Kartenausschnitt vergrößern / verkleinern) (z.B. Google Maps-API [65]). Komplexere dynamische WebGIS bieten darüber hinaus noch Editierfunktionen, mit denen die Karte verändert werden kann (z.B. OpenStreetMap-Protocol [66]).

2.2.7 Arbeitsdefinitionen

Ein grundlegender Denkansatz in dieser Arbeit ist die saubere Trennung des Kartenmaterials der Kartenanbieter und der Darstellung der kartenbezogenen Informationen aus den Webservices.

In der Geoinformatik wurde bisher der Ansatz verfolgt, diese Informationen bereits serverseitig zusammenzuführen, indem beispielsweise alle Anbieter ihre Dienste OGC-konform gestalten, um die dafür notwendige Interoperabilität zu gewährleisten. Da bei weitem nicht alle Dienstanbieter ihre Webservices mit dem Ziel der Integration in ein GIS entwickeln, ist nicht zu erwarten, dass jeder Anbieter eines potentiell interessanten, raumbezogenen Webservices sich an die OGC-Standardisierung hält.

Das zu entwickelnde Konzept soll dem Rechnung tragen und ermöglichen, dass nahezu beliebige Webservices in ein WebGIS-basiertes System eingebunden werden können, ohne die Dienste extra anzupassen. Da hierbei leicht der aktuelle Konzeptraum der Geoinformatik verlassen wird, ist es unabdinglich, neue Begriffe in Form von Arbeitsdefinitionen festzulegen. Sie sollen vor allem dazu beitragen, effektiv und einfach über die Thematik und das neue Konzept sprechen zu können.

2.2.7.1 MapItem

Ein MapItem repräsentiert ein Geodatum auf der Karte, dass ein einzelnes materielles Objekt aus der realen Welt beschreibt. Die Position des Objekts kann ausreichend durch ein Koordinatenpaar beschrieben werden. Seine räumliche Ausdehnung wird hierbei vernachlässigt. Es wird durch eine kleine Grafik an seiner koordinatengegebenen Position auf der Karte symbolisiert. Die Interaktion mit dem MapItem erfolgt über dieses Symbol. Ein MapItem kann im Zusammenhang mit einer geeigneten Funktion eines MapService (vgl. Kap. 2.2.7.2) verwendet werden,z.B. als Zielort in einer Navigationsanwendung.

2.2.7.2 MapService

Ein MapService ist ein über das Internet verfügbarer Location Based (Web)Service (LBS), der Informationen über MapItems bereithält oder Funktionen zur Verfügung stellt, die in Bezug auf solche Objekte verwendet werden können.

Zur Darstellung in der Karte und Nutzung der Funktionalität (durch Interaktion mit einem MapItem) muss mindestens ein MapService verfügbar sein, der genau diesem MapItem seine Koordinaten eindeutig zuordnet.

Ein MapService kann hierbei ein OGC-konformer OpenGIS Location Service (OpenLS) sein, muss es aber nicht zwangsläufig. Ein gutes Beispiel für einen OGC-MapService der als MapItems Points of interests (POI) anbietet, ist beispielsweise der Directory Service von openrouteservice.org.

2.2.7.3 MapProvider

Ein MapProvider ist die technische und organisatorische Abstraktion eines Anbieters eines GIS. Ein MapProvider abstrahiert den Online-Zugriff auf das von ihm zur Verfügung gestellte Kartenmaterial mit einer API. Diese kann eine Web-API, aber auch eine native Programm-API sein. Weitere Details und Anforderungen an MapProvider werden in den Kapiteln 3.3.3 und 4.1 diskutiert.

2.2.8 Schlussfolgerungen

Aus den im Kapitel 2.2.1 genannten Gründen wird vorerst die Koordinatentransformation auf dem mobilen Endgerät ausgeschlossen. Daraus folgt, dass ein einzubindender ortsbezogener Webservice („MapService“) seine Geodaten mit den international üblichen WGS84 Koordinaten referenzieren muss. Eine eventuell notwendige Umrechnung muss dann serverseitig erfolgen, was zu Performancezwecken einmalig erfolgen und dann persistent gemacht werden sollte.

Alternativ könnte auch ein Webservice zur Koordinatentransformation entwickelt und vom mobilen Endgerät zur Transformation verwendet werden.

In Deutschland ist dies ebenfalls sehr relevant, da hier viele Koordinaten von immobilen Objekten wie Gebäuden oder infrastrukturelle Einrichtungen wie Haltestationen als für deutsche Landesvermessungsämter typische Gauß-Krüger-Koordinaten vorliegen [67]. Die Umrechnung in das WGS84 Format ist auch hier sehr komplex und rechenaufwendig.

Die OGC standardisiert die serverseitige Zusammenarbeit von Kartenerzeugung und Informationsintegration. Jedoch sind nicht alle ortsbezogenen Webservices mit diesem Standard konform, wodurch diese nicht ohne Anpassung von OGC-konformen WebGIS eingebunden werden können.

Um diesem Problem zu begegnen ist es Ziel dieser Arbeit, einen Ansatz zu finden, sowohl OGC-konforme als auch nicht-standardkonforme Webservices in digitale Karten zu integrieren. Da dies eine Trennung von Kartengenerierung und Informationsintegration bewirkt, wurde zur Abgrenzung vom OGC-Ansatz die Begriffe MapItem, MapService und MapProvider eingeführt.

2.3 Fazit

Kapitel 2.1 führte in die Grundlagen der mobilen Plattformen ein, verglich die vorhandenen Technologien und wählte anhand zahlreicher Kriterien die für diese Arbeit am besten geeignete Plattform Google Android aus. Im Kapitel 2.2 wurde in die Grundlagen der Geoinformatik im begrenzten, aber notwendigen Maß eingeführt, so dass das begriffliche Fundament für den weiteren Verlauf der Arbeit gelegt werden konnte. Ebenso wurden Arbeitsdefinitionen angegeben, die den Umgang mit den Entitäten dieser Domäne sprachlich vereinfachen sollen.

3 Anforderungsanalyse

In diesem Kapitel sollen die Anforderungen des Beispielszenarios analysiert werden, um davon zu abstrahieren und so die Anforderungen für ein breites Spektrum von MapServices und ihrer Kartenintegration zu erfassen. Dazu wird das Beispielszenario, dass in der Einführung erwähnt wurde zunächst näher beschrieben und verallgemeinert. Auf Basis der Verallgemeinerung werden dann funktionale und nicht-funktionale Anforderungen indentifiziert. Die szenariospezifischen Anforderungen werden abschließend in Pflicht- und Wunschkriterien unterteilt und aufgelistet.

3.1 Beschreibung des Beispielszenarios

Im Rahmen der Zusammenarbeit mit dem Verkehrsverbund Oberelbe (VVO) wurde ein Beispielszenario entwicklet, dessen Umsetzung im Interesse des VVO ist und eine gute Möglichkeit bietet, den zu entwickelnden generischen Lösungsansatz an einem konkreten praktischen Beispiel zu testen und auch die Eignung des Ansatzes daran zu validieren.

Im Mittelpunkt des Systems steht eine Anwendung für Smartphones mit dem Android-Betriebssystem. Da für das iPhone bereits eine Entwicklung durch den VVO in Auftrag ist und Android bisher nicht betrachtetet wurde, ist der VVO sehr interessiert an den sich durch diese weitere Plattform bietenden Möglichkeiten. Die im Grundlagenkapitel getroffene Auswahl der Android-Plattform kommt dem Wunsch der VVO daher sehr entgegen. Die Anwendung soll in den Karten eines Online-Kartendienstes (MapProvider) die Haltestellen innerhalb des VVO-Gebiets darstellen, deren Namen, Zusatzinformationen und Geokoordinaten aus einem Webservice bezogen werden. Die grundlegende Nutzerinteraktion ist das Auswählen dieser Haltestellen mittels Klick in der Karte.

Die Auswahl einer Haltestelle soll den zu dieser Haltestelle gehörigen, aktuellen Abfahrtsmonitor in einem separaten Fenster anzeigen. Für den Bezug dieser Informationen wird derselbe VVO-Webservice genutzt, der auch die VVO-Widgets

für Desktopsysteme versorgt. Es ist also wichtig, dass die vorhandene Webservice-Infrastruktur der VVO unverändert genutzt werden kann, sodass durch diesen zusätzlichen Webservicekonsumenten außer der Zugriffslast kein zusätzlicher Aufwand anfällt.

Die Anwendung soll das nachträgliche Hinzufügen weiterer Dienste der VVO-Webserivces aufwandsarm ermöglichen. Ein Beispiel hierfür ist die Routenplanung mit Anzeige der Route in der Karte. Vor allem wäre hierbei auf die zusätzliche Darstellung von Umstiegsinformationen zu achten. Start- und Zieladresse, welche sowohl Haltestellen als auch Gebäudeadressen sein können, sollten hierbei ebenfalls bequem durch direkte Interaktion mit der Karten festlegbar sein. Alternativ ist auch eine textuelle Eingabe vorzusehen.
Da ein soches Modul bzw. ein solcher Modulkomplex wesentlich komplexer ist als die Intergration des Abfahrtsmonitors ist diese Funktion lediglich ein Wunschkriterium. Es ist verständlich, das ein so umfangreiches Modul in Rahmen einer Diplomarbeit nicht zufriedenstellen fertiggestellt werden kann. Da zur Demonstration des zu entwickelnden Konzepts die Integration des Abfahrtsmonitor-Webservices vollkommen genügt, wird sich in der Implementation vorwiegend auf diesen konzentriert. Die Belange des Routenplanungs-Modulkomplexes sind aber in Konzeption und Implementierung in ausreichendem Maße mitzubetrachten, um die Ansprüche an Felxibilität und Erweiterbarkeit des Konzeptes an einem konkreten Beispiel festzumachen.

3.2 Verallgemeinerung des Beispielszenarios

Um im Rahmen des Beispielszenarios keine monolithische, speziell auf die Belange des VVO zugeschnittene Software zu erstellen, sondern einen allgemeingültigen, generischen Ansatz zu finden, sind die Anforderungen des VVO nun zu verallgemeinern. Kernpunkt der Verallgemeinerung ist, dass beliebige Webservices eingebunden werden können, deren Inhalte sinnvoll auf einer Karte dargestellt werden können (bspw. OGC konforme Dienste). Diese Eigenschaft limitiert die sinnvoll einbindbaren Webservices auf die zuvor definierten „MapServices“.

Die Software soll das Anzeigen von kartenbezogenen Objekten (MapItems) in einem Online-Karten-System ermöglichen. Diese MapItems sind die zentralen Objekte der jeweiligen kartenbezogenen WebServices („MapService“). Der Nutzer soll mit diesen MapItems im Sinne des jeweiligen MapService interagieren können. Vorrangig durch direkte Auswahl eines MapItems (Auswahl durch Klick auf das MapItem in der Karte) sollen die durch den zugrundeliegenden, jeweiligen

MapService zur Verfügung gestellten Funktionionalitäten dem Nutzer auf eine intuitive und integrierte Weise zur Verfügung stehen. Sind zu einem MapItem mehrere Funktionen möglich, wird dem Nutzer eine Liste dieser gezeigt, die eine Auswahl der gewünschten konkreten Funktion ermöglicht. Nach Auswahl der gewünschten Funktion dieses MapItems (bspw. die Verwendung als Startadresse eines Navigationsdienstes) wird der Inhalt der Funktionalität entweder in der Karte integriert oder in einem separaten Fenster über der Karte dargestellt.

3.3 Funktionale Anforderungen

In diesem Kapitel sollen die Anforderungen, die aus dem Beispielszenario, dem daraus verallgemeinerten Szenario und den nicht-funktionalen Anforderungen resultieren, systematisch und aus funktionalem Gesichtspunkten aufgelistet werden.

Das Softwaresystem besteht aus verschieden Aspekten und darum gliedern sich auch die funktionalen Anforderungen in diese Kategorien.

Diese Aspekte sind:

- F_1: Modularisierung in dynamische Softwarekomponenten
- F_2: Einbindung der MapServices
- F_3: Einbindung der MapProvider
- F_4: Spezifische Funktionen des Beispielszenarios

3.3.1 F_1 Modularisierung in dynamische Softwarekomponenten

F_1.1: Die Software ist in ein Hauptprogramm (Android-Applikation) und die von ihm verwendeten Moduls („PlugIns“) zu gliedern.

F_1.2: Die Zugriffslogik auf einen MapService soll in dem Modul gekapselt sein, welches die von diesem MapService bereitgestellte (Teil-)Funktionalität dem Hauptprogramm verfügbar macht. (Funktionsorientierte Gliederung) Eine Untergliederung dieser Module in Datenprovider-Module und zugehörigem Android-User-Interface-Module ist denkbar, aber nicht unbedingt nötig. Es kann die Wiederverwendbarkeit steigern, aber erhöht auch die Komplexität geringfügig.

F_1.3: Für die verschiedenen Funktionen einunddesselben MapServices können wenn sinnvoll oder nötig verschiedene Module entwickelt werden. Es ist hierbei darauf zu achten, die Überlappungen zwischen den einzelnen Modulen gering zu halten und somit die Wiederverwendbarkeit zu stärken.

F_1.4: Die Module zur Einbindung von Teilfunktionalitäten sollen möglichst unabhängig von anderen Modulen sein, um die funktionalen Abhängigkeiten zu reduzieren. Ein Modul, welches eine komplexe Funktion aus diesen Teilfunktionalitäten kombiniert, ist logischerweise immer abhängig von diesen Modulen.

Allerdings soll die Abhängigkeit zwischen den Teilfunktionsmodulen minimal sein, um ihre flexible Wiederverwendbarkeit zu gewähren.

F_1.5: Neben den Modulen zur Einbindung der MapServices soll auch die Einbindung verschiedener MapProvider modularisiert gekapselt werden. Dies soll die einfache Austauschbarkeit und Erweiterbarkeit auf den Bereich der unterstützten Online-Karten ausdehnen.

F_1.6: Das Hauptprogramm soll ein Modulmangement beinhalten, mit dem aus einer Liste online verfügbarer Modul-PlugIns die Erweiterungsmodule ausgewählt, heruntergladen und in das Programm integriert werden können.

F_1.7: Das Modulmanagement soll auch das Entfernen nicht mehr benötigter Module ermöglichen

F_1.8: Eventuelle Abhängigkeiten zwischen Modulen sollen automatisch aufgelöst werden, ohne zu viel Fachkenntnis und Interaktion vom Nutzer zu erwarten.

3.3.2 F_2 Einbindung der MapServices

F_2.1: Jeder integrierte MapService wird durch mindestens ein Modul eingebunden.

F_2.2: Die Module sollen untereinander möglichst unabhängig sein, insofern sie keine kompositen Module sind, die eine komplexe Funktionalität aus den Teilfunktionalitäten verschiedener Module kombinieren.

F_2.3: Die Einbindung ist unabhängig von der Art des Webservice (.do, XML-RPC,JSON-RPC, SOAP, REST, HTML-Parsing), seiner Schnittstellenspezifikation, der verwendeten Daten- und Nachrichtenformate und der zu Grunde liegenden Implementierung möglich. Dies kann lediglich durch die Möglichkeiten der verwendten Plattform (Android) limitiert werden.

F_2.4: Grundsätzlich ist es nur sinnvoll Webservices einzubinden, die insofern zur Kartendarstellung geeignet sind, dass sie ihre zentralen Objekte georeferenziert zur Verfügung stellen und sie somit der Definition der MapServices entsprechen. Ein MapService muss also alle zur Darstellung in einer Karte notwendigen Informationen anbieten, sonst ist es nicht möglich den Dienst sinnvoll in eine an direkter Karteninteraktion orientierten Anwendugn zu integrieren.

F_2.5: Die vorhandenen Webservices sollen nicht verändert werden müssen, um die Zusammenarbeit mit der Android-Anwendung zu ermöglichen. Das Konzept ist so auszulegen, das die hierfür notwendige Flexibilität auf Webservice-Konsumentenseite liegt.

3.3.3 F_3 Einbindung der MapProvider

Da die Einbindung verschiedener MapProvider außer dem in Android integrierten Zugriff auf Google Maps bezüglich Komplexität, Zeitaufwand und genereller Machbarkeit zur Zeit nicht abschätzbar ist, muss dieser Abschnitt genrell als Wunschkriterienliste gelesen werden.

F_3.1: Die Integration des Zugriffs auf das Kartenmaterial eines MapProviders ist vergleichbar zur Einbindung der MapServices in einzelne Module zu kapseln.

F_3.2: Die MapProvider-Module müssen unabhängig von anderen Modulen funktionieren. Eine evtl. benötigte Bibliothek ist in das Modul zu integrieren.

F_3.3: Der Funktionsumfang des MapProvider-Moduls soll möglichst der Funktionalität der Google Maps Bibliothek für Android entsprechen. Da der Prototyp sich sinnvollerweise auf Google Maps stützen sollte, ist es sinnvoll diese Bibliothek in ihrem Funktionsumfang als Maßstab zu nehmen. Diese Funktionen sind unter anderem:

* Navigation in der Karte durch Nutzerinteraktion (Zoom, Pan)
* Programmatische Navigation in der Karte
* Geocoding und Reverse Geocoding
* Darstellung von MapItems in/über der Karte
* Darstellung von Routen in/über der Karte

F_3.4: Performance, Verfügbarkeit und Interaktionmöglichkeiten sollen sich nicht nennenswert von der Android-eigenen Google Maps Anwendung unterscheiden, damit eine einheitliche User-experience erreicht wird

F_3.5: Eingebundene MapProvider dürfen bezüglich ihrer Lizenzmodelle nicht konfligär mit den Lizenzen bzw. Nutzungsbedingungen der integrierten MapServices sein

F_3.6: Die zugrunde liegenden MapProvider-APIs sollten möglichst quelloffen sein und nicht-virale Lizenzmodelle verwenden

3.4 Nicht-funktionale Anforderungen

Damit die Software wie bereits angeführt nicht monolithisch auf das Beispielszenario zugeschnitten ist, ist es erforderlich, die Aspekte Flexibilität und Erweiterbarkeit zu betrachten. Aus diesen beiden Aspekten resultieren die Aspekte der Modularität der Architektur und der Dynamik und Austauschbarkeit der Softwarekomponenten (Module).

NF_1: Die Flexibilität bezieht sich vor allem auf die Beliebigkeit der einzubindenden MapServices, beispielsweise bezüglich ihres Typs, Zugriffsmechanismen oder Datenformaten. Die Schnittstellen sollen daher die Implementierung nicht zu sehr einschränken und es sind Konzepte vorzusehen, die die Schnittstellenspezifikation unabhängig vom konkreten Einsatzszenario halten. Es soll auf die Gemeinsamkeiten der verschiedenen MapServices, wie zum Beispiel das MapItem als ein zentrales Objekt hingearbeitet werden.

NF_2: Die Erweiterbarkeit und Modularität bezieht sich auf die parallele Existenz verschiedener Module, die wiederum den Zugriff auf verschiedene MapServices verkörpern. Die Funktionalität, um die die Anwendung erweitert werden soll, muss daher in ein Modul gekapselt werden. Eine komplexe Funktionalität, die sich aus verschiedenen Teilfunktionalitäten zusammensetzt, besteht daher möglicherweise auch aus verschiedenen Modulen. Diese Vorgehensweise ermöglicht oft die Wiederverwendbarkeit einzelner Module in einem anderen Kontext und ist daher zu bevorzugen.

NF_3: Die genannten Module sollen dynamisch hinzuladbar sein (z.B. Dynamsiches Deployment per Download) und automatisch integriert werden. Bei Bedarf sollen sie auch wieder entfernt werden können, um beispielsweise Speicherplatz freizugeben. Hierfür ist ein Modulmanagement vorzusehen, mit dem der Nutzer neue Module zum Download und Installation wählen kann und vorhandene Module deintallieren kann.

Die Austauschbarkeit bezieht sich auf zwei Aspekte.

NF_4: Der erste Aspekt ist die Möglichkeit der Aktualisierung des Moduls durch Bereitstellung der aktuellen Version zum Download. Die neue Version soll die alte problemlos ersetzen können, ohne dass die Stammanwendung verändert werden muss (Dynamik-Aspekt). Der parallele Betrieb zweier unterschiedlicher Versionen des gleichen Moduls ist nicht sonderlich sinnvoll und ist aus Speicherplatzgründen möglichst zu meiden.

NF_5: Der zweite Aspekt der Austauschbarkeit ist das Ersetzen einer Modulimplementierung eines Herstellers mit der eines anderen. Dies soll möglich sein, für den Fall, dass beide Implementierungen die gleiche Schnittstelle realisieren. Flexibilität und Austauschbarkeit sollen also unter anderem durch den konsequenten Einsatz von Schnittstellenspezifikationen erreicht werden.

NF_6: Ein weiterer Aspekt der nicht-funkionalen Anforderungen ist die Performance. Die Anwendnung für das Smartphone soll ausreichend performant sein um eine angenehme User-experience zu erreichen. Hierfür ist insbesondere bei Smartphones auf Speichersparsamkeit zu achten. Rechenintensive Prozesse dürfen die Nutzeroberfläche nicht interaktionsunfähig machen und müssen durch geeignete Maßnahmen wie Multithreading von dieser getrennt werden.

NF_7: Die bei den MapServices verursachte Rechenlast bzw. Anzahl und Frequenz der Anfragen ist auf das Notwendigste zu reduzieren.

NF_8: Gleiches gilt auch für die zu übertragenden Daten, was vor allem bei schwacher Netzanbindung des Smartphones an das Internet sehr relevant ist. Beispielsweise durch Mechanismen wie Caching oder nutzerdefinierte Filtrierung (Ausschluss von Bildern und Videos etc.) der übertragenen Daten könnte dem Rechnung getragen werden. Da das Online-Kartenmaterial bereits einen großen Teil der Bandbreite beansprucht, ist die Reduktion der MapSerivce-Daten zusätzlich von erhöhter Wichtigkeit. Außerdem soll betrachtet werden, inwieweit das Kartenmaterial gecacht werden kann, um die Menge der übertragenen Daten pro Zeiteinheit zu reduzieren.

3.5 Funktionen des Beispielszenarios

3.5.1 Pflichtkriterien

P_1: Darstellung der Haltestellen der VVO im Onlinekartenmaterial innerhalb der Android-Anwendung auf Basis eines Webservice (VVO-eigen oder OpenLS-konformer Dienst)

P_2: Interaktion (durch Klick) mit einem Haltestellensymbol öffnet deren aktuellen Abfahrtsmonitor dieser Haltestelle auf Basis des VVO-eigenen Webservice

P_3: Es ist eine Möglichkeit vorzusehen, VVO-fremde POI-Dienste leicht in die Anwendung zu integrieren, um Synergieeffekte zwischen interessanten Örtlichkeiten und der Verwendung des Nahverkehrs zu erreichen.

P_4: Diese POI-Dienste sollen hierfür nicht von der VVO betreut oder gewartet werden müssen.

3.5.2 Wunschkriterien

W_1: Routendarstellung in der Karte zur Anzeige einer durch einen VVO Webservice berechnete Verbindungsauskunft.

W_2: Eingabe von Start und Ziel der Verbindung durch Interaktion mit der Karte

W_3: Eingabe von Start und Ziel der Verbindung durch textuelle Eingabe

W_4: Berechnung der Route auf Basis eines VVO Webservice

W_5: Auswahlbildschirm zur Auswahl aus der Menge der unterschiedlichen Routen und Abfahrtszeiten

W_6: Auswahl, ob aktuelle Zeit oder gewählte Zeit als Abfahrts oder Ankunftszeit verwendet werden soll

W_7: Auswahl der zu verwendenden Transportmittel

3.6 Fazit

In diesem Kapitel wurde das VVO-Beispielszenario aus der Einführung näher beschrieben und von ihm ausgehend wurde auf eine allgemeinere Problemstellung geschlossen. Dabei konnten eine Reihe von funktionalen und nicht-funktionalen Anforderungen indentifiziert werden.

Die gefundenen Anforderungen werden in den folgenden Kapiteln verwendet um beispielsweise die Eignung einer Technologie oder ein Artefakt der Konzeption oder Implmentierung mit einer konkreten Anforderung zu verknüpfen. Das Kaptel Evaluation untersucht dann die Erfüllung der hier indentifizierten Anforderungen durch den Prototypen.

4 State-of-the-art-Analyse

In diesem Kapitel soll der aktuelle Stand der Technik der verschiedenen Aspekte dieser Arbeit untersucht werden. Dabei werden drei Hauptaspekte betrachtet. Zunächst werden die verfügbaren Anbieter von Online-Kartenmaterial (MapProvider) untersucht und auch deren Eignung für Android betrachtet. Anschließend sollen die verschiedenen vorhandenen Ansätze zur Integration von Geodaten in die WebGIS Karten der zuvor untersuchten MapProvider betrachtet werden. Die anschließende Analyse der im Gebiet des Beispielszenarios befindlichen, vorhandenen Anwendungen soll Bewusstsein für die Probleme und Unzulänglichkeiten dieser Lösungen schaffen und klären, wie sich der Ansatz dieser Diplomarbeit diesen gegenüber positioniert. Abschließend wird der wichtigste Aspekt dieser Abeit, die Dynamik der Softwarekomponenten, gründlich untersucht. Ziel dieses Kapitels ist es, umfangreiches Wissen in den genannten Gebieten zu erarbeiten, welches in der anschließenden Konzeption Anwendung finden wird.

4.1 MapProvider

Ein MapProvider wurde zuvor definiert als Anbieter, der digitales Kartenmaterial online verfügbar macht. In den letzen Jahren haben verschiedene große IT Firmen wie Google, Microsoft und Yahoo eigene WebGIS entwickelt und stellen sie den Internetnutzern frei zur Verfügung. Darüber hinaus gibt es noch kommerzielle Angebote, die kostenpflichtigen Zugang zu sehr genauem oder mit speziellen Informationen ausgestatteten Kartenmaterial anbieten. Diese kommen allerdings aus Kostengründen im Rahmen dieser Arbeit nicht in Frage. Die frei verfügbaren WebGIS haben außerdem eine hohe Qualität erreicht, die für die meisten zivilen und privaten Zwecke vollkommen ausreicht.

Als weiterer MapProvider und Gegenpol zu den kommerziellen WebGIS hat sich ein Open Source Community-Projekt namens OpenStreetMap (OSM) gegründet. Das Kartenmaterial wird hier nach dem Wikipedia-Prinzip von Nutzern gesammelt und eingestellt. Der Nutzer kann die vorhandene Karte editieren und erweitern.

Da das OSM Projekt ein freies GIS ist und auch in Zukunft bei der Verwendung des Materials keine Lizenzprobleme auftreten dürften [68], ist dieses Projekt

sehr interessant für diese Arbeit. Auch kann sich die Qualität, Genauigkeit und Detailreiche oft mit den kommerziellen WebGIS messen.

Das angebotene Kartenmaterial muss ausreichend detailliert mit Kartenfeatures wie Straßen und Straßennamen ausgestattet sein, um Orientierung im Kartenmaterial zu ermöglichen. Diese Anfordungen an die in der Karte integrierten Informationen werden durch die beiden genannten MapProvider ausreichend abgedeckt.

Als wichtige Voraussetzung muss ein MapProvider in seiner API das Anzeigen eines mittels Koordinaten gegebenen Punktes ermöglichen. Desweiteren muss das Setzen von Markierungen an diesen Punkten möglich sein. Es ist völlig ausreichend, wenn dies ausschließlich clientseitig erfolgt.

Zusätzliche Funktionen, die von manchen MapProvidern angeboten werden, wie das Geocoding oder Reverse Geocoding sind nützlich und wünschenswert, aber nicht unbedingt erforderlich. Geocoding ist das Zuordnen eines Koordinatenpaares zu einem Objekt (MapItem), bzw. das Abrufen dieser Zuordnung. Hat man beispielsweise die Adresse eines Hauses, so liefert Geocoding die Koordinaten dieser Adresse. Das Reverse Geocoding ermittelt zu einem Koordinatenpaar die Adresse oder eine Bezeichnung eines Objekts, häufig auch von Objekten in einem bestimmten Umkreis um das Koordinatenpaar.

Bei der Verwendung der Android-Plattform ist Google Maps als MapProvider naheliegend. Dem Android SDK liegt die Google API bei, die den programmatischen Zugriff auf Google Maps ermöglicht. Von dieser macht auch die Android-eigene Google Maps - Betrachtungssoftware „Android Maps“ Gebrauch. Durch diese Bibliothek kann Google Maps Kartenmaterial leicht in eigene Anwendungen integriert werden.

Die zwei Vertreter Google Maps und OSM sind Beispiele für MapProvider, die nun detailierter beschrieben und auf ihre Eignung für Android und diese Diplomarbeit untersucht werden sollen. Die MapProvider Microsoft Virtual Earth und Yahoo Maps werden hier nicht weiter betrachtet, da für sie zur Zeit noch keine Zugriffs-API für Android verfügbar ist [69, 70] (Stand August 2009).

4.1.1 Google Maps

Google Maps ist das wohl bekannteste WebGIS im Internet. Es wird den Nutzern von Google frei zur Verfügung gestellt und finanziert sich Google-typisch durch Werbung und den bezahlten georeferenzierten Eintrag von Firmen in die Google Maps Karte.

Google Maps liefert grafisches Kartenmaterial mit überlagertem Straßennetz und wahlweise Satellitenbilder. Eine Hybrid-Darstellung, die das Staßennetz auf den

Satellitenbildern darstellt, ist ebenfalls vorhanden.

Das Karten- und Satellitenbildmaterial ist in Form einer weltweiten, kontinuierlichen Karte dargestellt, allerdings unterscheidet sich der Detailgrad des Kartenmaterials von Region zu Region zum Teil stark. Grund dafür ist, dass Google Maps die verschiedensten Quellen für Kartenmaterial nutzt und sie in einem zentralen Geoportal konsolidiert.

Das macht Google abhängig von der Verfügbarkeit brauchbaren Materials für eine Region, von der Qualität dieses Materials und von den Preisen für Lizenzen der Kartenmaterial-Urheber. In Europa sind diesbezügliche Probleme vor allem in den Gebieten von mittleren und großen Städten und Ballungsgebieten nicht so gravierend, da Google vermutlich hier auch höhere Prioritäten gesetzt hat, als beispielsweise bei den ländlichen Regionen von Mecklenburg-Vorpommern.

Der Nutzer hat bei Google Maps keine Möglichkeit, die Karte zu editieren, um sie zu aktualisieren oder bei erkannten Fehlern zu korrigieren. Dies muss völlig Google und seinen Datenlieferanten überlassen werden.

Ein Vorzug von Google Maps ist sein leistungsfähiges Geocoding, welches auch von der Suchfunktion auf der Google Maps Webseite verwendet wird (F_3.3). Somit kann nach einer Adresse gesucht werden und sie wird daraufhin in der Karte per Marker angezeigt. Per Klick auf den Marker werden weitere Informationen zu dem Objekt in einer „Sprechblase“ angezeigt. Diese kann auch Bilder und Links enthalten (F_3.3).

Zusätzlich wird auch ein Reverse Geocoding unterstützt. Das Geoportal von Google Maps (maps.google.com) bietet diese Funktion nicht direkt an. Sie ist aber über die Google Maps API verfügbar [71]. Diese Funktion ist vor allem nützlich, wenn ein Nutzer nicht die genaue Adresse kennt, er aber grob die Zielgegend oder ein Objekt auf der Karte zeigen kann. Die Navigation wird dadurch intuitiver und vor allem Navigations-Anwendungen können davon profitieren.

In das Kartenmaterial sind auch einige Points of Interest fest eingezeichnet, wie zum Beispiel die Straßenbahnhaltestellen in Dresden. Als Reaktion auf einen Klick auf die Haltestelle werden dort zur Zeit nur die dort verkehrenden Bahnlinien und ein Link zu den Dresdener Verkehrsbetrieben (DVB) angezeigt. Die Bushaltestellen werden gar nicht angezeigt. Diese Situation führte zu der Idee, die Haltestellen direkt mit dem Webservice der DVB bzw. VVO zu verknüpfen um den jeweiligen aktuellen Abfahrtsmonitor einer ausgewählten Haltestelle abzurufen. Diese Idee wurde zum Grundgedanken und Beispielszenario dieser Diplomarbeit, da sie sich auch auf zahlreiche andere Aspekte der Verwendung von Webservices in Kartenmaterial erweitern lässt. Hinzu kam der Anspruch, dass das Konzept flexibel und dynamisch bezüglich der Einbindung verschiedener Webservices sein sollte und dies alles

auf einem mobilen Endgerät verfügbar sein soll. Die Möglichkeiten zur Darstellung zusätzlicher Informationen in Google Maps Karten werden im Kapitel 4.2 detailliert besprochen.

Die Hauptnachteile der Nutzung von Google Maps sind seine Abhängigkeit von den Anbietern des Kartenmaterials und die damit einhergehenden Probleme bzgl. Lizenzen und Aktualität (F_3.5, F_3.6). Beides kann vom Nutzer nicht beeinflusst werden. Unter diesen Gesichtspunkten ist OpenStreetMap eine interessante Alternative und soll im nun folgenden Abschnitt untersucht werden.

4.1.2 OpenStreetMap

OpenStreetMap (OSM) ist ein freier MapProvider auf Basis von Open Source Software [72]. Wie eingangs erwähnt, wird das Kartenmaterial nach dem Wikipedia-Prinzip von den Nutzern zusammengetragen.

Die Informationen für die Karten, wie zum Beispiel Straßenverläufe, werden durch freiwillige Helfer mit GPS Empfängern getrackt. Die so aufgezeichneten Koordinatenverläufe werden dann als Vektordaten zusammen mit semantischen Informationen wie dem Straßennamen mittels Editier-Software in die Datenbank von OpenStreetMap via OSM-Protocol eingespeist. Hierbei werden die eingefügten Elemente nach den Klassifikationsvorgaben von OSM in Gruppen eingeteilt. So muss eine Straße beispielsweise entsprechend als Bundesstraße typisiert werden.

Da sich immer mehr Nutzer bei OSM einbringen, wird das Kartenmaterial stetig genauer und umfangreicher. Durch die Verteilung der Nutzer ist es allerdings häufig so, dass ländliche Regionen in OSM noch nicht sonderlich erschlossen sind.

Neuerdings ist für die Android-Plattform das Programm Android OSM contributor verfügbar, das den Vorgang des Sammeln von Geoinformationen stark vereinfacht und auch den Upload in die OSM-Datenbank direkt vornimmt [73]. Für Desktop-Systeme unter Windows und Linux sind weitere Editoren wie Online-Editor Potlatch oder die Offline Editoren JOSM und Merkaartor bekannt, die das Editieren und Erweitern des OSM Kartenmaterials am heimischen Computer vereinfachen [74, 75, 76].

Da das Material unter der Creative Commons Attribution-ShareAlike 2.0 Lizenz gesammelt und eingestellt wird, kann es frei in der eigenen Webseite oder in eigenen Programmen verwendet werden, ohne hohe Gebühren befürchten zu müssen (F_3.5) [77].

Wie bei Google Maps gibt es ein Geoportal, das die grundlegenden Funktionen wie Pan und Zoom anbietet (F_3.3). Die Seite selbst enthält im Gegensatz zu

Google Maps keine weitere Funktionalität, allerdings kann dank der freien API ein eigenes Geoportal entwickelt, mit Funktionalität angereichert und auf dem Kartenmaterial von OSM angezeigt werden. Beispiele hierfür sind: öpnvkarte.de, gazetteer.openstreetmap.org/namefinder/ und vor allem openrouteservice.org. Öpnvkarte.de blendet den Linienverlauf des öffentlichen Nahverkehrs in vielen deutschen Städten ein. Namefinder ist eine Seite, die Geocoding verwendet, um Orte in der Karte von OSM mittels Marker darzustellen. Die Datenbank scheint allerdings noch nicht sonderlich umfangreich zu sein. Openrouteservice ist ein Webportal, welches das Erstellen von Routenplänen ermöglicht. Außerdem sind noch weitere Portale vorhanden, die auf Basis der Daten von OSM Spezialkarten zum Wandern, Reiten oder Skifahren anbieten. Damit ist OSM vielseitiger als seine kommerziellen Konkurrenten.

Die OSM-gestützte Navigations-Anwendung für Android-basierte Geräte AndNav 2 ist ein gutes Beispiel für die native Nutzung und Einbindung von OSM-Karten in eigene Anwendungen [78]. Es handelt sich hierbei um ein mobiles Online-Navigationssystem. Zur Berechnung der Routen werden die Dienste von openrouteservice.org in Anspruch genommen [79].

Der Hauptvorteil von OSM ist auch gleichzeitig sein Hauptnachteil. Da alle Daten von Nutzern gesammelt werden, muss sich zum Teil auf die Zuverlässigkeit weniger Personen verlassen werden. Zwar findet nach dem Wikipedia-Prinzip eine Überwachung und Kontrolle durch die Community und zum Teil auch durch professionelle Kartographen statt, jedoch kann man sich, wie auch bei Wikipedia, nicht in jedem Punkt sicher sein, dass die Daten korrekt sind. Auch für Sabotage ist solch ein Konzept anfällig.

Zudem sind aufgrund der Abhängigkeit von den Eingaben der Nutzer die Ballungsgebiete stets detaillierter und zuverlässiger kartographiert als ländliche Gebiete. Es ist zwar bei Google Maps ein ähnlicher Effekt zu sehen, jedoch ist es dort bei weitem nicht so drastisch. Durch die einfache Erweiterbarkeit und einer steigenden Anzahl freiwilliger Mitarbeiter (mittlerweile mehr als 130.000) ist es aber mittelfristig zu erwarten, dass sich diese Situation verbessern wird [80].

4.1.3 Eignung der vorgestellten MapProvider für Android

In diesem Kapitel soll die Eignung der MapProvider Google Maps und OpenStreetMap bezüglich des programmatischen Zugriffs auf das Kartenmaterial in Android-Anwendungen untersucht werden. Serverseitige Ansätze werden im Kapitel 4.2 separat betrachtet.

Die Eignung wird an folgenden Punkten festgemacht:

1. Die freie Verfügbarkeit einer Bibliothek, die den Zugriff auf das Kartenmaterial und dessen Darstellung auf dem Display erlaubt.
2. Es müssen Kartenbetrachtungsfunktionen wie Zoom und Pan verfügbar sein.
3. Es muss eine Möglichkeit zur Darstellung von Objekten in der Karte vorhanden sein. Hierzu wird vor allem eine Projektionsfunktion benötigt, die die Umwandlung zwischen Displaykoordinaten und Geokoordinaten ermöglicht.

Da Android selbst schon eine Maps-Anwendung mitbringt, die solche Funktionen anbietet, ist anzunehmen, dass diese Funktionen auch als Bibliothek anderen Anwendungen zur Verfügung steht.

Bei OpenStreetMap ist dies nicht so selbstverständlich. Hier ist man auf die Arbeit einiger OSM-Enthusiasten mit Interesse für Android angewiesen. Größte Hoffnung diesbezüglich geht von der Entwicklung der Navigationsanwendung AndNav 2 aus.

Selbstverständlich wäre es auch möglich, eine eigene Bibliothek für den Zugriff und die Einbindung in Android für OSM oder auch andere MapProvider zu implementieren. Auf diese Möglichkeit wird verzichtet, da die Aufgabenstellung eine dynamische Einbindung vorhandener Dienste (Webservices und Kartendienste) in den Vordergrund stellt.

Die Auswahl für die prototypische Entwicklung wird je nach Eignung und den vorhandenen Möglichkeiten entschieden.

4.1.3.1 Eignung von Google Maps

Um mit einer Android-Anwendung auf das Kartenmaterial von Google Maps programmatisch zugreifen zu können, sind zwei Dinge erforderlich.

Zunächst muss ein Maps API Key von Google beantragt werden, was aber schnell und einfach möglich ist. Dieser Code wird automatisch generiert, indem der MD5-Hash des Zertifikats eingesendet wird, mit dem man die Android-Anwendung signieren wird. Passend zu diesem MD5 Hash wird dann der Maps API Key generiert und ausgegeben. Dieser muss in der Anwendung verwendet werden und ist nur für Anwendungen gültig, die mit dem zugehörigen Zertifikat signiert wurden.

Zudem ist eine Bibliothek erforderlich, welche den Zugriff auf die Google APIs aus der Android-Anwendung heraus ermöglicht (vgl. Abbildung 4.1). Dieses Google APIs Addon wird zur Zeit im Android SDK mitgeliefert, wird aber als selbständig betrachtet und enthält die Android Google Maps API. Eine Anwendung, welche die Google-API verwenden möchte, muss als Minimalvoraussetzung ein entsprechendes API-Level angeben.

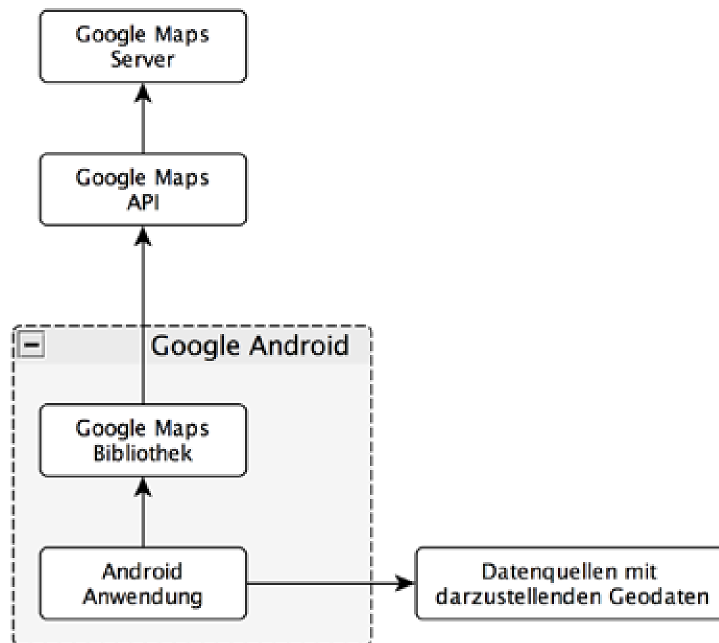


Abbildung 4.1: Schematische Darstellung der Android Google Maps API

Zur Prüfung der tatsächlichen Eignung für den vorgesehenen Einsatz von Google Maps und auch später von OpenStreetMap kommt man nicht umhin, einige Details der Anwendungsentwicklung unter Android zu untersuchen. Eine Beschreibung der Implementierungsdetails ist in Anhang A2 nachzulesen. Eine allgemeine, umfassende Einführung in die Softwareentwicklung für Android findet sich in [81] und in [82] und soll nicht Teil dieser Arbeit sein.

Die Untersuchung in Anhang A2 zeigt, dass die Android Google Maps API tatsächlich alle Anforderungen erfüllt, um in dieser Arbeit verwendet zu werden. Da die Verwendung von Google Maps Kartenmaterial aber auch Nachteile bezüglich Aktualität und Lizenzen hat, soll auch eine Alternative betrachtet werden. Diese ist OpenStreetMap und wird im nachfolgenden Kapitel besprochen.

4.1.3.2 Eignung von OpenStreetMap

Für den MapProvider OpenStreetMap gibt es ein Projekt namens OSMdroid, welches OSM Kartenmaterial für Android programmatisch verfügbar macht [83]. Der Autor Nicloas Gramlich entwickelte im Rahmen seines Hauptprojektes AndNav2 einige Klassen, die Androids eigene MapView vollständig ersetzen soll [78]. OSMdroid ist Teil von AndNav 2 und steht im Gegensatz zu AndNav 2 selbst unter GPLv3 frei und quelloffen zur Verfügung.

Laut dem Entwickler unterstützt sein Code Zoomen, Scrollen (Pan) und

Transformation zwischen Geokoordinaten und Bildschirmkoordinaten [84]. Ebenso wird das Overlay-Konzept unterstützt, um Objekte und Routen in der Karte darstellen und mit ihnen interagieren zu können (F_3.3).

OSMdroid kommuniziert direkt mit der OSM API (vgl. Abbildung 4.2), verwendet dabei aber nicht die Features des OpenStreetMap Binary Protocol, welches für den Einsatz auf mobilen Endgeräten vorgesehen ist. Dies könnte daran liegen, dass dieses Protokoll noch nicht ausreichend weit entwickelt ist oder die Verwendung wesentlich komplizierter ist als die normale OSM API.

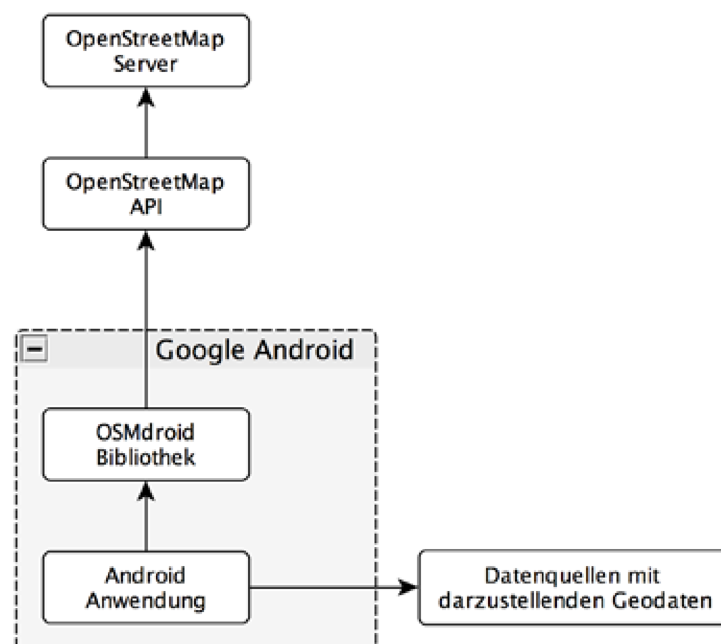


Abbildung 4.2: Schematische Darstellung der Android OpenStreetMap API

Wenn dieses Protokoll seine Ziele erreicht, wäre ein spürbarer Geschwindigkeitszuwachs von OSMdroid zu erwarten.

Generell ist der Aufbau sehr an den Aufbau von Androids Google API Bibliothek angelehnt, weshalb hier nicht weiter auf die Implementierung eingegangen werden soll, um Redundanzen zu vermeiden (vgl. Anhang A2). Über den Funktionsumfang von Googles Bibliothek hinaus gibt es noch Unterstützung für eine Minimap, eine kleine Karte in der Karte, die eine Zuordnung des aktuellen Kartenauschnitts zu einem Kartenausschnitt kleineren Maßstabes vornimmt, um Übersicht zur Position des gerade angezeigten Ausschnitts zu schaffen.

Interessant ist noch das Caching der bereits geladenen Kartenauschnitte in Form von Tiles-Bilddateien auf der SD-Karte des Android-Gerätes. Es wird angestrebt, dass

sich die Anwendungen, die OSMdroid verwenden, den sogenannten Tile-Cache mit AndNav 2 teilen können. Dadurch kann Datentransfervolumen gespart werden und im günstigen Fall auch Unterbrechungen der Netzwerksverbindung kompensiert werden. Dies wird darüber hinaus noch durch eine Tile-Downloader-Software unterstützt, die es dem Nutzer ermöglicht, bestimmtes Kartenmaterial am Desktop-Rechner herunterzuladen und auf die SD-Karte des Android-Gerätes zu kopieren.

Die OSMdroid Implementierung kann allerdings noch nicht als ausgereift betrachtet werden. Neben kleineren Mankos wurden auch im Rahmen dieser Untersuchung einige Probleme entdeckt, die das Beispielprogramm zum Absturz brachten. Die Probleme wurden gemeldet, es bleibt allerdings abzuwarten, ob dieses Projekt tatsächlich noch aktiv betreut wird. Die letzten nennenswerten Änderungen hat OSMdroid zuletzt im April 2009 erfahren.

Da das Projekt zwar an die Architektur der Google API Bibliothek angelehnt ist, aber trotzdem eine eigene Namenskonvention verfolgt, sind die beiden Bibliotheken nicht ohne Änderungen untereinander austauschbar.

Eine Einschränkung ist die mangelnde Unterstützung von Geocoding bzw. Reverse Geocoding. Diese Funktionen sind aber auch nicht Teil der Google Maps Bibliothek selbst, sondern Teil des Android-Frameworks. Unglücklicherweise verbieten die Bestimmungen von Google das Verwenden ihrer Geocodingdienste in anderem Kartenmaterial als Google Maps [85]. An die Stelle des Google-eigenenen Geocodings könnten freie Alternativen treten oder kommerzielle Alternativen mit weniger Restriktionen wie z.B. Dienste von Yahoo. Umgekehrt gelten bei der Verwendung des OpenStreetMap Kartenmaterials in Kombination mit anderen OSM-fremden Diensten keine Einschränkungen [86].

4.1.4 Schlussfolgerungen

Anhand der angeführten Einschränkungen der bislang einzigen Android-Bibliothek für OSM wird deutlich, dass sich OpenStreetMap im Vergleich zur Google Maps zum aktuellen Zeitpunkt weniger gut für diese Arbeit eignet. Es bleibt abzuwarten, ob sich dies in absehbarer Zeit ändern wird. Um unvorhergesehenen Problemen aus dem Weg zu gehen, wird in dieser Arbeit vorerst die Google API Bibliothek bevorzugt, welche bei Android bereits mitgeliefert wird.

Darüber hinaus ist zu erwarten, dass Google die Einbindung ihres Kartendienstes in ihre eigene mobile Plattform optimiert hat, wodurch das Performancekriterium (F_3.4, NF_6) ebenfalls für den Einsatz von Google Maps spricht.

4.2 Darstellung von Geodaten in WebGIS-Karten

In diesem Kapitel soll untersucht werden, welche vorhandenen serverseitigen Möglichkeiten es gibt, Geodaten wie MapItems, aber auch Routen oder Flächen in WebGIS Karten anzuzeigen bzw. hervorzuheben. Viele der untersuchten Ansätze sind aus dem Desktopbereich bekannt und werden dort in Webanwendungen genutzt. Zwar sind Webanwendungen nur eingeschränkt für mobile Plattformen geeignet, jedoch könnten bei der Untersuchung der vorhandenen Ansätze wertvolle Erkenntnisse für diese Arbeit zu Tage kommen. So können die vorhandenen Weblösungen über die Möglichkeiten der zugrunde liegenden Dienste Aufschluss geben und Ideen liefern, welche Funktionen in diesem Kontext auch in einer nativen Anwendung auf einem mobilen Endgerät sinnvoll oder nützlich wären.

Die Ursache der aktuellen Entwicklung bezüglich der serverseitigen WebGIS-Ansätze liegt auf der Hand: wenn das zugrunde liegende WebGIS eine Webanwendung ist, dann lässt diese sich unter Verwendung weiterer Webtechnologien relativ leicht zu einer neuen Webanwendung mit erweitertem Funktionsumfang kombinieren. Diese Kombination, das Verweben von WebGIS mit zusätzlichen Webdiensten zu einer Webanwendung ist der Grundgedanke, der mit dem Begriff GeoMashUp bezeichnet wird.

Das traditionelle GeoMashUp ist eine eigenständige Webseite, die über eine Web-API meist mittels einer Scriptsprache wie JavaScript das Kartenmaterial eines WebGIS (MapProvider) dynamisch einbindet. Die Webseite hat außerdem Zugriff auf eigene kartenbezogene Inhalte (Geodaten), die sie wiederum mittels der Web-API des WebGIS in der Karte darstellt. Die Web-API ermöglicht häufig auch eine Interaktion mit den dargestellten Geodaten, um beispielsweise zusätzliche Informationen zu einem MapItem anzuzeigen. Bekannte Beispiele hierfür sind die im Kapitel 4.1 vorgestellten WebGIS Google Maps mit seiner Google Maps API und OpenStreetMap mit dem OpenStreetMap Protocol.

Ein neuerer Ansatz, der zur Zeit von Google verfolgt wird, ist es, die Funktionen der GeoMashUps in Google Maps flexibel zu integrieren. Hierbei fügt eine personalisierte Variante von Google Maps die Zusatzfunktionalität mittels einer speziellen Scriptsprache ein. Die Funktionen befinden also nicht mehr auf einer eigenen Webseite, sondern werden von Google gehostet. Dieser Ansatz gründet auf dem personalisierten Google Maps System namens My Maps.

In den folgenden Abschnitten werden repräsentativ die verschiedenen Möglichkeiten zur Darstellung von Geodaten in den Karten der zwei MapProvider Google Maps und OpenStreetMap untersucht.

4.2.1 Google Maps (Web) API

Die Google Maps-(Web)-API ermöglicht das Einbetten von Google Maps in eigene Webseiten mittels JavaScript. Der grundlegende Aufbau ist in Abbildung 4.3 zu sehen.

„Die API bietet einige Hilfsprogramme zur Bearbeitung von Karten (wie auf der Webseite http://maps.google.de) und zum Hinzufügen von Inhalten zur Karte über eine Vielzahl von Services, die es Ihnen ermöglichen, stabile Kartenanwendungen auf Ihrer Website zu erstellen.“ [87]. Die API kann kostenfrei genutzt werden, wenn die Webseite, die die Google Maps Karte einbettet, ebenfalls frei zugänglich ist.

Um die API verwenden zu können, muss man den Nutzungsbedingungen von Google zustimmen und einen Google Maps API Key für die eigene Webseite beantragen. Dem Webseitenentwickler, der Google Maps in seine Seite integrieren will, stehen verschiedene Funktionen der Google Maps API zur Verfügung. Die grundlegenden Funktionen wie Zoom und Pan stehen nach der Einbettung wie von maps.google.com gewohnt zur Verfügung.

Weitere Funktionen sind die Anzeige von Zusatzinfos zu MapItems durch Sprechblasen, Bedienelemente für den Zoom, Pan und Kartentypen (Satellitenbild, Straßenkarte, Hybrid). Außerdem gibt es noch Overlays und Services.

Overlays sind transparente Ebenen über der Karte, in denen beispielsweise

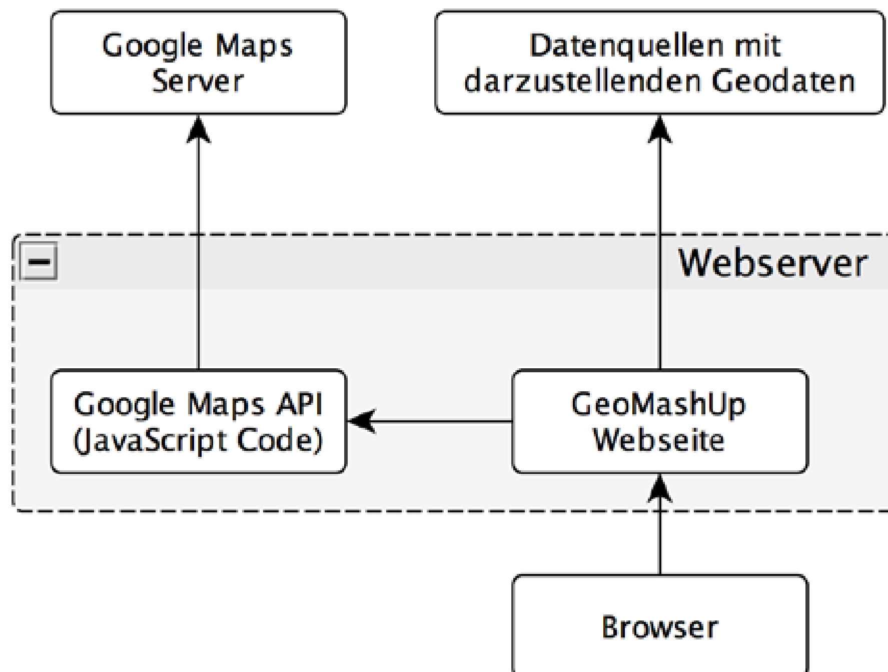


Abbildung 4.3: Aufbau GeoMashUp-Seite auf Basis von Google Maps

MapItems und Routen dargestellt werden können (F_3.3). Es gibt einige vorgegebene Standardoverlays, aber es sind auch eigene Overlays definierbar. Hierfür sei auf die Dokumentation der Google Maps API Overlays verwiesen [88]. Mittels Overlays können Objekte mit Koordinaten in der Karte dargestellt werden. Es gibt verschiedene Typen von Overlays: Markierungen, die die Position von MapItems anzeigen, Polylinien, die Verbindungen zwischen Punkten darstellen und Polygone für Flächen mit unregelmäßiger Form. Für Flächen mit rechteckiger Form können Overlays vom Typ Boden-Overlay oder Kachel-Overlay verwendet werden.

Services sind komplexere Funktionen wie die in Google Maps integrierte Routenplanung. Google Maps selbst wird von den Google-Entwicklern kontinuierlich erweitert. Hierbei hinzugefügte Funktionen werden zunächst auf maps.google.com getestet und nach einiger Zeit in der Google Maps API verfügbar gemacht.

So kann sich nach dem Baukastenprinzip an den fertigen Google Maps Funktionen bedient werden, um diese nahtlos in die eigene Karten-Webanwendung zu integrieren. Interessante Beispiele hierfür sind die Routenplanung und der Traffic Overlay Service, der aktuelle Staus und Verkehrsstörungen in das Straßennetz einzeichnet. Dieser Dienst ist zur Zeit nur in den USA verfügbar. Generell zeigen diese Funktionen, dass es durchaus möglich ist, Routen oder (Bahn-)Linienverläufe in Google Maps darzustellen.

Der Service, der Geodaten anzeigt, die durch Daten im Format Keyhole Markup Language (KML) oder GeoRSS-Feeds beschrieben werden, bietet eine einfache Schnittstelle zur Einbindung einer großen Anzahl an Geodaten in die Karte oder zur Darstellung von Routen durch Koordinatenlisten [89, 90].

Ein allgemeiner Nachteil, der so auch auf andere Google-Lösungen zutrifft, ist die starke Abhängikeit von Google. Dies betrifft sowohl die Verfügbarkeit, Aktualität und Genauigkeit des Kartenmaterials, wie auch die freie Verfügbarkeit der Google Maps API.

4.2.1.1 Eignung der Google Maps API

Greift man mit einem Android-Gerät auf maps.google.com mittels des Android-Browsers zu, wird schnell klar, warum sich die Android-Plattformentwickler zu der separaten Maps-Applikation Android Maps entschieden haben. Der Aufbau der Karten ist extrem langsam und selbst Standardinteraktionen wie Pan und Zoom funktionieren nicht wie gewohnt.

Das intuitive Pan wird schon auf Browserebene abgefangen, um ein Scrollen in der Webseite selbst zu ermöglichen. Dadurch wird allerdings verhindert, dass der Kartenausschnitt wie gewünscht verschoben wird. Beim Zoom sieht es ähnlich aus, außer dass hier noch auf das Zoom-Bedienelement in der Google Maps

Karte zurückgegriffen werden kann, was allerdings wenig komfortabel ist. Das Nutzen der Google Maps Webseite im Android-Browser ist damit aus praktischen Gesichtspunkten auszuschließen.

Unter denselben Problemen leiden auch Webseiten, die Google Maps bei sich einbetten, indem sie die Google Maps API verwenden. Zwar ließen sich die Webseiten für die Verwendung im Android-Browser optimieren, zum Beispiel durch Android-geeignete Bedienelemente, dennoch lässt sich das langsame Kartenladen damit nicht beseitigen. Zudem wäre der zentrale Vorteil von Webanwendungen, nämlich die Plattformunabhängigkeit, in gewissem Sinne untergraben.

Ein Hauptargument gegen die Nutzung einer Google Maps basierten Webseite im Rahmen dieser Arbeit ist die sehr langsame Aufbaugeschwindigkeit. Die Betrachtung der Google Maps API ist in diesem Rahmen jedoch nicht vergebens, denn sie verrät viel über das Potential dieser Dienste, welche in diesem Umfang eventuell von der nativen Android-Implementierung noch gar nicht völlig ausgeschöpft werden. Es ist somit denkbar, dass die „Google Maps API Bibliothek für Android“ seitens Google noch in ihrem Funktionsumfang erweitert wird und die angeführten, interessanten Möglichkeiten somit auch für Android ausreichend performant zur Verfügung stehen werden. Details zur Verwendung der Google Maps API Bibliothek für Android sind dem Anhang A2 zu entnehmen.

4.2.2 Google Maps - My Maps

My Maps ist Googles Ansatz, auch Nutzern ohne Programmierkenntnisse die Möglichkeit zu geben, sich Google Maps zu personalisieren, indem benutzerdefinierte Karten angelegt werden. My Maps ermöglicht es somit, eigene Karten auf Basis von Google Maps zu erstellen. Es können Ortsmarken (Tagging), Linien und Formen angelegt werden (F_3.3). Tagging beschreibt das Anreichern der Karten mittels Setzen von Markern in der Karte und das Hinzufügen von Informationen über den jeweiligen Ort bzw. das Objekt, welches durch diesen Marker dargestellt wird. Diese Geoobjekte können mit textuellen Informationen beschrieben und mit Fotos oder Videos angereichert werden. Außerdem können die eigenen Karten für andere angemeldetete Nutzer oder für alle Nutzer freigegeben werden.

Seitens Google werden auch einige My Maps veröffentlicht, die die Möglichkeiten dieses Ansatzes demonstrieren sollen. Zu beachten ist, dass dies kein programmatisch nutzbarer Ansatz ist. Es gibt keine offene API oder ähnliche Möglichkeiten, um mit einer eigenen Anwendung My Maps nutzen zu können. Die angebotenen Wege zur Informationsdarstellung sind relativ primitiv und lassen wenig Gestaltungsspielraum.

Die allgemeinen Nachteile decken sich verständlicherweise mit den Nachteilen eines GeoMashUps auf Basis der Google Maps API. Gleichermaßen erbt dieser

Ansatz auch die Nachteile beim Einsatz dieser unter Android. Diesbezüglich hat Google jedoch bereits reagiert und eine Android-Anwendung zur Anzeige und Erzeugung von personalisierten My Maps Karten Namens My Maps Editor for Android entwickelt.

Da die gegebenen Möglichkeiten dieses Ansatzes relativ eingeschränkt sind und er zu diesem Zeitpunkt noch nicht frei programmatisch nutzbar ist (wobei der My Maps Editor die Machbarkeit bereits zeigt), sind My Maps für diese Arbeit zunächst nicht weiter interessant. Weitaus interessanter, da im Funktionsumfang deutlich flexibler, könnte ein anderer Ansatz von Google - die Mapplets - sein, der nachfolgend untersucht wird.

4.2.3 Google Mapplets

„Mapplets sind kleine Anwendungen, die innerhalb von Google Maps ausgeführt werden“ [91]. Es ist Googles Ansatz, um neue, komplexere Funktionalität in das Google Maps WebGIS zu bringen. Der Aufbau und Funktionsumfang ähnelt stark dem der Google Maps API. Der schematische Aufbau ist in Abbildung 4.4 ersichtlich.

Der Hauptunterschied zur Google Maps API ist, dass der Code nicht mehr in eine eigene Webseite eingebunden wird, sondern von Google gehostet wird. Ein Mapplet selbst ist ein spezieller Typ eines Google Gadgets und basiert damit auf der Google Gadget-API [92]. Bei diesen handelt es sich um einfache HTML- oder JavaScript-Anwendungen. Ein Mapplet ist ein XML-Wrapper, der für solche Anwendung einen Container bildet.

Der Code kann in Form einer XML Datei auf einem eigenen Server abgelegt werden und wird bei Verwendung in den Mapplet-Cache geladen. Die Mapplets werden dadurch anderen Nutzern öffentlich zugänglich. Alternativ kann man auch beantragen, dass eigene Mapplets mit in Googles Mappletverzeichnis aufgenommen werden. Diese können dann im My Maps Bereich von Google Maps in die Liste der eigenen Karten aufgenommen und wahlweise eingeblendet werden.

Die Spanne der zur Zeit verfügbaren Mapplets reicht von einfachen Kartentools wie Streckenvermessung auf der Karte bis zu komplexeren, wie die transparente Überlagerung von OpenStreetMap Kartenmaterial über den Google Maps Karten.

Dadurch, dass der Code von Google gehostet oder zumindest gecacht wird, hat Google auch die Hand darauf. Dies betrifft nicht nur die Kenntnis des Codes, sondern bringt auch gewisse Einschränkungen im Vergleich zur Google Maps API mit sich.

Ein weiterer Unterschied zur Google Maps API ist die asynchrone Kommunikation

zwischen Mapplet und dem Google Maps Dienst. Einen Rückgabewert von der Karte erhalten diese Methoden mittels Callback Funktion vom aufgerufenen Dienst zurück. Außerdem ist noch zu erwähnen, dass das Nachladen von Daten durch das Mapplet über den Proxyserver von Google erfolgen muss.

Hauptvorteil des Einsatzes von Mapplets ist die Ersparnis einer eigenen GeoMashUp-Webseite, indem der die Google Maps API nutzende Teil in Form eines Mapplets von Google gehostet wird.

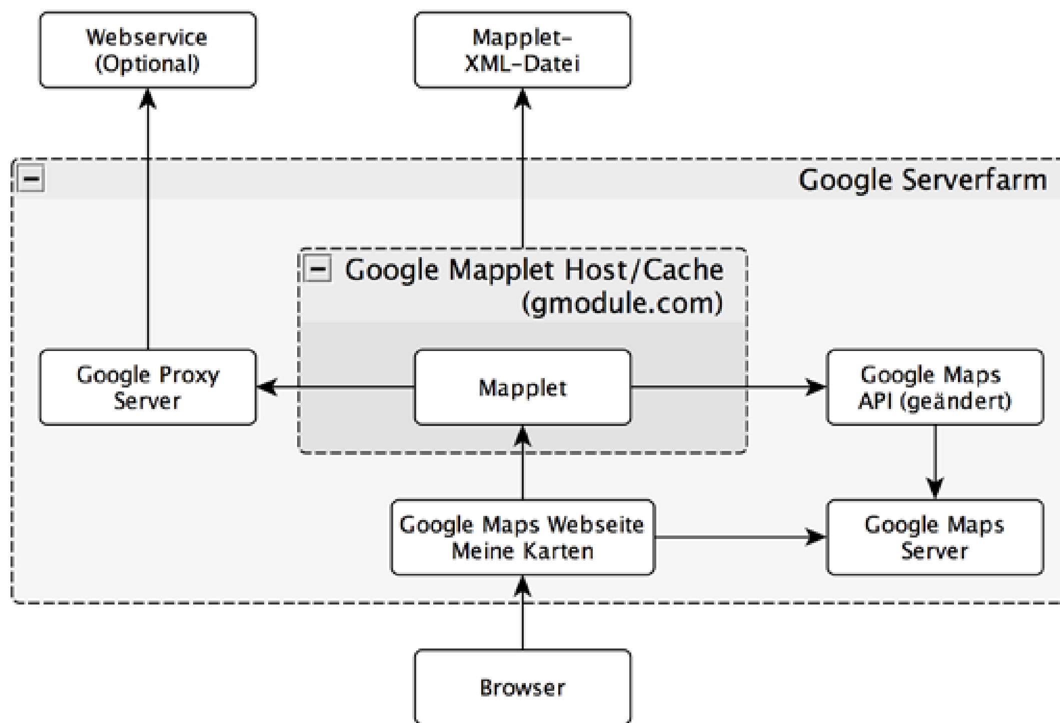


Abbildung 4.4: Mapplet - Architektur

Es ist eine gute Performance zu erwarten, da die Mapplets auf Servern der sehr potenten Google Server Farm liegen. Es ist ebenfalls denkbar, dass Verzögerungen durch die Kommunikation zwischen Mapplet-Host und Google Maps Dienst deutlich geringer ausfallen, als bei einem klassischen GeoMashUp.

Nachteil an der Mapplet-Lösung ist die Abhängigkeit von Google, diesmal allerdings im doppelten Maße. Zum einen besteht Abhängigkeit vom Kartendienst, der von den Mapplets verwendeten Google Maps API und der Grundlage der Mapplets in Form der Gadgets-API. Zum anderen besteht Abhängigkeit vom Hosting des Mapplets seitens Googles. Damit geht ein erhöhter Einfluss Googles auf die nutzbaren Möglichkeiten einher. Vielen kommerziellen Entwicklern wird auch nicht gefallen, dass sie ihren Quellcode Google gegenüber offenlegen müssen.

4.2.3.1 Eignung für Android

Die Anzeige eines Mapplets im Android-Browser hat logischerweise dieselben Nachteile, wie die Anzeige einer GeoMashUp Webseite oder von Google Maps selbst. Die Interaktions- und Darstellungsmöglichkeiten sind genauso beschränkt bzw. unkomfortabel. Da Nutzer von mobilen Endgeräten hierbei ungern Abstriche machen, ist es wohl nur noch eine Frage der Zeit, bis Google hier nachlegt.

Dies kann eventuell zukünftig durch den im Kapitel 4.2.2 beschriebenen My Maps Editor for Android umgangen werden. Da auch auf der Google Maps Webseite die Mapplets unter dem Reiter „Meine Karten" geführt werden, ist es denkbar, dass Mapplets auch mithilfe dieses Programms performant angezeigt werden können. Somit eignet es sich gleichermaßen wie der Ansatz, der bei My Maps verfolgt wird, hat jedoch nicht dessen Einschränkungen, da es nahezu alle Möglichkeiten bietet, die von der Google Maps API angeboten werden.

4.2.4 OpenStreetMap APIs

Um die Daten des MapProviders OpenStreetMap (OSM) serverseitig zu nutzen, gibt es mehrere Wege.

Es gibt das OpenStreetMap Protocol, ein auf HTTP basierender Protokollstack, welcher direkten Zugriff auf die rohen Geodaten der OpenStreetMap-Datenbanken ermöglicht. Die API ist in Form eines RESTful Webservice gehalten (siehe hierzu Anhang A1). Die API ermöglicht nicht nur den Download von Geodaten, sondern auch den Upload.

Neben dieser API gibt es noch die OSM Extended API, auch XAPI (gesprochen Zappy) genannt. Es ist eine read-only API, welche die Abfragemöglichkeiten der normalen OSM API erweitert.

Außerdem gibt es noch das OpenStreetMap Binary Protocol. Es ist eine API, die vor allem für den Einsatz auf mobilen Endgeräten gedacht ist. Sie soll ermöglichen, dass nur so viele Daten, wie wirklich zur Darstellung benötigt heruntergeladen werden. Außerdem soll durch binäre Datenformate das Datentransfervolumen reduziert werden, was sich vor allem bei schwacher Netzanbindung stark bemerkbar bei der Anzeigeperformance machen dürfte.

Will man das OSM-Kartenmaterial vergleichbar zur Google Maps API auf einer eigenen Webseite integrieren, verwendet man jedoch nicht diese APIs direkt, sondern die JavaScript-Bibliotheken OpenLayers oder CloudMade's Web Maps Lite [93, 94, 95]. Ähnlich einfach wie bei Google Maps kann mithilfe dieser Bibliotheken die OSM Karte in die eigene Webseite integriert werden (vgl. Abbildung 4.5). Außerdem lassen sich MapItems oder Routen in einer transparenten Ebene über

der Karte darstellen (F_3.3).

OpenLayers ist ein von dem Open Geospacial Consortium (OGC) (vgl. Kap. 2.2) aufgenommenes Projekt. Durch die Einhaltung der Standards der OGC bei OpenLayers, OpenStreetMap sowie Google Maps ist es möglich, mit OpenLayers nicht nur OpenStreetMap, sondern auch das Kartenmaterial anderer MapProvider, wie Google Maps, Yahoo Maps oder Virtual Earth nahtlos in eine eigene Webseite einzubinden. Man kann somit einfach zwischen den Karten der einzelnen MapProvider umschalten. Ebenfalls werden diverse OGC -standardisierte Formate und Dienste wie Web Feature Service und Web Map Service unterstützt, da OpenLayers die dazugehörigen Protokolle implementiert [96].

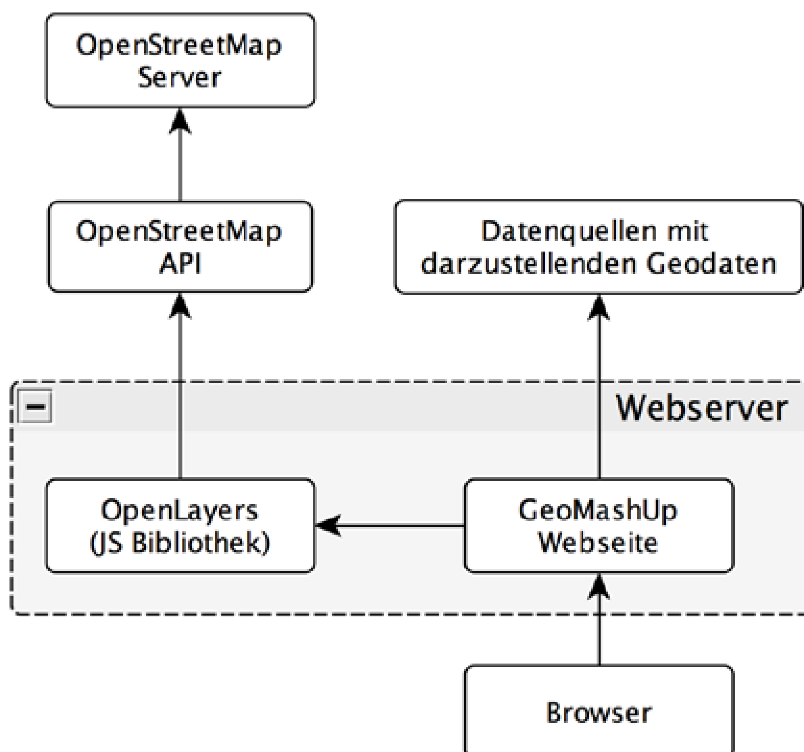


Abbildung 4.5: Aufbau GeoMashUp-Seite auf OSM-Basis

4.2.4.1 Eignung für Android

Mit OSM und OpenLayers sind beliebige Karten-Webanwendungen erstellbar. Was für den Desktopbereich noch wunderbar funktioniert, ist auf dem mobilen Endgerät zum Teil nicht gebrauchbar. Dies wird deutlich, wenn man mit dem Android-Browser die Webseite openstreetmap.org besucht. Die Seite kann nicht ordnungsgemäß dargestellt werden und die Interaktionsmöglichkeiten sind genauso eingeschränkt wie bei Google Maps. Lediglich das Zoomen mittels der Bedienelemente funktioniert, wobei das auch die einzige Funktion war, die bei Google Maps in Ordnung war.

Es bleibt abzuwarten, ob nachträgliche Verbesserungen am Android-Browser hier Abhilfe schaffen werden. Denkbar wäre auch, dass, speziell für die korrekte Darstellung im Android-Browser oder generell auf mobilen Endgeräten, mithilfe von OpenLayers und OSM eine Webanwendung erstellt wird. Immerhin sollten sich so die Engpässe bezüglich der Bedienbarkeit der Karte überwinden lassen.

Die Geschwindigkeit des Kartenaufbaus würde vermutlich weiter durch JavaScript-Interpretation im Android-Browser langsamer sein, als vergleichweise der Zugriff von einer nativen Android-Anwendung aus, welche die OSM API verwendet.

Die Darstellung zusätzlicher Informationen zu einem MapItem in Form einer „Sprechblase“, wie es bei OSM und Google Maps möglich ist, eignet sich nicht für mobile Endgeräte. Die Schrift ist hierbei schlichtweg zu klein. Zwar ermöglicht es der Android-Browser, in Webseiten hinein zu zoomen, aber dies wäre bei häufiger Verwendung sehr unkomfortabel und anstrengend. Bei einer nativen Anwendung könnte man diese Informationen in einer separaten Activity übersichtlich und gut lesbar darstellen.

Eine Bibliothek für diesen Ansatz ist OSMdroid, welche schon im Kapitel 4.1.3.2 beschrieben wurde. Mit dieser Bibliothek ist es möglich, aus einer Android-Anwendung heraus direkt über das OpenStreetMap-Protocol auf das Kartenmaterial von OSM zuzugreifen und somit die Nutzung wesentlich komfortabler zu machen.

4.2.5 Schlussfolgerungen

Es wurden einige Ansätze zur Darstellung von Geodaten in Kartenmaterial von WebGIS-Systemen vorgestellt. Da WebGIS Webanwendungen sind, war es naheliegend, dass sich die vorhandenen Ansätze ebenfalls auf die Verwendung in Webanwendungen konzentrieren werden. Im Desktop-Bereich funktioniert das auch ganz gut, was sich zur Zeit leider nicht auf den Bereich der mobilen Endgeräte übertragen lässt.

Die Anzeige der Webseiten mit eingebetteten WebGIS Karten bereitet dem Android-Browser ebensolche Probleme, wie die Interaktion mit der Karte (Pan, Zoom). Das übliche Paradigma zur Darstellung von Zusatzinformationen von MapItems - die „Sprechblase“ - eignet sich nicht zum Einsatz in Geräten mit kleinen Bildschirmen. Vor allem die Lesbarkeit leidet darunter, die Zoomfunktion des mobilen Browsers löst dieses Problem nur bedingt und auf eher unpraktische Weise.

Ein weiteres großes Problem der Karten-Webanwendungen bei mobilen Endgeräten ist die Performance. Die wenigen funktionierenden Interaktionsmöglichkeiten, wie das Zoomen, zeigen gleich, dass der Kartenaufbau im Vergleich zu einer nativen

Android-Anwendung zur Kartendarstellung deutlich langsamer ist und zwar so langsam, dass die Verwendbarkeit darunter stark leidet.

Daher hat es sich für den Zweck der Kartenbetrachtung und Interaktion auf Android, aber auch auf dem iPhone, durchgesetzt, native Anwendungen wie Android Maps oder iPhone Maps zu verwenden. Die von diesen Anwendungen verwendeten Bibliotheken stehen mindestens im Fall von Android dem Entwickler ebenfalls zur Verfügung. Es ist also möglich, im Rahmen einer nativen Anwendung performant auf das Kartenmaterial zuzugreifen und zusätzliche Funktionen, wie das Anzeigen von eigenen Objekten oder von Zusatzinformationen programmatisch zu integrieren (vgl. Kap. 4.1).

Aufgrund der genannten Nachteile der WebGIS-basierten Webanwendung und der dem gegenüberstehenden Vorzüge der nativen Einbindung ist letztere für diese Arbeit zu bevorzugen. Der direkte Zugriff auf die API mittels nativer Bibliothek ist wesentlich performanter, als der Umweg über Browser. Darüber hinaus ermöglicht dieser Weg den Einsatz der intuitiven und praktischen Plattform- und Geräte-spezifischen Interaktionsparadigmen, die beispielsweise durch TouchScreen und Bewegungssensoren ermöglicht werden.

Die zur Karteneinbindung notwendigen Bibliotheken stehen für die MapProvider Google Maps und OpenStreetMap frei zur Verfügung. Die Google Maps API ist allerdings im Vergleich wesentlich ausgereifter und auch seitens Google für den Einsatz auf der Android-Plattform optimiert worden. Für den zu entwickelnden Prototypen bietet sich also eher die Verwendung von Google Maps als MapProvider an. Betrachtet man diese Arbeit über den Forschungskontext hinaus, könnten allein Lizenz- und urheberrechtliche Probleme einen Wechsel zu OpenStreetMap verursachen, wie dies auch in dem AndNav-Projekt geschehen ist [97].

4.3 Vorhandene Anwendungen

In diesem Kapitel soll betrachtet werden, welche vorhandenen Ansätze im Bereich des Beispielszenarios „Mobiler ÖPNV-Informationsdienst“ bereits existieren und welche Eigenschaften und unterschiedliche Merkmale sie haben.

Fast alle vorhandenen und so auch die hier vorgestellten Ansätze vereint, dass sie spezifisch auf die jeweilige Aufgabenstellung zugeschnitten sind und ihre Konzepte kaum Potential zur flexiblen, nachträglichen Erweiterung haben. Es ist zwar von außen schwierig zu beurteilen, inwiefern die jeweilige Softwarearchitektur Spielraum für Erweiterungen auf statischer Ebene bietet, jedoch ist festzustellen, dass keine dynamische Erweiterbarkeit vorgesehen ist.

Häufig entstanden die vorgestellen Möglichkeiten im Auftrag der verschiedenen Dienstanbieter und verwirklichen daher auch nur deren Interessen. Obwohl die verschiedenen Programme sich in ihren Funktionen überlappen und sich daher an vielen Punkten ähnlich sind, sind sie unabhängig voneinander entworfen worden und können auch nicht miteinander kooperieren. Es findet daher keine Wiederverwendung von Softwarekomponenten statt und auch die Interoperabilität untereinander ist nicht vorgesehen.

Es wurde also offensichtlich kein allgemeiner, generischer Ansatz verfolgt, um all diese Dienste in einer zentralen Anwendung zu vereinen. Dadurch entstehen die Redundanzen, welche sich in Speicherplatzverbrauch und Ressourcenhunger bei gleichzeitiger Verwendung äußern.

Zumindest für die hier vorgestellten Anwendungen ist es jeweils kein Problem, dass sie nicht interoperabel sind, da sie als Alternativen zueinander zu betrachten sind. Bei allen drei Anwendungen geht es um die Umsetzung eines mobilen Fahrgastinformationssystems für den Bereich der öffentlichen Verkehrsmittel. Der DB Railnavigator ist der für den Endnutzer kostenlose Ansatz der Deutschen Bahn. Dazu gesellen sich die beiden ebenfalls kostenlosen Anwendungen Fahrplan für Apple iPhone und Métro, welches für zahlreiche Plattformen (außer Android) verfügbar ist und auf den öffentlichen Personennahverkehr zugeschnitten ist.

Eine Gemeinsamkeit von DB Railnavigator und Fahrplan für Apple iPhone ist, dass sie Onlinedienste zur Beantwortung der Anfragen des Nutzers verwenden, wobei der DB Railnavigator einen ausgeprägten Offlinemodus hat. Inwieweit es sich bei DB Railnavigator um Nutzung von Webservices im engeren Sinne oder gar OGC -konformer Informationsdienste handelt, ist bei der äußerlichen Untersuchung nicht ermittelbar.

Vom Grundkonzept her sind alle drei vorwiegend textorientierte Anwendungen. Der Nutzer muss die Namen von Start- und Zieladresse, bzw. den Namen des Point of interest (POI) textuell eingeben. Zusätzlich ist meist die Anzeige der Route oder des Fußwegs zur Haltestelle bzw. Bahnhof in einer digitalen Karte visualisiert. Die Karte dient hier allerdings vorwiegend der grafischen Darstellung und ist weniger als direktes, intuitives Interaktionsmittel angedacht. Auch wenn Fahrplan hierbei eine gewisse Ausnahme bildet, muss doch bemerkt werden, dass auch da der Bedienfluss überwiegend textbasiert ist. Dieses Vorgehen ist sicherlich bei allen Anwendungen beabsichtigt, in der Hoffnung die Menge der zu übertragenden Daten gering zu halten.

Eine von Endnutzern häufig gewünschte und in den vorgestellten Anwendungen auch vorhandene Funktion ist das lokale Ablegen von abgefragten Verbindungen, um diese auch noch offline lesen zu können. Dies ist vor allem bei Bahnfahrten in Gegenden ohne mobile Internetversorgung relevant.

Durch die bereits erwähnte statische Architektur bei allen diesen Ansätzen lassen sich die Anwendungen nicht ohne hohen Aufwand (Codeänderung, Neukompilierung, Testen der Gesamtanwendung, Deployment) mit zusätzlichen Informationen (z.B. aus anderen Webservices) erweitern.

Die Synergieeffekte aus einem neuen Veranstaltungsinformationsdienst und der aus mehreren Aspekten wünschenswerten Nutzung der öffentlichen Verkehrsmittel sind folglich aus genannten Gründen nur durch eine höhere Entwicklungsdauer erreichbar.

Die folgenden Abschnitte sollen nun kurz einen Überblick über den aktuellen Stand der vorhandenen Anwendungen geben und vor allem den von ihnen erfüllten Funktionsumfang beschreiben.

4.3.1 DB Railnavigator

Der DB Railnavigator ist ein auf der Java Micro Edition (JavaME) basierendes Fahrgastinformationssystem der Deutschen Bahn AG. Dadurch ist es auf Handys und Smartphones mit JavaME-Unterstützung lauffähig. Für Blackberry existiert eine besondere, angepasste Version.

Das Programm bietet den folgenden Funktionsumfang:

- Fahrpläne speichern und jederzeit – auch ohne Internet-Verbindung – abrufen
- Aktuelle Ankunftszeiten inkl. Gleisinformationen sowie Pünktlichkeitsprognosen und Zwischenhalte zu einer Verbindung
- Tür zu Tür Planung Ihrer Reise – deutschlandweit, inkl. Umgebungskarten und detaillierter Fußwegbeschreibungen zu Start- und Zielorten
- Umgebungskarten und detaillierte Fußwegbeschreibungen zu Start- und Zielorten
- Zielrouting per GPS-Funktion Ihres Handys
- Handy-Ticket und Sitzplatzreservierung
- viele weitere Services wie Lagepläne ausgewählter Bahnhöfe, „optischer Ticker“ für noch zu erreichende Verbindungen, Export einer Verbindung in den Kalender Ihres Handys (abhängig vom Endgerät).

(Quelle: http://www.bahn.de/p/view/buchung/mobil/railnavigator.shtml)

Das besondere ist, dass die Fahrplandaten auch über einen angeschlossenen Rechner zu Hause auf das Gerät geladen werden können, um so später kostenlos und unabhängig von mobiler Internetabdeckung auf die Daten zugreifen zu können.

Der Ansatz, solch eine Software möglichst herstellerunabhängig auf vielen verschiedenen Plattformen und Endgeräten lauffähig zu machen, ist generell

lobenswert, geht aber auch mit den üblichen Einschränkungen und Kompromissen einher. Manche Funktionen sind daher bei dieser Anwendung nicht auf allen unterstützten Endgeräten verfügbar.

Alternativ zu dem JavaME-Programm bietet die Deutsche Bahn AG auf mobile. bahn.de ein Serviceportal für den Zugriff per Handy-Browser an. Es ist eine stark vereinfachte Webseite, die für kleine Bildschirme und geringe Bandbreite angepasst wurde. Sie ermöglicht eine einfache, textuelle Verbindungsauskunft vom Handy über das Internet. Manche Smartphones, genauer ihre Browser, haben allerdings Probleme, sogar diese einfache Webseite korrekt darzustellen.

4.3.2 Métro

Métro ist eine Freeware, die Informationen zu Nahverkehrsnetzen zahlreicher Großstädte der Welt auf das Handy, das Smartphone oder den PDA holt [98]. Ihr großer Vorteil ist die Menge der unterstützten Plattformen, genauer die Menge der für diese Plattformen geeigneten Versionen von Métro.

Die unterstützten Plattformen sind Palm, PocketPC, MS Smartphone, Symbian UIQ, Symbian S60, S80, S90, BlackBerry und iPhone. Die Android-Plattform wird noch nicht unterstützt.

Métro ist vorwiegend ein Offline-Programm. Die einzelnen Streckennetze der verschiedenen Städte werden am Computer heruntergeladen und per Synchronisierungssoftware oder Dateitransfer auf das mobile Endgerät geladen. Lediglich eine Online-Update-Funktion für diese Streckennetz-Daten existiert. Für die Auskunft wird also kein Webservice verwendet, was die Anwendung im technologischen Vergleich für diese Arbeit eher uninteressant macht.

Es wird schnell deutlich, dass es sich um einen sehr statischen Ansatz handelt. Veränderungen der jeweiligen Streckennetze müssen von den Entwicklern von Métro in den entsprechenden Datensatz eingearbeitet werden und die Nutzer müssen die aktualisierte Version nachladen. Eine Integration zusätzlicher Informationen aus anderen Quellen zum Beispiel zur Anzeige in der Karte bei der iPhone-Version von Métro ist nicht möglich. Wie auch DB Railnavigator ist Métro textgesteuert und damit ebenfalls weniger intuitiv im Vergleich zu Anwendungen mit direkter Karteninteraktion als zentralem Interaktionsparadigma.

4.3.3 Fahrplan für Apple iPhone

Fahrplan ist eine kostenlose iPhone-Anwendung von Frank Vercruesse. Es beinhaltet einen Abfahrtsmonitor, eine Verbindungsauskunft und die Anzeige der hierfür relevanten Informationen in der Google Maps Karte. Die Adressen können

nicht nur textuell, sondern auch durch Interaktion mit der integrierten Google Maps-Karte erfolgen. Damit erfüllt es an sich das Ziel des Beispielszenarios für die iPhone-Plattform. Für die angebotenen Dienste verwendet diese Anwendung jedoch keinen Webservice im eigentlichen Sinn. Vielmehr wird die Webseite der Deutschen Bahn parametrisiert aufgerufen und die zurückgegebene HTML-Seite nach den gesuchten Informationen geparst. Der Parser ist hierbei auf den aktuellen Aufbau der Bahn-Webseite angewiesen, was die Software bei jeder Änderung der Webseite durch die Deutsche Bahn AG änderungsbedürftig macht. Dies ist naturgemäß der Nachteil, wenn statt einer standardisierten, offiziellen Schnittstelle mehr oder minder legal eine Webseite zur Datenbeschaffung ausgelesen wird. Da sich dieses Vorgehen rechtlich in einer Grauzone befindet, ist es durchaus schnell möglich, dass dieser Anwendung die Grundlage entzogen wird. Denkbar ist aber, dass dem rechtzeitig durch eine Kooperation von Entwickler und der Deutschen Bahn AG entgegengewirkt werden kann.

Auch bei dieser Anwendung, die ihrerseits eine reine Online-Anwendung ist und damit den Anforderungen des Beispielszenarios vergleichsweise am nächsten kommt, ist zu bemängeln, dass sie nicht ohne weiteres um die Anzeige von Informationen weiterer Dienste erweitert werden kann. Lediglich Informationen, die durch die Verwendung von Google Maps als Geocoding- und Kartengrundlage hinzukommen, können dargestellt werden.

Kritiker bemängeln die statische Abhängigkeit von Googles Kartenmaterial und deren bereits beschriebenen Unzulänglichkeiten. In der Tat ist bei Fahrplan die Austauschbarkeit des MapProvider nicht vorgesehen. Ein weiterer Kritikpunkt ist die Verwendung der Informationen der Deutschen Bahn. Neben der beschriebenen rechtlichen Grauzone ist die Aktualität im Nahverkehrsbereich, welcher durch die jeweiligen Verkehrsverbünde abgedeckt wird, unzureichend. Zwar wird im Abstand weniger Wochen der Datenstand der Deutschen Bahn durch die Verkehrsverbünde aktualisiert, jedoch können kurzfristige Änderungen im Streckennetz hierbei nicht berücksichtigt werden. Die Verkehrsverbünde betonen, dass nur die Informationen aus ihren eigenen Webservices wirklich verlässlich aktuell sind.

Diese Tatsache wollte sich auch der Verkehrsverbund-Informationsdienste konsolidierende Webservice DELFI zunutze machen und bei jeder Abfrage die jeweils zuständigen Webservices der Verkehrsverbünde benutzen [99]. Leider führt dieser Ansatz zu deutlichen Performance-Problemen bezüglich der Antwortzeit einer Abfrage (5-10 Sekunden). Vermutlich deshalb konnte sich die Nutzung von DELFI in der Bevölkerung noch nicht nennenswert durchsetzen. Eine Untersuchung zur Performance-Optimierung von DELFI wäre evtl. eine interessante Belegarbeit.

Trotz der Nachteile erfreut sich Fahrplan bei den iPhone-Nutzern großer Beliebtheit. Das über den Apple App Store erhältliche Programm wurde dort schon ca. 10000 mal allein in der aktuellen Version heruntergeladen (Stand August 2009).

4.3.4 Schlussfolgerungen

In diesem Kapitel wurden einige im Bereich des Beispielszenarios existierenden Anwendungen vorgestellt und bezüglich ihrer Funktionalität untersucht.

Der in dieser Arbeit zu entwickelnde Ansatz muss sich im Vergleich zu den existenten, statisch auf diesen Anwendungsfall zugeschnittenen Programmen behaupten können und zusätzlich mit seinen Vorzügen bzgl. Dynamik, Flexibilität und Erweiterbarkeit punkten.

Bei dieser Untersuchung wurde deutlich, dass die zur Zeit am Markt verfügbaren Lösungen sehr statisch sind und nur für ihr eingeschränktes Einsatzszenario geeignet sind. Erweiterungswünsche müssen bei allen vorgestellten Ansätzen durch Anpassung des vorhandenen Quellcodes erfolgen. Die Anwendungen ermöglichen keine Wiederverwendung oder Mitbenutzung ihrer Programmteile und können auch in keiner Form interoperieren.

Nennenswert ist die beachtliche Anzahl an Plattformen, die durch die Anwendungen DB Railnavigation und Metro unterstützt werden. Erstere hat diese Fähigkeit aufgrund der relativen Plattformunabhängigkeit von JavaME, dem ehemaligen Hoffnungsträger der mobilen Anwendungsentwicklung. Vermutlich durch die massiven Einschnitte, die JavaME beinhaltet, ist diese Technologie auf einem absteigenden Ast, was sich am deutlichsten durch die Einstellung der Entwicklung seitens Sun zeigt [100].

Metro hat dafür einen anderen, allerdings wesentlich aufwendigeren Ansatz. Die Entwickler haben einfach für jede große verbreitete Plattform inklusive der zahlreichen Symbian-Versionen eine separate für diese Plattform zugeschnittene Anwendung entwickelt. Die Nachteile sind klar: eine Aktualisierung propagiert sich durch alle Plattformversionen. Der Entwicklungsaufwand skaliert also linear mit der Anzahl der unterstützten Plattformen.

Fahrplan ist nur für das iPhone verfügbar und ist ein gutes Beispiel dafür, dass dies der Verbreitung einer Anwendung nicht im Wege steht, zumindest dann nicht, wenn die unterstützte Plattform so überwältigenden Erfolg hat wie das iPhone.

4.4 Dynamische Softwarekomponenten

In diesem Kapitel soll untersucht werden, welche Ansätze zur Erfüllung der nicht-funktionalen Anforderungen „Modularität“ (NF_2), „Dynamik“ (NF_4) und „(dynamische) Erweiterbarkeit“ (NF_3) bereits existieren und somit im Konzept angewendet werden könnten. Diese Anforderungen beziehen sich auf die Integration der MapServices in das Kartenmaterial der MapProvider, weshalb dieses Kapitel Möglichkeiten untersucht, wie die MapService-Zugriffslogik in dynamische Module gekapselt werden kann. Die Modularität soll die Austauschbarkeit und Erweiterbarkeit der Lösung ermöglichen. Der Dynamikaspekt soll den Ansatz von den vorhandenen statischen Lösungen (vgl. Kap. 4.3) deutlich abgrenzen und den Vorgang der MapService-Integration flexibel machen. Die genannten Eigenschaften sollen wenn möglich nicht nur die Einbindung der MapServices haben sondern auch die Integration der verschiedenen MapProvider. Die in diesem Kapitel untersuchte Technologie dient daher dem Erreichen zahlreicher Ziele dieser Arbeit, weshalb die Ergebnisse dieses Kapitels sehr wichtig für Konzeption und Implementierung sein werden.

Zunächst werden grundlegende Begriffe für dynamische Softwarekomponentenmodelle definiert und danach die vorhandenen Ansätze untersucht. Abschließend wird betrachtet, welche der Ansätze für eine mobile Plattform, konkret für die im Grundlagenkapitel ausgewählte Android-Plattform, geeignet sind.

Da die Arbeit im Grundlagenkapitel auf die Android-Plattform eingegrenzt wurde, kommen für den Fall der Positionierung der Dynamik auf dem Endgerät nur Ansätze für die Sprache Java (SE) in Frage. Zwar hat Google durch das Native Development Kit (NDK) die Möglichkeit geschaffen, auch in C zu entwickeln, jedoch soll von dieser Alternative nur möglichst selten und nur für rechenintensive Prozesse Gebrauch gemacht werden. Google bittet die Entwickler, vor allem wegen der Fehleranfälligkeit der Softwareentwicklung in C, bei der Anwendungsentwicklung für Android Java zu verwenden [101].

Diese Einschränkung verringert die Anzahl der existierenden, geeigneten Ansätze für dynamische Software auf einen einzigen: OSGi. In Ermangelung alternativer Ansätze entfällt dadurch in diesem Kapitel der sonst übliche Technologievergleich.

Sollte sich herausstellen, dass OSGi nicht für Android käme man nicht umhin, die Möglichkeiten für eine Android-geeignete Eigenentwicklung zu überprüfen. Dies würde den Fokus der Arbeit auf die Konzeptionierung und prototypische Entwicklung eines dynamischen Modularisierungsansatzes für die Android-Plattform verschieben, um den zentralen Aspekt der Dynamik zu berücksichtigen. Das würde aber zu stark von der eigentlichen Aufgabenstellung abweichen und so müsste von dem Gedanken der clientseitigen dynamischen Modularisierung der Anwendung

Abstand genommen werden und ein serverseitiger, dynamisch erweiterbarer Ansatz anvisiert werden.

Da es aber nach der Untersuchung der vorhandenen MapService-Integrationsmechanismen ein Teilziel dieser Arbeit ist, den etablierten, serverseitigen Ansätzen einen konsequent clientseitigen, dynamisch-modularen Ansatz vergleichend entgegenzusetzen, soll der verheißungsvollste Ansatz OSGi nun ausführlich diskutiert werden. Am Ende der State-of-the-art-Analyse werden die hier gewonnenen Erkenntnisse zusammengefasst und der weitere Werdegang dieser Arbeit festgelegt.

4.4.1 Grundbegriffe der dynamischen Softwareentwicklung

Bevor mit der eigentlichen Untersuchung von OSGi begonnen werden kann, müssen einige Begriffe dynamischer Softwareentwicklung formuliert werden. Da sich der Dynamik-Aspekt (NF_3) und die damit verbundenen Aspekte Erweiterbarkeit, Modularität (NF_2) und Flexibilität (NF_1) sowohl für eine client- als auch für eine serverseitige Lösung einsetzen lassen, müssen diese Begriffe möglichst allgemein formuliert werden. OSGi ist prinzipiell sowohl für Server als auch für mobile Anwendungen geeignet. Somit ist die grundlegende Positionierung der Dynamik im Gesamtsystem durch diese Technologie nicht eingeschränkt. Alle Freiheitsgrade bleiben für die Konzeption erhalten.

4.4.1.1 Modularität

Modularität lässt sich am besten dadurch beschreiben, indem erklärt wird, was ein Modul ist. Ein Modul hat eine Reihe von Eigenschaften die nachfolgend beschrieben werden:

- Self-contained: Ein Modul ist eine Komposition aus kleineren, unabhängigen Teilen, mit Ausnahme von wohl definierten Abhängigkeiten und kann nur als Ganzes verwendet werden.
- Cohesive: Die Elemente eines Moduls haben einen starken logischen Bezug zueinander.
- Loose coupling: Module besitzen untereinander eine geringe Kopplung.
- Public API: Ein Modul besitzt eine wohl definierte öffentliche Schnittstelle und verbirgt die Interna der Implementierung.
- Dependencies: Die Abhängigkeiten eines Moduls sind wohl definiert.

- Deployment format: Ein Modul „läuft“ in einem Container bzw. einer Laufzeitumgebung und wird dort in einem wohl definierten Format installiert.

(Quelle: [102])

Software-Modularisierung ist ein Organisationsparadigma zur Reduktion der Komplexität. Sie ermöglicht das Zergliedern einer Anwendung in potentiell austauschbare Programmteile. In anderen Worten: „Module abstrahieren vom schwer überschaubaren Problem hin zu übersichtlichen Herausforderungen. Diese können isoliert voneinander und möglicherweise auch arbeitsteilig in Angriff genommen werden und bieten die Möglichkeit der Wiederverwendung“ [102]. Die Modularisierung einer Anwendung ist die Grundlage für ihre Flexibilität und die Erweiterbarkeit. Schnittstellen sichern die Austauschbarkeit der Programmteile.

4.4.1.2 Komponentenmodell

Die im Absatz „Modularisierung“ beschriebenen Module werden oft auch als Komponenten bezeichnet. Ein Komponentenmodell definiert konkret das Modularisierungsparadigma mit den Eigenschaften der Komponenten und ihrer Zusammenarbeit in einem Modell.

Gruhn und Thiel definieren diesen Begriff folgendermaßen:

„Ein Komponentenmodell legt einen Rahmen für die Entwicklung und Ausführung von Komponenten fest, der strukturelle Anforderungen hinsichtlich Verknüpfungs- bzw. Kompositionsmöglichkeiten sowie verhaltensorientierte Anforderungen hinsichtlich Kollaborationsmöglichkeiten an die Komponenten stellt. Darüber hinaus wird durch ein Komponentenmodell eine Infrastruktur angeboten, die häufig benötigte Mechanismen wie Verteilung, Persistenz, Nachrichtenaustausch, Sicherheit und Versionierung implementieren kann.“ [103]

Bekannte Beispiele für Komponentenmodelle sind Distributed Component Object Model (DCOM), Enterprise Java Beans (EJB), OSGi und das Komponentenmodell von CORBA. Nicht jedes Komponentenmodell ist dynamisch. Von den hier angeführten Beispielen ist hat nur OSGi dynamische Aspekte von vorneherein grundlegend integriert.

4.4.1.3 Dynamik

Dynamik im Sinne von Softwarekomponenten bezieht sich auf die Einsetzbarkeit von Komponenten bzw. Modulen, die zur Entwicklungszeit noch nicht oder nicht vollständig bekannt waren. Eine Hauptanwendung ist dynamisch, wenn sie auf die Änderungen der ihr verfügbaren Module reagiert und diese verwenden kann.

Die Dynamik wird meist erreicht, indem ein Hauptprogramm durch dynamische Komponenten (PlugIns) erweitert wird. Die Dynamik bezieht sich hierbei auf den Lebenszyklus des Moduls (Installation, Verwendung, Deinstallation). Das Hauptprogramm wird durch die PlugIns in seinem Funktionsumfang erweitert oder der gesamte Funktionsumfang ist in den PlugIns gekapselt und die einzige Aufgabe des Hauptprogramms ist das Verwenden der Plugins. Dies erreicht das Hauptprogramm, ohne extra für jedes neu hinzugekommene Plugin programmatisch angepasst zu werden. Im Idealfall muss in der Lebenszeit einer Anwendung nichts am Hauptprogramm geändert werden, sondern alle Aktualisierungen und Erweiterungen werden an den Plugins vorgenommen.

Ein Sonderfall der Dynamik im Softwarekontext ist die Möglichkeit, Plugins zur Laufzeit das Hauptprogramms zu installieren und dem Hauptprogramm zugänglich zu machen, ohne dass ein Neustart der Hauptanwendung erforderlich ist, was Laufzeitdynamik genannt wird. Dieser komplexe Sachverhalt kann nur durch eine von der Hauptanwendung verwalteten Laufzeitumgebung realisiert werden, wie es beispielsweise bei OSGi der Fall ist.

Grundpfeiler der Dynamik ist die Modularisierung und die dabei oft verwendete Spezifikation und Entwicklung von Schnittstellen. Die Module implementieren Schnittstellen, die während der Entwicklung der Hauptanwendung definiert wurden und daher dem Hauptprogramm bekannt sind. Die konkrete Implementierung, die im Modul gekapselt ist, bleibt dem Hauptprogramm verborgen.

4.4.2 Ausgangssituation

Die OSGi Technologie ist das einzige zum Zeitpunkt (Sommer 2009) verfügbare dynamische Softwarekomponentenmodell für JavaSE, das kompatibel zu Androids DalvikVM ist. Lange Zeit hat Sun aus nicht-technischen Vorbehalten gegenüber OSGi nach Alternativen zu OSGi bezüglich der Modularisierung von Java gesucht und das, obwohl Sun Gründungsmitglied der OSGi Alliance war. Diese Suche gipfelte in der Entwicklung von Java Module System (JSR 277), welches aber lediglich statische Modularisierung vorsieht.

Peter Kriens, OSGi Technical Director der OSGi Alliance, betrachtet Suns eigenen Ansatz kritisch und schreibt in seinem Review zu JSR 277: „The Expert Group took a simplistic module loading model and ignored many of the lessons that we (OSGi Alliance, Anm. d. Verf.) learned over the past 8 years.“[104]

Mittelfristig wird es keine Alternative zu OSGi im Bereich „Laufzeitdynamisches Modulssystem für JavaSE“ geben. Da der Technologievergleich in diesem Fall entfällt, wird sich nun besonders gründlich mit OSGi auseinander gesetzt werden.

Ob in Client oder Server: Hauptziel der Verwendung dynamischer

Sofwarekomponenten in dieser Arbeit ist die Kapselung und dynamische Integration verschiedener Dienste bzw. Zugriffslogiken in Plugins, die von der Anwendung (Client oder Server) genutzt werden, um die verschiedenen Dienste standardisiert zu integrieren. Dieser Ansatz ist das „Allgemeine Plugin-Konzept". Dieses Konzept, welches die Frage nach einer geeigneten Lösung für dieses Ziel beantworten soll, stützt sich auf Technologien, die dieses Konzept ermöglichen. Diese Technologie gibt es zur Zeit für JavaSE nur in Form von OSGi.

Wie bereits in den vorhergehenden Kapiteln festgestellt wurde, sind die vorhandenen, wenig dynamischen Ansätze allesamt serverseitig. Es ist daher naheliegend und interessant, den clientseitigen Weg als Alternative zu untersuchen. Daher wird sich bei der Betrachtung verfügbarer Technologien zur dynamischen Softwaremodularisierung für die Clientseite auf die Sprache Java konzentriert, da sie die einzige relevante Sprache für die Entwicklung von Android-Anwendungen ist.

Gefahr dem Fall, dass OSGi auf Android nicht zu den erhofften Ergebnissen führen wird, können die Ergebnisse immernoch der dynamischen Modularisierung einer Java-basierten Serveranwendung dienen.

Serverseitige Java Enterprise Edition (J2EE) Lösungen wie Enterprise Java Beans (EJB) bieten zwar ausgereifte Möglichkeiten für Modularisierung und Dynamik, ihr Einsatz auf einem Android-basierten Endgerät ist aber auf Grund der Einschränkungen von Androids DalvikVM und auch aus Performance-Sicht nicht sinnvoll. Da dieser Ansatz die Konzeption unnötig auf serverseitige Lösungen einschränken würde, wurde von seiner Verwendung abgesehen.

Ungeachtet des konkreten Ergebnisses der folgenden Untersuchung, kann durch das beschriebene Vorgehen in der Arbeit fortgefahren werden und gemäß der gewonnenen Erkenntnisse die Konzeption erarbeitet werden.

4.4.3 OSGi Service Plattform

Die OSGi Service Platform ist ein (Laufzeit-) dynamisches, serviceorientiertes Modulsystem und Komponentenmodell für Java. Sie ermöglicht die dynamische Modularisierung, Laufzeitdynamik und dynamische (Fern-)Wartung von Anwendungen. Die Zergliederung von Anwendungsteilen in Module (Bundles) und die Kapselung von Funktionalität in Dienste steht dabei als eines der Hauptparadigmen im Vordergrund.

Die OSGi Service Platform wurde von der OSGi Alliance spezifiziert, die von Unternehmen wie Sun, Ericsson und IBM gegründet wurde. Die Spezifikation umfasst die API und die Testfälle, die von den OSGi Implementierungen bestanden werden müssen um zertifiziert zu werden. Ein bekanntes Einsatzbeispiel ist die

Eclipse IDE die seit Version 3.0 auf der OSGi-Implementierung Equinox aufsetzt.

Die Verwendung der OSGi Service Platform bietet als dynamisches und serviceorientiertes Modulsystem für die Anwendung drei Eigenschaften: Modularisierung, Laufzeitdynamik und Serviceorientierung. „Das Konzept der Modularisierung ist heute als Mittel zur Komplexitätsreduzierung bei der Entwicklung großer Anwendungsysteme unumstritten. In Java gestaltete sich die Umsetzung dieses Konzepts bisher schwierig, da unterstützende Sprachkonzepte fehlten. Monolithische Anwendungssysteme waren die häufige Folge. Die OSGi Plattform löst dieses Problem, indem sie ein dynamisches Modulsystem für Java bereitstellt" [105].

Das Kapitel 4.4.4 soll eine kompakte Einführung in die OSGi-Technologie und die damit verbundenen, realen Möglichkeiten bieten. Diese Möglichkeiten ergeben Denkansätze, die im Kapitel 5 aufgegriffen werden können. Zum tieferen Verständnis von OSGi sei auf [105] so wie die OSGi-Spezifikationen verwiesen.

Die Objektorientierung liefert im Allgemeinen und so auch bei Java im Speziellen einige Möglichkeiten zur Modularisierung in Form von Klassen, Packages und Sichtbarkeitsregeln. Es gibt in Java aber kein Modulkonzept oberhalb von Packages. Hier greift das Komponentenmodell von OSGi und bildet damit ein Gegenmittel gegen monolithische Systeme. OSGi ermöglicht ein Component Based Software Engeneering (CBSE) für die Java-Anwendungsentwicklung. Softwaresysteme werden hierbei aus verschiedenen Modulen mit wohl definierten öffentlichen Schnittstellen und Abhängigkeiten zusammengesetzt.

Die Dynamik der Modularisierung bezieht sich nicht nur auf die Entwicklungszeit. So können bei OSGi Module auch im laufenden Betrieb unter Berücksichtigung ihrer Abhängigkeiten (Abhängigkeitsmanagement) installiert, aktualisiert und entfernt werden. Dies macht OSGi für serverseitige und hochverfügbare Systeme interessant und wird als Hot Deployment bezeichnet.

Ein weiterer zentraler Punkt von OSGi ist die Serviceorientierung. Die einzelnen Module können Objekte und ihre Methoden als Services bereitstellen. Dies ermöglicht die lose Kopplung der Module und ist Grundlage von Dynamik und Austauschbarkeit. Die Nutzung eines Moduls durch ein anderes erfolgt auf Package-Ebene und erzeugt Abhängigkeiten, die vom Abhängigkeitsmanagement von OSGi aufgelöst werden.

Der Einsatz von OSGi (und generell dynamischen Modularisierungssystemen) bietet zahlreiche Nutzenpotenziale:

- Erhöhte Flexibilität durch rigorose Trennung von API und Implementierung
- Einsparung von Entwicklungskosten durch Wiederverwendung von Modulen
- Einsparung von Betriebskosten durch standardisiertes Lifecycle Management

- Hohe Qualitätseffizienz durch gute Testbarkeit aufgrund Nutzung neuer Möglichkeiten, z.B. in der Softwareverteilung (Deployment)

(Quelle: [106])

Alle genannten Vorzüge der OSGi Service Platform sind für diese Arbeit interessant und nützlich, sodass nun der Aufbau diskutiert werden kann.

4.4.4 Aufbau der OSGi Service Platform

4.4.4.1 OSGi Service Platform Spezifikation

Die Spezifikation der OSGi Service Platform untergliedert sich in Core Specification und Service Compendium. Die Core Specification definiert das OSGi Framework und die auf ihm aufbauenden Framework Services. Das OSGi Framework bildet die grundlegende Infrastruktur für die Module.

Das Service Compendium definiert eine Reihe von auf dem OSGi Framework aufbauenden Standardkomponenten (Standard Services), die in eigenen Anwendungen von den eigenen Komponenten bzw. Modulen genutzt werden können. Es handelt sich dabei um fertige Lösungen für häufig auftretende Probleme und Anforderungen. Ein Beispiel hierfür ist der HTTP-Service, der einen einfachen Webserver stellt, an dem die Anwendungsmodule ihre Ressourcen anmelden können, welche dann über HTTP ansprechbar sind.

Seit Version 4.1 der Spezifikation ist die Mobile Specification (JSR 232) Teil des Service Compendium. Es handelt sich dabei um einen Satz spezieller Services, die für den Einsatz auf mobilen Endgeräten gedacht sind. Abbildung 4.6 verdeutlicht diesen Zusammenhang.

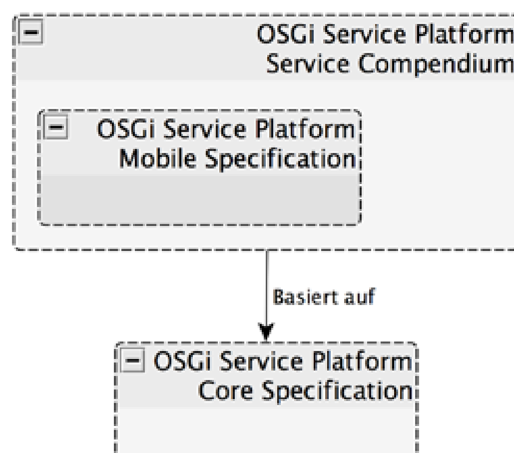


Abbildung 4.6: Struktur der OSGi Spezifikation

4.4.4.2 Das OSGi Framework

Die Basiskomponente der OSGI Service Platform ist das OSGi Framework, welches in der OSGi Core Specification definiert ist. Es bildet die Laufzeitumgebung für die Module und ermöglicht Installation, Ausführung und Deinstallation dynamisch zur Laufzeit. Das OSGi Framework ist in mehrere logische Schichten oder Aufgabenbereiche strukturiert, die aufeinander aufbauen, wie es Abbildung 4.7 zeigt.

Das Framework ist auf den verschiedensten Java Laufzeitumgebungen (JRE) lauffähig, wie z.B. die Java Standard Edition (JavaSE) oder die Java Micro Editon (JavaME). Um unabhängig von einer JRE zu sein, ist in OSGi die benötigte Laufzeitumgebung mit ihren jeweiligen unterstützten Methoden über die OSGi Specification Execution Environments definiert. Die minimal benötigte Umgebung wird in der Spezifikation OSGi/Minimum-1.1 festgelegt, deren Anforderungen eine Untermenge von JavaME sind. Bei der Installation eines Moduls kann das Framework zuvor überprüfen, ob es in der vorliegenden Umgebung überhaupt lauffähig ist.

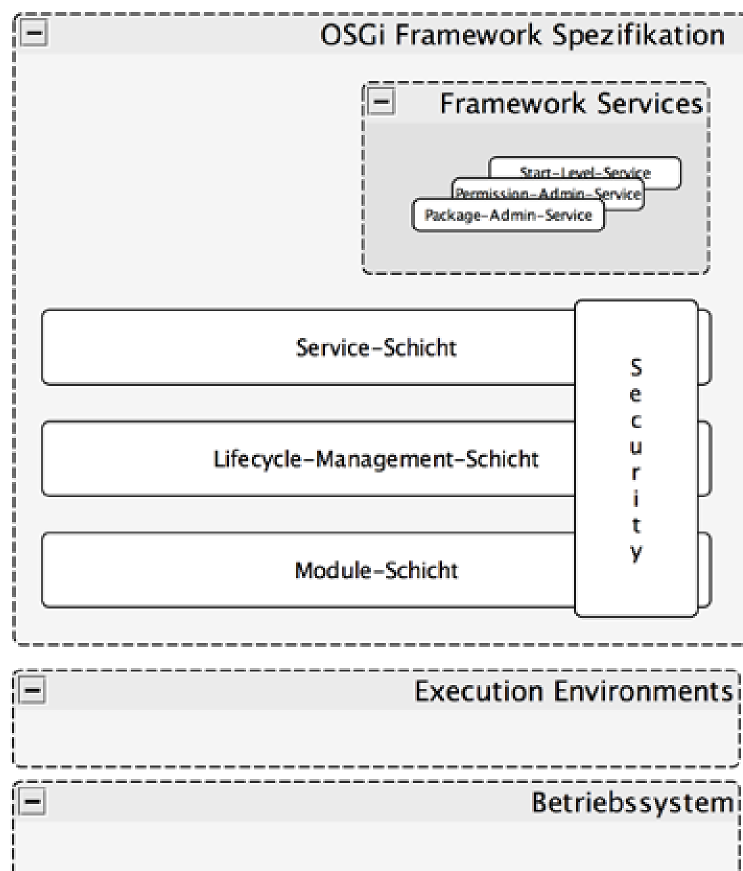


Abbildung 4.7: Schichten des OSGi Framework

4.4.4.3 Module-Schicht

Diese unterste logische Schicht des OSGi Framework definiert das Bundle als grundlegende Modularisierungseinheit und damit den statischen Aspekt des Modulkonzepts der OSGi Service Platform. Vereinfacht gesagt werden Module bzw. PlugIns bei OSGi Bundles genannt. Die Bundles „leben“ in ihrer Laufzeitumgebung - dem OSGi Framework. Sie können eigenständig im Framework installiert, aber erst bei Erfüllung einiger Voraussetzungen genutzt werden.

Technisch handelt es sich bei einem Bundle um ein Java-Archiv (Jar-Datei), welches die Implementierung, Ressourcen und eventuell andere Java-Archive als Bibliotheken beinhaltet. Zusätzlich befindet sich in der Jar-Datei ein Verzeichnis META-INF mit der Datei MANIFEST.MF. Diese deklariert Informationen über das Bundle, wie z.B. Name, Version, benötigtes Execution Environment, angebotene und importierte Packages sowie Abhängigkeiten zu anderen Bundles.

Das Framework verwendet diese Informationen, um Abhängigkeiten explizit zu verwalten und so die notwendigen Voraussetzungen zu erfüllen, um das Bundle betreiben zu können.

„Jedes im OSGi Framework installierte Bundle besitzt einen eigenen Class Loader, so dass die im System installierten Bundle voneinander separiert sind und die Import-Export-Beziehungen zwischen Bundles explizit gesteuert werden können“ ([105], S.89) Dies bewirkt, dass sogar gleiche Bundles in unterschiedlichen Versionen voneinander separtiert gleichzeitig nebeneinander existieren dürfen.

Um eine Klasse eines Bundles in einem anderen Bundle nutzen zu können, muss das andere Bundle das Package, in dem diese Klasse ist, explizit in seiner Manifest-Datei exportieren (Package-Abhängigkeit). Der Export ist nicht-rekursiv, wodurch die Sub-Packages verborgen bleiben. Es genügt hierbei nur das öffentliche Package zu exportieren, welches die anzubietenden öffentlichen Methoden enthält. Die zugrunde liegende Implementierung kann dem nutzenden Bundle verborgen bleiben. In der Praxis wird das Nutzungsverhältnis zwischen Bundles meist über die Abstraktion zu OSGi Services beschrieben (vgl. Kap. 4.4.4.5).

Das öffentliche Package muss daher dem nutzenden und dem implementierenden Bundle bekannt sein, um eine Abhängigkeit formulieren zu können. Diese Abhängigkeit wird zur Laufzeit durch das OSGi Framework aufgelöst, zur Entwicklungszeit muss eine IDE wie Eclipse für die OSGi-Plugin Entwicklung konfiguriert werden.

Ein Beispiel für eine Manifest-Datei befindet sich in Listing 4.1. Das Bundle „My Bundle“ macht das Package org.mnsoft.mybundle öffentlich und fordert die Anwesenheit eines Packages Namens org.mnsoft.anotherbundle aus einem anderen Bundle (Package-Abhängigkeit).

Import-Package Angaben sind zwingend, d.h. sollte kein Bundle verfügbar sein, welches das gewünschte Package exportiert, kann das nutzende Bundle nicht gestartet werden. Mit der Angabe „resolution:=optional" kann die Abwesenheit des Package während des Starts ignoriert werden, was aber auch mit sich bringt, dass die importierte Funktion zur Laufzeit nicht zur Verfügung steht.

Alternativ zu Import-Package können mit Require-Bundle alle von dem angegebenen Bundle exportieren Packages gleichzeitig importiert werden (Bundle-Abhängigkeit). Durch „visibility:=reexport" können die importierten Packages vom nutzenden Bundle selbst wiederum exportiert werden. Diese Vorgehensweise hat allerdings den Nachteil, dass man sich unter Umständen an eine spezifische Implementierung dieses Bundles bindet und somit auf Herstellerunabhängigkeit unnötigerweise verzichtet.

Listing 4.1: Beispiel einer Bundle Manifest-Datei

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.mnsoft.mybundle
Bundle-Name: My Bundle
Bundle-Version: 1.2.3
Export-Package: org.mnsoft.mybundle
Import-Package: org.mnsoft.anotherbundle;version="1.0.0";resolution:=optional
Require-Bundle: org.mnsoft.justanotherbundle;visibility:=reexport
```

In Listing 4.1 wird in der Import-Package-Zeile eine Versionsnummer angegeben. Diese Angabe bezieht sich auf die minimal benötigte Bundle-Version bzw. Package-Version. Alternativ kann beim Import von ‚Packages auch ein Versionsbereich angegeben werden. In Listing 4.2 sind einige Beispiele für die Angabe eines Versionsbereiches.

Listing 4.2: Möglichkeiten der Versionsangabe in der Manifest-Datei

```
[1.2.3, 4.5.6)    1.2.3 <= x <  4.5.6
[1.2.3, 4.5.6]    1.2.3 <= x <= 4.5.6
(1.2.3, 4.5.6)    1.2.3 <  x <  4.5.6
(1.2.3, 4.5.6]    1.2.3 <  x <= 4.5.6
1.2.3             1.2.3 <= x
```

(Quelle: OSGi R4 Core Specification, S.28)

Die Versionsnummern sind folgendermaßen aufgebaut: major.minor.micro. qualifier, zum Beispiel Version 1.4.2.beta . Im Framework sind installierte Bundles eindeutig durch ihren symbolischen Namen (Bundle-SymbolicName) und ihre Versionsnummer identifiziert. Dadurch ist es möglich, verschiedene Versionen

eines Bundles gleichzeitig im System installiert zu haben.

Die beschriebene Verdrahtung der Bundles durch expliziten Import und Export von Packages erscheint zunächst etwas statisch, da zumindest die öffentlichen Packages zur Entwicklungszeit den jeweilgen Bundles bekannt sein müssen. Der eigentliche Ansatz, das Nutzungsverhältnis zwischen den Bundles lose gekoppelt und dynamisch zu gestalten, ist die Abstraktion zu OSGi Services und den mit ihnen verbundenen Vermittlungsmechanismen wie z.B. die Service Registry. Im Abschnitt 4.4.4.5 wird detailierter auf dieses Thema eingegangen.

Neben dem klassischen Package-Import der zu Startzeit des Bundles aufgelöst wird, gibt es als dynamische Alternative die Manifest-Anweisung DynamicImport-Package, welche zur Laufzeit bei Bedarf Packages bzw. Package-Gruppen importiert. Beispielsweise importiert die Anweisung DynamicImport-Package: org.mnsoft.* bei Bedarf aus der Menge der installierten Bundles die benötigten Packages der Gruppe „org.mnsoft". Die Auflösungsversuche erfolgen hierbei nicht nur zur Startzeit, sondern auch bei Verwendung des Packages.

4.4.4.4 Life-Cycle-Management Schicht

In dieser Schicht wird der Lebenszyklus der Bundles und damit die dynamischen Aspekte der Modularisierung beschrieben. Es werden hierfür Zustände definiert, die ein Bundle während seiner „Lebenszeit" im OSGi Framework annehmen kann sowie Operationen festgelegt, die zur Änderung dieser Zustände führen. Vermittels Management Agents kann das OSGi Framework von außen gesteuert werden, indem die Zustände von Bundles durch Operationen (z.B. install, start, stop...) manipuliert werden. Ein Management Agent kann eine einfache Konsole wie z.B. die Equinox-Konsole sein, es sind aber auch komplexe Verwaltungsanwendungen mit graphischer Benutzeroberfläche möglich.

Die Bundle-Zustände und die Operationen, die zu Zustandsübergängen führen, werden in Abbildung 4.8 dargestellt. Die Manipulation der Bundles kann sowohl über einen Management Agent als auch direkt programmatisch erfolgen.

Damit ein Bundle im OSGi Framework verfügbar ist, muss es mit install in das Framework installiert werden und ist damit im Zustand INSTALLED. Danach kann das Framework versuchen, alle in der Manifest-Datei des Bundles beschriebenen Abhängigkeiten aufzulösen (resolving), indem es prüft, ob die genannten Packages von bereits installierten Bundles exportiert werden. Konnten alle Abhängigkeiten aufgelöst werden, wird das Bundle in den Zustand RESOLVED versetzt, wenn nicht, bleibt es im Zustand INSTALLED. Das Bundle ist im Zustand RESOLVED bereit, gestartet zu werden.

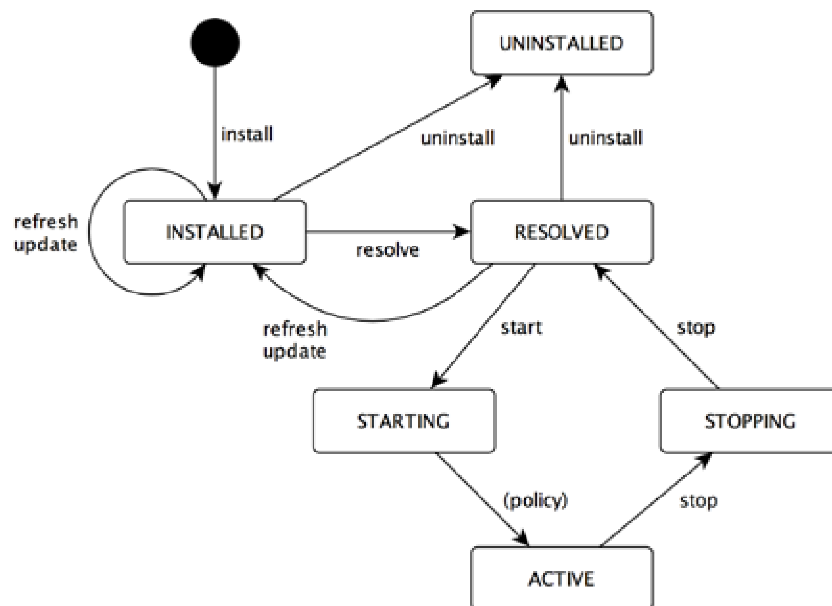


Abbildung 4.8: Lebenzyklus der OSGi Bundles

Wie aus Abbildung 4.8 ersichtlich korrespondieren die Manipulations-Methodennamen mit den zu erreichenden Zuständen. So kann mit start der Startvorgang des Bundles angestoßen werden, wofür das Bundle in den Zustand STARTING geht. Da dies je nach Komplexität des Bundles ein zeitaufwändiger Prozess sein kann, gibt es die Möglichkeit, in der Manifest-Datei Aktivierungsstrategien (Bundle Activation Policies) festzulegen. Wird die Strategie Lazy gewählt, bleibt das Bundle solange im STARTING Zustand, bis eine Klasse dieses Bundle benötigt. Nach erfolgreichem start()-Aufruf ist das Bundle im Zustand ACTIVE und kann nun verwendet werden. Bei Problemen beim Start fällt das Bundle in den Zustand RESOLVED zurück. Analog zu diesem Ablauf erfolgt das Stoppen eines Bundles mittels stop.

Wie bereits beschrieben, bietet OSGi ein Hot Deployment auf Modulebene. Softwaresysteme können dadurch im laufenden Betrieb auf Bundle-Ebene aktualisiert werden. Um dies zu erreichen, gibt es die Operationen update und refresh.

Um den Lebenszyklus eines Bundles direkt programmatisch zu steuern, müssen die Interfaces Bundle (org.osgi.framework.Bundle) und BundleContext betrachtet werden. Der BundleContext ist die Schnittstelle zur Interaktion mit dem OSGi Framework. So ermöglicht beispielsweise die Methode installBundle() die Installation einer Bundle-Jar-Datei ins Framework unter Angabe seiner URL. Das Bundle-Interface definiert Methoden zum Steuern des Lebenszyklus des Bundles.

4.4.4.5 Service-Schicht

In der Service-Schicht wird festgelegt, wie innerhalb des Frameworks die in Bundles strukturierten Klassen in Form von OSGi Services verwendet werden können. Die OSGi Services sind also die Abstraktion der nutzbaren Funktionen eines Bundles, die anderen Bundles in loser Kopplung zur Verfügung gestellt werden. Damit wird in dieser Schicht die Serviceorientiertheit von OSGi produziert. Das service-orientierte Programmiermodell ist Grundlage der dynamischen Zusammenarbeit der Bundles, und muss - um den Dynamikaspekt von OSGi einzuführen - genauer betrachtet werden. Erweiterte Ansätze für Dynamik bei OSGi werden dann im Kapitel 4.4.5 diskutiert.

Ein OSGi Service ist ein normales Java-Objekt (POJO - Plain Old Java Object), dessen Klasse ein selbst-definiertes Java-Interface implementiert. Dieses Interface beschreibt die öffentlichen Methoden, die das Java-Objekt anderen Bundles anbieten soll, wodurch Unabhängigkeit von der konkreten Implementierung eines Anbieters erreicht wird.

Der Ablauf ist schematisch in Abbildung 4.9 dargestellt, wobei die Ähnlichkeit zum SOA-Schema deutlich wird.

1. Das anbietende Bundle implementiert ein selbst-definiertes Service-Interface und registriert sich zu einem beliebigen Zeitpunkt zur Laufzeit als Implementierung (Anbieter) dieses Interfaces bei der Service Registry.

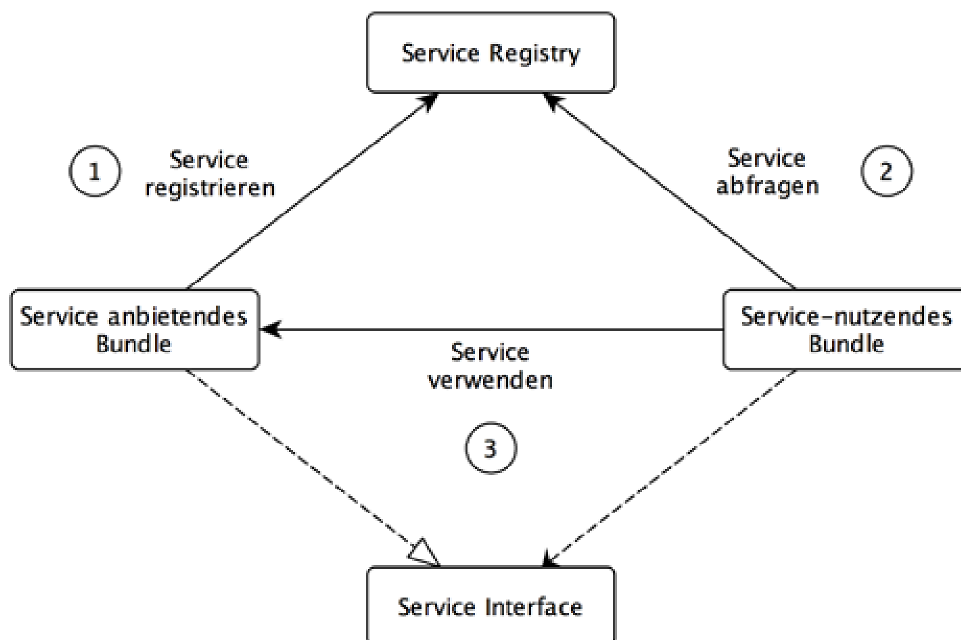


Abbildung 4.9: schematischer Ablauf OSGi Service Registry

2. Das nutzende Bundle fragt den Service dann unter Verwendung des Service-Interfaces von der Service Registry ab, um ihn zu nutzen. Dieses Nutzungsverhältnis bringt die beteiligten Bundles in Service-Abängigkeit, welche gleichzeitig Package-Abhängigkeit impliziert, da das Package der Service-Interface-Klasse importiert werden muss. Die Service Registry sucht bei einer Anfrage nach einer registrierten Implementierung des gewünschten Service-Interface.

3. Der Service wird genutzt, indem die Methoden dieser Implementierung direkt aufgerufen werden.

Die Service Registry ist eine Indirektionsstufe, die eine lose, dynamische Kopplung zwischen den Bundles insofern ermöglicht, dass das nutzende Bundle nicht in der Erzeugung des angebotenen Services involviert ist.

Mit der Laufzeitdynamik der Bundles geht einher, dass auch die OSGi Services dynamisch sind, d.h. zur Laufzeit muss immer damit gerechnet werden, dass ein registrierter Service abgemeldet wird und dadurch nicht mehr zur Verfügung steht. Um dem Rechnung zu tragen, sind in der OSGi Spezifikation einige Möglichkeiten angegeben, um den Umgang mit dynamischen Services zu ermöglichen und auf Änderungen zu reagieren. Dies sind unter anderem der Service Tracker und der Declarative Services-Ansatz (vgl. Kap. 4.4.5).

Seine Service-Registrierung nimmt das Bundle in seiner Implementierung des Interface BundleActivator jeweils manuell über den BundleContext in der start()-Methode und automatisch in der stop()-Methode vor (siehe Listing 4.3). In der BundleActivator implementierenden Klasse muss lediglich noch das Service Interface importiert werden.

Listing 4.3: Auszug aus dem BundleActivator des anbietenden Bundles

```
import org.mnsoft.mybundle.MyService;
...
ServiceRegistration sreg = bundleContext.registerService(MyService.class.getName(), new
MyServiceImpl(), null )
```

Die bei der Registrierung optional angebbaren Properties können zum Beschreiben des Service verwendet werden, nach denen das nutzende Bundle bei der Suche nach einem passenden Service filtern kann. Diese Filterausdrücke sind Zeichenketten des Formats „String Representation of LDAP Search Filters“ [107]. Der Filter-Syntax kann Abbildung 4.10 entnommen werden.

Die Service-Nutzung ist in Listing 4.4 ersichtlich. Das Service-nutzende Bundle importiert ebenfalls dieses Service Interface, damit es per Reflection auf dessen Servicenamen zugreifen kann und seine angebotenen Methoden kennt. Im

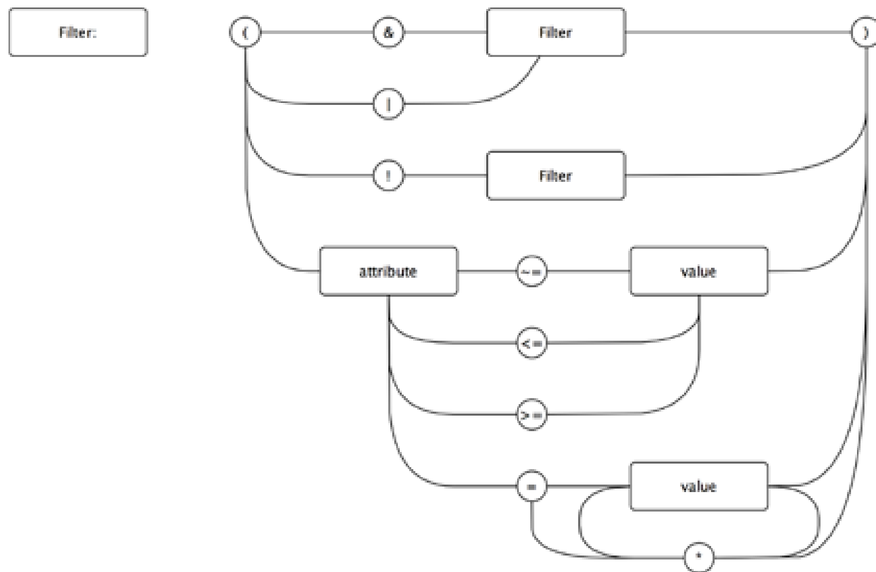


Abbildung 4.10: OSGi Filter Syntax

BundleActivator des nutzenden Bundles kann über den BundleContext die ServiceReference auf den gewünschten Service von der Service Registry abgerufen werden. Da auf Grund der Laufzeitdynamik ein gerade noch registriertes Bundle nach Erhalt der ServiceReference plötzlich abgemeldet und damit nicht mehr verfügbar sein könnte, muss vor Verwendung des mittels dieser ServiceReference abgerufene Service sichergestellt werden, das der Service (immer noch) verfügbar (ungleich null) ist.

Listing 4.4: Auszug aus dem BundleActivator des nutzenden Bundles

```
import org.mnsoft.mybundle.MyService;
...
ServiceReference sref = bundleContext.getServiceReference(MyService.class.getName());
if (sref != null)
MyService service = (MyService) bundleContext.getService(sref);
if (service != null)
service.myMethod();
```

Damit dies wie beschrieben funktionieren kann, muss das anbietende Bundle vor dem nutzenden Bundle gestartet werden. Im einfachsten, aber nicht sonderlich dynamischen Fall, kann dies bei OSGi durch die Angabe von Start-Leveln (je kleiner desto früher) zur Beeinflussung der Bundle-Startreihenfolge sichergestellt werden. Im Kapitel 4.4.5 wird auf dieses Thema näher eingegangen.

4.4.4.6 Security-Schicht

In der Security-Schicht wird das Sicherheitskonzept von OSGi spezifiziert. Es ermöglicht, die Ausführungsrechte (OSGi Permissions) einzelner Bundles gezielt einzuschränken und individuell zu konfigurieren. Es basiert auf dem Java-Sicherheitsmodell (ab JDK 1.2 eingeführt), den Java permissions, welche um OSGi-spezifische Anforderungen erweitert werden.

4.4.5 Dynamische Services mit OSGi

In diesem Kapitel sollen Ansätze vorgestellt werden, die eine echte lose Kopplung zwischen den Bundles und damit dynamische Modularität ermöglichen. Die in Kapitel 4.4.4.5 vorgestellte Kopplung hat den Nachteil, dass die Startreihenfolge relevant ist, die bisher nur durch das statische Mittel der Start-Level beeinflusst wurde. Das anbietende Bundle musste vor dem nutzenden Bundle gestartet werden, da sonst (bei nicht-optionalen Abhängigkeiten) das nutzende Bundle nicht starten konnte, was keiner wirklich losen Kopplung entspricht. Um der Laufzeitdynamik der Bundles vollständig Raum zu geben sind erweiterte Mechanismen vonnöten.

Das Grundkonzept, um diese Dynamik zu erreichen ist das Warten auf Veränderungen bezüglich der verfügbaren Bundles und deren angemeldeten Services. Das Bundle, welches einen Service nutzen möchte, wartet, bis es darüber informiert wird, dass der gewünschte Service verfügbar ist und beginnt erst dann mit der Nutzung.

Im OSGi Service Compendium wird der Service Tracker-Ansatz beschrieben. Es handelt sich um einen programmatischen Weg auf die Verfügbarkeit eines Services zu warten [108].

Das OSGi Service Compendium definiert einen weiteren Ansatz zum Umgang mit dynamischen Services. Bei den Declarative Services werden die Service-Abhängigkeiten deklarativ beschrieben und von einer ‚Service Component Runtime interpretiert [109].

Neben diesen beiden Ansätzen existieren unter anderem zwei weitere, die nicht in der OSGi Spezifikation festgelegt sind: Spring Dynamic Modules und Apache Felix iPOJO [110, 111]. Beide ähneln dem Declarative Service-Ansatz, erweitern aber die Möglichkeiten deutlich. Sehr interessant für diese Arbeit ist Apache Felix iPOJO, da es bereits auf Android erfolgreich getestet wurde.[112, 113] Daher soll es wie die beiden ersten Ansätze in einem eigenen Abschnitt detaillierter betrachtet werden.

Das Whiteboard-Pattern ist ein weiterer, programmatischer Ansatz, mit der Servicedynamik umzugehen. Er basiert ebenfalls auf dem Service Tracker, weshalb nicht weiter darauf eingegangen werden soll.
Alle hier vorgestellten Ansätze sind untereinander interoperabel, da die über das

Komponentenmodell und deren Mechanismen bereitgestellten Artefakte allesamt Services sind und auf diese Art nach außen hin gleich sind.

4.4.5.1 Service Tracker

Der programmatische Ansatz des Service Tracker erhöht die Dynamik und Flexibilität in der Kopplung zwischen Bundles bzw. Services. Der Laufzeitdynamik wird Rechnung getragen, indem die Problematik der Startreihenfolge elegant und dynamikfördernd gelöst wird.

In den Abschnitten 4.4.4.3 und 4.4.4.5 wurde bereits die manuelle Kopplung über die Service-Registry beschrieben. Hierbei enthielt das anbietende Bundle das Service Interface, welches es exportiert und dessen Implementierung. Das nutzende Bundle importiert dieses öffentliche Service-Interface über OSGi Mechanismen wie Import-Package, um es nutzen zu können, ohne die Implementierung zu kennen. Dies verursacht allerdings eine Startreihenfolgen-Abhängigkeit zwischen diesen Bundles, was bisher mit dem statischen Mittel der Start-Level vor allem in Hinblick auf Dynamik nicht befriedigend gelöst wurde.

Bei der Verwendung des Service Trackers (vgl. Beispiel in Abbildung 4.11) nutzt man zunächst die Möglichkeit, das Service-Interface (org.mnsoft.mybundle) und dessen Implementierung (org.mnsoft.mybundle.impl) in zwei unterschiedlichen Bundles unterzubringen. Dadurch kann das nutzende Bundle schon bei Anwesenheit des Service-Interface-Bundles erfolgreich gestartet werden, auch ohne dass ein den Service implementierendes Bundle installiert ist. Das Service-implementierende

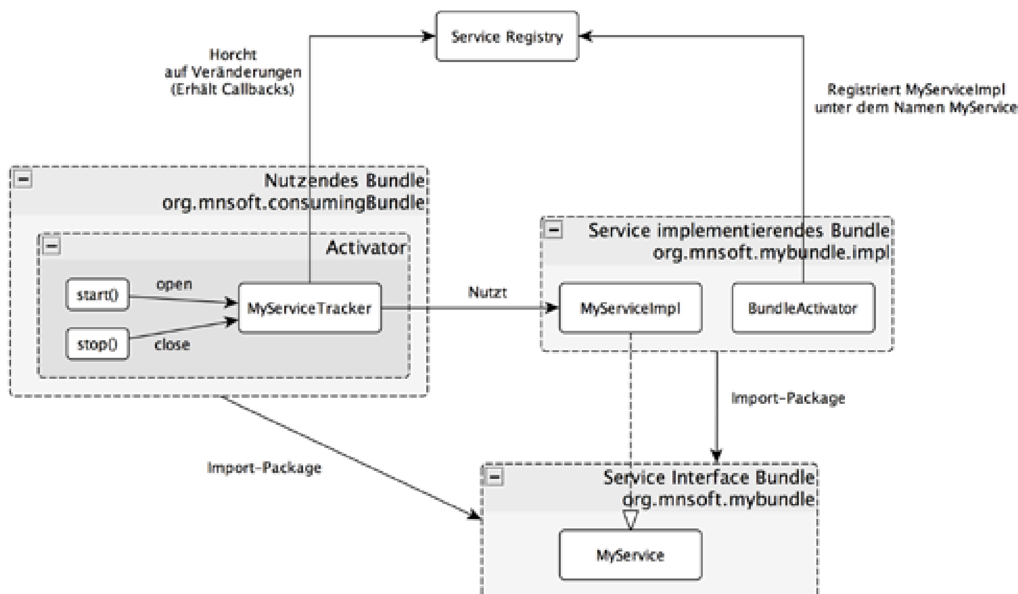


Abbildung 4.11: Einsatz des Service Tracker

Bundle muss hierfür das Package mit dem Service-Interface explizit in seiner Manifest-Datei importieren.

Wird nach Start des nutzenden Bundles von dessen Service Tracker die Anwesenheit des Service-implementierenden Bundles bemerkt, kann im nutzenden Bundle entsprechend reagiert werden und mit der Servicenutzung begonnen werden. Das nutzende Bundle steuert seinen Service Tracker meist in seinem BundleActivator. In der start() und stop() Methode des BundleActivators wird der Service Tracker entsprechend explizit geöffnet bzw. geschlossen.

Sobald das Bundle, welches die Implementierung enthält, installiert, gestartet und sein Service registriert wurde, reagiert der Service Tracker entsprechend und macht den Service für das nutzende Bundle verfügbar. Er ermöglicht den direkten programmatischen Zugriff auf das Service-Objekt (bzw. Liste von Service-Objekten bei mehreren Implementierungen dieses Services) und kann beim Zugriff darauf auch synchron auf die Verfügbarkeit dieses Services warten.

Auf eine Deregistrierung (z.B. bei einer Aktualisierung) kann analog zur Registrierung eine entsprechende Reaktion eingeleitet werden. Da potentiell mehrere Bundles installiert sein können, die den gleichen Service implementieren, kann der zu trackende Service mittels seines Servicenamen und weiterer Metainformationen mittels OSGi-Filter eingeschränkt werden. Alternativ kann auch mit der Liste von Service-Objekten weitergearbeitet werden, um beispielsweise dem Nutzer die Auswahl zu überlassen.

4.4.5.2 Declarative Services

Der im Kapitel 4.4.5.1 beschriebene programmatische Ansatz ist zwar mächtig und verhältnismäßig komfortabel zu verwenden, hat aber auch einige Nachteile. Zum einen benötigt der Ansatz, dass die Bundles instanziiert und registriert sind. Dadurch müssen bei großen Projekten viele Bundles meist gleich beim Anwendungsstart mitgestartet werden, was die Startdauer stark verlängern kann. Da aber nicht in jedem Nutzungsfall alle diese beim Start mitgeladenen Bundles tatsächlich benötigt werden, kommt es zu einem unnötig hohen Speicherverbrauch. Ein weiterer Nachteil ist die Komplexität des Codes in den Bundles, da die Bundle- und Servicedynamik jeweils im Programmcode behandelt werden muss. Der Code wird damit abhängig vom verwendeten Service- und Komponentenmodell, was die Wiederverwendbarkeit einschränkt.

Vor allem die Argumente Startdauer und Speicherverbrauch sind beim Einsatz auf mobilen Plattformen relevant, weshalb es sich lohnt, im Rahmen dieser Arbeit Alternativen zu untersuchen, die bei diesen Problemen Abhilfe schaffen.
Einer dieser Ansätze sind die Declarative Services (DS), die Teil des OSGi Service Compendium sind. Er ist also kein Teil des OSGi Frameworks und damit optional.

Bei DS werden die Service-Abängigkeiten in deklarativer Form beschrieben. Dies verringert die Programmcode-Komplexität, da die Abhängigkeiten ohne potentiell komplexen Service Tracker Code aufgelöst werden können. Das Problem der langen Ladezeiten und der Speicherverbrauch wird umgangen, indem die Bundles standardmäßig nur bei Bedarf gestartet werden, was auch Delayed Components genannt wird. Dieses Verhalten kann in der Komponentenbeschreibung beeinflusst werden.

Um diese Eigenschaften zu erreichen, führt die Declarative Services Spezifikation das Konzept der Service Component ein. Es wird dadurch ein separates Komponentenmodell auf Basis von OSGi eingeführt, welches aber mit dem bereits eingeführten Bundle-Komponentenmodell vom OSGi-Framework kombinierbar ist. Dabei kann ein Bundle mehrere Service Components enthalten. Um DS verwenden zu können, muss eine Implementierungs-Bundle des Declarative-Services-Standardservice im Framework installiert sein.

Eine Service Component besteht aus 2 Teilen: Einer Komponentenklasse, welche die Implementierung enthält und eine XML-basierte Komponentenbeschreibung, welche die Komponente, die angebotenen Services und die Service-Abhängigkeiten deklarativ beschreibt. Service Components haben einen Lebenszyklus, der dem der Bundles ähnlich ist (vgl. Abbildung 4.12).

Zur Servicenutzung muss ein Bundle zwar nach wie vor gestartet sein, es ist

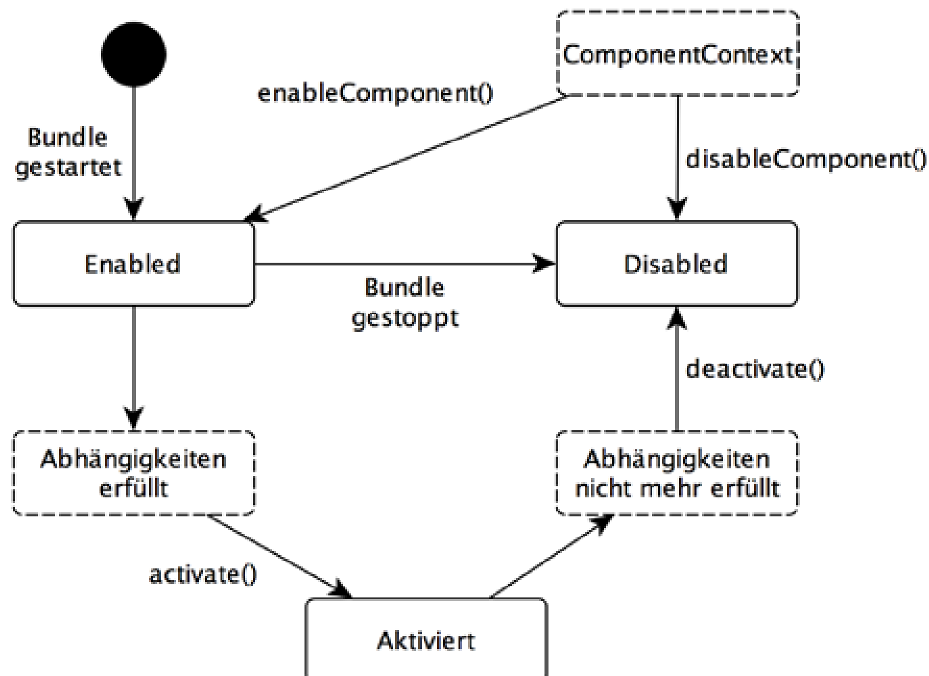


Abbildung 4.12: Lebenszyklus einer Service Component

aber kein BundleActivator mehr erforderlich. Die Instanziierung,Steuerung des Lebenszyklus und Auflösung der Abhängigkeiten der Service Component übernimmt die Service Component Runtime (SCR) anhand der Komponentenbeschreibung. Auch die Service-Registrierung wird von der Service Component Runtime erledigt. Die Zusammenhänge werden in Abbildung 4.13 noch einmal verdeutlicht. Die implementierende Klasse muss lediglich eine Callbackmethode zur Aktivierung der Komponente implementieren, dafür wird keine BundleActivator-Klasse mehr benötigt.

Die Komponentenbeschreibung einer Service-nutzenden Service Component ist in Listing 4.5 dargestellt. Im „Service"-Tag kann spezifiziert werden, welche Service Interfaces diese Service Component bei der Service Registry angemeldet. Unter dem „reference"-Tag werden die Abhängigkeiten zu anderen Bundles angegeben.

Listing 4.5: service-component.xml von Bundle org.mnsoft.mybundle mit Event-Strategie

```
<component name="MyComponent" immediate=false>
 <implementation
   class="org.mnsoft.mybundle.impl.MyServiceImpl"/>
 <service>
   <provide
     interface="org.mnsoft.mybundle.MyService"/>
 </service>
 <reference name="AnotherService"
    interface="org.mnsoft.anotherbundle.AnotherService"
    bind="setAnotherService"
    unbind="unsetAnotherService"
 />
</component>
```

Zur Nutzung von Services aus einer Service Component heraus gibt es verschiedene Strategien. Beispielsweise nutzt die Event-Strategie die bind-Methoden (siehe Listing 4.5), um der Service Component Runtime die Möglichkeit zu geben, bei plötzlicher Verfügbarkeit eines von der Service Component referenzierten Services „AnotherService" diesen bei der nutzenden Service Component „MyComponent" zu setzen. Ist der Service nicht mehr verfügbar, so wird die unbind-Methode analog verwendet.

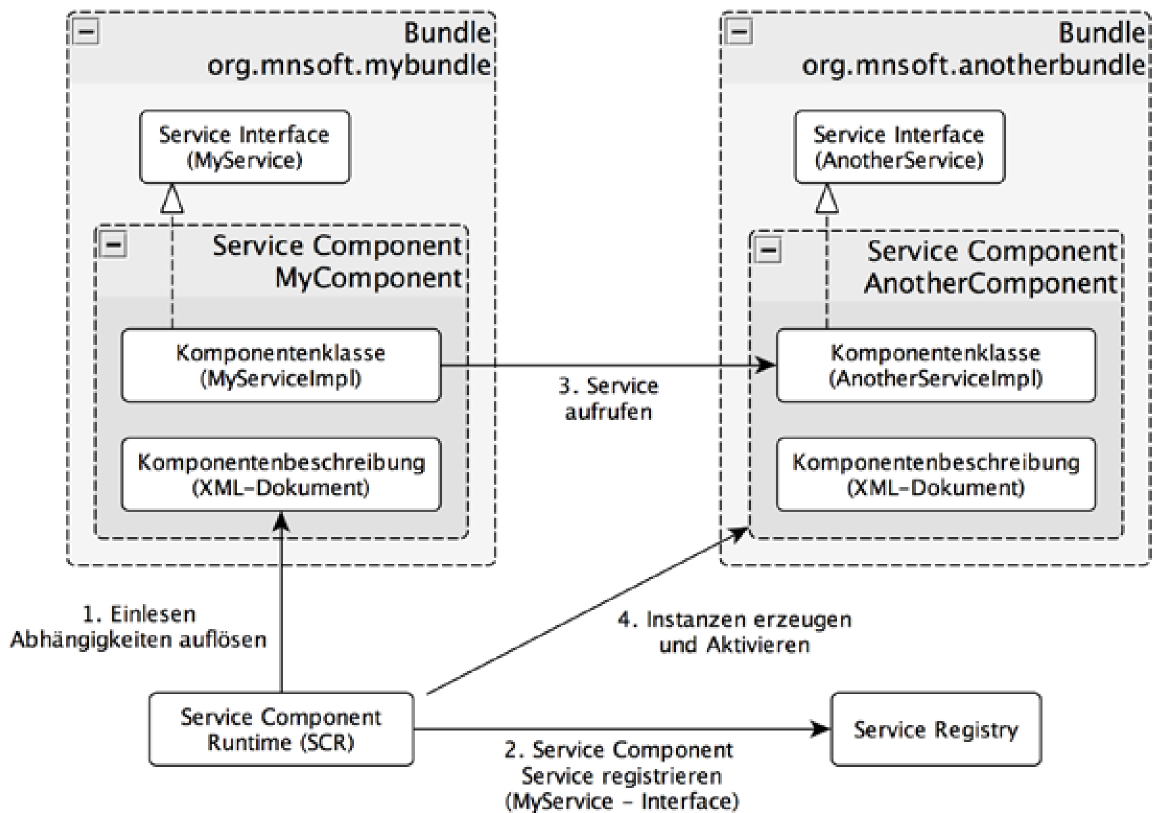


Abbildung 4.13: Zusammenhänge bei Declarative Services

4.4.5.3 Apache Felix iPOJO

Apache Felix iPOJO ist ein Unterprojekt von Apache Felix, einer freien Open Source OSGi-Framework Implementierung. Hauptziel ist es, auf Basis der Declarative Services (DS) ein sauberes, einfaches POJO-orientiertes Programmiermodell für Komponenten zu entwickeln. Während DS selbst bezüglich der Einfachheit noch Defizite aufweist, macht iPOJO die Nutzung von DS deutlich einfacher.

Die Vereinfachung wird durch einen zusätzlichen Build-Schritt, der „bytecode-instrumentation“ erreicht. Ein iPOJO nutzendendes Bundle muss nach dem Packen als Bundle-Jar („Bundling“) in diesem Build-Schritt nachträglich manipuliert werden. Die hierfür benötigte Komponentenbeschreibung kann in Form von XML-Dokumenten oder im Code in Form von iPOJO-Annotations vorliegen.

Durch diese Manipulation kann Mittels der Komponentenbeschreibung und dem ‚Field-injection-Mechanismus ein referenzierter Service genutzt werden, indem ein privates Feld dieses Typs (Service-Interface) angelegt und nach null-Prüfung direkt verwendet wird [114] (siehe Listing 4.6).

Um die notwendigen weiteren Schritte wie Instanziierung und Servicevermittlung kümmert sich iPOJOs Service Component Runtime (iPOJO SCR), welche den gesamten Lebenzyklus der iPOJO-Komponenteninstanzen kontrolliert. iPOJO SCR sichert die Verfügbarkeit eines referenzierten Services, sobald dieser verwendet wird.(Delayed-Components-Ansatz)

Listing 4.6: iPOJO Service Verwendung mit metadata.xml-Beschreibung

```
package org.mnsoft.mybundle;
import org.mnsoft.anotherbundle.AnotherService;
...
public class MyServiceComponentImpl implements MyService {
 private AnotherService m_service;
 public void start() {
  ...
  if (m_service != null) m_service.aMethod();
  ...
 }
 public void stop() { ... }
 public void bindAnotherService(AnotherService s) {
  m_service = s;
  ...
 }
 public void unbindAnotherService(AnotherService s) {
  m_service = null;
  ...
 }
 public void myMethod() { ...}
}
```

Im Beispiel in Listing 4.6 verwendet die Komponente einen Service (AnotherService) und implementiert selbst ein Service Interface (MyService). Die (un)bind-Methoden können optional selbst definiert werden und sind analog zu den Declarative Services.

Für den iPOJO-Build-Schritt bedarf es einer Komponentenbeschreibung. In Listing 4.7 befindet sich für das beschriebene Beispiel die notwendige metadata.xml . Im component-Tag wird die Komponente beschrieben. Der provides-Tag ist eine Kurzform, um alle Interfaces, die diese Komponente implementiert anzubieten. Der requires-Tag beschreibt die Abhängigkeiten und gibt die Namen der jeweiligen (un-) bind-Methoden an (Method Invocation) [115]. Der instance-Tag weist iPOJO an, bei Start des Bundles eine Instanz von der beschriebenen Komponente zu erzeugen.

Listing 4.7: Beispiel für eine iPOJO metadata.xml - Datei

```
<ipojo>
  <component classname="org.mnsoft.mybundle.MyServiceComponentImpl">
    <provides/>
    <requires field="m_service" optional="true">
      <callback type="bind" method="bindAnotherService"/>
      <callback type="unbind" method="unbindAnotherService"/>
    </requires>
  </component>
  <instance component="org.mnsoft.mybundle.MyServiceComponentImpl"/>
</ipojo>
```

Durch das Verwenden von iPOJO Annotations kann auf die Komponentenbeschreibung in der XML-Datei verzichtet werden. Hier unterscheidet sich iPOJO signifikant von den Declarative Services. Implementierende Klassen, Felder und Methoden werden hierbei mittels @-Annotation für den iPOJO-Build-Schritt beschrieben, um anzubietende und referenzierte Services zu kennzeichnen. Dabei wird das POJO-Konzept allerdings etwas untergraben wird.

Listing 4.8 im Anhang A3 zeigt für das vorige Beispiel die Verwendung von iPOJO-Annotations. Völlig kann auf die XML-Datei trotzdem nicht verzichtet werden Die Komponentenbeschreibung kann zwar vollständig mittels Annotation erfolgen, die Erzeugung einer Instanz der Komponente muss aber nachwievor im instance-Tag der XML-Beschreibung deklariert werden.

Es wurde deutlich, dass iPOJO auf den Declarative Services aufsetzt, diese aber erweitert und vor allem die Deklaration als Komponenten und die Servicenutzung zwischen diesen spürbar vereinfacht. Daher ist es sinnvoll, iPOJO der Verwendung von Declarative Services vorzuziehen.

4.4.6 OSGi-Framework Implementierungen und ihre Eignung für Android

In Kapitel 4.4.4 und 4.4.5 wurde die OSGi-Framework Spezifikation und einige Ansätze zur Erleichterung des Umgangs mit dynamischen Services untersucht. Dabei handelte es sich mit Ausnahme von iPOJO lediglich um Spezifikationen. Die konkrete Implementierung muss dabei von Softwareunternehmen bzw. Open Source-Communities übernommen werden. Tatsächlich sind eine Reihe kommerzieller und freier OSGi-Implementierungen in den letzten Jahren entstanden und gewachsen.

Dies ist eine (nicht vollständige) Liste bekannter freier Implementierungen der OSGi-Framework-Spezifikation (Core-Specification)

- Eclipse Equinox [116]
- Apache Felix [117]
- ProSyst mBedded Server (mBS) Equinox Edition [118]
- Makewave Knopflerfish [119]

Einige Projekte liefern auch Implementierungen einiger Standardservices und fügen teilweise weitere Services hinzu. Da von allen Implementierungen die OSGi-Spezifikationen eingehalten werden müssen, sind die Frameworks und die Standardservices beliebig miteinander kombinierbar. So ist Apache Felix iPOJO nicht nur auf Apache Felix, sondern prinzipiell auch auf allen anderen Framework-Implementierungen lauffähig. Lediglich bei Einsatz von Implementierungs-spezifischen Services oder bei Besonderheiten der Java Virtual Machine (JVM) treten folgerichtig Schwierigkeiten auf.

Durch die Einhaltung der Standards sollte es daher nicht sehr relevant sein, welche OSGi Implementierung für diese Arbeit ausgewählt wird. Die Praxis zeigt aber, dass längst nicht jede Implementierung auf Android problemfrei lauffähig ist. Dies hat vorwiegend mit einigen Besonderheiten von Androids Virtual Machine DalvikVM zu tun.

So ist Equinox zur Zeit noch nicht erfolgreich unter Android verfügbar, was bedauerlich ist, da unter der Eclipse IDE die Entwicklung von OSGi Bundles sehr komfortabel integriert ist. Zwar wurde Equinox 2008 angeblich erfolgreich auf Android portiert, jedoch sind seitdem keine weiteren Informationen darüber veröffentlicht worden, was darauf schließen lässt, dass sich dies im Zeitrahmen dieser Arbeit nicht mehr ändern wird [120]. Gleiches trifft auch auf ProSyst mBedded Server Equinox Edition zu.

ProSyst hat es aber geschafft eine mBedded Server Android Edition auf die Android-Plattform ab Version 1.5 zu bringen. Diese Edition steht zum freien Download, ist aber nicht quelloffen, da sie auf der mBedded Server Professional Edition basiert [121].

Bisher waren noch keine Anzeichen erkennbar, dass Knopflerfish auf Android uneingeschränkt lauffähig ist. Makewave hat diesbezüglich noch keine Auskünfte gegeben.

Das freie Open Source Toplevel-Projekt Apache Felix ist seit Version 2.0.0 auf Endgeräten mit Android 1.5 ohne Anpassungen uneingeschränkt lauffähig. Das Framework kann in eine Android-Anwendung eingebettet werden, indem diese die Felix Jar-Datei als Bibliothek einbindet und in ihrer Start-Activity eine Instanz von Felix erzeugt und startet [113]. Dieses Vorgehen erleichtert das Deployment einer

Android-Anwendung, die aus OSGi Bundles bestehen soll, da so die Existenz einer OSGi-Framework-Implementierung bei Installation (auch über den Android Market) sichergestellt werden kann.

ProSysts Lösung geht hier einen anderen Weg und verwendet eine zentrale OSGi-Framework-Installation, die alle installierten OSGi-basierten Anwendungen teilen. Wie dies trotz der Besonderheiten von Android erreicht werden konnte, ist aufgrund des geschlossenen Quellcodes noch unklar.

Ein großes Problem beim Einsatz von Felix auf Android war die Classloader-Problematik. OSGi-Frameworks machen für gewöhnlich davon reichlich Gebrauch und bis Android 1.1 gab es keine offizielle Methode zum dynamischen Nachladen von Klassen. In Felix wurde dies dank Clement Escoffier (bekannt durch iPOJO) gelöst, indem eine ab Android 1.5 offiziell verfügbare Funktion genutzt wird [122]. Verbleibende Probleme konnten im Rahmen der Recherche zu dieser Arbeit auf dem users@felix.apache.org-Verteiler Dank der Hilfe von Richard S. Hall und Karl Pauls gelöst werden. An dieser Stelle möchte ich mich nochmals bei allen genannten Felix-Entwicklern bedanken und auch aufzeigen, wie wertvoll eine aktive Entwickler-Community ist.

Aufgrund der genannten Eigenschaften und den Vorzügen der aktiven Unterstützung durch erfahrene OSGi-Framework Entwickler ist die Framework-Implementierung Apache Felix gegenüber der ProSyst-Lösung zu bevorzugen. Die Unterstützung ist vor allem deshalb wichtig, da mit der Nutzung von OSGi unter Android zur Zeit noch Neuland betreten wird, wodurch während der prototypischen Implementierung durchaus noch weitere, unerwartete Probleme auftauchen können.

Da die bereits erwähnten Probleme innerhalb weniger Tage durch die Felix-Entwickler gelöst wurden, ist anzunehmen, dass im Fall von Problemen mit Unterstützung gerechnet werden kann. Bei ProSyst ist dieser Sachverhalt fraglich, da nicht vollständig klar ist welche (kommerziellen) Ziele ProSyst mit dieser Veröffentlichung beabsichtigt und wieviel Energie zukünftig darin investiert werden wird. Die fehlende Quelloffenheit der ProSyst mBS Android Edition und ihrer Basis ist ein weiteres Argument für diese Befürchtung.

4.5 Fazit

Das State-of-the-art Kapitel analysierte den aktuellen Stand der technologischen Möglichkeiten bezüglich der verschiedenen Aspekte dieser Arbeit. Zuerst wurden die Anbieter von frei verfügbaren WebGIS Systemen (MapProvider) untersucht und anhand ihrer Tauglichkeit für den programmatischen Einsatz unter Android bewertet.

Die beiden untersuchten, repräsentativen Vertreter „Google Maps“ und das Open Source-Projekt „OpenStreetMap“ sind sich in vielen Gebieten ebenbürtig.

Bei der Integration auf Android schnitt Google Maps erwartungsgemäß besser ab, was OpenStreetMap aber durch die Weiterentwicklung des OSMDroid Projektes in absehbarer Zeit aufholen könnte. Die umfangreichen Möglichkeiten, das Kartenmaterial auf einer lokalen Speicherkarte zu cachen, macht OpenStreetMap besonders interessant. Vor allem die Performance der Kartendarstellung kann gesteigert und die mobile Internetverbindung deutlich entlastet werden.

Trotzdem muss für die Sicherung des Erfolgs des Prototypen eine konservative Entscheidung zugunsten von Google Maps getroffen werden. Durch die fortgeschrittene Implementierung existiert hier eine nahezu fehlerfreie Möglichkeit, das Google-WebGIS in eine native Android-Anwendung zu integrieren. Die Konzeption soll eine Möglichkeit in Betracht ziehen, die Kartenanbieter flexibel auszutauschen.

Die Alternative zur programmatischen, nativen Integration in eine Android-Anwendung ist die Anzeige einer WebGIS-basierten Webanwendung im Androidbrowser. Zwar wurden Gründe gegen eine Lösung als Webanwendung dargelegt, dennoch sind Realisierungen dieser Art weit verbreitet. Die Untersuchung dieser Ansätze sollte neue Erkenntnisse und Ideen zum Thema der Informationsintegration in digitale Karten hervorbringen.

Zwar bestätigten sich zunächst die Annahmen über Webanwendungen auf Android im Allgemeinen, bezüglich WebGIS dann im Speziellen hinsichtlich von Defiziten bei Interaktion und Performance. Jedoch wurden darüber hinaus diese serverseitigen Ansätze auch auf die Möglichkeit des programmatischen Zugriffs untersucht.

Ein interessanter Punkt in dieser Untersuchung war Googles „My Maps Editor for Android“, welcher nicht quelloffen ist. Dieser zeigt die Möglichkeit der nativen Integration der serverseitigen Google-eigenen Ansätze: der Kartenzugriff erfolgt nativ in der Android-Anwendung, die Dienstelogik wird online im (ebenfalls über Webbrowser zugänglichen) My Maps Bereich gehalten.

Da aber auch dieser Ansatz zu unflexibel und Google-abhängig ist, musste hingenommen werden, dass diese Art von WebGIS-basierten Webanwendungen mittelfristig für Android und auch für andere mobile Plattformen untauglich ist.

Außerdem wurden vorhandene Alternativen bezüglich des Beispielszenarios untersucht. Das Ergebnis war, dass es sich dabei um unflexible, monolithische, auf das jeweilige Aufgabenfeld speziell zugeschnittene Anwendungen handelt. Das kartenorientierte Interaktionsparadigma steht bei ihnen nicht im Vordergrund, sondern spielt nur die Rolle einer Zusatzinformation.

Im Kapitel 4.4 wurde zunächst festgestellt, dass es zum aktuellen Zeitpunkt für JavaSE keine geeignete Alternative zur OSGi Service Plattform gibt. Die Wahl von Java hat ihre Ursache in der Wiederverwendbarkeit des erarbeiteten Wissens: Sowohl die angestrebte, clientseitige (Android) als auch serverseitige Lösungen (nahezu beliebiges Server-OS) sind mit Java und OSGi erstellbar. Die von der OSGi-Technologie abgedeckten Möglichkeiten bezüglich der nicht-funktionalen Anforderungen Flexibilität (NF_1), Austauschbarkeit (NF_5), Erweiterbarkeit (NF_2), Wartbarkeit (NF_4) und Modularität (NF_2) sind der zentrale Enabling Factor zur Konzeption einer neuen Webservice-Integrationsmöglichkeit für digitale Karten mit den gleichen nicht-funktionalen Anforderungen.

Die tiefgreifende Untersuchung von OSGi zeigte Wege und Möglichkeiten auf, wie die Ziele dieser Arbeit mithilfe von OSGi hervorragend erreicht werden können. Abschließend wurden verschiedene Implementierungen des OSGi-Frameworks auf ihre Tauglichkeit für Android für den Fall einer clientseitigen Lösung untersucht.

Die OSGi-Framework-Implementierungen von ProSyst und der Apache Foundation sind hierbei bereits erprobt, wobei die Apache-Lösung durch ihre Quelloffenheit, Leichtgewichtigkeit, flexible Systemintegration und aktive Entwicklergemeinde beeindruckt und daher ausgewählt wurde.

Im folgenden Kapitel 5 sollen nun die Erkenntnisse über die einzelnen Aspekte der Aufgabenstellung gebündelt und zu einem großen Gesamtkonzept verwoben werden.

5 Konzeption

An dieser Stelle soll auf Basis der Erkenntnisse der vorhergehenden Kapitel eine Lösung für die Aufgabenstellung gefunden werden. Zunächst müssen zwei grundlegende Konzepte eingeführt werden, die unabhängig von der konkreten Systemarchitektur sind.

Das erste Grundkonzept entwickelt das im Kapitel 4.4 erwähnte „Allgemeine PlugIn Konzept“ weiter. Dieses beschreibt die Kapselung der MapService-Integration und potentiell auch der MapProvider-Integration in separate, unabhängige PlugIns, die eine Host-Anwendung erweitern, um die gewünschte Gesamtfunktionalität zu erreichen. Hierbei wird die Anforderung der Flexibilität und Generik der zu entwickelnden Systemarchitektur adressiert. Der Dynamikaspekt für diese PlugIns wurde in Kapitel 4.4 ausführlich behandelt.

Hinsichtlich der Heterogenität der einzubindenden MapServices wird ein Konzept benötigt, welches die MapServices strukturiert und ihre Nutzung vereinheitlicht. In Kapitel 5.1.2 wird hierzu das Dienstklassenkonzept als zweites Grundkonzept eingeführt.

Auf Grund der außerordentlich guten Eignung von OSGi bezüglich der geforderten Eigenschaften werden vorwiegend Architekturkonzepte betrachtet, die den Einsatz von OSGi bzw. einem vergleichbaren dynamischen Modulsystem vorsehen. Die Wahl von OSGi in Kapitel 4.4 hatte die Ursache in der Verwendbarkeit sowohl für client- als auch serverorientierte Architekturen, wodurch unnötige Einschränkungen des Konzeptraums vermieden werden konnten. Da mit Apache Felix eine OSGi-Framework-Implementierung vorliegt, die unter Android funktioniert, werden auch Architekturmöglichkeiten betrachtet, welche das dynamische Modulsystem auf Clientseite unterbringen. Im Kapitel 5.2 sollen daher verschiedene client- und serverseitige OSGi-Lösungen mit konventionelleren Ansätzen verglichen werden, welche in ihrer Architektur auf dynamische Webanwendungen zurückgreifen.

Abschließend wird das am besten geeignete Konzept ausgewählt und zu einem detaillierten Konzept verfeinert. Dieses muss sich den im Laufe der vorherigen Kapitel aufgetretenen Anforderungen stellen. Das ausgearbeitete Detailkonzept wird dann im Kapitel 6 prototypisch als proof-of-concept umgesetzt, um die Erfüllbarkeit der Anforderungen durch das Konzept zu validieren.

5.1 Grundkonzepte

In diesem Kapitel werden die Architektur-unabhängigen, grundlegenden Konzepte „PlugIn-Konzept" und das „Dienstklassen-Konzept" näher beschrieben.

5.1.1 PlugIn Konzept

Im Kapitel 4.4.2 wurde bereits das „Allgemeine PlugIn Konzept" beschrieben. Dabei ging es um die Grundidee, die Einbindung der verschiedenen MapServices in jeweilige, für den konkreten Dienst entwickelte PlugIns zu kapseln, um so Austauschbarkeit, Erweiterbarkeit und Flexibilität zu sichern.

Das PlugIn ist dabei ein lose gekoppeltes Modul, das von der Hauptanwendung genutzt wird, um ihre Funktionalität zu erweitern bzw. überhaupt erst zu erzeugen. Im Kapitel 4.4 wurde die OSGi-Technologie vorgestellt, die beispielsweise auch das PlugIn-Konzept der Eclipse IDE ermöglicht und bei der diese PlugIn-Modularisierung unter dem Begriff „Bundle" eingeführt wurde. Diese Technologie bringt einige bewährte Konzepte mit sich, die in dieser Konzeption dankbare Wiederverwendung finden und somit gleichzeitig die Möglichkeit der Umsetzung des Konzepts mit Hilfe von OSGi sichern. Indem die PlugIns durch OSGi-Bundles repräsentiert werden, erben sie all die im Kapitel 4.4 angeführten positiven Eigenschaften, welche die Verwendung von OSGi bzw. eines vergleichbaren dynamischen Modularisierungskonzepts mit sich bringt.

Das PlugIn-Konzept kann jedoch über die beschriebene Kapselung der Zugriffslogik auf die MapServices hinaus erweitert werden. So ist es denkbar, dass die Austauschbarkeit des MapProviders ebenfalls durch Kapselung in MapProvider-Bundles erreicht werden kann. Diese können die dafür benötigten Bibliotheken in ihrem Bundle-Container beinhalten, um so die Abhängigkeit von der Installationsumgebung zu reduzieren.

Dem selben Prinzip folgend kann auch die Visualisierung der MapService-Informationen in eigene Bundles gekapselt werden. Diese könnten entweder allgemeine Visualisierungen von „typischen" MapServices sein, oder spezifisch für bestimmte MapService-Bundle zugeschnitten sein. In letzterem Fall treten dann Abhängigkeiten vom jeweiligen MapService-Bundle auf. Auf diesem Weg können Datenzugriff bzw. Datenhaltung und Visualisierung sauber getrennt werden, was wiederum der Wiederverwendbarkeit zugute kommt.

5.1.2 Dienstklassen-Konzept

Um mit beliebigen, heterogenen MapServices umgehen zu können, muss der generischen Hauptanwendung eine wohldefinierte Beschreibung der verschiedenen „Typen" bzw. „Klassen" von MapServices bekannt sein. Die Hauptanwendung bzw. das nutzende PlugIn muss daher die Schnittstellen zur Entwicklungszeit kennen, um mit den Services-implementierenden PlugIns arbeiten zu können.

Um der Vielfalt der MapServices Herr zu werden, ist es sinnvoll, sie bzw. ihre Funktionalität in verschiedene Gruppen einzuteilen, die von nun an „Dienstklassen" genannt werden sollen. Das Dienstklassen-Konzept klassifiziert die MapServices bzw. die Aspekte ihrer Funktionalität bezüglich ihrer darzustellenden Informationen. Die Darstellung dieser Informationen setzt für die jeweilige Dienstklasse typische Abfragen voraus. Da diese Abfragen aber bei den unterschiedlichen Diensten unterschiedlich heißen, parametrisiert sind und verschiedene Antwortformate haben, ist es ein Ziel dieser Arbeit, diese Heterogenität auf eine jeweilige, für diese Dienstklasse typische, vereinheitlichte Abfragemöglichkeit zu reduzieren.

Hier greift das Dienstklassen-Konzept: ein MapService oder eine Teilfunktion eines MapServices erfüllt eine Dienstklasse und wird gemäß seiner API bezüglich dieser Dienstklasseninformationen abgefragt. Die Diversität der MapServices wird in der Implementierung des jeweiligen MapService-PlugIns weggekapselt. Nach außen hin ist dieses PlugIn gemäß seiner Dienstklassen-Spezifikation verwendbar und verbirgt die zugrunde liegene Komplexität des Informationsabrufs vor dem Nutzer des PlugIns.

Hier wird deutlich, dass das Dienstklassen-Konzept auf die Service-Interfaces von OSGi abbildet. Die Dienstklassen nehmen die Kategorisierung vor und erzeugen eine semantische Beschreibung eines MapService-Typus. Diese Beschreibung erklärt die Beschaffenheit der abzurufenden Information. Ihre Umsetzung - das Service-Interface - spezifiziert entsprechend dieser semantischen Beschreibung eine API, die ein MapService-Bundle (PlugIn) implementieren muss, um die jeweilige Dienstklasse zu erfüllen. Um die Hauptanwendung flexibel zu halten, können diese Service-Interfaces selbst in Interface-Bundles gekapselt werden.

Die Granularisierung der MapService-Funktionalitäten in einzelne PlugIns steigert die Wiederverwendbarkeit und verhindert Redundanzen. So kann eine konkrete Diensterbringung eine Komposition aus Implementierungen verschiedener Dienstklassen sein. Beispielsweise könnte eine Dienstklasse eine Liste von POI in Form von Namen und Koordinaten liefern, eine andere die Möglichkeit der Abfrage von weiteren Informationen zu diesem POI. Dabei kann es durchaus sein, dass beide Informationen letzen Endes beim selben MapService abgefragt werden, jedoch erleichtert diese Granularisierungsstufe die Wiederverwendung der einzelnen Teilfunktionen dieses MapServices. Zusammen komponiert ergibt sich im Beispiel eine POI-darstellende Diensterbringung, die sich aus zwei einzelnen

PlugIns zweier Dienstklassen zusammensetzt. Dieses Vorgehen produziert verständlicherweise eine gewisse Abhängigkeit zwischen diesen beiden PlugIns, da beide zur Diensterbringung notwendig sind.

Das eben angeführte Beispiel betrachtete vorwiegend den Datenzugriff mittels Dienstklassen-implementierender PlugIns. Zur vollständigen Diensterbringung müssen die abgerufenen Informationen auch auf dem Endgerät visualisiert werden. Hierzu könnten standardisierte oder aber spezifisch für diesen MapService entwickelte Dienstklassen-GUI-PlugIns (zum Beispiel zur Anzeige der Informationen zu einem POI in der Karte) verwendet werden. Die Komposition aus einem oder mehreren MapService- und MapService-GUI- PlugIns ergibt dann eine vollständige Diensterbringung auf dem Endgerät. Die Trennung von Daten und GUI in unterschiedliche PlugIns ist aber nicht unbedingt notwendig und nicht in jedem Fall sinnvoll.

Das Dienstklassen-Konzept ist wie das PlugIn-Konzept durch seine Allgemeinheit sowohl im Serverkontext (Webanwendungsähnliche Ansätze) als auch im Clientkontext auf dem mobilen Endgerät (Android) anwendbar. Technisch wird dies vor allem dadurch erreicht, dass die dynamische Modularisierungstechnologie OSGi in beiden Fällen zur Verfügung steht und diese Art flexibler, modularer Architektur ermöglicht. Das Kapitel 5.2 diskutiert die verschiedenen Möglichkeiten der Positionierung dieser Technologie und dem PlugIn-Konzept und damit auch die Position von Dynamik, Austauschbarkeit und Flexibilität.

Die statische Strukturierung der MapServices in Dienstklassen hat allerdings eine Schwachstelle für den Fall der Erweiterung eines MapServices um eine zuvor nicht bekannte, zusätzliche Funktion. Das Einordnen eines MapService bzw. einer MapService-Teilfunktionalität in eine Dienstklasse ist generell ein typisches Klassifizierungsproblem wie man es aus der Neuroinformatik oder dem Clustering in der KI kennt. Wenn aber eine neue Zusatzfunktion so häufg ist, das sie zum Quasi-Standard eines MapServices gehört, welcher eine vorhandene Dienstklasse erfüllt und diese Zusatzfunktion nützlich bei der Erfüllung der Aufgabe dieser Dienstklasse ist, so muss die Dienstklasse angepasst werden. Im Normalfall genügt aber die Spezifikation einer neuen Dienstklasse.

Im Kapitel 5.3 werden konkrete Beispiele für Dienstklassen definiert. Die dort angeführten Dienstklassen sind die grundlegenden, häufig im MapService-Kontext auftretenden Funktionskomplexe. Selbstverständlich kann dieses Konzept um weitere Dienstklassen erweitert werden. Die angeführten Beispiele sollen der Orientierung und dem Verständnis dienen.

5.2 Architekturkonzepte

In diesem Kapitel sollen einige Architekturkonzepte vorgestellt werden. Zunächst werden zwei modulare Ansätze für WebGIS-Webanwendungen beschrieben, die den klassischen, vorhandenen Ansätzen ähneln, aber auf die Modularisierung fokussieren. Dem gegenüber werden drei Ansätze gestellt, die eine Android-Anwendung vorsehen, um somit den Schwachstellen des Android-Browsers zu begegnen und die Möglichkeiten des Endgerätes auszuschöpfen. Abschließend soll ein Konzept ausgewählt werden, das im Kapitel 5.3 dann vertieft wird.

Die dargestellten Konzepte weichen insofern vom eigentlich angedachten WebGIS-Konzept der OGC ab, indem Kartengenerierung und Informationsintegration nicht ausschließlich serverseitig in einem Netz von Webservices statt findet, sondern dass eine Instanz außerhalb der OGC-Welt die Kartendienste (die selbst wiederum OGC-konform sein können) und die Informationsdienste (MapServices) zusammenführt.

Die MapServices selbst können auch OGC-konform sein, müssen es aber durch die Losgelöstheit dieser Instanz aus der OGC-Welt nicht unbedingt sein. Dadurch wird es grundlegend möglich, beliebige MapServices einzubinden. Wo diese integrierende Instanz positioniert wird, ist der Hauptunterschied der folgenden Architekturkonzepte.

5.2.1 Mapplet Webanwendung

Die Google Mapplets wurden in Kapitel 4.2.3 bereits besprochen. Die Abbildung 5.1 stellt die Einbindung von beliebigen MapServices von einem Mapplet aus dar.

Die Modularisierung findet auf Mapplet-Ebene statt, kommt also gänzlich ohne OSGi aus. Je einzubindenden MapService wird jeweils ein entsprechendes Mapplet erstellt, welches Datenzugriff und Visualisierung der Informationen in der Google Maps Karte vornimmt. Die Rahmenwebanwendung Google Maps bindet die Mapplets in die Seite und ins Kartenmaterial ein. Hosting und die Rahmenbedingungen werden durch Google gestellt.

Nachteile dieser Lösung sind seine Zentralität und die damit verbundenen Nachteile, die Abhängigkeit von Google und vor allem, dass die Darstellung auf dem Endgerät im Android-Browser erfolgen müsste, was sich nach eigenen Tests als kaum praktikabel herausgestellt hat. Außerdem gibt der Mappletentwickler MapService-Zugriffsdetails in Mapplet-Skript-Form an Google preis, was von Unternehmen teilweise mit Skepsis gesehen wird.

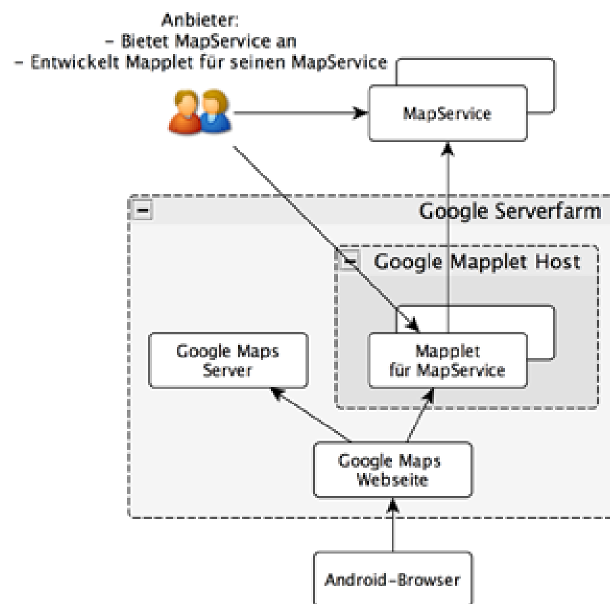


Abbildung 5.1: Mapplet-Webanwendung

Es ist aber denkbar, dass der My Maps Editor for Android, die Android Maps-Anwendung selbst oder eine speziell dafür vorgesehene Software einen komfortableren und performanteren Zugriff auf Mapplets ermöglichen wird. Endgerätefunktionen wie GPS könnten im Gegensatz zum Browser bequem genutzt werden. Zum aktuellen Zeitpunkt (Stand Sommer 2009) liegt eine solche Möglichkeit noch nicht vor.

5.2.2 Webanwendung auf OSGi Basis

Die klassische Webanwendung, wie sie in diesem Themenbereich oft vorkommt, kann durch Einsatz von OSGi auf Serverseite Modularisierung, Austauschbarkeit und Flexibilität erfahren. Der Aufbau ist in Abbildung 5.2 schematisch beschrieben.

So könnte eine Webanwendung auf MapService-Bundles zugreifen, um die Zugriffsdetails zu den unterschiedlichen, zugrundeliegenden MapServices (ob OGC-konform oder nicht) nicht kennen zu müssen.

Allerdings ist das Endergebnis nach außen wieder eine Webseite, die im Android-Browser dargestellt würde. Es ist zwar denkbar, die Webseite entsprechend der Verwendung unter Android zu optimieren, dennoch bestünde kein Zugriff auf Gerätefunktionen und man wäre durch die eingeschränkten Interaktionsmöglichkeiten des Browsers limitiert.

Ein weiterer Nachteil ist die Zentralität, mit ihren Konsequenzen bezüglich der Ausfallsicherheit, Performance und Skalierbarkeit. Im Fall der zuvor beschriebenen

Google Mapplets wirkt die Google Server Farm diesem Nachteil mit dem klassischen Mittel der Redundanz und Lastenverteilung entgegen. Beim Hosting der Serveranwendung in diesem Konzept müssten also bei entsprechender erwarteter Auslastung ähnliche, sehr kostenintensive Mittel in Anspruch genommen werden.

Der Vorteil dieser Lösung (wie auch bei der Mapplet-Lösung) wäre die Auslagerung der Rechenlast der Integration und des MapService-Zugriffs vom mobilen Endgerät hin zum Server. Dadurch könnte der Internet-Datenverkehr auf die Kommunikation mit dem zentralen Server reduziert werden, was wiederum Optimierungspotential bietet. Dies kann selbst unter optimalen Bedingungen allerdings die Nachteile des Browserzugriffes nicht ausgleichen.

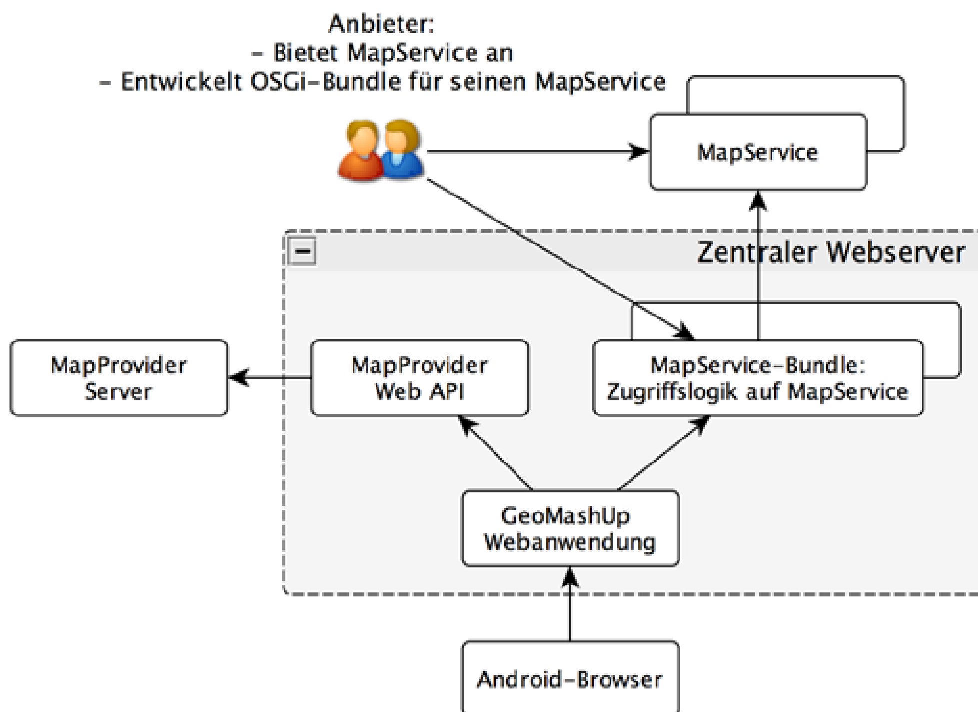


Abbildung 5.2: Webanwendung auf OSGi Basis

5.2.3 OSGi-basierter Adapterserver

Das Adapterserver-Konzept in Abbildung 5.3 sieht eine Adaptierung der verschiedenartigen MapServices (im Server repräsentiert durch registrierte OSGi-Bundles) auf eine zentrale, wohldefinierte, standardisierte Schnittstelle vor. Die Betrachtungs-Anwendung für Android (MapService-Viewer) benutzt diese, um die Informationen des MapServices auf einer nativ eingebundenen MapProvider-Karte darzustellen (vergleichbar mit dem My Maps Editor).

Da es eine native Anwendung ist, können alle vom Android-Softwarestack zur

Verfügung gestellten Hardwarefeatures wie z.B. GPS und Android GUI-Elemente verwendet werden, die eine intuitivere und einfachere Interaktionmöglichkeit bieten als Webseiten-Elemente. Der MapService-Viewer könnte damit zur aktuellen GPS-Position oder zum aktuell angezeigten Kartenausschnitt eine Liste der hierfür sinnvoll darstellbaren MapServices abfragen. Nach Auswahl aus dieser Liste durch den Nutzer werden die gewünschten Informationen in der Karte integriert dargestellt.

Dieses Konzept hat allerdings einige Nachteile. Der erste ist die Standardschnittstelle des Adapterservers: Aufgrund der Vielfältigkeit der MapServices wird es schwierig werden, diese so allgemein zu formulieren, dass alle möglichen Dienste unterstützt werden können. Diese einzelne Schnittstelle würde alle bis dahin definierten Dienstklassen abdecken müssen und dadurch unübersichtlich werden.

Der zweite Nachteil ist die Android-Anwendung, die recht statisch auf diese Standardschnittstelle zugeschnitten wäre. Bei einer Änderung der Schnittstelle z.B. durch neue Dienstklassen müsste auch die Android-Anwendung aktualisiert werden.

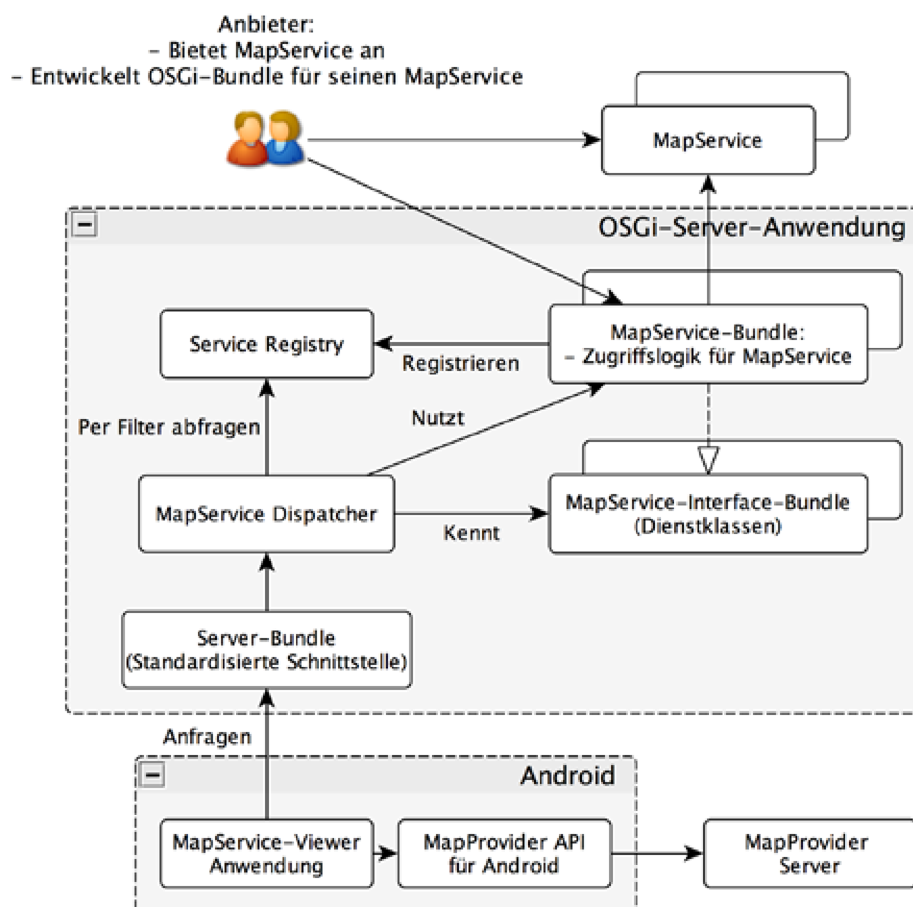


Abbildung 5.3: OSGi-basierter Adapterserver

Der dritte Nachteil ist die Zentralität. Eine zentrale Adapterkomponente wäre die Vermittlungszentrale zwischen Android-Anwendung und den MapServices. Dadurch liegt die Last größtenteils auf dem Adapterserver, wodurch dieser zum Flaschenhals wird, was zeigt, dass das Konzept schlecht skaliert.

Der vierte Nachteil dieses Ansatzes ist, dass der MapService-Viewer fest auf einen MapProvider zugeschnitten ist.

5.2.4 D-OSGi basierter Ansatz

Ein sehr neuer Ansatz aus der OSGi-Welt, dessen Spezifikation noch nicht vollständig abgeschlossen ist, eröffnet eine weitere interessante Alternative. Distributed OSGi (D-OSGi) ist Teil des aktuellen Entwurfes der OSGi Version R4.2 [123]. Bei diesem Ansatz erfolgt die Kopplung zwischen Client (OSGi-Android-Anwendung) und

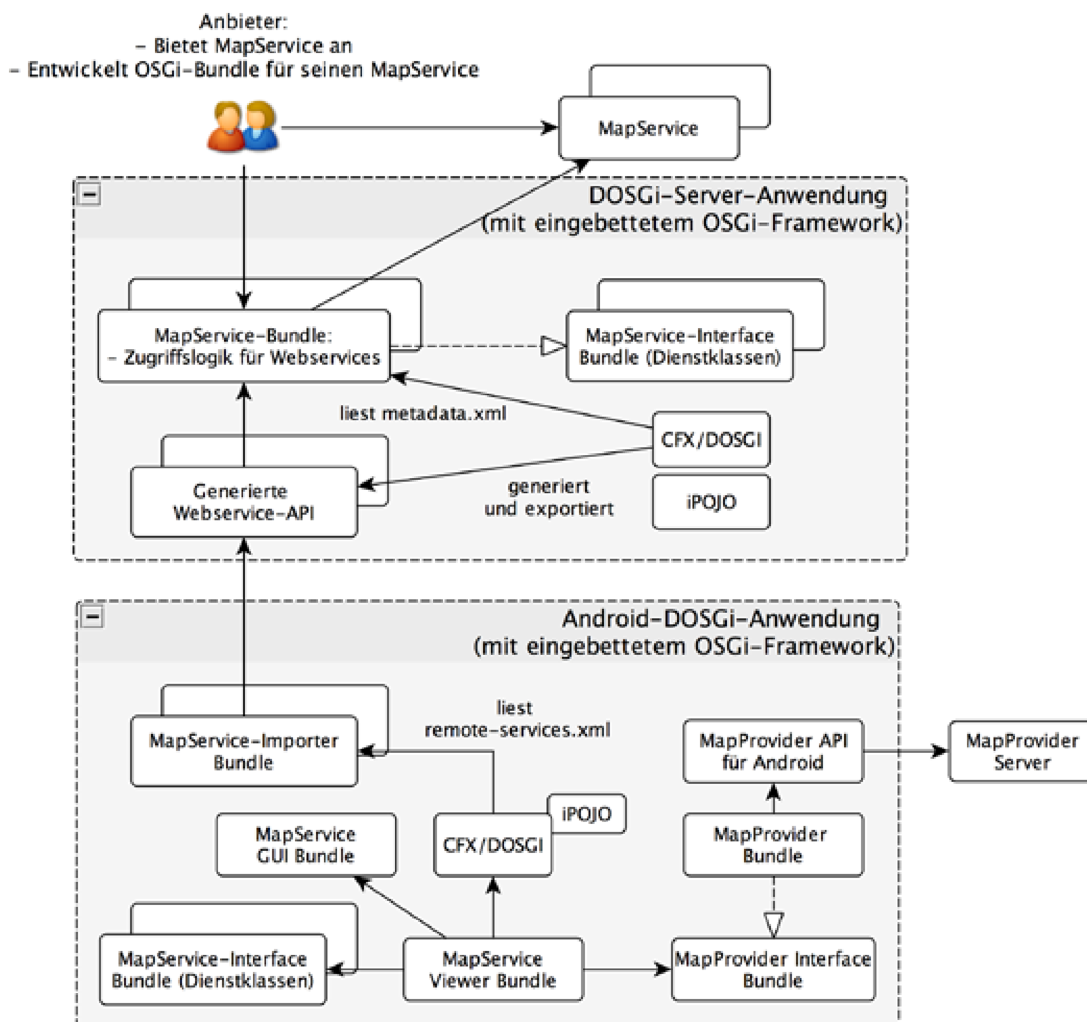


Abbildung 5.4: D-OSGi-basierter Ansatz

Server (OSGi-Server-Anwendung) über die D-OSGi Technologie.

In Abbildung 5.4 ist eine beispielhafte Umsetzung mit D-OSGi auf Basis von Apache Felix iPOJO und CFX dargestellt [124]. Auf Server und Client läuft das OSGi Framework. In diesem sind auf beiden Seiten die Bundles iPOJO, CFX sowie die MapService-Interface-Bundles installiert. Im Server beschreiben und propagieren die MapService-Bundles mittels ihrer metadata.xml ihre Dienste und binden diese an eine Webadresse.

Im Client existiert für jedes dieser MapService-Bundles ein MapService-Importer-Bundle, welches bis auf die remote-services.xml leer ist. Diese Datei beschreibt, welcher Service unter welcher Webadresse erreichbar ist. Die Verwendung des entfernten OSGi-Services geschieht im nutzenden Bundle direkt wie bei normalen OSGi Anwendungen. iPOJO übernimmt im Beispiel die Dienstevermittlung und CFX die Weiterleitung an das entfernte Bundle, welches die Anfragen über eine generierte Webservice-API erhält. Die MapService-GUI Bundle müssen wegen der Verwendung von Android-eigenen UI-Elementen allerdings lokal im Client vorliegen.

Der clientseitige Zugriff auf den MapProvider zur Darstellung der Karte kann dank OSGi in MapProvider-Bundles gekapselt werden, was die Austauschbarkeit des MapProviders ermöglicht.

Mit D-OSGi kann über vorhandene OSGi-Paradigmen Framework-übergreifend zwischen Bundles kommuniziert werden und ihre Services genutzt werden. Der Ansatz ist ähnlich zum Adapter-Server-Konzept und genießt auch dessen Vorteile, bietet aber vor allem Flexibilität auf Clientseite und eine einfachere Kopplung zwischen Client und Server, was der Entwicklungszeit zu Gute kommen könnte.

Es gibt zwar schon eine Referenzimplementierung, jedoch ist die Standardisierung noch nicht so weit abgeschlossen, dass diese Diplomarbeit auf diese Grundlage gestellt werden sollte. Wie auch der Adapter-Server-Ansatz ist das Problem der Zentralität (Flaschenhals) gleichermaßen vorhanden.

5.2.5 OSGi-basierte Android-Anwendung

Bei diesem Ansatz wird eine OSGi-basierte Android-Anwendung durch PlugIns (MapService-Bundles) um die Fähigkeit (Zugriffslogik) erweitert, direkt auf die MapServices zuzugreifen. Der schematische Aufbau ist in Abbildung 5.5 dargestellt.

Dafür muss das für den MapService entwickelte MapService-Bundle im Endgerät installiert sein. Durch den direkten Zugriff auf den MapService wird keine vermittelnde Serverkomponente mehr benötigt, wodurch das Problem der Zentralität der anderen Ansätze umgangen wird.

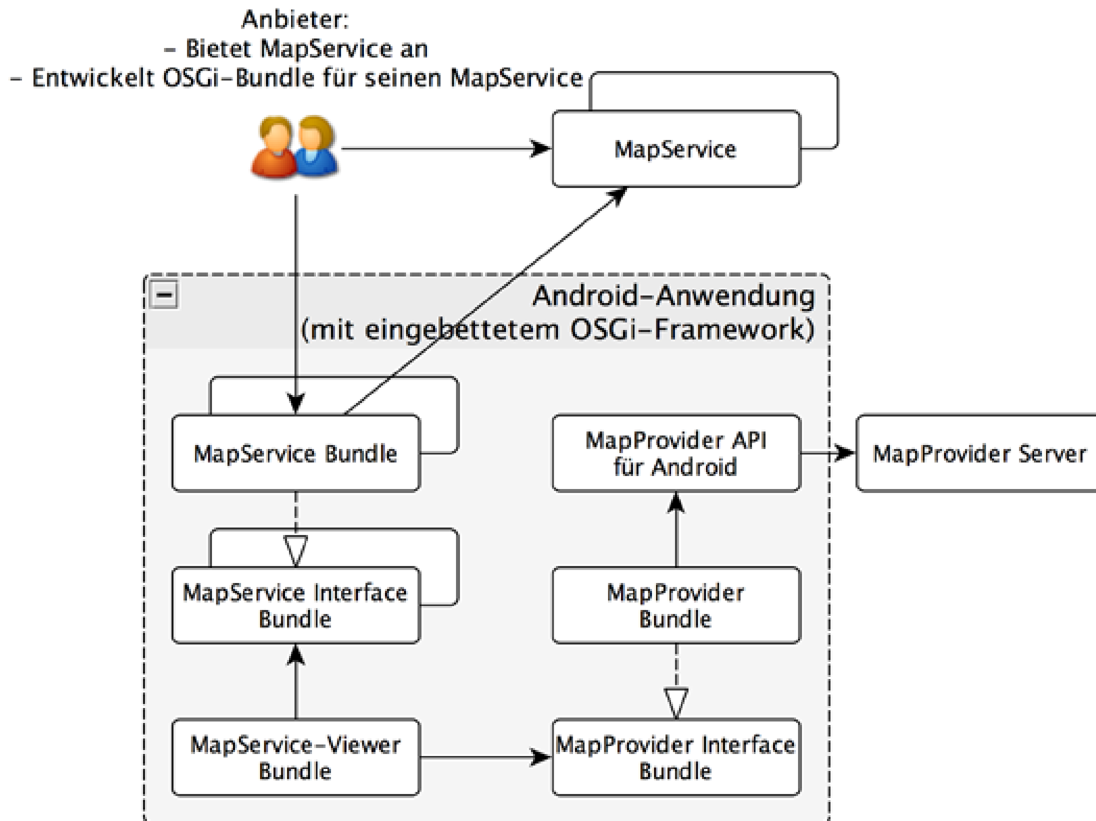


Abbildung 5.5: OSGi-basierte Android-Anwendung

OSGi-typisch können MapService-Bundles (Dienstklassen-Implementierungen) direkt aus dem Internet geladen und installiert werden. Somit wird die Dienstevermittlung indirekt durch ein Bundle-Repository erledigt, welche die MapService-Bundles zum einmaligen Download anbietet.

Nach dem Download erfolgt die Installation, was die Anwendung dazu befähigt, auf den MapService zuzugreifen. Wie beim D-OSGi Ansatz kann die Visualisierung entweder durch Wiederverwendung standardisierter MapService-GUI-Bundles erledigt werden oder der MapService-Anbieter liefert eine spezifische MapService-GUI mit, je nach Granularität integriert oder als separates Bundle. Eine feine Granularität der Bundles verbessert deren Wiederverwendbarkeit und spart dadurch Speicher, was sehr relevant bei mobilen Endgeräten ist.

Da es keine zentrale Server-Komponente gibt, wird die Zugriffslast auf die tatsächlich genutzten MapServices verteilt und der Flaschenhals wird vermieden. Die MapService-Bundles werden von den Anbietern in kompilierter Form zur Verfügung gestellt. Die Zugriffsdetails auf den MapService bleiben auf diese Weise geschützt. Die Verteilung des MapService-Bundles kann über die Webseite des Anbieters oder über ein zentrales Bundle Repository erfolgen. Auch die Aktualisierung ist auf

diesem Wege möglich. (Hot-Deployment Mechanismus von OSGi)

Wie bei der D-OSGi Lösung kapseln MapProvider-Bundles die Kartendarstellung mit Hilfe von entsprechenden Android-geeigneten APIs, wodurch Flexibilität der Kartenanbieter erreicht wird (F_3.1).

Generell teilt sich dieser Ansatz die Vorzüge des D-OSGi-Ansatzes, jedoch ohne sich auf noch nicht vollständig verabschiedete OSGi-Spezifikationen zu verlassen oder eine zentrale Komponente zu benötigen. Ebenfalls wie bei der D-OSGi-Lösung können Android-eigene Darstellungsmittel und Interaktionswege genutzt werden und auch alle Hardwarefeatures sind verfügbar.

Nachteilig ist, dass mehr Rechenlast auf das mobile Endgerät zukommt, was allerdings durch die Vorzüge Flexibilität, Unabhängigkeit (von einem Adapterserver) und Skalierbarkeit dank der Dezentralität aufgewogen wird. Hierdurch könnten auch verbesserte Antwortzeiten erreicht werden, da es keinen Flaschenhals und damit keine erhöhten Latenzzeiten durch eine vermittelnde Instanz gibt.

Es ist möglich, dass der Kommunikationsaufwand höher ist, als beim Einsatz einer zentralen Server-Komponente, welche eventuell Optimierungen der von den MapServices bezogenen Daten vornehmen könnte, wie zum Beispiel Extraktion der relevanten Informationen aus großen XML-Dokumenten. Bei zunehmender Auslastung einer solchen zentralen Server-Komponente könnten sich wiederum die Antwortzeiten so verschlechtern, dass trotz Optimierung die Gesamtantwortdauer länger ist, als der nicht-optimierte direkte Zugriff auf den MapService.

Da es sich bei den abgerufenen Daten meist um kleine Einheiten textueller Informationen (z.B. Namen, Koordinaten) handelt, sind die Vorzüge durch Optimierung zu gering, als dass die durch die zusätzliche Schicht bedingte erhöhte Anfragelaufzeit dadurch ausgeglichen werden würde. Da dies nur Annahmen sind, müsste das reale Verhalten empirisch untersucht werden. Letztlich lässt es sich auf das Verhältnis zwischen der Datenmenge und Komplexität der Extraktion der abzurufenden Informationen und den serverseitigen Optimierungsmöglichkeiten gewichtet mit ihrem Zeitbedarf reduzieren.

5.2.6 Auswahl des Architekturkonzepts

Da ein - bezüglich einer zentralen bzw. vermittelnden Server-Komponente - dezentraler Ansatz generell mehr Potential zum guten Skalieren hat und wie im D-OSGi-Ansatz alle Darstellungs- und Interaktionsmittel von Android genutzt werden könnnen, ist der Ansatz aus Kapitel 5.2.5 für diese Arbeit am interessantesten. Dank dem OSGi-Service-Paradigma und der dadurch guten Umsetzbarkeit des

Dienstklassen-Konzepts erfüllt diese Lösung alle nicht-funktionalen Anforderungen der Anforderungsanalyse und der Aufgabenstellung (vgl. Kap. 4.4).

Eine Ausnahme ist der Wunsch nach einem völlig plattformunabhängigen Ansatz. Da sich dieser Weg stark auf OSGi stützt, muss die Zielplattform mindestens JavaME unterstützen. Bei den vorgestellen Plattformen des Kapitels 2.1 bildet hier nur das Apple iPhone eine Ausnahme, bei dem Java aus Prinzip nicht unterstützt wird. Für die anderen Plattformen sind Java-VMs entweder integriert oder verfügbar.

Da OSGi zwar auf Java konzentriert ist, die Konzepte aber nicht unbedingt auf Java angewiesen sind, ist es denkbar, dass es in Zukunft auch OSGi-Framework-Implementierungen für andere Sprachen geben wird. Zumindest eine C++-Version war bereits Thema von Diskussionen [125]. Das Projekt „SOF“ zeigt auf, dass der OSGi Gedanke auf C++ übertragbar ist [126]. Sollten hier weitere Fortschritte erzielt werden, wird auch eine Objective-C Version wahrscheinlicher, was wiederum dieses Konzept für das iPhone zugänglich machen würde.

Zum aktuellen Zeitpunkt (Sommer 2009) sind derartige Möglichkeiten noch nicht verfügbar. Für die im Kapitel 2.1 gewählte Android-Plattform ist OSGi bereits jetzt verfügbar, sodass der Umsetzung dieses Konzepts nichts im Wege zu stehen scheint. Zunächst soll im Kapitel 5.3 dieser Ansatz verfeinert und mit Details angereichert werden. Im Kapitel 6 wird dann auf mögliche Schwierigkeiten bei der Umsetzung eingegangen.

5.3 Detailliertes Konzept

In diesem Kapitel soll das zuvor ausgewählte Konzept „OSGi-basierte Android-Anwendung“ (vgl. Kap. 5.2.5) bezüglich Aufbau und verwendeten Technologien genauer beschrieben werden. Eine detaillierte schematische Darstellung ist Abbildung 5.6 zu entnehmen. Dieses Detailkonzept wird dann im Kapitel 6 genutzt, um die Umsetzung anhand des Prototypen zu validieren.

5.3.1 Eingesetzte Technologie

Die Anwendung ist eine Android-Anwendung, welche die OSGi-Framework-Implementierung Apache Felix als Jar-Datei enthält und eingebettet startet. Beim Start werden ebenfalls die OSGi-Bundles Apache Felix iPOJO und FileInstall installiert und gestartet. Apache Felix iPOJO stellt das Komponentenmodell und vereinfacht die Beschreibung der Bundles als Komponenten. Wie im Kapitel 4.4 beschrieben, kontrolliert es das Service-orientierte Nutzungsverhältnis zwischen den Bundles.

FileInstall ist ebenfalls ein Apache Felix Sub-Projekt, dessen Funktion es ist, einen frei wählbaren Ordner im Dateisystem (bei Android z.B. auf der SD-Karte) zu überwachen. Werden beispielsweise durch Download OSGi-Bundles in diesen Ordner kopiert, so installiert FileInstall diese Bundles und versucht, sie zu starten. Diese Komponenten bilden zusammen das Host-System.

Ist es erwünscht, dass auch Bundles installiert werden können, die iPOJO Annotations verwenden, so muss die entsprechende Jar-Datei dem Host-System hinzugefügt werden. Alternativ kann sie auch in den von FileInstall überwachten Ordner kopiert werden.

Die einzelnen Aspekte der Anwendung (Kartendarstellung, MapService-Zugriff, Darstellung der MapService-Informationen) werden in die jeweiligen Bundles gekapselt. Diese Bundles müssen die in den jeweiligen Interface-Bundles beschriebenen Service-Interfaces implementieren. Das „MapService-Viewer"-Bundle enthält den Einstiegspunkt in die MapService-Viewer-Anwendung zur Darstellung von MapServices in Onlinekarten. Dieses Bundle ist das „nutzende Bundle", welches die von den anderen Bundles angebotenen Dienste verwendet.

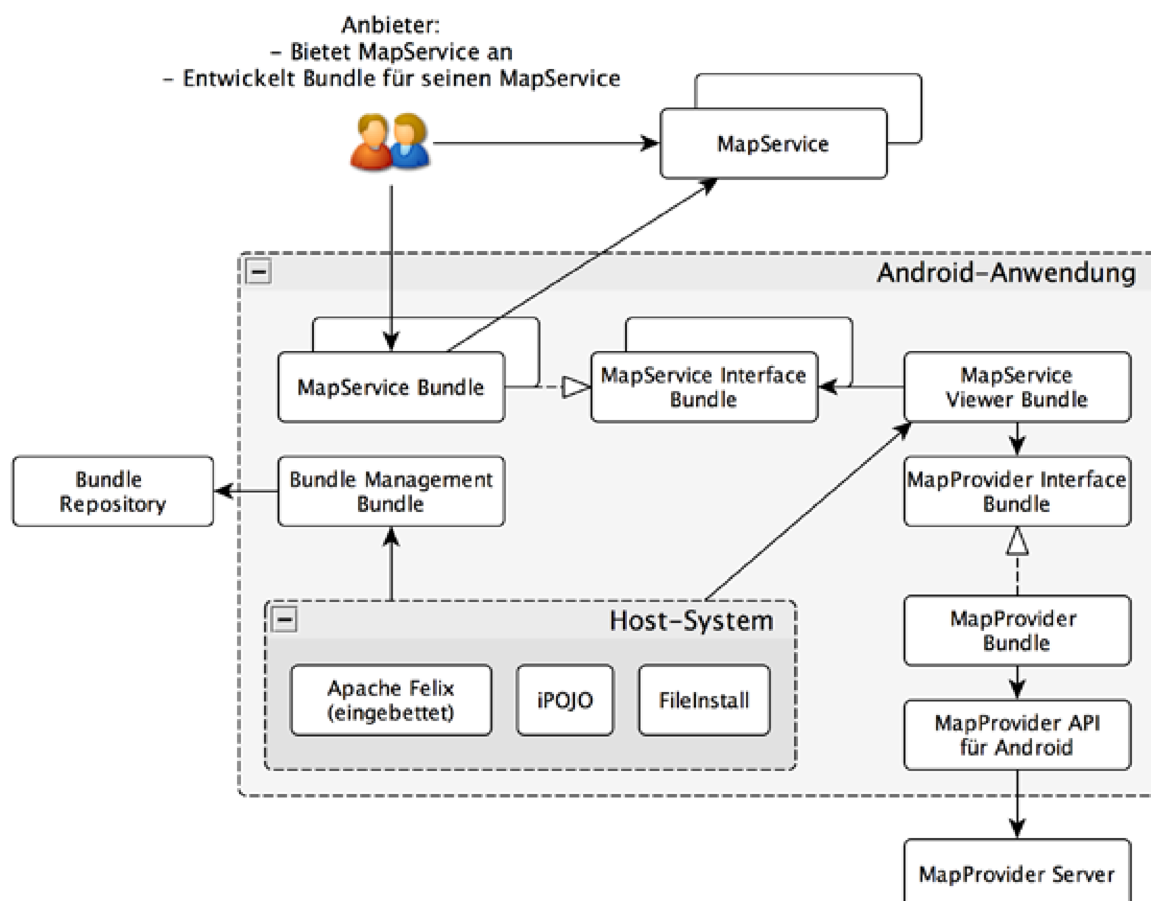


Abbildung 5.6: OSGi-basierte Android-Anwendung (Detailiertes Konzept)

Das nutzende Bundle kennt lediglich die Service-Interfaces, die lose Kopplung der tatsächlich installierten, implementierenden Bundles wird durch das OSGi-Framework durchgeführt.

Das „Bundle-Managment"-Bundle ist ein Android-GUI-Management-Agent, mit dem die Verwaltung (Laden, Installieren, Deinstallieren) der Bundles vom Nutzer vorgenommen werden kann. All die erwähnten Bundles werden im eingebetteten OSGi-Framework des Host-Systems installiert und je nach Bedarf gestartet. Diese Trennung von OSGi-Host-Anwendung und MapService-Viewer-Bundle ermöglicht prinzipiell die Integration beliebiger OSGi-basierter Anwendungen, die für die Android-Plattform entwickelt wurden und sich zur Integration an diesem Detailkonzept orientieren. Damit ist dieses Konzept leicht verallgemeinerbar auf beliebige, modulare, dynamische, PlugIn-orientierte Android-Anwendungen und ist nicht auf das MapService-Viewer-Szenario begrenzt.

5.3.2 Dienstklassen und die MapService-Interfaces

Das Dienstklassen-Konzept (vgl. Kap. 5.1.2) teilt die möglichen, unterstützten MapServices in Gruppen ein und weist jeder Gruppe eine Menge von Methoden zu, die ein Vertreter dieser Gruppe zur Verfügung stellen muss. Um aus den Dienstklassen Service-Interfaces für die MapService-Bundles ableiten zu können, müssen diese zunächst definiert werden. Dabei sollen sich die Dienstklassen nicht an dem ÖPNV-Beispielszenario des Prototypen orientieren, sondern so allgemein wie möglich bezüglich der darzustellenden Informationen gehalten sein.

Die hier vorgestellten Dienstklassen sind die grundlegenden und reichen für viele Anwendungsfälle, so auch für das Beispiel-Szenario, aus. Das Konzept kann problemlos um weitere Dienstklassen erweitert werden. Die Anpassung vorhandener Dienstklassen ist durch die strikte Versionierung bei OSGi konsistent und sicher möglich. Sollten durch die Darstellungsvorhaben einer Dienstklasse bestimmte Fähigkeiten der MapProvider-Implementierung benötigt werden, so muss die konkrete MapProvider-Implementierung diese Dienstklasse explizit unterstützen.

5.3.2.1 Dienstklasse 1: Darstellung von MapItems in der Karte

Die Dienstklasse 1 (MapService-Layer 1) beschreibt den Abruf einer Menge von MapItems. Diese werden in Form eines Markers in der Karte durch den aktuellen MapProvider in einem Overlay dargestellt. Ein MapItem enthält eine Bezeichnung, eine Beschreibung und Geokoordinaten. Der Abruf erfolgt sinnvollerweise ortsabhängig. So soll es möglich sein, alle vom MapService verfügbaren MapItems im Umkreis (in Metern) einer Koordinate oder innerhalb des aktuell angezeigten Kartenausschnittes (Koordinatenrechteck) abzurufen. Optional soll der Marker im

MapService-Bundle integriert sein. Ist keiner verfügbar, muss ein Standardmarker verwendet werden. Ein MapItem in der Karte ist generell anklickbar. Eine Reaktion auf den Klick wird in dieser Dienstklasse aber nicht direkt behandelt.

Die Visualisierung dieser Dienstklasse in der Karte muss vom MapProvider gestellt werden und beschränkt sich auf einen MapItem-Overlay, der mit den abgerufenen MapItems gefüllt wird. Ein Overlay ist eine transparente Ebene über der Karte, auf der MapItems und andere Darstellungen platziert werden können. Es ist möglich, einer Karte mehrere Overlays zuzuordnen (Details siehe Anhang A2). Dem MapItem-Overlay wird sein zugehöriger Layer 1 MapService (MSL1) und ein Standardmarker übergeben.

Der MSL1 dient als Datenquelle für MapItems, der eventuell mit dem MSL1 assoziierte MapService-Layer-3 (MSL3) wird zur GUI Darstellung bei Klick auf ein MapItem des MapItem-Overlays aufgerufen. Der MSL3 ist für die Darstellung von MapItem-Informationen in Form eines eigenen Bildschirms zuständig (siehe auch Dienstklasse 3).

Das Klick-Handling für die MapItems wird zwar im Overlay des MapProviders abgehandelt, ist aber dynamisch, da nur mit den Service-Interfaces und nicht mit konkreten Implementierungen gearbeitet wird. Ein MapProvider ist damit nur an die Version des ihm zur Entwicklungszeit bekannten MapService-Interfaces gebunden. Da es keine Verpflichtung eines MapProviders gibt, alle MapService-Layer zu unterstützen, muss dieser eine Liste von unterstützten MapServices anbieten. Diese kann dann vom Konfigurationsdialog genutzt werden, um nur die vom gewählten MapProvider anzeigbaren MapServices anzubieten.

MapServices dieser Dienstklasse haben oft die Möglichkeit, unterschiedliche Typen von MapItems darzustellen und danach zu filtern (z.B. POI-Dienst: nur Restaurants abrufen). Dafür muss eine Liste von Filter-Optionen abruf- und setzbar sein, um den Abruf entsprechend zu parametrisieren.

Das im MSL1 intern gesetzte Anzeige-Zoomlevel bestimmt das maximale Zoomlevel der Kartendarstellung, bei dem die Marker dieses MapItemOverlays gerade noch angezeigt werden und Abfragen nach neuen MapItems durchgeführt werden. Sinnvollerweise sollte jeder MSL1 seine abgerufenen MapItems cachen, was zur Optimierung von Antwortzeit und Datentransferlast genutzt werden kann.

In Dienstklasse 1 ist im Beispiel-Szenario die Darstellung der Haltestellen in der Karte angesiedelt.

5.3.2.2 Dienstklasse 2: Darstellung von Verbindungen zwischen MapItems

Wie Dienstklasse 1 ist auch Dienstklasse 2 eine kartenintegrierte Darstellung, wodurch Bundles dieser Klasse (ebenfalls wie Dienstklasse 1) reine Datenquellen sind, während der MapProvider ihre Visualisierung übernimmt.

Diese Dienstklasse deckt die Anzeige von Routen zwischen MapItems ab. Damit lassen sich beispielsweise Navigationsanwendungen oder Verbindungsauskünfte realisieren. Eine Route besteht aus einem Startpunkt, einem Endpunkt und endlich vielen Wegpunkten. Alle Punkte entsprechen MapItems. Das OGC-Standardisierte XML-Datenformat KML eignet sich gut als Datenaustauschformat von Routen [89]. Das MapService-Layer-2 -Bundle (MSL2) ruft diese Daten vom MapService ab und wandelt sie in eine Liste von MapItems um, bei welcher der erste Eintrag der Start und der letzte Eintrag das Ziel sind.

Die GUI besteht aus einem MapItem-Overlay, welche die Overlay-Items aus der vom MSL2 abgefragten MapItem-Liste bezieht. Da hier die Visualisierung vom MapProvider vorgenommen wird, muss dieser Dienstklasse 2 explizit unterstützen. Neben der reinen Darstellung einer Route werden noch Interaktionsmöglichkeiten zum Setzen von Start-, Weg- und Zielpunkt (MapItems) benötigt. Hierfür muss der MSL2 einen Dialog anbieten, der vom MapProvider abgerufen und eingebunden wird. Alternativ kann auch ein separater MSL4 benutzt werden.

5.3.2.3 Dienstklasse 3: Darstellung von Informationen zu einem MapItem

Die Dienstklasse 3 (MapService-Layer 3, MSL3) bietet an, eine neue Activity (einen kompletten Android-Bildschirm) zu starten, die mit der View gefüllt ist, welche in diesem MSL3-Bundle erzeugt wurde. Da die View-Erzeugung in dem Bundle, das dieses Service-Interface implementiert, gekapselt ist, sind den Gestaltungsmöglichkeiten keine Grenzen gesetzt. Da diese Activity zur Darstellung von Informationen zu einem einzelnen, konkreten MapItem dient, muss das betreffende MapItem beim Aufruf des MSL3 übergeben werden.

In dieser Dienstklasse ist im Beispiel-Szenario die Darstellung des Abfahrtsmonitors angesiedelt.

5.3.2.4 Dienstklasse 4: Darstellung von Informationen zu beliebig vielen MapItems

Diese Dienstklasse ähnelt der Dienstklasse 3, orientiert sich aber an einer Liste von MapItems und stellt diese ebenfalls in einer separaten Activity dar. Beispielsweise kann in dieser Dienstklasse die Visualisierung einer Routenbeschreibung vorgenommen werden. In diesem Fall ist das MSL4-Bundle mit einem MSL2-Bundle assoziiert.

5.3.3 MapProvider-Bundles

Um verschiedene MapProvider in das System einzubinden, definiert das MapProvider-Service-Interface lediglich eine Methode um die Activity, in der die Karte dargestellt werden soll, zu starten. Die Activity, von der aus die Karten-Activity (bei Google Maps entspricht das einer MapActivity) gestartet wird, muss als Context-Träger übergeben werden. Ebenfalls wird eine Konfiguration übergeben, welche die vom Nutzer ausgewählten, verfügbaren MapServices darstellt. Es ist als Liste angedacht, die die eindeutigen Namen der ausgewählten MapService-Bundles enthält. Durch dieses Vorgehen werden sämtliche Details der Kartenanzeige in das MapProvider-Bundle gekapselt, wodurch deren Einbindung erleichtert wird.

Um Inkompatibilitäten zwischen dem vom Nutzer gewählten MapProvider und den MapServices zu vermeiden, kann eine Konfigurations-GUI den betreffenden MapProvider auf Kompatibilität bezüglich eines MapService-Layers abfragen. Die Kompatibilität bezieht sich also auf die von diesem MapProvider unterstützten Dienstklassen.

Das MapProvider-Bundle bedient sich für die Kartendarstellung an der API-Bibliothek, die für den einzubindenden MapProvider für Android zur Verfügung stehen muss. Diese kann auf dem Classpath der Android-Hostanwendung liegen (wie im Fall von Google Maps lizenzbedingt notwendig) oder können auch als eingebettete Jar-Datei in das MapProvider-Bundle integriert werden.

Die allgemeinen Datentypen, mit denen ein MapProvider-Bundle umgehen muss, werden zum einen durch die MapService-Layer-Interfaces definiert und zum anderen durch ein Bundle, welches globale Datentypen wie MapItem, MapItemList, GeoPoint und GeoRect allgemein und damit unabhängig von einem konkreten MapProvider bzw. MapService definiert. Einem MapItem-Overlay, wie ihn ein MapProvider-Bundle zur MapItem-Darstellung nutzt, liegt daher eine systemweit einheitliche MapItemList zugrunde.

5.4 Erfüllung der Anforderungen an das Konzept

Anhand der Tabelle 5.1 soll die Erfüllung der zusätzlich im Verlauf der Arbeit angesammelten Anforderungen an das Konzept überprüft werden. Die Erfüllung der Anforderungen aus dem Kapitel 3 wird in der Evaluation im Kapitel 7 besprochen, in welchem die Ergebnisse des Kapitels 6 dahingehend untersucht werden sollen.

Anforderung	Detailkonzept
Anwendung „Allg. PlugIn-Konzepts“	Ja, durch OSGi-Bundle
Positionierung der Dynamik	In Android-Anwendung
Übertragbarkeit auf andere Plattformen	Ja, bei OSGi mind. JavaME Support nötig
Austauschbarkeit des MapProviders	Ja, durch MapProvider-Bundles
Skalierbarkeit der Lösung	Gut, kein zentraler Flaschenhals
Möglichkeiten, verschiedenste Dienste zu integrieren	Ja, durch MapService-Bundle-Abstraktion
Möglichkeit der Einbindung eines Routenservices	Ja, durch Dienstklasse 2
Einschränkungen bzgl. der darzustellenden MapService-Funktionalitäten	Teilweise, durch Dienstklassendef. bedingt, Dienstklassen sind erweiterbar
Möglichkeit zur Nutzung der Hardware-Features des Endgerätes	Ja, da native Android-Anwendung
Offlineverfügbarkeit von Informationen	Ja, Persistenz in Android-Anwendung mgl.

Tabelle 5.1 Zusätzliche Anforderungen an das Konzept

5.5 Fazit

In diesem Kapitel wurden zur Findung einer Lösung der Aufgabenstellung auf Basis der vorhergehenden Kapitel allgemeine Konzepte vorgestellt und verschiedene Architekturkonzepte verglichen. Dabei wurde der Architektur-Ansatz, eine Android-Anwendung um OSGi-basierte PlugIns für jeden MapService bzw. MapProvider zu erweitern, präferiert. Dieser Ansatz wurde dann weiter detailliert und auch die Umsetzung des grundlegenden Dienstklassenkonzepts wurde beschrieben. Im Kapitel 6 soll nun die prototypische Umsetzung dieses Detailkonzepts diskutiert werden.

6 Implementierung

In diesem Kapitel soll die prototypische Umsetzung des favorisierten Detailkonzepts aus dem Konzeptionskapitel beschrieben werden. Eingangs werden einige Besonderheiten und Einschränkungen der Softwareentwicklung für Android beschrieben, die erst während der Entwicklung des Prototypen entdeckt wurden. Dies ist notwendig, um einige Implementierungsdetails und Architekturentscheidungen zu verstehen, da hierdurch einige Anpassungen des Detailkonzepts notwendig wurden.

Anschließend wird das Gesamt-System übersichtsweise beschrieben und die Zusammenhänge und Aufgabenverteilung mittels Architektur- und Aufrufgraphen zwischen den Komponenten erklärt. Danach werden einige interessante Details der Umsetzung behandelt, wie zum Beispiel die Umsetzung des Dienstklassenkonzepts und die Kapselung der Kartendarstellung in ein MapProvider-Bundle.

Im Kapitel 6.2 werden die Service-Interfaces beschrieben. Darauf aufbauend werden dann im Kapitel 6.3 Beispiele für die Implementierungen dieser Services so detailliert dargestellt, dass die Implementierung weiterer Module für dieses System anhand der gegebenen Informationen problemlos erfolgen kann. Abschließend werden weitere Ergebnisse der prototypischen Umsetzung vorgestellt, wie zum Beipsiel das Bundle-Management-Bundle oder das in der Entwicklungszeit verwendete und modifizierte iPOJO-Eclipse-PlugIn.

6.1 Einschränkungen und Besonderheiten

Ist man mit einer Software-Plattform nocht nicht vertraut und die eingesetzte Technologie noch vergleichsweise jung, muss stets damit gerechnet werden, dass unvorhergesehene Zwischenfälle auftreten und bestimmte Einschränkungen erst im Verlauf der Entwicklung bekannt werden. So sind auch während der Entwicklung dieses Prototypen einige Besonderheiten aufgetreten, die in diesem Kapitel beschrieben werden sollen. Viele von ihnen bewirkten Änderungen in der geplanten Architektur und im Kontrollfluss. Einige Probleme waren derart schwerwiegend, dass der Erfolg des ganzen Projekts auf dem Spiel stand.

Dank der tatkräftigen Unterstützung durch einige Entwickler des Apache Felix Projekts war es jedoch möglich, diese Schwierigkeiten zu überwinden.

Sie halfen maßgeblich, das dynamische Classloading, von dem Apache Felix so umfangreich gebrauch macht, auf Android ab Version 1.5 lauffähig zu machen. Die zuvor notwendigen Modifikationen am Endgerät waren dadurch nicht mehr erforderlich.

Ebenfalls schwerwiegend und von projektgefährdender Natur war ein Problem in der DalvikVM, das als Android Issue 2711 bekannt ist [127]. Dieses Problem, das ProSyst in seiner Android Edition ihrer OSGI-Framework-Implementierung zum Releasezeitpunkt der Version 1.0 noch nicht lösen konnte, tritt beispielsweise auf, wenn in einem Bundle von einer Klasse geerbt werden soll, die nur über den Classpath der Host-Anwendung erreicht werden kann [128]. Hierfür importiert das Bundle das Package, welches die gewünschte Oberklasse enthält, von der Host-Anwendung, tut dies allerdings unter Verwendung des eigenen Classloaders (bei OSGi hat jedes Bundle seinen eigenen Classlaoder). Da hierdurch die Host-Anwendung die Oberklasse mit einem anderen Classloader als das „erbende Bundle“ lädt (nämlich mit seinem eigenen), schlägt das Erzeugen einer Instanz dieser Unterklasse zur Laufzeit mit einem DalvikVM-Verfication-Fehler fehl.

Um dieses Problem zu umgehen, ist es notwendig, dass das „erbende Bundle“ die Oberklasse mittels desselben Classloaders erhält wie die Host-Anwendung, welche direkten Zugang zu dieser Klasse über ihren Classpath hat. Im konkreten Fall schlug der Versuch fehl, von der Klasse MapView aus der maps.jar, die in jedem Android-Gerät ab API Level 3 integriert ist, in einem Bundle zu erben. Da Vererbung bei Android ein sehr häufig genutztes Mittel zur individuellen Anpassung der Android-Standardkomponenten ist, war klar, dass dieses Problem nicht nur in diesem Sonderfall auftreten würde.

Die Lösung erfolgte durch einen Felix-eigenen Mechanismus namens Bootdelegation. Dadurch werden Anfragen eines Bundles nach einer Klasse (genauer nach einem Package), die nicht explizit in seiner Manifest-Datei importiert wird und auch nicht durch den Bundle-lokalen Classpath befriedigt werden können, an den Classloader des Frameworks weitergeleitet (delegiert). Zur Steuerung dieses Mechanismus wurden im Felix 2.0 Release Konfigurationsmöglichkeiten geschaffen, die es ermöglichen, das Bootdelegation-Verhalten von Felix für Android so anzupassen, dass der beschriebene Fehler nicht auftritt.

Eine andere Einschränkung von Android trat bezüglich dynamischer GUI zutage. Jede Activity, die in einer Android-Anwendung verwendet werden soll, muss vorher per Name und Package in der AndroidManifest.xml der Android-Anwendung deklariert werden, da die Activity sonst nicht gestartet wird und eine Exception geworfen wird. Dadurch kann die Erzeugung einer neuen Activity nicht in einem OSGi Bundle selbst erfolgen (die selbst im eigentlichen Sinne keine Android-Anwendungen

sind), sondern muss in der Host-Anwendung vorgenommen werden. Umgangen wurde dieses Problem durch einen ActivityService, dessen Service-Interface die Host-Anwendung exportiert und dessen Implementierung in der Host-Anwendung selbst beinhaltet ist. Dieser Service erzeugt für nutzende Bundles eine leere „Activity-Hülse“ des gewünschten Activity-Typs.

In diesem Kontext wurde noch eine weitere Problematik deutlich, nämlich dass bisher kein Weg bekannt ist, um threadsicher eine Referenz auf die gerade gestartete Activity zu erhalten. Dies ist dem Android-GUI-Programmierparadigma geschuldet: Es betrachtet Activitys als funktionsbeladene, selbständige Anwendungsblöcke, die sich autonom um sich kümmern und nicht von außen gesteuert werden sollen- außer durch Nutzerinteraktion, auf die die Activitys aber selbst reagieren. Eine externe GUI Generierung durch Bundles scheidet dadurch scheinbar aus.

Ein erster Ansatz im GUI-erzeugenden Thread, auf die Erzeugung der neuen Activity zu warten und diese sich global registrieren zu lassen, hat sich als unerwartet kompliziert und vergleichsweise threadunsicher herausgestellt. Auch wenn, oder vielleicht gerade weil Android ein Single-UI-Thread-Modell verwendet, kam es bei diesem Ansatz zu Thread-Konkurrenz-bedingten Abstürzen. Da es also nicht günstig war, die GUI einer Activity von außen zu setzen, wurde der Kontrollfluss umgekehrt, sodass nun die Activity beim Start eine Referenz auf einen ViewProviderService erhält, bei dem sie sich in ihrer onCreate() Methode die View abrufen kann. Das ViewProviderService-Interface wird ebenfalls von der Host-Anwendung gestellt und exportiert.

Eine weitere, aber eventuell behebbare Einschränkung ist, dass die GUI, insofern sie in den Bundles erzeugt wird, nicht durch XML-Layout-Dateien beschrieben werden kann, da dies einen Build-Schritt beim Erstellen einer Android-Anwendungsdatei (APK) voraussetzt, der nicht ohne weiteres auf OSGi-Bundles angewendet werden kann. Dies hat zur Folge, dass die GUI, wie zum Beispiel Views, Viewgroups, Layouts und Buttons, programmatisch im Java-Code erzeugt werden müssen. Leider umfasst die programmatische Umsetzung nicht alle Gestaltungsmöglichkeiten der XML-Beschreibungs-Variante, was allerdings nur selten eine tatsächliche Einschränkung darstellt (siehe auch [129]). Die programmatische GUI Erzeugung benötigt stets den Context der Activity, in der die UI Elemente dargestellt werden sollen. Auch dieser Sachverhalt benötigte die beschriebene Umkehrung des Kontrollflusses.

Im Kapitel 6.2 werden die dargestellten Ansätze in den Kontext des Gesamtsystems gestellt und positioniert.

6.2 Aufbau der Anwendung

Die Beschreibung der Anwendung erfolgt gegliedert in ihre einzelnen Komponenten. Die Abbildung 6.1 zeigt das vollständige Gesamtsystem als Schema. Generell sind die Service-Interface-Bundles, welche nur die Schnittstellenbeschreibung enthalten, normale OSGi-Bundles, aber keine iPOJO-Komponenten. Die implementierenden Bundles hingegen sind OSGi-Bundles, die mindestens eine iPOJO-Komponente beinhalten.

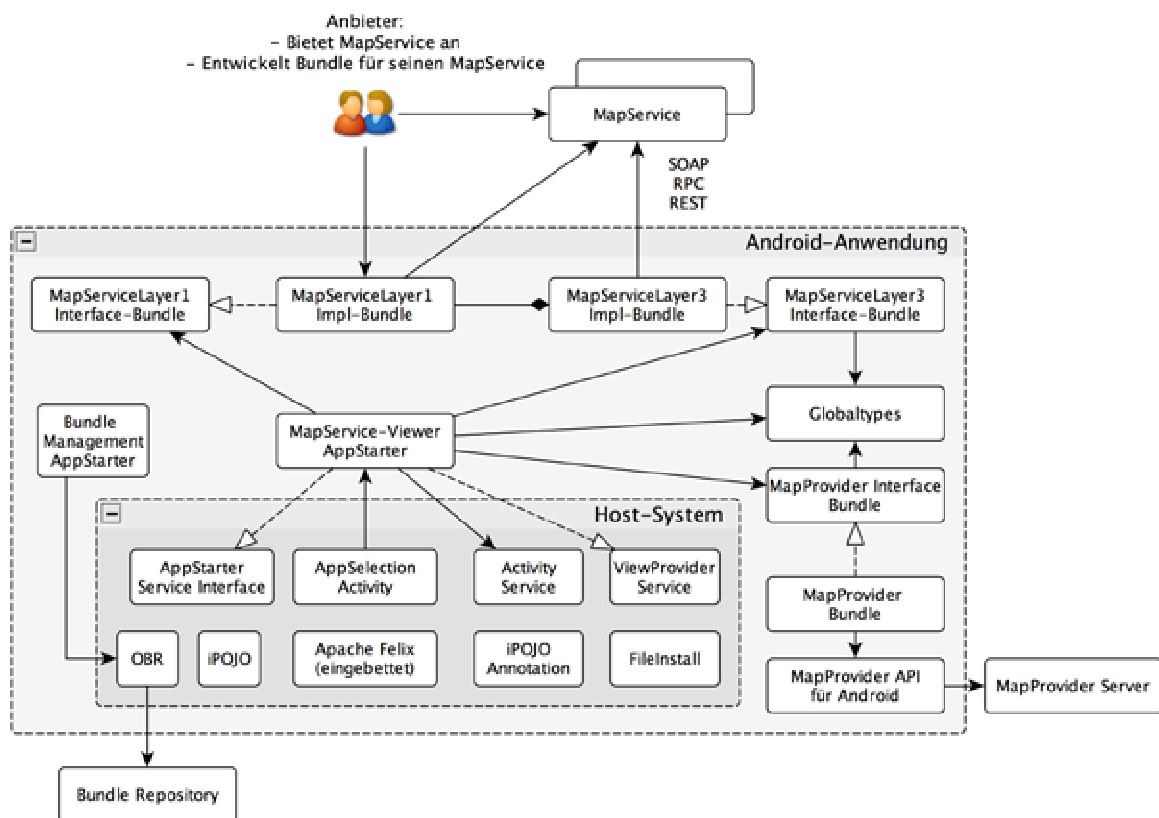


Abbildung 6.1: Schematische Darstellung des Gesamtsystems

6.2.1 Die Host-Anwendung

Die Host-Anwendung (Package de.mnsoft.felixhostapp) ist die eigentliche (und damit einzige) Android-Anwendung im gesamten System. Ihr Aufbau ist in Abbildung 6.2 dargestellt. Sie bildet die Anwendung, die vom Nutzer installiert und über das Android-Menü gestartet wird. 4 Bundles aus Apache Felix Sub-Projekten werden verwendet: iPOJO, iPOJO Annotations, FileInstall und OSGi Bundle Repository.

Da die Bundles als iPOJO Komponenten definiert und auf Basis von iPOJO Mechanismen erzeugt werden und zusammenarbeiten, muss das iPOJO Bundle installiert sein. Falls auch Bundles installiert werden sollen, die iPOJO-Annotations verwenden, so muss das iPOJO Annotations Bundle installiert sein.

Das FileInstall Bundle überwacht den Ordner „FelixOSGiBundles" auf der SD-Karte, in welchen manuell oder durch einen Management-Agent neue Bundles abgelegt werden. Sobald in diesen Ordner neue Bundles hinzugefügt oder vorhandene entfernt werden, so installiert und startet bzw. stoppt und deinstalliert FileInstall das jeweilige Bundle.

Um das Bundle-Management-Bundle zu betreiben, muss auch das OSGi Bundle Repository-Bundle (OBR) installiert sein. Diese vier Bundles sind die „Basis-Bundles", die in dem Ressourcen-Ordner für binäre Daten „res/raw/" der Android-Host-Anwendung abgelegt sind.

Folgende Auflistung soll die Zusammenhänge anhand von Abbildung 6.2 verdeutlichen:

1. Start der Host-Anwendung durch den Nutzer. Die StartActivity ist der Einstiegspunkt in die Host-Anwendung und ist für den Start der Felix-Instanz verantwortlich.

2. Beim Start der StartActivity wird eine vorkonfigurierte eingebettete Felix-Instanz in einem Android-Service (Klasse FelixService) gestartet, die als OSGi-Framework-Implementierung den Lebensraum der OSGi Bundles darstellt.

3. Ein in der Host-Anwendung implementierter BundleActivator (BasicBundleLoader) lädt die in der APK der Host-Anwendung integrierten Jar-Dateien der Basis-Bundles und installiert sie in das eingebettete Felix-OSGi-Framework. Dieser BundleActivator wird in der Felix-Konfiguration im FelixService der Liste der AutoStart-Activators (Config-Schlüssel „felix.systembundle.activators") hinzugefügt. Auf die gleiche Weise wird auch der ActivityService gestartet. (vgl. Kap. 6.2.3)

4. Nach erfolgreichem Start des OSGi-Frameworks wird zur AppSelectionActivity übergeleitet, deren GUI aus einer Liste von Buttons besteht. Für jeden im System installierten AppStarter-Anbieter des Service-Interfaces „de.mnsoft.felixhostapp.appstarter" wird ein Button, der mit dem Applikationsnamen dieses AppStarters beschriftet ist, der Liste hinzugefügt. Wird ein AppStarter oder ein vom AppStarter benötigtes Bundle deinstalliert, so wird der Button zur Laufzeit aus der Liste entfernt.

5. Dieser Mechanismus wird durch einen ServiceTracker ermöglicht, der auf Anbieter des AppStarter-Interfaces wartet.

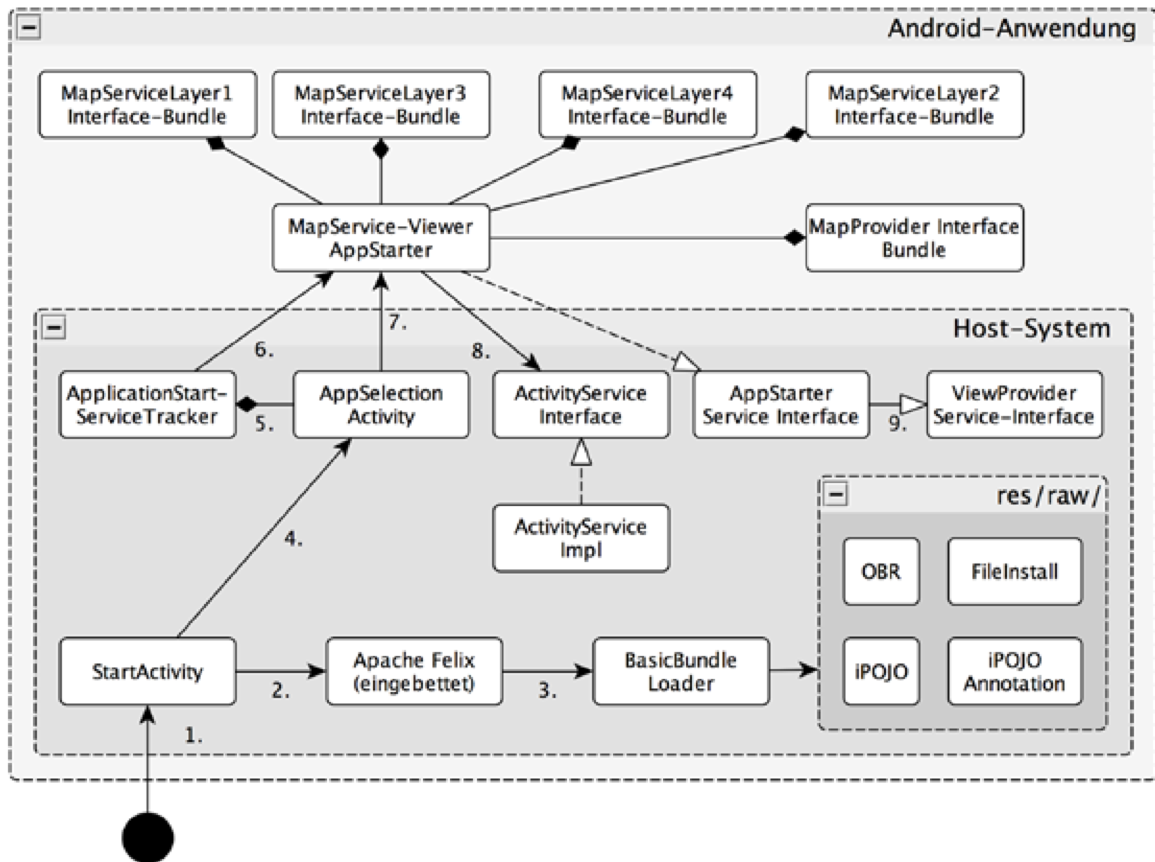


Abbildung 6.2: Schematische Darstellung der Host-Anwendung

6. Warten auf das AppStarter-Interface implementierende Bundles. Die AppSelectionActivity zeigt durch diesen Mechanismus nur die tatsächlich aktuell verfügbaren, installierten OSGi-basierten Anwendungen (repräsentiert durch ihren AppStarter) an. Dieser Weg ist notwendig, da die iPOJO Möglichkeiten innerhalb der Host-Anwendung selbst nicht verfügbar sind, da es sich bei ihr nicht im eine iPOJO Komponente handelt, sondern um eine normale Android-Anwendung.

7. Der Einstiegspunkt in eine jede Android-OSGi-Anwendung ist der bereits erwähnte „AppStarter“.

8. Bei Klick auf den zugehörigen Button in der AppSelectionActivity wird eine neue Activity mit Hilfe des ActivityServices gestartet.

9. Da jeder AppStarter durch eine Activity repräsentiert wird, muss er auch eine View anbieten, um diese mit Inhalt zu befüllen. Hierfür erbt das AppStarter-Interface vom ViewProvider-Interface.

6.2.2 Das AppStarter-Interface

Im Kapitel 6.2.1 wurde das AppStarter-Interface (Package de.mnsoft.felixhostapp.appstarter) als Einstiegspunkt in eine Android-OSGi-Anwendung eingeführt. Prinzipiell ist er aber auch gleichzeitig die logische Repräsentation der Anwendung selbst. Die Activity des AppStarters wird gestartet, indem seine startApplication(Activity) - Methode von einer anderen (Host-)Activity (wie der AppSelectionActivity) aufgerufen wird. In dieser Activity befindet man sich bereits in der logischen OSGi-Anwendung, von der aus die Funktionen der Anwendung angesteuert werden können. Dazu bindet ein AppStarter die Bundles, aus denen seine konkrete OSGi-Anwendung besteht und verwendet ihre angebotenen Services.

Die hier umgesetzte AppStarter-Implementierung „MapService-Viewer“ (Package de.mnsoft.mapserviceviewer.appstarter) bindet alle installierten MapServices und MapProvider. Das Binden findet durch den iPOJO Managed-Services Mechanismus statt, bei dem in diesem Fall aggregierte Requirements durch Injection auf private Felder in Form von Arrays von MapService-Interface-Typen bzw. vom MapProvider-Interface-Typ abgebildet werden. Darauf basierend ermöglicht seine AppStarter-Activity den Start der MapProvider-Activity per Button-Klick. Zudem bietet die AppStarter-Activity auch eine Einhängmöglichkeit für ein Konfigurationsmenü zur Auswahl des zu verwendenden MapProviders und der anzuzeigenden MapServices aus der Menge der installierten und damit gebundenen Services.

Da es sich bei dem vorgestellten Prototypen lediglich um ein proof-of-concept der modularen, dynamischen MapService-Integration mittels OSGi-Bundles handelt, wurde auf komfortable Konfigurations-Dialoge zur Auswahl von MapProvider und MapServices verzichtet. Die Service-Interfaces und auch die Beispiel-Implementierung sehen allerdings Methoden vor, um in einem Dialog setzbare Einstellungen abzurufen und die vom Nutzer vorgenommenen Einstellungen auch tatsächlich zu verwenden. Im Prototypen wird zur Zeit eine feste Default-Konfiguration verwendet.

6.2.3 ActivityService und ViewProviderService

Soll eine neue Activity gestartet werden, so wird stets der ActivityService genutzt, weshalb das Bundle, welches eine neue Activity starten möchte, den ActivityService binden muss. Die neue Activity wird zum Beispiel durch Aufruf von startGenericActivity(Activity, ViewProviderService) gestartet, wodurch eine GenericActivity erzeugt wird, welche von Activity erbt. Dabei wird der neuen Activity ein das ViewProviderService-Interface implementierendes Objekt zugänglich gemacht, von dem die Activity in ihrer onCreate() Methode ihre View durch createViewForActivity(Activity callingActivity) und ihren Titel durch getTitle() beziehen kann.

Ebenso bedient sich die Activity bei der Erzeugung und Befüllung von Optionsmenü bzw. Kontextmenü am ihrem ViewProviderService. Da letzteres aber optional ist (längst nicht jede Activity benötigt ein Options- bzw. Kontextmenü) muss die View-ProviderService-Implementierung angeben, ob sie die jeweiligen Menüs unterstützt (durch supportsOptionsMenu() und supportsContextMenu()).

Der AppStarter und das in der Konzeption angeführte Prinzip der MapService-GUI-Bundles (realisiert beispielsweise im MapService-Layer 3) lässt sich durch die Implementierung des ViewProviderService-Interfaces umsetzen. Ein Bundle kann auch mehrere Activities beinhalten, was am einfachsten dadurch realisiert wird, dass das Bundle für jede Activity eine separate Klasse bereitstellt, die das Interface ViewProviderService implementiert. Beim Start einer Activity durch Nutzung des ActivityService wird dann einfach eine neue Instanz dieser Klasse als ViewProviderService übergeben. Das Bundle „Bundle Management" ist ein gutes Beispiel dafür und auch für den Einsatz von Optionsmenüs (siehe Abschnitt 6.4.1).

6.2.4 Globaltypes Interface und Implementierung

Das Globaltypes-Bundle (Package de.mnsoft.mapserviceviewer.globaltypes und .impl) definiert global einheitliche Datentypen für den MapService-Viewer wie z.B. MapItem, MapItemList, GeoPoint und GeoRect, die einen gemeinsamen, von konkreten MapProvidern und MapServices unabhängigen Nenner schaffen. Das Interface definiert einen FactoryService, der von den anderen Bundles der logischen MapService-Viewer-Anwendung verwendet werden kann, um Instanzen dieser globalen Datentypen zu erzeugen.

6.2.5 MapService-Interface-Bundles

Jede Dienstklasse wird durch ein MapService-Interface beschrieben. Ein Bundle, das eine Dienstklasse erfüllen soll, muss das entsprechende Service-Interface für die angestrebte Dienstklasse adäquat implementieren. Es existiert eine Reihe von Methoden, die von jeder Dienstklasse (von jedem MapService-Layer) unterstützt werden müssen. Daher erben die MapService-Layer-Interfaces vom allgemeinen Interface „MapService", um die folgenden Methoden automatisch zu beinhalten:

- List<MapServiceFilterOption> getAvailableFilterOptions()
- List<MapServiceFilterOption> getUsedFilterOptions()
- void setFilterOptionsToUse(List<MapServiceFilterOption>)
- Sting getShortName()
- String getUniqueName()
- String getDescription()
- void setCurrentZoomLevel(int)

Alle Methodennamen sollten selbsterklärend sein. Für die Methode getUniqueName() hat es sich bewährt, den vollqualifizierten Klassennamen der Klasse zu verwenden, die das MapService-Layer-Interface implementiert. Die Zusammenhänge werden in Abbildung 6.3 schematisch dargestellt.

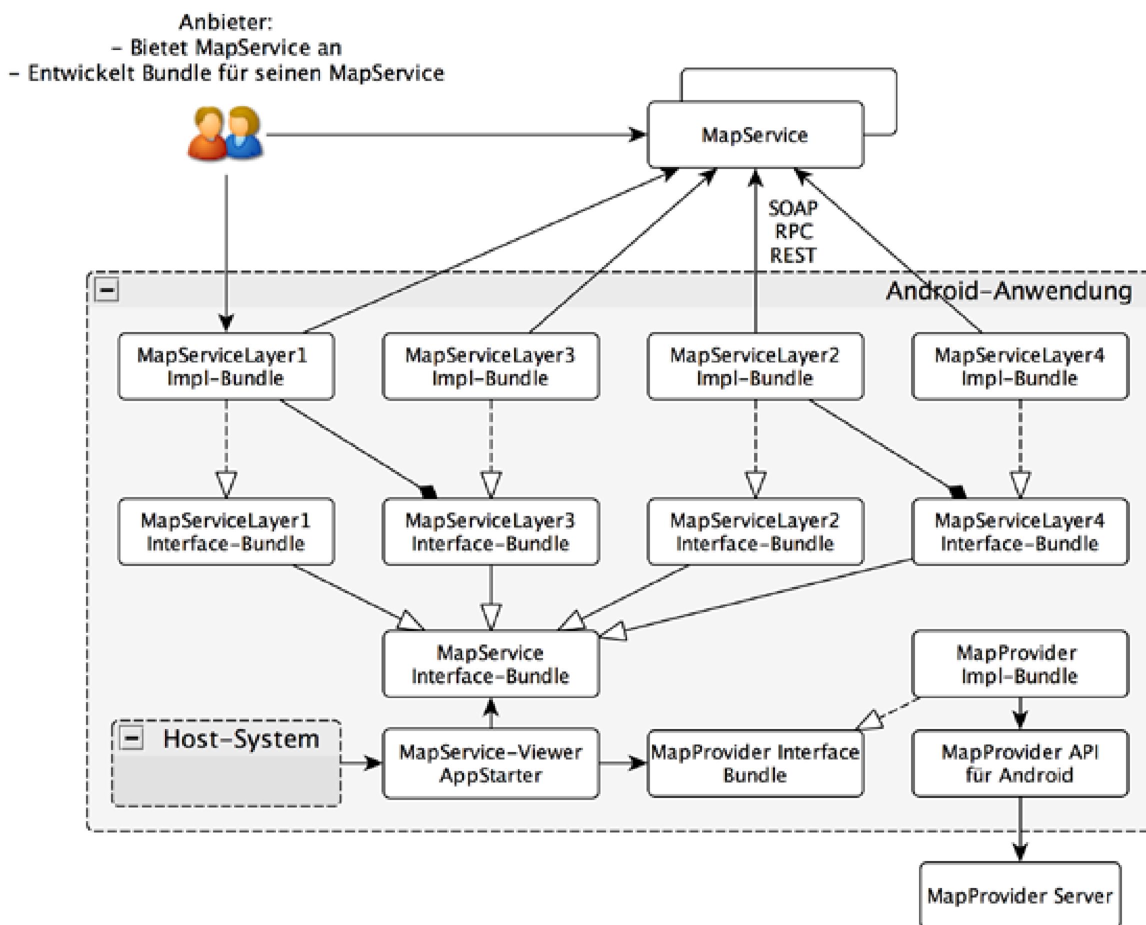


Abbildung 6.3: Schematische Darstellung der MapServices

6.2.5.1 MapService-Layer 1

Dieses Interface definiert die Dienstklasse 1 (MSL1). Bei Aufruf der Datenabfrage (z.B. getMapItemsInRadius()) wird zuvor das aktuelle Zoomlevel des aufrufenden MapProviders übergeben. Anhand diesem wird dann entschieden, ob überhaupt eine Abfrage stattfindet und auch, ob die MapItems in der Karte überhaupt angezeigt werden. Das entsprechende Service Interface enthält folgende Methoden:

- Drawable getListDefaultMarker();
- MapItemList getMapItemsInRadius(GeoPoint, int);
- MapItemList getMapItemsInGeoRect(GeoRect);

- boolean supportsGeoRectQuery();
- boolean hasAssociatedMSL3();
- MapServiceLayer3 getAssociatedMSL3();

getListDefaultMarker() liefert den Standardmarker dieses MapService-Layer 1. Er wird für alle MapItems dieses Layers verwendet, insofern die MapItems in der MapItemList keine eigenen Marker enthalten. Im letzteren Fall werden diese bevorzugt verwendet. Dieses Verhalten wird allerdings vom MapProvider gesteuert, da dieser für die Visualisierung von MSL1 (und auch MSL2) zuständig ist. Ein MSL1 kann ein das MapServiceLayer3-Service-Interface (siehe Abschnitt 6.2.5.3) implementierendes Bundle an sich binden und es für die Darstellung von Details zu seinen MapItems nutzen.

Dieses Binden kann durch LDAP -Filter Ausdrücke in der iPOJO-metadata.xml des konkreten MapService-Layer 1 Bundles festgelegt werden. Je nach Vorhandensein einer solchen MSL3 Bindung startet das MapProvider-Bundle die Activity des assoziierten MSL3 oder zeigt beispielsweise nur einen Android-Toast an.

6.2.5.2 MapService-Layer 2

Ein MSL2 dient der Routendarstellung. Eine Route entspricht einer MapItemlist, deren erstes Element der Startpunkt und deren letztes Element der Zielpunkt ist. Die verbleibenden MapItems sind Wegpunkte. Die Route wird wie MSL1 in der Karte von der MapProvider-Implementierung dargestellt, da dessen Overlay-Funktionen hierfür gut geeignet sind. Um die Darstellung der Route vorzunehmen, muss ein MSL2 für Start, Weg und Zielpunkt entsprechende Marker zur Verfügung stellen.

Für weitere UI Elemente kann ein MSL2 einen MSL3 oder einen MSL4 verwenden, oder selbst eine View anbieten und hierzu das ViewProvider-Interface implementieren.

- MapItemList getRouteForMapItems(MapItem start, MapItem end)
- Drawable getStartMarker()
- Drawable getEndMarker()
- Drawable getRouteMarker()
- boolean hasAssociatedMSL3();
- MapServiceLayer3 getAssociatedMSL3();
- boolean hasAssociatedMSL4();
- MapServiceLayer4 getAssociatedMSL4();

6.2.5.3 MapService-Layer 3

Beim Start eines MSL3 mit der Methode startMapServceLayer3Activity(Activity callingActivity, MapItem mapitem) wird für das übergebene MapItem eine Activity mit Hilfe des ActivitySevice gestartet. Da das MapService-Layer 3 -Interface vom ViewProviderService-Interface erbt, muss eine MSL3 Implementierung die entsprechenden Methoden zum Abruf seiner View bereitstellen. Die Gestaltung der View obliegt gänzlich dem MSL3-Bundle.

6.2.5.4 MapService-Layer 4

Das MSL4 Interface ist wegen der großen Ähnlichkeit zwischen Dienstklasse 3 und 4 analog dem MSL3 sehr ähnlich. Als einziger Unterschied wird statt eines MapItems eine ganze MapItemList beim Start übergeben.

- void startMapServceLayer4Activity(Activity, MapItemList)

6.2.6 MapProvider-Interface Bundle

Das MapProvider-Interface kapselt die Darstellung einer dynamischen Onlinekarte in einer View, welche in einer separaten Activity eingebettet ist. Die einzelnen Methoden zur Manipulation der Karte und ihrer Darstellung werden nicht nach außen hin angeboten, sondern Steuerfluss und Karten-View Erstellung finden nach außen hin verborgen im Innern der jeweiligen MapProvider-Service Implementierung statt. Ein MapProvider wird also von einer Karten-Activity repräsentiert, die vorwiegend eine Karten-View enthält. Da MapProvider APIs, auf die sich die jeweiligen MapProvider-Implementierungen stützen, häufig schon ein eigenes abgestimmtes Overlaykonzept bieten, wurde es Aufgabe der MapProvider, die Kartenintegration (Darstellung in der Karte) der MapItems aus den MapServices der Klassen 1 und 2 zu übernehmen.

Um absolute Unabhängigkeit der Visualisierung von MSL1 und MSL2 vom konkreten MapProvider zu erreichen und die MapProvider-Implementierung von dieser Aufgabe zu entbinden, müsste ein separates Overlay-Bundle eingeführt werden, welches eine völlig transparente View stellt, die oberhalb der kartenanzeigenden View die MapItems einblendet, auf Änderungen der Karte (z.B. durch Nutzerinteraktion) reagiert und die Darstellung anpasst. Kurz gesagt, müsste die Overlay-Funktionalität, wie sie aus der Android Google Maps API bekannt ist, vollständig nachimplementiert werden.

Da dies eine sehr anspruchsvolle und zeitraubende Arbeit ist, wurde darauf im Prototypen verzichtet und die vorhandene Overlayfunktionalität im Beispiel der Google Maps API in Form des ItemizedOverlays verwendet. Der Vorteil dieses

Overlay-Bundles wäre neben der Entlastung der MapProvider-Implementierungen eine einheitliche Darstellung und die Möglichkeit, auch MapProvider-APIs zu unterstützen, die keine eigene Overlay-Funktionalität besitzen. In diesem Fall könnte ein solches Overlay-Bundle in Form eines Services vom MapProvider-Bundle genutzt werden.

Auf Grund der starken Kapselung der aktuellen Lösung definiert das MapProvider-Interface nur 2 Methoden:

- void startMapScreen(Activity, HashMap<String, List<String>>)
- boolean checkMapServiceLayerIsSupported(String)

In der startMapScreen() Methode soll die Activity gestartet und die Konfiguration (ausgewählte MapServices) zur Verwendung übernommen werden. Aufgrund des Darstellungsverhältnisses zwischen MapProvider und MSL1 und 2 muss die MapProvider-Implementierung mittels checkMapServiceLayerIsSupported() beantworten können, ob die Dienstklasse (zur Zeit ist der MapProvider nur für Klasse 1 und 2 zuständig) von ihm unterstützt wird. Hierfür muss der vollqualifizierte Name des MapServiceLayer-Interfaces übergeben werden.

6.3 Beispiele der prototypischen Umsetzung

Neben den Bundles, welche die Service-Interfaces enthalten, wurden auch einige implementierende Bundles prototypisch erstellt. So wurden jeweils zwei MapService-Layer 1 Bundles zum Abruf der Haltestellen der VVO und zum Abruf von POI aus dem Qype-Webservice implementiert. Passend zu diesen beiden MSL1 wurden entsprechende MapService-Layer 3 Bundles erstellt. Der VVO-MSL3 stellt den Abfahrtsmonitor für eine vom VVO-MSL1 abgerufene Haltestelle dar. Der QYPE-MSL3 stellt Namen, Adresse und Kontaktinformationen zu dem POI-MapItem dar, die wiederum vom Qype-MSL1 abgerufen wurden.

Als Beispiel für ein MapProvider-Bundle wurde ein Google-Maps-basiertes Bundle erstellt. Es unterstützt die Darstellung von MSL1. Unterstützung für MSL2 kann mit geringem Aufwand nachgerüstet werden. In den folgenden Abschnitten soll das Google Maps MapProvider-Bundle, ein MSL1-Bundle am Beispiel des Qype-MSL1 und ein MSL3 am Beispiel des VVO-MSL3 (Abfahrtsmonitor) beschrieben werden.

Die Bundle GeoCoder und LocationService sind sehr einfache Bundles, die vom MapProvider-Bundle zum Geocoding genutzt werden, beziehungsweise um den

aktuellen Ort des Gerätes anzuzeigen.

Im MapProvider wird der LocationService genutzt, um die im Optionsmenü enthalte Funktion „Akutelle Position anzeigen“ umzusetzen. Im Abschnitt 6.3.1 wird die Umsetzung des MapProvider-Bundles beschrieben. Danach wird jeweils die Umsetzung eines MSL1 (Qype-Service) und eines MSL3 (VVO-Abfahrsmonitor) vorgestellt.

6.3.1 Umsetzung eines MapProvider Bundles am Beispiel von Google Maps

Im MapProvider wird beim Start durch Aufruf des ActivityService eine MapActivity gestartet, da eine MapView (welche die Karte darstellt) nur in einer MapActivity angezeigt werden kann (siehe Anhang A2). So muss auch beim Erzeugen der MapView im ViewBuilder eine MapActivity übergeben werden. Diese Zusammenhänge werden im Klassendiagramm in Abbildung 6.4 im Anhang A4 verdeutlicht. Zur Darstellung der MapItems (die von MSL1 bezogen werden) verwendet die MapView die Klasse MapItemOverlay, die vom ItemizedOverlay des Package com.google.android.maps erbt.

Die das MapProviderService-Interface implementierende iPOJO Komponente (Klasse MapProviderServiceImpl) hält aggregierte Requirements auf die unterstützten MapService-Layer in Form von Listen. Dadurch werden alle installierten MSL1 (die aktuelle Implementierung unterstützt zur Zeit noch kein MSL2) dynamisch zur Laufzeit an diese Komponente gebunden. Beim Start wird dem MapProvider vom MapService-Viewer-AppStarter die Konfigurationsliste unter anderem mit den gewählten MSL1 übergeben.

Anhand dieser Konfigurations-Liste wird entschieden, welche der verfügbaren MSL1 „registriert“ und damit tatsächlich verwendet werden sollen.(Methode registerAllSelectedMSL1Services()) Wird ein MSL1 registriert, so wird im gleichen Zuge ein MapItemOverlay für den MSL1 angelegt. Sobald durch Aufruf der ViewProvider-Service-Methode createViewForActivity() die MapView erzeugt wurde, wird die Liste der MapItemOverlays der registrierten MSL1 zur Liste der Overlays dieser MapView hinzugefügt.(nutzt Methode addMapItemOverlayForMSL1()) Wird zur Laufzeit ein MSL1 entfernt, so wird dieser automatisch deregistriert und der Overlay wird vom MapView entfernt. Dadurch kann die MapView dynamisch auf das Vorhandensein von MSL1 reagieren.

Jedem MapItemOverlay wird bei der Erzeugung sein zugehöriger MSL1 übergeben. Dadurch kann der MapItemOverlay die MapItemList vom MSL1 abfragen, welche er selbst dann in der MapView in Form von Markern darstellt. Dabei wird geprüft, ob das MapItem einen eigenen speziellen Marker hat.

Wenn nicht, wird der Defaultmarker seiner MapItemList verwendet. Ist auch dieser nicht verfügbar, so wird der Fallback-Marker des MapProviders selbst verwendet.

Im MapItemOverlay wird auch das Auswählen eines MapItems abgehandelt. Bei einem Tap wird in der Methode onTap() zunächst mittels getHitMapItem() geprüft, ob ein MapItem des MapItemOverlays getroffen wurde. Ist dies nicht der Fall, so wird lediglich eine Abfrage nach neuen MapItems an den MSL1 gerichtet, wobei der geographische Ort des „Tappens“ als Ausgangspunkt verwendet wird.(Methode updateBaseMSL1()) Außerdem gibt die Methode onTap() dann false zurück, um das Tap-Ereignis an die anderen MapItemOverlays der anderen registrierten MSL1 weiterzureichen.

Wurde ein MapItem getroffen, so wird im MapItemOverlay geprüft, ob der zugehörige MSL1 einen MSL3 assoziert. Wenn ja, wird dessen Activity durch Aufruf seiner Start-Methode aufgerufen. Falls nicht, wird mit showToastForSelectedItem() lediglich ein Toast, der den Titel des MapItems anzeigt, eingeblendet. Dadurch kann der MapItemOverlay dynamisch auf das Vorhandensein von MSL3 reagieren.

Die vorgestellten Mechanismen lassen sich äquivalent auf die MapService-Layer 2 und 4 übertragen, bei denen sehr ähnliche Mechanismen zur Darstellung und dynamischen Integration benötigt werden. Diese MapProvider-Implementierung könnte daher leicht um Support für MSL2 mit assoziiertem MSL4 erweitert werden. Leider lag zur Erstellungszeit des Prototypen kein Webservice zur Verbindungsauskunft seitens des VVO vor.

6.3.2 Umsetzung eines MapService - Layer 1 Bundles: Der Qype.com POI Dienst

Das Vorgehen zur Umsetzung eines MapService-Layer 1 Bundles soll am Beispiel des im Prototyp enthaltenen Qype.com MSL1 (Package de.mnsoft.mapservice.layer1.qypeimpl) demonstriert werden.

Qype.com ist eine Webseite, die es angemeldeten Nutzern ermöglicht, Points of interests (POIs) einzutragen, zu bewerten und mit Informationen wie Kontaktdaten, Rufnummern und Bildern anzureichern. Die Anbieter dieser Webseite stellen einen REST Webservice (siehe Anhang A1) als QYPE-API kostenfrei angemeldeten Entwicklern zu Verfügung. Darüber hinaus wird eine ausführliche Dokumentation bereitgestellt, die über alle Möglichkeiten dieser API Auskunft gibt.

Der Qype.com POI Dienst wurde als Ersatz für den ursprünglich geplanten OGC-konformen OpenLS Directory Service vom OpenRouteService.org- Projekt (ORS) ausgewählt. Nach anfänglichen Versuchen, ORS zu verwenden, musste schnell festgestellt werden, dass vermutlich durch starke Auslastung oder

technische Gegebenheiten die Antwortzeiten dieses MapServices erheblich schwanken. Auch außerhalb der Stoßzeiten wurden Antwortzeiten von über einer Minute gemessen. Dies ist für die Abfrage lediglich einiger ÖPNV-Haltestellen in einem 2-km-Radius (Test-Anfrage) viel zu lang. Ursprünglich sollte mit dem Einsatz eines OGC-konformen MapServices ein Beispiel für die Einbindbarkeit dieser standardisierten Welt in das Konzept dieser Arbeit demonstriert werden.
Da aber durch die Einbindung der ausgewählten nicht-OGC-konformen Dienste gezeigt werden konnte, dass wirklich beliebige MapServices eingebunden werden können, kann geschlussfolgert werden, dass dies auch für die wohldefinierten OGC-Dienste der Fall ist.

Die Abbildung 6.5 zeigt das Klassendiagramm dieses Bundles inklusive der assoziierten Service-Interfaces. Den Mittelpunkt eines MSL1 bildet die Klasse, die das MapServiceLayer1-Interface implementiert, was in diesem Fall die Klasse QypeMSL1ServiceImpl im Package de.mnsoft.mapservices.layer1.qypeimpl ist.

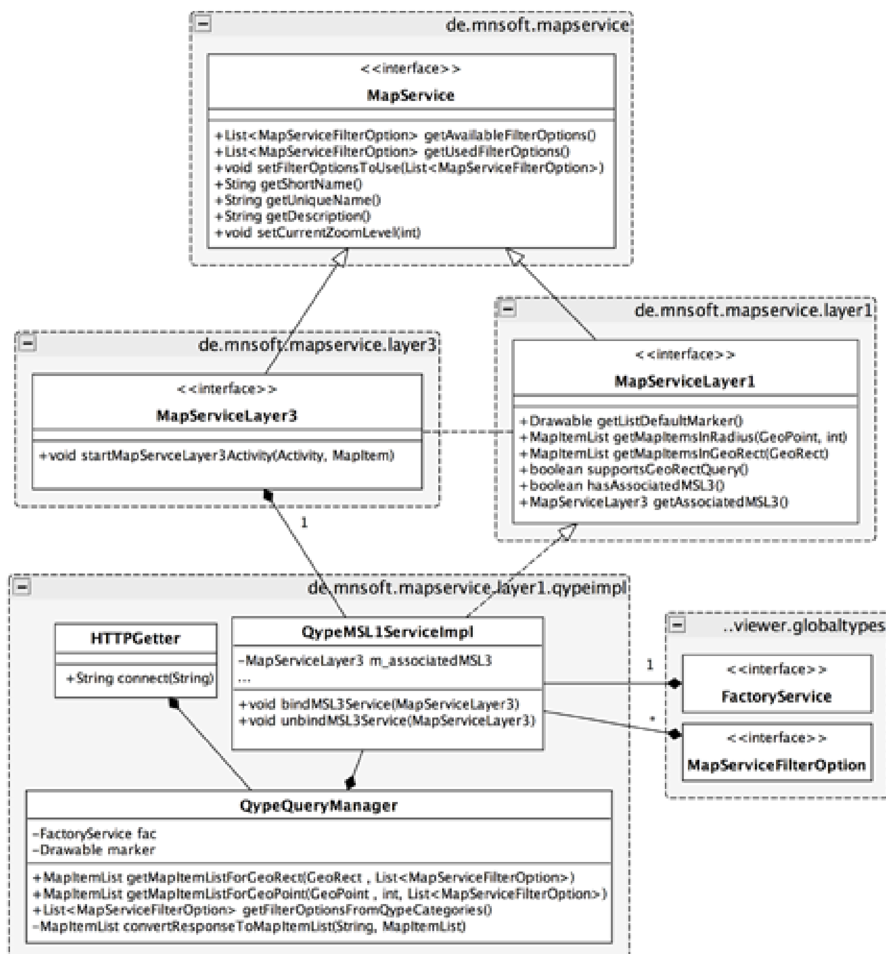


Abbildung 6.5: Klassendiagramm MapServiceLayer1-Implementation - Qype.com

Diese Klasse muss nach außen hin alle Methoden anbieten, die vom MapService-Layer1-Interface und vom MapService-Interface definiert werden.

Die einfachen Methoden wie Getter für Name und Beschreibung werden in der Klasse direkt behandelt, die Zugriffslogik auf dem Qype.com RESTful Webservice wurde der Übersichtlichkeit halber in die Klasse QypeQueryManager ausgelagert. Diese benutzt die kleine Klasse HTTPGetter, um HTTP-GET Anfragen an diesen MapService zu richten. Da es sich hierbei nur um Leseanfragen handelt, entfällt die sonst bei der Qype API übliche QAuth- Authentifizierung. Es genügt, den nach kostenloser Anmeldung als Qype.com Developer erhältlichen Consumer-Key jeder Leseanfrage als Parameter beizufügen.

Da der Qype MapService sowohl Umkreis-Anfragen (Koordinatenpaar und Radius um dieses) als auch Kartenausschnitt-Anfragen (rechteckiger Kartenausschnitt, bestimmt durch den nord-östlichen und den süd-westlichen Eckpunkt des Ausschnitts) unterstützt, können im Gegensatz zum VVO-MSL1 beide MSL1-Methoden bedient werden und supportsGeoRectQuery() darf „true“ zurückgeben.

Die MapService-Antworten haben ein definiertes, strukturiertes Format und können in Form von XML oder JSON Ausdrücken geliefert werden. Das Format wird in der HTTP-GET Anfrage im Accept-Header gesetzt. Da Android von Haus aus über einen performanten JSON-Parser verfügt, wurde das JSON Format als Ausgabeformat gewählt. Die MapServiceFilterOptions des globaltypes-Packages entsprechen in diesem Fall den Kategorien, in welche die Qype-Nutzer die POIs einordnen. Mittels der 3 Filteroptionen-Zugriffsmethoden im MapService-Interface könnte dieses MapService-Bundle von einer zentralen Konfigurations-UI aus eingestellt werden. Letztere ist aber nicht Teil dieses proof-of-concept.

Diese MSL1 Implementierung verfügt über eine optionale Assoziation zu einem MSL3. Dieser soll genutzt werden, um die Detailinformationen zu einem vom Nutzer gewählten Qype-MapItem auf der Karte anzuzeigen. Prinzipiell wäre auch ein allgemeiner, nicht-Qype-spezifischer MSL3 geeignet, jedoch soll an diesem Beispiel gleich das Binden zu bestimmten MSL3 auf Basis von Filtern gezeigt werden.

Um das Qype-MSL1 Bundle nach außen hin als OSGi-Bundle und als iPOJO-Komponente zu beschreiben, sind zwei Post-Build-Schritte notwendig. Beide Schritte werden in zwei getrennten speziellen Textdateien beschrieben.

Zum Bundling, also zum Erstellen der OSGi-Bundle Jar-Datei, kann unter anderem das Tool Bnd verwendet werden [130]. Die Bnd.jar kann über die Konsole, über Ant oder Maven oder als Eclipse-PlugIn (Rechtsklick auf *.bnd, im Kontextmenü - „Make Bundle“) verwendet werden. In jedem Fall wird eine *.bnd Textdatei verwendet, die vom Syntax her der Manifest-Datei der Bundles ähnelt. In ihr werden auf sehr einfache Weise Imports, Exports und andere Bundling-Details definiert. Da Bnd

den im Build erzeugten Bytecode und den Classpath analysiert, können in der Bnd-Datei beispielsweise Imports mit Wildcards (*) enthalten sein, die nach der Analyse zu OSGi-konformen Imports transformiert werden. Aus den Informationen der Bnd-Datei wird die eigentliche Manifest-Datei generiert und diese in die ebenfalls erstellte Jar-Datei integriert. Nach diesem Prozess ist die Jar-Datei ein vollwertiges OSGi-Bundle. Die Bnd-Datei für diesen MSL1 ist in Listing 6.1 zu sehen.

Listing 6.1: Bnd - Datei des Qype MSL1

```
Import-Package: de.mnsoft.mapserviceviewer.globaltypes, de.mnsoft.mapservice, de.mnsoft.
mapservice.layer1, de.mnsoft.mapservice.layer3
DynamicImport-Package: android.*, org.apache.*, org.xml.*, org.w3c.dom, javax.*, com.android.*,
org.json, org.json.*
Private-Package: de.mnsoft.mapservice.layer1.qypeimpl
Include-Resource: res/defaultMarkerRed.png = res/defaultMarkerRed.png
```

„DynamicImport-Package“ ist kein Bnd-Befehl und wird unverändert in die Manifest-Datei übernommen. Hier werden die Packages angeführt, die - wenn möglich - über den im Kapitel 6.1 beschriebenen Bootdelegation-Mechanismus über den Classloader der Host-Anwendung geladen werden sollen. Ist dies nicht möglich, so wird versucht, die Packages über den normalen Package-Import zu erreichen.

Mit „Include-Resource“ können beliebige Daten leicht in die Jar-Datei eingeschlossen werden. Dies eignet sich sowohl für Ressourcen wie Bilder, aber auch für einzubettende Bibliotheken in Form von Jar-Dateien. Um eingebettete Bibliotheken verwenden zu können, müssen die betreffenden Packages vom betreffenden Bundle exportiert und wieder (von sich selbst) importiert werden.

Im Gegensatz zu den Bundles, die nur die Service-Interfaces beschreiben (z.B. de.mnsoft.mapservice) sollen Service-Implementierungen (z.B. de.mnsoft. mapservice.layer.qypeimpl) zusätzlich zur Vereinfachung von Dynamik und Servicenutzungsverhältnis iPOJO Mechanismen verwenden. Dafür ist es zusätzlich notwendig, die Service-implementierende Klasse als iPOJO-Komponente zu beschreiben.

Die Klassse QypeMSL1ServiceImpl hält zwei private Felder vom Typ zweier Service-Interfaces. Im iPOJO-Buildschritt wird durch den Field-Injection Mechanismus an diese Felder bei Start der iPOJO-Komponente eine geeignete (durch Filter bestimmt) und verfügbare (installiert und gestartet) Service-Implementierung gebunden. Dieser Mechanismus wurde im Kapitel 4.4.5.3 beschrieben. Die für den Mechanismus notwendige metadata.xml ist in Listing 6.2 angeführt. Sie enthält alle iPOJO Informationen, da auf die Verwendung von iPOJO-Annotations verzichtet wurde.

Listing 6.2: metadata.xml des Qype MSL1

```
<ipojo
 <component classname="de.mnsoft.mapservice.layer1.qypeimpl.QypeMSL1ServiceImpl">
  <provides specifications="de.mnsoft.mapservice.layer1.MapServiceLayer1"
strategy="singleton"/>
  <requires field="m_fac" optional="false"/>
  <requires field="m_associatedMSL3" optional="true" filter="(vendor.name=QYPEMSL3)" >
   <callback type="bind" method="bindMSL3Service"/>
   <callback type="unbind" method="unbindMSL3Service"/>
  </requires>
 </component>
 <instance component="de.mnsoft.mapservice.layer1.qypeimpl.QypeMSL1ServiceImpl"/>
</ipojo>
```

Die Komponente wird durch ihren vollqualifizierten Klassennamen angegeben. Der provides Tag beschreibt das implementierte Interface MapServiceLayer1 und die requires Tags beschreiben die Bindungen zu den benötigten Services.

Die optionale Bindung zum MSL3 wurde um einen LDAP-Filter-Ausdruck erweitert. Es werden dadurch nur MSL3 gebunden, die ein Property namens „vendor. name“ anbieten, dessen Wert „QYPEMSL3“ ist. Durch dieses Property wird sichergestellt, dass der Qype-MSL1 nur den passenden Qype-MSL3 verwendet. Die Callback-Methoden zum Binden und Trennen dienen in dieser Implementierung nur zu Logging-Zwecken.

6.3.3 Umsetzung eines MapService - Layer 3 Bundles: Der VVO-Abfahrtsmonitor

Der VVO-MSL3 ist der VVO-Abfahrtsmonitor, der für die in der Karte durch den VVO-MSL1 dargestellte, vom Nutzer ausgewählte Haltestelle den Echtzeit-Abfahrtsmonitor anzeigt. Abbildung 6.6 zeigt das Klassendiagramm dieses MSL3-Bundles.

Der MSL3 erhält die Haltestelle in Form eines MapItems.(Methode startMapServceLayer3Activity()) Mit Hilfe der HTTPGetter Klasse wird mit dem darin enthaltenen Haltestellennamen eine Abfrage an den Abfahrsmonitor-MapService der VVO getätigt. Die Antwort ist standardmäßig im JSON Format, bzw. eine serialisierte Liste von Listen. Die inneren Listen enthalten Liniennummer, Endhaltestelle und Wartezeit in Minuten. Diese Listeneinträge werden im ViewBuilder zu StopMonitorEntrys objektifiziert.

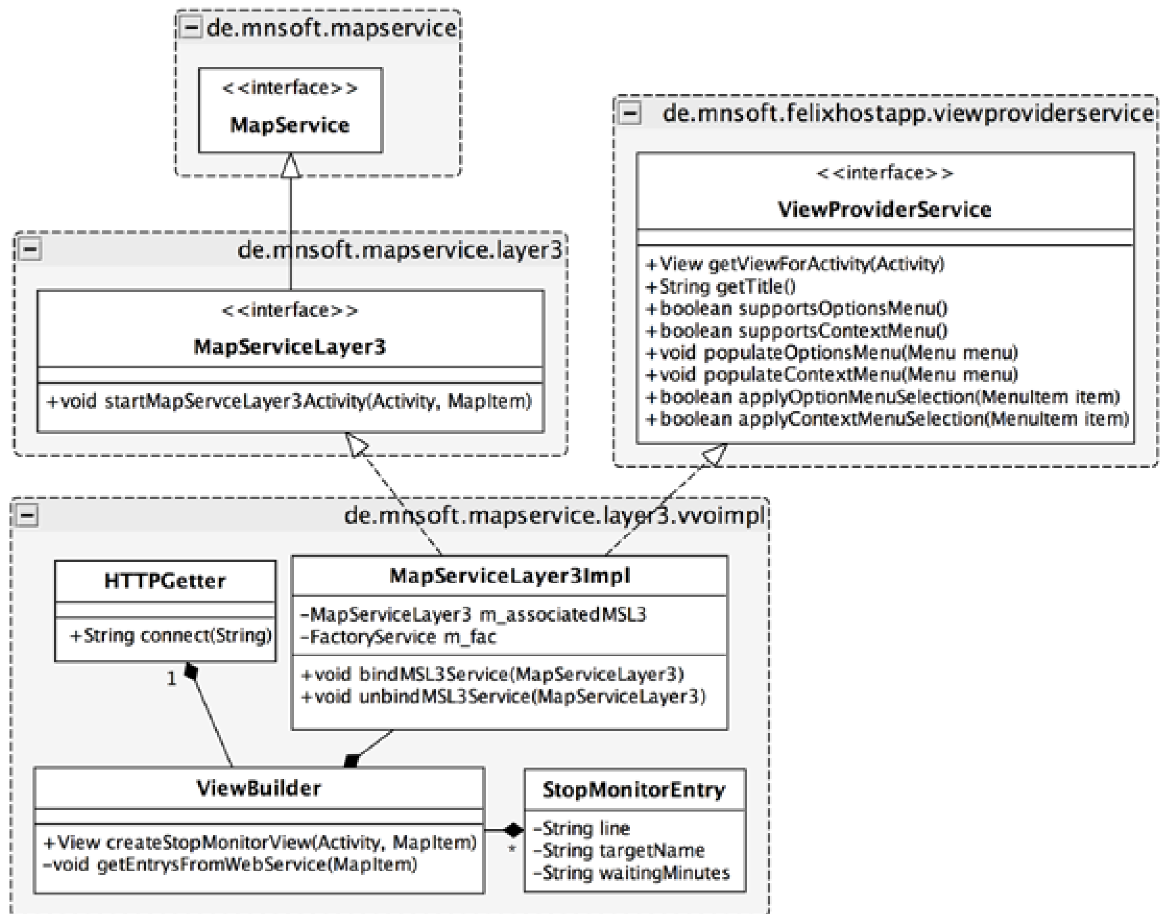


Abbildung 6.6: Klassendiagramm MSL3-Implementierung VVO-Abfahrtsmonitor

Das Bnd-Bundling und der iPOJO Build-Schritt sind ähnlich der MSL1 Implementierung im Abschnitt 6.3.2. Die VVO MSL3 iPOJO-Komponentendefinition sieht noch das Festlegen des filterbaren Property „vendor.name“ mit dem Wert „VVOMSL3“ vor, damit der VVO-MSL1 gezielt nach einem MSL3 vom VVO filtern kann und nur diesen bindet.

Der ViewBuilder erzeugt aus der Abfrage-Antwort eine Scrollview, die die einzelnen Abfahrtsmonitor-Einträge (Klasse StopMonitorEntry) in einzelnen TextViews darstellt. Diese View-erzeugende Methode wird von der ViewProviderService-Methode createViewForActivity(Activity) aufgerufen, welche von der Activity gerufen wird, die diesen MSL3 mit der Methode startMapServceLayer3Activity(Activity, MapItem) gestartet hat. Dies ist meist die Karten-Activity des MapProvider-Bundles.

6.4 Weitere Ergebnisse der prototypischen Umsetzung

In diesem Kapitel sollen einige zusätzlich erreichte Ergebnisse der prototypischen Umsetzung vorgestellt werden. Zunächst wird das Bundle-Management-Bundle beschrieben, welches eine GUI zum bequemen Nachladen von Bundles beinhaltet. Danach wird das iPOJO-Eclipse-PlugIn beschrieben, welches im Rahmen dieser Arbeit für den Einsatz der Bundles auf Android angepasst wurde. Abschließend soll über die Möglichkeiten des Signierens und Optimierens der Bundles gesprochen werden.

6.4.1 Bundle-Management und Bundle Repository

Um die dynamische Erweiterbarkeit für den Nutzer komfortabel zugänglich zu machen, existiert ein Apache Felix Sub-Projekt namens Apache Felix OSGi Bundle Repository, eine Implementierung des OSGi Bundle Repository (OBR).
Dabei ist OBR keine offizielle OSGI-Spezifikation. Richard S. Hall (Apache Felix Project) entwickelte es und Peter Kriens, Mitglied des OSGi Konsortiums, nahm den Ansatz auf und erstellte ein zentrales Repository auf der OSGi Webseite [131].

Dieses OSGi Bundle ermöglicht ein einfaches Deployment der Komponeten, indem es ermöglicht, Bundles von einem Bundle-Repository aufzulisten, herunterzuladen und zu installieren. Dabei ist das Bundle-Repository lediglich ein Webserver, der eine XML-basierte Beschreibung gemäß OSGi RFC-0112 über die bei ihm verfügbaren OSGi Bundles und meist auch die Bundles selbst anbietet [132]. Die gesamte Logik zum Umgang mit diesem Repository ist bei OBR clientseitig im OBR-Bundle selbst positioniert. Es werden also keine besonderen Anforderungen an das Bundle Repository gestellt, bis auf die XML-Beschreibung der verfügbaren Bundles.

OBR ermöglicht das Hinzufügen beliebig vieler Bundle-Repository URLs, wodurch Bundle-Abhängigkeiten Repository-übergreifend aufgelöst werden können. Dadurch sind zentralisierte und dezentrale Deployment-Konzepte umsetzbar. Im Beispiel des MapService-Viewers wäre es sinnvoll, die Bundles, die nur die Service-Interfaces beschreiben, die grundlegenden Bundles und den AppStarter in einem Repository anzubieten. Die jeweiligen Service-Implementierungen (MapService oder MapProvider) können von ihren Anbietern in eigenen Bundle-Repositorys angeboten werden. Alternativ könnten auch die MapProvider-Bundles und die MapService-Bundles in jeweils separate zentrale Repositorys untergebracht werden. Darüber hinaus ist es möglich, in der Meta-Beschreibung eines Repositorys (repository.xml) Verweise auf andere Repositorys unterzubringen. Dadurch ermöglicht OBR föderierte Repositorys.

Ein gutes Beispiel für die XML Beschreibung eines Repository ist das Apache Felix-eigene Bundle Repository[133]. Die Beschreibungsform verallgemeinert etwas vom normalen OSGi Sprachgebrauch, um der Entwicklung der OSGi-Technologie Rechnung zu tragen.

Bundles heißen „Resource", angebotene Packages heißen „Capability" und Benötigte heißen „Requirements". Die Requirements werden durch OSGi-Filter-Ausdrücke (LDAP) beschrieben. Zwei sehr wichtige Einträge sind der eindeutige „Symbolic Name" und die Version des Bundles, da sich OBR's Update-Prozess daran orientiert. Eine gute Einführung zum Aufbau der XML Datei findet sich unter [134]. Da viele der einzutragenden Informationen aus der Manifest-Datei der Bundles abgeleitet werden können, hat Peter Kriens (der auch Bnd entwickelt) ein Tool namens OSGi OBR BIndex entwickelt, welches die Generierung der XML Beschreibung aus der Manifest-Datei ermöglicht.(Jar: [135], Sources(SVN): [136])

6.4.1.1 OBR Client

Das Bundle-Management Bundle ist eine GUI für den OBR Client. Der OBR Client bedient sich des Bundles „Apache Felix OSGi Bundle Repository" (OBR) in Version 1.4.1 [137], welches von der Host-Anwendung bei Start von Felix mitgeladen wird (wie dies auch bei FileInstall und iPOJO der Fall ist).

Da aber zum automatisierten Installieren, Starten, Stoppen und Deinstallieren FileInstall zum Einsatz kommt, wird von OBR nicht der vollständige Deployment-Mechanismus benutzt. Stattdessen wird das Ergebnis des Resolving-Prozesses für die vom Nutzer aus einer Liste (Inhalt des Bundle Repository) ausgewählten Bundles verwendet, um die URLs der geforderten, benötigten und optionalen Jar-Dateien zu erhalten und sie herunterzuladen. Der Downloadprozess legt die neuen Bundle-Jars in den von FileInstall überwachten Ordner. FileInstall führt dann die Installation und den Start durch.

Um ein installiertes Bundle wieder zu entfernen, wird kein OBR Mechanismus verwendet, sondern das zum Löschen ausgewählte Bundle wird lediglich aus dem FileInstall-überwachten Ordner gelöscht. Dies regt FileInstall dazu an, das Bundle zu stoppen und zu deinstallieren.

Dieses Verfahren ist sinnvoller als das Standardvorgehen von OBR, die Bundles direkt in den Cache von Felix zu laden und dort zu starten. Würde die Host-Anwendung beispielsweise wegen eines Absturzes den Felix-Cache vollständig löschen, müssten die auf diese Weise installierten Bundles erneut heruntergeladen und installiert werden. Durch den vorgestellten Mechanismus werden in diesem Fall von FileInstall alle im überwachten Ordner befindlichen Bundle-Jars installiert.

6.4.1.2 Bundle Repository

Um für das Bundle Repository die repository.xml zu generieren, wurde Peter Kriens OSGi OBR BIndex verwendet. Damit BIndex aus dem Bundle Manifest ausreichend Informationen lesen kann, um die Bundles gut zu beschreiben, sollte die Bnd-Datei (und damit indirekt die Manifest-Datei) des Bundles angepasst werden. Listing 6.3 im Anhang A3 zeigt ein Beispiel hierfür.

Ein einfacher Weg, das Bundle Repository anzulegen, ist der folgende: die fertigen Bundle-Jars werden in einen Ordner kopiert, zum Beispiel namens „bundles“, wobei die Jars dann in einen gleichnamigen Ordner im Webspeicher abgelegt werden.

BIndex generiert mittels Konsolenaufruf „java -jar bindex.jar bundles/*.jar“ die repository.xml, welche dann ebenfalls auf den Webspeicher geladen wird (eine Verzeichnisebene höher als das bundles-Verzeichnis). Listing 6.4 im Anhang A3 zeigt den dem Beispiel entsprechenden Eintrag in der repository.xml.

6.4.2 Eclipse iPOJO-PlugIn - Android Edition

Der zweistufige Build-Prozess von iPOJO-basierten OSGi-Bundles, welcher im Kapitel 6.3.2 erwähnt wurde, kann durch das Eclipse iPOJO PlugIn von Clement Escoffier vereinfacht werden. Durch dieses Eclipse PlugIn kann per Rechtsklick auf die metadata.xml unter Nutzung der im gleichen Verzeichnis befindlichen <projektname>.bnd Datei das Bnd-Bundling und der iPOJO Buildschritt in einem Durchgang ausgeführt werden.

Da dieses PlugIn längere Zeit nicht mehr aktualisiert wurde und beim Nutzen von Bundles auf Android noch ein dritter Build-Schritt (das „dexen“ des Bundles mit dem SDK-Tool „dx“) notwendig ist, wurde im Rahmen dieser Diplomarbeit das quelloffene iPOJO-PlugIn aktualisiert und bzgl. des „dx-Build-Schrittes“ erweitert.

Im gleichen Zuge wurde eine komfortable Möglichkeit geschaffen, in einer *.shell Datei angelegte Shellbefehle von Eclipse aus auszuführen. Voraussetzung ist, dass Eclipse von der Konsole aus gestartet wird. Die Funktion wurde in der Entwicklungszeit dazu genutzt, um Bundle-Jars auf den Emulator in das FileInstall-Verzeichnis zu laden (install.shell) oder sie zu entfernen (uninstall.shell). Für jedes Projekt (= Bundle) wurden nach Bedarf dafür separate *.shell Dateien angelegt.

Dieser flexible Mechanismus konnte abschließend auch für das Signieren und Optimieren (vgl. Kap. 6.4.3) verwendet werden. Das modifizierte PlugIn ist Teil der Anhangs-CD dieser Arbeit und auch im Internet verfügbar [113]. Alternativ kann dieser Build-Prozess auch mit Build-Tools wie Ant oder Maven automatisiert werden. Zum Debuggen ist das PlugIn jedoch recht nützlich.

6.4.3 Signieren und Optimieren von OSGi-Bundles

Damit eine Android-Anwendung auf einem echten Endgerät und nicht nur auf dem Emulator ausgeführt werden kann, muss das exportierte Android-Anwendungs-Paket (*.apk) zuvor digital signiert werden. Das hierfür notwendige Vorgehen wird auf der Android-Entwickler-Webseite beschrieben [138].

Das Android Eclipse-PlugIn (ADT) erleichtert die hierfür notwendige Erzeugung eines Keystores stark. Für eine Release-Version genügt der Einsatz des debug-Keystore nicht, sodass für ein Release ein separater Release-Keystore angelegt und mit einem zu einem Alias zugewiesenen Passwort versehen werden muss. Letzteres schützt den eigentlichen privaten Schlüssel separat. Soll die Anwendung über den Google Market (Googles Pendant zum Apple App Store) vertrieben werden, muss das Ablaufdatum des Signierschlüssels nach dem 22.Oktober 2033 liegen.
Der MD5 Hash dieses Signierschlüssels muss verwendet werden, um einen Maps-API-Key zu beziehen, insofern Google Maps verwendet werden soll [139]. Da dieser Schlüssel mit der signierten Host-Anwendung verbunden ist, muss das Bundle, welches den MapProvider Google Maps implementiert, den Maps-API-Key von der Host-Anwendung beziehen.

Zu Performancezwecken hat Google das SDK-Tool zipalign entwickelt, mit dem die fertig exportierten und signierten Android-Anwendungen, aber auch normale Jar-Dateien, für den Einsatz auf dem Endgerät optimiert werden können [140]. Da dies sowohl für die Host-Anwendung als auch für die einzelnen Bundles empfehlenswert ist, müssen die Bundles ebenfalls digital signiert werden, um dann mit dem zipalign-Tool bearbeitet zu werden. Unsignierte Dateien können laut Google mit ,‘zipalign“ nicht optimiert werden.

Für die Host-Anwendung werden das Signieren und das Optimieren durch ADT beim Export automatisch durchgeführt. Für die OSGi-Bundles muss dies manuell in der Shell gemacht werden, allerdings kann der Prozess auch durch Ant oder Maven automatisiert werden. Listing 6.5 zeigt das Kommando zum Signieren und das Kommando zum Optimieren des signierten Bundles. JAVA_HOME muss hierfür in der PATH Variable der Shell eingetragen sein. Der Parameter 4 optimiert die Bundles für den Einsatz auf 32 bit Endgeräten. Das modifizierte iPOJO-Eclipse-PlugIn kann diese beiden Befehle in Form von *.shell Dateien verarbeiten.

Listing 6.5: Shell-Befehle für das Signieren und Optimieren

```
jarsigner -keystore <Pfad zum Android KeyStore> -storepass <KeyStorePasswort> -keypass
<KeyPasswort> bundle.jar <AliasName>

zipalign -c -v 4 <Name der Jar-Datei>
```

6.5 Evaluation der Implementierung

Aufgrund der umfangreichen Vorüberlegungen und der Korrelation zwischen Konzept und OSGi-Technologie lief die Implementierung des Prototypen verhältnismäßig problemarm ab.

Nach der Überwindung von Problemen mit der Zusammenarbeit von Android-Plattform, der DalvikVM und der OSGi-Implementierung in Zusammenarbeit mit den Entwicklern von Apache Felix, konnte der Prototyp zügig fertig gestellt werden. Trotzdem gab es einige unerwartete Schwierigkeiten, auf die nun kurz eingegangen werden soll.

Das während der Entwicklung mühsamste war die fehlende Debugging-Möglichkeit bei den Bundles. Während eine normale Android-Anwendung sowie auch die Hostanwendung normal im Eclipse-Debugger gedebuggt werden können, ist dies für die OSGi Bundles zur Zeit leider nicht der Fall. Zur Ausführungszeit liegen die Bundles im Emulator als Jar-Dateien vor, sodass Haltepunkte und Variablenbelegungen nicht einsehbar sind. Dies erschwert die Fehlersuche und erzwang, das Debugging über Konsolenausgaben durchzuführen.

Im gleichen Zuge ist zu erwähnen, dass auch das Unit-Testing der Bundles nicht möglich war. Der Code des Prototypen ist also ungetestet. Die Fehlersuche wurde durch Probieren und durch Nutzerinteraktion durchgeführt. Es ist daher nicht ausgeschlossen, dass Fehlfunktionen, unerwartetes Verhalten oder sogar Programmabstürze auftreten können.

Um Nebenläufigkeitsprobleme bei der GUI-Erzeugung (in einer ViewProvider-Service-Implementierung) in separaten nicht-UI-Threads zu vermeiden, wurde Androids eigener Thread-Handler-Mechanismus [141] verwendet. Ohne dieses Vorgehen friert die GUI zu lange ein und das Android-System bietet dem Nutzer das Abbrechen der Anwendung an. Anhand der Befolgung von Googles Responsiveness Richtlinien [142] konnte dies durch sofortige Anzeige einer Wartenachricht-View und deren spätere Ersetzung durch die in einem anderen Thread erzeugte View des Bundles vermieden werden.

Ein Performance-Problem ist der erste Programmstart der Hostanwendung. Da neben der Android-Anwendung selbst noch das Felix Framework und die integrierten Basis-Bundles installiert und gestartet werden müssen, dauert der erste Start mit ca. 20 Sekunden recht lange. (Gemessen auf dem G1) Wird dann der MapService-Viewer gestartet, werden zunächst alle benötigten iPOJO Komponenten der benötigten installierten Bundles gestartet, wodurch dieser Start ca. 10 Sekunden bis zur Anzeige der Karte benötigt. Ist die Anwendung einmal gestartet, ergibt sich für den Nutzer jedoch kein Unterschied in der wahrnehmbaren Performance im Vergleich zu

einer normalen Anwendung. Durch Befolgung der Google Performance Richtlinien [143], dem sinnvollen Einsatz von Multithreading und durch OSGi-spezifische Optimierungsmöglichkeiten könnte der Prototyp an diesem Punkt noch verbessert werden. Durch die Auslagerung der Felix-Initialisierung in einen Android-Service ist dieser zeitaufwendige Startvorgang jedoch nur einmalig notwendig. Wird die Host-Anwendung normal geschlossen, bleibt der Android-Service (Klasse FelixService) im Hintergrund gestartet. Der nächste Start der Host-Anwendung hat dann eine vernachlässigbar kurze Startdauer.

Das aktuelle Modell zur Erzeugung von Activities (ActivityService) ist vermutlich noch ausbaufähig, was bei der nachträglichen Einführung von Options- und Kontextmenü deutlich wurde. Das Hinzufügen verursachte Änderungen im ViewProviderService, was Änderungen an den implementierenden Bundles zur Folge hatte. Durch die strikte OSGI-Versionierung und die Zulässigkeit mehrerer parallel installierter Bundles ist es normalerweise kein Problem, dem robust zu begegnen. Da aber der ActivityService aus gegebenen Anlass (vgl. Kap. 6.1) fester Bestandteil der Host-Anwendung ist, gilt diese Flexiblität hier nicht. Zwar wird der ActivityService von der Host-Anwendung ebenfalls versioniert exportiert, jedoch ist die Parallelität verschiedener Versionen nicht ohne weiteres möglich, da die exportierte Version an die Version der Host-Anwendung geknüpft ist.

Es müsste zur Erstellung eines ausgereifteren ActivityService untersucht werden, welche UI-Elemente noch existieren, die eine direkte Integration in die Activity-Klasse erfordern, so wie dies bei Options- und Kontextmenü der Fall war. In diesem Fall müssen Einhängepunkte geschaffen werden, welche mittels des ViewProviderService der Activity mit Inhalt gefüllt werden.

6.6 Fazit

In diesem Kapitel wurde die Umsetzung der Android Host-Anwendung, der Dienstklassen (MapService-Layer-Interfaces), des MapProvider-Interfaces und die beispielhafte Umsetzung eines MapService-Layer 1, eines MapService-Layer 3 und eines MapProvider-Bundles diskutiert. Eingangs wurden die Besonderheiten und Probleme beim Einsatz von OSGi auf dem Android beschrieben, welche Einfluss auf die Gestaltung von Host-Anwendung und Bundles nahmen. Trotz der dort beschriebenen Widrigkeiten konnte die Entwicklung des Prototypen erfolgreich abgeschlossen werden. Darüber hinaus konnten noch weitere Ergebnisse der Implementierung, wie das Management-Bundle und das abgeänderte iPOJO-Eclipse-PlugIn erstellt und hier besprochen werden.

Was mit dem Prototypen erreicht werden konnte und inwieweit er das Konzept validiert, wird im Kapitel 7 diskutiert.

7 Evaluation

In diesem Kapitel wird zusammengefasst, was mit dem Prototypen des Implementierungskapitels erreicht wurde und inwieweit die Anforderungen des Kapitels 3 erfüllt werden konnten. Im Fazit wird darauf eingegangen, wie gut der konzeptionierte und implementierte Ansatz für die Lösung der Aufgabenstellung geeignet ist. Abschließend wird aufgezeigt, was bereits erreicht wurde und an welcher Stelle noch weiterentwickelt werden könnte.

7.1 Erfüllung der Anforderungen

Die Anforderungen gliedern sich in funktionale und nicht-funktionale Anforderungen sowie Anforderungen an die Funktionen im Rahmen des Beispielszenarios. Es soll in diesem Kapitel aufgezeigt werden, in welchem Umfang die prototypische Umsetzung diesen Anforderungen gerecht wird, in dem zu jeder Anforderung jene Aspekte des Ansatzes beschrieben werden, welche die jeweilige Anforderung erfüllen.

7.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen sind in die Hauptaspekte Modularisierung, Einbindung von MapServices und MapProvider-Funktionalität unterteilt. Gemäß der Gliederung, die in Kapitel 3.3 eingeführt wurde, sind diese Aspekte in Teilfunktionen zerlegt.

7.1.1.1 F_1: Modularisierung

F_1.1 - Modularisierter Aufbau: Die Modularisierung wird durch die OSGi-Bundles im Allgemeinen und durch die Teilung in Service-Interface-Bundles und implementierende Bundles im Speziellen erreicht.

F_1.2 - Kapselung: Ein PlugIn ist ein Bundle, welches mindestens ein in einem Interface-Bundle definiertes Service-Interface implementiert.

F_1.3 + 1.4 - Abhängigkeiten der MapService-Bundles: Eine komposite Diensterbringung kann durch die Kombination mehrerer MapService-Layer Bundles (z.B. MSL1 und MSL3) erreicht werden. Die Abhängigkeiten zwischen bestimmten Implementierungen (im Prototypen zwischen den MSL1 und den MSL3 der MapServices Qype und VVO) können durch Filter-Ausdrücke, die das Binden der iPOJO-Komponenten beeinflussen, formuliert werden.

F_1.5 - Kapselung der MapProvider: Das PlugIn-Konzept wurde von den MapServices auch auf die MapProvider ausgedehnt. Anhand des Beispiels von Google Maps im Prototypen könnte so auch OpenStreetMap integriert werden.

F_1.6 - 1.8 - Modulmanagement: Das Modulmanagement wird durch das Bundle-Management-Bundle und durch das von ihm verwendete OBR-Bundle bereitgestellt.

7.1.1.2 F_2: Einbindung der MapServices

F_2.1, 2.2, 2.4 - Unabhängigkeit der kartendarstellungsbezogenen MapService-Module: Die Einbindung der MapServices wird durch die MapService-Layer Service-Interfaces vereinheitlicht.

F_2.3 - Unabhängigkeit vom Webservice-Protokoll: Die Einführung des Globaltypes Services ermöglicht Bundle-übergreifende Interoperabilität. Die Details des Zugriffs und die Datenformate sind Angelegenheit des Bundles und sind nicht eingeschränkt.

F_2.5 - Keine Anpassung der MapServices: Die MapServices müssen für eine Einbindung nicht geändert werden. Lediglich die bezüglich der Dienstklasse geforderte Art von Information muss auf beliebige Art bezogen werden können. Eine Änderung des MapServices hat meist auch eine Anpassung des MapService-Bundles zur Folge. Dafür sind die strikte Versionierung und das einfache Hot Deployment zwei nützliche Mechanismen.

7.1.1.3 F_3: MapProvider

F_3.1 - Kapselung der MapProvider: Die MapProvider können auf ähnliche Weise modularisiert und eingebunden werden wie die MapServices.

F_3.2 - Unabhängigkeit der MapProvider-Bundles: Je nachdem, ob es zulässig ist, kann die Bibliothek für den Zugriff auf das Kartenmaterial in das Bundle integriert werden. Im Fall von Google Maps nicht dies nicht zulässig, was aber kein Problem darstellt, da Android diese Bibliothek mitliefert.

F_3.3 - Funktionsumfang des MapProvider-Bundles: Die Darstellung von MapItems und auch einer Route (Kette von MapItems) ist Aufgabe des MapProviders. Damit ist eine zusätzliche Aufgabe neben der Kartendarstellung die Visualisierung von MSL1 und MSL2.

F_3.4 - Eigenschaften vergleichbar zu Android Maps- Anwendung: Durch Nutzung von MapActivity und MapView entsprechen die Eigenschaften der Android-eigenen Maps-Anwendung.

F_3.5 - Lizenzen: Die aktuelle Nutzung der MapProvider API von Google Maps verletzt nach aktuellem Kenntnisstand keine Lizenzvereinbarungen.

F_3.6 - Quelloffenheit der API: Die MapProvider API von Google Maps ist leider nicht quelloffen.

7.1.2 Nicht-Funktionale Anforderungen

Dieses Kapitel beschreibt die Erfüllung der nicht-funktionalen Anforderungen, welche oft dank OSGi im Prototypen erreicht werden konnten. Diese beziehen sich auf Eigenschaften der Integration von MapServices und MapProvider. Die Integration soll dynamisch, flexibel, erweiterbar, austauschbar und aktualisierbar sein und ein dynamsiches Deployment ermöglichen. Weitere Anforderungen an das System waren Performance, Resourcenschonung und Caching.

7.1.2.1 NF_1: Flexibilität

Ein MapService-Bundle hat bezüglich seiner Implementierung alle Freiheiten (Zugriffsmechanismen etc.). Diese Details sind im Bundle gekapselt. Nach außen hin ist das Bundle durch die Methoden des Service-Interface-Bundles, welches es implementiert, ansprechbar. Diese MapService-Interface-Bundles sind separate Bundles, welche jeweils die Beschreibung eines MapService-Layers beinhalten. Damit liegen Schnittstellenspezifikation und Umsetzung eines MapService-Bundles in getrennten Bundles vor. Die Service-Interfaces sind dadurch nicht an bestimmte Umsetzungsszenarien gebunden.

Um die Integration in die MapProvider-Karten und die Interoperabilität der verschiedenen MapService-Bundles untereinander zu gewährleisten, wurde ein Bundle zur Bereitstellung der global einheitlichen Datentypen wie z.B. MapItem oder MapSerivceFilterOption eingeführt (Globaltypes-Bundle). Die Service-Interface Bundles nutzen diese allgemeinen, anbieterunabhängigen Datentypen, was die MapService-Bundles dazu zwingt, ihre inneren Daten standardkonform nach außen anzubieten.

7.1.2.2 NF_2: Erweiterbarkeit

Die vorliegende Infrastruktur kann sehr einfach um weitere Module erweitert werden. Der Entwickler muss nur entscheiden, welcher Art sein Dienst ist (welcher MapService-Layer bzw. ob es ein MapProvider ist) und ob er eine eigene GUI anbieten soll (ViewProviderService) oder vom MapProvider dargestellt wird (MapService-Layer 1 und 2). Die zur Integration notwendigen Methoden werden durch die jeweiligen Service-Interfaces vorgegeben.

7.1.2.3 NF_3: Dynamisches Deployment

Hot Deployment und die damit einhergehende Laufzeitdynamik sind Vorzüge, welche durch die OSGi Technologie abgedeckt werden. Das Bundle Management Bundle zeigt, wie einfach die Umsetzung dieser Anforderung mit Hilfe von OSGi Bundle Repository erfüllt werden konnte. Download und Installation neuer Bundles sowie das Entfernen vorhandener sind zur Laufzeit per simpler Nutzerinteraktion möglich.

7.1.2.4 NF_4: Dynamische Aktualisierbarkeit

Da sich die Implementierungen an die versionierten Service-Interfaces halten, ist der Austausch (wie beispielsweise bei einer Aktualisierung) einer konkreten Implementierung problemlos möglich. Die Implementierungen haben eine eigene Versionierung und müssen angeben, welche Version des Service-Interfaces sie implementieren. Durch die strikte, explizite Versionierung werden Konflikte bei der Aktualisierung der Module vermieden.

7.1.2.5 NF_5: Austauschbarkeit

Auf Basis desselben Mechanismus wie bei der Erfüllung von NF_4 ist auch der komplette Austausch eines Bundles durch eine Implementierung eines anderen Anbieters möglich. Selbstverständlich kann gleichzeitig eine Vielzahl von Bundles installiert sein, welche dasselbe Service-Interface implementieren. Der Nutzer kann im Beispiel des MapService-Viewers aus den vorhandenen Bundles die zu verwendenen auswählen. Logischerweise ist es nicht vernünftig, zwei MapProvider gleichzeitig zu benutzen, weshalb der MapService-Viewer diese Möglichkeit ausschließt. Sehr wohl aber können nahezu beliebig viele MapSerivces verschiedenster Layer gleichzeitig in einer MapProvider-Karte dargestellt werden.

7.1.2.6 NF_6: Performance

Ist die OSGi-Android-Anwendung einmal gestartet, unterscheidet sich die vom Nutzer empfundene Geschwindigkeit und Reaktionsfreudigkeit nicht von einer normalen Android-Anwendung. Auch ist die Verwendung der MapServices in Form von Bundles nicht auffällig langsamer als ohne den „OSGi-Overhead". Für konkretere Aussagen wären Performance-Messungen nötig, die nicht Teil dieser Arbeit sind.

Trotzdem ist die Startzeit der Host-Anwendung und auch die Startzeit des MapSerivce-Viewers beim ersten Start noch verbesserungswürdig. Dank der Auslagerung der OSGi Framework Instanz als Android Service ist das Startverhalten nur bei erstmaligem Start der Hostanwendung wie beschrieben. Da das Framework auch nach Schließen der Host-Anwendung mit allen gestarteten Bundles im Hintergrund verfügbar bleibt, muss es beim Neustart der Host-Anwendung nicht neu gestartet werden. Sollte der Felix-Service aus Speichermangel zwischenzeitlich vom Android-System beendet worden sein, dauert der Neustart so lange wie beim allerersten Start.

7.1.2.7 NF_7: Rechenlast und Abfragehäufigkeit

Rechenlast wird bei den MapServices größtenteils durch das Parsen von XML- bzw. JSON-kodierten MapService-Antworten verursacht. Im Fall von XML wurde in der Implementierung der performante SAX-Parser verwendet, welcher einen vergleichsweise niedrigen Speicherbedarf aufweist. Die MapServices werden nur nach Nutzerinteraktion (Tippen auf einen Punkt in der Karte) abgerufen. Somit wird die Anzahl der Abfragen auf das Notwendigste reduziert.

7.1.2.8 NF_8: Caching

Die MapServices betreiben ein Caching der abgerufenen Informationen. Alle abgerufenen, extrahierten MapItems werden im MapService-Bundle vorgehalten. Die dem MapProvider-Bundle zugrunde liegenden Bibliotheken leisten meist selbstständig ein effektives Caching, wodurch auf eigene Maßnahmen verzichtet werden konnte. Persistenz-Mechanismen wurden keine implementiert. Sie sind aber nachträglich mittels Android-eigenen Persistenz-Mechanismen einfügbar.

7.1.3 Funktionen des Beispielszenarios

Im Beispielszenario sollte der Abruf der VVO Haltestellen und anderer POI und deren Darstellung in der digitalen Karte im mobilen Endgerät erreicht werden. Dafür können im Prototyp durch ein Tippen auf die Karte die MapItems (Haltestellen oder POI) der installierten und ausgewählten MapService-Layer 1 (MSL1) Bundles abgerufen werden.

P_1 - Darstellung der Haltestellen: Die MapItems werden mit Hilfe von Icons („Marker“) in einem Overlay auf der Karte angezeigt. (MSL1 für VVO und Qype)

P_2 - Interaktion mit Haltestelle: Durch Tippen auf eines dieser Icons wird ein mit dem MSL1 assoziierter MSL3 verwendet, um zusätzlich Informationen zu diesem MapItem anzuzeigen. Im VVO-Szenario zeigt der MSL1 die VVO Haltestellen an, der MSL3 ist der Abfahrtsmonitor dieser Haltestelle.

P_3 - VVO-fremde POI Dienste: Um zusätzliche POI unabhängig von VVO Diensten anzeigen zu können, wurde der POI Dienst von Qype.com als MSL1 und ein passender MSL3 zur Anzeige der Informationen entwickelt.

P_4 - Unabhängigkeit zwischen VVO und POI Dienst: Die Qype-Bundles sind völlig unabhängig von den VVO Bundles.

Eine VVO-Verbindungsauskunft und damit einhergehende Routenanzeige in der Karte (W_1 - W_7) konnte im Rahmen des Prototyps nicht realisiert werden. Der Grund hierfür ist die mangelnde Verfügbarkeit eines Verbindungsauskunfts-MapServices seitens der VVO. Alternativ hätte nur die VVO-Webseite ausgelesen werden können, was allerdings nur eine mäßige Möglichkeit des Datenzugriffs ist. Nach aktuellem Kenntnisstand ist ein solcher Dienst aber in Planung, sodass dieser als MSL2, am besten kombiniert mit einem MSL4, in den MapService-Viewer integriert werden kann. Die vorhandene Google Maps MapProvider-Implementierung müsste dann noch um Support für MSL2 erweitert werden.

7.2 Funktionsweise des Prototypen

In diesem Kapitel soll kurz der Ablauf bzw. die Funktionsweise des Prototypen beschrieben werden. Die Realisierung der Anforderungen soll kurz beschrieben und in Form von Screenshots visualisiert werden.

Nach Start der Host-Anwendung namens „Felix“ wird das Felix-OSGi-Framwork mit allen verfügbaren Bundles gestartet. Nach dem Start wird ein Bildschirm zur Auswahl der OSGi-Android-Anwendung angezeigt (Abb. 7.1). Hier werden alle aktiven AppStarter in Form eines Buttons angezeigt. Das Bundle Management und der MapService-Viewer sind also zwei getrennte OSGi-Anwendungen.

Die Bundle-Management-Anwendung ermöglicht die Installation und Deinstallation von Bundles (Abb. 7.2 und 7.3). Dadurch kann Funktionalität wie MapService- bzw. MapProvider-Support hinzugefügt bzw. entfernt werden.

Der Mapservice-Viewer ermöglich die Auswahl zwischen Konfiguration und Start der Kartendarstellung. Bei letzterer wird zunächst ein erklärender Willkommensbildschirm angezeigt (Abb. 7.4). Durch Tippen auf die Karte werden

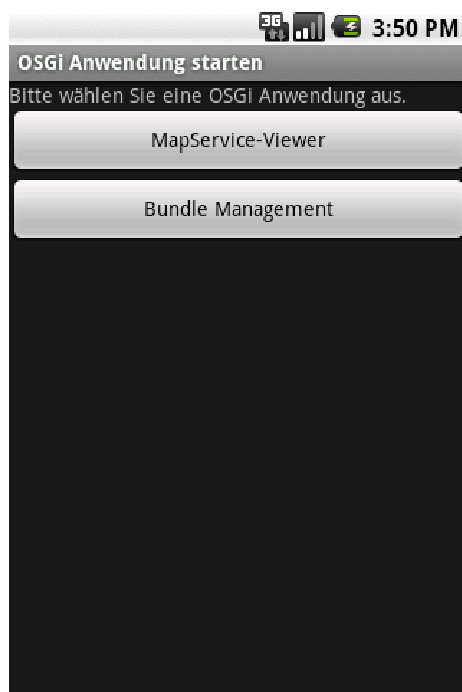


Abbildung 7.1: Auswahl der OSGi-Android-Anwendung in der AppSelectionActivity

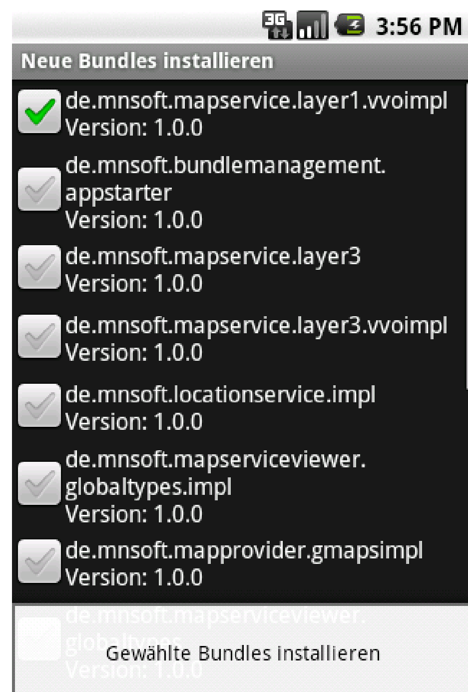


Abbildung 7.2: Installation eines Bundles im Bundle Management

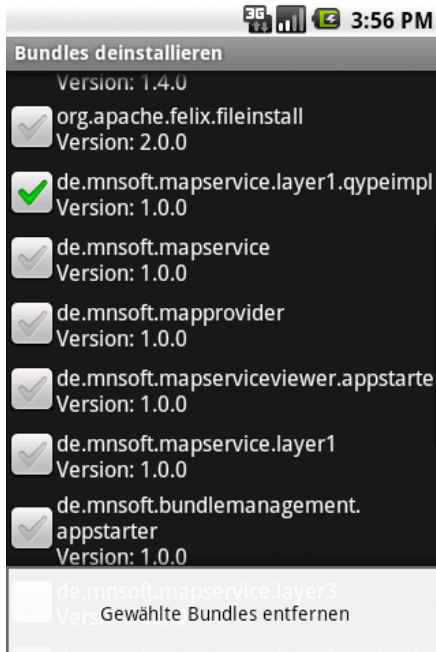


Abbildung 7.3: Deinstallation eines Bundles im Bundle Management

Abbildung 7.4: MapService-Viewer Startbildschirm

MapItems von den Installierten MapServices abgerufen und im Kartenausschnitt dargestellt (Abb. 7.5). Durch Tippen auf ein Haltestellensymbol, welches die Position einer VVO-Haltestelle in der Karte darstellt, wird der zu dieser Haltestelle gehörige Abfahrtsmonitor angezeigt (Abb. 7.6).

Wird auf einen der roten Marker getippt (Qype-MapItem), so werden die Qype-Informationen in einer separaten Activity dargestellt (Abb. 7.7). Dies funktioniert allerdings nur, wenn das Qype-MSL3-Bundle installiert ist. Ist dies nicht der Fall, so wird ersatzweise die Qype-MapItem Information in einem Toast angezeigt (Abb. 7.8).

Abbildung 7.5: Darstellung der MapService-MapItems in der Karte

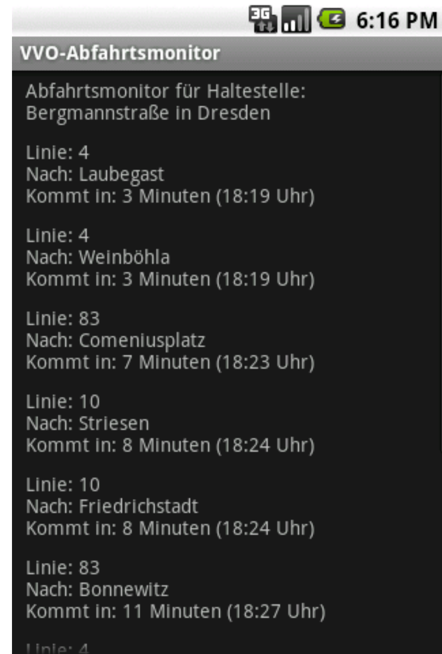


Abbildung 7.6: Der VVO Abfahrtsmonitor

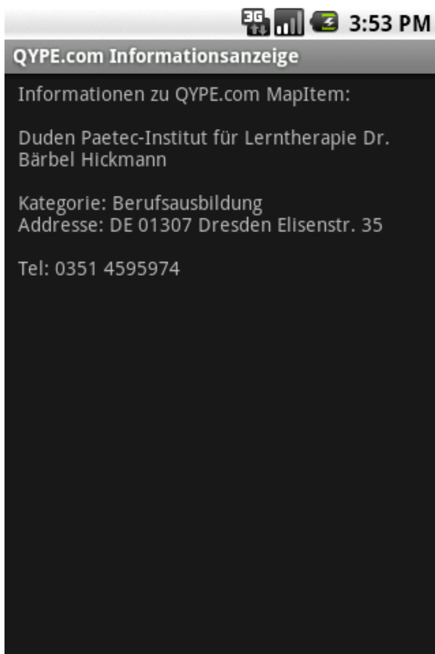


Abbildung 7.7: Die Qype Informationsanzeige

Abbildung 7.8: Die Qype Informationen als Toast (ohne installierten MSL3)

7.3 Fazit

Die Erstellung einer nativen Android-Anwendung, die einen bestimmten MapService nutzbar macht, ist aktuell der beste Weg, um den Unzulänglichkeiten der Webbrowser-basierten Darstellung und Interaktion im Android-Browser beizukommen. Dadurch entsteht jedoch eine Vielzahl von ähnlichen, separaten Anwendungen, die sich fast nur darin unterscheiden, welchen MapService sie integrieren (z.B. Qype.com und der Qype Radar). Diese Programme ähneln sich alle in Aufbau und Funktionsweise, was zu viel Code-Redundanz und dadurch auch zu höheren Entwicklungskosten führt. Diese Situation ist in Abbildung 7.9 beschrieben.

Hier greift der MapService-Viewer-Ansatz: statt einer Vielzahl von ähnlichen Programmen wird mit dem MapService-Viewer eine allgemeine, generische Grundplattform zur Integration von MapServices erreicht. Dabei werden die generischen Teile von den spezifischen Teilen gemeinsam genutzt, wodurch Wiederverwendung einer gemeinsamen Codebasis praktiziert wird (vgl. Abbildung 7.10).

Neben der Kostenersparnis erspart dieses Vorgehen Haupt- und Festspeicher und reduziert die Anzahl von Fehlern, die bei der Neuentwicklung einer separaten

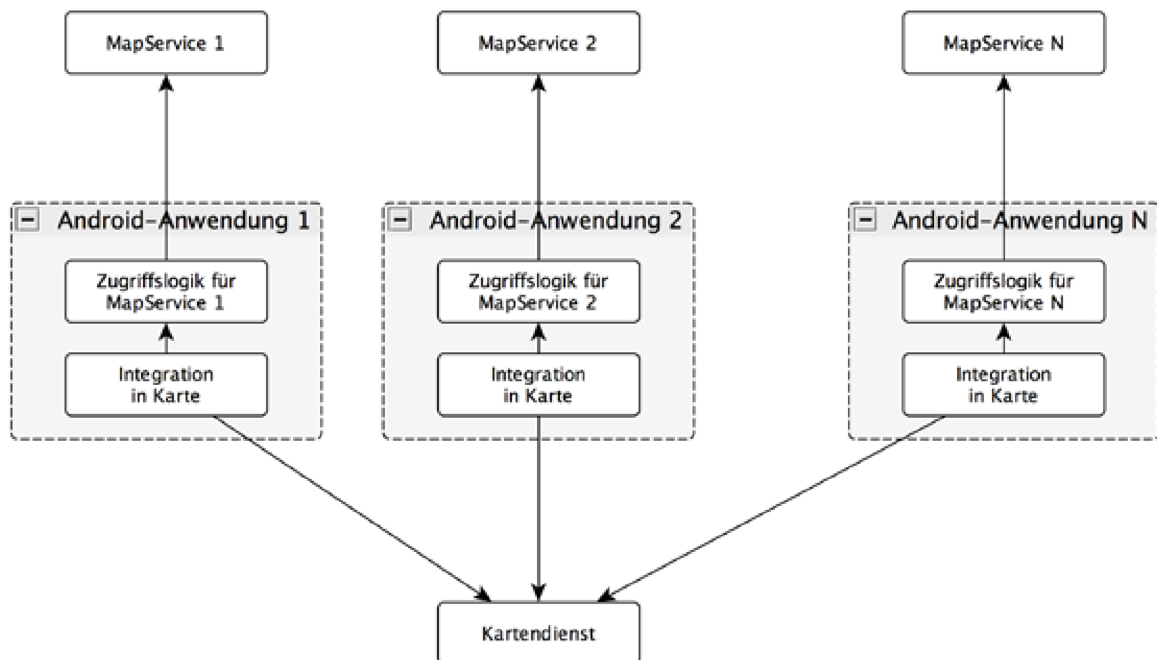


Abbildung 7.9: Separate Android-Anwendungen pro MapService

Anwendung auftreten können. Desweiteren vereinfacht sich die Arbeit des MapService-Anbieters auf die Erstellung eines oder je nach Funktion mehrerer MapService-Bundles (PlugIns), was zu konkreten Kostenersparnissen führt und die MapService-Viewer-Plattform für den realen Einsatz interessant machen könnte.

Der MapService-Viewer ist als zentrale Grundlage weiterentwickelbar und so profitieren alle installierten MapService-Bundles automatisch von Verbesserungen und Bugfixes. Desweiteren ist es im Gegensatz zu den klassischen MapService-einbindenden Android Anwendungen möglich, mehrere MapService-Informationen in Form von MapItems in einer einzigen Karte gleichzeitig darzustellen. Im Prototypen wurde dies demonstriert, indem die POI des Qype-Services und die VVO Haltestellen in der Nähe dieses POI in einer Karte dargestellt werden. Dies ist natürlich beliebig erweiterbar und hat maximal darstellungsbedingte Höchstgrenzen.

Der gefundene Ansatz und so auch die prototypische Umsetzung deckt die Anforderungen des in der Einführung angesprochenen VVO-Beispielszenarios sehr gut ab, ist jedoch noch generischer als der MapService-Viewer selbst. Die Host-Anwendung kann beliebige OSGi-basierte Android-Anwendungen aufnehmen

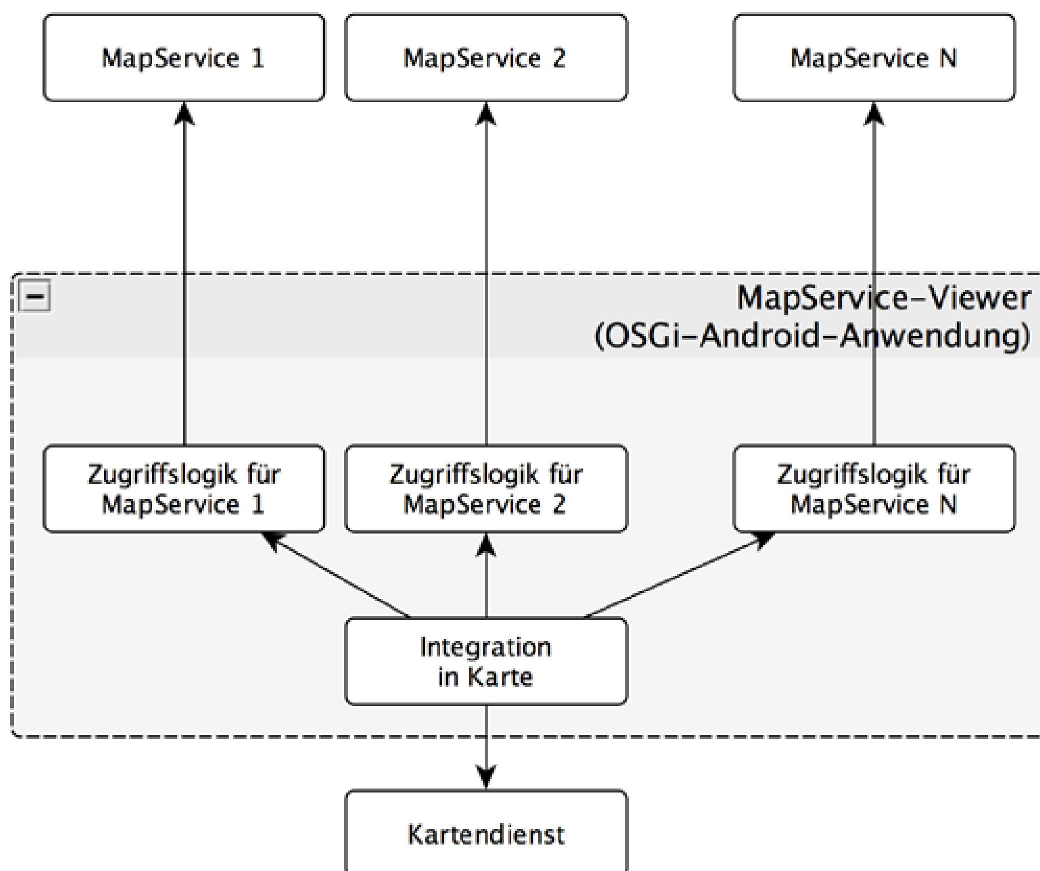


Abbildung 7.10: MapService-Viewer: Generische Plattform für MapServices

und starten. Diese beliebigen Anwendungen genießen dann ebenfalls die Vorteile der Laufzeitdynamik und des Hot Deployment, die vom OSGi-Framework ermöglicht werden. Ein Beispiel hierfür ist die Bundle-Management Anwendung. Diese OSGi-Android-Anwendung ist völlig unabhängig vom MapService-Viewer und ermöglicht die Verwaltung der installierten OSGi-Bundles (vgl. Kap. 6.4.1 und 7.2). Andere Beispielszenarien für Android-Anwendungen, die von Laufzeitdynamik und PlugIn-Konzept profitieren könnten, wären z.B. eine generische Social-network Anwendung mit PlugIns für StudiVZ, Facebook, etc. oder eine Chat-Anwendung, welche die einzelnen Protokolle als OSGi-Bundles einbindet.

Anforderungen an diese Anwendungen sind das Implementieren eines AppStarters mit passender GUI (ViewProviderService). Die Anwendungen müssen zum Starten neuer Activitys den ActivityService verwenden. Sie können alle sonst über die Host-Anwendung installierten Bundles mitnutzen und somit auch Komponenten des MapService-Viewers wiederverwenden.

Ebenso konnte relative Unabhängigkeit von einem konkreten Kartenanbieter (MapProvider) erreicht werden. Weitere Kartenanbieter können mit verhältnismäßig geringem Aufwand in Form neuer MapProvider-Bundles (PlugIns) hinzugefügt werden. Die Unabhängigkeit ist relativ, da man bei der Entwicklung eines MapProvider-Bundles auf die Verfügbarkeit einer entsprechenden Bibliothek für Android angewiesen ist.
Durch die Positionierung der MapService-Zugriffslogik auf dem Endgerät wurde ein zentraler Flaschenhals in Form eines vermittlenden Servers vermieden. Die vom Nutzer empfundene Performance des Prototypen beim Auslesen der MapService-Anworten ist selbst auf dem vergleichsweise leistungsschwachen G1 ausreichend gut (bei Versuchen stets deutlich unter 5 Sekunden). Schwerwiegende Verzögerungen sind in der Praxis daher eher von ausgelasteten MapService-Servern oder schwacher Internetverbindung zu befürchten. Mit zunehmender Zahl gleichzeitig dargestellter MapServices wird sich allerdings die Abfragezeit entsprechend erhöhen. Da es aber aus Gründen der Übersichtlichkeit (kleine Bildschrime mobiler Endgeräte) nicht sinnvoll ist, sehr viele MapServices gleichzeitig zu verwenden, wird dieses Problem in der Praxis wahrscheinlich selten Sorgen bereiten.

7.3.1 Möglichkeiten zur Weiterentwicklung

Im Rahmen dieser Arbeit konnten im Prototypen nicht alle Ideen verwirklicht werden. Ein Ansatzpunkt zur Weiterentwicklung des Konzepts und zur Verbesserung des Prototypen wäre die ortsbezogene Einbindung neuer MapServices. Damit könnte sich ein Nutzer die zu dem aktuell angezeigten Kartenausschnitt verfügbaren (aber noch nicht installierten) MapService-Bundles anzeigen lassen. Aus diesen verfügbaren MapServices kann der Nutzer dann (wie im Bundle-Management Bundle) das gewünschte Bundle zu Download und Installation auswählen. Dafür

müsste jedoch die Beschreibung der MapService-Bundles um Informationen zum abgedeckten Gebiet erweitert werden. Alternativ müsste ein separater Server diese Zuordnung übernehmen und diesbezüglich abfragbar sein. Diese Funktion war einer der Vorzüge des Adapter-Server-Konzepts, auf das zugunsten der Dezentralität des MapService-Zugriffs verzichtet wurde.

Im Konzept ist aktuell noch kein Einhängepunkt für eine Suchwort-Eingabe zur Parametrisierung der MapService-Anfragen vorgesehen. Hierdurch könnte der Nutzer vergleichbar zu Androids eigener Maps-Anwendung nach einem Begriff suchen, der dann in den Abfragen entsprechender MapService-Bundles verwendet wird, um zum Suchbegriff passende MapItems abzurufen. Im Beispielszenario der VVO-Bundles könnte der Nutzer den Namen einer Straße oder Haltestelle eingeben, um die jeweiligen Haltestellen in der Karte anzuzeigen. Dies ist durch das flexible Design der MapService-Bundles mit verhältnismäßig geringem Aufwand umsetzbar. Allerdings wäre dafür noch eine Anpassung des MapProvider-Bundles erforderlich.

Durch mangelnde Verfügbarkeit eines VVO-Verbindungsauskunfts-MapService konnten die Wunsch-Kriterien des Beispielszenarios nicht praktisch umgesetzt werden.

Trotzdem wurden diese Kriterien beim Konzept berücksichtigt, sodass diese Funktion in Form eines MapService-Layer 2 und Layer 4 Bundles nachträglich eingefügt werden kann. Eine entsprechende Anpassung des MapProviders für die Unterstützung von MapService-Layer 2 wäre noch notwendig.

Etwas nachteilig ist die Tatsache, dass die MapProvider-Bundles die Darstellung von MSL1 und 2 übernehmen müssen. Dies macht auch die Entwicklung eines MapProvider-Bundles schwieriger. Optimal wäre die Auslagerung der transparenten Overlayfunktion in ein separates Bundle, welches von beliebigen MapProvider-Bundles verwendet werden kann, um die Darstellung von MSL1 und MSL2 vorzunehmen. Dies setzt jedoch eine Kartenanbieter-unabhängige Implementierung der Overlayfunktionalität voraus, die recht komplex ist.

Im Prototypen wurde stattdessen die bereits existierende Overlay-Implementierung von Google Maps benutzt. Es ist fraglich, ob die Delegation der Overlay-Aufgabe an ein separates Overlay-Bundle aus Performance-Sicht sinnvoll wäre. Für konkrete Aussagen zu dieser Frage müssten weitere Nachforschungen vorgenommen werden.

Im folgenden Kapitel 8 werden alle Erkenntisse dieser Arbeit abschließend kurz zusammengefasst und ein allgemeiner Ausblick gegeben.

8 Zusammenfassung

In diesem Kapitel werden die Ergebnisse der einzelnen Kapitel dieser Arbeit zusammen gefasst. Abschließend werden im Ausblick Anschlusspunkte für weitere Arbeiten gegeben.

8.1 Überblick über alle Kapitel

Im ersten Kapitel wurde die Motivation und ein einleitender Überblick über die Arbeit formuliert.

Bei der Analyse und dem Vergleich der vorhandenen mobilen Plattformen in Kapitel 2 stellte sich Google Android als am besten für diese Arbeit geeignete Plattform heraus. Zusätzlich wurden in diesem Kapitel die Begriffe und Konzepte der Geoinformatik eingeführt. Dabei wurde festgestellt, dass das Open Geospatial Consortium für kartenorientierte Webservices Standardisierungsspezifikationen eingeführt hat und dass diese Ansätze ausschließlich serverseitig orientiert sind. Daraus wurde geschlossen, dass der zu entwickelnde, generischer Ansatz sowohl OCG-standardisierte als auch nicht standardkonforme Webservices unterstützen sollte. Außerdem wurde das Ziel formuliert, den serverseitigen OGC-Ansätzen einen clientseitigen Ansatz gegenüberzustellen. Abschließend wurden die zentralen Arbeitsbegriffe MapService, MapProvider und MapItem definiert.

In der Anforderungsanalyse (Kapitel 3) wurde vom VVO-Beispielszenario auf die allgemeine Problemstellung geschlossen und funktionale sowie nicht-funktionale Anforderungen an Konzeption und Implementierung herausgestellt.

Kapitel 4 - die State-of-the-Art-Analyse - beinhaltet eine ausführliche Untersuchung der technischen Möglichkeiten bezogen auf unterschiedliche Aspekte der Arbeit. Der programmatischen Kartenintegration in einer nativen Android-Anwendung wurde das Konzept der Karten-Webanwendung gegenübergestellt. Aus dem Vergleich resultierte, dass Karten-Webanwendungen für die Anzeige im Android-Browser ungeeignet sind und folglich die Kartendarstellung in einer Android-Anwendung stattfinden muss.

Beim Vergleich der verschiedenen Anbieter von Online-Kartensystemen („MapProvider“) wurde eine Entscheidung zugunsten von „Google Maps“ getroffen, welches zur Zeit am besten in eine Android-Anwendung integriert werden kann.

Die Untersuchung verschiedener vorhandener Umsetzungen im Bereich des Beispielszenarios zeigte, dass sie den hier gestellten Anforderungen nicht genügen.

Weiterhin wurden die Möglichkeiten der dynamischen Softwareentwicklung im allgemeinen und für die in Kapitel 2 ausgewählte Android-Plattform im speziellen analysiert und verglichen. Die OSGi-Service Plattform stellte sich als hierfür sehr gut geeigneter Weg zur Erreichung der Ziele „Modularisierung“ und „Laufzeitdynamik“ heraus. Für Android eignet sich zur Zeit die OSGi-Framework-Implementierung Apache Felix am besten. Dadurch wird ein clientseitiger Ansatz zur (Laufzeit-) dynamischen Integration von Webservices ermöglicht.

Auf Basis der vorhergehenden Kapitel wurden in der Konzeption (Kapitel 5) allgemeine Konzepte und verschiedene mögliche Architekturkonzepte zur Lösung der Aufgabenstellung dargestellt und verglichen. Das abschließend präferierte Konzept, eine native Android-Anwendung über OSGi-PlugIns für einzelne Webservices und Kartenanbieter zu erweitern, wurde verfeinert. Zudem wurde die Klassifikation von kartenorientierten Webservices („MapServices“) in Form des Diensklassenkonzepts beschrieben und im Detailkonzept konkretisiert.

Die prototypische Umsetzung des detaillierten Konzeptes wird in Kapitel 6 (Implementierung) vorgestellt. Es wurde ausführlich beschrieben, wie die dynamische Integration beliebiger MapServices und MapProvider im Detail erreicht werden konnte und somit das Wissen vermittelt, um weitere Dienste in das System integrieren zu können. Dabei wurde auch auf die Besonderheiten und Probleme des Einsatzes von OSGi auf der Android-Plattform und die weiteren Ergebnisse der Implementierung, wie das Management-Bundle zur Verwaltung der Bundles auf dem Endgerät und das abgeänderte iPOJO-Eclipse-PlugIn zur Erleichterung der Entwicklung von iPOJO-basierten OSGi-Bundles für Android separat eingegangen.

Im Kapitel 7 (Evaluation) wurde das Konzept und die erfolgte Umsetzung den Anforderungen aus Kapitel 3 gegenübergestellt. Diesen Anforderungen konnten die Ergebnisse dieser Arbeit bestens gerecht werden. Zudem ist der Ansatz und die Implementierung in hohem Maße generisch, da er nicht nur ermöglicht beliebige MapServices und MapProvider dynamisch in eine Android-Anwendung zu integrieren („MapService-Viewer-Anwendung“) sondern eine Plattform geschaffen wurde, beliebige OSGi-basierte Android-Anwendungen auf dem Gerät auszuführen. Diese profitieren - wie der MapService-Viewer - von den positiven Eigenschaften Laufzeitdynamik, Flexibilität, Erweiterbarkeit, dynamisches Deployment und Wiederverwendbarkeit, die von der OSGi-Technologie ermöglicht werden. Desweiteren wurden in Kapitel 7 die Potentiale zur weiteren Entwicklung identifiziert.

8.2 Ausblick

In diesem Kapitel werden Anschlusspunkte an diese Arbeit beschrieben. Aus diesen Ansätzen könnten Themen für weitere Beleg- bzw. Diplomarbeiten entstehen.

Zwei technische Aspekte dieser Arbeit könnten zu interessanten Folgethemen werden. Zur Lösung des ersten Aspekts ist es notwendig, Wege zur Ermöglichung des Debuggings und Unit Testens von OSGi-basierten Android-Anwendungen zu finden. Diese wichtigen Schritte des Softwareentwicklungszyklus benötigen adäquate softwaretechnische Unterstützung in der Entwicklungsumgebung. Entwicklung und Qualitätssicherung der in dieser Arbeit beschriebenen Softwaremodule könnten auf diese Weise erleichtert werden.

Der zweite Aspekt adressiert die XML-basierte Erzeugung grafischer Nutzeroberflächen, wie sie bei der Entwicklung normaler Android-Anwendungen üblich ist. Es wäre interessant zu untersuchen, wie sich dies auf die Erstellung von GUI für Android-OSGi-Bundles übertragen ließe.

Neben den genannten technischen Aspekten bieten einige konzeptionelle Fragestellungen weitere Anschlusspunkte. So wäre es interessant, für die in dieser Arbeit vorgestellte Systemarchitektur Wege der dynamsichen Dienstevermittlung zu untersuchen. Hierbei sollten Konzepte für integrierte, dynamsiche Deployment-Maßnahmen entwickelt werden.

Es könnte beispielsweise ein orts- bzw. situationsabhängiger Download von Bundles angestrebt werden. In Ubiquitous- und Mobile Computing Szenarien, die auf den Einsatz von SOA-orientierten Diensten (Webservices) bauen, könnten diese Funktionen dem Nutzer integriert vermittelt werden.

Im Beispiel des MapService-Viewer könnten ortsgebunde MapServices je nach aktuellem Ort des Nutzers angeboten werden (vgl. Kap. 7.3.1).

Mit Hilfe des Bundle-Repository Ansatzes könnte auf diese Weise als Gegenstück zum Vertrieb über den Android Market, ein zusätzlicher Softwaredistributionsweg der völlig unabhängig von Google ist, geschaffen werden. Die D-OSGi Technologie könnte in Zukunft diesbezüglich ebenfalls noch interessant werden.

Nicht zuletzt wäre es interessant zu untersuchen, wie die OSGi Service Plattform für andere Programmiersprachen und somit auch für andere mobile Plattformen wie dem Apple iPhone verfügbar gemacht werden könnte, um das in dieser Arbeit beschriebene Konzept auf weitere Plattformen übertragen zu können.

Quellenverzeichnis

Alle Quellenverweise in Form von Webseiten oder online verfügbaren PDF-Dateien wurden zuletzt am 30.10.2009 probehalber heruntergeladen.

[1] Open Handset Alliance
http://www.openhandsetalliance.com/oha_members.html

[2] Ortiz, C. Enrique: A Survey of Java ME Today; November 2007
http://developers.sun.com/mobility/getstart/articles/survey/

[3] Google inc.: Android Developer Challenge
http://code.google.com/intl/de-DE/android/adc/adc_gallery/

[4] The Apache Software Foundation: Apache License, Version 2.0; Januar 2004
http://www.apache.org/licenses/LICENSE-2.0

[5] Bort, Dave : Android is now available as open source, 21.10.2008
http://source.android.com/posts/opensource

[6] Theiss, Bernd: Revolutioniert „Android“ das Handy?; 8.1.2007
http://www.stern.de/computer-technik/telefon/
Handy-Betriebssystem-Revolutioniert-Android-Handy/602079.html

[7] Android Developers: Sensors
http://developer.android.com/reference/android/hardware/Sensor.html

[8] Fey, Jürgen; Jäger, Moritz: T-Mobile G1 startet stark; 04.02.2009
http://www.computerwoche.de/knowledge_center/
mobile_wireless/1886205/index5.html

[9] Schütz, Anja; Kalenda, Florian: Computex: Benq verspricht für 2010 ein Android-Netbook und ein Smartphone; 05.06.2009
http://www.zdnet.de/news/mobile_wirtschaft_computex_benq_verspricht_
fuer_2010_ein_android_netbook_und_ein_smartphone_story-39002365-
41005001-1.htm

[10] Android Developers
http://developer.android.com/

[11] Android Developers: What is Android?
http://developer.android.com/guide/basics/what-is-android.html

[12] Android Developers: Developing In Eclipse, with ADT
http://developer.android.com/guide/developing/eclipse-adt.html

[13] Apple inc.: Apple App Store
http://www.apple.com/de/iphone/appstore/

[14] Nokia: Ovi-Store
https://store.ovi.com/

[15] Jobs, Steve et. al: United States Patent: 7479949, 20.01.2009
http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=%2
Fnetahtml%2FPTO%2Fsearch-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=
PN%2F7479949

[16] Jacobsen, Jens: Lernen vom und für das iPhone, August 2007
http://www.commercemanager.de/magazin/
artikel_1552_iphone_apple_konzept.html

[17] Loganbill, Scott : Webmonkey Maps iPhone App Developers' Frustration, 26.09.2008
http://www.webmonkey.com/blog/
Webmonkey_Maps_iPhone_App_Developer_s_Frustration

[18] Apple Developer Connection: iPhone Developer Programm
http://developer.apple.com/iphone/program/

[19] Apple Developer Connection:: Sign up to become a registerd iPhone Developer
http://developer.apple.com/iphone/program/start/register/

[20] Green, David: Android versus iPhone Development: A comparison; 2.07.2009
http://greensopinion.blogspot.com/2009/07/
android-versus-iphone-development.html

[21] http://pressetext.at/news/090722003/
mobile-apps-werden-bis-2020-wichtiger-als-web/

[22] Heise: Programmierer kritisieren Apples App Store; 15.09.2008
http://www.heise.de/newsticker/meldung/Programmierer-kritisieren-Apples-App-Store-205628.html

[23] Sorrel, Charlie: App Store is a Goldmine; 19.09.2008
http://www.wired.com/gadgetlab/2008/09/app-store-is-a

[24] Bremmer, Manfred. Hill, Jürgen: Smartphone: Die Karten werden neu gemischt; 17.11.08
http://www.computerwoche.de/knowledge_center/
mobile_wireless/1878912/index2.html

[25] Microsoft: Microsoft Software Terms for Windows Mobile 6.0 Software; 2008
http://epsfiles.intermec.com/eps_files/eps_man/943-132.pdf

[26] Microsoft: .NET Compact Framework 3.5 Redistributable
http://www.microsoft.com/downloads/details.aspx
?FamilyID=e3821449-3c6b-42f1-9fd9-0041345b3385&displaylang=de

[27] Microsoft: Windows Mobile 6.5 Developer Tool Kit
http://www.microsoft.com/downloads/details.aspx
?displaylang=en&FamilyID=20686a1d-97a8-4f80-bc6a-ae010e085a6e

[28] Lai, Eric: Microsoft readies challenge to rivals in mobile apps market; 17.05.2009
http://www.computerworld.com/action/article.do
?command=viewArticleBasic&articleId=9133186

[29] Microsoft: Windows Mobile für Entwickler (Registrierung)
http://developer.windowsmobile.com/Help.aspx
?id=fe93abaa-1445-45ef-8416-40a507159c3e

[30] focus.de: Handyproduktion: Nokia hängt alle ab; 24.01.200
http://www.focus.de/finanzen/boerse/aktien/
handyproduktion_aid_234844.html

[31] Golem: Symbian wird erst kostenlos, dann Open Source ; 24.06.2008
http://www.golem.de/0806/60584.html

[32] Nokia Ovi
http://www.ovi.com/

[33] Vollmuth, Jan: Smartphone-Markt: Apple liegt auf Platz drei der Smartphone-Betriebssysteme; 09.12.2008
http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/
softwarekomponenten/articles/156882/

[34] IDG BUSINESS MEDIA GMBH München : Apple überholt Windows Mobile; 08.12.2008
http://www.channelpartner.de/knowledgecenter/tk-business/270113/

[35] Informa Telecoms & Media: Prognose: Android wird das iPhone abhängen; 10. März 2009
http://derstandard.at/fs/o1234509043679/Prognose-Android-wird-das-iPhone-abhaengen

[36] Golem: Apple erhält Multi-Touch-Paten; 27.01.2009
http://www.golem.de/0901/64846.html

[37] Forward Concepts: Marktanteile der Hersteller und der Betriebssysteme: Prognose: 2009 13 % mehr verkaufte Smartphones; 26. Mai 2009
http://www.mobile2day.de/news/news_details.html?nid=38481

[38] Generator Research: Apple reißt bis 2013 die Smartphone-Führung an sich; 14.01.2009
http://www.computerwoche.de/knowledge_center/mobile_wireless/1883945/

[39] Junginger, Markus: Mindestens 18 Android Geräte noch in diesem Jahr; 28.05.2009
http://www.android-os.de/content/view/222/1/

[40] Kretschmann, Thomas: Google: Jede Menge Android-Smartphones; 28.05.2009
http://www.tomshardware.com/de/Android-Google-HTC-Magic,news-242948.html

[41] voip-information.de:Vorteile der „Android"-Plattform gegenüber existierenden Smartphone;
http://www.voip-information.de/android/android-vorteile.php

[42] Kretschmann, Thomas: HTC plant Handys für jedermann - mit Hilfe von Googles Android 14.02.2008
http://www.tomshardware.com/de/
HTC-Android-Google-Windows-Mobile,news-240508.html

[43] ntbooknews.de: Auch Dell Netbooks bald mit Android; 7.05.2009
http://www.netbooknews.de/5034/auch-dell-netbooks-bald-mit-android/

[44] Dr. Köhne, Anja, Dr. Wößner, Michael: Kartenbezugssysteme; 04.02.2007
http://www.kowoma.de/gps/geo/mapdatum.htm

[45] Dr. Köhne, Anja, Dr. Wößner, Michael: NAVSTAR GPS - Geschichtliches; 30.09.2008
http://www.kowoma.de/gps/Geschichte.htm

[46] Dr. Köhne, Anja, Dr. Wößner, Michael: Erreichbare Genauigkeit; 18.06.2007
http://www.kowoma.de/gps/Genauigkeit.htm

[47] Dr. Köhne, Anja, Dr. Wößner, Michael: Kontrollsegment (Bodenstationen); 18.11.2008
http://www.kowoma.de/gps/Bodenstationen.htm

[48] Dr. Köhne, Anja, Dr. Wößner, Michael: Positionsbestimmung ; 29.11.2007
http://www.kowoma.de/gps/Positionsbestimmung.htm

[49] GisWiki: ISO 19107
http://www.giswiki.org/wiki/ISO_19107

[50] GISWiki: Geoinformation
http://www.giswiki.org/wiki/Geoinformation

[51] Geoinformatiklexikon: Geodaten
http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=762

[52] GISWiki: Geodateninfrastruktur
http://www.giswiki.org/wiki/Geodateninfrastruktur#Allgemein

[53] Geoinformatiklexikon : ISO/TC 211
http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=84254593

[54] Open Geospatial Consortium, Inc.® (OGC)
http://www.opengeospatial.org/

[55] GISWiki: Open Geospatial Consortium
http://www.giswiki.org/wiki/Open_GIS_Consortium

[56] OGC: Geography Markup Language
http://www.opengeospatial.org/standards/gml

[57] OGC: OpenGIS Web Map Service (WMS) Implementation Specification
http://www.opengeospatial.org/standards/wms

[58] GeoServer
http://geoserver.org/display/GEOS/Welcome

[59] OpenLayers
http://openlayers.org/

[60] Deegree: deegree - Free Software for Spatial Data Infrastructures
http://www.deegree.org/

[61] MapServer
http://mapserver.org/

[62] OGC: OpenGIS Location Service (OpenLS) Implementation Standards
http://www.opengeospatial.org/standards/ols

[63] Geoinformatiklexikon: WebGIS
http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=1420317245

[64] Google inc: Static Maps-API-Entwicklerhandbuch
http://code.google.com/intl/de-DE/apis/maps/documentation/staticmaps/

[65] Google inc.: Google Maps-API
http://code.google.com/intl/de-DE/apis/maps/

[66] OSM Protocol Version 0.6
http://wiki.openstreetmap.org/wiki/OSM_Protocol_Version_0.6

[67] Geoinformatiklexikon: Gauß-Krüger-Projektion
http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=743

[68] Creative Commons Attribution-Share Alike 2.0 Germany
http://creativecommons.org/licenses/by-sa/2.0/de/

[69] Microsoft: Bing Maps for Enterprise
http://www.microsoft.com/maps/

[70] Yahoo Routenplaner und Verkehrshinwerise
http://de.maps.yahoo.com/

[71] Google Geo Developers Blog
http://googlegeodevelopers.blogspot.com/2008/10/
geocoding-inreverse.html

[72] OpenStreetMap
http://www.openstreetmap.org/

[73] Gramlich, Nicolas: Automatic Android OpenStreetMap Contributor
http://www.anddev.org/openstreetmap_android_contributor-t2902.html

[74] OSM Wiki: Potlatch OSM Editor
http://wiki.openstreetmap.org/wiki/DE:Potlatch

[75] Java OpenStreetMap Editor
http://josm.openstreetmap.de/

[76] OSM Wiki: Merkaartor
http://wiki.openstreetmap.org/index.php/Merkaartor

[77] Creative Commons Attribution-Share Alike 2.0
http://creativecommons.org/licenses/by-sa/2.0/

[78] Android Navigation System
http://www.andnav.org/

[79] Bernhard Harzer Verlag GmbH Karlsruhe
http://www.geobranchen.de/index.php&option=com_geonews&page=details&id=3788

[80] OpenStreetMap stats report
http://www.openstreetmap.org/stats/data_stats.html

[81] Becker, Arno, Pant, Marcus: „Android: Grundlagen und Programmierung“,
dpunkt Verlag; Auflage: 1 (25. Mai 2009),
Sprache: Deutsch,
ISBN: 978-3898645744

[82] Burnette , Ed: „Hello, Android: Introducing Google‘s Mobile Development Platform“,
Pragmatic Programmers; ezember 2008
Sprache: Englisch
ISBN: 978-1934356173

[83] osmdroid: OpenStreetMap-Tools for Android
http://code.google.com/p/osmdroid/

[84] Android OSM Navigation
http://www.nabble.com/
Android-OSM-Navigation-%28AndNav.org%29-tp20994873p20995592.html

[85] Google Maps/Google Earth APIs Terms of Service, 27.05.2009
http://code.google.com/intl/de-DE/apis/maps/terms.html

[86] OpenStreetMap Legal FAQ (Lizenzinformationen)
http://wiki.openstreetmap.org/index.php/Legal_FAQ

[87] Google Maps-API
http://code.google.com/intl/de/apis/maps/

[88] Googel Maps API - Kartenoverlays - Benutzerdefinierte Overlays
http://code.google.com/intl/de/apis/maps/
documentation/overlays.html#Custom_Overlays

[89] OGC KML
http://www.opengeospatial.org/standards/kml/

[90] GeoRSS
http://georss.org/Main_Page

[91] Google inc. : Google Mapplets-Entwicklerhandbuch
http://code.google.com/intl/de/apis/maps/documentation/ mapplets/guide.html

[92] Google inc.: Gadgets API
http://code.google.com/intl/de/apis/gadgets/

[93] OSM WIKI: OpenLayers
http://wiki.openstreetmap.org/wiki/OpenLayers

[94] OpenLayer Sourceforge
http://sourceforge.net/projects/openlayer/

[95] Cloudmade: Web Maps Lite
http://developers.cloudmade.com/projects/show/web-maps-lite

[96] Palk, Justin: Displaying maps with OpenLayers; 24.12.2008
http://www.linux.com/archive/feature/154814

[97] Gramlich, Nicolas: AndNav in a Nutshelll; 03.11.2008
http://www.andnav.org/index.php/de/component/content/article/25-andnav1/46-andnav-nutshell

[98] Patrice Bernard, Frank Van Caenegem: Métro-Webseite
http://metro.nanika.net

[99] DELFI - Durchgängige ELektronische FahrplanInformation
http://www.delfi.de/

[100] Shankland, Stephen: Sun starts bidding adieu to mobile-specific Java; 19.10.2007
http://news.cnet.com/8301-13580_3-9800679-39.html?part=rss&subj=news&tag=2547-1_3-0-5

[101] Android Developers: Android 1.5 NDK, Release 1
http://developer.android.com/sdk/ndk/1.5_r1/index.html#overview

[102] Seeberger, Heiko: OSGi in kleinen Dosen – Bundles und Life Cycle; Feb. 2009
http://it-republik.de/jaxenter/artikel/OSGi-in-kleinen-Dosen-%96-Bundles-und-Life-Cycle-2118.html

[103] Gruhn, Volker; Thiel;Andreas: „Komponentenmodelle. DCOM, Javabeans, Enterprise Java Beans, CORBA“, S. 293
Addison-Wesley, 2000
ISBN 382731724X

[104] Kriens, Peter: JSR 277 Review
http://www.osgi.org/blog/2006/10/jsr-277-review.html

[105] Wütherich, Gerd; Hartmann, Nils ; Kolb, Bernd; Lübken, Matthias: Die OSGi Service Platform - Eine Einführung mit Eclipse Equinox
dpunkt Verlag; 1. Auflage, 18.04.2008,
ISBN: 978-3-89864-457-0

[106] Seeberger, Heiko: Erste Schritte mit OSGi; Dezember 2008
http://it-republik.de/jaxenter/artikel/Erste-Schritte-mit-OSGi-2077.html

[107] Howes, T., University of Michigan: A String Representation of LDAP Search Filters; Juni 1996
http://www.ietf.org/rfc/rfc1960.txt

[108] OSGi Service Compendium R4.1
http://www.osgi.org/Download/File?url=/download/r4v41/r4.cmpn.pdf
Kapitel 701

[109] OSGi Service Compendium R4.1
http://www.osgi.org/Download/File?url=/download/r4v41/r4.cmpn.pdf
Kapitel 112

[110] Costin Leau on Thu: Spring Dynamic Modules for OSGi(tm) Service Platforms, 02.04.2009
http://www.springsource.org/osgi

[111] Apache Felix iPOJO
http://felix.apache.org/site/apache-felix-ipojo.html

[112] Escoffier, Clement: iPOJO on Android; 21.10.2008
http://ipojo-dark-side.blogspot.com/2008/10/ipojo-on-android.html

[113] Neubert, Matthias: Testing Project about Google Android and Apache Felix Integration
http://code.google.com/p/felixembeddedonandroid/

[114] Service Requirement Handler - Field injection
http://felix.apache.org/site/service-requirement-handler.html
#ServiceRequirementHandler-Fieldinjection

[115] Service Requirement Handler Mehtod invocation
http://felix.apache.org/site/service-requirement-handler.html
#ServiceRequirementHandler-Methodinvocation

[116] Eclipse Equinox
http://www.eclipse.org/equinox/

[117] Apache Felix
http://felix.apache.org

[118] OSGi Framework implementations - open source Equinox and commercial - ProSyst
http://www.prosyst.com/products/osgi_framework.html

[119] Knopflerfish - Open Source OSGi
http://www.knopflerfish.org/

[120] Bartlett, Neil; Hargrave, BJ (IBM Lotus): Android and OSGi: Can they work together?
http://www.eclipsecon.org/2008/?page=sub/&id=380

[121] ProSyst: Next Evolution of OSGi for Android
http://www.prosyst.com/news/news/2009/evo_osgi_android.html

[122] Clement Escoffier: Allow Felix to work on Android 1.5 with no hack
https://cwiki.apache.org/jira/browse/FELIX-1156

[123] OSGi Service Platform Release 4 Version 4.2 - Early Draft 3
http://www.osgi.org/download/osgi-4.2-early-draft3.pdf

[124] Apache Felix: Using Distributed Services with iPOJO
http://felix.apache.org/site/apache-felix-ipojo-dosgi.html

[125] Hackbarth, Kai : „Das Jahr von OSGi“ - Gründung des deutschen User Forum; 10.09.2008
http://entwickler.de/zonen/portale/psecom,id,99,news,45103.html

[126] SOF - An OSGI-like modularization framework for C++
http://www.codeproject.com/KB/library/SOF_.aspx

[127] Grigorov, Ewgeni: Issue 2711:Dalvik VM checks methods, which are not part of the class
http://code.google.com/p/android/issues/detail?id=2711

[128] ProSyst: mBedded Server for Android v1.0.0 - Release Notes
http://dz.prosyst.com/pdoc/changes/
Release-Notes_ProSyst-mBS-OSGi-Android_v1.0.0.pdf

[129] „xDude - XML & Dynamic user interface design in Google Android“
Belegarbeit von Thomas Matzke,
TU-Dresden

[130] Kriens, Peter: Bnd - Bundle Tool
http://www.aqute.biz/Code/Bnd

[131] Kriens, Peter: OSGi Bundle Repository Indexer Open Sourced
http://www.osgi.org/blog/2007/07/osgi-bundle-repository-indexer-open.html

[132] OSGi Alliance 2005 and Richard S. Hall: RFC-0112 Bundle Repository
http://www2.osgi.org/download/rfc-0112_BundleRepository.pdf

[133] Apache Felix Releases OBR Repository XMl Datei
http://felix.apache.org/obr/releases.xml

[134] Apache Felix OSGi Bundle Repository (OBR)
http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html

[135] BIndex
http://www.osgi.org/download/bindex.jar

[136] BIndex SVN
http://www.osgi.org/svn/public/trunk/org.osgi.impl.bundle.bindex/

[137] OSGi Bundle Repository 1.4.1
http://apache.prosite.de/felix/org.apache.felix.bundlerepository-1.4.1.jar

[138] Google inc.: Signing Your Applications
http://developer.android.com/intl/de/guide/publishing/app-signing.html

[139] Googel Projects for Android: Getting a Maps API Key
http://code.google.com/intl/de-DE/android/add-ons/google-apis/
maps-overview.html#apikey

[140] Android Developers: zipalign
http://developer.android.com/intl/de/guide/developing/tools/zipalign.html

[141] Android Developers: Handler
http://developer.android.com/intl/de/reference/android/os/Handler.html

[142] Android Developers: Designing for Responsiveness
http://developer.android.com/intl/de/guide/practices/design/responsiveness.html

[143] Android Developers: Designing for Performance
http://developer.android.com/intl/de/guide/practices/design/performance.html

Anhang

Anhang A1

Webanwendung

Einführung

Eine Webanwendung ist eine Anwendung welche von einem Webserver gehostet wird. Die graphische Benutzeroberfäche wird vom Anwender mittels Webbrowser aufgerufen. Anwender-System und Webserver sind hierfür in der Regel über ein Netzwerk wie z.B. dem Internet verbunden. Eine Webanwendung stellt dem Anwender ihre Funktionalität in Form einer Webseite zur Verfügung.

Durch Interaktion wie zum Beispiel das Klicken auf Hyperlinks oder das Ausfüllen von Formularen werden Anfragen an den Webserver formuliert (HTTP-Request). Der Webserver antwortet hierauf mit einem HTTP-Response. Hierfür greift der Webserver meist mittels Common Gateway Interface (CGI) auf einen Backend-Server zurück, welcher die Buisnesslogik in Form eines Standalone-Programms enthält. Dieses wiederum stützt sich meist auf eine Datenbank. Man spricht hierbei von einer klassischen 3-Tier Architektur.

Alternativ können Webserver und Webanwendungen auch integriert sein, wie dies bei Java Servlets, Java Server Pages oder ASP.NET der Fall ist. Das Backendprogramm generiert HTML-Code und übergibt diesen an den Webserver zur Beantwortung der Anfrage. Diese Anwort wird entsprechend im Webbrowser des Anwenders als Webseite dargestellt.

Aus Nutzersicht ähnlich zu den Webanwendungen sind die Rich Client Anwendungen. Hierbei findet eine Verlagerung der Anwendungslogik auf den Client statt. Der Programmcode wird beim Zugriff des Browsers auf die Webseite heruntergeladen und ausgeführt. Der klassische Vertreter hierführ ist JavaScript. Es handelt sich dabei um eine Java-ähnliche Scriptsprache die im Browser von einer JavaScript-Engine interpretiert wird. Dies ermöglicht die Erstellung ansprechender und funktionaler Benutzeroberflächen.

Desweiteren existiert eine Technologie namens AJAX [1]. Diese setzt auf verschiedene Webtechnologien auf, unter anderem auch eine Client-seitig ausgeführte AJAX-Engine. Letztere ist eine vermittelnde Ebene zwischen

Benutzeroberfläche (Webseite) und Webserver. Diese Technologie ermöglicht, dass nur tatschlich benötigte Daten vom Webserver angefordert werden, und nicht die gesamte Seite neu geladen werden muss.

Der Einsatz von JavaScript ist trotz einiger Nachteile bezüglich Sicherheit und Performance sehr weit verbreitet. Der Rechenaufwand für die Verwendung JavaScript-basierter Webanwendungen ist zumindest auf rechenschwachen mobilen Endgeräten nach wie vor ein Thema. Allerdings zeichnet sich, wie auch in den vergangenen Jahren im Desktop-Bereich erlebbar, ein Trend zur Optimierung und Performancesteigerung bei JavaScript-Engines für Browser auf mobilen Endgeräten ab. Vor allem auf den Plattformen Apple iPhone und Google Android wurden große Erfolge erzielt [2].

Trotz allen Performance-Optimierungen kann sich eine Webanwendung zur Zeit noch nicht mit nativ auf dem mobilen Endgerät ausgeführten Programmen messen. Ebenso bleiben oft Look-and-feel, Reaktionsfreudigkeit und gerätespezifische Interaktionsmöglichkeiten auf der Strecke. Darüber hinaus fehlen Webanwendungen oft noch die Möglichkeiten sich auf die Gegebenheiten des mobilen Endgeräts (Leistung, Bildschrimgröße, Interaktionsmöglichkeiten) einzustellen.

Während native Anwendung vollen Zugriff auf die System-APIs des Endgerätes haben, laufen Webanwendungen und Rich Client Applications „in ihrer eigenen Welt“, in der ihnen aus Sicherheits- und Kompatibilitätsgründen der Zugriff auf systemnahe Ressourcen verwährt bleibt.

Der große Nachteil von Webanwendungen, die absolute Notwendigkeit einer dauerhaften Netzverbindungen wird im mobilen Bereich mit seinen zum Teil stark schwankenden Netzqualität erheblich verschärft. Die Palette reicht von drastisch verlängerten Wartezeiten auf Interaktionsreaktionen bis hin zu totalen Anwendungsausfällen bei Verbindungsabbrüchen. Die bereits angesprochene Technologie AJAX wirkt dem in einem gewissen Maße entgegen. Sie verbessert auch bei langsamer oder schlechter Netzwerkanbindung die Reaktionsfreudigkeit der Anwendung, da alle potentiell länger brauchenden Anfragen asynchron abgesetzt werden. Auch die Menge der zu transportierenden Daten wird dadurch aufs Nötigste reduziert.

Die genannten Kriterien können auf das Stichwort Nutzungskomfort reduziert werden, das vor allem im Bereich der mobilen Endgeräte extrem wichtig ist.[3], [4], [5] Die Erwartungshaltung und die Geduld der Endanwender sind nicht direkt mit der Situation bei Desktop-Systemen vergleichbar.

Während im Desktop-Segment dank Mutli-tasking und Mutil-Windowing trotz eines rechen- oder zeitintensiven Prozesses der Eindruck der Reaktionsfreudigkeit und Lebendigkeit erhalten bleibt, laufen mobile Anwendungen quasi im Vollbild-Modus. Reagiert eine mobile Anwendung auch nur wenige Sekunden nicht, entsteht

beim Nutzer schnell der Eindruck das gesamte Gerät wäre lahmgelegt. Einen im Desktop-Segment verbreiteten Verständnis-Bonus der Art „so ist eben Windows“ gibt es im Bereich der mobilen Anwendungen zur Zeit noch nicht.

Dieses eher psychologische Gegenargument gegen den Einsatz von Webanwendungen auf mobilen Endgeräten ist vor allem bezüglich des in der Einführung erwähnten Szenarios relevant. Ein schnell entstandener, schlechter Eindruck bezüglich der Nutzbarkeit eines Dienstes wird schnell auf den Dienste-Anbieter übertragen. Daher ist bei der Auswahl einer geeingeten Technologie vermehrt auf das Kriterium des Nutzungskomforts Wert zu legen.

Fazit

Obwohl Webanwendungen auch nach der New-Economy Krise starke Verbreitung gefunden haben und durch das Stichwort „Web 2.0“ massiven Auftrieb erhalten haben, sind die angeführten Nachteile bei der Nutzung dieser Dienste auf mobilen Endgeräten nicht zu vernachlässigen.

Vor allem die Unterschiede in den Bereichen Performance und Nutzungskomfort sprechen dafür, dass zumindestens zum aktuellen Zeitpunkt der Einsatz reiner Webanwendungen auf mobilen Endgeräten zwar nicht unbedingt vermieden werden sollte, aber zumindest der Einsatz nativer Anwendungen zur Nutzung der gleichen Dienste ernsthaft in Betracht gezogen werden sollte. Es ist zu erwarten, dass der damit einhergehende Mehraufwand an Entwicklungszeit und Kosten durch eine erhöhte Nutzerakzeptanz belohnt wird.

Außerdem bietet der Trend zur Service-Orientierten Architektur (SOA) im Bereich der Webanwendungen einiges an Einsparpotential. Die Trennung in graphische Benutzeroberfläche (Webfrontend) und das funktionale Dienste anbietende Backend (Webservice) führt zur Entkopplung von Inhalt und Präsentation. Diese Entwicklung hat zur Folge das der vollständige Funktionsumfang einer Webanwendung als Webservice API zur Verfügung steht. Da ein Webservice von verschiedensten Anwendungen genutzt werden kann sind neben der Weboberfläche auch native Anwendungen denkbar, die lokal auf dem mobilen Endgerät ausgeführt werden, aber ihren Funktionsumfang aus der Nutzung eines Webservices beziehen. Zu einem Webservice Backend können also beliebige Frontends als Benutzeroberflächen entwickelt werden, wobei der Entwicklungsaufwand für das Backend durch Wiederverwendbarkeit nur einmal anfällt.

Die jeweiligen nativen Frontend-Anwendungen können den vollen Umfang der plattformspezifischen Möglichkeiten nutzen und somit dem Nutzer sehr komfortabel den Funktionsumfang des Backend - Webservices zur Verfügung stellen. Den Flaschenhälsen JavaScript-Engine, Browser-Perfomance und Interaktionsmöglichkeiten sowie dem verringerten Nutzungskomfort wird dadurch erfolgreich

entgegengewirkt. Aus diesem Grund ist es sinnvoll die Entwicklung eines Systems anzustreben, welches einen natives Client-Programm mit einschließt.

Das dies auch von den Anbietern mobiler Plattformen so gesehen wird zeigen Apple (iPhone) und Google (Android). Zwar verfügen beide Plattformen über performante Webbrowser, dennoch wird die Nutzung des Web-basierten Geoinformationsdienstes „Google Maps“ [6], welcher interaktives, digitales Kartenmaterial der ganzen Erde für den Zugriff per Webbrowser zur Verfügung stellt, mittels separater Anwendungen optimiert ermöglicht. Navigation in und Interaktion mit der Karte sind im Vergleich zur Browservariante komfortabler, der Bildaufbau ist schneller und flüssiger und die Nutzung der integrierter GPS-Sensoren des mobilen Endgeräts wird ermöglicht. An diesem Beispiel werden die erwarteten Vorzüge einer nativen Anwendung deutlich.

Diese Vorzüge stehen der Allgemeinheit und Plattform-Unabhängigkeit von Webanwendungen gegenüber. Im Rahmen der Entwicklung nativer Anwendungen wird hierauf nahezu vollständig verzichtet. Wie jede Auswahlentscheidung ist es ein trade-off verschiedener Argumente, wobei die Fülle der genannten Vorzüge zum aktuellen Zeitpunkt die eindeutige Richtung für den Einsatz nativer Anwendungen zur Webservice-Nutzung aufzeigt.

Da auf Basis dieser Argumente die Entwicklung einer Webanwendung zur Nutzung auf mobilen Endgeräten vorerst nachteilig wäre, sind nun die Bedingungnen für die Entwicklung einer nativen Anwendung zu untersuchen. Hierzu wird im Kapitel 2.1 eine Auswahl mobiler Plattformen untersucht und miteinander verglichen.

Webservice

In diesem Kapitel soll der Begriff Webservice eingeführt werden, sein Konzept, Funktionsweise und Umsetzung aufgezeigt und auf die zwei großen Webservicetypen eingangen werden. Es soll hier kein Vergleich gemacht werden, welcher Typ der Bessere ist, da es ein Ziel dieser Arbeit ist, einen Ansatz zu finden wie man verschiedenste Webservices ungeachtet ihrere Beschaffenheit in mobile WebGIS-basierte Systeme integrieren kann. Ziel ist es, dass Konzept „Webservice“ zu erklären um es im weiteren Verlauf benutzen zu können.

Ein Webservice ist eine Programmierschnittstelle (API) die über ein Netzwerk wie dem Internet angesprochen wird. Er ist das funktionale Backend einer Service-orientierten Architektur-Umsetzung (SOA). Der Webservice-Konsument, beispielsweise eine Frontend-Anwendung, erreicht den Webservice über einen eindeutigen Uniform Resource Identifier (URI) [7]. Die Kommunikation findet in Form von XML Nachrichten statt, welche über ein Netzwerktransportprotokoll transportiert werden. In der einfachsten Konfiguration (siehe Abbildung A1.1) stellt

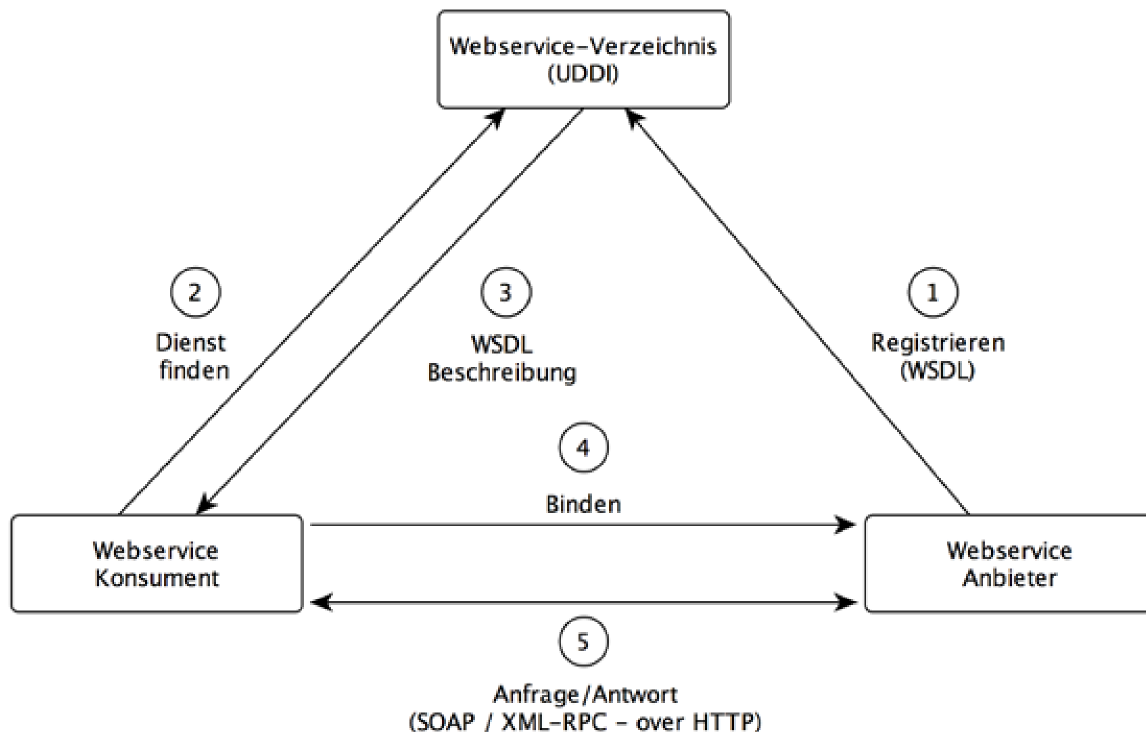


Abbildung A1.1: Webservice - Schematischer Aufbau

ein Client-Programm (Rolle: Konsument) Remote Precedure Call (RPC) - ähnliche Anfragen bezüglich angebotener Funktionen an den Server (Rolle: Anbieter) der den Webservice hostet. Dieser beantwortet die Anfrage mittels Ausführung der Funktion und der Rückgabe eines Ergebnisses in Form einer XML-Nachricht.

Zuvor muss der Client den Anbieter aber erst finden und sich an ihn binden. Hierzu kann ein Verzeichnisdienst - Universal Description, Discovery and Integration (UDDI) - verwendet werden. In diesem Verzeichnisdienst werden XML-ähnliche Beschreibungen in der Sprache Webservice Description Language (WSDL) der bei ihm registrierten Webservices abgelegt. Diese Beschreibungen enthalten die Informationen die der Konsument braucht um sich mit dem Anbieter zu verbinden und dessen Webservice zu nutzen, wie zum Beispiel Informationen über die angebotenen Dienste, Bindungsinformationen, Methodennamen und ihre Parameter.

Zur Kommunikation zwischen Konsument und Anbieter werden die Protokolle SOAP oder dessen leichtgewichtige „Variante“ XML-RPC verwendet, welche das Nachrichtenaustauschformat spezifizieren. Zum Transport wird meist das HTTP Protokoll eingesetzt.

Ein Webservice ist also eine Schnittstelle für Anwendungen, nicht für den Benutzer. Deshalb ist eine Webanwendung von einem Webservice zu unterscheiden. Eine

Webanwendung kann allerdings einen Webservice nutzen, um seine Funktionalität in ihrer Benutzeroberfläche (z.B. einer Webseite) darzustellen.

Der beschriebene Webservice entspricht dem „klassischen“ Webservice. Eine javabasierte Implementierung ist beispielsweise JAX-WS [8],[9] , [10] , seit Version 5 Teil des J2EE (Java Enterprise Editon). Sie ist ein gutes Beispiel dafür, wie eine Webservice-basierte SOA-Architektur in einer objektorientierten Spache umgesetzt werden kann.

Bei dieser Implementierung kommt das Proxy-Object-Pattern zum Einsatz. Der Webservice wird gegen ein selbst spezifiziertes Java-Interface implementiert. Der Client kennt dieses Interface, und generiert hieraus ein Proxy-Object. Dieses enthält die Funktionalität nicht selbst, sondern leitet die Aufrufe meist mittels XML-basierteter SOAP-Nachrichten via HTTP an den Webservice weiter. Für den Client wirkt es so, als sei die Implementierung lokal vorhanden. Die verteilte Verarbeitung bleibt für den Client transparent. JAXB (Java Architecture for XML Binding) übernimmt die Konvertierung der XML-Datentypen aus der Antwort in Java-Datentypen.

Eine etwas neuere Alternative zu den klassischen Webservices sind die REST oder RESTful Webservices [11] . REST steht hierbei für REpresentational State Transfer. Eine Java-API Spezifikation ist JAX-RS [12]. Suns freie Referenzimplementierung Jersey [13] ist eine von mehreren Implementierungen [14]. Der Reichtum an verfügbaren Implementierungen steigert die Einsatzfreundlichkeit von RESTful Webservices und trägt zu deren Verbreitung bei.

REST-Webservices unterscheiden sich von klassischen SOAP/WSDL - Webservices vor allem dadurch, dass das Interface auf eine kleine Menge standardisierter Operationen reduziert wird, welche die Ressourcen unterstützen müssen und die für den Abruf, die Manipulation oder das Löschen von Ressourcen verwendet werden. Diese Operationen entsprechen den HTTP-Operationen:

- PUT (Ressource anlegen/aktualisieren)
- GET(Daten von der Ressource lesen)
- POST(Anlegen einer Unter-Ressource)
- DELETE (löschen von Daten)
- HEAD
- OPTIONS

PUT, GET und Delete sind idempotent: ist man sich nicht sicher ob die Operation erfolgreich war, kann sie gefahrlos wiederholt werden. Es gibt keine spezifischen Methoden mit spezifischen Parametern, sondern die Webservicefunktionalität muss auf die genannten Operationen auf den jeweiligen Ressourcen abgebildet werden.

Zentral ist der Begriff der Ressource. Bei REST-orientierten Architekturen sind Anwendungen als Netz von Ressourcen zu sehen, welche mittels URI eindeutig

identifiziert und adressiert werden. Ressourcen sind zunächst alle Entitäten des Domänenmodells, wodurch jede abgebildete Entität in einer Anwendungen in Form eines Links bzw. einer Adresse potentiellen anderen Anwendungen zur Verfügung steht.(Adressierbarkeit) Dies erleichtert die Wiederverwendung enorm. Die URI ermöglichen den Zugriff auf die Ressourcen via HTTP.

Eine Ressource kann mehrere Repräsentationen haben, die in verschiedenen (Standard-) Formaten vorliegen können. Ein Browser kann z.B. eine HTML-Repräsentation erhalten, eine Anwendung oder eine XML- bzw. JSON-Repräsentation der Ressource. Darüber hinaus können Ressourcen auch Links zu anderen Resourcen enthalten, die wiederum programmatisch verwendet werden können. Dies ermöglicht ein Hypermedia/Hyperlink - Prinzip auf Webservice-Ebene.(Verbindungshaftigkeit)

In der REST-Architektur muss der Server keine Informationen über den Zustand des Clients (zB Session-Informationen) halten.(Zustandslosigkeit) Der Client sendet in jeder Anfrage alle für die Beantwortung notwendigen Zustandsinformationen eindeutig und völlig verständlich an den Server. Der Client wiederum erfährt über den Zustand des Webservice (des Servers) nur soviel wie er über die für ihn gedachte Repräsentation erfahren muss. Da keine Session-Verwaltung auf Serverseite stattfinden muss, skalieren REST-Architekturen durch einfaches Server-Load-Balancing recht gut. [15]
Vor allem Bezüglich der Wiederverwendbarkeit und Kompatibilität zu den verschiedensten Web-Technologien sind REST Webservices den klassischen Webservices überlegen. Roy Fielding, der REST Webservices im Rahmen seiner Disertation [16] entwickelte, sagte hierzu „SOAP was known to be a bad idea in 1999, but in spite of our comments to this effect, the industry insisted on proving that for themselves“ (Roy T. Fielding on the rest-discuss, mailing list, Sunday, 12 Nov 2006 14:16:36)

Weiterführende Informationen zu REST Webservices finden sich in „Web Services mit REST“ von Leonard Richardson und Sam Ruby aus dem O‘Reilly Verlag.

Anhang A2

Untersuchung der Implementierung der Android-Google Maps API

Das Klassendiagramm in Abbildung A2.1 stellt die Zusammenhänge grafisch dar.

Die zentrale und grundlegende GUI-Komponente von Android ist die Activity. Sie entspricht einem Bildschirm einer Android-Anwendung. Um eine neue Activity zu entwickeln, muss die eigene Klasse von der Klasse Activity erben. Der Einstiegspunkt in die Anwendung ist ebenfalls eine Activity. Die Rolle dieser Activity wird mittels Intent in der Konfigurationsdatei AndroidManifest.xml festgelegt. Eine Anwendung besteht meist aus mehreren Activitys, die den Hauptbildschirm, Menüs und Dialoge darstellen und die untereinander entsprechend mittels Intents verknüpft sind. Activitys haben einen Lebenszyklus, welcher durch die in einer von Activity erbenden Klasse zu implentierenden Methoden onCreate(), onStart() oder onStop() gesteuert wird. Mittels dieser kann der Entwickler Einfluss auf das Verhalten der Activity nehmen.

Das Google API Addon liefert die Klasse MapActivity welche von Activity erbt. Sie bildet die Grundlage eines Bildschirms, der das Kartenmaterial von Google Maps in der Android-Anwendung darstellt. Möchte man eine Activity erstellen, welche die Google Maps Karte darstellt, so muss diese selbst anstatt von Activity von der Klasse MapActivity aus dem Package com.google.android.maps erben. Generell stammen die meisten der folgenden Klassen aus diesem Package.

Die grafische Oberfläche einer Activity wird aus Views zusammengesetzt. Die Klasse View ist die Oberklasse von GUI-Elementen wie Buttons oder Texteingabefelder. Sie definieren für einen rechteckigen Bereich das baumstrukturierte Layout und den Inhalt. Die zentrale View in einer MapActivity ist die Klasse MapView. In der MapView wird die Karte dargestellt. Sie unterstützt dank der Klasse MapController von Haus aus das Scrollen in der Karte (Pan) und durch setBuiltInZoomControls(true) kann die Zoomsteuerung aktiviert werden. (F_3.3) Diese Funktionen können auch programmatisch verwendet werden, um beispielsweise einen Kartenausschnitt als Standard beim Start des Programms festzulegen.

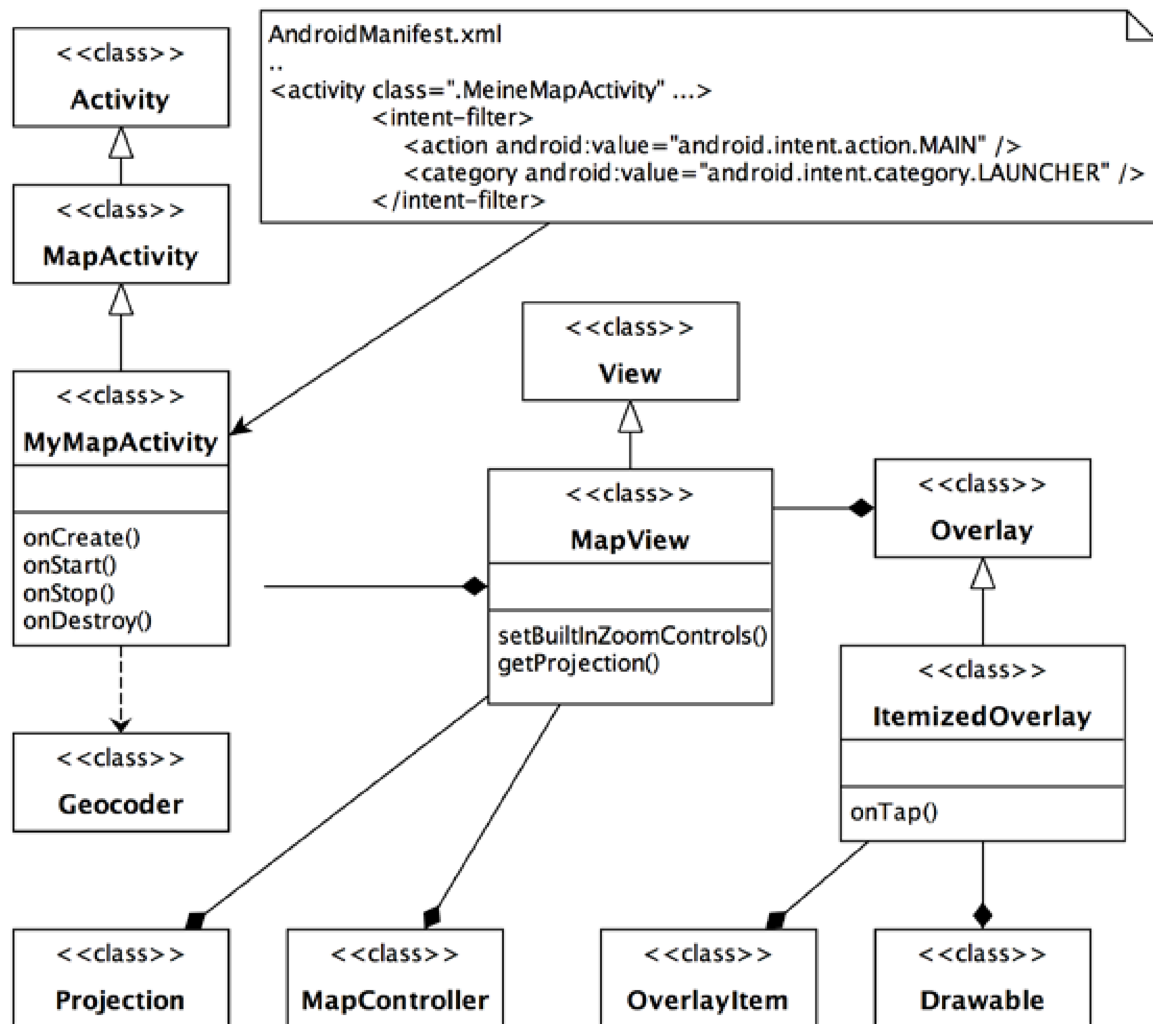


Abbildung A2.1 Klassendiagramm Google Maps Bibliothek für Android

Mittels Overlays kann man kartenbezogene Objekte (MapItems) an einem Punkt in der Karte darstellen.(F_3.3) Overlays sind im Prinzip Listen von MapItems. Die Klasse ItemizedOverlay erbt von Overlay und ist eine bequeme Implementierung zur einfachen Darstellung von Objekten in der Karte. Einem ItemizedOverlay kann ein Marker in Form einer Grafik (Klasse Drawable), beispielsweise eine kleine Bilddatei, übergeben werden, welche dann dazu benutzt wird, jedes darzustellende Objekt dieses Overlays jeweils an seiner Position zu repräsentieren. Die Position des Markers wird durch die Koordinaten des Objekts bestimmt und jeweils in einem Objekt der Klasse GeoPoint gehalten.

Mit Hilfe der Methode getProjection der Klasse MapView kann ein Objekt erzeugt werden, welches das Interface Projection implementiert. Dieses Objekt bietet je nach aktuellem Kartenzustand (z.B. Zoomlevel und dargestellter Kartenausschnitt) die passenden Umrechnungsmethoden zur Umformung zwischen Geokoordinaten

und Pixel-Koordinaten auf dem Bildschirm.

Es ist auch möglich, mit den Elementen eines Overlays zu interagieren. Hierzu ist in einer eigenen Klasse, die beispielsweise von ItemizedOverlay erbt, die Methode onTap(GeoPoint , MapView) zu implementieren. Hierbei wird die MapView, welche das tap-Ereignis entgegengenommen und an ihre Overlays weitergeleitet hat, mit übergeben. Dies kann dafür genutzt werden, auf den Context der MapView zuzugreifen, um z.B. einen Toast (kurz angezeigte Nachricht an den Nutzer) über der MapView anzuzeigen.

Der übergebene GeoPoint repräsentiert die Geokoordinaten des Punktes, der in der Karte getapt (geklickt) wurde. In der onTap() Methode des Overlays muss dann geprüft werden, ob der übergebene GeoPoint im Bereich eines OverlayItems dieses Overlays liegt. Ist dies der Fall, kann darauf beliebig reagiert werden, wie eben mit der Anzeige eines Toast.

Durch die Klasse Geocoder im Package „android.location.Geocoder“ kann auf die Geocoding und Reverse-Geocoding Funktionalität der Google Maps API zugegriffen werden. Die Suche erfolgt beim Geocoding mittels Suchtext-String. Der Ergebnisse können in ihrer Anzahl beschnitten werden und auf ein Koordinatenrechteck eingeschränkt werden.

Anhang A3 - Listings

Listing 4.8: iPOJO Service Verwendung mit iPOJO-Annotations

```
package org.mnsoft.mybundle;
import org.mnsoft.anotherbundle.AnotherService;
...
@Component
@Provides
public class MyServiceComponentImpl implements MyService {
 @Requires
 private AnotherService m_service;
 @Validate
 public void start() {
  ...
  if (m_service != null) m_service.aMethod();
  ...
 }
 @Invalidate
 public void stop() { ... }
 @Bind
 public void bindAnotherService(AnotherService s) {
  m_service = s;
  ...
 }
 @Unbind
 public void unbindAnotherService(AnotherService s) {
  m_service = null;
  ...
 }
 public void myMethod() {
 ...
 }
}
```

Listing 6.3: Bnd-Datei erweitert für BIndex repository.xml-Generierung

```
Import-Package: de.mnsoft.mapserviceviewer.globaltypes;version=1.0.0,
                de.mnsoft.felixhostapp.viewproviderservice;version=1.0.0
DynamicImport-Package:  android.*, com.google.*
Export-Package:      de.mnsoft.mapprovider;version=1.0.0
Bundle-Version:      1.0.0
Bundle-Vendor:       MNSoft (Matthias Neubert)
Bundle-Description: Interface-Bundle welches einen MapProvider beschreibt.
Bundle-Copyright:   (c) Copyright Matthias Neubert
Bundle-License:   http://www.apache.org/licenses/LICENSE-2.0 ; \ description=“This material is
licensed under the Apache Software License, Version 2.0“; link=“http://www.apache.org/licenses/
LICENSE-2.0“
```

Listing 6.4: Beispiel für die repository.xml - Datei

```
<resource id=‘de.mnsoft.mapprovider/1.0.0‘ presentationname=‘de.mnsoft.mapprovider‘
symbolicname=‘de.mnsoft.mapprovider‘ uri=‘bundles/de.mnsoft.mapprovider.jar‘ version=‘1.0.0‘>
 <description>
   Interface-Bundle welches einen MapProvider beschreibt.
 </description>
 <size>
   2124
 </size>
 <license>
   http://www.apache.org/licenses/LICENSE-2.0 ; description=“This
   material is licensed under the Apache Software License, Version
   2.0“; link=“http://www.apache.org/licenses/LICENSE-2.0“
 </license>
 <copyright>
   (c) Copyright Matthias Neubert
 </copyright>
 <capability name=‘bundle‘>
   <p n=‘manifestversion‘ v=‘2‘/>
   <p n=‘presentationname‘ v=‘de.mnsoft.mapprovider‘/>
   <p n=‘symbolicname‘ v=‘de.mnsoft.mapprovider‘/>
   <p n=‘version‘ t=‘version‘ v=‘1.0.0‘/>
 </capability>
 <capability name=‘package‘>
   <p n=‘package‘ v=‘de.mnsoft.mapprovider‘/>
   <p n=‘uses‘ v=‘de.mnsoft.felixhostapp.viewproviderservice‘/>
   <p n=‘version‘ t=‘version‘ v=‘1.0.0‘/>
 </capability>
```

<require extend=‘false‘ filter=‘(&(package=de.mnsoft.felixhostapp.viewproviderservice)
(version>=1.0.0))‘ multiple=‘false‘ name=‘package‘ optional=‘false‘>
Import package de.mnsoft.felixhostapp.viewproviderservice ;version=1.0.0
</require>
<require extend=‘false‘ filter=‘(&(package=de.mnsoft.mapserviceviewer.globaltypes)
(version>=1.0.0))‘ multiple=‘false‘ name=‘package‘ optional=‘false‘>
Import package de.mnsoft.mapserviceviewer.globaltypes ;version=1.0.0
</require>
</resource>

Anhang A4
Klassendiagramme

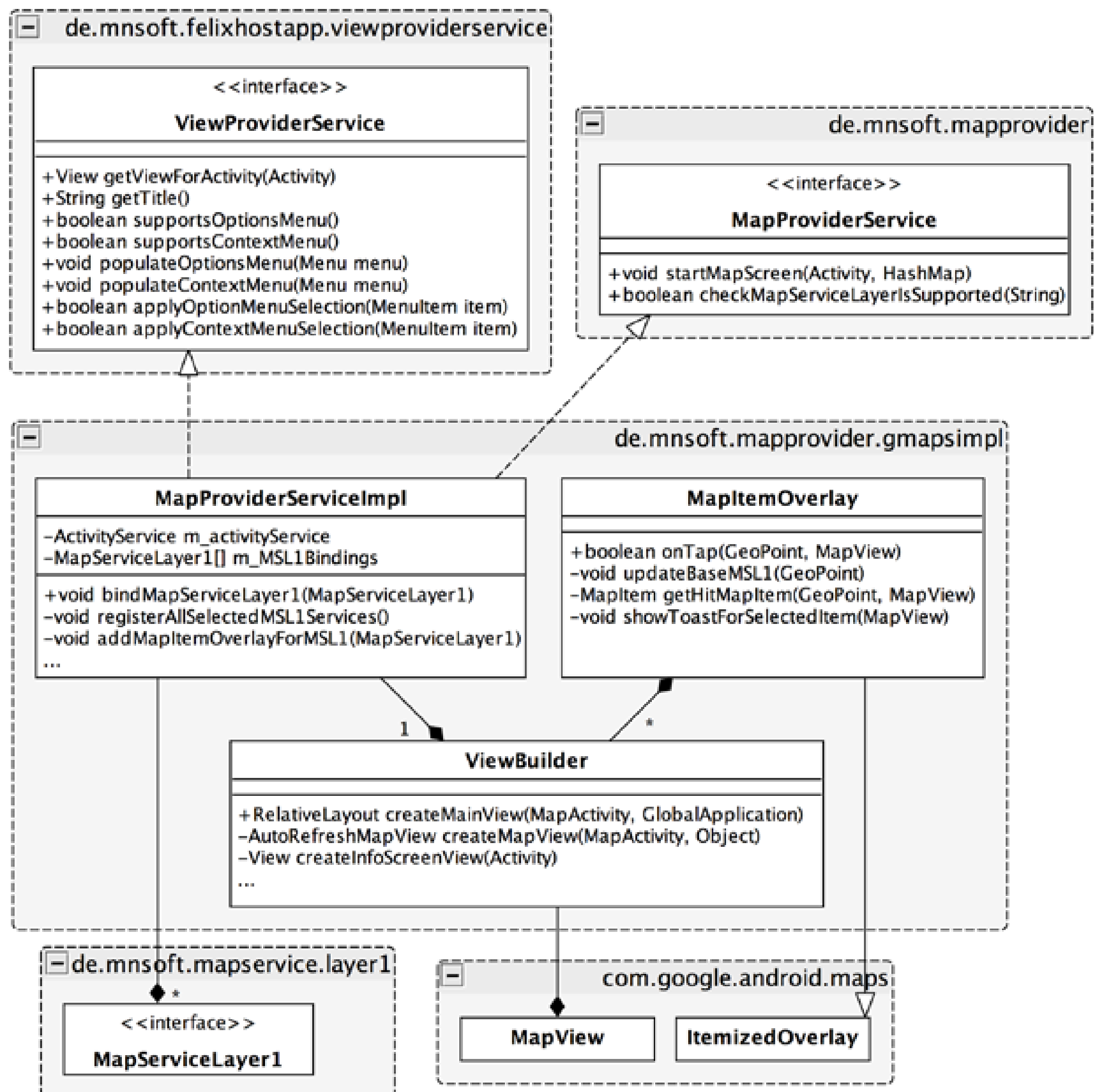


Abbildung 6.4: Klassendiagramm MapProvider-Implementierung für Google Maps