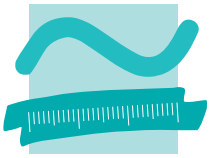

TECHNISCHE FACHHOCHSCHULE BERLIN
University of Applied Sciences

DIPLOMARBEIT

Evaluierung des Einsatzes von Scala bei der Entwicklung für die Android-Plattform

Verfasser:
Meiko RACHIMOW

Betreuer:
Prof. Christoph KNABE

11. Februar 2009

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (FH)

an der

Technischen Fachhochschule Berlin

Fachbereich Informatik und Medien

im

Studiengang Medieninformatik

Studienschwerpunkt Softwaretechnik

Thema:	**Evaluierung des Einsatzes von Scala bei der Entwicklung für die Android-Plattform**
Diplomand:	Meiko Rachimow Mat-Nr. 726338 Am Zernsee 15, 14542 Werder
Betreuer:	Prof. Christoph Knabe
Gutachter:	Prof. Dr. Rüdiger Weis
vorgelegt am:	

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Werder, den 11. Februar 2009 Meiko Rachimow

Inhaltsverzeichnis

1 Einleitung

In der vorliegenden Arbeit wird die Programmiersprache Scala auf ihre Eignung für die Entwicklung von Android-Anwendungen untersucht.

Die Zahl von Programmiersprachen für die Java-Plattform nimmt ständig zu. Die Vorteile von Java sind die große Popularität, die Existenz hochwertiger Werkzeuge und Bibliotheken und die virtuellen Maschinen, die hardwareunabhängige Anwendungen ermöglichen. Die neuen Sprachen nutzen diese Vorteile und setzen alternative und moderne Konzepte um. Dazu gehört auch Scala, in der unterschiedliche Programmierparadigmen, wie die objektorientierte und die funktionale Programmierung, vereint sind. Im Gegensatz zu den meisten anderen Sprachen, die für die Java-VM entwickelt wurden, ist Scala statisch typisiert und ermöglicht trotzdem ein ausdrucksstarkes und elegantes Programmieren.

Android ist ein neues auf Linux basierendes Betriebssystem für mobile Geräte, dessen Entwickler die Popularität von Java nutzen, um die Verbreitung ihres Systems zu erreichen. In die Android-Plattform ist eine Laufzeitumgebung integriert, die mit Java entwickelte Anwendungen ausführen kann. Wegen der Kompatibilität von Scala und Java ist auch der Einsatz von Scala bei der Entwicklung von Android-Anwendungen möglich. Mit dieser Arbeit wird anhand einer größeren Anwendung eingehend untersucht, ob Scala produktiv für die Android-Entwicklung geeignet ist. Um eine abschließende Bewertung zu ermöglichen, wurden folgende Ziele definiert:

- Untersuchung der Möglichkeiten zur Integration von Scala in das Android-System
- Integration von Scala in den Android-Entwicklungsprozess
- Entwurf einer umfangreichen Beispielanwendung
- Umsetzung der Beispielanwendung unter Ausnutzung der Vorteile von Scala
- Evaluierung verschiedener Testwerkzeuge

1.1 Aufbau der Arbeit

Im zweiten Kapitel werden die sprachlichen Mittel von Scala vorgestellt. Ein besonderer Schwerpunkt liegt auf den Bereichen, die für die entwickelte Beispielanwendung eine bedeutende Rolle spielen und Scala von Java abheben.

Im dritten Kapitel wird auf die Android-Plattform eingegangen. Dabei werden die Architektur des Betriebssystems, die Laufzeitumgebung und die wichtigsten Komponenten und Dienste des Android-Framework erläutert.

In vierten Kapitel wird gezeigt, wie die Entwicklungsprozesse von Scala und Java verbunden werden können. Ergeben sich mehrere Möglichkeiten werden diese untersucht und bewertet. Es wird auf Probleme bei der Nutzung von Scala im Android-System eingegangen, sowie die Performance untersucht.

Im fünften Kapitel wird das Konzept der entwickelten Anwendung TribOSMap vorgestellt.

In Kapitel sechs wird auf die Erfahrungen eingegangen, die beim Design und der Implementierung der Beispielanwendung gemacht wurden. Dabei werden Probleme benannt und Lösungen diskutiert, die mit Hilfe der sprachlichen Mittel von Scala umgesetzt wurden.

Im siebten Kapitel werden Möglichkeiten zum Test von Scala-Anwendungen in und außerhalb der Android-Plattform anhand verschiedener Werkzeuge aufgezeigt.

Im achten Kapitel wird auf Erfahrungen eingegangen, die beim Einsatz von Scala und Android in der IDE Eclipse gemacht wurden.

Ein Fazit beendet die Arbeit.

1.2 Wichtige Anmerkungen

Es stellte sich heraus, dass die Mittel von UML 2.1 für die Darstellung von Scala-Programmen nicht ausreichen. Deshalb wurden Erweiterungen vorgenommen, die im Anhang (Abschnitt C auf Seite 145) dargelegt sind.

In den Programmauszügen sind weggelassene Programmteile mit der Zeichenkette /*...*/ gekennzeichnet. Wenn der Leser an den kompletten Programmen interessiert ist, dann sei an dieser Stelle auf die beiliegende CD verwiesen. Nach erfolgreicher Verteidigung der Diplomarbeit wird der Quellcode im Internet veröffentlicht [Siehe: Rac].

Für die Arbeit wurde die Version 2.7.2 von Scala und die Version 1.0 der Android-Plattform verwendet.

2 Die Programmiersprache Scala

In diesem Kapitel werden wichtige sprachliche Mittel von Scala vorgestellt und erläutert. Dabei wird besonders auf die syntaktischen Details und Konzepte eingegangen, die Scala von der Programmiersprache Java abheben und im weiteren Verlauf der Diplomarbeit eine besondere Rolle spielen.

2.1 Einleitung

Die Programmiersprache Scala wird seit 2001 am "Programming Methods Laboratory" der "École Polytechnique Fédérale de Lausanne" unter der Leitung von Prof. Martin Odersky entwickelt [Ode08a].

Scala kann in Java-Bytecode kompiliert werden, so dass die Programme in einer Java-Laufzeitumgebung ausführbar sind. Mit Scala entwickelte Komponenten lassen sich in Java Programme integrieren und umgekehrt ist es möglich Java-Bibliotheken in Scala zu nutzen. Einige Einschränkungen der Kompatibilität zwischen beiden Sprachen werden in Kapitel 4 besprochen.

Während funktionale Programmiersprachen vortrefflich für die Entwicklung von Algorithmen geeignet sind, liegen die Vorteile objektorientierter Sprachen in der Gestaltung komplexer Systeme, die aus der realen Problemdomäne abgeleitet werden. Scala ist sowohl eine funktionale als auch eine objektorientierte Programmiersprache. Die Kombination beider Paradigmen ermöglicht neuartige Lösungsansätze für die Probleme der Softwareentwicklung.

2.2 Sprachliche Mittel zur Funktionalen Programmierung

Mit funktionalen Sprachen wird versucht Eigenschaften einer mathematischen Ausdrucksweise wie Klarheit, Eleganz und Präzision in die Programmierung zu übernehmen [Pep03, Seite 2]. Während bei der imperativen Programmierung eine Folge von Arbeitsanweisungen für zustandsverändernde Rechenmaschinen programmiert wird, basieren funktionale Sprachen auf Funktionen im mathematischen Sinne. Funktionale Programme bestehen

allein aus der Definition, Applikation und Komposition von Funktionen. In ihnen verwendete Bezeichner repräsentieren innerhalb ihres Kontextes immer den selben Wert. Der Wert eines Ausdruckes, also auch das Resultat einer Funktion, ist unabhängig von Reihenfolge und Zeitpunkt der Auswertung. Diese Unabhängigkeit von Systemzuständen wird als Wirkungs- oder auch Seiteneffektfreiheit bezeichnet [Pep03, Seite 3]. Beispiele für rein funktionale Sprachen sind Lisp, Haskell oder Opal. Andere Sprachen wie zum Beispiel Python oder auch Scala beinhalten sowohl funktionale als auch imperative Konzepte.

Auf die Möglichkeiten zur funktionalen Programmierung mit Scala wird nun näher eingegangen.

2.2.1 Wichtige mathematische Begriffe

Im mathematischem Sinne sind Funktionen spezielle binäre Relationen. Eine binäre Relation R bildet Elemente einer Quellmenge X auf Elemente einer Zielmenge Y ab. Sie wird als eine Teilmenge des Kreuzproduktes von X und Y definiert, also sind deren Elemente Tupel mit je einem Element aus X und Y :

$$R \subseteq X \times Y = \{(x,y) : x \in X \wedge y \in Y\}$$

R ist eine Funktion F , wenn sie eindeutig (rechtseindeutig) ist:

$$\bigwedge_{x \in X} \bigwedge_{y_1, y_2 \in Y} (x,y_1) \in F \wedge (x,y_2) \in F \rightarrow y_1 = y_2$$

Der Definitionsbereich einer Funktion ist die Teilmenge von X , für die eine Abbildung in Y existiert. Der Wertebereich einer Funktion ist die Teilmenge von Y , deren Elemente eine Abbildung aus X sind.

Dies war nur eine Möglichkeit, die Begriffe Funktion, Werte- und Definitionsbereich mathematisch zu definieren [Neh03, Seite 175 ff.]. Eine andere mathematische Definition bezeichnet das Tripel der Mengen X, Y, F als Funktion. Die Relation F wird dann Funktionsgraph genannt [Pep03, Seite 15].

2.2.2 Definition und Applikation von Funktionen

Funktionen werden in Scala mit dem Schlüsselwort *def* definiert. Zudem wird die Quellmenge als n-Tupel, sowie die Zielmenge angegeben. Die eigentliche Funktion wird algorithmisch in den Funktionsrumpf implementiert. Im Folgenden die Definition einer Funktion *f* mit genau einem Parameter *x*:

```
def f(x : X): Y = /*...*/
```

Programmauszug 2.1: Funktionsdefinition

Die Quellmenge von f ist somit ein 1-Tupel. Das Element dieses Tupel ist die Menge X, wobei der Parameter x Element dieser Menge ist. Die Zielmenge der Funktion ist Y. Wobei die Mengen X und Y in Programmiersprachen üblicherweise als Typen bezeichnet werden. Es gibt in Scala noch einige andere Möglichkeiten, Funktionen zu definieren. Auf diese wird im Laufe des Kapitels eingegangen.

Bei der Applikation einer Funktion wird zu einem Element des Definitionsbereichs das abgebildete Element geliefert. Nun ein Beispiel, bei der auf die Funktion f eine Applikation ausgeführt wird. Der Wert des Resultates der Funktion wird einem Bezeichner y zugewiesen.

```
val y = f(x)
```

Programmauszug 2.2: Funktionsapplikation und Wertzuweisung

Diese Bezeichner werden auch Variablen genannt. Wobei sie den mathematischen Variablen entsprechen, da sie an einen Ausdruck gebunden sind. Das bedeutet auch, dass eine Variable, sobald sie einen Wert zugewiesen bekommen hat, nicht mit einem anderen Wert belegt werden kann. Somit kann kein Zustand dauerhaft gespeichert werden, und es ist sichergestellt, dass eine Funktion für eine Applikation mit den selben Parametern immer das selbe Resultat liefert. Damit setzt Scala das Prinzip der referenziellen Transparenz um, was einen mathematischen Beweis für die Korrektheit von Programmen erleichtert. [Vgl. Zhe06]

Ausdrücke, die an eine Variable (*val*) gebunden sind, werden bei der Initialisierung der Variablen ausgewertet. Im Unterschied dazu werden die den Funktionsbezeichnern (*def*) zugehörigen Ausdrücke bei einer Applikation ausgewertet.

2.2.3 Innere Funktionen und Closures

Mit Scala können auch Funktionen innerhalb anderer Funktionen definiert werden. Sie sind dann außerhalb der sie umgebenden Funktion nicht sichtbar. Innerhalb der inneren Funktion kann auf die lokalen Variablen der äußeren Funktion zugegriffen werden. Werden diese sogenannten gebundenen Variablen benutzt, nennt man die innere Funktion ein "Closure" [Pep06, Seite 276]. Das Resultat hängt somit nicht nur von ihren eigenen, sondern auch den Parametern der äußeren Funktionen ab. Zur Illustration ein Beispiel, bei der eine Funktion *bar* innerhalb einer Funktion *foo* deklariert wird.

```
def foo(some : Int) = {
   def bar(other : Int) = { some + other }
   bar(2)
}
foo(1) //liefert 3
```

Programmauszug 2.3: Closure

Die Funktion *bar* ist ein Closure, denn das Resultat von *bar* ist abhängig von der Variablen *some*. Die Variable *some* ist ein Parameter und damit eine lokale Variable der Funktion *foo* und wird hier in der inneren Funktion verwendet.

2.2.4 Totalisierung partieller Funktionen

Zu einer wichtigen Eigenschaft von Funktionen gehört die Totalität. Eine Funktion ist total, bzw. vollständig definiert, wenn die Quellmenge mit dem Definitionsbereich identisch ist. Ist eine Funktion nicht total, nennt man sie auch partielle Funktion. Ein Beispiel ist die Division, denn es gibt kein Element der Zielmenge, das für eine Division mit dem Element Null der Quellmenge definiert ist. Bei der Programmierung können partiell definierte Funktionen zu Ausnahmen führen. Wie in Java käme es zum Abbruch der Ausführung, wenn das Programm die Ausnahme nicht geeignet behandelt.

Man kann Funktionen totalisieren, indem man der Zielmenge ein sogenanntes "undefiniertes Element" $\perp$ hinzufügt. Es entstehen somit die neuen Mengen $Y_\perp$ und $R_\perp$. In der Relation $R_\perp$ werden die Elemente der Quellmenge X, die kein Element der ursprünglichen Zielmenge Y abbilden, mit dem undefinierten Element der neuen Zielmenge $Y_\perp$ verknüpft. [Pep03, Seite 16]

Die Umsetzung der Totalisierung kann in Scala mit der generischen Klasse *Option[T]* vorgenommen werden. Generische Klassen werden noch genauer erläutert. Hier ist zunächst nur wichtig, dass mit dieser Klasse eine beliebige Menge (Typ) um das "undefinierte Element" *None* erweitert werden kann. Die Menge *Option[Double]* ist somit im Prinzip die totalisierte Menge von *Double*.

```
def totalisierteDivision(x1 : Double, x2 : Double): Option[Double] = {
   if(x2 == 0) None
   else Some(x1/x2)
}
```

Programmauszug 2.4: Totalisieren einer Funktion

Die obige Funktion wird, bei einer versuchten Division durch Null, das "undefinierte Element" *None* zurückgegeben und keinen Fehler verursachen. Existiert eine Abbildung für die gegebenen Parameter, ist das Resultat der Applikation das Objekt *Some*. Das

Resultat der eigentlichen Division muss aus diesem Objekt extrahiert werden. Dabei muss zwangsläufig beachtet werden, dass auch das "undefinierte Element" *None* geliefert werden kann.

2.2.5 Anonyme Funktionen und Lambda-Schreibweise

Anonyme Funktionen haben zum Zeitpunkt ihrer Erzeugung keinen Namen. Im mathematischen Lambda-Kalkül werden Funktionen generell anonym definiert, deshalb wird diese Art der Funktionsdefinition auch als Lambda-Schreibweise bezeichnet. Auf das Lambda-Kalkül wird hier nicht näher eingegangen. Es sei lediglich erwähnt, dass es sich dabei um eine Turing-vollständige mathematische Sprache handelt, die vor allem zur Untersuchung von Funktionen verwendet wird [Bar00]. Zudem hat es die funktionalen Programmiersprachen maßgeblich beeinflusst.

Im Folgenden eine anonyme Funktion, die einen Parameter vom Typ *Int* besitzt und als Resultat ein Objekt vom Typ *Int* liefert. Der Typ des Resultates wird bei anonymen Funktionen nicht explizit spezifiziert; er ergibt sich durch die Typinferenz.

```
(x : Int) ⇒ x*x
```

Programmauszug 2.5: Anonyme Funktion

Anonyme Funktionen werden vor allem direkt als Parameter anderer Funktionen verwendet. Die Lambda-Schreibweise kann allerdings auch zur Definition nicht anonymer Funktionen verwendet werden, indem der Ausdruck an eine Variable gebunden wird.

2.2.6 Funktionen als Objekte

In Scala sind alle Funktionen Objekte im klassischen Sinne der objektorientierten Programmierung und besitzen einen Typ. Die Funktionstypen sind in der Scala-API durch Traits definiert. Traits sind den Schnittstellen in Java ähnlich. Mit ihrer Hilfe können Funktionsobjekte erzeugt werden, indem eine von den Funktions-Traits abgeleitete Klasse instantiiert wird. Dabei muss eine Methode *apply* überschrieben werden, die dann bei der Applikation implizit aufgerufen wird. Somit ist es möglich Funktionen wie folgt zu definieren:

```
val f = new Function1[Int, Int] {
   def apply(x : Int): Int = x * 2
}
f(2) //die Applikation liefert als Resultat 4
```

Programmauszug 2.6: Funktionsdefinition mit einem Trait

Für die Kennzeichnung der Funktionstypen existieren zwei Schreibweisen. Dazu ein Beispiel zweier gleichwertiger Funktionsdefinitionen *f1* und *f2*, die in ihrer Parameterliste ein *Int* und ein *Double* erwarten und als Resultat ein *String*-Objekt liefern:

```
val f1 : Function2[Int, Double, String] = /*...*/
val f2 : (Int, Double) ⇒ String = /*...*/
```

Programmauszug 2.7: Deklaration von Funktionen

2.2.7 Funktionen höherer Ordnung und Currying

In der funktionalen Programmierung ist es möglich Funktionen zu definieren, die Funktionen sowohl als Parameter besitzen, als auch als Resultate liefern können. Diese werden ebenso wie in der Mathematik als "Funktionen höherer Ordnung" bezeichnet.

Da Funktionen Objekte sind, können sie als Parameter verwendet werden. Dazu ein Beispiel für eine Funktionsdefinition, die als Parameter eine Funktion erwartet, die ihrerseits ein Objekt vom Typ *Int* erwartet und als Resultat ein Objekt vom Typ *String* liefert:

```
def f(g : Int ⇒ String) = /*...*/
```

Programmauszug 2.8: Funktionsparameter

Eine Funktion als Rückgabetyp zu definieren, ist dann natürlich ebenso möglich. So kann auf ein Resultat einer Applikation eine weitere Applikation durchgeführt werden.

```
def mul(x :Int): Int ⇒ Int = {
   y ⇒ { x*y }
}
mul(2)(3)//liefert 6
```

Programmauszug 2.9: Funktion als Resultat

Bei der Verwendung der eben definierten Funktion *mul* ist es natürlich möglich, nur eine Applikation durchzuführen. Das Resultat, also eine Funktion, kann als Wert an eine Variable gebunden werden.

```
val mul21 : Int ⇒ Int = mul(21)
```

Programmauszug 2.10: Partielle Anwendung einer Funktion

In diesem Zusammenhang wird auch der Begriff "partiell angewandte Funktion" verwendet. Der Wert der Variablen *mul21* ist eine "partielle Anwendung" der Funktion *mul*. Die Funktion *mul* definiert somit eine Menge von Funktionen. Generell kann jede Funktion mit mehreren Parametern in mehrere Funktionen mit einem Parameter transformiert werden. Dieses Verfahren wird "Currying" genannt [Pep06, Seite 16 ff.]. Die sogenannte

Currying-Schreibweise ermöglicht eine äquivalente Definition der Funktion *mul*:

```
def mul(x : Int)(y : Int) = x * y
```

Programmauszug 2.11: Currying Schreibweise

2.2.8 Arbeiten mit Sequenzen: map und reduce

Alle funktionale Sprachen verfügen über die Funktionen höherer Ordnung *map* und *reduce*, die zum Arbeiten mit Sequenzen verwendet werden.

Die Funktion *map* besitzt eine Funktion und eine Sequenz von Elementen als Parameter. Das Resultat von *map* ist eine Sequenz, die die Resultate der Applikationen der übergebenen Funktion auf die einzelnen Elemente beinhaltet. Es gibt in Scala mehrere Möglichkeiten Sequenzen zu definieren. Zum Beispiel mit Hilfe des Operators *to*, der für den Typ *Int* definiert ist. Dieser erzeugt eine Sequenz von Zahlen für einen angegebenen Bereich.

```
(0 to 10).map(x ⇒ {x*x}) //Sequenz aller Quadratzahlen von 0 bis 10
```

Programmauszug 2.12: Arbeiten mit Sequenzen: map

Dagegen kann die Funktion *reduce* verwendet werden, um alle Elemente einer Sequenz auf ein Resultat zu reduzieren. Dazu nutzt sie eine Funktion, die - auf das erste und zweite Element der Sequenz angewandt - ein Resultat liefert, das durch die Funktion zusammen mit dem dritten Element der Sequenz erneut ein Ergebnis liefert. So wird bis zum Ende der Sequenz weiter verfahren. Das Resultat der letzten Applikation ist das endgültige Resultat der Funktion *reduce*.

```
(0 to 10).reduceLeft((x,y) ⇒ {x+y}) //55
```

Programmauszug 2.13: Arbeiten mit Sequenzen: reduce

In Scala gibt es weitere Funktionen höherer Ordnung für das Arbeiten mit Sequenzen.

2.3 Objektorientiertes Programmieren

In diesem Abschnitt wird auf die Umsetzung des Konzeptes der Objektorientierung in Scala eingegangen. Es werden die sprachlichen Möglichkeiten und ihre Verwendung erläutert.

Objekte im Sinne der objektorientierten Programmierung bestehen aus Daten (Werte) und Operationen (Funktionen). Man könnte versuchen, ein Objekt als eine Menge von Funktionen anzusehen, wobei die Werte dann parameterlose Funktionen sind. Diese

Sichtweise ist aber nicht ausreichend, denn in objektorientierten Sprachen ist der Zustand der Objekte veränderbar. Dies lässt sich nicht mit dem bisher verwendeten Funktionsbegriff vereinbaren.

Einige objektorientierte Programmiersprachen sind klassenbasiert. Sie haben spezielle Programmiermittel zur Definition der sogenannten Klassen. Dann ergibt sich der Typ von Objekten aus der Zugehörigkeit zu diesen Klassen. Scala ist sowohl klassenorientiert, als auch "rein objektorientiert". In rein objektorientierten Sprachen gibt es keine Operationen und Daten, die nicht einem Objekt zugehörig sind. Java ist somit ein Beispiel für eine nicht "rein objektorientierte" Sprache. Denn es können zum Beispiel statische Operationen definiert werden, die global zur Verfügung stehen.

2.3.1 Definition von Klassen, Attributen und Methoden

Klassen definieren die Daten und Operationen von Objekten. Die Daten werden Attribute und die Operationen Methoden genannt. Objekte, die auch als Instanzen bezeichnet werden, können mit Hilfe der Klassen erzeugt (instantiiert) werden.

Die Definition von Klassen in Scala ist der Syntax anderer Sprachen wie zum Beispiel Java sehr ähnlich. Zur Erläuterung ein Beispiel für eine Klasse *Warenkorb*. Eine Instanz dieser Klasse soll mit Waren, also Objekten vom Typ *Ware*, gefüllt werden können. Dazu wird eine Methode *legeRein(ware: Ware)* definiert. Diese Methode verändert also den Zustand einer Instanz der Klasse *Warenkorb*. Eine weitere Methode *gesamtpreis* soll den Gesamtpreis aller Waren zurückgeben. Dabei fließt in die Berechnung ein Rabatt mit ein, der bei der Instantiierung als Konstruktorparameter übergeben wird.

```
class WarenKorb(rabattFaktor: Double) {
  private var inhalt = List[Ware]()
  def gesamtpreis() = /*...*/
  def legeRein(ware: Ware) = /*...*/
}
val korb = new WarenKorb(0.9)
korb.legeRein(ware) //verändert den Zustand
korb.gesamtpreis //liefert den Warenpreis
```

Programmauszug 2.14: Klassendefinition und Verwendung einer Instanz

Andere Objekte haben Zugriff auf die definierten Methoden der Klasse *WarenKorb*. Dies ist auch der Fall, wenn diese Objekte einer anderen Klasse angehören, da diese Methoden hier veröffentlicht wurden. In Scala sind alle Methoden und Attribute zunächst veröffentlicht. Um die Sichtbarkeit einzuschränken, wie bei dem Attribut *inhalt: List[Ware]* geschehen, wird zum Beispiel das Schlüsselwort *private* verwendet.

Konstruktorparameter, wie der *rabattFaktor* im Beispiel, sind zunächst immer unver-

öffentlicht und auch unveränderlich. Innerhalb von Methoden der Klasse können sie als gebundene Variablen verwendet werden. Jeder Parameter des Konstruktors kann als Attribut veröffentlicht werden. Dazu muss ein Parameter mit der Modifizierbarkeit gekennzeichnet werden. Das Schlüsselwort *var* kennzeichnet veränderliche Variablen, *val* hingegen unveränderliche. Auf die veröffentlichten Attribute kann nun über eine Instanz zugegriffen werden. Dazu das Beispiel einer Klasse *Kunde*, die ein veröffentlichtes und veränderliches Attribut *name* besitzt, und ein "privates" unveränderliches Attribut *password*.

```
class Kunde(var name : String, private val password : String)
val kunde = new Kunde("Lisa Simpson","******")
kunde.name = "Lisa Parkfield" //Aufruf des Setter
kunde.name //Getter liefert "Lisa Parkfield"
```

Programmauszug 2.15: Veröffentlichen von Attributen

Man beachte, dass bei einem externen Zugriff auf die Attribute nicht direkt mit der Wertreferenz gearbeitet wird. Stattdessen werden immer "Getter" und "Setter" verwendet. Diese werden implizit für alle veröffentlichten Attribute erzeugt. Dazu zunächst ein fehlerhaftes Beispiel, bei dem versucht wird, diese bereits implizit definierten Methoden zusätzlich explizit zu definieren.

```
class Kunde(var name : String, private val password : String) {
   //Versuche Setter zu definieren
   def name_= (neuerName : String) { }  //liefert Compiler-Fehler !
}
```

Programmauszug 2.16: Fehlerhafte explizite Setter-Definition

Der Name des "Setter" entspricht dem Namen des zugehörigen Attributes, an den ein Unterstrich und ein Gleichheitszeichen angehängt wird. Die Kompilierung des Beispiels schlägt mit dem Hinweis fehl, dass eine Methode mit dieser Signatur bereits existiert. Sie ist also bereits implizit erzeugt worden.

Bei der Instantiierung werden den Attributen, die im Klassenrumpf definiert wurden, die Werte der zugehörigen Ausdrücke zugewiesen. Wird dagegen das Attribut mit dem Schlüsselwort *lazy* versehen, findet die Wertzuweisung (bzw. Auswertung der Ausdrücke) erst statt, wenn das Attribut das erste Mal verwendet wird. Damit können eventuell aufwändige Operationen verhindert werden, wenn der Benutzer des Objektes ein Attribut nicht benötigt.

2.3.2 Einfachvererbung

Klassen können Funktionalität vererben. Die erbende Klasse wird Subklasse und die vererbende Basisklasse genannt. Vererbung kann verwendet werden, um Redundanzen zu vermeiden. Dazu wird die mehrfach benötigte Funktionalität in den Basisklassen implementiert. Eine Instanz der Subklasse hat auch den Typ der Basisklasse. Sie verfügt, wie Objekte der Basisklasse, über die dort definierten Methoden und Attribute. Durch das Überschreiben von Methoden und Attributen der Basisklasse wird eine Spezialisierung ermöglicht, wodurch sich Instanzen einer Subklasse anders verhalten können als Instanzen ihrer Basisklasse.

Im folgenden Beispiel wird eine Basisklasse *Person* definiert. Ihr Erbe ist die Klasse *Prominenter*. Beide implementieren eine Methode *printName*, wobei die Klasse *Prominenter* die Methode der Basisklasse explizit überschreibt (*override*).

```
class Person(name: String) {
  def printName() { println(name) }
}
class Prominenter(name: String) extends Person(name) {
  override def printName() { //überschreibende Methode
    print("Superwichtig: ")
    super.printName() //Aufruf der Methode der Basisklasse
  }
}
val promi: Person = new Prominenter("Bart Simpson")
promi.printName() //gibt "Superwichtig: Bart Simpson" aus
```

Programmauszug 2.17: Vererbung und Polymorphie

Die Variable *promi* ist vom Typ *Person* und referenziert eine Instanz der Klasse *Prominenter*. Obwohl die Variable den Typ der Basisklasse besitzt, führt der Aufruf der Methode *printName* dazu, dass die Methode der Subklasse verwendet wird. Dies nennt man Polymorphie ("Polymorphism") [Vgl. Car85]. Verschiedene Objekte können trotz ihrer Verwendung im Kontext des selben Typs ein anderes Verhalten besitzen.

Abstrakte Klassen können, im Gegensatz zu konkreten Klassen, nicht instantiiert werden. Sie dienen lediglich dem Aufbau von Klassenhierarchien, in denen sie Funktionalität zur Verfügung stellen und das Vorhandensein von Methoden und Attributen sicherstellen können. Im folgenden Beispiel sind in einer abstrakten Klasse *Person* eine abstrakte Methode *kommuniziere* und eine bereits implementierte Methode *esse* definiert.

```
abstract class Person(name : String) {
   def esse(nahrung : Nahrung) = { nahrung.loesche }
   def kommuniziere(person : Person)
}
```

Programmauszug 2.18: Abstrakte Klassen

Bei allen von dieser Klasse erbenden Subklassen ist das Vorhandensein dieser Methoden sichergestellt. Abstrakte Methoden und abstrakte Attribute müssen in konkreten Subklassen implementiert werden.

2.3.3 Scala und das Konzept der Traits

Dieser Abschnitt befasst sich zunächst mit der Mehrfachvererbung. Als Alternative zur Mehrfachvererbung und als Erweiterung zur Einfachvererbung wurden die Konzepte der Mixins und Traits entwickelt. Diese werden erläutert, bevor auf die Traits von Scala eingegangen wird.

Probleme und Möglichkeiten der Mehrfachvererbung

Mit der Mehrfachvererbung ist es möglich, dass eine Subklasse mehrere Basisklassen besitzt. Bei einem objektorientierten Design von Software ist dies oftmals eine naheliegende Möglichkeit, weitere Redundanzen des Programms zu vermeiden. Allerdings birgt die Mehrfachvererbung einige fundamentale Probleme. Zum einen gibt es technische Probleme bei der Implementierung. Zum anderen müssen Namenskonflikte, die durch das Erben von Methoden mit gleicher Signatur aus unterschiedlichen Klassen entstehen, manuell in der Subklasse aufgelöst werden. Dazu kommt das sogenannte Diamond-Problem, bei dem eine Klasse auf unterschiedlichen Vererbungslinien eine Basisklasse mehrfach beerbt [Vgl. Tru04].

Dazu folgendes Beispiel: Angenommen, eine Basisklasse *Fahrzeug* definiert eine Methode *bewegen*. Die Klassen *Kraftfahrzeug* und *Schiff* beerben diese Klasse und implementieren diese Methode. Nun erbt eine weitere Klasse *Amphicar* von *Kraftfahrzeug* und *Schiff*. Jetzt ist zunächst nicht klar, welche Implementierung der Methode *bewegen* genutzt werden soll, wenn sie auf einer Instanz der Klasse *Amphicar* aufgerufen wird. Dieses Problem kann gelöst werden, indem die Klasse *Amphicar* die geerbte Methode *bewegen* überschreibt.

```
//Achtung - kein korrekter Scala Code
abstract class Fahrzeug {
   def bewegen: Unit
abstract class Kraftfahrzeug extends Fahrzeug {
   def bewegen = {/* fährt */} }
abstract class Schiff extends Fahrzeug {
   def bewegen = {/* schwimmt */} }
class Amphicar extends Kraftfahrzeug, Schiff {
   override def bewegen = {/* schwimmt oder fährt */} }
```

Programmauszug 2.19: Konfliktlösung durch Überschreiben

Das obige Beispiel ist kein lauffähiger Scala Code, denn Mehrfachvererbung wird in Scala nicht unterstützt. Es soll hier lediglich verdeutlichen, dass es unter Umständen sinnvoll ist Mehrfachvererbung einzusetzen. Da das *Amphicar* sowohl schwimmen als auch fahren kann, wird die Funktionalität der Klassen *Schiff* und *Kraftfahrzeug* benötigt.

Die eben gezeigte Auflösung der Namenskonflikte soll nicht darüber hinwegtäuschen, dass Mehrfachvererbung sehr große Probleme hervorrufen kann. Besonders beim Erweitern und Verändern ("Refactoring") bestehender Architekturen kann die Komplexität der Klassenhierarchien problematisch sein. Besonders schwierig ist das Extrahieren von Methoden aus Vererbungshierarchien mit Mehrfachvererbung, wenn diese Methoden andere Methoden mehrerer Basisklassen verwenden. Ein weiteres Problem ist der Zugriff auf Methoden von Basisklassen, die in tiefergelegenen Basisklassen überschrieben wurden. [Duc06, Seite 2 ff.]

Das Strategie-Entwurfsmuster ist eine Alternative zur Mehrfachvererbung, allerdings werden dabei weitere Klassenhierarchien eingeführt [Gam95, Seite 349 ff.].

Mixins

Ein Mixin stellt Implementierungsdetails für andere Klassen zur Verfügung. Wird eine Klasse um ein Mixin erweitert, entsteht eine neue Klasse, auf die nun ebenfalls ein Mixin angewandt werden kann. Eventuell bereits vorhandene Methoden mit gleicher Signatur werden durch das Einmischen("mix-in") implizit überschrieben. Dies nennt man "lineare Komposition". Dadurch werden die Namenskonflikte der Mehrfachvererbung verhindert. Allerdings ist die Reihenfolge der Anwendung von Mixins auf eine Klasse von Bedeutung. Zum Teil kann es sehr schwer oder auch unmöglich sein, eine korrekte Reihenfolge für die Einmischung der Mixins zu finden. [Sch02, Seite 4]

Ein Hauptproblem bei der Verwendung von Mixins sind die entstehenden tiefen Klassenhierarchien. Wird zum Beispiel einem Mixin eine neue Methode hinzugefügt, werden alle Methoden mit gleicher Signatur in den Klassen überschrieben, die dieses Mixin als letztes

einmischen. Zudem ist es nicht möglich, mehrere Mixins zu einem zusammenzusetzen, um komplexere widerverwendbare Komponenten zu schaffen. [Duc06, Seite 5 ff.]

Traits

Das Konzept der Traits wurde entwickelt, um über einfach wiederverwendbare Komponenten zu verfügen, die die genannten Probleme der Mehrfachvererbung und der Mixins lösen. Traits verbinden die Vorteile der Mixins und der aus Java bekannten Schnittstellen. Einige große Probleme bei der Mehrfachvererbung werden verhindert, da Traits unabhängig von jeglicher Klassenhierarchie sind. Im Gegensatz zu den Mixins können sie in beliebiger Reihenfolge in eine Klasse eingemischt werden.

Traits stellen Funktionalität in Form von Methoden zur Verfügung, die auch als *plugs* bezeichnet werden. Zudem kann ein Trait bestimmte Methoden erfordern, die sich *sockets* nennen. Die *sockets* müssen von einem Klienten befriedigt werden, dagegen können die *plugs* von den Klienten genutzt werden. Ein Klient kann sowohl ein anderes Trait, als auch eine Klasse sein, die das Trait einmischt.

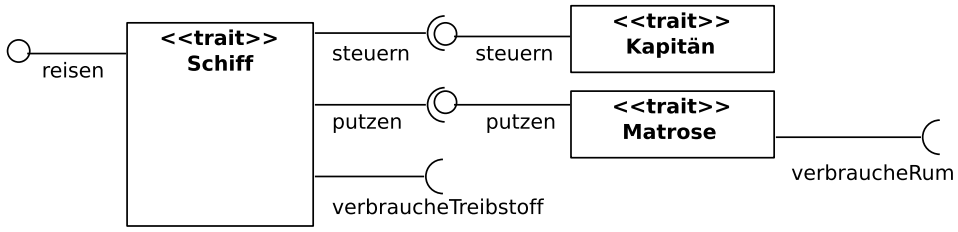

Abbildung 2.1: Traits: plugs und sockets

Das Trait *Schiff* im obigen Bild bietet eine Methode *reisen* an. Damit das Schiff in See stechen kann, muss dessen *socket steuern* befriedigt werden. Dazu kann das Trait *Kapitän* eingemischt werden, denn es verfügt über die implementierte veröffentlichte Methode *steuern* (*plug*).

Traits besitzen keine veränderlichen Variablen, sie sind also zustandslos. Sie können Zustände allerdings mit den als *sockets* definierten Zugriffsmethoden verändern. So wird das Trait *Schiff* aus dem oben gezeigtem Bild bei einem Methodenaufruf (*reisen*) seinerseits die Methode *verbraucheTreibstoff* aufrufen. Das kann den Zustand des Klienten des Trait, also der das Trait einmischenden Klasse, verändern.

Eine weitere wichtige Eigenschaft von Traits ist, dass sie einer Klasse auch außerhalb ihrer Definition direkt bei der Instantiierung, hinzugefügt werden können. Dieses sogenannte “Flattening” ermöglicht die Kombination von Funktionalität, ohne neue Klassen zu schaffen. So könnte direkt bei der Instantiierung eines Klienten des Trait *Schiff* entschieden werden, anstelle der Matrosen ein anderes Trait zu verwenden. Dieses muss allerdings

ebenso ein "plug" *putzen* besitzen, hat aber eventuell kein *socket verbraucheRum*.

Im Prinzip werden bei einem Einmischen alle Methoden des Trait in die Klasse hineinkopiert (mix-in). Die Funktionalität ist dann ebenso in der Klasse vorhanden, als wenn sie dort implementiert wäre. Dafür gibt es zwei Ausnahmen. Zum einen wenn in der Klasse Methoden des Trait überschrieben werden. Ein anderer Konflikt ergibt sich, wenn mehrere verwendete Traits Methoden mit gleicher Signatur anbieten. Dieser muss dann in der Klasse durch ein Überschreiben oder eine explizite Auswahl einer der Methoden gelöst werden. [Sch05, Seite 16 ff.]

Traits ersetzen die Einfachvererbung nicht, sondern liefern eine weitere Möglichkeit zur Gestaltung von Programmen. Im Trait-Konzept wird den Klassen als primäre Rolle die Objektinstantiierung zugewiesen. Die Rolle der Wiederverwendbarkeit soll möglichst auf die Traits übergehen, die keinem anderen Zweck dienen.

Traits und Mixins in Scala

Scala bietet Traits als Sprachelement an. Dabei wurde das ursprüngliche Konzept weiter ausgebaut. [Vgl. Ode05] Zum Beispiel ist es nun auch möglich, Variablen und sogar abstrakte Typen als *sockets* zu definieren. Auf einige wichtige Details, sowie Implementierungsbeispiele der Traits in Scala, wird im Folgenden eingegangen.

Traits definieren und einmischen: Traits in Scala sind spezielle abstrakte Klassen, die nur andere Traits beerben können und keinen Konstruktor besitzen. Sie können wie abstrakte Klassen beerbt werden, wobei es eben auch möglich ist, mehrere Traits gleichzeitig in eine Klasse einzumischen. Dazu ein Beispiel, in dem eine Klasse *Amphicar* zwei definierte Traits einmischt.

```
trait Kraftfahrzeug { def fahren() {/*...*/} }
trait Schiff { def schwimmen() {/*...*/} }
class Amphicar extends Kraftfahrzeug with Schiff
```

Programmauszug 2.20: Definition eines Trait

Die Klasse *Amphicar* ist instantiierbar und erhält die Methoden *fahren* und *schwimmen* von den gezeigten Traits.

Konflikte auflösen: Werden Methoden mehrerer Traits mit gleicher Signatur in eine Klasse eingemischt und existiert dort keine überschreibende Methode, wird der Compiler einen Fehler feststellen. Dies ist das bereits erwähnte Problem der Namenskonflikte (Abschnitt 2.3.3 auf Seite 15). Im nächsten Beispiel "erbt" die Klasse *Amphicar* eine Methode *bewegen* mehrfach.

```
trait Kraftfahrzeug { def bewegen() {/*...*/} }
trait Schiff { def bewegen() {/*...*/} }
class Amphicar extends Kraftfahrzeug with Schiff { } // Kompilierfehler !
```

Programmauszug 2.21: Konflikte und Traits

Zur Lösung dieses Konfliktes muss die Klasse *Amphicar* die Methode *bewegen* überschreiben. Das Schlüsselwort *super* in Kombination mit dem Typparameter erlaubt einen direkten Zugriff auf die *bewegen*-Methoden der eingemischten Traits:

```
class Amphicar extends Kraftfahrzeug with Schiff {
   override def bewegen() {
      if(aufSee) super[Schiff].bewegen()
      else super[Kraftfahrzeug].bewegen()
   /*...*/
}
```

Programmauszug 2.22: Konfliktlösung und Traits

Würde *super* ohne eine Typangabe verwendet, dann würde im Programmauszug 2.22 die Methode *bewegen* aus dem Trait *Schiff* priorisiert. Das Verfahren für diese Auswahl wird "Klassen Linearisierung" genannt. Grundsätzlich werden die zuletzt eingemischten Traits priorisiert. [Ode08b, Seite 226 ff.]

Flattening: Das "Flattening" erlaubt, Traits direkt bei einer Instantiierung in Klassen einzumischen. Dabei kann aus mehreren Traits, deren Kombination keine weiteren *sockets* benötigt, eine anonyme Klasse erzeugt werden. Im nächsten Beispiel wird das Trait *Schiff* verwendet, um mit einem weiteren Trait *Feuerwehr* eine anonyme Klasse zu definieren und zu instantiieren.

```
trait Schiff { def bewegen() {/*...*/} }
trait Feuerwehr { def loeschen() {/*...*/} }
val feuerWehrBoot = new Schiff with Feuerwehr {/*...*/}
feuerWehrBoot.bewegen()
feuerWehrBoot.loeschen()
```

Programmauszug 2.23: Flattening

Die Variable *feuerWehrBoot* besitzt den Typ *Schiff with Feuerwehr*. Dadurch ist es möglich die aus den eingemischten Traits stammenden Methoden aufzurufen. Ebenso wäre es denkbar das Trait *Feuerwehr* mit einem Trait *Kraftfahrzeug* zu kombinieren.

Abhängigkeiten definieren: Zur Definition der *sockets* eines Trait werden in Scala abstrakte Methoden oder auch abstrakte Attribute definiert. Dazu ein Beispiel für die Definition

eines Trait *Matrose*. Dessen *socket* ist das Attribut *amphicar*. Dieses Attribut hat den Typ der bereits in Programmauszug 2.22 auf der vorherigen Seite definierten Klasse *Amphicar*.

```
trait Matrose {
   val amphicar : Amphicar //socket
   def putzen() {/*...*/}
}
val matroseWithAmphicar = new Matrose {
   val amphicar = new Amphicar
}
matroseWithAmphicar.putzen()
```

Programmauszug 2.24: Attribute als sockets

Das Trait *Matrose* wird direkt in eine anonyme Klasse eingemischt, in der ein *Amphicar* instantiiert wird. Die Anforderungen des eingemischten Trait *Matrose* sind somit erfüllt, und die anonyme Klasse kann instantiiert werden.

Eine andere Art und Weise *sockets* zu definieren sind die selbstreferenzierenden Typen. Mit ihnen setzt ein Trait einen bestimmten Typ in seinem Klienten voraus. Dadurch können alle Methoden des selbstreferenzierten Typs innerhalb des Trait verwendet werden. Zu dessen Definition wird innerhalb des Trait folgendes Konstrukt programmiert: *this: T =>*. Hier ein Beispiel, in dem das Trait *Matrose* einen selbstreferenzierten Typ *Amphicar* als *socket* definiert. Dieses Trait wird nun bei einer Instantiierung der Klasse *Amphicar* per "Flattening" eingemischt.

```
trait Matrose {
   this : Amphicar ⇒ //socket
   def putzen() {/*...*/}
}
val amphiCarWithMatrose = new Amphicar with Matrose
amphiCarWithMatrose.putzen()
```

Programmauszug 2.25: Selbstreferenzierte Typen

Es sei angemerkt, dass ein definierter selbstreferenzierter Typ durchaus auch aus mehreren Traits zusammengesetzt sein kann. Zum Beispiel: *this: T => Amphicar with Feuerwehr*.

2.3.4 Singleton-Objekte und Kompagnon-Klassen

In Scala gibt es, im Gegensatz zu anderen Programmiersprachen wie zum Beispiel Java, keine statischen Operationen oder Variablen. Es können allerdings mit dem Schlüsselwort *object* global verfügbare Objekte definiert werden. Diese werden Singleton-Objekte genannt und können wie herkömmliche Objekte nicht beerbt werden. Eine definierte Klassen mit dem gleichen Namen wie ein Singleton-Objekt wird Kompagnon-Klasse genannt.

In der Definition von Singleton-Objekten kann, ähnlich wie bei der Erzeugung von Funktionsobjekten, eine *apply*-Methode definiert werden. Um diese Methode aufzurufen, kann der Funktionsname bei der Applikation weggelassen werden.

```
class Person(private var name : String)
object Person {
   def apply(name : String) = new Person(name)
   def nameVon(person : Person) = person.name
}
val bart = Person("Bart Simpson") //Achtung: hier ist kein "new"
Person.nameVon(bart) //gibt als Resultat: "Bart Simpson"
```

Programmauszug 2.26: Singleton-Objekt und Kompagnon-Klasse

Im Programmauszug 2.26 wurde ein Objekt der Klasse *Person* instantiiert (*bart*), ohne den Konstruktor der Klasse direkt zu verwenden. Es wurde stattdessen die Methode *apply* des Singleton-Objektes benutzt. Weiterhin wurde gezeigt, dass ein Singleton-Objekt Zugriff auf die privaten Attribute und Methoden der Kompagnon-Klasse hat. Denn in der Methode *nameVon* wird das private Attribut *name* der Klasse *Person* verwendet.

2.4 Typsystem und Generizität

Ein Typ bezeichnet die Klassenzugehörigkeit, kann aber auch lediglich das Vorhandensein bestimmter Funktionalität kennzeichnen. Durch die Ausnutzung der Polymorphie kann eine Subklasseninstanz im Kontext des Typs der Basisklasse verwendet werden, wobei das Verhalten der Objekte durch die Implementierung der Subklasse bestimmt wird.

Eine statische Typisierung verhindert bestimmte Fehler, vereinfacht das Testen, beschleunigt die Ausführung und verringert den Speicherverbrauch einer Anwendung. Dafür muss der Programmierer mehr Aufwand für die Verwaltung der Typisierung betreiben. Scala ist statisch typisiert und verringert diesen Aufwand mit Hilfe der Typinferenz.

Scala ist strikt objektorientiert. Jeder Wert in Scala ist ein Objekt. Wie die Instanz einer selbst definierten Klasse, ist damit auch jede Funktion ein Wert. Alle Operationen sind Methoden von Klassen. Es gibt keinerlei statische Werte oder primitive Typen wie zum Beispiel in Java.

In diesem Abschnitt wird das Typsystem von Scala näher betrachtet. Es wird auf die Typinferenz, auf die Möglichkeit zur Typumwandlung und auf die generischen Klassen von Scala eingegangen.

2.4.1 Any - Basis aller Klassen

Die Klasse *scala.Any* ist die Basis aller Klassen in Scala. *scala.AnyVal* ist eine der zwei Subklassen von *Any*. Die Subklassen von *AnyVal* werden bei einer Kompilierung zu Java-Bytecode als native Datentypen umgesetzt. Zu diesen von *AnyVal* abgeleiteten Klassen gehört neben bekannten Vertretern wie *Boolean*, *Float* oder *Int* auch die Klasse *Unit*. Diese kann mit dem Typ *void* anderer Sprachen verglichen werden. Sie wird immer dann verwendet, wenn eine Funktion kein "greifbares" Resultat liefert. Eine weitere von *Any* abgeleitete Klasse ist *scala.AnyRef*. Von dieser Klasse werden alle Klassen abgeleitet, die nicht Kinder von *AnyVal* sind. Der Name suggeriert, dass in der Java-VM mit Referenzen auf Objekte dieser Klassen gearbeitet wird.

2.4.2 Implizite Typzuweisung (Typinferenz)

In den vorangegangenen Beispielen wurde bereits von der Typinferenz Gebrauch gemacht. Zum Beispiel bei der Initialisierung einer Variablen: *val i = 0*. Bei dieser Zuweisung wird der Variablen *i* implizit der Typ *Int* zugewiesen. Ohne diese Typinferenz zu nutzen, müsste Folgendes geschrieben werden: *val i: Int = 0*. Nun ein etwas komplexeres Beispiel für die Typinferenz.

```
def createAuto(sollSchwimmen : Boolean) = {
   if (sollSchwimmen) new Amphicar
   else new Auto
}
```

Programmauszug 2.27: Typinferenz

Wird vorausgesetzt, dass die Klassen *Amphicar* und *Auto* ein Trait *Kraftfahrzeug* implementieren, dann hat das Resultat der Methode *createAuto* den Typ *Kraftfahrzeug*. Ihre Signatur ist also äquivalent zu:

createAuto(sollSchwimmen: Boolean): Kraftfahrzeug

Der Compiler versucht bei der Typinferenz selbständig einen passenden Typ zu finden. Dabei wird der nächstliegende gemeinsame Basistyp aller möglichen Resultate eines Ausdruckes gewählt.

2.4.3 Explizite und implizite Typumwandlung

Einem Wert ist immer ein bestimmter Typ zugewiesen. Will man diesen Wert nun im Kontext eines anderen Typs verwenden, muss eine Typumwandlung stattfinden. Für alle Basistypen des Wertes ist das ohne weiteres möglich und kann durch die Typinferenz implizit erfolgen (upcast). In die andere Richtung, also bei einer Typumwandlung zu einem

Subtyp, wird dies mit der aus *scala.Any* stammenden Methode *asInstanceOf* umgesetzt (downcast):

```
val any :Any = new Amphicar //upcast
val amphicar :Amphicar = any.asInstanceOf[Amphicar] //downcast
```

Programmauszug 2.28: Typumwandlung

Dabei wird der Methode *asInstanceOf* ein Typ als Parameter übergeben. Diese erzwungene Typumwandlung führt zu einem Laufzeitfehler, wenn das dem Wert zugehörige Objekt nicht diesen Typ besitzt. Im nächsten Beispiel wird vergeblich versucht, eine Zeichenkette (*String*) in eine Zahl (*Int*) umzuwandeln.

```
"Ich will eine Zahl sein".asInstanceOf[Int] //Laufzeitfehler !
```

Programmauszug 2.29: Typumwandlungsfehler zur Laufzeit

2.4.4 Generische Klassen und Funktionen

Ein weiteres wichtiges Konzept ist die Generizität. Es können sogenannte generische Klassen und Traits definiert werden, die mit Objekten eines nicht genau spezifizierten Typs arbeiten können. Dazu wird bei einer Klassendefinition ein abstrakter Typ als Parameter definiert, indem der Konstruktor eine zusätzliche Parameterliste erhält, die von eckigen Klammern umgeben ist.

```
class Wiederholer[T](t : T) {
   def printOut = print(t.toString + ", " + t.toString)
}
val p = new Wiederholer(1)
p.printOut //gibt "1, 1" auf der Konsole aus
```

Programmauszug 2.30: Generische Klasse und Instantiierung

Im gezeigtem Beispiel wurde die generische Klasse *Wiederholer[T]* erzeugt. Der in ihr verwendete abstrakte Typ *T* wird erst bei der Instantiierung der Klasse angegeben. Hier ergibt er sich durch die Typinferenz aus dem Typ des Konstruktorparameters. Eine explizite Angabe würde wie folgt aussehen:

```
val p = new Wiederholer[Int](1)
```

Programmauszug 2.31: Generische Klasse mit explizitem Typparameter

Im vorigen Beispiel wurde die generische Klasse *Wiederholer[T]* mit dem Typ *Int* parametrisiert. Dadurch wurde von einer generischen Klasse eine abgeleitete Klasse *Wiederholer[Int]* erzeugt.

Innerhalb der generischen Klasse wurde die Methode *toString* des Objektes aufgerufen, dessen Typ mit dem den abstrakten Typ *T* spezifiziert ist. Das ist möglich, da der Typ *T* eine implizite obere Typgrenze *scala.Any* besitzt, die festlegt, dass als Typen alle Subtypen von *Any* und *Any* selbst erlaubt ist. Eine explizite obere Typgrenze kann wie folgt definiert werden:

```
class Wiederholer[T <: MyInt](t: T) {/*...*/}
```

Programmauszug 2.32: Generische Klasse und explizite obere Typgrenze

In Scala gibt es auch die Möglichkeit, polymorphe (bzw. generische) Funktionen zu definieren, die einen Typparameter erwarten. Ein Beispiel ist die in Abschnitt 2.4.3 auf Seite 22 genutzte Methode *scala.Any.asInstanceOf[T]*.

```
def doppelterPrintOut[T](t: T) {
   print(t.toString + ", " + t.toString)
}
doppelterPrintOut(1) //gibt "1, 1" auf der Konsole aus
```

Programmauszug 2.33: Generische Funktion

Zu den abstrakten Typen gibt es noch einiges mehr zu schreiben. Zum Beispiel kann durch die Angabe von oberen und unteren Grenzen genau eingeschränkt werden, welche Typen als Parameter zulässig sind. Zudem kann das Klassenhierarchieverhalten der Klassen beeinflusst werden, die von generischen Klassen abgeleitet wurden [Ode08b, Seite 388 ff.]. Genauere Ausführungen würden den vorgesehenen Umfang der Arbeit übersteigen.

2.5 Objektorientierter Musterabgleich

Mit Hilfe des objektorientierten Musterabgleichs werden Objekthierarchien untersucht, wobei das über externe Zugriffe auf die Objekte vorgenommen werden kann. Ziel ist es festzustellen, ob ein Objekt eine bestimmte Struktur, einen bestimmten Typ oder bestimmte Attribute besitzt. [Vgl. Emi07]

2.5.1 Case-Klassen

In Scala erben alle von *Any* abgeleiteten Klassen die Methode *equals(arg0 : Any): Boolean*. Diese soll für die Instanzen einer Klasse immer den Wert *true* liefern, wenn eine Wertgleichheit zwischen den verglichenen Objekten vorliegt.

Um eine Wertgleichheit zu prüfen, wird zunächst die Referenzgleichheit geprüft. Sind die Referenzen nicht gleich, wird eine Typprüfung vorgenommen und im Anschluss

werden die Attribute verglichen. Der Aufwand für die Implementierung steigt mit Anzahl der Attribute. Zudem müssen auch die Klassen der Attribute eine *equals* Methode implementieren.

In Scala können sogenannte Case-Klassen erzeugt werden. Diese verfügen über eine Standard Implementierung von *equals* und auch von *hashCode*. Dabei fließen in diese Implementierungen allerdings nur die im Konstruktor definierten Attribute ein. Zudem ist die Case-Klasse eine Kompagnon-Klasse eines Singleton-Objektes. Die *apply*-Methode des Singleton-Objektes hat eine Parameterliste, die mit der des Konstruktors der zugehörigen Case-Klasse identisch ist.

```
case class Person(var name : String, var tel : Int)
val bart = Person("Bart Simpson", 911)
bart.equals(Person("Bart Simpson", 911)) // liefert true
bart.equals(Person("Clancy Wiggum", 911)) // liefert false
```

Programmauszug 2.34: Case-Klasse

In diesem Beispiel werden Objekte der Klasse *Person* unter Verwendung der *apply*-Methode des Singleton-Objektes erzeugt. Die Methode *equals* kann nun verwendet werden, um die instantiierten Objekte zu vergleichen.

2.5.2 Match-Ausdruck

In den meisten imperativen Sprachen, wie zum Beispiel Java, existiert eine "Switch"-Anweisung. Mit ihr kann der Wert einer Variablen für bedingte Anweisungen genutzt werden. Es findet dabei ein Vergleich der Variablen mit angegebenen Werten statt.

In Scala gibt es eine ähnliche Anweisung. Mit ihr können sowohl Wert- als auch Typvergleiche vorgenommen werden.

```
def test(obj : Any): String = obj match {
   case 1 ⇒ "Int: eins"
   case y : Int ⇒ "Int: " + y.toString
   case _ ⇒ "kein Int"
}
```

Programmauszug 2.35: Match Ausdruck

Der Platzhalter Unterstrich (_) innerhalb eines "Pattern" steht für jeden möglichen Wert. Dieser kann durch die Angabe des Typs eingeschränkt werden. Zudem können in einem Pattern neue Variablen definiert werden, die im Rumpf des "Case"-Ausdrucks verwendet werden können.

Nun ist es auch möglich, Objekte von Case-Klassen als Pattern zu verwenden. Diese können innerhalb des Match-Ausdruckes mit Hilfe des zugehörigen Singleton-Objektes

erzeugt werden. Dabei wird die apply Methode des Singleton-Objektes aufgerufen, um Objekte der Case-Klassen zu instantiieren.

```
case class Adresse (ort : String)
case class Person (name : String, adresse : Adresse)
def ausSpringfield(person : Person) = person match {
   case Person(_,Adresse("Springfield")) ⇒ true
   case _ ⇒ false
}
```

Programmauszug 2.36: Match Ausdruck und Case-Klassen

Der Platzhalter (_) führt dazu, dass bei einem Vergleich des Attributes *name* mit irgend einem Wert immer *true* zurückgegeben wird. Dadurch wird die Operation *ausSpringfield* für alle Objekte des Typs *Person* ein *true* zurückgeben, sobald die Adresse übereinstimmt.

2.6 Implizite Definitionen

Oft möchte man vorhandenen Klassen weitere Funktionalität hinzufügen. Dies kann mit Hilfe der Vererbung umgesetzt werden. Es ist aber manchmal nicht möglich, Vererbung einzusetzen. Denn Klassen können dies verbieten, indem ihrer Signatur das Schlüsselwort *final* hinzugefügt wurde. Als Beispiele für diese "finalen" Klassen seien die Klassen *Int* oder *String* aus der Scala-API genannt. Eine Lösung für dieses Problem bietet zum Beispiel das "Adapter"-Entwurfsmuster [Gam95, Seite 157 ff.].

Der Nachteil bei der Vererbung und auch dem Adapter-Entwurfsmuster ist, dass neue Klassen erzeugt werden. Zudem müssen die Objekte, mit denen auf die neue Funktionalität zugegriffen werden kann, auch mit Hilfe der neu definierten Klasse (oder Schnittstellen) instantiiert werden. Also muss unter Umständen der Quellcode an vielen Stellen verändert werden.

In Scala gibt es eine weitere sehr interessante Möglichkeit. Es können Operationen implizit definiert werden. Damit kann Klassen Funktionalität hinzugefügt werden, indem man diese implizit in eine andere Klasse transformiert.

```
implicit def intSquare(num : Int) = new {
   def square(pow : Int) = Math.pow(num, pow)
}
2.square(3) // liefert 8
```

Programmauszug 2.37: Implizite Definitionen

Im diesem Beispiel wurde eine implizite Operation *intSquare* erzeugt. Diese fügt Objekten des Typs *Int* eine neue Methode *square* hinzu. Die Operation *intSquare* muss nicht explizit

aufgerufen werden.

Die Methode *square* wird direkt auf ein Objekt des Typs *Int* angewandt. Bei der Kompilierung wird zunächst in der Klasse *Int* nach dieser Methode gesucht. Wird hier keine gefunden, wird geprüft, ob implizite Definitionen dieses Problem beheben können.

2.7 Operatoren und Operatorschreibweise

In diesem Abschnitt soll auf die Operator Schreibweise eingegangen werden. In Scala ist die Definition von Operatoren als Methoden erlaubt, dabei können sogar Unicode-Zeichen verwendet werden.

In den vorangegangenen Beispielen sind diese Operatoren schon mehrfach verwendet worden, wenn mit Objekten der Klasse *Int* gearbeitet wurde. Denn die Klasse *Int* verfügt unter anderem über eine Methode mit der Signatur *def +(num: Int): Int*.

Ein Beispiel für die Verwendung der sogenannten "Infix"-Schreibweise von Operatoren ist somit der Ausdruck *1 + 1*. Dieser Ausdruck ist äquivalent zu *1.+(1)*. Es ist in Scala erlaubt, die "Infix"-Schreibweise zu nutzen, wenn keine weitere Methode gleichen Namens ohne Parameter existiert.

Die Klasse *Int* verfügt nicht über einen "Postfix"-Operator ++, wie er zum Beispiel in Java gern verwendet wird. Diesen kann man sich aber mit Hilfe von impliziten Definitionen selbst definieren:

```
implicit def intToMyInt(num : Int) = new {
   def ++ = num + 1 //Operator ++
}
3++ //liefert 4
```

Programmauszug 2.38: Postfix-Operator

Um Methoden für die "Präfix"-Schreibweise zu definieren, ist man auf einstellige Operatoren beschränkt. Für ihre Definition muss vor dem eigentlichen Operatorsymbol die Zeichenkette *unary_* angegeben werden:

```
class MyInt(num : Int) {
   def unary_- = new MyInt(-num)
}
val m = new MyInt(10)
-m //liefert MyInt(-10)
```

Programmauszug 2.39: Präfix-Operator

Funktionen für die sogenannte "Mixfix"-Schreibweise zu definieren, bei der mehrere Parameter mit selbst definierten Schlüsselwörtern oder Operatoren getrennt werden,

wird grundsätzlich nicht mit integrierten sprachlichen Mitteln unterstützt, wie sie in anderen Sprachen (z.B. Haskell) zu finden sind. Ein Beispiel für die Nutzung der "Mixfix"-Operatorschreibweise sind die Vergleiche mehrerer Zahlen (z.B. Int):

$x < y < z$

Der obige Ausdruck würde in Scala von links nach rechts ausgewertet und eine Funktion mit dem Resultat der vorigen appliziert. Das heißt, wenn x und y verglichen werden, dann ist das Resultat vom Typ *Boolean*. Ein Vergleich von y und z kann nicht mehr stattfinden.

Es können allerdings implizite Definitionen und Klassen verwendet werden, um eine "Mixfix"-Schreibweise zu ermöglichen. Dabei wird das Resultat und der Parameter einer Applikation in einer Objektinstanz gekapselt.

```
class IntBool(val bool: Boolean, value: Int) {
   def less(other: Int) = new IntBool(bool && value < other, other) }
implicit def extendInt(num : Int) = new {
   def less(b : Int) = new IntBool(num < b, b) }
implicit def boolFromIntBool(intbool : IntBool) = intbool.bool
if(1 less 2 less 3 less 4){ println("ja") } //gibt "ja" aus
```

Programmauszug 2.40: Mixfix-Operatorschreibweise

Hier stoßen die impliziten Definitionen an ihre Grenzen, denn wenn statt dem Methodennamen *less* versucht wird den Operator < zu verwenden, wird dies bei der Kompilierung nicht aufgelöst, da die Klasse *Int* bereits über eine Methode mit gleicher Signatur verfügt.

2.8 Zusammenfassung

Wie gezeigt, stehen in Scala die Mittel funktionaler und objektorientierter Sprachen zur Verfügung. Dabei ist es dem Programmierer überlassen, wie weit er das grundlegende Konzept der funktionalen Programmierung - die referenzielle Transparenz - in seinen Programmen umsetzt. Das objektorientierte Konzept ist hingegen strikter Bestandteil der Sprache.

Die sprachlichen Mittel von Scala gehen über die von Java deutlich hinaus. Partielle Funktionen, impliziten Definitionen und Operatoren erlauben es, eigene sprachliche Konstrukte zu definieren. Mit Hilfe der Traits von Scala, als Alternative zur Mehrfachvererbung, können leicht wiederverwendbare Komponenten entwickelt werden. Scala und Java sind statisch typisierte Sprachen. Mit der Typinferenz von Scala ist dabei eine explizite Angabe von Typen meist unnötig.

3 Die Android-Plattform

In diesem Kapitel wird Android eine Open-Source-Plattform für mobile Geräte vorgestellt. Dazu werden die Architektur des Systems, das Android-Framework und die Hauptbestandteile einer Android-Anwendung erläutert.

3.1 Einleitung

Android ist ein Produkt der Open-Handset-Alliance unter der Führung der Google Inc.. In dieser Allianz finden sich bedeutende Unternehmen der Kommunikations- und IT-Branche.

Der Großteil der Android-Plattform ist unter der "Apache Software Licence 2.0" veröffentlicht, weshalb kommerzielle Anwendungen ohne Erlaubnis der Open Handset Alliance entwickelt werden können. Auch das Entwickeln proprietärer Anwendungen ist mit dieser Lizenz möglich. [Apa04]

Der Kernel der Android-Plattform ist unter der "Gnu Public License v2" (GPL) veröffentlicht, denn er basiert auf einem herkömmlichen Linux-Kernel. Die GPL sagt aus, dass alle Änderungen am Linux-Kernel veröffentlicht werden müssen. [Tor91]

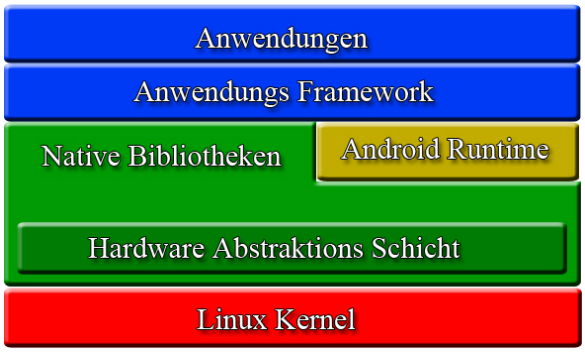

Abbildung 3.1: Android-System-Architektur

Anwendungen für Android werden in der Programmiersprache Java entwickelt. Die Kernbibliothek der Android-API ist fast so umfangreich wie die API des Standard Java-SDK. Mit Hilfe eines speziellen Compilers kann Java-Bytecode in Android-Bytecode übersetzt werden. Das hat den Vorteil, dass bereits existierende Java-Bibliotheken meist mühelos verwendet werden können. Durch die Entwicklung einer eigenen virtuellen Maschine, die diesen speziellen ByteCode benötigt, wird eine Unabhängigkeit von den "Sun Lizenzen" erreicht.

Kenntnisse über die Architektur und die technischen Details des Android-Systems sind für die Entwicklung von Anwendungen durchaus von Bedeutung, denn das Java-

Framework und auch die Laufzeitumgebung sind eng mit den unteren Schichten des Systems verknüpft. Einige Details haben vor allem ihre Ursache darin, dass das Android-System stark auf Performance und geringen Speicherverbrauch ausgelegt wurde, da die Zielgeräte in etwa die Leistung herkömmlicher PCs der 1990er Jahre besitzen.

In den folgenden Abschnitten wird auf die einzelnen Teile der Plattform genauer eingegangen. Einen ersten Überblick gibt die Abbildung 3.1 auf der vorherigen Seite.

3.2 Die Basis der Plattform

Dieser Abschnitt beschäftigt sich mit den unteren Schichten der Android-Plattform. Zunächst wird auf die Kernel-Ebene eingegangen. Im Anschluss werden die spezielle "User Space"-Hardwareabstarktionsebene und einige native Bibliotheken vorgestellt. Am Ende des Abschnittes wird der für die Anwendungsentwickler wichtigste Teil, die Android-Laufzeitumgebung, erläutert. Sie umfasst die virtuelle Maschine und die Java-Kern-Bibliotheken.

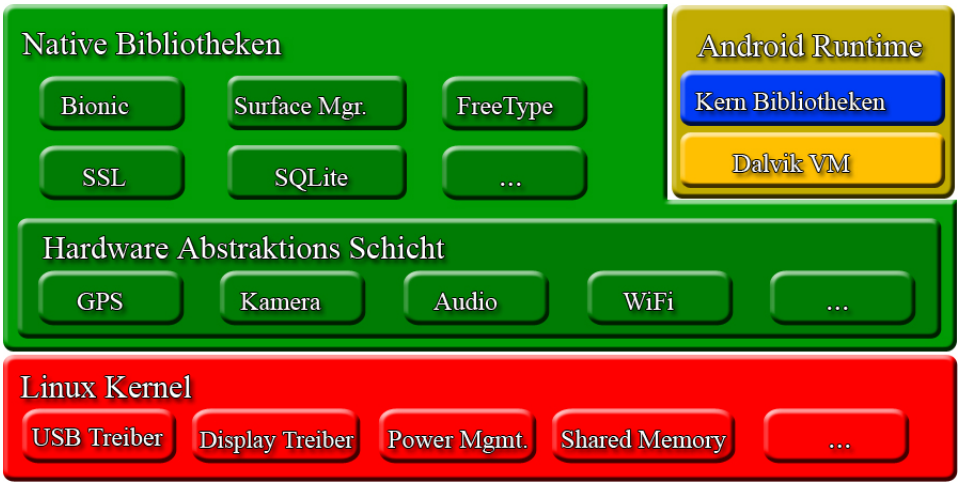

Abbildung 3.2: Android-System-Architektur: Basis

3.2.1 Linux-Kernel

In der Android-Plattform kommt eine modifizierte Version des Linux-Kernel (2.6.25) zum Einsatz. Der Kernel wurde für den Einsatz in mobilen Geräten optimiert und an die speziellen Anforderungen der Hardware angepasst. Zur Zeit ist er in allen Systemen mit ARM-Prozessoren lauffähig. Es gibt Bestrebungen, ihn auch für andere Prozessorarchitekturen umzusetzen, womit die Android-Plattform dann auch auf anderen Architekturen lauffähig wäre.

Ein Grund für die Modifizierung des Kernel sind die speziellen in Handys verwendeten Hardwarekomponenten. Für diese muss der Kernel geeignete Treiber zur Verfügung

stellen. Zudem stellt der Einsatz in mobilen Geräten sehr hohe Anforderungen an Ressourcenverbrauch, Performance und Stromverbrauch. Aus diesem Grund wurden eigene Kernel-Module für kritische Bereiche entwickelt und andere nicht benötigte Module (z.B. ACPI) aus dem Kernel entfernt.

Erweiterungen des Kernel sind zum Beispiel der Kernel-Debugger, ein Logger und der sogenannte "low memory killer". Dieser beendet Prozesse, wenn der Hauptspeicher des Gerätes nicht mehr ausreicht. In Anwendungen muss also beachtet werden, dass sie ohne ausdrücklichen Wunsch des Benutzers beendet werden können.

Die Interprozesskommunikation des Kernel ist mit einer eigenen "shared memory" Lösung umgesetzt (ashmem driver). Indem sich Prozesse Speicherbereiche für den Informationsaustausch teilen, entfällt ein aufwändiges Kopieren der Daten. In dieser Umsetzung arbeiten die Prozesse mit Proxy-Objekten, die die Referenzen auf die eigentlichen Daten im Speicher halten. Die Synchronisierung der Prozesse wird mit einem sogenannten "Binder-Driver" realisiert. Dieser regelt wann welcher Prozess mit welchen Daten arbeiten kann.

3.2.2 Native Bibliotheken

Die Android-Plattform nutzt viele native Bibliotheken, also Bibliotheken, die für den Prozessor des Gerätes kompiliert wurden. Im Folgenden wird auf die unterschiedlichen Kategorien dieser Bibliotheken eingegangen.

Hardware-Abstraktions-Ebene

Die Hardware-Abstraktions-Ebene (HAL) im "User Space"[1] setzt auf die HAL des Kernel auf. Mit ihr können Treiber für die Geräte-Komponenten außerhalb des Kernel realisiert werden, die dann auch außerhalb der strengen GPL-Lizenz liegen. Damit wurde den Anforderungen vieler Hardwarehersteller Rechnung getragen, die den Treiber-Quellcode unveröffentlicht lassen möchten.

In der "User Space"-HAL sind die Schnittstellen der Treiber definiert, die von den Hardwareherstellern implementiert werden müssen. Es gibt eine Reihe optionaler Hardware-Komponenten, die dabei nicht zwangsläufig umgesetzt werden müssen. Dazu gehören zum Beispiel die Treiber für ein GPS-Modul, einen Touchscreen oder eine Kamera. In der Android-Plattform selbst (für den Emulator) sind lediglich Referenzimplementierungen enthalten.

1 User-Space: Der Bereich des virtuellen Speicherbereiches, der für Benutzer-Anwendungen zur Verfügung steht. Im Gegensatz dazu gibt es den Kernel-Space, der nur vom Kernel und seinen Erweiterungen (z.B. Treiber) genutzt werden darf.

Die libC-Implementierung Bionic

Im Android-System wurde die "Gnu C"-Bibliothek (Gnu-libc) herkömmlicher Linux-Systeme durch die libC-Implementierung Bionic ersetzt. Bionic ist für den Einsatz in eingebetteten Systemen optimiert und entstammt der Berkeley-Software-Distribution (BSD). Damit steht Bionic unter der BSD-Lizenz steht, die es erlaubt, auf ihrer Basis Software zu entwickeln, ohne den Quellcode offen zu legen [Bsd06], was durch die GPL verboten ist, unter der die Gnu-libc steht.

Bionic setzt nicht alle Funktionen des Posix-Standards um; zum Beispiel werden C++ Ausnahmen (exceptions) nicht unterstützt. Andererseits ist die Bibliothek nur in etwa halb so groß wie die Gnu-libc, was sehr vorteilhaft ist, da sie für jeden Prozess in den Speicher geladen wird.

Funktions-Bibliotheken

Zum Android-Systems gehören bekannte Bibliotheken, wie zum Beispiel die Browserengine "WebKit". Für die Umsetzung von Datenbanken wird die SQLite-Bibliothek verwendet. Unter den Media-Bibliotheken findet man unter anderem die libjpeg, für das komprimieren und dekomprimieren von JPEG Bildern. Weitere Bibliotheken sind u.a. Freetype oder OpenGL | ES. Schwerpunkt bei der Auswahl für diese Bibliotheken war in erster Linie die Performance und die Quelloffenheit [PB08].

Native Dienste

Die nativen Dienste ermöglichen höheren Ebenen den Zugriff auf bestimmte Hardwarekomponenten. Zu ihnen gehört der Surface-Manager zur Verwaltung des Framebuffer des Displays, der für das Zeichnen der grafischen Komponenten verantwortlich ist. Alle Anwendungen mit einer grafischen Oberfläche müssen ihn nutzen. Ein anderes Beispiel ist der Audio-Manager, der die verschiedenen Audiokanäle verwaltet.

3.2.3 Android-Laufzeit-Umgebung

Die Laufzeitumgebung (Runtime) im Android-System besteht aus der speziell für Android entwickelten virtuellen Maschine Dalvik, sowie einer Anzahl von Java-Kern-Bibliotheken.

Virtuelle Maschine Dalvik

Durch den Einsatz einer virtuellen Maschine (VM) sind Anwendungen von der Hardware unabhängig und damit portabel. Es ist lediglich von Bedeutung, dass die virtuelle Maschine auf dieser Hardware läuft. Die virtuelle Maschine Dalvik ist für den mobilen Einsatz

im Android-System optimiert und führt speziellen Dalvik-Bytecode aus. Sie wurde unter der Leitung des Google-Mitarbeiters Dan Bornstein entwickelt.

Im Unterschied zu der stapelbasierten Java-VM basiert die Dalvik-VM auf einer Register-basierten virtuellen Prozessor-Architektur. Während für Registermaschinen die einzelnen Register in den Instruktionen kodiert werden, müssen Stapelmaschinen die Speicher-adressen auf einen Stapel laden. Dadurch besitzen Stapelmaschinen im Allgemeinen eine höhere Code-Dichte und die kompilierten Programme sind kleiner. Dagegen kann mit Register-basierten Systemen eine höhere Performance erreicht werden, da der Zugriff auf einzelne Speicherzellen (Register) effizienter möglich ist. [DB08]

Dalvik-Bytecode Die für die Android-Plattform entwickelten Java-Programme werden zunächst zu herkömmlichen Java-Bytecode (class-Dateien) kompiliert. Im Anschluss ist eine Kompilierung in das DEX (Dalvik Executable) Format notwendig, womit alle class-Dateien in eine dex-Datei überführt werden, die den unkomprimierten Dalvik-Bytecode enthält.

Der optimierte Dalvik-Bytecode ist etwa halb so groß wie der Java-Bytecode, was unter anderem durch das Entfernen der Redundanz identischer Konstanten in den verschiedenen Java-Bytecode Dateien erreicht wird. Zudem benötigen alle diese Konstanten des sogenannten “constant pool” nur 16 Bit. Dieser “constant pool” ist eine Tabelle, in der unter anderem Bezeichnernamen und Referenzen auf Klassen oder Methoden vorgehalten werden. Im Java-Bytecode werden für einige dieser Konstanten sogar 64 Bit benötigt. [DB08]

Java-Kern-Bibliotheken

Die Java-Kern-Bibliotheken von Android decken den Großteil der Funktionalität ab, den die herkömmliche Bibliothek des Java-Standards von Sun zur Verfügung stellt. Die Spezifikation der vorhandenen Schnittstellen ist identisch, da einige fehlen ist eine vollständige Kompatibilität nicht gegeben, weshalb der Einsatz vieler Java-Bibliotheken nicht möglich ist. Es folgt eine Auflistung der wichtigsten Bibliotheken der “Java 5.0 Standard Edition”, die in der Android-API nicht oder nur zu einem sehr geringen Teil enthalten sind.

- java.applet (Applets)
- java.awt (grafische Oberflächen)
- java.beans (JavaBeans-Komponenten)
- java.lang.management (Überwachung und Verwaltung der Java-VM)
- java.rmi (Remote-Method-Invocation)

- javax.imageio (Bildverarbeitung)
- javax.naming (Namensdienste z.B. DNS)
- javax.swing (grafische Oberflächen)
- javax.xml (XML-Verarbeitung)

Die genannten Pakete werden zum Teil durch Eigenentwicklungen, aber auch durch Pakete von Drittanbietern ersetzt. Es handelt sich dabei um speziell angepasste Implementierungen:

- org.apache.http (HTTP-Client Implementierung)
- junit (Testframework)
- org.json (Javascript-Object-Notation)

Für die Programmierung der grafischen Oberfläche, den Zugriff auf die SQLite-Datenbank oder auch auf die Funktionalität des Telefons, existiert eine API im Paket *android*, die speziell für das Android-System entwickelt wurde. Hier ein Überblick über deren wichtige Bibliotheken:

- android.app (Android Anwendungsmodell)
- android.content (Zugriff auf die Daten des Systems)
- android.database.sqlite (Zugriff auf die SQLite-Datenbank)
- android.location (GPS - “location based services”)
- android.sax (XML Verarbeitung - SAX)
- android.test (Testen der Anwendung)
- android.view (allgemeine Klassen der grafischen Oberfläche)
- android.widget (spezielle Komponenten der grafischen Oberfläche)

3.3 Anwendungs-Framework

Das eigentliche Anwendungs-Framework besteht aus einer Reihe von Java-Klassen, die dem Paket *android* angehören. Dazu gehören einige Dienste von *android.app*, die im Hintergrund arbeiten und die Anwendungen verwalten. Dabei interagieren sie mit den nativen Diensten der unteren Schichten des Systems.

Im Folgenden werden wichtige Bestandteile, wie zum Beispiel die System-Dienste des Android-Framework erläutert. Dabei werden die zugehörigen Java-Klassen und Schnittstellen genannt. Die interne Umsetzung der Komponenten ist nicht ausschließlich in

Java implementiert. Die Java-Klassen und Schnittstellen dienen meist dazu, die nativen Systemdienste aus den entwickelten Java-Anwendungen zu nutzen.

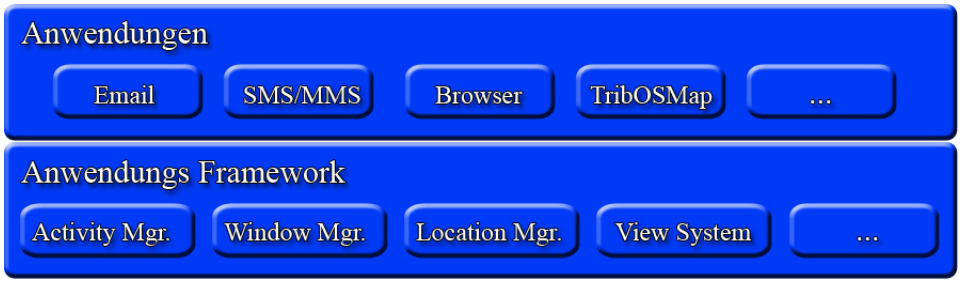

Abbildung 3.3: Android-System-Architektur: Framework und Anwendungen

3.3.1 Activity-Manager

Der Activity-Manager (*android.app.ActivityManager*) verwaltet den Lebens-Zyklus gestarteter Anwendungen, die aus mehreren Activities bestehen kann. Ein Activity ist ein Objekt, das einen unabhängigen Teil von Funktionalität zur Verfügung stellt und in der Regel über eine Bildschirmrepräsentation verfügt. Die Activities werden von der Klasse *android.app.Activity* abgeleitet und müssen beim Activity-Manager angemeldet werden, der unter anderem die Interaktion zwischen den einzelnen Activities ermöglicht.

Diese kommunizierenden Activities laufen in separaten virtuellen Maschinen. Der Activity-Manager arbeitet mit dem Binder-Driver des Kernel zusammen, der die Interprozesskommunikation umsetzt. Dabei nutzen die gestarteten virtuellen Maschinen gemeinsame Speicherbereiche, wodurch die Java-Kernbibliotheken nicht mehrfach im Speicher gehalten. Auch die Daten für die Kommunikation zwischen den Activities werden nicht kopiert, sondern lediglich Referenzen auf die Speicherbereiche ausgetauscht.

Der Activity-Manager kann Activities starten, pausieren oder beenden. Dies kann durch eine Benutzerinteraktion oder ein Ereignis des Systems ausgelöst werden. Die laufende Activities werden vom Activity-Manager in einem Stapel verwaltet. Wird eines geschlossen, wird das vorherige Activity aktiviert.

3.3.2 Package-Manager

Der Package-Manager (*android.content.pm.PackageManager*) verwaltet Metainformationen zu den installierten Anwendungen und Diensten. Wird beim Package-Manager eine bestimmte Funktionalität angefragt (Intent), sucht dieser passende Activities. Er verwaltet die Daten der Anwendungen, die alle Arten von Ressourcen wie zum Beispiel Bilder, Texte oder XML-Layout-Dateien umfassen. Der Package-Manager ist grundsätzlich auch notwendig, um auf die Ressourcen des Systems wie Datenbanken oder Dateien zuzugreifen.

3.3.3 Window-Manager

Der Window-Manager (*android.view.WindowManager*) ermöglicht den Zugriff auf die grafische Oberfläche, zum Beispiel um Komponenten der Benutzerschnittstelle zu zeichnen oder Videos auszugeben. Dabei arbeitet er mit dem Surface-Manager der nativen Ebene zusammen. Der vom Window-Manager abgeleitete View-Manager wird vom Activity verwendet, um Widgets wie "Buttons" oder "TextViews" auf dem Bildschirm anzuzeigen.

3.3.4 Hardware-Dienste

Zu den Hardware-Diensten gehört zum Beispiel der Location-Manager, der aus der Anwendung heraus angesprochen werden kann, um Positionsangaben vom GPS-Gerät zu erhalten (android.location.LocationManager). Andere Hardwaredienste erlauben den Zugriff auf die Telefonfunktion, auf Audiogeräte oder auch auf verschiedene Sensoren.

3.3.5 View-System

Das View-System besteht aus einer Menge von Klassen, mit der die Komponenten der grafische Benutzerschnittstelle programmiert werden, diese werden Widgets genannt. Alle Widgets beerben die Klasse *android.view.View*, die das Kompositum-Entwurfsmuster umsetzt [Gam95, Seite 183 ff.]. Das bedeutet, dass ein Widget andere Widgets beinhalten kann und somit die Oberfläche eine Baumstruktur besitzt.

Die bereits implementierten Oberflächen-Komponenten des Paketes *android.widget* umfassen unter anderem Klassen für die Ausgabe von Text, Schaltflächen oder auch für die Strukturierung von Layouts. Um eigene Widgets zu implementieren, muss die Klasse *View* beerbt und müssen Methoden geeignet überschrieben werden.

3.4 Anwendungen

Die oberste Schicht der Architektur wird von den in Java implementierten Anwendungen gebildet, die ihre Funktionalität untereinander teilen und miteinander interagieren können. Im Folgenden werden die wichtigsten Bestandteile einer typischen Android-Anwendung erläutert.

3.4.1 Intent / Intent-Filter

Ein Intent (*android.content.Intent*), bestehend aus einer geforderten Aktion und einer Datenkomponente, beschreibt den Bedarf für eine gewisse Funktionalität eines Activity. Der Intent-Filter (*android.content.IntentFilter*) spezifiziert für welche Aktionen auf welchen

Daten ein bestimmtes Activity verantwortlich ist und ist in der Manifest-Datei einer Anwendung definiert. Mit Hilfe des Package-Manager, der die Meta-Informationen der Anwendungen verwaltet, kann der Activity-Manager entscheiden, welches Activity für ein gegebenes Intent in Frage kommt.

Grundsätzlich spezifiziert ein Intent kein bestimmtes Activity für die Ausführung, da mehrere Activities passende Filter definiert haben können. Der Benutzer kann in der Konfiguration des Systems Prioritäten für die Auswahl der Activities einstellen. Es gibt die Möglichkeit Intents zu erzeugen, die ein bestimmtes Activity referenzieren. Dies ist allerdings nur für Activities der gleichen Anwendung möglich, da dafür die betreffende Activity-Klasse direkt benutzt (importiert) werden muss.

3.4.2 Activity

Ein Activity wird von der Klasse *android.app.Activity* abgeleitet. In ihm wird die Funktionalität dieser unabhängigen Komponente einer Anwendung implementiert. Im Activity müssen die einzelnen Widgets und das Layout initialisiert werden. Zudem werden die Komponenten mit Aktionen verknüpft, die eintreten, wenn der Benutzer mit den Widgets interagiert.

Ähnlich wie ein Prozess verfügt ein Activity über einen Lebenszyklus, der vom Activity-Manager gesteuert wird. Bei der Implementierung eines Activity müssen Methoden überschrieben werden, die bei einem Eintritt in einen neuen Zustand des Zyklus aufgerufen werden.

In Abbildung 3.4 sind die wichtigsten Stadien eines Activity mit den zugehörigen Methodenaufrufen aufgeführt. Wird ein Activity gestartet, wird zunächst die Methode *onCreate* aufgerufen, die die Komponenten der Oberfläche initialisiert und die Widgets mit bestimmten Aktionen verknüpft. Im Anschluss werden vom Activity-Manager noch weitere Methoden aufgerufen, bevor Benutzereingaben entgegengenommen werden.

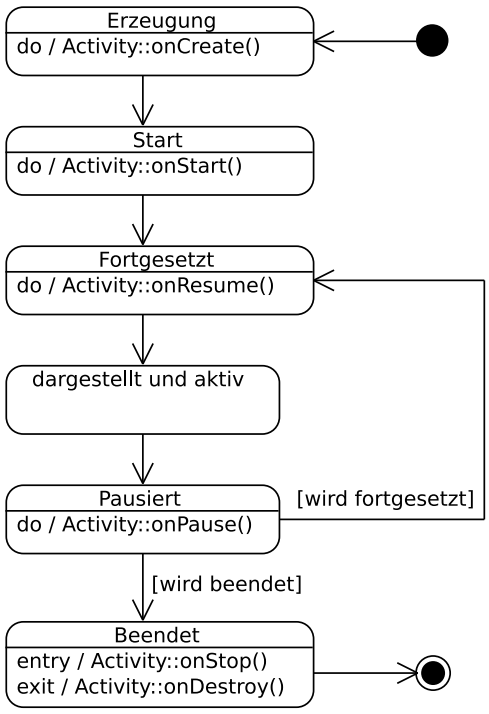

Abbildung 3.4: Activity - Lebenszyklus

Ein Activity kann pausiert werden, indem zum Beispiel ein anderes Activity gestartet wird. Das neu gestartete Activity wird sichtbar und befindet sich im Stapel des Activity-Manager über dem pausierten Activity. Nach

Beendigung des oberen (neu gestarteten) Activity, wird das vorherige fortgesetzt, wobei dessen *onResume*-Methode aufgerufen wird. Um flüchtige Daten zu speichern, wenn das System ein Beenden des Activity veranlasst hat, kann die Methode *onStop* überschrieben werden.

Zur Kommunikation mit anderen Activities dienen die Intents und sogenannte "Ressource Bundles", die mit diversen Daten gefüllt werden können. Verlangt ein Activity vom Activity-Manager den Start eines anderen Activity, wird es pausiert und werden diese Bundles übermittelt. Informationen werden dabei zunächst nur in eine Richtung, also zu dem startenden Activity übertragen. Wird das neue Activity beendet, erhält der Aufrufer die Kontrolle zurück.

Es ist möglich, von einem aufgerufenen Activity Informationen zu erlangen. Dafür muss es mit der Methode *startActivityForResult* gestartet werden. Beim Beenden dieses Activity wird dann die Methode *onActivityResult* aufgerufen. In ihr können zurückgegebene Ressource-Bundles verarbeitet werden. Dieser Aufruf von *onActivityResult* kann als ein asynchroner RPC[1] verstanden werden, da er implizit durch ein Activity erfolgt, das in einem anderen Prozess läuft. Die Daten des Ressource-Bundle werden serialisiert[2], da der für die Interprozesskommunikation zuständige "Binder-Driver" lediglich mit nativen Datentypen arbeiten kann.

3.4.3 Service

Im System kann zu einem Zeitpunkt nur ein Activity aktiv sein. Alle anderen Activities auf dem Stapel des Activity-Manager werden pausiert. Wenn es nötig ist, bestimmte Aktionen auszuführen, die im Hintergrund unabhängig vom aktivierten Activity laufen sollen, müssen Dienste (Services) verwendet werden. Diese Dienste werden von der Klasse *android.app.Service* abgeleitet. Sie besitzen, ähnlich wie die Activities, einen definierten Lebenszyklus, der allerdings unabhängig von anderen Diensten oder Activities ist. Startet ein Activity einen Dienst, muss ein Beenden explizit gewünscht werden, was auch in einem anderen Activity geschehen kann.

Das System verfügt über bestimmte Dienste, die beim Hochfahren automatisch gestartet werden. Beispiele sind die erwähnten Manager (z.B. Activity-Manager). Diese System-Dienste können im Gegensatz zu selbst implementierten Diensten nicht beendet werden, da sie vom System für alle Anwendungen benötigt werden.

Die Dienste laufen in separaten Prozessen. Für die Kommunikation mittels RPC müssen die Daten (Parameter und Resultate) serialisiert werden. Die Umsetzung ist aufwändig

1 RPC Abk. für "remote procedure call" englisch für "Aufruf einer entfernten Prozedur"
2 Serialisierung bedeute, dass der Zustand eines Objektes in Datenströme umgewandelt wird.

und kann deshalb automatisch generiert werden. Dazu muss für einen Dienst eine Remote-Schnittstelle spezifiziert werden. Die Implementierungen dieser Schnittstellen, auch Stubs genannt, werden mit Hilfe des Werkzeugs "aidl" automatisch erzeugt. Das Stub lässt sich nutzen, um Methoden des Dienstes aus anderen Diensten oder Activities heraus direkt aufzurufen.

3.4.4 Identifikationsnummern für Ressourcen und Widgets

Ein wichtiger Bestandteil einer Android Anwendung sind die Ressourcen. Zu den Ressourcen zählen Layout-Dateien, Bilder, Videos oder Texte. Diese sind in einer festgelegten Ordner-Struktur innerhalb des Projektes abgelegt.

Während der Kompilierung einer Anwendung wird im Projekt eine Datei "R.java" automatisch erzeugt. Diese enthält Identifikationsnummern als statische Felder, mit denen die Ressourcen und auch alle in den Layout-Dateien definierten Widgets identifiziert werden. Der Package-Manager verwendet die in der Datei "R.java" definierte Klasse *R* als Adressbuch.

Um einen Zugriff auf eine der Ressourcen oder eine Referenz auf eines der Widgets zu erhalten, werden die statischen Felder innerhalb der Implementierung verwendet. Sie dienen dabei als Parameter für bestimmte Methoden.

3.4.5 Context

Die abstrakte Klasse *android.content.Context* dient dem Zugriff auf globale Informationen der Anwendungs-Umgebung. Sie kann nicht direkt instantiiert werden. Stattdessen wird ihre Implementierung vom Android-System zur Verfügung gestellt und ist über die Klassen *Activity* und *Service* verfügbar.

Ein Context-Objekt ermöglicht den Zugriff auf Ressourcen und Klassen der Anwendung und auch auf bestimmte Ressourcen und Dienste des Systems. Ein solches Objekt ist zum Beispiel notwendig, um Intents zu erzeugen, auf Dateien oder die SQLite-Datenbank zuzugreifen oder Referenzen auf Widgets zu erlangen.

3.4.6 Manifest-Datei

In der Android-Manifest-Datei werden die Activities und Services mit ihren Intent-Filtern definiert. Sie enthält also die vom Package-Manager verwendeten Meta-Informationen der Anwendung. In ihr ist auch definiert, welches Activity bei dem Start der Anwendung zuerst geöffnet werden soll.

Neben verschiedenen Informationen, zum Beispiel dem Namen der Anwendung, werden hier auch Rechte spezifiziert. Benötigt eine Anwendung zum Beispiel den Zugriff auf das

Dateisystem, muss dies in die Manifest-Datei eingetragen werden. Auch für den Zugriff auf spezielle Systemdienste, zum Beispiel auf den Location-Manager, ist ein Eintrag notwendig. Bei der Installation wird die Manifest-Datei ausgewertet. Das System erfragt dann bei dem Benutzer, ob er der Anwendung diese Rechte gewährt.

3.5 Zusammenfassung

Die Android-Plattform ist ein für mobile Geräte angepasstes System. Auch wenn die Anwendungen in Java entwickelt werden können, ist es nicht immer ohne weiteres möglich, Java-Bibliotheken in Android einzusetzen, denn in der Android-Plattform wird nur ein Teil der Standard-Java-API angeboten.

Die Anwendungen sind in das Android-Framework eingebettet, das zwar in Java implementiert ist, allerdings mit den unteren Schichten des Systems eng verbunden ist. Bestimmte Dienste des Android-Framework verwalten die Anwendungen. Ein Zugriff auf die Ressourcen des Systems ist nur mit Hilfe zentraler Dienste und Objekte des Framework möglich. Bei der Gestaltung der Architektur von Android-Anwendungen muss dies beachtet werden.

Daten, die zwischen Activities oder Diensten ausgetauscht werden, müssen serialisiert werden. Während ein Activity mit einem Dienst über RPC direkt kommunizieren kann, ist die Kommunikation zwischen Activities nur mittels der Intents und Ressource-Bundles möglich.

4 Entwicklung von ausführbaren Programmen mit Scala und Android

Nachdem die Programmiersprache Scala und das System Android vorgestellt wurden, soll es in diesem Kapitel um die technischen Details des Aufbaus und der Erstellung von Scala- und Android-Anwendungen gehen. Zunächst wird auf die Kompilierung und Ausführung von Scala-Anwendungen für die Java-Plattform eingegangen. Danach wird gezeigt, wie Android-Anwendungen kompiliert und paketiert werden, um sie installieren zu können. Zum Schluss wird diskutiert, wie es möglich ist Scala-Anwendungen, in ein Android-System zu integrieren.

4.1 Ausführen von Scala Anwendungen in einer Java-Umgebung

Um in Scala entwickelte Anwendungen in einer virtuellen Java-Maschine auszuführen, muss der Scala-Quellcode mit dem Scala-Compiler zunächst in Java-Bytecode übersetzt werden. Daneben gibt es die Möglichkeit, den Scala-Interpreter zu nutzen.

4.1.1 Scala-Compiler

Der Compiler ist in Scala implementiert. Er ist Bestandteil der Datei *scala-compiler.jar*, die herkömmlichen Java-Bytecode beinhaltet. Mit der enthaltenen Klasse *scala.tools.nsc.Main* kann der Compiler somit wie jede andere Java Anwendung gestartet werden. Dazu ist lediglich das Vorhandensein der Scala-Kernbilbiothek *scala-library.jar* im Klassenpfad[1] notwendig.

Die Quellen des Scala-Compilers und der Scala-Bibliotheken sind frei verfügbar. Es ist ohne Probleme möglich, diesen Compiler in eigene Java- oder Scala-Anwendungen zu integrieren.

1 Der Klassenpfad enthält die Dateipfade zu Java-Bibliotheken auf die ausführende Java-Anwendungen Zugriff haben.

Kompilierung von gemischten Scala/Java Projekten

Der Scala-Compiler kann keinen Java-Quellcode kompilieren. Deshalb muss bei gemischten Projekten (mit Scala- und Java-Quellcode) zusätzlich der Java-Compiler verwendet werden. Das kann zur Folge haben, dass nicht in einem Zug kompiliert werden kann.

Vor der Version 2.7.2 des Scala-Compilers war es nicht möglich, rekursive Abhängigkeiten zwischen Java und Scala-Dateien aufzulösen. Eine solche rekursive Abhängigkeit entsteht, wenn zum Beispiel eine in Java implementierte Klasse *A* eine in Scala implementierte Klasse *B* nutzt, aber auch *B* die Klasse *A* verwendet. Das hatte zur Folge, dass man bei gemischten Projekten darauf achten musste, die Abhängigkeiten zwischen Modulen der beiden Sprachen möglichst einseitig zu halten.

Mit dem neuen Compiler ist dies kein Problem mehr. Dieser kann Java-Quellcode nach den notwendigen Abhängigkeiten untersuchen. Auch er kompiliert dabei den Java-Quellcode nicht. Im Prinzip wird dieser nur zum "Linken" verwendet. [Vgl. Sca08a]

Dadurch wird es generell möglich, gemischte Projekte in zwei Schritten zu kompilieren. Zuerst werden mit dem Scala-Compiler alle Scala-Dateien kompiliert, wobei auch die verwendeten Java-Dateien angegeben werden. Danach können mit dem Java-Compiler die Java-Dateien kompiliert werden.

4.1.2 Entsprechungen von Scala-Komponenten in Java

Da die Sprache Java nicht alle Konstrukte beinhaltet, die bei der Programmierung in Scala verwendet werden können, muss der Scala-Compiler solche Scala-Konstrukte in Java-Konstrukte überführen. Dies wird dann interessant, wenn Scala-Module aus Java heraus verwendet werden sollen.

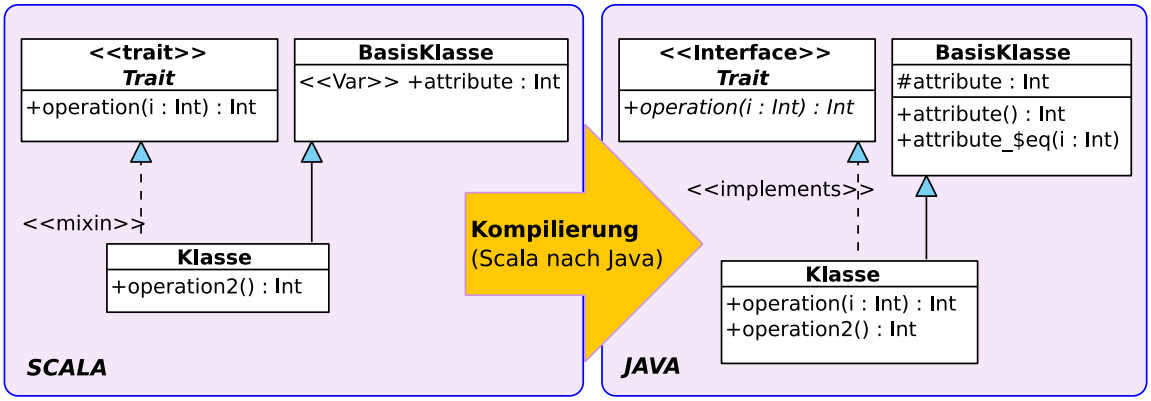

Abbildung 4.1: Traits und Klassenattribute in Scala und Java

Für jedes Trait wird vom Compiler eine Java-Schnittstelle erzeugt. Die Scala-Klassen, die Traits einmischen, werden als "normale" Java-Klassen umgesetzt, denen die Funktionalität

der Traits direkt hinzugefügt wird. Die aus den Traits erzeugten Schnittstellen werden von diesen Klassen dann direkt implementiert. Somit können die Scala-Traits in Java wie Schnittstellen verwendet werden. Die Scala-Klassen, die Traits einmischen, sind in Java voll verwendbar.

In Abbildung 4.1 auf der vorherigen Seite wird gezeigt, wie Klassen mit veränderbaren und veröffentlichten Attributen nach einer Kompilierung in Java abgebildet werden. Dabei sind die impliziten "getter" und "setter" in der Java-Klasse implementiert.

Bei der Kompilierung von Scala Singleton-Objekten wird das Singleton-Entwurfsmuster verwendet [Seite 144 Gam95, Singleton]. Das Singleton-Objekt wird als eine separate Klasse umgesetzt, wobei der Name durch ein "$" erweitert wird. Die Kompagnon-Klassen werden zu herkömmlichen Java-Klassen transformiert.

Die aus den Singleton-Objekten generierten Klassen stehen in keiner Typbeziehung zu ihren Kompagnon-Klassen. Auffällig ist, dass die "getter" und "setter" privater Attribute der Kompagnon-Klasse nach der Kompilierung veröffentlicht werden, wenn das Singleton-Objekt diese verwendet. Die Abbildung 4.2 zeigt, die geschilderte Umsetzung eines Singleton-Objektes und seiner Kompagnon-Klasse in Java.

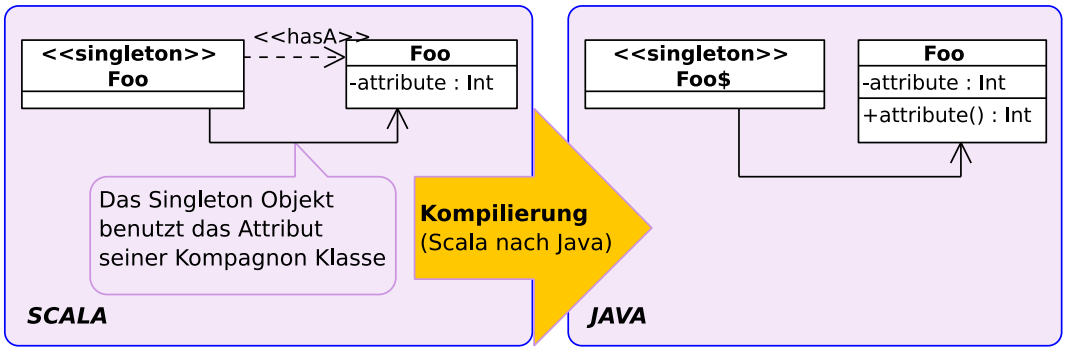

Abbildung 4.2: Singleton in Scala und Java

Es kann hier nicht auf alle Feinheiten der Umsetzung von Scala-Konstrukten in Java eingegangen werden. Erwähnt sei aber noch, dass Operatoren als normale Methoden umgesetzt werden. Dabei erhalten diese Operatoren Namen, die den verwendeten Symbolen entsprechen.

Der Operator "+" wird zu einer Methode *plus*. Der Operator "+-" wird zu einer Methode *plus$minus*. Bei mehreren verwendeten Sonderzeichen wird also ein "$" in den Methodennamen eingefügt. Dies ist auch in Abbildung 4.1 bei dem Setter des Attributes (*attribute*) zu erkennen.

Wenn für einen Operator kein entsprechender Name existiert, zum Beispiel wenn spezielle Unicode-Sonderzeichen verwendet wurden, dann wird für den Methodennamen die Unicode-Nummer mit vorangestelltem "u" verwendet.

Abschließend lässt sich feststellen, dass bei der Kompilierung von Scala-Code einige grundlegende Änderungen vorgenommen werden. Dies sollte man beachten, wenn in Scala entwickelte Bibliotheken auch in Java genutzt werden sollen. Sonst können die geänderten Sichtbarkeiten und Namen von Methoden bei der Verwendung dieser Bibliotheken in Java zu Problemen führen. Einige Komponenten von Scala, wie zum Beispiel Traits, sind in Java nur eingeschränkt verwendbar.

4.1.3 Scala-Interpreter

Scala besitzt einen Interpreter, der Scala-Quellcode zur Laufzeit ausführt. Dieser Interpreter gehört dem gleichen Paket wie der Compiler an. Auch sein Quellcode ist verfügbar. Grundsätzlich kann der Interpreter somit einfach in andere Anwendungen integriert werden.

Der Scala-Interpreter nimmt Scala-Ausdrücke zeilenweise entgegen. Diese werden unverzüglich "just-in-time" (JIT) in Java-Bytecode übersetzt und ausgeführt. Dabei wird nach Eingabe eines Ausdruckes immer das Resultat angezeigt. Zum Beispiel liefert die Eingabe

val i = 0 das Resultat *i: Int = 0*.

Für eine Eingabe ohne Einführung von Bezeichnern und direkte Zuweisung werden temporäre Variable eingeführt. Diese können zwar in den nächsten Eingaben verwendet werden, dürfen aber auch überschrieben werden. Zum Beispiel kann die Ausgabe für

0 + 1 wie folgt aussehen: *res0: Int = 1*.

Im Interpreter können grundsätzlich alle Operationen ausgeführt werden, die auch in Scala-Dateien möglich sind. Lediglich Pakete stellen eine Ausnahme dar. Diese können zwar importiert werden, eine Definition ist allerdings nicht möglich.

Der Interpreter ist ein wirklich sehr gutes Mittel, um kleine Algorithmen oder Operationen auszuprobieren, bevor man sie in einem großen Programm verwendet.

4.2 Entwicklungsprozess von Android-Anwendungen

Mit dem Android-SDK erhält der Entwickler eine Anzahl von Werkzeugen für die unterschiedlichen Aufgaben bei der Entwicklung von Android-Anwendungen. In diesem Abschnitt wird auf die wichtigsten kurz eingegangen, bevor der Ablauf der Kompilierung und Erzeugung einer Android-Anwendung näher erläutert wird.

4.2.1 Entwicklungswerkzeuge

Die Entwicklungswerkzeuge gehören zum Android SDK, das von der Webseite des Android Projektes bezogen werden kann [And]. In diesem Abschnitt werden nur jene aufgeführt, die für die Entwicklung einer Android-Anwendung unentbehrlich sind.

Android Interface Definition Language (aidl)

Wie bereits in Abschnitt 3.4.3 auf Seite 38 erwähnt, werden Remote-Schnittstellen von Diensten in aidl-Dateien spezifiziert. Das Werkzeug "aidl" erzeugt die implementierten Schnittstellen (Stubs), die dann in Java-Quellcode-Dateien vorliegen.

Dalvik-Bytecode Converter (dx)

Mit dem in Java implementierten Kommandozeilentool "dx" wird Java-Bytecode in Dalvik-Bytecode konvertiert. Der resultierende Dalvik-Bytecode liegt dann in einer einzigen dex-Datei vor. Eine JIT-Kompilierung ist nicht möglich, da das Konvertieren sehr CPU- und speicherlastig ist. Auch aus diesem Grund kann das Konvertieren innerhalb eines herkömmlichen Android-Systems selbst nicht erfolgen. Die Zielgeräte der Android-Plattform besitzen in der Regel nur sehr wenig Speicher.

Android Asset Packaging Tool (aapt)

Mit dem Werkzeug "aapt" können Ressourcen, wie zum Beispiel Bilder oder Layout-XML-Dateien, in für Android-Anwendungen verwendbare Form gebracht werden. Dieses Werkzeug erzeugt auch die in Abschnitt 3.4.4 auf Seite 39 vorgestellte Datei "R.java". Zudem dient es der Paketierung der gesamten Anwendung, wobei es die dex-Datei und alle Ressourcen in einer apk-Datei zusammenfasst. Diese apk-Dateien sind mit den jar-Dateien herkömmlicher Java-Pakete vergleichbar.

Im Build-Zyklus nimmt das Tool "aapt" somit verschiedene Rollen ein. Es wird sowohl zum Kompilieren als auch zum Paketieren einer Anwendung benötigt.

Emulator

Der Android-Emulator basiert auf der Software QEMU [Qem08]. Er emuliert ein Handy, das auf einer ARM-Prozessorarchitektur basiert. Der Emulator umfasst die gesamte vorgestellte Android-Architektur vom Kernel bis zu den Standard Anwendungen, inklusive der grafische Darstellung.

Im Emulator können Android-Anwendungen bei der Entwicklung installiert und getestet werden. Der Binärcode der virtuellen ARM-Maschine wird zur Laufzeit in den Code für die

Prozessor-Architektur des Gastsystems übersetzt. Einige Komponenten werden allerdings nicht oder nicht vollständig unterstützt. Dazu gehören zum Beispiel USB-Ports oder Bluetooth.

Android Debug Bridge (adb)

Das Werkzeug “adb” hat viele Funktionen. Zum einem dient es dem Absetzen von Kommandos in der Shell des Emulators. Zum anderen kann es zum Debuggen verwendet werden. Es ermöglicht das Anzeigen der Logdaten und das Kopieren von Dateien zwischen dem Gastsystem und dem Emulator. Es wird auch verwendet, um eine Anwendung in Form einer apk-Datei im Emulator zu installieren und Ports des Gastsystems an den Emulator weiterzuleiten.

4.2.2 Prozess der Kompilierung und Paketierung

Im vorigen Abschnitt wurden die für die Kompilierung und Paketierung mindestens notwendigen Werkzeuge vorgestellt. Jetzt wird erläutert, wie diese Werkzeuge zusammenspielen, um eine Anwendung für das Android-System zu erzeugen.

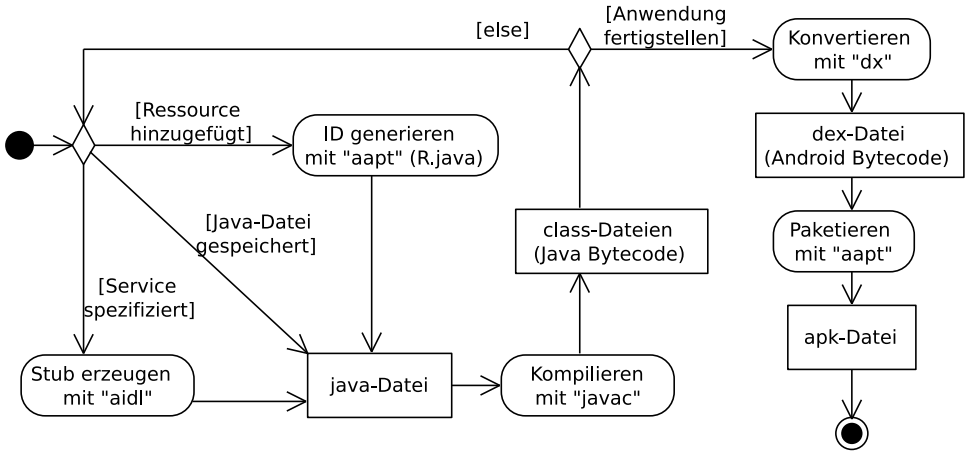

Abbildung 4.3: Android-Entwicklung mit JIT-Kompilierung

In Abbildung 4.3 ist ein Vorgehen dargestellt, bei dem erzeugte Java-Dateien unverzüglich (JIT) kompiliert werden. Dadurch können mögliche Kompilierfehler im Programm sofort entdeckt werden.

Zunächst ist es notwendig, ein Projekt in einer festgelegten Struktur aufzubauen. Werden dem Projekt neue Ressourcen oder Layout-Dateien hinzugefügt, muss mit dem Werkzeug “aapt” eine neue Datei “R.java” erzeugt werden, die die Identifikationsnummern enthält. Die Verwendung des Werkzeugs “aidl” wird notwendig, sobald ein neuer Service spezifiziert wurde.

Der Java Quellcode, inklusive der mit “aidl” erzeugten Stubs und der mit “aapt” erzeugten Datei “R.java” müssen mit dem Java-Compiler (javac) in Java-Bytecode kompiliert werden. Im Anschluss ist es möglich, mit dem Werkzeug “dx” diesen Java-Bytecode inklusive verwendeter Bibliotheken (jar-Dateien) in Dalvik-Bytecode zu übersetzen.

Schließlich wird mit Hilfe von “aapt” ein Android-Paket für die Anwendung erzeugt, das alle Ressourcen, die Manifest-Datei und die dex-Datei enthält.

4.3 Integration von Scala in den Android-Entwicklungsprozess

Es wurde bereits darauf eingegangen, dass Java-Bibliotheken im Android-System verwendet werden können. Auch die Scala-API ist zunächst eine Java-Bibliothek. Da der Scala-Quellcode mit Hilfe des Scala-Compilers in Java-Bytecode kompiliert wird, steht einem Einsatz im Android-System prinzipiell nichts entgegenstehen.

Es ergeben sich einige Probleme und Einschränkungen beim Einsatz von Scala, auf die in diesem Abschnitt eingegangen wird. Verschiedene Lösungsmöglichkeiten werden diskutiert. Ziel ist es zu zeigen, wie Android-Anwendungen mit Scala realisiert werden können. Zudem sollen die Grenzen des Einsatzes von Scala innerhalb des Android-Systems herausgearbeitet werden.

4.3.1 Einsatz des Scala-Compilers

Auf den Ablauf bei der Kompilierung und Paketierung einer Android-Anwendung ist bereits eingegangen worden. Es spricht nichts dagegen, den Scala-Compiler in diesen Ablauf zu integrieren. Er muss dabei vor dem Java-Compiler zum Einsatz kommen (Abbildung 4.3 auf der vorherigen Seite).

Den Java-Compiler komplett durch den Scala-Compiler zu ersetzen ist derzeit nicht möglich, da das Werkzeug “aidl” nur Java-Quellcode erzeugt und “aapt” die Datei “R.java” liefert. Ein Anwendung zu entwickeln, die stattdessen Scala-Quellcode erzeugt, ist aber durchaus vorstellbar, worauf hier nicht weiter eingegangen wird.

Für ein erfolgreiches Konvertieren von Java-Bytecode zu Dalvik-Bytecode ist valider Java-Bytecode notwendig, der vom Scala-Compiler erwartet wird.

4.3.2 Nicht erfüllte Abhängigkeiten der Scala-API

Da in Scala Java-Klassen uneingeschränkt verwendet werden können, gilt dies auch für die Klassen des Android-Framework. Es können also Schnittstellen implementiert, Klassen beerbt und Methoden überschrieben werden, um zum Beispiel die Dienste und Activities

zu implementieren. Auch das Verwenden der anderen Bestandteile der Android-API stellt kein Problem dar.

Problematisch ist allerdings, dass die Android-Laufzeitumgebung keine vollständige Java-API besitzt und die Scala-API von den Klassen der Java-Kern-Bibliothek abhängt. Es gibt genau vier Klassen, die von der Android-Java-API (1.0) nicht angeboten, aber von der Scala Bibliothek (2.7.2) verwendet werden:

1. java.rmi.RemoteException
2. java.beans.MethodDescriptor
3. java.beans.PropertyDescriptor
4. java.beans.SimpleBeanInfo

Untersucht man den Bytecode der Scala-Bibliothek[1], so kann man sehen, dass die drei Klassen des *java.beans* Paketes nur von der Klasse *scala.reflect.BeanInfo* verwendet werden. Allerdings verwendet keine weitere Komponente der Scala-Bibliothek die Klasse *BeanInfo*. Sie kann somit aus der Bibliothek entfernt werden.

Dagegen wird die Klasse *java.rmi.RemoteException* sehr häufig verwendet. Der Scala-Quellcode verwendet im Trait *scala.ScalaObject* eine Annotation *@remote*. Das Trait *ScalaObject* wird von allen Klassen implizit eingemischt, die *scala.AnyRef* beerben und keine direkte Entsprechung in der Java-API besitzen ([Sca08b, Vgl.]). Mit der *@remote*-Annotation wird spezifiziert, dass eine Methode über einen entfernten Zugriff aufgerufen werden kann. Es kann somit zu einer *RemoteException* kommen. Der Scala-Compiler fügt allen Methoden mit dieser Annotation eine entsprechende "Throws-Klausel" hinzu. Nun nutzt Android allerdings nicht die Java-RMI-API.

Der genaue Grund für die Verwendung der Annotation "remote" im Trait *ScalaObject* ist dem Autor nicht bekannt. Laut der Scala-Dokumentation wird sie für Optimierungen beim Pattern-Matching verwendet. Mit einer Anfrage bei den Entwicklern von Scala wurde in Erfahrung gebracht, dass diese Annotation prinzipiell unnötig ist und in zukünftigen Versionen von Scala (ab 2.8) aus dem Trait *ScalaObject* entfernt wird.

Da der Quellcode von Scala verfügbar ist, lässt sich das Trait *ScalaObject* anpassen. Das Entfernen der Annotation führt dazu, dass die Klasse *RemoteException* von keiner Komponente der Scala-Bibliothek mehr benötigt wird. Tests führten bisher zu keinen Problemen.

Wie gezeigt wurde, konnte durch punktuelles Anpassen der Scala-Bibliothek an zwei Stellen eine Kompatibilität zur Android-API hergestellt werden.[2]

1 Es wurde das Werkzeug "jarjar"[Jar] verwendet.

2 Für die aktuelle Scala-Version ist diese Änderung notwendig, da momentan keine für Android angepasste Bibliothek angeboten wird.

4.3.3 Unterschiede der Android-API und der Java-API

Ein weiteres Problem beim Einsatz von Scala sind die unterschiedlichen Implementierungen der Android-API und der Java-API. Unter Umständen wird eine unterschiedliche Verwendung der Schnittstellen von der jeweiligen API verlangt. Da komplexere Prozesse in einigen Klassen der Scala-API verborgen sind, kann der Einsatz bestimmter Scala-Module zu Fehlern führen.

Ein Beispiel ist das Einlesen von XML-Dateien. Das kann in Scala sehr komfortabel mit der Funktion *load(file): Node* des Singleton-Objektes *scala.xml.XML* vorgenommen werden. Dabei wird die SAX-API (*org.xml.sax*) verwendet. Sowohl in der Android-API als auch in der Java-API (Version ab 1.5.0) werden die gleichen Schnittstellen für diese SAX-API angeboten (Vgl. [And08] [Jav]). Beim Lesen von XML-Dateien wird die Implementierung der Schnittstelle *org.xml.sax.XMLReader* verwendet. Die verwendete Implementierung im Android-System ist die Klasse:

org.apache.harmony.xml.ExpatReader.

Die verwendete Klasse bei einem Ausführen in der Standard Java-Laufzeitumgebung ist: *com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl*.

Zusammenfassend ist festzustellen, dass die Verwendung der Scala-API in Android zu einem Fehler führt, der durch eine andersartige Implementierung der verwendeten Schnittstelle verursacht wird. Um solche Probleme zu lösen, muss relativ aufwändig nach der Ursache gesucht werden.

4.3.4 Einsatz des Scala-Interpreter im Android-System

Da der Scala-Interpreter in Java implementiert ist, wäre es denkbar, ihn im Android-System zu nutzen.

Der Interpreter kompiliert Scala-Code zur Laufzeit in Java-Bytecode. Um diesen Bytecode im Android-System auszuführen, ist aber zunächst ein Konvertieren zu Dalvik-Bytecode notwendig. Der geringe Speicher der aktuellen Geräte verhindert den Einsatz des Scala-Interpreter im Android-System. Erst wenn für die Dalvik-Bytecode-Konvertierung weniger Speicher- und CPU-lastige Realisierungen existieren, wäre der Einsatz denkbar.

Es sei darauf hingewiesen, dass es Java-basierte Skriptsprachen gibt, die nicht zur Laufzeit in Java-Bytecode kompiliert werden. Diese nutzen die Java-Reflection-API. Eine dieser Sprachen, für die ein Interpreter in das Android-System integriert wurde, ist "Beanshell" [Mur08].

4.3.5 Performance-Tests im Android-System

Um die Performance von Scala-Anwendungen innerhalb des Android-Systems zu prüfen, wurden Android-Anwendungen sowohl in Scala als auch in Java implementiert und im Emulator getestet.

In diese Anwendungen wurden Algorithmen des Projektes "The Computer Language Benchmarks Game" (CLBG) integriert [Perc]. Dieses Projekt vergleicht die Laufzeiten und den Speicherverbrauch von Test-Programmen, die in verschiedenen Programmiersprachen entwickelt sind. Unter diesen Sprachen befindet sich auch Java und Scala. Allerdings werden dort keine Tests im Android-System ausgeführt, sondern lediglich die herkömmliche Java-VM verwendet.

Zwei der verwendeten Scala-Tests ([Perd], [Perb]) mussten vom Autor für Scala 2.7.2 angepasst werden. Die neuen Versionen sind auch an das Projekt CLBG übermittelt worden, so dass sie dort jetzt wieder lauffähig sind.

In Tabelle 4.1 sind die Resultate der in Android ausgeführten Scala- und Java-Tests aufgeführt. In der ersten Spalte steht der Name des Tests, wie er im CLBG-Projekt verwendet wird. In der zweiten Spalte ist der Eingabeparameter für den Test angegeben. In Spalte drei und vier finden sich die Zeiten in Millisekunden, die die Tests in der Implementierung für die jeweiligen Sprache benötigten. Spalte fünf gibt an, wieviel mal weniger Zeit der in Java implementierte Test im Vergleich zum Scala-Test benötigte.

Tabelle 4.1: Android Performance Tests: Java und Scala

Testname	Parameter	Java	Scala	Java x-mal schneller
fannkuch	9	9262ms	14532ms	1,6
nsieve	8	12990ms	15380ms	1,2
spectral-norm	200	10576ms	11125ms	1,1
n-body	20000	8291ms	14547ms	1,8
recursive	3	6876ms	8641ms	1,3

Beim direkten Vergleich der Laufzeiten der Scala und Java-Implementierungen, sind die Testergebnisse mit denen des CLBG-Projektes, dessen Tests in der Java-VM laufen, vergleichbar. Eine Ausnahme sind jene Scala-Tests, die - wie den Seiten des CLBG-Projektes zu entnehmen ist - wesentlich mehr Speicher benötigen als die Java-Tests [Vgl. Pera]. Das Verhältnis der Laufzeiten beider Tests im Android-System hat sich bei speicherintensiven Scala-Tests zu Gunsten von Java verschoben. Die Ursache ist vermutlich in Auslagerungsprozessen zu suchen (Swapping).

Die Tests wurden im Emulator ausgeführt. Es ist davon auszugehen, dass in den Android-Geräten andere Resultate erzielt werden können. Es lässt sich vorerst der Schluss

ziehen, dass mit Scala ausreichend performante Android-Anwendungen entwickelt werden können, denn die Laufzeiten der Scala-Tests unterscheiden sich nicht stark von denen der Java-Tests.

4.4 Zusammenfassung

Mit Scala können grundsätzlich Anwendungen für die Android-Plattform entwickelt werden. Dazu muss die Scala-API angepasst werden. Der Scala-Compiler lässt sich ohne Probleme in den Entwicklungsprozess von Android-Anwendungen integrieren. Die volle Funktionalität von Scala ist aber nicht uneingeschränkt nutzbar. Der Scala-Interpreter kann im Android-System wegen der aufwändigen Erzeugung von Dalvik-Bytecode nicht verwendet werden. In Folge der Unterschiede der Android- und Standard-Java-API können Teile der Scala-API in der Android-Laufzeitumgebung zu Problemen führen. Diese lassen sich allerdings beheben, da die Java-Schnittstellen der Android-API in Scala direkt verwendet werden können.

5 Konzept der Beispielanwendung

Dieses Kapitel befasst sich mit den fachlichen Anforderungen der im Rahmen der Diplomarbeit entwickelten Anwendung TribOSMap. Es wird die Idee für diese Anwendung erläutert, Anforderungen werden definiert und Anwendungsfälle sowie Fachklassen spezifiziert.

Mit der Entwicklung dieser Anwendung sollen die Möglichkeiten und Probleme des Einsatzes von Scala bei der Android-Entwicklung untersucht werden. Deshalb ist eine praxisbezogene und hinreichend umfangreiche Anwendung entworfen worden, die den Einsatz unterschiedlicher Technologien und sprachlicher Mittel rechtfertigte.

Das fachliche Umfeld wird in diesem Kapitel nicht näher erläutert. Dazu sei auf den Anhang A auf Seite 135 verwiesen. Dort werden einige fachliche Begriffe der Geodäsie vorgestellt. Auch nähere Informationen zu dem OpenStreetMap-Projekt [Ope] sind dort zu finden, wie zum Beispiel Erläuterungen zu Abläufen, Dateiformaten und Werkzeugen.

5.1 Idee für die Anwendung TribOSMap

Das Wort “tribos” ist altgriechisch und bedeutet so viel wie Pfad oder Weg. “OSM” ist die herkömmliche Abkürzung für OpenStreetMap. Damit verdeutlicht der Name den Hauptzweck der Anwendung: Daten aufgezeichneter Wege für die Erzeugung der OpenStreetMap-Weltkarte zu nutzen.

Viele mobile Geräte verfügen über eingebaute GPS-Empfänger. Auch in den meisten Android-Geräten, die in der nächsten Zeit auf den Markt kommen, wird dies der Fall sein. Damit lassen sich diese Geräte nutzen, um mit den aufgezeichneten geografischen Daten die umfassende digitale Weltkarte des Projektes OpenStreetMap zu erweitern.

Abbildung 5.1: OSM-Logo [Ope]

Die Idee war, eine Anwendung zu entwickeln, die dem Benutzer beim Navigieren in offenem Gelände hilft. Dazu können Landkarten betrachtet werden, die

aus dem Internet oder aus Dateien geladen werden. Diese Funktionalität soll die Attraktivität der Anwendung steigern, aber der Hauptzweck ist das Sammeln der geografischen Daten, wobei zur Sicherung Dateiformate verwendet werden, die es ermöglichen, gewonnene Daten direkt für OpenStreetMap zu verwenden.

5.2 Anforderungen

5.2.1 Zielbestimmung

Die Anwendung ist so konzipiert, dass sie auf Android-Geräten lauffähig ist. Sie soll das OpenStreetMap-Projekt mit neuen Daten versorgen, dem Benutzer beim Zeichnen der OpenStreetMap-Karte helfen und eine Positionsbestimmung mit digitalen Landkarten ermöglichen.

Die Anwendung wird die Landkarte des OpenStreetMap-Projektes und auch Rastergrafik-Karten, die der Nutzer selbst digitalisiert hat, darstellen können. Besonderer Schwerpunkt wird auf das einfache Aufzeichnen und Exportieren von zurückgelegten Wegen (Traces[1]) gelegt.

5.2.2 Muss-Kriterien

Im Folgenden werden die Kriterien aufgeführt, die von der Anwendung zwingend unterstützt werden müssen.

Traces: Es sollen Traces aufgenommen werden können, die den zurückgelegten Weg eines Benutzers wiedergeben. Ein Benutzer kann zu einem beliebigen Zeitpunkt die Aufzeichnung von Traces veranlassen. In diesem werden in bestimmten zeitlichen Intervallen die geografischen Koordinaten der aktuellen Position des Benutzers gespeichert. Während der Aufzeichnung wird der Trace unverzüglich gesichert. Außer der geordneten Liste mit Positionsangaben enthält dieser auch Zusatzinformationen und besitzt einen Namen. Diese zusätzlichen Daten muss der Benutzer angeben und auch nachträglich verändern können.

Export Trace: Ein gesicherter Trace muss vom Handy auf ein externes Gerät exportiert werden können. Diese Datei soll im GPX-Format (Anhang A.2.2 auf Seite 138) vorliegen. Dadurch ist die Kompatibilität zum OpenStreetMap-Projekt und dessen Karteneditoren gewährleistet.

1 traces: englisch für Spuren

Markierungen: Es soll möglich sein, bestimmte Orte in der Anwendung fest zu halten, die in der Kartenansicht grafisch dargestellt werden. Diese Markierungen haben einen Namen, eine Position und optionale Zusatzinformationen, zu denen auch ein aufgenommenes Foto gehören kann. Wird eine Markierung während einer Spuraufzeichnung erzeugt, soll sie dem aktiven Trace zugewiesen werden, so dass die Markierung beim Exportieren des Trace in die GPX-Datei integriert wird. Markierungen müssen nachträglich bearbeitet und auch gelöscht werden können. Die Positionsangabe soll manuell editiert werden, wobei geografische oder UTM-Koordinaten (Anhang A.1.4 auf Seite 137) genutzt werden können.

Kartendarstellung: Eine digitale Landkarte soll visuell dargestellt werden können. Dabei kann der sichtbare Bereich der Karte vergrößert, verkleinert und bewegt werden. Verändert sich dieser Bereich, muss dafür gesorgt werden, dass die notwendigen Teile möglichst schnell nachgeladen werden. In der Kartendarstellung sollen gespeicherte Markierungen und die aktuelle Position sichtbar gemacht werden.

Positionsermittlung: Über einen mit dem Handy verbundenen oder im Handy integrierten GPS-Empfänger kann der Benutzer seine aktuelle Position ermitteln. Diese Position soll sowohl auf der Karte grafisch dargestellt als auch als Text in Form von geografischen Koordinaten angezeigt werden. Es sollen sowohl geografische Koordinaten als auch UTM-Koordinaten für das Anzeigen unterstützt werden.

Darstellung OpenStreetMap-Karte Online: Die aktuelle OpenStreetMap-Landkarte wird als Rasterkarte aus dem Internet geladen und grafisch dargestellt. Bereits geladene Kartenteile sollen in einem Cache abgelegt werden. Wenn der sichtbare Kartenbereich verändert wird, sollen die notwendigen Teile möglichst unverzüglich aus dem Cache oder, wenn notwendig, aus dem Internet nachgeladen werden.

Darstellung Rasterbild-Landkarten Offline: Vom Benutzer selbst erzeugte georeferenzierte Bilder sollen als digitale Landkarten in die Anwendung importiert und danach dargestellt werden können. Für die Darstellung dieser importierten Karten wird also keine Internetverbindung benötigt. Zunächst sollen Karten unterstützt werden, die eine UTM-Projektion (Anhang A.1.3 auf Seite 136) und das WGS84-Datum (Anhang A.1.2 auf Seite 136) verwenden.

5.2.3 Kann-Kriterien

Im Folgenden werden die Kriterien aufgeführt, die von der Anwendung nicht zwingend unterstützt werden müssen. Eine Umsetzung dieser Funktionalität wäre allerdings wünschenswert und soll beim Entwurf der Software eingeplant werden.

Navigation: Es können Abstände und Richtungen zwischen gewählten Markierungen bestimmt werden. Dies kann komfortabel über die Kartenansicht erfolgen.

Trace Darstellung: Ein in der Aufzeichnung befindlicher Trace wird auf der Kartenansicht als eine Menge von Punkten, die durch Linien verbunden sind, dargestellt. Dabei muss eine Lösung gefunden werden, die möglichst wenig Speicher verbraucht.

Wegpunktnavigation: Eine Navigationsunterstützung mit Hilfe von Wegpunkten soll dem Benutzer bei der Wanderung immer ein aktuelles Zwischenziel anzeigen. Diese Wegpunktlisten können mit speziellen Markierungen erzeugt werden. Die Umschaltung von einem Teilziel zum Nächsten erfolgt automatisch, wenn ein bestimmter Abstand zum aktuellen Teilziel unterschritten wurde.

Darstellung Konfigurieren: Die Konfiguration der Darstellung von Markierungen und Traces in Farbe und Form soll ermöglicht werden. Ebenso soll es möglich sein, die verwendeten Maßsysteme zur Angabe der Entfernungen umzuschalten.

Tonaufnahmen: Der Benutzer kann Tonaufnahmen vornehmen, die sowohl an Traces als auch an Markierungen gebunden werden können.

Kartenprojektionen: Es sollen viele Kartenprojektionen unterstützt werden, um damit eine möglichst große Menge unterschiedlicher Landkarten in der Anwendung nutzbar zu machen.

5.2.4 Abgrenzungskriterien

Um Missverständnisse zu vermeiden, werden hier Funktionalitäten aufgezählt, die die Anwendung nicht abdecken wird.

Routenplanung: Bekannte Routenplaner arbeiten mit Vektor-Karten. In diesen sind Wege so erfasst, dass es möglich ist, dem Benutzer genaue Anweisungen zu geben, auf welchen Straßen er sich wie bewegen muss, um ein angegebenes Ziel zu erreichen. Die hier geplante Anwendung TribOSMap unterstützt dagegen nur Rastergrafik-Karten, mit denen eine solche Routenplanung nicht möglich ist.

OSM-Vektorkarten zeichnen: Das Editieren der OSM-Vektorkarten ist ein komplexer Prozess. Dies auf einem mobilen Gerät zu ermöglichen ist, ergonomisch betrachtet, unmöglich. Ohne Maus und großen Bildschirm würde der Benutzer sofort die Lust daran verlieren.

5.3 Anwendungsfälle

In diesem Abschnitt wird auf die wichtigsten Anwendungsfälle eingegangen, die im Zusammenhang mit der TribOSMap stehen, auch die, die außerhalb der zu entwickelnden Software liegen, um die Integration in das OpenStreetMap-Projekt zu verdeutlichen.

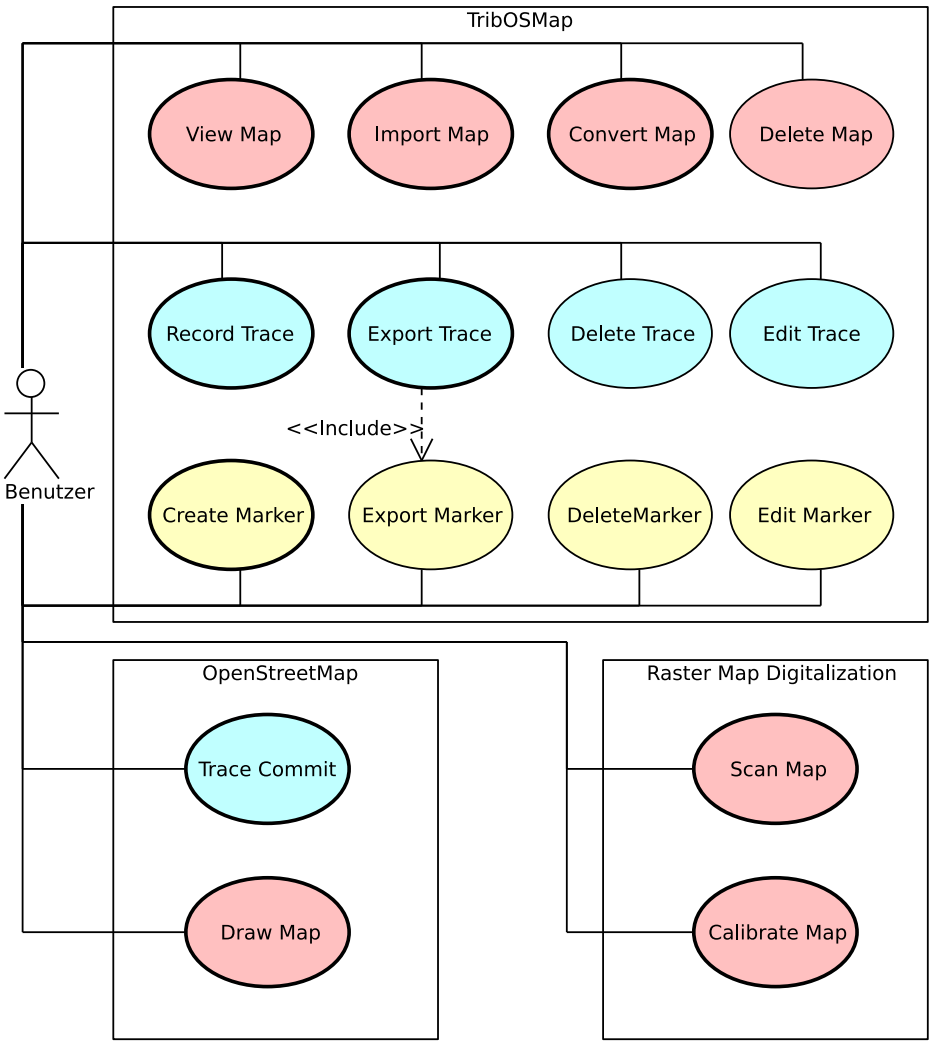

Abbildung 5.2: Anwendungsfälle - Übersichtsdiagramm

5.3.1 OpenStreetMap

Es werden zwei Anwendungsfälle unterschieden, bei denen der Anwender aktiv die Daten des OpenStreetMap-Projektes verändert. Die im Folgenden beschriebene Funktionalität wird nicht von der entwickelten Anwendung umgesetzt. Es werden allerdings Daten verwendet, die mit der Anwendung TribOSMap gewonnen werden sollen.

Trace Commit (Einen Trace übermitteln): Aufgezeichnete Traces in Form von GPX-Dateien können an den OpenStreetMap-Server übermittelt werden. Sie sind dann auch für

andere OSM-Aktivisten nutzbar. Für die Übermittlung benötigt der Anwender ein Konto beim OpenStreetMap-Projekt. Auf einer Weboberfläche kann er seine GPX-Datei auf dem lokalen Rechner auswählen, mit weiteren Text-Informationen versehen und im Anschluss übertragen.

Draw Map (Die OpenStreetMap-Karte zeichnen): Mit Hilfe der OSM-Karteneditoren kann der Anwender Traces importieren. Es ist möglich, einen Trace aus einer lokalen GPX-Datei zu laden. Auch das Laden von Traces vom OpenStreetMap Server ist möglich. Diese Karteneditoren können neben den geografischen Positionen auch zusätzliche Informationen aus GPX-Dateien verarbeiten, wie geografisch positionierte Bilder, Töne und Texte. Der Anwender kann neue Objekte in die Karte einzeichnen und mit notwendigen sowie optionalen Informationen versehen. Am Ende des Zeichnens übermittelt er die neuen Vektordaten sowie die Daten der lokal importierten GPX-Dateien an den OSM-Server.

5.3.2 Rasterkarten Digitalisierung

TribOSMap kann nicht nur die OSM-Rasterkarte sondern auch andere digitale Rasterkarten darstellen. Um in Papierform vorhandene Landkarten nutzen zu können, müssen diese zunächst digitalisiert sowie georeferenziert werden. Dafür wird spezielle Software benötigt. Die im Folgenden genannten Anwendungsfälle sind nicht mit der Anwendung TribOSMap abgedeckt. Allerdings werden die erstellten georeferenzierten Landkarten von TribOSMap verwendet.

Scan Map (Eine Landkarte digitalisieren): Der Benutzer digitalisiert eine Landkarte, indem er sie zum Beispiel einscannt oder abfotografiert. Das muss möglichst sorgfältig geschehen, um Verzerrungen weitestgehend zu vermeiden. Das entstandene Bild ist oft sehr groß und sollte dann in komprimierter Form, zum Beispiel als JPEG, archiviert werden.

Calibrate Map (Eine digital vorhandene Landkarte kalibrieren): Ein digitales Bild der Landkarte muss für die Nutzung in Programmen wie TribOSMap georeferenziert (kalibriert) werden, so dass für jeden Pixel oder Subpixel eine geografische Position berechnet werden kann. Dazu müssen einigen Pixeln geografische Positionen zugewiesen, sowie die verwendete Kartenprojektion und das geodätische Datum der Karte angegeben werden. Um die Informationen für die Georeferenzierung in TribOSMap zu verwenden, wurde das TOSM-XML-Formates entwickelt, auf das am Ende dieses Kapitels in Abschnitt 5.5.2 auf Seite 64 eingegangen wird.

5.3.3 TribOSMap

Die Anwendungsfälle, die von der Funktionalität der Anwendung TribOSMap abgedeckt werden sollen, können in drei Rubriken aufgeteilt werden, die sich durch die Arbeit mit unterschiedlichen Objekten auszeichnen: Karte, Markierungen oder Traces.

Anwendungsfälle im Zusammenhang mit Landkarten (Maps)

Convert Map (Eine Landkarte konvertieren): Digitalisierte Karten in Form von Rasterbildern müssen für das Handy aufbereitet werden. Dazu werden die Bilder zunächst in viele Teilbilder zerlegt, die anschließend in einer großen Datei wieder zusammengefügt werden. Das ermöglicht das Laden von Teilen des Gesamtbildes. Für die unterschiedlichen Zoomstufen werden separate Teilbilder erzeugt, die jeweils eine bestimmte Auflösung besitzen. Ergebnis ist eine Binärdatei, auf deren Format TOSM am Ende dieses Kapitels in Abschnitt 5.5.1 auf Seite 64 eingegangen wird.

Import Map (Eine Landkarte importieren): Nach dem Konvertieren in ein für das Handy notwendige Format und der Georeferenzierung kann die Landkarte importiert werden. Dabei werden die Kalibrierungsinformationen sowie die Binärdatei mit den Bildinformationen im Handy abgelegt. Darüber hinaus kann die Karte mit einem Namen versehen werden. Wichtige Informationen zu der importierten Karte werden dem Nutzer angezeigt. Dazu gehören der abgedeckte geografische Bereich und die Auflösung der Karte (Meter pro Pixel).

View Map (Eine Landkarte betrachten): Eine importierte Landkarte bzw. die OpenStreetMap-Karte kann vom Benutzer im Handy betrachtet werden. Dabei ist es möglich, diese Karte zu verschieben und zu verkleinern bzw. zu vergrößern. Auf der Karte wird der aktuelle Standort sowie die Lage vorgenommener Markierungen angezeigt.

Delete Map (Eine Landkarte löschen): Um Speicherplatz freizugeben, kann der Anwender eine Karte aus dem Handy entfernen.

Anwendungsfälle im Zusammenhang mit Traces

Record Trace (Einen Trace aufzeichnen): Mit TribOSMap können neue Traces aufgezeichnet und archiviert werden. Dazu muss zunächst ein GPS-Signal im Gerät vorliegen. Nach dem Start der Aufzeichnung werden automatisch neue Positionen zu einem Trace hinzugefügt, die im Handy dauerhaft abgelegt werden. Während der Aufzeichnung werden die aktuell zurückgelegte Strecke sowie die Anzahl der aufgezeichneten Punkte als Text ausgegeben.

Export Trace (Einen Trace exportieren): Ein aufgezeichneter Trace kann vom Anwender im GPX-Format exportiert werden. Dazu muss er den Ort für die Speicherung wählen können. Diese GPX-Datei liegt dann im Dateisystem des Handys vor und steht damit für weitere Zwecke zur Verfügung.

Edit Trace (Einen Trace editieren): Bereits erzeugte Traces können mit einem anderen Namen und zusätzlichen Text-Informationen versehen werden.

Delete Trace (Einen Trace löschen): Traces können auf Wunsch vom System entfernt werden.

Anwendungsfälle im Zusammenhang mit Markierungen (Marker)

Create Marker (Einen Marker erzeugen): Der Anwender kann bestimmte geografische Markierungen erzeugen. Diese sind - wie die Punkte von Traces - mit einer Ortsangabe versehen. Zusätzlich können weitere Informationen - ein Photo, ein Name oder längerer Text - zu der Markierung hinzugefügt werden. Alle erzeugten Markierungen können optional auf der Landkarte angezeigt werden.

Export Marker (Einen Marker exportieren): Eine Markierung, die vom Anwender während der Aufzeichnung eines Trace erzeugt wurde, wird beim Export des Trace mit in die GPX-Datei geschrieben.

Edit Marker (Einen Marker editieren): Der Name, das Photo und auch die zusätzlichen Informationen einer Markierung können vom Anwender verändert werden. Bei der Positionsangabe ist es möglich, geografische oder UTM-Koordinaten zu verwenden.

Delete Marker (Einen Marker löschen): Markierungen können auf Wunsch vom System entfernt werden.

5.3.4 Arbeitsabläufe

Die einzelnen Anwendungsfälle sind in bestimmte Arbeitsabläufe integriert, von denen zwei hier dargestellt werden. Es wird gezeigt, wie TribOSMap mit dem OpenStreetMap-Projekt interagiert und wie digitalisierte Karten verwendet werden können.

Editieren der OpenStreetMap-Karte mit gesammelten Geodaten

Um gesammelte Geodaten für OpenStreetMap zur Verfügung zu stellen, muss der Nutzer einen Trace zunächst aufzeichnen. Dabei werden seine Positionsänderungen gespeichert sofern er sich bewegt. Während der Aufzeichnung erzeugte Marker werden dem in der Aufzeichnung befindlichen Trace hinzugefügt. Bei jedem neuen Marker oder dem Abbrechen des GPS-Signals wird ein neues Trace-Segement erzeugt. Nachdem die Aufnahme

beendet wird, kann der Trace inklusive der zugehörigen Marker exportiert werden. Er liegt dann in Form einer GPX-Datei vor. Diese Datei lässt sich nun an den OSM-Server übertragen, so dass alle OSM-Nutzer die Möglichkeit erhalten sie zu nutzen.

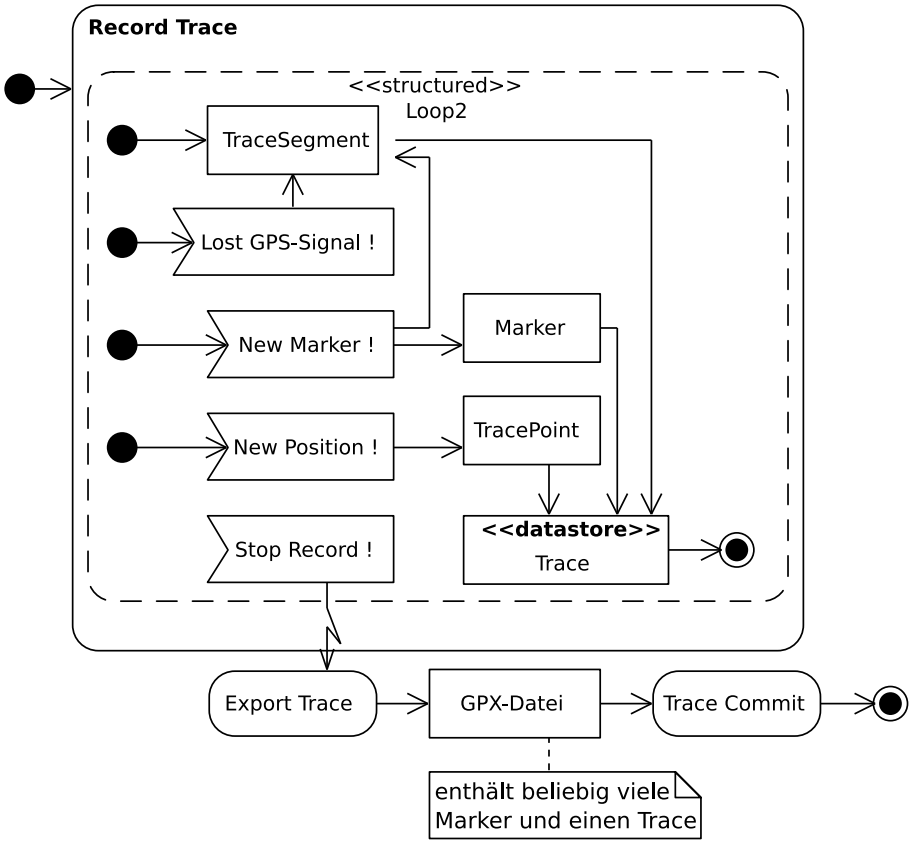

Abbildung 5.3: Trace aufzeichnen und exportieren

Eine neue Landkarte in TribOSMap integrieren

Wie bereits erwähnt, kann eine georeferenzierte Landkarte, die dem Anwender in Papierform vorliegt, in TribOSMap integriert werden. Dazu muss das Bild in das TOSM-Format umgewandelt werden und die zugehörige Georeferenzierung in einer TOSM-XML-Datei vorliegen. Nachdem die erzeugten Kartendateien in das Handy importiert wurden, kann die Landkarte im Handy betrachtet werden.

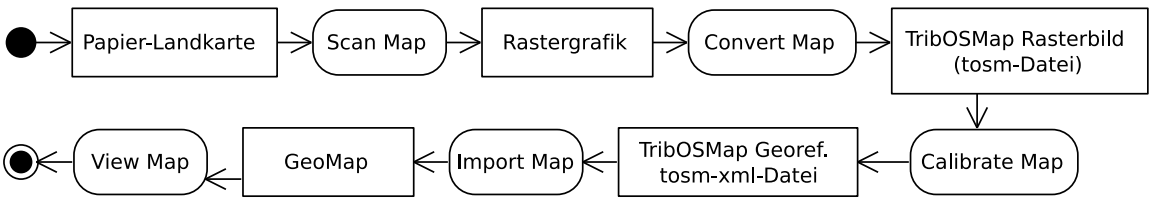

Abbildung 5.4: Landkarte in die Anwendung integrieren

5.4 Fachklassen

In diesem Abschnitt werden die Fachklassen aufgeführt und kurz erläutert, die sich aus der Analyse der Anwendungsfälle ergeben und für die Umsetzung der Anwendung TribOSMap relevant sind.

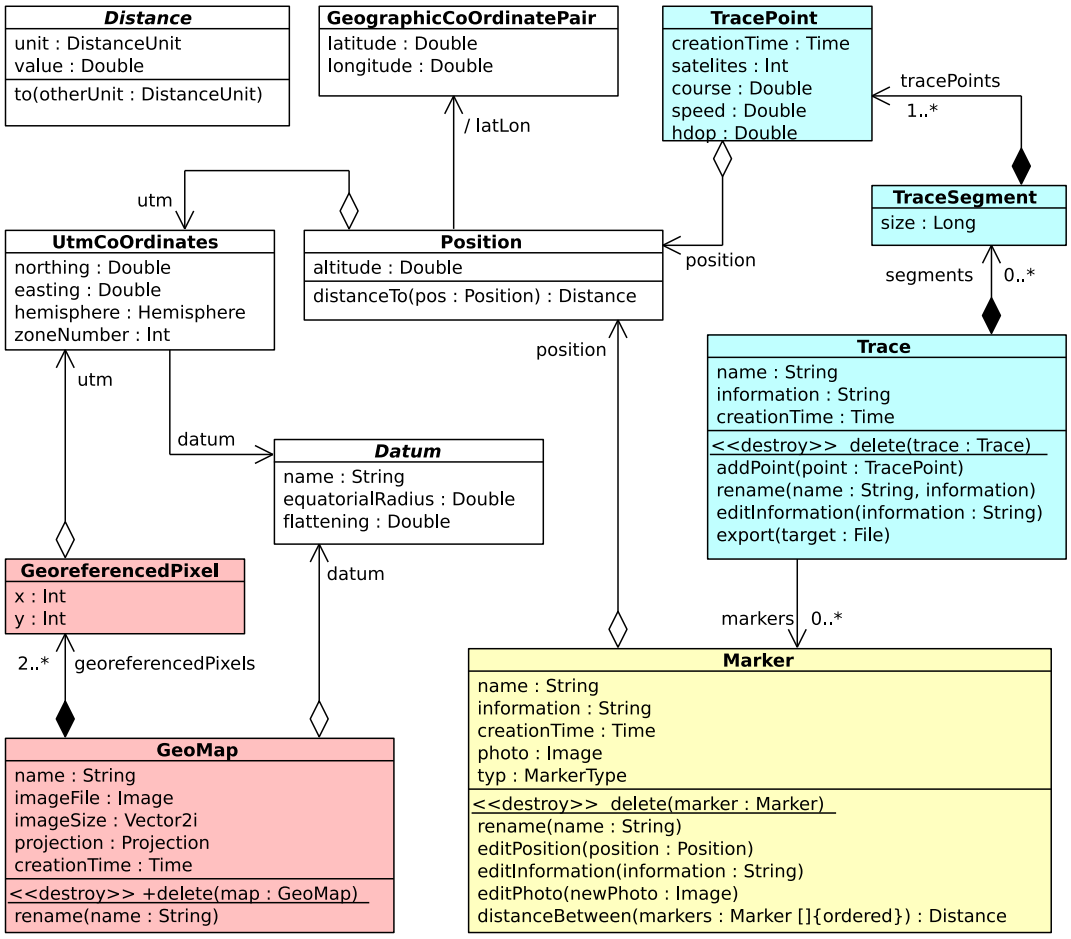

Abbildung 5.5: Fachklassen

GeographicCoOrdinatePair (Geografischen Koordinaten): Die geografischen Koordinaten sind ein Tupel aus Längen- und Breitengrad.

UtmCoOrdinates (UTM-Koordinaten): Die UTM-Koordinaten besitzen einen Northing-Wert in Metern (inkl. False-Northing) und einen Easting-Wert in Metern (inkl. False-Easting), eine Zonennummer für die Längenzone, ein geodätisches Datum sowie die Angabe zur Hemisphäre (Nord oder Süd).

Distance (Distanz): Die Klasse Distanz beschreibt eine Entfernung. Dazu gehört ein Abstandsmaßsystem (*unit*) und einen Wert (*value*). Zudem wird die Funktionalität angeboten, um eine Entfernung in ein anderes Abstandsmaßsystem zu überführen.

Datum (Geodätisches Datum): Das geodätische Datum beschreibt einen Ellipsoid. Er wird mit einem äquatorialen Radius (*equatorialRadius*) in Metern und dem sogenannten "Flattening" beschrieben, das das Verhältnis des äquatorialen zum polaren Radius ausdrückt. Der Name des Datums gibt den spezifizierten Standard an (z.B. "WGS84").

Position (Position): Die Position beschreibt die geografische Lage eines Ortes durch Angabe der Höhe in Metern (*altitude*) und der UTM-Koordinaten (*utm*). Aus diesen Informationen können die geografischen Koordinaten (*latLon*) abgeleitet werden. Darüber hinaus bietet die Klasse die Funktionalität zur Ermittlung einer Entfernung zwischen zwei Positionen an.

GeoMap (Karte): Die Klasse *GeoMap*[1], beschreibt eine Landkarte. Diese Klasse besitzt Informationen zur verwendeten Projektion und dem verwendeten geodätischen Datum dieser Projektion. Darüber hinaus muss sie mindestens zwei georeferenzierte Pixel enthalten.

GeoreferencedPixel (Georeferenzierter Pixel): Die Georeferenzierung einer Landkarte wird mit Hilfe der georeferenzierten Pixel umgesetzt. Die Klasse *GeoreferencedPixel* bildet die Koordinaten eines Pixels auf UTM-Koordinaten ab.

TracePoint (Spurpunkt): Ein *TracePoint* umfasst eine aufgezeichnete Position, sowie zusätzliche Informationen, des GPS-Receiver. Dazu gehören Angaben zur Signalqualität (*hdop*), zur aktuellen Geschwindigkeit oder zur Anzahl der aktuell verbundenen Satelliten.

TraceSegment (Spursegment): Das *TraceSegment* besteht aus einer Menge von *TracePoints*.

Trace (Spur): Ein Trace besteht aus einer Menge von Spursegmenten. Er verfügt über einen Namen und andere (optionale) Text-Informationen. Zudem kann er mehrere Marker referenzieren. Er bietet die Funktionalität an, neue Punkte aufzuzeichnen, die dann in Spursegmenten gespeichert werden. Neue Spursegmente werden erzeugt, wenn nach dem Abbrechen des GPS-Signals ein neues Signal empfangen wird oder wenn während der Aufzeichnung ein Marker erzeugt wurde.

Marker (Markierung): Ein *Marker* benötigt - wie ein *TracePoint* - eine *Position*. Weiterhin besitzt er unter anderem einen Namen, weitere (optionale) Text-Informationen und optional ein Bild (*image*). Eine Markierung kann von unterschiedlichem Typ sein (*MarkerType*). Zunächst sind zwei verschiedene Marker-Typen für die Anwendung

1 Hier wurde der Name "GeoMap" gewählt, da der Begriff "Map" in der Informatik bereits belegt ist.

geplant: der Eine für die aktuelle Position (*ActualPosition*) und der Andere für einen sogenannten interessanten Punkt (*PointOfInterest*). Wird eine Markierung während der Aufzeichnung erzeugt, dann wird er vom Trace referenziert.

5.5 Für TribOSMap entwickelte Dateiformate

In diesem Abschnitt geht es um die TOSM-Dateiformate, die für die Anwendung TribOSMap entwickelt wurden: das Dateiformat für die Rastergrafiken und das der XML-Dateien für die Georeferenzierung.

5.5.1 TOSM-Rasterbilder

Digitale Rasterbilder von Landkarten sind zu groß, um sie in einem Handy zu verarbeiten. Daher müssen sie geeignet aufbereitet werden.

Das Prinzip der Lösung basiert auf dem Verfahren, wie im OSM-Projekt Rasterlandkarten für das Internet verfügbar gemacht werden (Anhang A.2.5 auf Seite 139). Diese sind in viele kleine Einzelbilder zerlegt, so dass der Benutzer nicht alle Daten herunterladen muss, sondern nur Teilbilder, die den gerade betrachteten Ausschnitt abdecken. Außerdem verfügen diese Landkarten über mehrere Zoomstufen, für die Rasterbilder in einer bestimmten Auflösung vorgehalten werden.

Das TOSM-Dateiformat soll die Teilbilder speichern, so dass es möglich ist, auf einzelne Teile der Karte zuzugreifen, ohne die gesamte Datei in den Speicher laden zu müssen. Der Header der Datei enthält Informationen, die angegeben in welchem Zoomlevel welcher Ausschnitt des Gesamtbildes aus der Datei herauszulesen ist. Die Dateispezifikation wird im Anhang B.1 auf Seite 141 erläutert.

Die ursprüngliche Idee war, das Konvertieren der digitalen Landkarten innerhalb von Android vorzunehmen. Es stellte sich heraus, dass dies wegen des geringen Speichervolumens, das für Anwendungen in aktuellen Android-Systemen zur Verfügung steht, nicht möglich ist. Selbst kleine Landkarten sind bereits größer als der gesamte verfügbare Speicherplatz. Folglich wurde entschieden, für diese Konvertierung eine Kommandozeilenanwendung zu entwickeln, die auf einem PC innerhalb einer Standard Java-VM lauffähig ist. Auf diese Anwendung wird nicht weiter eingegangen.

5.5.2 TOSM-XML-Dateien für die Georeferenzierung

Für die Georeferenzierung der TOSM-Bilder wurde ein eigenes XML-Dateiformat entwickelt, das die notwendigen Informationen spezifiziert, um TOSM-Dateien zu georeferenzieren. Dazu gehören neben den Angaben zur Projektion und zum geodätischen

Datum eine Anzahl von georeferenzierten Pixeln. Diese georeferenzierten Pixel weisen einer Bildschirmkoordinate eine geografische Position zu. Eine DTD-Spezifikation des Formates, ein Beispiel und einige Erläuterungen sind im Anhang B.2 auf Seite 142 zu finden.

5.6 Zusammenfassung

In diesem Kapitel wurde geklärt, welche Funktionalität die Anwendung TribOSMap zur Verfügung stellen muss. Es wurden Anwendungsfälle und Dateiformate beschrieben sowie Fachklassen spezifiziert. Bei der Entwicklung der Anwendung soll eine leichte Erweiterung um weitere Algorithmen und Koordinatentypen unterstützt werden. Dazu gehören die Karten-Projektionen, die geodätischen Daten und die Abstandsmaßeinheiten.

6 Erfahrungen bei der Entwicklung der Anwendung TribOSMap

Nachdem auf Android und Scala eingegangen und ein Überblick über die zu entwickelnde Funktionalität der Beispielanwendung gegeben wurde, wird in diesem Kapitel über die Erfahrungen bei der Entwicklung berichtet. Es werden dabei Designentscheidungen besprochen und wichtige Teile des Programms ausführlich untersucht. Es wird gezeigt, wie die sprachlichen Besonderheiten von Scala genutzt wurden, um spezifische Probleme bei der Entwicklung zu lösen.

Zunächst wird die Paketstruktur der Anwendung besprochen, um einen Überblick über die gesamte Architektur zu gegeben. Im Anschluss wird auf die Programmierung der Benutzerschnittstelle eingegangen. Danach werden die Implementierung der Geschäftslogik und technische Details vorgestellt. Bei der Implementierung traten grundlegende Probleme auf, die im Zusammenhang mit dem Android-Framework stehen. Diese werden gezeigt und ihre Lösung unter Ausnutzung der Fähigkeiten von Scala dargelegt.

6.1 Paketstruktur

Die Architektur der Anwendung TribOSMap ist in zwei Ebenen aufgeteilt. Die Module des Frontend befinden sich im Paket *org.tribosmap.frontend*. Hier ist die Benutzerschnittstelle umgesetzt. Die untere Ebene ist die Modellschicht (Backend), die sich im Paket *org.tribosmap.model* befindet. In ihr sind die Geschäftslogik, der Datenbankzugriff und diverse Anwendungsdienste implementiert.

6.1.1 Pakete des Frontend

frontend: Die Pakete *map*, *marker* und *trace* tragen die Namen der zentralen Fachklassen der in Abschnitt 5.3.3 auf Seite 59 definierten Anwendungsfälle und enthalten die zugehörigen Activities und Widgets. Somit ist der Inhalt dieser drei Pakete eng mit der Funktionalität der Anwendung verbunden. Die implementierten "Fachklassen-Activities" benötigen spezielle Schnittstellen und konkrete Implementierungen der

Modellschicht.

frontend.common: Im Paket *common* existieren Klassen, die durchaus auch für die Entwicklung anderer Android-Anwendungen interessant sein dürften. Dazu gehören unter anderem Widgets und allgemeine (abstrakte) Activities, die von konkreten Activities der Anwendung verwendet werden.

Grundsätzlich sind die Pakete des Frontend stark vom Android-Framework abhängig.

6.1.2 Pakete der Modellschicht

Direkt im Paket *model* befindet sich das Trait *ServiceAccess*, das eine bedeutende Rolle in der Architektur der Anwendung spielt. Es verknüpft das Frontend mit der Modellschicht und die Geschäftslogik mit der Datenbankimplementierung.

model.math: Die mathematischen Klassen in *math* setzen diverse mathematische Algorithmen um und führen einige mathematische Typen ein. Dazu gehört zum Beispiel die Klasse *Vector2d*, die zum Rechnen mit einem Tupel von Fließkommazahlen dient. Andere Klassen liefern neue Datentypen zum Rechnen mit geografischen Koordinaten. Das Paket *math* ist weder abhängig vom Android-Framework noch von anderen Teilen der entwickelten Anwendung. Folglich sind die dort vorhandenen Komponenten vollständig wiederverwendbar.

model.business: Die Geschäftslogik für die persistenten Fachklassen, realisiert im Paket *business*, ist ebenfalls unabhängig vom Android-System. Dies wurde durch das Entwickeln einer Schnittstelle für die Datenbankimplementierung erreicht, die im Paket *business.repository* implementiert wurde. Zudem ist die Geschäftslogik für andere Anwendungen wiederverwendbar und die Datenbankimplementierung auf einfache Art und Weise austauschbar.

model.database: Im Paket *database* ist die Persistenz realisiert, indem die Spezifikation aus dem Paket *business.repository* umgesetzt wurde. Hierbei wird eine spezifische Datenbankimplementierung des Android-Framework verwendet.

model.app: Im Paket *app* sind spezifische Implementierungen für technische Dienste zu finden (z.B. der *ImageLoader* zum Laden von Bildern).

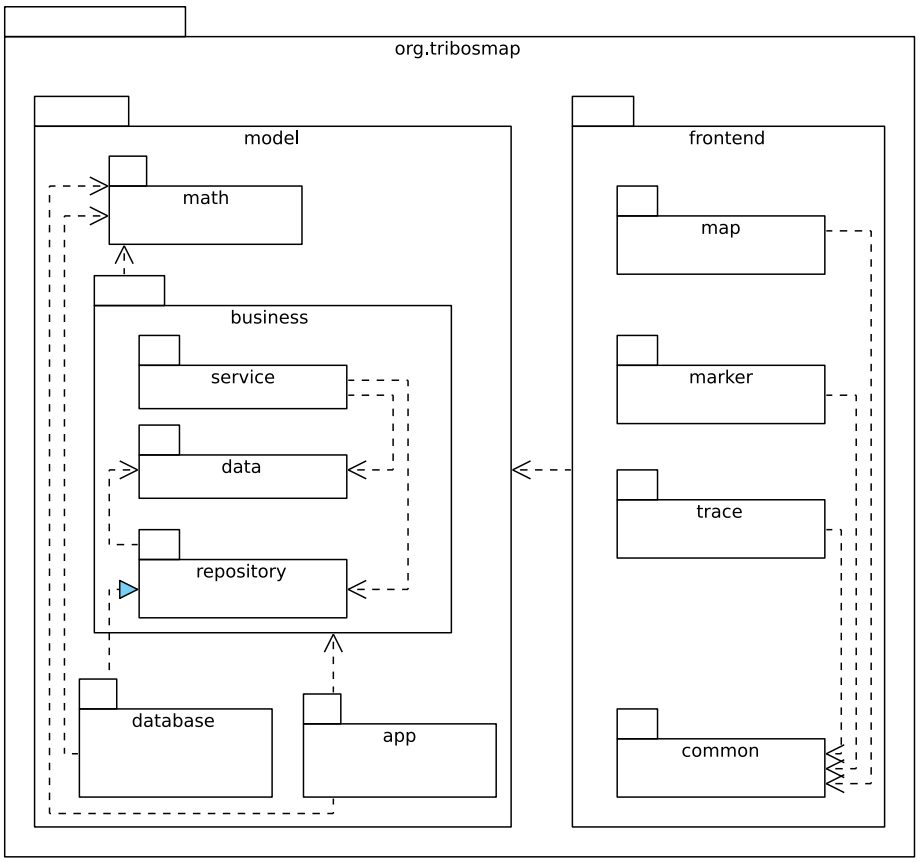

Abbildung 6.1: Paketstruktur - TribOSMap

6.1.3 Schlussbemerkung zur Paketstruktur

Ein grundsätzliches Ziel bei der Gestaltung der Architektur war es, möglichst viele zusammenhängende Teile der Implementierung für eine Nutzung in anderen Systemen wiederverwendbar zu machen. Für einige Pakete der Modellschicht ist dies auch gelungen (z.B. *math*, *business*), da sie nicht vom Android-Framework abhängig sind.

6.2 Architektur und Implementierungsdetails des Frontend

Die wichtigsten Komponenten des Frontend sind die Activities, um die es in diesem Abschnitt hauptsächlich geht. Dabei werden auch Komponenten vorgestellt, die zur Vermeidung von Redundanzen zwischen den verschiedenen Activities entwickelt wurden.

6.2.1 Allgemeine Umsetzung der Activities

Ein Activity setzt die Interaktion für bestimmte Geschäftsprozesse um. Dazu arbeitet es mit den Widgets zusammen. Eingaben der Benutzer werden verarbeitet und die Darstellung der Widgets verändert. In Abschnitt 3.4.2 auf Seite 37 wurde auf die grundsätzliche

Arbeitsweise der Activities bereits eingegangen und erwähnt, dass sie von einer Klasse *Activity* des Android-Framework abgeleitet sein müssen.

Zu jedem hier entwickelten Activity gehört grundsätzlich eine Layout-XML-Datei, in der festgelegt wird, welche Widgets wo dargestellt werden. Diese Widgets besitzen eine Identifikationsnummer (ID). Um Zugriff auf die Instanzen der Widgets zu erlangen, wird die Methode *findViewById(id: Int): Object* der Klasse *Activity* genutzt, wobei als Parameter die ID des gewünschten Widget verwendet wird. Diese IDs stammen von statischen Attributen der automatisch generierten Klasse *R* (Abschnitt 3.4.4 auf Seite 39).

Mit einer Referenz auf ein Widget-Objekt kann ein Activity den Zustand der Darstellung verändern. Aktive Widgets[1] können mit Aktionen verknüpft werden, indem ein Widget Methoden bereitstellt, die mit sogenannten Listeners parametrisiert werden. Finden Interaktionen des Benutzers mit einem aktiven Widget statt (Ereignisse), werden diese Listener-Methoden aufgerufen, in denen die zugehörigen Aktionen implementiert sind. Dieses Konzept basiert auf dem Beobachter-Entwurfsmuster. Dabei ist das Widget das "Subjekt" und das Listener-Objekt der "Beobachter"[Gam95, Seite 326 ff.].

Problem der veränderlichen Variablen für Widget-Referenzen

Auf den Lebenszyklus der Activites ist in Abschnitt 3.4.2 auf Seite 37 bereits eingegangen worden. Danach sind die Widgets erst nach der Instantiierung der Activities verfügbar. Wird die Methode *findViewById* vor dem ersten Aufruf der *onCreate*-Methode verwendet, ist das Resultat immer *null*.

Bei einer Java-Implementierung werden deshalb die Referenzen zu den Widgets, die klassenweit zur Verfügung stehen sollen, zunächst im Klassenkörper deklariert, um sie dann in der *onCreate* Methode zu initialisieren. Aus diesem Grund können die Variablen für diese Referenzen nicht unveränderlich sein.

Lösung mit lazy-Variablen: In Scala dagegen kann mit den in Abschnitt 2.3.1 auf Seite 12 vorgestellten unveränderlichen *lazy*-Variablen gearbeitet werden. Die Referenzen werden direkt im Klassenrumpf initialisiert, wobei erst bei dem ersten Zugriff die eigentliche Zuweisung vollzogen wird. Die Widget-Referenzen können - wie gehabt - in den Methoden des Activity verwendet werden. In welcher Methode dies das erste Mal geschieht, also die eigentliche Initialisierung stattfindet, ist unerheblich. Es ist lediglich darauf zu achten, dass die Referenzen nicht in Ausdrücken des Klassenrumpfes benutzt werden, die bereits bei der Instantiierung des Activity ausgewertet werden.

1 Aktiv werden im Folgenden die Widgets genannt, die Eingaben des Benutzers erhalten können. Die passiven Widgets zeigen lediglich Informationen an.

Grundsätzlicher Aufbau der Activities anhand eines Beispiels

Im Folgenden wird ein Auszug der Klasse *MainActivity* des Paketes *frontend* gezeigt und erläutert. Dieses Activity enthält mehrere Schaltflächen mit denen der Benutzer die einzelnen Fachklassen-Activities starten kann. Zudem gibt es Widgets, die die aktuelle Position anzeigen.

```
class MainActivity extends Activity with LocationAdapter {
  lazy val lblAltitude = findViewById(R.id.lblAltitude).asInstanceOf[TextView]
  lazy val btnImport = findViewById(R.id.btnImport).asInstanceOf[ImageTextButton]
  def onCreate(bundle : Bundle) { btnImport.setOnClickAction(/*...*/) /*...*/}
  override def onLocationChanged(loc : Position) = lblAltitude.setText(/*...*/)
  /*...*/
}
```

Programmauszug 6.1: Activity und Widgets

Die *lazy*-Variable *lblAltitude* wird erst beim ersten Aufruf der Methode *onLocationChanged* initialisiert. Diese Variable referenziert ein passives Widget, welches die aktuelle Höhe anzeigt. Ein aktives Widget ist mit der *lazy*-Variablen *btnImport* referenziert; es wird in der *onCreate*-Methode mit einer Aktion verknüpft. Durch Einmischen des Trait *LocationAdapter* wird bei Positionswechsel die Methode *onLocationChanged* aufgerufen.

6.2.2 Activities und Traits

Um Redundanzen zwischen Activities zu vermeiden, kann Vererbung oder das Strategie-Muster eingesetzt werden. Die Vererbung hat den großen Nachteil, dass komplexe Klassenhierarchien aufgebaut werden müssen, wobei die neue Basisklasse auch von der Klasse *Activity* abgeleitet sein muss. Gerade in größeren Projekten führt dies zu eingeschränkt wiederverwendbaren Komponenten und großer Komplexität. Wird dagegen das Strategiemuster verwendet, müssen in den Activities weitere Objekte instantiiert werden. Das Injizieren von zusätzlicher Funktionalität über den Konstruktor ist nicht möglich, da Activities keinen aufrufbaren Konstruktor besitzen. Auch über Methodenaufrufe ist dies nicht möglich, denn mit Activities kann lediglich über die Bundles der Intents kommuniziert werden (Abschnitt 3.4.2 auf Seite 37).

In Abschnitt 2.3.3 auf Seite 17 ist auf die Vorteile der Traits bereits eingegangen worden. Im Folgenden wird gezeigt, wie diese bei der Implementierung der Activities genutzt wurden, um die besprochenen Probleme zu lösen.

Verwendung von Traits in Activities

Im Programmauszug 6.1 auf der vorherigen Seite wurde das Trait *LocationAdapter* verwendet, um das Activity um zusätzliche Funktionalität zu erweitern. Interessant ist es, wenn eine größere Zahl von Traits eingesetzt wird. Im folgenden sind in der Klasse *MainActivity* neben dem Trait *LocationAdapter* noch die Traits *ActivityHelper* und *ServiceAccess* eingemischt. Das Trait *ActivityHelper* stellt eine Hilfsmethode für die Erzeugung der Widgetreferenzen zur Verfügung (*getView*). Das Trait *ServiceAccess* besitzt ein "plug" *mapService*, das die Funktionalität für das Arbeiten mit der Fachklasse *GeoMap* anbietet.

```
class MainActivity extends Activity
    with LocationAdapter with ActivityHelper with ServiceAccess {
    lazy val lblAltitude: TextView = getView(R.id.lblAltitude)
    def importMapAction(fileName: String) { mapService.importMapFromXml(/*...*/) }
    /*...*/
}
```

Programmauszug 6.2: Activity und Traits

Im Programmauszug 6.2 ist zu erkennen, wie die Methode *getView* und der Geschäftslogik-Dienst *mapService* verwendet wird. Das Importieren einer Karte wird im MainActivity initiiert, da der Benutzer das Importieren einer Karte direkt über das Hauptmenü veranlassen kann.

In Abbildung 6.2 werden die Zusammenhänge zwischen den Traits und dem MainActivity illustriert.

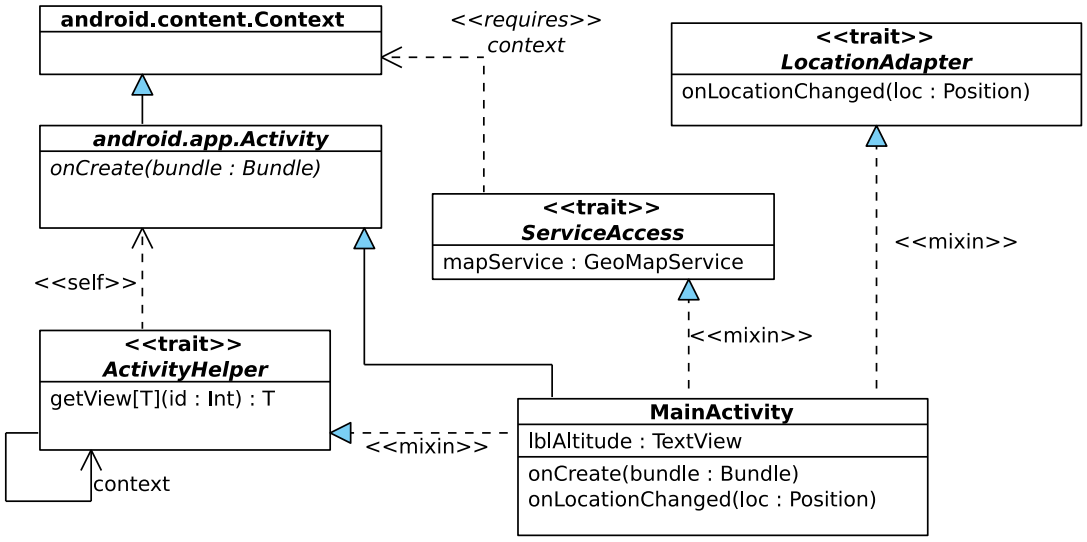

Abbildung 6.2: MainActivity und Traits

Genutzte Traits im Frontend

Im Folgenden werden die verwendeten Traits kurz vorgestellt. Sie werden von vielen Activities in unterschiedlicher Kombination verwendet.

LocationAdapter: Das Trait LocationAdapter des Paketes *model.app* implementiert alle Methoden der Schnittstelle *android.location.LocationListener*. Zudem kapselt es die Methoden zum An- und Abmelden des Listener an den Location-Manager (Abschnitt 3.3.4 auf Seite 36). In der dieses Trait einmischenden Klasse müssen lediglich relevante Methoden überschrieben und das Empfangen der Positionsdaten initiiert werden.

ServiceAccess: Das Trait ServiceAccess kapselt den Zugriff auf die Geschäftslogik des Paketes *model.business* und bietet deren Dienste als "plugs" an. Es verwendet keinen selbstreferenzierten Typ *android.content.Context* (Abschnitt 2.3.3 auf Seite 19), dadurch kann es auch von Klassen eingemischt werden die nicht von *Context* abgeleitet sind. Stattdessen besitzt es ein abstraktes Attribut *context* vom Typ *Context*, da die Modellschicht eine Referenz zu einem Context-Objekt benötigt (Abschnitt 3.4.5 auf Seite 39).

ActivityHelper: Das Trait *ActivityHelper* verfügt über Funktionalität, die in vielen Activities benötigt wird. Durch sein Einmischen wird das "socket" *context* des Trait *ServiceAccess* befriedigt. Denn der *ActivityHelper* hat einen selbstreferenzierten Typ *Context* und ein Attribut *context*, das die eigene Instanz referenziert (Abbildung 6.2 auf der vorherigen Seite).

Abschließende Bemerkungen

Die Verwendung der Traits als Komponenten mit spezifischer Funktionalität hat sich bei der Implementierung der Activities als sehr komfortabel erwiesen. Zum Beispiel wenn durch ein Refactoring der Zugriff auf die GPS-Daten in einem Activity nicht mehr nötig war, konnte das betreffende Trait einfach aus der Klassensignatur entfernt werden. Umgekehrt ist es genauso möglich, einem Activity im Nachhinein den Zugriff auf die Geschäftslogik zu ermöglichen, indem das Trait *ServiceAccess* der Klassensignatur hinzugefügt wird.

Die Modularisierung durch Traits ermöglich eine gute Kapselung von Funktionalität nach Zuständigkeiten und eine flexible Kombination. Mit Hilfe abstrakter Attribute ("sockets") lassen sich auf einfache Art und Weise notwendige Objekte in den Traits verwenden, die zum Teil nicht außerhalb eines Activity instantiiert werden können (Context-Objekt).

6.2.3 Details zum Trait ActivityHelper

Das erwähnte Trait *ActivityHelper* wird von fast allen Activities eingemischt. Um die Activities übersichtlicher und redundanzarmer implementieren zu können, stellt es Hilfs-Methoden zur Verfügung. Auf die wichtigsten Methoden wird in den folgenden Abschnitten eingegangen.

Behandlung von Fehlern mit Funktionen höherer Ordnung

Die Activities nutzen Methoden, die von Komponenten der Modellschicht zur Verfügung gestellt werden. Da diese Methoden zu bestimmten Fehlern führen können, müssen die Aufrufe in *try-catch*-Konstrukte integriert werden[1]. Unbehandelte Fehler würden ein Activity beenden. Dies muss verhindert werden, um den Benutzer über die Art eines Fehlers zu informieren und es ihm zu ermöglichen zu reagieren.

Die Abbildung 6.3 auf der nächsten Seite zeigt einen typischen Fehlerdialog. Um diesen Anzuzeigen, wurde im Trait *ActivityHelper* eine Methode *displayError(error: Throwable)* implementiert. Die Funktion höherer Ordnung *tryCatch* nutzt diese Methode und appliziert eine übergebene Funktion innerhalb eines *try-catch*-Konstruktes. Da *tryCatch* polymorph und totalisiert ist, kann die übergebene Funktion ein Resultat beliebigen Typs besitzen.

```
def tryCatch[T](fun : ⇒ T): Option[T] =
  try { Some(fun) } catch { case e : Throwable ⇒ displayError(e); None }
```

Programmauszug 6.3: Definition der tryCatch-Methode

Die Klasse *MarkerEditactivity* nutzt ebenfalls die Funktion *tryCatch*. In einer Methode *saveAction* wird aus einem Widget *editNameField* ein eingegebener Name ausgelesen, der mit der validierenden Methode *rename* einem Marker zugewiesen wird. Schlägt die Validierung fehl, dann wird die Ausführung der Methode *saveAction* terminiert, der Fehlerdialog geöffnet und der Marker nicht persistiert.

```
def saveAction() = tryCatch {
  marker.rename(editNameField.getText.toString)
  /*...*/
  marker.save() //persistiert Änderungen
}
```

Programmauszug 6.4: Nutzung der tryCatch-Methode

1 Das try-catch-Konstrukt zur Fehlerbehandlung ist Bestandteil der meisten modernen Programmiersprachen (Java, Scala, C++, Python etc.).

Abschließende Bemerkung: Mit der vorgestellten Methode *tryCatch* ist es mit minimalem Aufwand möglich, Zugriffe des Activity auf die Modellschicht abzusichern und für die Benachrichtigung des Benutzers über auftretende Fehler zu sorgen. Ein Block von Anweisungen wird einfach als anonyme Funktion der Methode *tryCatch* als Parameter übergeben. Das aufwändige Behandeln der Ausnahme und das Anzeigen des Fehlerdialoges wird zentral im *ActivityHelper* vorgenommen.

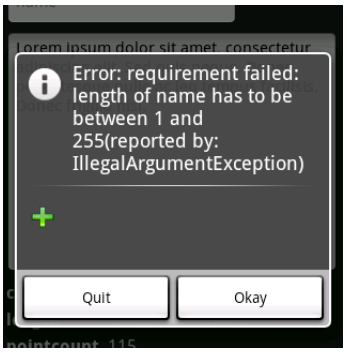

Abbildung 6.3: Fehlerdialog

Nutzung impliziter Typparameter zur Erzeugung von Widget-Referenzen

Damit ein Activity eine Referenz auf eine GUI-Komponente erhalten kann, muss die Methode *findViewById* der Klasse *android.app.Activity* verwendet werden. Im Anschluss ist eine explizite Typumwandlung mit dem Aufruf der Methode *asInstanceOf* notwendig. In einigen Activities wären sehr große Blöcke dieser sehr ähnlichen Ausdrücke zu schreiben, da viele Widgets verwendet werden (Programmauszug 6.1 auf Seite 71). Deshalb wurde im *ActivityHelper* eine Methode *getView* implementiert. Diese kapselt die Methode *findViewById* und die explizite Typumwandlung. Das Trait *ActivityHelper* besitzt einen selbstreferenzierten Typ auf die Klasse *Activity*, womit es möglich wird die Methode *findViewById* hier zu nutzen.

```
def getView[T](id : Int) = findViewById(id).asInstanceOf[T]
```

Programmauszug 6.5: Vereinfachen des Zugriffes auf Widget-Referenzen

Die Methode *getView* erwartet den Typ des Widget als Parameter. Es kann aber auch die Typinferenz von Scala genutzt werden, um den Typ implizit zu übergeben. Dazu wird einer explizit typisierten Variablen das Resultat der Methode *getView* zugewiesen.

```
val lblAltitude : TextView = getView(R.id.lblAltitude)
```

Programmauszug 6.6: Widget-Referenzen und implizite Typparameter

Abschließende Bemerkung: Die Variablen für die Widget-Referenzen können mit der polymorphen Methode *getView* sehr elegant initialisiert werden. Zu kritisieren ist, dass die explizite Typumwandlung mit *asInstanceOf* verborgen wird. Im Programmcode fällt nicht auf, dass beim Aufruf der Methode *getView* Laufzeitfehler stattfinden können. Deshalb sollte das gezeigten Vorgehen nur nach Abwägen der Vor- und Nachteile verwendet werden.

Implizite Transformation von Funktionen zu Listener-Objekten

Einige der in Abschnitt 6.2.1 auf Seite 69 erwähnten "Listeners" besitzen lediglich **eine** Methode. In diesem Fall sind sie mit den "Callback"-Mechanismen von Sprachen vergleichbar, die die Möglichkeiten besitzen, Funktionen als Parameter zu verwenden. Zum Beispiel kann an ein Objekt der Klasse *android.widget.ImageButton* ein Listener vom Typ *android.view.OnClickListener* angemeldet werden. *OnClickListener* ist eine Schnittstelle, die nur die Methode *onClick* spezifiziert. Bei der herkömmlichen Verwendung dieses Listener wird eine anonyme Klasse für die Schnittstelle umgesetzt und instantiiert. Die eigentliche Aktion ist innerhalb der Methode *onClick* implementiert. Das instantiierte Listener-Objekt wird mit der Methode *setOnClickListener* beim Widget angemeldet. Sobald das Widget das Ereignis "Click" erhält, wird die Methode *onClick* aller angemeldeten Listener ausgeführt.

```
button.setOnClickListener( new OnClickListener{
   def onClick(v :View){ /*...*/ }
})
```

Programmauszug 6.7: Herkömmliche Verwendung des Listener-Konzeptes

Im *ActivityHelper* wurde die implizite Funktion *clickBtn* definiert, die eine parameterlose Funktion mit beliebigem Resultat in einen *OnClickListener* transformiert. Die der *clickBtn*-Methode übergebene Funktion, wird dann direkt in der *onClick*-Methode des *OnClickListener* aufgerufen.

```
implicit def clickBtn(action : () ⇒ Any) = new OnClickListener {
   def onClick(v :View){ action() }
}
```

Programmauszug 6.8: Umwandlung Funktionen zu Listener-Objekten

Abschließende Bemerkung: Alle das Trait *ActivityHelper* einmischenden Klassen können direkt Funktionen als Parameter für Methoden verwenden, die Objekte des Typs *OnClickListener* erwarten. Die mehrmalige Implementierung anonymer Klassen entfällt und die Activities werden wesentlich übersichtlicher.

6.2.4 Umsetzung der Listenansichten

In der Anwendung TribOSMap gibt es mehrere Activities, die eine Liste von Objekten visualisieren und eine Interaktion mit dem Benutzer ermöglichen: Listenansichten von Markierungen, Traces, Karten und Dateien. Das Android-Framework bietet für die Implementierung solcher Listenansichten die abstrakte Klasse *ListActivity* an. Um sie zu verwenden, muss für die in der Liste dargestellten Daten ein *ArrayAdapter[T]* implementiert werden. In ihm werden Objekte auf die Zeilen der Listenansicht abgebildet. Alle

Listenansichten in TribOSMap ähneln sich grundsätzlich, weil die Zeilen meist eine gleiche Anzahl und Art von grafischen Komponenten enthalten (Abbildung 6.4). Aus diesem Grund ähnelten sich auch der Code der implementierten *ArrayAdapter* und *ListActivities*.

Abbildung 6.4: Bildschirmfotos der Listactivities

Um Redundanzen zu vermeiden, wurde eine von *ArrayAdapter* abgeleitete generische Klasse *TribosListAdapter* implementiert. Diese kann sämtliche Objekte des mit einer Schnittstelle spezifizierten Typs *TribosListRow* auf der Oberfläche darstellen. Der allgemeinere *ArrayAdapter* ist damit um spezielle Funktionalität erweitert worden. Darüber hinaus wurde eine abstrakte generische Klasse *TribosListActivity* implementiert, die die Funktionalität für das Arbeiten mit dem *TribosListAdapter* realisiert.

Eine von *TribosListActivity* abgeleitete Klasse ist zum Beispiel der *FileChooser*. Dieses Activity stellt Objekte der Klasse *File* auf der Oberfläche dar. Dazu wird die Klasse *FileRow* von *TribosListRow* abgeleitet und Attribute der Klasse *File* auf bestimmte Elemente der Zeilenansicht abgebildet. Bei dem Beerben der generischen Klasse *TribosListActivity* wird der Typ *FileRow* als Parameter verwendet. Innerhalb des *FileChooser* muss lediglich eine Liste mit den darzustellenden Elementen gefüllt und die Interaktion mit dem Benutzer implementiert werden. Die aufwändige Implementierung eines spezifischen *ArrayAdapter* entfällt.

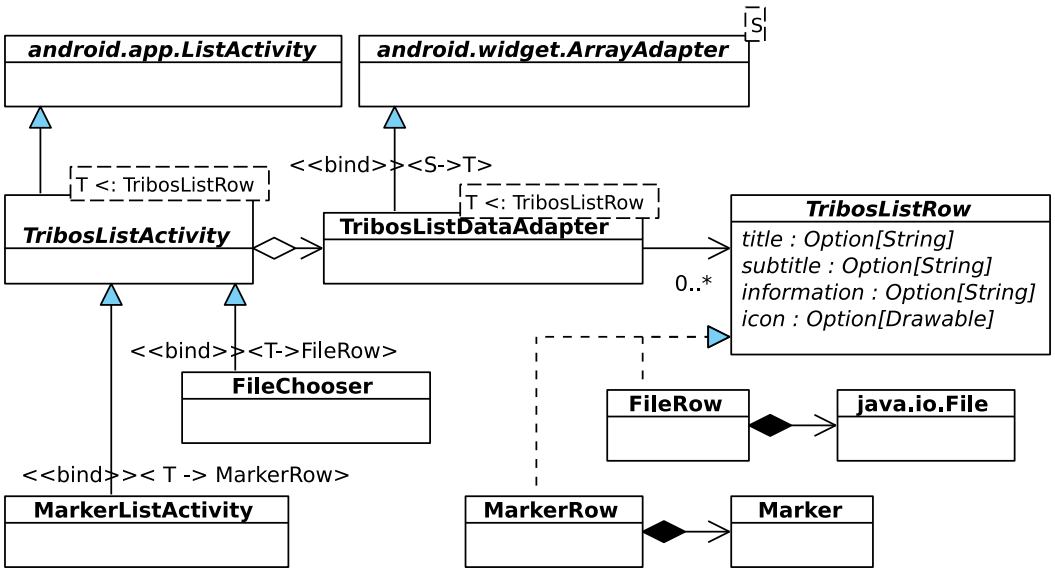

Abbildung 6.5: Die Activities der Listenansichten

Abschließende Bemerkung: Der auf die Anforderungen der Listenansichten angepasste *ArrayAdapter* ist für alle Datentypen von TribOSMap nutzbar. Prinzipiell lässt sich dieses Vorgehen auch mit Java umsetzen. Bis auf die Besonderheit das die abstrakten Attribute des Trait *TribosListRow* den Typ *Option* besitzen, um ein Arbeiten mit dem Wert *null* zu verhindern, falls sie undefiniert sind (Abbildung 6.5).

6.2.5 Umsetzung der Kartenansicht

Für das Betrachten der Karte wurde die Klasse *MapViewActivity* implementiert. Sie fungiert als Controller, der Benutzereingaben sowie die Positionsdaten des GPS-Dienstes entgegennimmt, wodurch die Ansicht aktualisiert wird, die in einer von *android.view.View* abgeleiteten Klasse *MapView* implementiert ist. Sie stellt die *MapTiles*, also die Teilbilder der Karte, dar.

Abbildung 6.6: Kartenteile

Beim Bewegen der Karte sorgt die Klasse *MapViewActivity* für eine Aktualisierung der *MapTiles*. Den *MapTiles* werden die neuen Bildschirmkoordinaten zugewiesen, an denen *MapView* sie zeichnen soll. Zudem werden die Bilder der *MapTiles* ausgetauscht, wenn ein neuer geografischer Bereich sichtbar wird.

Der betrachtete Bereich des Kartenbildes hängt von der Bildschirmauflösung ab. Die Teilbilder der Karte besitzen eine festgelegte Größe. In Abbildung 6.6 ist die Auflösung

des Bildschirms 320 x 240 Pixel. Die Kartenteile haben eine Kantenlänge von 256 Pixeln. Um den Bereich der Ansicht in jeder Situation abzudecken, sind 3 x 3 Teile notwendig.

Um das Verhalten der Kartenansicht, das Nachladen der Kartenteile und auch das Einzeichnen von Markierungen auf die betrachtete Karte abzustimmen, verwendet das *MapViewActivity* Algorithmen, die abhängig von der Projektion der Karte, der Anzahl der Zoomlevel, der Größe der Kartenteile und einigen anderen Details sind. Umgesetzt ist diese Funktionalität in Klassen, die das Trait *MapAlgorithm* einmischen. Im Falle der OpenStreetMap-Karte ist dies die Klasse *SlippyMap*. Für das Laden der Teile aus einer Datei oder dem Internet verwendet das *MapViewActivity* das Singleton-Objekt *ImageLoader*.

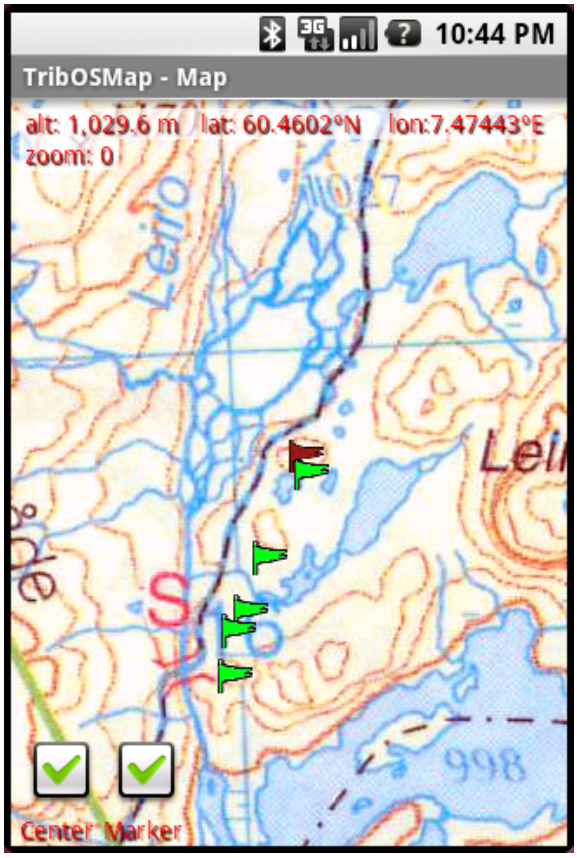

Abbildung 6.7: Offline-Karte mit Markierungen

6.3 Modellschicht

Bei der Umsetzung der Modellschicht wurden folgende Ziele verfolgt:

- Die Geschäftslogik ist weitgehend unabhängig vom Android-Framework zu implementieren. Da Klassen des Android-Framework (z.B. Context-Objekt) für den Zugriff auf System-Ressourcen benötigt werden, muss die Geschäftslogik von der Datenbankimplementierung getrennt werden.
- Zugriffe auf die Datenbank sollen nicht im Frontend verstreut, sondern in der Modellschicht verborgen vorgenommen werden.
- Die Datenbankimplementierung soll einfach austauschbar sein.

Auf die Probleme und Lösungen bei der Umsetzung der genannten Ziele und auf Details der Implementierung der Modellschicht (*tribosmap.model*) wird im Folgenden eingegangen.

6.3.1 Persistenzschicht

Die Objekte einiger Fachklassen, wie zum Beispiel *Marker* oder *Trace*, müssen im System dauerhaft (persistent) gespeichert werden.

SQLite-Datenbank im Android-Framework

Für die Datenbankimplementierung wird die Bibliothek SQLite des Android-Framework genutzt. SQLite ist ein sehr kleines relationales Datenbanksystem, das wenig Systemressourcen verbraucht. Die API für SQLite besteht aus einigen Klassen im Paket *android.database.sqlite*.

Um einen Zugriff auf die Datenbank zu erhalten, muss zunächst die Klasse *SQLiteOpenHelper* instantiiert werden, wobei unter anderem der Name der Datenbank angegeben wird. Mit der erzeugten Instanz kann ein Objekt der Klasse *SQLiteDatabase* instantiiert werden, das für das Absetzen von SQL-Befehlen im Datenbanksystem genutzt wird.

DAO-Entwurfsmuster

Um die persistenten Fachklassen auf die Tabellen einer relationalen Datenbank abzubilden, bietet sich das Entwurfsmuster “Data Access Object” (DAO) an. Dabei wird für jede zu persistierende Klasse eine DAO-Schnittstelle und eine DAO-Klasse implementiert. Die DAO-Klassen kapseln die notwendigen Datenbankaufrufe zum Persistieren der zugehörigen Geschäftsobjekte. Die Geschäftslogik nutzt die DAO-Schnittstellen, wodurch eine Trennung von der Datenbankimplementierung erreicht wird. [Vgl. DAO02]

In Abbildung 6.8 ist die grundsätzliche Umsetzung des DAO-Entwurfsmusters am Beispiel der Fachklasse *Marker* dargestellt. Der Dienst[1] *MarkerService*, der sich im Paket *business.service* befindet, wird auf eine Schnittstelle *MarkerDAO* programmiert. Diese Schnittstelle ist als Trait umgesetzt und befindet sich im Paket *business.repository*. Die konkrete DAO-Klasse *SQLiteMarkerDAO* des Paketes *database* kapselt den Zugriff auf die SQLite-Datenbank.

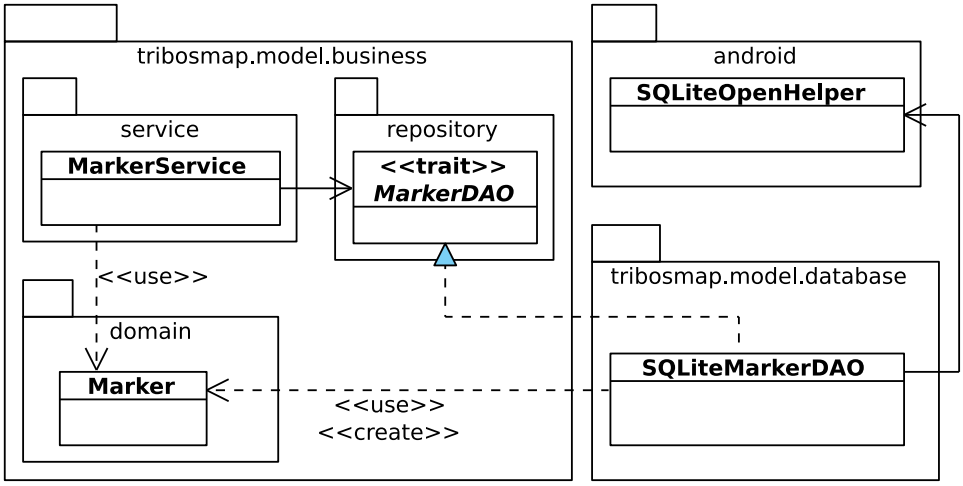

Abbildung 6.8: Umsetzung des DAO-Entwurfmusters

Abschließende Bemerkung: Das DAO-Entwurfsmuster vereinfacht den Austausch der Datenbankimplementierung und dadurch auch das Testen. Von Nachteil ist, dass die Fachklassen keine Möglichkeiten besitzen, Objekte aus der Datenbank zu lesen.

Zentrale Instantiierung der DAO-Klassen und das Trait ServiceAccess

Es stellt sich die Frage, wo die DAO-Klassen instantiiert werden und wie ein DAO-Objekt die für den Datenbankzugriff notwendige Instanz der Klasse *SQLiteOpenHelper* erhält. Für diese Instantiierung ist eine Referenz auf das Context-Objekt notwendig, die aber nur über ein gestartetes Activity zu erhalten ist. Das Activity könnte dem Dienst sein Context-Objekt direkt übergeben, um ihm zu ermöglichen, die notwendigen DAO-Objekte zu instantiieren. Das würde den Dienst aber direkt von Klassen des Android-Framework und den konkreten DAO-Klassen abhängig machen.

Daher wurde zunächst eine Schnittstelle *DAOAccessor* erstellt. In ihr ist für jede DAO-Klasse ein öffentliches Attribut deklariert. Eine Klasse *SQLiteDAOAccessor* implementiert diese Schnittstelle, erwartet ein Context-Objekt als Konstruktorparameter und instantiiert die DAO-Klassen nach Bedarf. Da das Trait *ServiceAccess* (Abschnitt 6.2.2 auf Seite 73) ein

1 Wenn in diesem Kapitel von Diensten gesprochen wird, dann sind die Dienste der Geschäftslogik gemeint.

Context-Objekt als "socket" definiert hat, kann es eine Instanz der Klasse *SQLiteDAOAccessor* erzeugen und den Diensten injizieren.

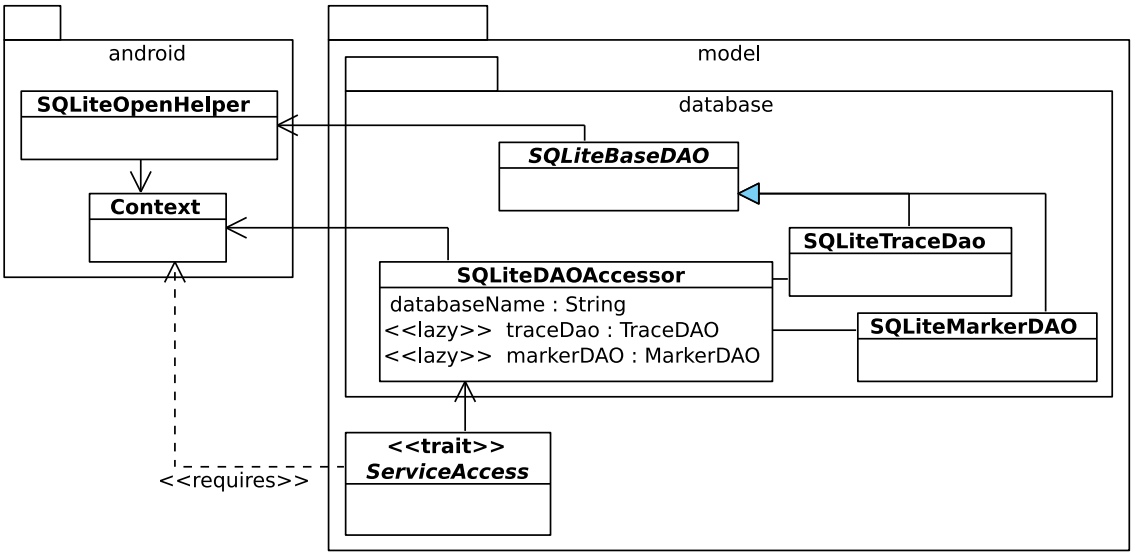

Abbildung 6.9: SQLiteDAOAccessor und ServiceAccess

Abschließende Bemerkungen: Lediglich die konkrete DAO-Implementierung und das Trait *ServiceAccess* sind vom Android-Framework abhängig. Dagegen bleibt die Geschäftslogik unabhängig, da sie die DAO-Schnittstellen und das Trait *DAO-Accessor* verwendet. Mit Hilfe von Traits wurde es möglich, Abhängigkeiten und Instantiierungen aus Klassen zu extrahieren. Auf Details bezüglich der Injektion dieser Abhängigkeiten wird eingegangen, wenn die Dienste vorgestellt werden (Abschnitt 6.3.2 auf Seite 87).

Implementierungsdetails des SQLiteDAOAccessor

Bedarfsorientierte Instantiierung der DAO-Klassen: Die Instantiierung der DAO-Klassen findet in Ausdrücken statt, die an Attribute des *SQLiteDAOAccessor* gebunden sind. Wenn für diese Attribute herkömmliche Variablen verwendet werden, führt eine Instantiierung des *SQLiteDAOAccessor* zur sofortigen Erzeugung aller DAO-Objekte. Es gibt aber Geschäftsprozesse, bei denen nicht alle notwendig sind. Bei der Verwendung von Methoden würden die DAO-Objekte innerhalb eines Geschäftsprozesses unter Umständen mehrfach erzeugt. Um sicher zu stellen, dass bei Verwendung einer Instanz der Klasse *SQLiteDAOAccessor* nur die benötigten Objekte genau einmal instantiiert werden, wurden die Attribute mit dem Schlüsselwort *lazy* versehen.

Sichtbarkeiten der Datenbankimplementierung: Ein Zugriff auf die Klasse *SQLiteDAOAccessor* ist auf das Paket *model* beschränkt, obwohl sie sich im Paket *model.database* befindet.

Dies ist möglich, da sich mit Scala Sichtbarkeiten auf übergeordnete Pakete einschränken lassen. Dadurch kann das Trait *ServiceAccess* die Klasse *SQLiteDAOAccessor* nutzen. Dagegen kann auf die konkreten DAO-Klassen nur innerhalb des Paketes *database* zugegriffen werden - außerhalb muss mit den DAO-Schnittstellen gearbeitet werden.

DAO-Schnittstellen und DAO-Klassen

Die DAO-Klassen implementieren die DAO-Schnittstellen aus dem Paket *business.repository*. Sie kapseln die SQL-Befehle, um die Geschäftsobjekte in der Datenbank abzulegen. Jede DAO-Klasse ist für eine bestimmte Klasse der Geschäftslogik und damit für eine Datenbanktabelle zuständig. Dabei ähnelte sich die Funktionalität der DAO-Klassen sehr. Bereits beim Entwickeln der Schnittstellen entstanden Redundanzen, weshalb ein generisches Trait *BaseDAO* entworfen wurde:

```
protected[model] trait BaseDAO[T <: {val id : Long}] {
  def foreach(fun : (T) ⇒ Unit): Unit
  def fetch(id : Long): T
  def fetchAll(): List[T]
  def create(obj : T): T
  def delete(obj : T): Boolean
  def update(obj : T): Boolean
}
```

Programmauszug 6.9: Generische Schnittstelle BaseDAO

Da der abstrakte Typ *T* die obere Grenze *{id:Long}* besitzt, kann das Trait mit Typen parametrisiert werden, die ein Attribut *id:Long* haben (obere Grenze: s. Abschnitt 2.4.4 auf Seite 23). Diese Art der Typisierung nennt sich auch "Duck Typing"[1]. Ohne die Verwendung eines strukturellen Typs wäre die Entwicklung einer Schnittstelle für die Fachklassen nötig gewesen.

Alle weiteren DAO-Schnittstellen beerben das Trait *BaseDAO*, wobei sie den Typ ihrer zugehörigen Fachklasse als Parameter verwenden. Einige von ihnen spezifizieren zusätzliche Methoden. Zum Beispiel wird im *TracePointDAO* eine Methode benötigt, die möglichst performant einen *TracePoint* in der Datenbank speichert. Das wird mit der Methode *createFast* realisiert.

1 Duck Typing: Nicht die Angehörigkeit zu Typen (Klassen/Schnittstellen) ist relevant, sondern lediglich das Vorhandensein von Funktionalität. "don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc"[Mar00]

```
protected[model] trait TracePointDAO extends BaseDAO[TracePoint] {
  def createFast(point : TracePoint): Long
  /*...*/
}
```

Programmauszug 6.10: Schnittstelle TracePointDAO

Das Trait *TracePointDAO* erbt alle weiteren Methoden des Trait *BaseDAO*. Die Implementierung für die SQLite-Datenbank ist die Klasse *SQLiteTracePointDao*. Für das Trait *BaseDAO* wurde eine abstrakte Klasse *SQLiteBaseDAO* implementiert, die von den konkreten DAO-Klassen beerbt wird.

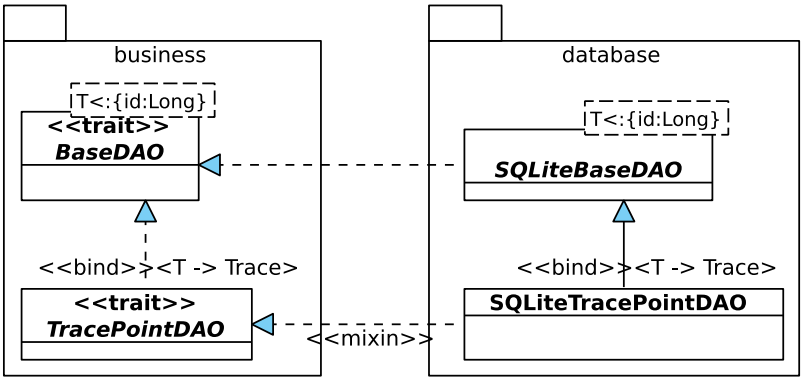

Abbildung 6.10: Das Trait BaseDAO

In den konkreten DAO-Klassen befinden sich nur für sie spezifische Methoden und Attribute. Alle den DAO-Klassen gemeine Funktionalität, zu der auch die Instantiierung des *SQLiteOpenHelper* gehört, sind in die abstrakte Klasse *SQLiteBaseDAO* ausgelagert worden.

Einsatz des Loan-Entwurfsmusters bei Datenbankzugriffen

In der Basis aller konkreten DAO-Klassen (*SQLiteBaseDAO*) finden sich zwei sehr wichtige Methoden, die für alle Datenbankzugriffe verwendet werden: *databaseWriteAccess* zum Schreiben und *databaseReadAccess* zum Lesen aus der Datenbank. Beim Zugriff auf die Datenbank sind immer die gleichen Schritte notwendig. Eine Datenbank muss geöffnet werden. Beim Auftreten einer Ausnahme oder nach regulärem Beenden des Zugriffes muss die Datenbankverbindung ordnungsgemäß geschlossen werden. Um Redundanzen aus den einzelnen Methoden der DAO-Objekten zu entfernen, wurde das “Loan Pattern” verwendet. Mit ihm kann beim Zugriff auf eine Ressource das korrekte Schließen sichergestellt werden, ohne jedes Mal einen “try finally”-Block implementieren zu müssen [Vgl. Loa06].

Die Umsetzung wird im Folgenden am Beispiel der Methode *databaseWriteAccess* erläu-

tert. Ähnlich wie die Methode *tryCatch* des *ActivityHelper* (Abschnitt 6.2.3 auf Seite 74) ist sie als Funktion höherer Ordnung definiert. Sie erwartet als Parameter eine Funktion, die ein Objekt vom Typ *SQLiteDatabase* als Parameter besitzt. Innerhalb der Methode *databaseWriteAccess* ist die Funktionalität für das Öffnen und ordnungsgemäße Schließen der Datenbank gekapselt. Dies wird durch das Applizieren der übergebenen Funktion innerhalb eines "try finally"-Blockes erreicht.

```
protected def databaseWriteAccess[A](fun : (SQLiteDatabase) ⇒ A ) : A = {
  val db = dbAccess.getWritableDatabase
  try {
    fun(db) // Resultat wird zurückgegeben
  } finally { db.close }
}
```

Programmauszug 6.11: Umsetzung des Datenbankzugriffs und das Loan-Pattern

In der als Parameter übergebenen Funktion kann mit der Instanz der Klasse *SQLiteDatabase* gearbeitet werden, die in der Methode *databaseWriteAccess* erzeugt und anschließend geschlossen wird.

Die Methode *databaseReadAccess* ist ähnlich umgesetzt. Sie erwartet in einer zweiten Parameterliste Angaben, die zum Lesen aus der Datenbank verwendet werden. Mit den Resultaten wird dann die übergebene Funktion appliziert.

Abschließende Bemerkung: Die Methoden der DAO-Klassen, die auf die Datenbank zugreifen, verwenden eine der beiden vorgestellten Methoden des Trait *BaseDAO*. Das Loan-Entwurfsmuster vermeidet Fehlerquellen und hält das Programm übersichtlicher.

6.3.2 Umsetzung der Geschäftslogik

Die Klassen und Schnittstellen der Geschäftslogik im Paket *tribosmap.business* sind in drei Subpakete aufgeteilt:

business.repository enthält die erläuterten DAO-Schnittstellen

business.domain enthält die persistenten Fachklassen

business.service enthält die Dienste, die mit den DAO-Schnittstellen zusammenarbeiten und vom Frontend verwendet werden

Kritik am Entwurf der Geschäftslogik

Zunächst wird eine grundsätzliche Kritik am Entwurf der Geschäftslogik geübt. Bedingt durch den Einsatz des DAO-Entwurfsmuster ist ein Teil der Geschäftslogik in den Diens-

ten zu finden und nicht in den eigentlichen Fachklassen. Die Geschäftslogik aus dem Domain-Modell in Dienste auszulagern, steht im krassen Widerspruch zu der Grundidee der objektorientierten Programmierung, zusammengehörige Daten und Operationen in Klassen zu kapseln. Eine Trennung führt schnell zu einer Art prozeduraler Programmierung. Es entsteht ein sogenanntes "Blutarmes Domain Modell", das Martin Fowler als Anti-Entwurfsmuster bezeichnet [Fow03].

Das Domain-Modell von TribOSMap ist nicht völlig "blutarm", da die Fachklassen noch Parameter validieren und Konstruktoren besitzen. Die Kritik richtet sich vornehmlich gegen zwei Methoden, die in den Diensten implementiert sind. Die eine exportiert die *Traces* und die andere importiert die TOSM-Landkarten. Diese Funktionalität wurde nicht in den Fachklassen umgesetzt, denn dafür muss auf die Datenbank zugegriffen werden. Das DAO-Entwurfsmuster sieht aber keinen Zugriff der Fachklassen auf die DAO-Objekte vor, wurde aber wegen der oben genannten Vorteile verwendet.

Persistenten Domain-Klassen

Aus den Fachklassen wurden die persistenten Domain-Klassen (z.B. *Marker*) abgeleitet. Der Umfang der Geschäftslogik in diesen Klassen ist relativ übersichtlich. Zudem wird durch die verwendeten sprachlichen Mittel von Scala die Anzahl der Programmzeilen weiter verringert.

Case-Klassen: Die Fachklassen sind als Case-Klassen umgesetzt. Alle Attribute werden im Konstruktor übergeben, deshalb muss eine Methode *equals* nicht implementiert werden. Zudem müssen für Attribute die mit der Modifizierbarkeit spezifiziert sind, keine "getter"-Methoden implementiert werden. Die Validierung der Konstruktorparameter wird im Klassenrumpf vorgenommen. Veränderliche Attribute werden innerhalb explizit definierter "setter"-Methoden validiert.

Serialisierung: Da die Geschäftsobjekte innerhalb der Oberfläche zwischen verschiedenen Activities ausgetauscht werden können, müssen sie alle die Schnittstelle *java.io.Serializable* implementieren. Damit können diese Objekte serialisiert und wieder deserialiserit werden [Ser06]. Dies ist notwendig, da jedes Activity wie bereits in Abschnitt 3.3.1 auf Seite 35 besprochen, in einem separaten Prozess läuft.

One-to-Many-Relationen: Es gibt einige persistente Fachklassen mit One-to-Many-Relationen. Zum Beispiel kann ein *TraceSegment* viele *TracePoint*-Objekte referenzieren. Die möglicherweise große Anzahl von Punkten eines Segmentes würde die Liste des *TraceSegment*-Objektes zu groß für den Speicher der Geräte werden lassen. Daher wird

keine Liste von *TracePoints* erzeugt, sondern die Klasse *TracePoint* hat ein Attribut, das dem Wert der Id des *TraceSegment* entspricht. Im derzeit entwickelten Prototyp sind auch die One-to-Many-Relationen zwischen den Klassen *Trace* und *TraceSegment* und zwischen *GeoMap* und *GeoreferencedPixel* auf diese Art und Weise umgesetzt worden. Bei einer Weiterentwicklung der Software sollte dieser Sachverhalt nochmals untersucht und gegebenenfalls eine andere Lösung implementiert werden.

Dienste der Geschäftslogik und das Cake-Entwurfsmuster

Nach Möglichkeit sollen die Dienste und die konkrete DAO-Implementierung leicht austauschbar sein. Dies ermöglicht nicht zuletzt ein einfacheres Testen der Anwendung. Dafür ist es vor allem notwendig, die Dienste und den *DAOAccessor* zentral zu instantiieren.

Konventioneller Lösungsvorschlag für die Instantiierungen der Dienste: Eine Möglichkeit wäre die Umsetzung eines global verfügbaren Objektes, mit dessen Hilfe die Activities die gewünschten Dienste instantiieren. Deshalb wurde zunächst an die in Abbildung 6.11 dargestellte Umsetzung gedacht.

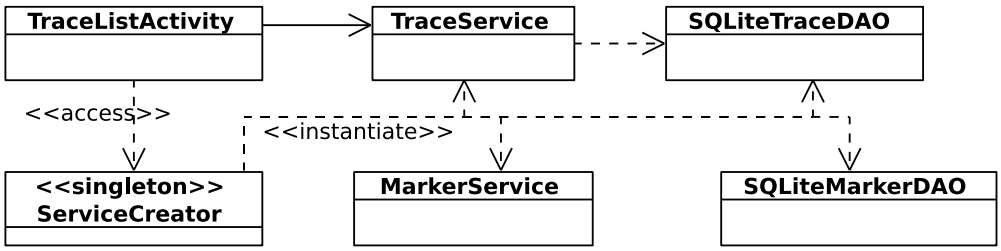

Abbildung 6.11: Idee: Serviceinstantiierung mit Singleton-Objekt

Das *TraceListActivity* kann die *Traces* exportieren. Dazu muss dieses Activity zunächst den *ServiceCreator* um eine Instanz der Klasse *TraceService* bitten. Im *TraceService* ist die Funktionalität für das Exportieren eines *Trace* implementiert. Für das Laden dieses *Trace* aus der Datenbank benötigt der Dienst ein Objekt des Typs *TraceDAO*. Die derzeitig einzige konkrete Implementierung dieses Trait ist die Klasse *SQLiteTraceDAO*. Diese benötigt ein Context-Objekt, das vom Activity dem *ServiceCreator* übergeben wird. Damit kann dieser für eine Instantiierung der notwendigen DAO-Objekte sorgen und sie an die Dienste weitergeben.

Das Exportieren eines *Markers* soll in einem Dienst (*MarkerService*) implementiert sein, um die Verantwortlichkeiten der Dienste möglichst klar nach den Anwendungsfällen aufzuteilen. Beim Export eines *Trace* müssen aber auch zugehörige *Marker* exportiert werden, was nicht redundant in *MarkerService* und *TraceService* implementiert sein soll. Das Exportieren von *Markern* in eine Basisklasse auszulagern oder das Strategie-Entwurfsmuster zu

nutzen, würde die Komplexität unverhältnismäßig steigern. Folglich benötigt der *TraceService* einen Zugriff auf die Funktionalität des *MarkerService*. Er kann diesen aber nicht ohne ein Context-Objekt instantiieren, denn der *MarkerService* benötigt wiederum das *MarkerDAO* und dieser seinerseits das Context-Objekt. Damit ist es schwierig die Geschäftslogik von der Android-API und der konkreten DAO-Implementierung zu trennen.

Lösung für die Dienste-Instantiierungen mit Traits: Eine elegante Lösung für das Problem der Instantiierung der Dienste wird durch den Einsatz der Traits möglich. Anstelle der Implementierung des Singleton-Objektes *ServiceCreator* (Abbildung 6.11) wurde das Trait *ServiceAccess* implementiert, das die Instantiierung der Dienste und des konkreten *DAOAccessor* zentralisiert. Dies ist möglich, da das Trait *ServiceAccess* ein definiertes "socket" auf ein Context-Objekt besitzt.

Um den Diensten Zugriff auf andere Dienste und den *DAOAccessor* zu ermöglichen, werden diese Abhängigkeiten injiziert. Was nicht mit "setter"-Methoden oder einem Konstruktor geschieht, sondern mit Hilfe der Traits und des sogenannten "Cake Pattern" [Vgl. Cak06, Cake Pattern]. Jeder der Dienste ist als eine innere Klasse in einem Trait implementiert. Diese Kombination aus Trait und Dienst wird im Folgenden Dienst-Komponente genannt. Traits definieren abstrakte Attribute ("sockets"), die andere Dienste referenzieren. In den inneren Klassen, also den eigentlichen Diensten, kann auf diese Attribute zugegriffen werden. Die Instantiierung der Dienste findet zentral im Trait *ServiceAccess* statt, der die genannten Traits der Dienst-Komponenten einmischt. Somit ist die Implementierung der Dienste von ihrer Instantiierung unabhängig.

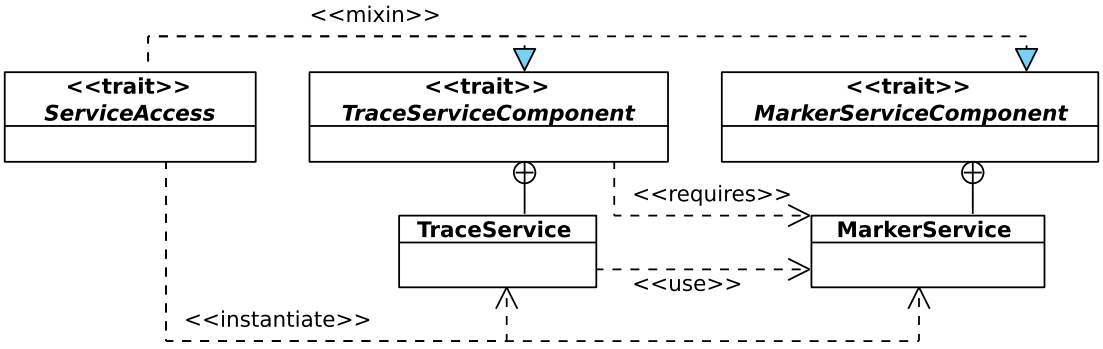

Abbildung 6.12: Dienste der Geschäftslogik und das Cake-Pattern

Der *SQLiteDAOAccessor* wird ebenfalls im Trait *ServiceAccess* instantiiert. Die Dienst-Komponenten sind durch definierte "sockets" lediglich von der Schnittstelle (Trait) *DAOAccessor* abhängig. Die "sockets" der Dienst-Komponenten werden durch das Einmischen in das Trait *ServiceAccess* befriedigt. Das "socket" zum Context-Objekt, das im Trait *ServiceAc-*

cess definiert ist, wird letzten Endes durch das Einmischen in ein Activity befriedigt.

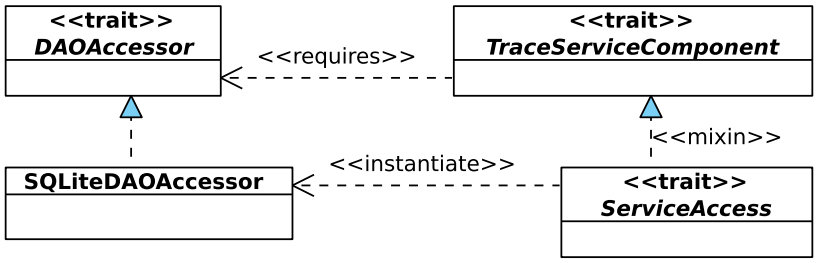

Abbildung 6.13: Dienste und DAO-Klassen

Ähnlich wie die DAO-Objekte des *SQLiteDAOAccessor* sind alle Dienste innerhalb des Trait *ServiceAccess* in *lazy*-Attributen deklariert. Damit werden nur die notwendigen Dienste erzeugt, die nur die von ihnen benötigten DAO-Objekte anfordern.

```
trait ServiceAccess extends
   TraceServiceComponent with MarkerServiceComponent with /*...*/ {
   protected val context: Context //Socket
   lazy val markerService = new MarkerService()
   lazy val traceService = new TraceService()
   /*...*/
   private[model] lazy val daoAccessor: DAOAccessor =
      new SQLiteDAOAccessor(context, "tribosmap", 1)
}
```

Programmauszug 6.12: ServiceAccess - Objektinstantiierung im Cake Pattern

Wie dem Programmauszug 6.12 zu entnehmen ist, ist die Sichtbarkeit des Attributes *aaoAccessor* auf das Paket *model* beschränkt. Dadurch erhält das Activity, das das Trait *ServiceAccess* einmischt, keinen Zugriff.

Um die Datenbankimplementierung auszutauschen, muss lediglich das Trait "Service-Access" verändert werden. Auch das Austauschen einzelner Dienste ist leicht möglich. Wobei dafür zunächst eine Schnittstelle für diese Dienste programmiert werden müsste. Dies ist bis jetzt nicht geschehen, wäre aber mit überschaubarem Aufwand möglich.

Abschließende Bemerkung: Das Injizieren mit Traits durch die Nutzung des Cake-Entwurfmusters ist eine elegante Möglichkeit, unabhängige, leicht austauschbare Komponenten zu schaffen. Erreicht wird dies durch die Kombination eines Trait mit einer inneren Klasse. Die Abhängigkeiten der inneren Klassen zueinander werden durch das Einmischen der Traits in eine zentrale Komponente befriedigt. Damit können die konkreten Implementierungen von Klassen, die sich gegenseitig benötigen, leicht ausgetauscht werden können.

Aufruf von Geschäftslogik über Geschäftsobjekte mit impliziten Definitionen

Es ist bereits kritisch das fast blutarme Domain Modell beschrieben worden, dass zur Trennung der Geschäftslogik von den zugehörigen Klassen führt. Auch die Verwendung der Geschäftslogik ist umständlich und nicht intuitiv, da nicht mit den Fachklassen selbst, sondern mit den Diensten gearbeitet werden muss. Es ist unter anderem notwendig, die Dienste zu verwenden, um die Geschäftsobjekte zu erzeugen oder zu speichern. Dazu ein Beispiel, bei dem mit Hilfe eines Objektes der Klasse *TraceService* ein Trace in eine Datei exportiert wird:

```
val trace : Trace = /*...*/
traceService.export(trace, targetFile)
```

Programmauszug 6.13: Aufruf der Geschäftslogik mittels der Dienste

Die eigentlich intuitiv in der Klasse Trace erwartete Methode *export* ist im *TraceService* implementiert. Damit beeinflusst das blutarme Domain-Modell auch die Benutzung der Geschäftslogik. Besser wäre es, die Methode *export* direkt auf dem Trace-Objekt aufrufen zu können:

```
val trace : Trace = /*...*/
trace.export(targetFile)
```

Programmauszug 6.14: Aufruf der Geschäftslogik mittels der Geschäftsobjekte

Deshalb wurden innerhalb der Dienst-Komponenten implizite Methoden definiert, die an die Objekte der Fachklassen neue Methoden binden, die die eigentlichen Methoden der Dienste aufrufen. Durch das Einmischen der Dienst-Komponenten in das Trait *ServiceAccess* sind diese impliziten Definitionen in den Klienten des Trait, also den Activities, verfügbar.

Abschließende Bemerkung: Implizite Definitionen sind ein mächtiges Werkzeug. Sie haben allerdings auch ihre Nachteile. Beim Lesen des Programms könnte man denken, dass die Methoden zum Speichern und Lesen der Objekte direkt in den Fachklassen implementiert sind. Die Dokumentation und der Quellcode der Fachklassen enthält dazu aber keine Informationen. Das kann zu schwer verständlichem (“magischen”) Quellcode führen. Deshalb sollte besonders in größeren Projekten von diesem Werkzeug nur vorsichtig Gebrauch gemacht werden.

XML-Repräsentation der Fachklassen

XML ist sprachlicher Bestandteil von Scala. Die Anwendung TribOSMap muss XML-Dokumente aus den Trace-Objekten erzeugen. Die einzelnen persistenten Fachklassen

besitzen Methoden, die eine XML-Repräsentation der Objekte erzeugen. Im folgenden Programmauszug wird eine dieser Methoden aus der Klasse *TraceSegment* vorgesellt:

```
def xml: Node = {
<trkseg >
  <extensions> <segmentSize>{ size }</segmentSize> </extensions>
</trkseg> }
```

Programmauszug 6.15: XML-Repräsentation eines TraceSegment

Wie dem Auszug zu entnehmen ist, werden XML Ausdrücke direkt verwendet, die der Compiler erkennt. Der Wert eines validen XML-Ausdruckes ist vom Typ *scala.xml.Node*. Innerhalb eines XML-Ausdruckes können anonyme Funktionen eingefügt werden, deren Resultate *String*- oder *Node*-Sequenzen oder Objekte des Typs *String* oder *Node* sind.

Äußerst elegant wäre es, wenn innerhalb des XML-Ausdrucks in der Klasse *TraceSegment* die XML-Ausdrücke der einzelnen refrenzierten *TracePoint*-Objekte enthalten wären. Angenommen, die Klasse *TraceSegment* besäße eine Sequenz der Punkte (*tracePoints*). Dann könnte die gezeigte Methode wie folgt definiert werden:

```
def xml: Node = {
<trkseg>
  <extensions> <segmentSize>{ size }</segmentSize> </extensions>
  { tracePoints.map(point ⇒ point.xml) }
</trkseg> }
```

Programmauszug 6.16: Sequenz innerhalb eines XML-Ausdruckes

Innerhalb des XML-Ausdruckes würde jetzt die Sequenz der Punkte mit Hilfe der Methode *map* aus der Klasse *Seq* zu einer Sequenz von *Node*-Objekten umgewandelt werden. Dies ist wie erwähnt wegen der zu großen Datenmenge so nicht umgesetzt worden.

Die Funktionalität für den Export der Traces in den Diensten zerlegt deshalb die *Node*-Objekte der Klassen *TraceSegment* und *Trace* in XML-Tags. Innerhalb der Tags werden die die beinhalteten Objekte der Reihe nach aus der Datenbank gelesen und ihre XML-Repräsentation in die Datei geschrieben.

6.3.3 Mathematische Fachklassen

Im Paket *tribosmap.model.math* sind die mathematischen Fachklassen definiert, die Distanzen geografischer Positionen repräsentieren und Algorithmen zur Verfügung stellen.

Implementierung der Entfernungen mit verschiedenen Maßeinheiten

TribOSMap soll dem Nutzer die Länge eines zurückgelegten Weges in unterschiedlichen Maßeinheiten präsentieren. Um dies zu realisieren, wurde zunächst daran gedacht, für jede Maßeinheit eine separate Klasse zu schaffen, die die Konvertierungs-Methoden enthält. Das hätte allerdings zur Folge, dass bei der Einführung einer neuen Maßeinheit alle bestehenden Klassen um eine neue Konvertierungs-Methode ergänzt werden müssten. Die Anzahl der nötigen Umrechnungsfunktionen bei einer Anzahl n der Maßeinheiten ist $n * (n-1)$. Zum Beispiel wären dann für 10 Maßeinheiten 90 Funktionen zu implementieren. Aus diesem Grund musste eine bessere Lösung gefunden werden.

Grundsätzlich gehören verschiedene Maßeinheiten einem speziellen Einheitensystem an. Es gibt zum Beispiel das metrische Einheitensystem und das amerikanische Maßsystem für Distanzen. Eine Umwandlung von Einheiten zwischen den verschiedenen Systemen wird durch eine vorherige Umwandlung in eine allgemeine Basis-Maßeinheit umgesetzt. Innerhalb des Einheitensystems wird direkt umgewandelt.

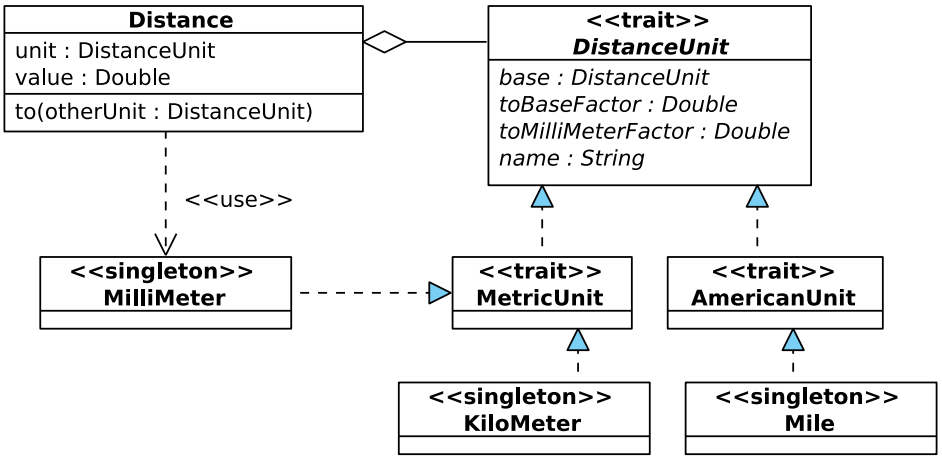

Abbildung 6.14: Maßsysteme und Maßeinheiten

Zunächst wurde eine Schnittstelle *DistanceUnit* spezifiziert, die von allen Maßeinheiten implementiert wird. Diese verlangt das Vorhandensein eines Attributes *toMilliMeterFactor* mit dessen Hilfe Einheiten zwischen den verschiedenen Systemen umgewandelt werden. Die Einheit *MilliMeter* ist somit die grundlegende Basis für diese Umrechnungen. Zur Umwandlung innerhalb des Einheitensystems wurden in der Schnittstelle *DistanceUnit* die Attribute *toBaseFactor* und *base* spezifiziert. Die Basis (*base*) ist die Maßeinheit, die bei Umrechnungen innerhalb eines Systems als Grundlage verwendet wird. Die eigentlichen Umrechnungen sind in der Klasse *Distance* implementiert. Zur Instantiierung muss ein Wert *value* und eine Maßeinheit *unit* im Konstruktor übergeben werden.

Die einzelnen Maßeinheiten sind als Singleton Objekte umgesetzt. Sie implementieren

das ihrem Maßsystem zugehörige Trait. Zur Veranschaulichung hier die Implementierung des amerikanischen Systems, der Maßeinheit Meile und der Maßeinheit Inch.

```
trait AmericanUnit extends DistanceUnit {
  val toMilliMeterFactor = 25.4; val base = Inch
}
object Mile extends AmericanUnit {
  val name = "mi"; val toBaseFactor = 63360.0
}
object Inch extends AmericanUnit {
  val name = "in"; val toBaseFactor = 1
}
```

Programmauszug 6.17: Amerikanisches Maßsystem mit Maßeinheiten

Weitere Maßeinheiten des amerikanischen Systems können nun genauso wie das Objekt *Mile* implementiert werden. Zur Umsetzung eines weiteren Maßsystems ist lediglich die Implementierung eines neuen von *DistanceUnit* abgeleiteten Trait und einer Maßeinheit für die Basis notwendig.

Abschließende Bemerkung: Die Nutzung von Singleton Objekten für die Definition der Maßeinheiten ermöglicht eine gute Erweiterbarkeit. Damit lassen sich neue Einheiten hinzufügen, ohne bestehende Klassen bzw. Dateien verändern zu müssen.

Funktionen höherer Ordnung und das Rechnen mit Distanzen

Mit Instanzen von *Distance* sollen arithmetische Berechnungen durchgeführt werden können (z.B. Mittelwert von Distanzen). Dabei muss jedesmal das Einheitensystem der beteiligten Instanzen in Erfahrung gebracht werden, um die eventuell notwendige Umwandlung vornehmen zu können. Ist dies notwendig, so muss geprüft werden, ob sie innerhalb des Maßsystems oder zwischen verschiedenen Systemen stattfinden muss. Es wäre unschön, ein metrisches Maß zurückzugeben, obwohl ausschließlich mit amerikanischen Maßen gerechnet wurde.

Um Redundanzen bei der Implementierung dieser arithmetischen Methoden zu vermeiden, wurde eine Methode *doFunBetweenTwo* implementiert, die wenn notwendig, eine Umwandlung durchführt und mit einer Funktion parametrisiert werden kann, die die eigentliche Berechnung vornimmt.

```
def doFunBetweenTwo[T](other : Distance)(fun : (Distance, Distance) ⇒ T) = {
  if(other.unit == unit) fun(this, other)
  else if(other.unit.base == unit.base) fun(this.to(unit.base), other.to(unit.base))
  else fun(this.to(MetricUnit.MilliMeter), other.to(MetricUnit.MilliMeter))
}
```

Programmauszug 6.18: Funktion höherer Ordnung zur Anpassung zweier Maße

Die gezeigte Methode entscheidet nun, ob für eine Berechnung zunächst eine Umwandlung vorgenommen werden muss. Dazu vergleicht sie die Maßeinheiten der eigenen Instanz und des übergebenen Objektes (*other*). Danach wird die übergebene Funktion *fun* mit den eventuell umgewandelten Distanzen aufgerufen.

Es kann zum Beispiel eine Funktion zum Addieren von Distanzen implementiert werden. Durch die Verwendung der Funktion *doFunBetweenTwo* ist sichergestellt, dass die eigentlichen Werte der Distanzen (*value*) auch addiert werden können:

```
def +(other : Distance) = doFunBetweenTwo(other) {
  (a, b) ⇒ Distance(a.value + b.value, a.unit ) }
```

Programmauszug 6.19: Addition von Distanzen

Um die Nutzung weiterer gleichartig definierter arithmetischer Methoden zu veranschaulichen, hier die Methode *average*, die den Mittelwert verschiedener Distanzen berechnet, dabei wurde neben der Addition auch die Division von Distanzen verwendet:

```
def average(others : Seq[Distance]) =
  others.reduceLeft(_ + _) + this / (others.length + 1)
```

Programmauszug 6.20: Berechnung des Mittelwertes von Distanzen

Implizite Definitionen zum Instantiieren von Distanzen

Besonders bei der Entwicklung der Unit-Tests ergab sich der Wunsch, Distanzen in möglichst kurzer Schreibweise definieren zu können. In der Mathematik rechnet man mit Maßeinheiten ähnlich wie mit Zahlen. An die Zahl wird dabei lediglich ein Kürzel für die Maßeinheit angehängt. “1 m” steht für einen Meter. Wie kann nun eine ähnliche Schreibweise umgesetzt werden?

Wieder wurde auf die impliziten Definitionen zurückgegriffen. Für jede Maßeinheit wurde je eine zusätzliche Methode den Klassen *Int* und *Double* hinzugefügt, die den jeweiligen Konstruktor der Maßeinheit aufruft.

```
class NumberToDistance(value: Double){
  def km = Distance(value, KiloMeter)
  def mi = Distance(value, Mile)
  def ft = Distance(value, Foot)
  /*...*/
}
implicit def intInDistance(value: Int) = new NumberToDistance(value)
implicit def doubleInDistance(value: Double) = new NumberToDistance(value)
```

Programmauszug 6.21: Implizite Erzeugung von Distanz-Objekten

Wenn diese impliziten Definitionen importiert werden, kann zum Beispiel folgender Ausdruck geschrieben werden:

```
val distance = 1.mi - 5280.ft + 1.km
```

Programmauszug 6.22: Rechnen mit Distanzen

Damit wurde eine Domain-spezifische Sprache zum Rechnen mit Distanzen geschaffen, die eine sehr komfortable Entwicklung umfangreicher Tests ermöglichte.

Entwicklung einer "Fabrikmethode" für geodätische Daten

In der Anwendung werden zwei verschiedene Arten von Koordinatensystemen zur Beschreibung der geografischen Lage verwendet. Zum einen das geografische Koordinatensystem (*GeographicCoOrdinatePair*) und zum anderen das UTM-Koordinatensystem (*UtmCoOrdinates*). Für die Umrechnung zwischen diesen Koordinatensystemen muss das geodätische Datum der UTM-Koordinaten berücksichtigt werden, dessen Name zum Beispiel in der TOSM-XML-Datei als Zeichenkette angegeben ist. Deshalb ist eine Abbildung von diesem Namen auf ein Datum-Objekt notwendig. Um eine gute Erweiterbarkeit der Anwendung um neue geodätische Daten sicherzustellen, wird eine Fabrikmethode genutzt. Das Singleton-Objekt *Datum* setzt diese in ihrer *apply*-Methode um. Diese Methode liefert die *Datum*-Objekte, die eine Schnittstelle *Datum* implementieren. Damit ist eine Fabrikmethode in Scala auf sehr einfache Art und Weise umgesetzt.[1]

```
object Datum { def apply(name: String): Datum = name match {
  case WGS84Datum.name ⇒ WGS84Datum
}}
```

Programmauszug 6.23: Fabrikmethode zur Erzeugung der geodätischen Daten

1 Im Gegensatz zum Entwurfsmuster "Fabrikmethode" wird keine abstrakte Erzeuger-Klasse verwendet.[Gam95, Seite 121 ff.]

6.3.4 Einsatz von Akteuren zum Laden der Kartenteile

Das Laden der Karten soll die Oberfläche möglichst nicht blockieren. Es müssen die Operationen für das Laden also nebenläufig umgesetzt werden. Nun kann es sein, dass die Oberfläche Kartenteile anfordert, die kurz darauf nicht mehr benötigt werden, da der sichtbare Bereich der Karte schon wieder völlig andere Kartenteile zeigt. Die Idee ist es, grundsätzlich nur das gleichzeitige Laden einer maximalen Anzahl von Kartenteilen zu erlauben. Die Aufträge für das Laden von Kartenteilen werden in einem Stapel gespeichert. Ist ein Auftrag beendet, wird die Oberfläche informiert und erhält das neue Kartenteil für die Darstellung. Es lassen sich folgende Module identifizieren: die Aufträge, der Stapel mit neuen Aufträgen, die Oberfläche und die Auftragsausführer.

Die Komponente der Oberfläche, die diese Aufträge initiiert, ist das bereits vorgestellte *MapActivity*. Es benutzt dabei einen spezifischen Algorithmus, der in Abhängigkeit von Typ und Projektion der Karte berechnet, wann welcher Kartenbereich wo sichtbar ist. Die Aufträge werden mit Hilfe dieser Algorithmen vom Activity instantiiert und an eine Komponente gereicht, die den Auftragsstapel verwaltet. Dort werden die einzelnen nebenläufigen Threads (Prozesse) gestartet, die die Aufträge bearbeiten und im Anschluss das Activity mit dem geladenen Bild versorgen.

Lösungsvorschlag für die Umsetzung mit Threads

Eine zentrale Instanz der Klasse *ImageLoader* verwaltet die für das Laden zuständigen *LoaderThreads*. Das *MapActivity* veranlasst Aufträge, indem es Instanzen der Klasse *TileJob* mit den notwendigen Informationen in Form eines *Tile*-Objektes parametrisiert und an den *ImageLoader* übergibt. Ist ein Kartenteil geladen, wird die Oberfläche aktualisiert.

In Abbildung 6.15 auf der nächsten Seite ist ein möglicher Ablauf dargestellt. Zunächst hat das *MapActivity* festgestellt, dass ein neues Kartenteil benötigt wird. Es erzeugt nun einen Auftrag (*Job*) zum Laden dieses Kartenteiles. Der *ImageLoader* startet einen neuen Thread (*LoaderThread*), da die maximale Anzahl von laufenden Threads (*loaderCount*) noch nicht erreicht wurde. Nachdem der *LoaderThread* das Bild geladen, ein Bitmap erzeugt und die Aktualisierung der Oberfläche eingeleitet hat, fragt dieser den *ImageLoader* nach weiteren Aufträgen. Im gezeigten Sequenzdiagramm gab es keine weiteren Aufträge, deshalb wurde die Variable, die die Anzahl der gestarteten Threads kennzeichnet, decrementiert und der Thread beendet.

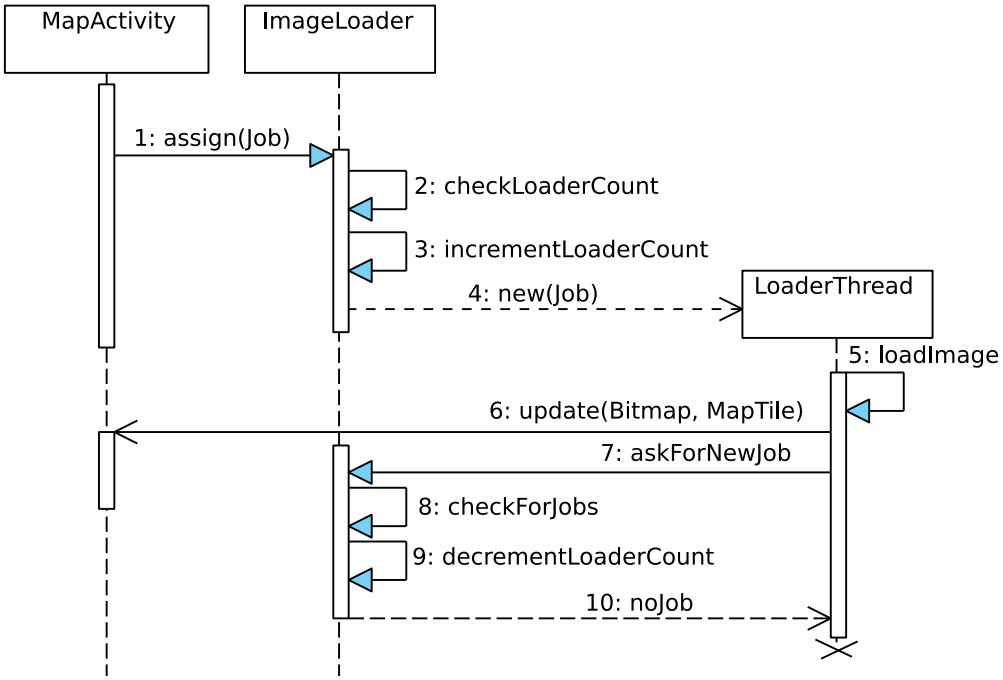

Abbildung 6.15: ImageLoader mit Threads

Durch die Threads wird der Zustand des *ImageLoader* nebenläufig verändert. Aus diesem Grund muss ein Teil der Funktionalität im *ImageLoader* Thread-sicher umgesetzt werden. Zum einen muss ein Thread-sicherer Datencontainer als Stapel für die Aufträge implementiert werden. Zum anderen muss eine Thread-sichere veränderliche Variable existieren, die die Anzahl der noch in Ausführung befindlichen Threads beinhaltet. Auch bei der Implementierung der Methoden müssen synchronisierte Blöcke implementiert werden, um den korrekten Ablauf sicherzustellen.

Akteure für nebenläufige Systeme

Threads sind nur eine Möglichkeit, mit Scala nebenläufige Funktionalität umzusetzen. Eine Andere ist es, die Actor-Bibliothek der API zu nutzen. Akteure arbeiten wie herkömmliche Threads nebenläufig. Zur Kommunikation mit einem Akteur sind keine gemeinsamen Daten notwendig, stattdessen wird ein sogenanntes Postfach genutzt. Das Kommunikationsmodell ist asynchron, deshalb werden der Aufrufer und der Akteur durch die Übermittlung von Nachrichten nicht blockiert. Auch wenn der Akteur gerade beschäftigt ist, können neue Nachrichten an ihn übermittelt werden. Sobald der Akteur eine Nachricht bearbeitet hat, wird er sich die nächste Nachricht aus dem Postfach vornehmen. Als Reaktion auf eine Nachricht kann er definierte Methoden ausführen, andere Akteure starten bzw. kontaktieren oder sich beenden.

Umsetzung mit Akteuren

Anstelle des Einsatzes von Threads (*LoaderThreads*) ist das Laden der Bilder mit Hilfe von Akteuren umgesetzt. Diese Akteure werden von nun an *Loader* genannt. Sie lösen aber nicht das Problem des nebenläufigen Zugriffes auf den Zustand der Klasse *ImageLoader*.

Die Idee ist, einen zentralen Akteur zu implementieren, der diesen Zugriff überflüssig macht. Dieser Akteur wird im Folgenden *Job-Center* genannt. Er verwaltet die *Jobs* (Aufträge). Die Anzahl der *Loader* wird von vornherein beschränkt. Dazu wird gleich bei der Instantiierung der Klasse *ImageLoader*, eine festgelegte Anzahl von *Loadern* initialisiert. Diese werden in einer Liste gespeichert, auf die das *Job-Center* Zugriff hat. Solange die Instanz der Klasse *ImageLoader* existiert, werden keine weiteren Akteure erzeugt, allerdings auch keine Akteure terminiert. Die Aufträge werden zunächst in ein Postfach des *Job-Centers* gesteckt. Entnimmt das *Job-Center* eine Nachricht (*Job*) aus seinem Postfach, schaut es solange in die Postfächer der *Loader*, bis er ein leeres findet. In dieses wird er den erhaltenen *Job* als neue Nachricht packen.

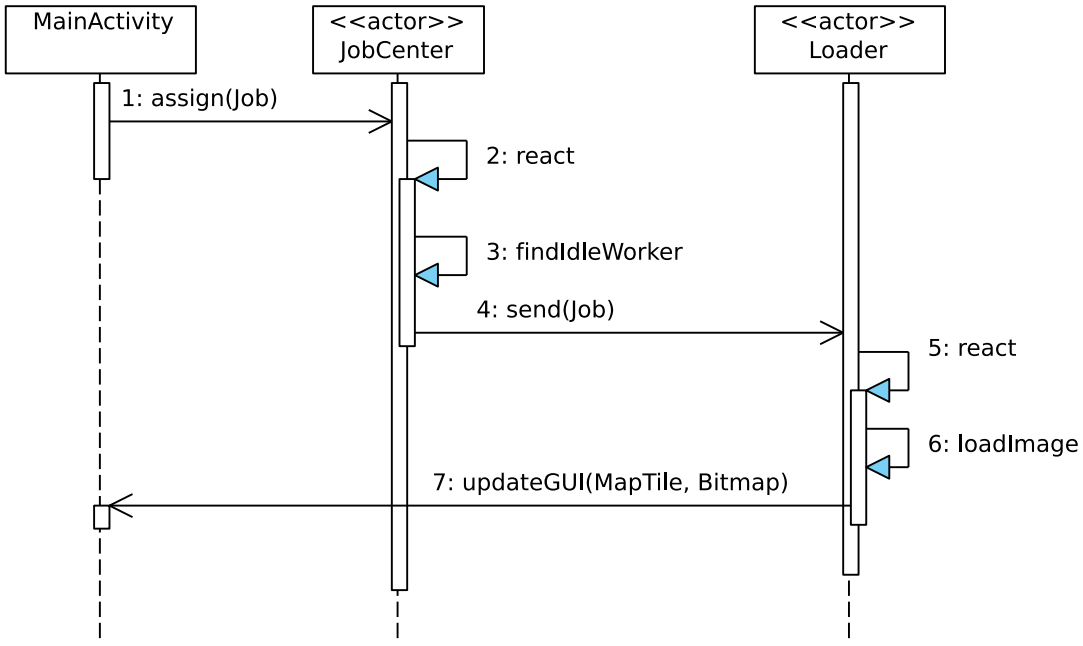

Abbildung 6.16: ImageLoader mit Akteuren

Der Abbildung 6.16 ist in erster Linie zu entnehmen, dass es keine synchronen Nachrichten zwischen dem *Job-Center* und einem *Loader* gibt. Es muss dementsprechend auch keine Thread-sichere Implementierung des Stapels erfolgen.

Das herkömmliche Postfach der Akteure ist als Warteschlange (Queue) umgesetzt. Für das *Job-Center* wurde die Implementierung angepasst. Dessen Postfach wurde als Stapel realisiert, wobei die Anzahl der Elemente begrenzt ist. Deshalb werden immer die neuesten Nachrichten, also die neuesten *Jobs*, zuerst bearbeitet, sehr alte werden ignoriert.

Abschließende Bemerkung

Die Actor-Bibliothek in Scala ist mit Java-Threads umgesetzt. Sie ermöglicht ein sehr einfaches Programmieren von nebenläufigen Operationen. Durch das asynchrone Kommunikationsmodell lassen sich synchronisierte Blöcke vermeiden. Das Implementieren solcher synchronisierter Blöcke ist fehleranfällig und diese Fehler wären schwer zu rekonstruieren. Die Actors-Bibliothek in Scala bietet eine sehr komfortable API für die Programmierung an.

6.4 Seiteneffekte in der Anwendung

Grundsätzlich ist bei der Entwicklung darauf geachtet worden, Funktionen und Komponenten möglichst frei von Zuständen zu implementieren. Da Scala über die notwendigen Mittel zur funktionalen Programmierung verfügt, war dies zumeist auch einfach und elegant möglich. Komplexe Fehler, die abhängig von bestimmten Zuständen auftreten, werden somit vermieden und das Testen der Anwendung wesentlich vereinfacht.

Gerade bei den zum Teil relativ komplexen Algorithmen, wie zum Beispiel der Umwandlung von UTM- in geografische Koordinaten, ist ein einfaches Testen wünschenswert. Aus diesem Grund sind die Klassen des Paketes *model.math* alle zustandslos implementiert. Dabei werden Performanceeinbußen in Kauf genommen, die durch das Erzeugen neuer Objekte entstehen. Mit veränderlichen Attributen könnte auf einige dieser Objektinstantiierungen verzichtet werden, da dann Objekte wiederverwendbar sind.

Genauso wäre es möglich, die persistenten Fachklassen zustandsfrei zu realisieren. Es wird allerdings als intuitiver angesehen, wenn veränderliche Felder der Datenbank sich auch durch veränderliche Attribute in den Fachklassen widerspiegeln. Dies war eine bewusste Designentscheidung, denn die persistenten Fachklassen werden in weiten Teilen der Anwendung verwendet. Dadurch sind alle diese Teile nun ebenfalls mit veränderlichen Zuständen versehen.

Auch beim Arbeiten mit Dateien sind veränderliche Variablen und Datencontainer verwendet worden. Zum Beispiel wird für das Laden von Binärdaten aus der TOSM-Datei in der Klasse *OfflineImageLoadingJob* ein Array verwendet. Theoretisch wäre es möglich gewesen, die Klasse *List* der Scala-API zu verwenden. Diese ist als verkettete Liste implementiert und unveränderlich. Bei den großen Datenmengen der Bilder in der TOSM-Datei wäre die Nutzung einer solchen unveränderlichen Liste schlichtweg unperformant, denn bei jeder Erweiterung der Liste würde diese kopiert werden. Stattdessen wird für das Lesen der TOSM-Dateien die Klasse *java.io.RandomAccessFile* genutzt, die performante Möglichkeiten besitzt, Daten blockweise (in Arrays) zu lesen.

6.5 Zusammenfassung

In diesem Abschnitt wurden interessante Details der Entwicklung vorgestellt. Es lässt sich feststellen, dass die sprachlichen Möglichkeiten von Scala das Implementieren von Entwurfsmustern oft wesentlich vereinfachen oder gar unnötig machen. Gerade die Traits ermöglichen eine leicht erweiterbare und gut strukturierte Architektur. Die funktionalen Fähigkeiten sind vor allem für Details bei der Implementierung interessant. Sie vermindern Redundanzen durch die Möglichkeit der Nutzung von Funktionen als Parameter.

7 Unit-Tests mit Scala und Android

Um Anwendungen auf mögliche Fehler zu testen, können diverse Werkzeuge verwendet werden. Aus der Java-Welt sind besonders die Frameworks TestNG [Beu] und JUnit [Jun] bekannt. Beide sind in Java implementiert und werden vornehmlich für das Testen von Java-Anwendungen verwendet. Ein weiteres in Scala implementiertes Werkzeug ist ScalaTest [Scab]. Dieses ermöglicht das Implementieren von Tests unter Ausnutzung der sprachlichen Möglichkeiten von Scala.

Das grundsätzliche Prinzip dieser Werkzeuge ist immer gleich. Innerhalb spezieller Testmethoden werden Teile der entwickelten Anwendung verwendet und überprüft, ob sie wie gefordert arbeiten. Dazu werden Methoden von Objektinstanzen der zu testenden Anwendung mit bestimmten Parametern appliziert und im Anschluss das Resultat mit angegebenen Werten verglichen. Die Testklassen, also die Klassen mit den implementierten Testmethoden, werden von den Frameworks verwendet, um die Tests auszuführen. Die Informationen über erfolgreiche oder fehlgeschlagene Tests werden dann ausgegeben.

Im Folgenden wird gezeigt, wie die genannten Werkzeuge für die Entwicklung von Tests mit Scala verwendet werden können. Dabei wird darauf eingegangen, ob sie in der Android-Plattform einsetzbar sind. Zuletzt wird die Architektur der für TribosMap entwickelten Tests dargestellt.

7.1 Auswahl des Testbeispiels

In diesem Kapitel werden als Beispiel die Tests der Klasse *CoordConverter* erläutert. Diese Klasse, die für die gegenseitigen Konvertierungen von UTM und geografischen Koordinaten verantwortlich ist, besitzt keine veränderlichen Zustände.

Die Testdaten bestehen aus Tupeln mit je zwei Elementen: einerseits den geografischen und andererseits den zugehörigen UTM-Koordinaten. Ein Element des Tupels wird als Parameter für die zu testende Methode verwendet und das Resultat mit dem anderen Element verglichen.

Bei diesem Vergleich müssen kleinere Abweichungen erlaubt werden, da das Rechnen mit Fließkommazahlen zu Rundungsfehlern führt. Aus diesem Grund wurde eine spezielle

Methode *alwaysEqual* in den beiden Koordinatenklassen implementiert. Sie prüft, ob Koordinaten ähnlich genug sind, um zu entscheiden ob der Test korrekt verlief oder nicht.

Die Variable *converter*, die in den folgenden Tests verwendet wird, referenziert eine Instanz der Klasse *CoordConverter*. Die Methode *latLonToUtm* dieser Klasse führt die Konvertierung von geografischen in UTM-Koordinaten durch.

7.2 TestNG

TestNG nutzt für die Implementierung der Tests Annotationen. Die Testklassen in TestNG müssen keinerlei Klassen beerben. Scala verfügt über Annotationen als Sprachbestandteil. Somit ist die Implementierung der Tests in Scala möglich.

Mit TestNG lassen sich Tests über eine Menge von Werten definieren. Dazu wird eine Methode, die ein zweidimensionales Objekt-Array (*Object[][]*) zurückgeben muss, mit einer Annotation *DataProvider* versehen. Jedes enthaltene eindimensionale Array des Resultats repräsentiert die Parameter für eine Test-Methode. Dabei müssen die Typen und die Anzahl der Parameter mit denen der Testdaten übereinstimmen.

Zunächst werden die Testdaten definiert, die von einer Methode der Testklasse geliefert werden. Dazu wird diese Methode mit einer Annotation (*DataProvider*) versehen, der ein Name zugewiesen wird.

```
@DataProvider{val name="coords"}
def createTestCoords: Array[Array[Object]] = Array(
  Array(GeographicCoOrdinatePair( 0, 0), UtmCoOrdinates(/*...*/)),
  /*...*/)
```

Programmauszug 7.1: TestNG - DataProvider Annotation

Im Programmauszug 7.1 ist das erste Element des Array aufgeführt. Es besteht selbst aus einem Array mit zwei Elementen. Dies ist das Tupel der Testdaten, die als Parameter für die Testmethoden verwendet werden sollen.

Die definierte Testmethode *latLonToUtm* verfügt über die zwei passenden Parameter. Zum Vergleich der konvertierten Koordinaten mit der gegebenen Referenz wird die bereits erwähnte Methode *alwaysEqual* verwendet. In der Annotation *Test*, die die Methode als Test-Methode kennzeichnet, wird der Name des definierten *DataProviders* angegeben.

```
@Test { val dataProvider= "coords" }
def latLonToUtm(
   latLonCoord : GeographicCoOrdinatePair, utmCoord : UtmCoOrdinates) {
   val convertedUtm = converter.latLonToUtm(latLonCoord)
   assert(utmCoord.alwaysEqual(convertedUtm), "Fehlschlag" /*...*/)
}
```

Programmauszug 7.2: TestNG - Definition einer Testmethode

Werden die Tests mit Hilfe des TestNG-Frameworks ausgeführt, wird diese Testmethode der Reihe nach mit allen definierten Testdaten appliziert. Mit Hilfe der Methode *assert* wird ein Fehler erzeugt, wenn die Methode *alwaysEqual* ein negatives Resultat (*false*) lieferte. Der große Vorteil des DataProviders ist, dass ein fehlerhafter Testdurchlauf mit einem Tupel nicht zum Abbruch der folgenden Tests führt.

Die Integration von TestNG in die Android-Plattform misslang, weil benötigte Pakete in der Android-API nicht vorhanden sind und die Abhängigkeiten so nicht aufgelöst werden konnten. Dies war ein großer Nachteil bei der Entwickelung der Beispielanwendung, denn wie in Abschnitt 4.3.3 auf Seite 49 besprochen, können Fehler unter Umständen nur innerhalb der Android-Plattform auftreten.

Ein Nachteil der Verwendung von Annotationen sind die dort enthaltenen Zeichenketten. Unterscheidet sich der *name* des *DataProvider* von dem in der Annotation *Test* verwendeten, dann führt dies zu einem Laufzeitfehler. Ein Laufzeitfehler entsteht auch, wenn die Anzahl oder Typen der Testtupel nicht mit den Parametern der Testmethode übereinstimmen.

7.3 JUnit3

Im Rahmen dieser Arbeit wurde mit Junit3 getestet, das in die Android-Plattform integriert ist. Die Namen der Test-Methoden einer JUnit3-Testklasse beginnen per Konvention mit der Zeichenkette "test" und die Testklassen müssen *junit.framework.TestCase* beerben. In JUnit3 gibt es keine Entsprechung für den *DataProvider* aus TestNG. Aus diesem Grund wird innerhalb der Testmethode direkt über die Liste der Testdaten iteriert. Schlägt ein Test für ein Tupel fehl, dann wird die Ausführung der Test-Methode abgebrochen. Alle restlichen Testtupel werden dann nicht mehr für diesen Test verwendet.

```
val coords = List(
   (GeographicCoOrdinatePair( 0, 0), UtmCoOrdinates(/*...*/)),
    /*...*/)
def testLatLonToUtm() {
   for ((latLonCoord,utmCoord) ← coords) {
      val convertedUtm = converter.latLonToUtm(latLonCoord)
      assert(utmCoord.alwaysEqual(convertedUtm), "Fehlschlag" /*...*/)
   }
}
```

Programmauszug 7.3: JUnit3 - Definition einer Testmethode

Die definierten Tests können sowohl innerhalb der Android-Plattform, als auch in der herkömmlichen Java-VM ausgeführt werden. Zum Starten der Tests in der Android-Plattform wird eine Android-Anwendung entwickelt, deren Activity bestimmte Klassen des JUnit-Frameworks nutzt, um die gewünschten Testklassen zu starten. Wenn zum Starten die Klasse *android.test.AndroidTestRunner*, eine Android-spezifische Erweiterung des JUnit-Frameworks, verwendet wird, werden die Testergebnisse in die Log-Datei des Systems geschrieben.

Für einige Tests, zum Beispiel für die DAO-Implementierung, wird das bekannte *Context*-Objekt benötigt. Dieses Objekt kann in die Testklassen direkt mit einem "setter" injiziert werden, indem die Testklassen die Klasse *android.test.AndroidTestCase* beerben, die die notwendigen Methoden bereits anbietet. Die Klasse *AndroidTestCase* beerbt ihrerseits die Klasse *TestCase* des Junit3-Frameworks.

Die Integration von Scala und JUnit3 ist problemlos möglich. Ein Nachteil gegenüber TestNG ist, dass mit JUnit3 die Testklassen von der Klasse *TestCase* abgeleitet sein müssen. Da Scala über das sprachliche Mittel der Traits verfügt, ist dies nicht von so großer Bedeutung. Ein weiterer Nachteil von JUnit3 sind die Konventionen bezüglich der Namen und das Fehlen der Funktionalität eines *DataProviders*.

7.4 ScalaTest

Die Testklassen des Framework ScalaTest müssen das Trait *org.scalatest.Suite* einmischen. Es gibt verschiedene von *Suite* abgeleitete Traits in diesem Framework. Im Folgenden wird nur das Trait *org.scalatest.FunSuite* für die Testklassen verwendet, denn Testklassen die *FunSuite* einmischen, heben sich syntaktisch von den TestNG- und JUnit-Testklassen deutlich ab. Anstelle von Konventionen oder Annotationen werden hier Funktionen höherer Ordnung für das Spezifizieren der Tests verwendet.

Wird eine definierte Testklasse vom ScalaTest-Framework ausgeführt, werden die eigentlichen Tests zunächst angemeldet. Dazu müssen diese als Funktions-Parameter einer

Methode *test* übergeben werden. Diese Methode ist im Trait *FunSuite* als Funktion höherer Ordnung in Currying-Schreibweise definiert und besitzt zwei Parameterlisten:

```
test(testName : String, testGroups : Group*)(f : ⇒ Unit)
```

Programmauszug 7.4: ScalaTest - Methode test des Trait FunSuite

Die erste erwartet die Angabe eines Namens für den Test. Dazu kann eine Menge von Testgruppen angegeben werden. Es ist möglich, nur die Tests zu starten, die bestimmten Gruppen angehören.[1] Die zweite Parameterliste erwartet eine Funktion, die den eigentlichen Programmcode für den Test beinhaltet.

Aufgerufen wird die Methode *test* direkt im Rumpf der Testklasse. Dabei ist es möglich, mehrere separate Tests durch mehrfaches Aufrufen zu spezifizieren. Die so angemeldeten Tests werden dann der Reihe nach ausgeführt.

```
test("latLonToUtm") {
   for ((latLonCoord,utmCoord) ← coords) {
      val convertedUtm = converter.latLonToUtm(latLonCoord)
      assert(utmCoord.alwaysEqual(convertedUtm), "Fehlschlag" /*...*/)
   }
}
```

Programmauszug 7.5: ScalaTest - Definition eines Tests

Grundsätzlich gibt es keine Funktionalität, die dem *DataProvider* von TestNG entspricht. Allerdings kann der Aufruf der Methode *test* natürlich in einer Schleife durchgeführt werden.

```
for ((latLonCoord,utmCoord) ← coords) test("latLonToUtm") { /*...*/ }
```

Programmauszug 7.6: ScalaTest - Umsetzung der Funktionalität eines TestNG-DataProvider

Damit würden für alle Testtupel der Testdaten separate Tests durchgeführt. Durch einen Fehler der aktuellen Version von Scala führt der der Programmcode im Auszug 7.6 zum Absturz des Scala-Compilers [Vgl. Myb09b, Bugreport]. Es ist davon auszugehen, dass in zukünftigen Versionen von Scala dieser Fehler behoben sein wird.

Eine Klasse, die *FunSuite* einmischt, kann direkt instantiiert und verwendet werden, um die dort angemeldeten Tests zu starten. Die Ausgabe der Resultate kann mit Hilfe einer von *Reporter* abgeleiteten Klasse selbst implementiert werden. Ein Objekt dieser Klasse wird dann bei der Ausführung mit angegeben.

Das Ausführen der Tests innerhalb der Android-Plattform ist grundsätzlich kein Problem,

1 Ähnliche Gruppen können auch mit TestNG oder JUnit definiert werden.

da die ScalaTest Bibliothek keine Abhängigkeiten besitzt, die die Android-API nicht erfüllen kann. Zudem ist es sehr leicht möglich, die Repräsentation der Testergebnisse mit einem *Reporter* umzusetzen.

7.5 Implementierung der Tests für die Anwendung TribOSMap

In diesem Abschnitt wird auf die Entwicklung der Tests eingegangen, die für die Anwendung TribOSMap entwickelt wurden. Diese Tests wurden mit dem Framework ScalaTest umgesetzt. Sie können innerhalb der Android-Plattform und der Java-VM ausgeführt werden. Einige Tests mussten speziell für die Ausführung in der Java-VM angepasst werden, da sie Klassen des Android-Framework benötigen. Generell lassen sich Tests außerhalb von Android öfter und schneller durchführen. Denn bei jeder Änderung der Tests und des zu testenden Programms muss die Testanwendung kompiliert und neu installiert werden, was ein relativ langwieriger und aufwändiger Prozess ist.

Unterschiedliche Einstellungen für Tests in verschiedenen Systemen

Grundsätzlich sollen Tests, die sich in ihrer Implementierung für das jeweilige System nicht unterscheiden, nur einmal entwickelt werden. Es soll aber möglich sein, bestimmte Parameter für die Tests beider Systeme unterschiedlich angeben zu können. Diese bestimmen zum Beispiel, wie oft Tests mit zufällig generierten Parametern aufgerufen werden.[1]

Die Tests sind mittels von *FunSuite* abgeleiteten Traits umgesetzt. Diese Test-Traits definieren “sockets” für bestimmte Einstellungen, wie zum Beispiel die Anzahl von Testdurchläufen mit automatisch generierten Testdaten.

Zwei weitere Traits stellen die jeweiligen Testeinstellungen für die Android-Plattform sowie die Java-Plattform als “plugs” zur Verfügung. Dadurch ist es möglich, direkt bei der Instantiierung mittels “Flattening” die Test-Traits mit den jeweiligen Einstellungen zu kombinieren.

Es wurden zwei von *SuperSuite* abgeleitete Klassen implementiert. Eine *SuperSuite* ist eine spezielle Klasse des Framework ScalaTest, die andere *Suites* beinhalten kann, die ihr in einer Liste als Konstruktorparameter übergeben werden. Bei dem Ausführen einer *SuperSuite* werden alle enthaltenen *Suites* ausgeführt.

Im Folgenden ein Auszug der Klasse *tribosmap.test.offline.AllTests*, die Tests außerhalb von Android zur Verfügung stellt.

1 Es wurde das Framework ScalaCheck [Nil] für das automatische Generieren von Testfällen verwendet.

```
class AllTests extends SuperSuite( List(
   new CoordinatesTest with SettingsOffline,
   /*...*/ ))
```

Programmauszug 7.7: Test von TribOSMap - SuperSuite für das Testen in der Java-VM

Man kann erkennen, wie das definierte Test-Trait (*CoordinatesTest*) mit dem Trait *SettingsOffline*, das die Einstellung zur Ausführung in der Java-VM vorhält, kombiniert wird. Außerhalb von Android kann mit Hilfe der Mittel von ScalaTest diese *SuperSuite* problemlos ausgeführt werden.

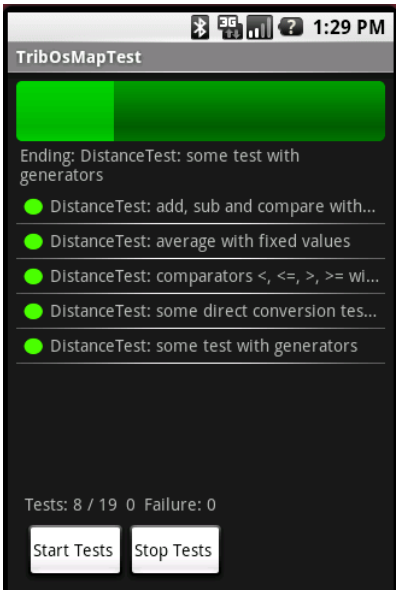

Abbildung 7.1: Testanwendung

Analog existiert eine *SuperSuite* für das Testen innerhalb von Android, in der die Tests mit dem Trait *SettingsOnline* kombiniert werden. Für Tests innerhalb der Android-Plattform wurde eine Android-Anwendung entwickelt, in dessen Activity die *SuperSuite* initialisiert und ausgeführt wird. Für die Ausgabe der Testresultate wurde die aus ScalaTest stammende Schnittstelle *Reporter* implementiert, so dass die Informationen von ScalaTest bezüglich der Testausführung auf dem Display grafisch dargestellt werden können.

Durch den Einsatz der Traits ist eine Implementierung unterschiedlicher Klassen für Tests, die sich lediglich in bestimmten Einstellungen unterscheiden, unnötig. Die gewünschten Einstellungen lassen sich zentral vornehmen und für das jeweilige System anpassen.

Testen der Geschäftslogik innerhalb der Android-Plattform

Die Geschäftslogik der Anwendung TribOSMap im Paket *model.business* ist grundsätzlich nicht vom Android-Framework abhängig. Allerdings wird mit Hilfe des Trait *ServiceAccess* die SQLite-Implementierung der DAO-Schnittstellen verwendet.

Wie in Abschnitt 6.3.2 auf Seite 87 erläutert, instantiiert das Trait *ServiceAccess* die Klasse *SQLiteDaoAccessor*, die wiederum die SQLite-DAO-Klassen instantiiert. Bei der Instantiierung des *SQLiteDaoAccessors* wird der Name der Datenbank angegeben, mit der die DAO-Schicht arbeiten soll.

Um die Geschäftslogik der Anwendung innerhalb von Android zu testen, wurden Testklassen entworfen, die Dienste (z.B. *TraceService*) verwenden. Dazu benötigen die Testklassen aus bekannten Gründen ein Context-Objekt, wenn die SQLite-DAO Implemen-

tierung genutzt werden soll (Abschnitt 6.3.2 auf Seite 87). Deshalb beerben diese Klassen die Klasse *AndroidTestCase* des Android-Framework.

Da für das Testen eine andere Datenbank verwendet werden soll als in der Anwendung, wurde ein von *ServiceAccess* abgeleiteter Trait *OnlineTestServiceAccess* implementiert, in dem der *SQLiteDaoAccessor* mit einem anderen Datenbanknamen initialisiert wird. Auch dieses Trait wird in die Testklasse eingemischt.

```
TraceServiceTest extends AndroidTestCase with FunSuite with OnlineTestServiceAccess
```

Programmauszug 7.8: Test von TribOSMap - TraceServiceTest

Auf diese Art und Weise kann sämtliche Funktionalität der Geschäftlogik getestet werden. Dabei wird durch die Verwendung des Trait *OnlineTestServiceAccess* eine Testdatenbank genutzt.

Testen der Geschäftslogik außerhalb der Android-Plattform

Außerhalb des Android-Systems sollen die Tests ohne Datenbank durchgeführt werden, denn die SQLite-DAO-Implementierung ist von Klassen des Android-Framework direkt abhängig. Dadurch wird das Context-Objekt nicht benötigt und es muss auch keine SQLite-Datenbank aufgesetzt werden.

Wieder zahlt sich die Komponenten-basierte Architektur der Dienste aus. Es wurde ein spezielles Trait *OfflineTestServiceAccess* implementiert, um die "sockets" der einzelnen Dienste (z.B. TraceService) zu befriedigen. Dieses Trait instantiiert eine anonyme Klasse mit Hilfe des Trait *DaoAccessor*, wobei für die einzelnen DAO-Schnittstellen Mock-Objekte erzeugt werden.

```
trait OfflineTestServiceAccess extends org.specs.mock.JMocker
   with TraceServiceComponent with /*...*/ {
   val daoAccessor = new DAOAccessor {
      val traceDAO: TraceDAO = mock[TraceDAO]
      /*...*/
   }
}
```

Programmauszug 7.9: Test von TribOSMap - OfflineTestServiceAccess

Diese Mock-Objekte werden mit Hilfe des Framework JMock ([Jmo]) erzeugt, indem die Methode *mock* der Klasse *JMocker* mit einer Schnittstelle parametrisiert wird. Resultat ist ein Mock-Objekt, das diese Schnittstelle implementiert. Für dieses Objekt kann mit Hilfe des JMock-Framework genau angegeben werden, welches Resultat bei einem Aufruf einer bestimmten Methode zurückgegeben werden soll. Bei einem Aufruf einer Methode eines

Dienstes ist klar, welche Aufrufe auf der DAO-Schnittstelle vorgenommen werden sollten. Somit lassen sich die Implementierungen der Dienste ohne die SQLite-DAO-Klassen überprüfen. Dazu mischen die definierten Testklassen neben der *FunSuite* auch das Trait *OfflineTestServiceAccess* ein.

Zusammenfassung

Die Testframeworks JUnit und TestNG lassen sich mit Scala sehr gut verwenden. Darüber hinaus gibt ScalaTest die Möglichkeit, die sprachlichen Mittel von Scala bei der Implementierung von Tests zu nutzen. Die implementierten Testklassen werden somit übersichtlicher und weniger fehleranfällig, da keine Annotationen verwendet werden und auch Namenskonventionen keine Rolle spielen.

Android unterstützt die Entwicklung von Tests mit JUnit3. Dadurch lassen sich Tests innerhalb und außerhalb von Android ausführen. Dies ist auch mit ScalaTest möglich. Generell kann das Context-Objekt vom Activity in die Testklassen injiziert werden. Dadurch lassen sich alle Teile einer Android-Anwendung testen. Eine Ausnahme sind die implementierten Activity-Klassen, Android-Services und alle Widget-Klassen, da diese nicht direkt instantiierbar sind.[1] Aus diesem Grund sollte darauf geachtet werden, nur notwendige Teile der Implementierungen in diesen Klassen vorzunehmen. Und es wird so auch möglich, die Tests außerhalb der Android-Plattform zu verwenden.

Scala erlaubt mit Hilfe der Traits ein flexibles Design der Softwarearchitektur. Wenn bei der Entwicklung darauf geachtet wird, möglichst unabhängige Komponenten mit Hilfe der Traits zu entwickeln, vereinfacht dies die Implementierung der Tests, da Komponenten mit angepasste Test-Komponenten ausgetauscht werden können.

Besonders bei der Entwicklung von Android-Anwendungen ist es wichtig, die Architektur so zu entwerfen, dass große Teile außerhalb von Android validierbar sind, da die Installation einer Testanwendung im Android-System sehr lange dauert. Andernfalls führt dies dazu, dass der Entwicklungsprozess qualitativ oder quantitativ leidet. Tests innerhalb des Android-Systems sind unbedingt notwendig, da bestimmte Fehler nur innerhalb von Android auftreten.

1 Android bietet spezielle Klassen zum Testen der GUI an. Tests dafür zu entwickeln ist komplex und wird in dieser Arbeit nicht näher erläutert.

8 Erfahrungen mit der Entwicklungsumgebung Eclipse

Für die Entwicklung der Beispielanwendung wurde die IDE Eclipse in der Version 3.4.1 verwendet. Eclipse ist quelloffen und kostenlos und wird vornehmlich für die Entwicklung von Java-Anwendungen genutzt. Eclipse stellt im Prinzip nur ein Grundgerüst zur Verfügung. Für den speziellen Einsatz werden Plugins integriert. Zum Beispiel gibt es neben einer ganzen Reihe von Java-spezifischen Plugins auch solche für die Entwicklung für mobile Geräte mit C++ oder auf Basis von J2ME. Zudem existieren Plugins für von Scala- und Android-Anwendungen, die für die Entwicklung der Anwendung TribOSMap verwendet wurden. Im Folgenden wird auf Details dieser Plugins eingegangen.

8.1 Scala-Eclipse-Plugin

Der Scala-Editor hebt Syntaxkomponenten und Schlüsselwörter farbig hervor. Bestimmte Fehler, wie zum Beispiel fehlende Klammern, werden sofort angezeigt. Fehler, die bei einer fehlgeschlagenen Kompilierung auftreten, werden im Editor markiert und auch das automatische Vervollständigen von Methoden-, Variablen- oder Klassennamen ist integriert. Dies funktioniert mit eingebundenen Scala- und Java-Bibliotheken. Ist der Quellcode oder die Dokumentation für genutzte Bibliotheken konfiguriert, werden beim "Überfahren" mit der Maus die Kommentare der Methoden oder Klassen angezeigt. Auch können Methoden und Klassennamen als Links verwendet werden, die dann im Editor die zugehörige Quellcode-Datei öffnen.

In der Projektübersicht (Package-Explorer) oder einem separaten Fenster (outline) können Methodensignaturen von Klassen oder Attributen angezeigt werden. Das Anzeigen von Paketinhalten ist allerdings problematisch, denn im Package-Explorer werden lediglich Ordnerstrukturen angezeigt. Das ist bei Java-Programmen durchaus sinnvoll, denn hier entsprechen die Pakete der Ordnerhierarchie. Bei Scala hingegen ist dies nicht zwangsläufig der Fall. Das Scala-Plugin hilft bei der Erzeugung von Projekten, Klassen, Traits und Singleton-Objekten. Es werden auf Wunsch passende Dateien erzeugt. Für Scala-Projekte werden die notwendigen Bibliotheken und Einstellungen automatisch konfiguriert. Die er-

zeugten Scala-Projekte können auch Java-Dateien enthalten, die automatisch mitkompiliert werden.

Grundsätzlich ist festzustellen, dass mit dem Scala-Plugin für Eclipse sehr gut gearbeitet werden kann. Allerdings ist dieses Plugin noch bei weitem nicht so umfangreich, wie die für Java vorhandenen Plugins. Besonders beim Refactoring wird der Entwickler nicht unterstützt. Das Umbenennen von Dateien, Methoden, Ordnern und Paketen muss komplett manuell vorgenommen werden. Dies wird noch dadurch erschwert, dass zum einen Klassennamen nicht an Dateinamen und zum anderen Paketnamen nicht an Ordnernamen gebunden sind.

8.2 Android-Eclipse-Plugins

Zur Entwicklung von Android-Anwendungen kann das Plugin ADT genutzt werden. Das Plugin DDMS dient dem Überwachen eines Android-Systems (z.B. des Emulators).

Mit dem Plugin ADT kann sehr komfortabel eine Projektstruktur für eine Android-Anwendung angelegt werden. Es werden automatisch alle notwendigen Dateien erzeugt und ein kleines “Hallo Welt”-Activity implementiert. Somit kann die Anwendung unverzüglich kompiliert und im Emulator ausgeführt werden. Zum Umfang dieses Plugin gehört ein Editor für die Layout-Dateien mit einer grafischen Vorschau und auch ein Android-Manifest-Editor. Alle relevanten XML-Dateien werden auf ihre Richtigkeit hin geprüft. Mit dem ADT-Plugin wird der Entwickler auch bei der Kompilierung und Paketierung unterstützt. Direkt aus Eclipse heraus kann ein Emulator gestartet und die fertig paketierte Anwendung in diesem installiert werden. Bei Fehlern, sowohl in den XML-Dateien als auch in den Java-Dateien, wird die Kompilierung abgebrochen. Diese Fehler werden dann in den XML-Dateien direkt angezeigt.

Mit Hilfe des Plugin DDMS lassen sich die gestarteten Emulatoren überwachen. Die Logging-Ausgaben können direkt in der IDE verfolgt werden. Zum Umfang von DDMS gehört ein Datei-Browser für das Android-System ebenso wie eine Übersicht über die laufenden Prozesse.

Generell kann bei Verwendung dieser Plugins auf die Kommandozeilen-Werkzeuge verzichtet werden. Lediglich das Übertragen der GPS-Koordinaten aus einer Datei funktionierte nicht, weshalb bei der Entwicklung von TribOSMap doch das Werkzeug “adb” zum Einsatz kam.

8.3 Kombination des Scala-Plugin mit den Android-Plugins

Für die Entwicklung kam sowohl das Scala-Plugin, als auch die beiden Android-Plugins zum Einsatz. Dies führte zunächst zu einigen Problemen. Das ADT-Plugin erzeugt eine bestimmte Projektstruktur. Es bindet diverse Bibliotheken in das Projekt ein und konfiguriert spezielle "Builder", die das Kompilieren und Paketieren der Anwendung vornehmen. Das Scala-Plugin besitzt ebenso einen "Builder", der die herkömmliche Java-Bibliothek und die Scala-Bibliothek benötigt. Diese Bibliotheken müssen als Abhängigkeiten im Projekt konfiguriert sein.

Um ein Android-Projekt zu erzeugen, das auch das Entwickeln mit Scala unterstützt, muss ein herkömmliches Android-Projekt mit Scala- und Java-Bibliotheken verknüpft werden. Dabei muss die für Android angepasste Scala-Bibliothek verwendet werden (Abschnitt 4.3.2 auf Seite 47). Die Scala-Bibliothek ist für das Kompilieren und Paketieren notwendig. Dagegen wird die Java-Bibliothek lediglich zum Starten des Scala-Builder, also für die Kompilierung, benötigt.

Auch die Integration des Scala-Builders in das Android-Projekt muss manuell vorgenommen werden. Das ADT Plugin konfiguriert bereits vier Builder. Hier eine Auflistung, bei der den Buildern die entsprechenden Kommandozeilenwerkzeuge zugeordnet werden. Die Reihenfolge entspricht der Ausführung.

1. Resource Manager (aapt)
2. Android Pre Compiler (aidl)
3. Java Builder (javac)
4. Android Package Builder (dx, aapt)

Bereits in Abschnitt 4.3.1 auf Seite 47 wurde besprochen, dass der Java-Compiler innerhalb des Android Entwicklungsprozesses durch den Scala-Compiler ausgetauscht werden kann. Deshalb wird hier der Java-Builder durch den bereits erwähnten Scala-Builder ersetzt. Dazu wird die Datei *.project* im Eclipse-Projekt entsprechend angepasst.

Ein weiteres Problem tritt bei der Implementierung von Activities mit Scala auf. Die Activities müssen in der Manifest-XML-Datei konfiguriert sein. Das ADT-Eclipse-Plugin prüft, ob die konfigurierten Vorgaben bestimmten Kriterien entsprechen. Für diese Prüfung (Quellcode-Analyse) wird vom Plugin die Funktionalität des Eclipse-JDT [Ecl] genutzt um festzustellen, ob eine als Activity spezifizierte Klasse die Klasse *android.app.Activity* beerbt. Diese Prüfung funktioniert nicht mit Scala-Quellcode. Um sie stattdessen mit dem generierten Java-Bytecode vornehmen zu lassen, muss das Verzeichnis mit dem generierten Bytecode (bin) im Eclipse als Abhängigkeit hinzugefügt werden.

Nachdem alle diese Probleme behoben wurden, lassen sich die Android-Anwendungen mit Scala umsetzen. Eine Ausnahme bilden die Stubs für die Remote-Schnittstellen und die Datei R.java. Diese werden, wie bereits besprochen, automatisch generiert.

Die Entwicklung von Android-Anwendungen mit den vorgestellten Plugins hat sich, nachdem diese anfänglichen technischen Probleme gelöst waren, als sehr komfortabel erwiesen. Es traten zwar immer wieder Fehler auf, die aber zum großen Teil durch neuere Versionen der Plugins behoben wurden.

Als Ergebnis der Untersuchungen entstand, auf Initiative des Autors und in Zusammenarbeit mit den Entwicklern von Scala, eine Anleitung im Internet, wie Scala und Android in einem Eclipse-Projekt genutzt werden können [Vgl. Scaa].

9 Fazit und Ausblick

In der vorliegenden Arbeit wurde gezeigt, dass die Programmiersprache Scala für die Entwicklung von Android-Anwendungen genutzt werden kann.

Der Scala-Compiler kann in den Prozess der Entwicklung von Android-Anwendungen problemlos integriert werden. Der Einsatz des Scala-Interpreter ist nicht möglich, da der aktuelle Dalvik-Bytecode-Konverter nicht im Android-System ausführbar ist. Um die mit Scala entwickelten Anwendungen ausführen zu können, mussten zunächst einige technische Probleme gelöst werden, die vor allem mit der Neuartigkeit der Android-Plattform zusammenhängen. Während der Anfertigung der Diplomarbeit wurde die erste offizielle Version von Android und eine neue Version von Scala veröffentlicht. Es wurde demonstriert, dass punktuelle Änderungen an der Scala-Bibliothek ausreichen, um eine grundsätzliche Kompatibilität der beiden Versionen herzustellen. Ebenso wurde gezeigt, dass die Unterschiede zwischen den Implementierungen der Android-API und der Standard-Java-API bedacht werden müssen, um Fehler durch die Nutzung der Scala-API im Android-System zu verhindern.

Anhand der Beispielanwendung wurde demonstriert, dass Scala über mächtige sprachliche Mittel für die Entwicklung verfügt. Besonders hervorzuheben sind die Traits, die ein flexibles modulares Design ermöglichten und damit das Refactoring und das Testen erleichterten. Darüber hinaus halfen Traits, die strikte Trennung Android-abhängiger technischer Details von der Geschäftslogik umzusetzen. Die funktionalen Fähigkeiten von Scala führten zu einem übersichtlichen und eleganten Code, vereinfachten die Entwicklung zustandsfreier Komponenten und vermieden Redundanzen.

Sowohl die Integration von Scala und Android in die Entwicklungsumgebung als auch der Einsatz von Testwerkzeugen sind gelungen. Damit wurde gezeigt, dass alle notwendigen Mittel vorhanden sind, um Scala produktiv bei der Entwicklung für die Android-Plattform einzusetzen.

Scala kann sich zu einer echten Alternative zu Java entwickeln. Scala entwickelt sich schnell weiter, die nächste Version (2.8) ist bereits angekündigt. Auch der Autor hat sich aktiv am Entwicklungsprozess beteiligt. Es wurden zwei Fehler ([Myb09b], [MyB09a]) nachgewiesen und an die Entwickler von Scala gemeldet. Für das CLBG-Projekt wurden Scala-Tests an die aktuelle Scala-Version angepasst und an das Projekt zurückgegeben.

Weiterhin wurde an einer Anleitung für den kombinierten Einsatz von Scala und Android in der IDE Eclipse mitgewirkt [Scaa].

In den letzten Wochen wurde das erste Mobilfunkgerät mit einem Android-System in Deutschland ausgeliefert. Die Popularität von Android wird weiter steigen, da viele Handy-Hersteller weitere Produkte angekündigt haben. Der Vorteil der Android-Plattform ist neben der Quelloffenheit das Vorhandensein eines umfangreichen Framework zur Entwicklung von Anwendungen.

Mit der im Rahmen der Diplomarbeit entwickelten Anwendung TribOSMap wird ein Grundstein für ein Open-Source-Projekt gelegt. Bis jetzt gibt es keine Scala-Anwendungen, die im OpenStreetMap-Projekt eine Rolle spielen. Mein Ziel ist es, sowohl Scala als auch OpenStreetMap bekannter zu machen. Deshalb wird nach der Verteidigung dieser Diplomarbeit die Anwendung unter Gnu-Public-License veröffentlicht [Rac].

Quellenverzeichnis

[And] Android Development Tools (ADT) Webseite.
http://code.google.com/intl/de-DE/android/intro/tools.html

[And08] Package org.xml.sax Webseite (2008),
http://code.google.com/intl/de-DE/android/reference/org/xml/sax/package-descr.html

[Apa04] Apache License, Version 2.0 (2004),
http://www.apache.org/licenses/LICENSE-2.0

[Bar00] Barendregt, Henk und Barendsen, Erik: Introduction to Lambda Calculus (2000),
http://www.cs.ru.nl/E.Barendsen/onderwijs/sl2/materiaal/lambda.pdf

[Beu] Beust, Cédric: TestNG Webseite.
http://testng.org/doc/index.html

[Bsd06] The BSD License (2006),
http://www.opensource.org/licenses/bsd-license.php

[Cak06] Cake Pattern (2006),
http://scala.sygneca.com/patterns/component-mixins

[Car85] Cardelli, Luca und Wegner, Peter: On Understanding Types, Data Abstraction, and Polymorphism (1985),
http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

[Cre] Creative-Commons Webseite.
http://creativecommons.org/

[DAO02] Core J2EE Patterns Data Access Object (2002),
http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[DB08] Dan Bornstein, Google: Dalvik VM Internals (2008),
http://sites.google.com/site/io/dalvik-vm-internals

[DeP08] DePriest, Dale: NMEA-data Webseite (2008),
http://www.gpsinformation.org/dale/nmea.htm

[Duc06] Ducasse, Stephane; Nierstrasz, Oscar; Schaerli, Nathanael; Wuyts, Roel und Black, Andrew P.: Traits: A Mechanism for Fine-grained Reuse (2006),
http://www.iam.unibe.ch/~scg/Archive/Papers/Duca06bTOPLASTraits.pdf

[Ecl] Eclipse Java development tools (JDT) Webseite.
http://www.eclipse.org/jdt/

[Emi07] Emir, Burak; Odersky, Martin und Williams, John: Matching Objects With Patterns (2007),
http://lamp.epfl.ch/~emir/written/MatchingObjectsWithPatterns-TR.pdf

[Fow03] Fowler, Martin: Anemic Domain Model (2003),
http://www.martinfowler.com/bliki/AnemicDomainModel.html

[Gam95] Gamma, Erich; Helm, Richard und Johnson, Ralph E.: *Design Patterns. Elements of Reusable Object-Oriented Software.*, Addison-Wesley Longman, auflage 1st ed. Aufl. (1995)

[GPX08] GPX 1.1 Schema Documentation (2008),
http://www.topografix.com/GPX/1/1/

[Jar] jarjar Webseite.
http://code.google.com/p/jarjar/

[Jav] Package org.xml.sax Webseite.
http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/package-summary.html

[Jmo] jMock Webseite.
http://www.jmock.org/

[Jos] Java-OpenStreetMap-Editor Webseite.
http://josm.openstreetmap.de/

[Jun] JUnit Webseite.
http://www.junit.org/

[Loa06] Loan-Pattern (2006),
http://scala.sygneca.com/patterns/loan

[Mar00] Martelli, Alex: polymorphism (was Re: Type checking in python?) (2000),
http://groups.google.com/group/comp.lang.python/msg/e230ca916be58835?hl=en&

[Mer] Merkaator Webseite.
http://www.merkaartor.org/

[Mur08] Murphy, Mark: Scripting Your Android Device (2008),
http://androidguys.com/?p=647

[MyB09a] Scala Trac: Ticket 1615 (2009),
https://lampsvn.epfl.ch/trac/scala/ticket/1615

[Myb09b] Scala Trac: Ticket 1675 (2009),
https://lampsvn.epfl.ch/trac/scala/ticket/1675

[Neh03] Nehrlich, Werner: *Diskrete Mathematik*, Fachbuchverlag Leipzig, 1. auflage Aufl. (2003)

[Nil] Nilsson, Rickard: ScalaCheck Webseite.
http://code.google.com/p/scalacheck/

[Ode05] Odersky, Martin und Zenger, Matthias: Scalable Component Abstractions (2005),
http://lamp.epfl.ch/~odersky/papers/ScalableComponent.pdf

[Ode08a] Odersky, Martin: Scalas Prehistory (2008),
http://www.scala-lang.org/node/239

[Ode08b] Odersky, Martin; Spoon, Lex und Venners, Bill: *Programming in Scala*, Artima Press, preprint edition version 3 Aufl. (2008)

[Ope] OpenStreetMap Webseite.
http://www.openstreetmap.org/

[PB08] Patrick Brady, Google: Anatomy and Physiology of an Android (2008),
http://sites.google.com/site/io/anatomy--physiology-of-an-android

[Pep03] Pepper, Peter: *Funktionale Programmierung in Opal, ML, Haskell und Gofer*, Springer, 2. auflage Aufl. (2003)

[Pep06] Pepper, Peter und Hofstedt, Petra: *Funktionale Programmierung*, Springer, 1. auflage Aufl. (2006)

[Pera] Computer-Language-Benchmarks-Game Java and Scala Webseite. http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=javaclient&lang2=scala

[Perb] Computer-Language-Benchmarks-Game Java and Scala Webseite: nbody. http://shootout.alioth.debian.org/u32q/benchmark.php?test=nbody&lang=scala&id=1

[Perc] Computer-Language-Benchmarks-Game Webseite. http://shootout.alioth.debian.org

[Perd] Computer-Language-Benchmarks-Game Webseite: spectralnorm. http://shootout.alioth.debian.org/u32q/benchmark.php?test=spectralnorm&lang=scala&id=1

[Pot] Potlatch Webseite. http://wiki.openstreetmap.org/index.php/Potlatch/

[Qem08] QEMU Documentation (2008), http://bellard.org/qemu/user-doc.html

[Rac] Rachimow, Meiko: TribOSMap Webseite. http://code.google.com/p/tribosmap/

[Scaa] Scala and Android in an Eclipse project. http://www.scala-lang.org/node/160

[Scab] ScalaTest Webseite. http://www.artima.com/scalatest/

[Sca08a] Scala 2.7.2 final (2008), http://www.scala-lang.org/node/348

[Sca08b] A Tour of Scala Unified Types (2008), http://www.scala-lang.org/node/128

[Sch02] Schärli, Nathanael; Ducasse, Stéphane und Nierstrasz, Oscar: Classes = Traits + States + Glue (2002), http://www.iam.unibe.ch/~scg/Archive/Papers/Scha02aTraitsPlusGlue2002.pdf

[Sch05] Schärli, Nathanael: Traits, Composing Classes from Behavioral Building Blocks (2005),
http://www.iam.unibe.ch/~scg/Archive/PhD/schaerli-phd.pdf

[Ser06] Object Serialization (2006),
http://java.sun.com/javase/6/docs/technotes/guides/serialization/

[Sli] Slippy-Map Webseite.
http://wiki.openstreetmap.org/wiki/Slippy_map/

[Sli08] Slippy map tilenames (2008),
http://wiki.openstreetmap.org/index.php/Slippy_map_tilenames

[Tor91] Torvalds, Linus: GNU GENERAL PUBLIC LICENSE Version 2 (1991),
http://www.kernel.org/pub/linux/kernel/COPYING

[Tru04] Truyen, Eddy; Joosen, Wouter; Jørgensen, Bo N und Verbaeten, Petrus: A Generalization and Solution to the Common Ancestor Dilemma Problem in Delegation-Based Object Systems (2004),
http://www.cs.kuleuven.ac.be/~eddy/PUBLICATIONS/DAW2004.ps

[Utm89] The Universal Grids: Universal Transverse Mercator (UTM) and Universal Polart Stereographic (UPS) (TM8358.2) (1989),
http://earth-info.nga.mil/GandG/publications/tm8358.2/TM8358_2.pdf

[Wgs00] World Geodetic System 1984 (TR8350.2) (2000),
http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf

[Zhe06] Zheng, Qiang: Beweise in der funktionalen Programmierung (2006),
www.tu-ilmenau.de/fakia/fileadmin/template/startIA/swt/Lehre/2006ss/HS_SS06/.../Beweise_in_der_funktionalen_Programmierung.pdf

Glossar

ACPI Abk. für Advanced Configuration and Power Interface: erlaubt die Steuerung des Power- und Geräte-Management eines PC

ADT Abk. für Android Development Tools: Plugin für Eclipse zur Entwicklung von Android-Anwendungen

API Abk. für Application-Programming-Interface

Application-Programming-Interface englisch für SSchnittstelle zur Anwendungsprogrammierung"; Schnittstelle mit der Softwarekomponenten ihre Funktionalität anbieten, um sie in andere Komponenten einbinden zu können

Berkeley-Software-Distribution ein Unix System (Betriebssystem)

BSD Abk. für Berkeley-Software-Distribution

Build-Zyklus der Zyklus bei der Entwicklung von Anwendungen, in dem diese kompiliert und paketiert werden

Bytecode enthält Befehle zur Ausführung in einer VM

CLBG The Computer Language Benchmark Game: Projekt, welches die Laufzeiten und den Speicherverbrauch von Programmen vergleicht, die in unterschiedlichen Programmiersprachen entwickelt wurden. [Perc]

DAO Abk. für Data Access Object: Entwurfsmuster für die Persistenzschicht von Anwendungen, dabei bilden die DAO-Klassen die Fachklassen auf die Datenbanktabellen ab

DDMS Abk. für Dalvik Debug Monitor Service: Eclipse Plugin für Android zur Verwaltung des Android-Systems, Debuggin, Logging, Dateibrowser etc.

DEX Abk. für Dalvik Executable: Dateiformat für den Bytecode im Android-System

DNS Abk. für Domain Name System: hierarchisches Namenssystem im Internet

Extensible-Markup-Language ist eine Auszeichnungssprache mit der hierarchische Strukturen dargestellt werden können

Flattening englisch für Abflachung, Richten, Strecken etc. 1. die Abflachung eines Ellipsoiden (Verhältnis vom äquatorialen zum polaren Radius) 2. das Einmischen von Traits direkt bei der Instantiierung

Global-Positioning-System ein satellitengestütztes Navigationssystem.

Gnu Public License Lizenz, die aussagt, dass alle Änderungen am Linux-Kernel veröffentlicht werden müssen

GPL Abk. für Gnu Public License

GPS Abk. für Global-Positioning-System

HAL Abk. für Hardware-Abstraction-Layer: englisch für Hardware-Abstraktions-Ebene

Hardware-Abstraktions-Ebene Schicht des Kernel mit Schnittstellen für Gerätetreiber (im Android-System gibt es eine zweite HAL im User-Space)

HTTP Abk. für Hypertext Transfer Protocol: Protokoll für die Datenübertragung im Netzwerk (vornehmlich Internet)

JIT Abkürzung für just-in-time

just-in-time englisch, etwas passiert bedarfsorientiert / zeitoptimal

Kernel das Fundament (der Kern) eines Betriebssystems

Kernel-Space virtueller Speicherbereich, der nur vom Kernel und seinen Erweiterungen (z.B. Treiber) genutzt werden darf (siehe auch User-Space)

Open source englisch für quelloffen; der Quelltext quelloffener Software ist unter einer Open-Source-Lizenz veröffentlicht, dadurch soll die Weiterentwicklung von Software vorangetrieben werden

plug englisch für Stecker; In dieser Arbeit sind die Methoden und Attribute gemeint, die von einem Trait zur Verfügung gestellt werden. (Gegenstück zu socket)

Remote-Schnittstelle spezifiziert Methoden eines Dienstes, die anderen Prozessen zur Ausführung angeboten werden (siehe auch Stub)

RPC Abk. für "remote procedure call": englisch für “Aufruf einer entfernten Prozedur”; entfernt meint hier: in anderen Prozessen, über das Netzwerk etc.

SAX Abk. für SSimple API for XML": Standard API zum sequentiellen Parsen von XML-Dateien

SDK Abk. für Software-Development-Kit

socket englisch für Steckdose: In dieser Arbeit sind die Attribute und Methoden eines Trait gemeint, die es benötigt und von einer einmischenden Klasse zur Verfügung gestellt werden müssen. (Gegenstück zu plug)

Software-Development-Kit eine Reihe von Werkzeugen und Programmbibliotheken zur Entwicklung von Software

Stub englisch für Stutzen, Halterung etc.; in dieser Arbeit ist die Implementierung einer Remote-Schnittstelle gemeint, erlaubt das Ausführen von Methoden in einem anderen Prozess

UML Abk. für Unified-Modeling-Language

Unified-Modeling-Language standardisierte Sprache, die für die Modellierung von Software eingesetzt wird

User-Space virtueller Speicherbereich, der für Benutzer-Anwendungen verwendet wird, (siehe auch Kernel-Space)

virtuelle Maschine simuliert ein Betriebssystem; Laufzeitumgebung für Programme; läuft als "Gastäuf einem Betriebssystem

VM Abk. für virtuelle Maschine

Widget Komponente einer grafischen Oberfläche (z.B. Schaltfläche, Texteingabefeld)

XML Abk. für Extensible-Markup-Language

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Programmauszüge

A Anhang: Fachliches Umfeld der Anwendung TribOSMap

In diesem Anhang wird auf das für die Entwicklung relevante fachliche Umfeld eingegangen. Es werden einige im weiteren Verlauf der Arbeit wiederkehrende Begriffe geklärt sowie grundlegende fachliche Zusammenhänge und Abläufe erläutert.

A.1 Geodäsie

Geodäsie ist die Wissenschaft von der Abmessung und Abbildung der Erdoberfläche.

A.1.1 Geografische Koordinaten

Eine Position auf einer Kugel ist mit zwei Winkeln beschreibbar. Der Koordinatenursprung gibt den Nullgradpunkt für beide Winkel an. Darauf basierend werden die geografischen Koordinaten definiert, die eine Position auf der Erdoberfläche beschreiben. In der Geodäsie hat sich dazu folgendes System durchgesetzt:

Die geografische Breite beschreibt den Winkel zum Äquator. Der Äquator liegt bei 0° geografischer Breite. Der Nordpol liegt bei +90° und der Südpol bei -90° Breite. Ohne Vorzeichen werden die Breiten im Norden auch nördliche Breite und die im Süden dementsprechend südliche Breite genannt.

Der Winkel, der vom Nullmeridian bis +180° nach Osten und bis -180° nach Westen reicht, wird als geografische Länge bezeichnet. Der Nullmeridian kreuzt beide Pole und die Stadt Greenwich in der Nähe Londons. Die Längengrade werden auch östliche und westliche Länge genannt, dann ist das Vorzeichen nicht nötig.

Ein Punkt auf der Erde wird mit dem Tupel von Längen- und Breitengrad geografisch beschrieben. Zum Beispiel liegt die Technische Fachhochschule Berlin ungefähr bei der geografischen Position mit den Koordinaten 13.35° östlicher Länge und 52.55° nördlicher Breite.

A.1.2 Geodätisches Datum

Die Erde kann durch ein Geoid, also eine Bezugsfläche, beschrieben werden, die in Höhe des mittleren Meeresspiegels liegt. Mit Hilfe von Pegelmessungen an Küstenorten, kann der Geoid bestimmt werden. In heutiger Zeit wird dies mit Satelliten gemacht. Da der Geoid sehr unförmig ist (mit vielen Beulen und Dellen), was durch die Erdrotation und die unregelmäßige Verteilung der Erdmasse verursacht ist, nutzt man diesen nicht für die Kartographie.

Konzentriert man sich auf die Rotationsverformung, kann ein Ellipsoid die Form der Erde sehr gut approximieren. Ein Ellipsoid ist eine drei-dimensionale Ellipse, beschrieben durch den äquatorialen und polaren Radius.[1]

Um ein Bezugssystem für die Vermessung zu schaffen, ist nun noch ein Referenzpunkt notwendig. Dieser wird Fundamentalpunkt genannt und beschreibt den Höhen-Nullpunkt auf dem verwendeten Ellipsoiden - also die geografische Höhe Null. Diese Kombination aus Fundamentalpunkt und Referenzellipsoid wird geodätisches Datum genannt.

Alle modernen GPS-Geräte unterstützen das geodätische Datum WGS84. Dieses wird auch im OpenStreetMap Projekt verwendet. WGS84 ist die Abkürzung für das "World Geodetic System" mit der Version des Jahres 1984 [Wgs00]. Dies ist nur einer der geografischen Standards, der einem sich ständig verändernden Geoiden angepasst wird.

A.1.3 Geografische Projektion

Die geografische Projektion beschreibt die Art und Weise, wie die Erdoberfläche auf eine zwei-dimensionale Landkarte projiziert wird. Dieses Problem ist nicht trivial, da sich eine Kugel nicht ohne Verzerrungen auf eine Ebene abbilden lässt. Es gibt eine große Anzahl von Projektionsverfahren, die alle ihre Vor- und Nachteile besitzen. Man unterscheidet hierbei Winkel- Flächen- und längenneutrale Projektionen.

Im OpenStreetMap Projekt wird die Mercator Projektion verwendet. Diese aus dem Beginn der Hochseefahrt stammende Projektionsart hat den Vorteil, dass die Erde auf eine quadratische Fläche projiziert werden kann. Zudem ist sie winkelneutral, was bedeutet, dass eine auf der Karte abgelesene Richtung mit der realen Himmelsrichtung übereinstimmt. Der große Nachteil ist die zum Teil sehr große Verzerrung der Fläche und Länge. Bei dieser Projektion wird die Erde auf einen die Erdkugel umfassenden Zylinder projiziert und dieser Zylinder anschließend sozusagen abgewickelt. Die Achse des Zylinders ist dabei identisch mit der Rotationsachse der Erde. Das führt zu sehr großen Verzerrungen der Flächen und Längen, je näher man den Polen kommt.

1 Auch eine Kugel ist ein Ellipsoid, bei der beide Radien identisch sind.

Um Verzerrungen zu minimieren und trotzdem ein globales System für die Projektion und Beschreibung von Positionen auf der Erde zu erhalten, wurde die "Universale Transverse Mercator" (UTM) Projektion entwickelt. Bei einer transversalen Mercator Projektion liegt die Achse des Projektionszylinders in der Äquatorebene der Erde. Im UTM-Projektionsverfahren wird die Erde in mehrere verschiedene Längen-Zonen aufgeteilt. Je nach geografischer Länge wird ein unterschiedlich ausgerichteter Zylinder verwendet und die Verzerrung für jede der Längen-Zonen minimiert. Das Problem bei diesem Verfahren ist, dass Projektionen von zwei nebeneinanderliegenden Zonen nicht mehr nahtlos aneinander passen. Der große Vorteil gegenüber der Mercator Projektion ist die sehr geringe Verzerrung von Längen. [Vgl. Utm89]

A.1.4 UTM-Koordinaten

Für die UTM-Projektion wurde ein eigenes Koordinaten-System entwickelt. Diese UTM-Koordinaten bestehen aus der Angabe der Hemisphäre, also Nord oder Süd, der Angabe der Längen-Zone und aus einer Meterangabe für die nördliche und die östliche Position. Die Referenz für die östliche Meterangabe ist der Mittelmeridian der Längenzone, wobei die Referenz um 500.000m nach Westen versetzt wird ("False Easting"). Dadurch verhindert man negative Werte. Die Referenz für die nördliche Meterangabe ist der Äquator, wobei die Referenz für Positionen auf der südlichen Hemisphäre auf 10.000.000m gesetzt wird ("False Northing"). Dadurch werden auch hier negative Werte vermieden. Die TFH in Berlin hat ungefähr folgende UTM-Koordinaten: Northing = 5823000 Meter, Easting = 388200 Meter, Längen-Zone = 33 und Hemisphäre = Nord (WGS84).

A.2 OpenStreetMap

Lizenzbestimmungen auf Kartenmaterial verhindern die Verbreitung und somit auch die Entwicklung neuer Anwendungen im Navigationsbereich. Im Projekt OpenStreetMap werden geografische Daten gesammelt und verarbeitet. Produkt ist eine Weltkarte, die unter der freien Creative-Commons-Lizenz [Cre] veröffentlicht ist. Somit ist das entstandene Kartenmaterial ohne Beschränkung verwendbar. Jeder kann an der Aktualisierung und Erweiterung dieser Landkarte mitwirken. Langfristig besteht die Chance, eine für alle Internet-Nutzer zugängliche Karte zu erhalten, die kostenlos, umfassend und aktuell ist.

OpenStreetMap benötigt viele Benutzer, die aktiv geografische Daten einpflegen. Diese Daten beschreiben zum Beispiel noch nicht erfasste Wege oder Objekte. Als Werkzeuge dienen alle Arten von GPS-Geräten mit denen automatisch oder auch manuell geografische Punkte aufgezeichnet werden können. Mit Hilfe dieser Daten und spezieller Software können nun relevante Objekte in die OSM-Karte eingezeichnet werden. Oft sind bei

diesen Daten weitere Zusatzinformationen hilfreich, die über reine Koordinatenlisten hinausgehen.

A.2.1 Erstellung der Landkarte

Ähnlich wie bei der freien Enzyklopädie Wikipedia kann jeder, der über relevante Daten verfügt, an der Erzeugung der OpenStreetMap-Weltkarte mitwirken. Die Basis dieser Daten sind die sogenannten Traces[1]. Diese werden mit GPS-Geräten aufgezeichnet und anschließend in einem Karteneditor als Grundlage für das Zeichnen der Karte verwendet.[2]

Traces sind eine zeitlich geordnete Menge von geografischen Punkten, die somit die geografische Lage und Form von Objekten beschreiben können. In den OSM-Karteneditoren werden bereits erfasste Objekte als Vektorgrafiken und importierte Traces als Punktmengen dargestellt. Verfügt der Benutzer nun über ausreichend Zusatzinformation, zum Beispiel Straßennamen, ist es möglich, die Karte um neue Objekte zu ergänzen. Nach Erzeugung der neuen Vektorgrafiken wird das Resultat an den OSM-Server übermittelt. Der OSM-Kartenserver kann diese Vektorgrafiken dann in Rastergrafiken umwandeln.

Somit können drei verschiedene Datentypen unterschieden werden: Geodaten, Vektordaten und Rasterdaten.

A.2.2 Geodaten

Die Geodaten beinhalten geografische Koordinaten, aber auch weitere relevante Informationen. Nur allein geografische Positionen und Höhenangaben reichen für das Zeichnen der Karte nicht aus. Zusätzlich sind Text-Informationen über das begangene Objekt notwendig.

Geodaten können auf vielfältige Art und Weise gespeichert werden. Einige OpenStreetMap Benutzer zeichnen zwar zurückgelegte Wege als Traces mit GPS-Geräten auf, notieren allerdings zusätzliche Informationen mit Zettel und Bleistift.

Die meisten GPS-Geräte nutzen äußerst technische Dateiformate für die Aufzeichnung von GPS-Daten. Hier sei das Format NMEA [DeP08] erwähnt, das diverse Informationen über den GPS-Empfänger beinhalten kann, allerdings keine Möglichkeit besitzt, zusätzliche textuelle oder gar bildliche Informationen zu integrieren.

Das GPS-Exchange-Format(GPX) [GPX08] ist ein XML basiertes Datei-Format zur Speicherung von GPS-Daten. In diesem Dateiformat werden die Daten in Form von Wegpunkten und Traces gespeichert. Jeder Datentyp verfügt über eine eigene Menge von Informationen. Zu diesen Informationen gehören zum Beispiel geografische Koordinaten, Zeitangaben, Textinformationen oder auch referenzierte Bilder. Dieses Dateiformat ist sehr

1 Trace (englisch) bedeutet soviel wie Spur oder Verfolgung
2 Es gibt drei offizielle OSM-Karteneditoren: JOSM [Jos], Merkaartor[Mer] und Potlatch[Pot].

verbreitet. OpenStreetMap nutzt es unter anderem für die Archivierung der Traces. Da auch die OSM-Karteneditoren dieses Dateiformat unterstützen, ist es für eine Speicherung der Geodaten in Anwendungen für das OSM-Projekt sehr geeignet.

A.2.3 Vektordaten

Die Vektordaten beschreiben die Art und Weise, wie Objekte der Karte gezeichnet werden sollen. Geodaten müssen für das Zeichnen einer OSM-Karte in Vektordaten transformiert werden. Dies geschieht im allgemeinem per Hand. In diesen Vektorzeichnungen hat der Anwender bestimmte Objekte innerhalb der Karte positioniert und ihm zu der Lage auch einen Typen, einen Namen und eventuell noch weitere beschreibende Informationen zugeordnet. Dadurch wird genau festgelegt, wo welche Linie, welche Fläche oder welcher Text in welcher Farbe auf der Karte dargestellt werden soll. Die Vektordaten des OpenStreetMap Projektes sind im übrigen frei verfügbar, was bei kommerziellen Landkartenprojekten zumeist nicht der Fall ist.

A.2.4 Rastergrafiken

Die Rastergrafiken beschreiben ein Bitmap Raster, also eine digitale Bilddatei. Sie besitzen eine feste Anzahl von Pixeln und haben im Gegensatz zu den Vektorgrafiken den Vorteil, dass sie nicht mehr für die Darstellung rasterisiert werden müssen. Allerdings benötigen sie je nach Auflösung im Allgemeinem wesentlich mehr Speicherplatz und sind qualitativ an eine bestimmte Auflösung gebunden. Mehr Qualität benötigt mehr Speicherplatz.

Im Allgemeinem werden für unterschiedliche Darstellungsgrößen mehrere Rasterbilder den gleichen Bereich abdecken. Dadurch lässt sich dann ein Vergrößern des Ausschnittes in Stufen umsetzen, ohne deutlich an Qualität zu verlieren. Auch für die Rastergrafiken muss bei Landkarten eine Georeferenzierung vorliegen, um sie für Navigationssoftware verwenden zu können.

Die durch den OpenStreetMap-Server aus den Vektordaten berechneten Rastergrafiken der Weltkarte werden in kleinen Teilbildern gespeichert. Bei der Darstellung der Weltkarte auf einem PC, im allgemeinem in einem Browser, ist nur ein bestimmter Ausschnitt der Weltkarte sichtbar. Um nicht die gesamte Karte zum Benutzer übermitteln zu müssen, erhält er nur die Teile der Weltkarte, die er im Moment gerade benötigt.

A.2.5 Slippy-Map Tileserver

Der OpenStreetMap Karten-Server “Slippy-Map” [Sli] liefert die gerasterten Kartenteile (Tiles). Die Tiles sind Bilder des Typs PNG, wobei jedes Bild 256 x 256 Pixel groß ist. Ein Tile kann über die folgende URL abgerufen werden:

http://tile.openstreetmap.org/**zoom**/**x**/**y**.png

Hierbei beschreibt der Wert **zoom** den aktuellen Zoomlevel, und **x** sowie **y** beschreiben die Koordinaten des Tiles.

Ein bestimmter Algorithmus kann verwendet werden, um zu bestimmen, welches Tile eine geografische Position abdeckt. Dieser basiert auf Berechnungen die sich durch die Mercator Projektion ergeben. [Sli08]

Es existieren 18 verschiedene Zoomlevel. Bei einem Zoomlevel von eins ist die Weltkarte in 4 Bilder aufgeteilt. Sie hat also eine Gesamtauflösung von 512 x 512 Pixeln. Das entspricht am Äquator einer Detailstufe von ungefähr 78 Kilometern pro Pixel. In der höchsten Detailstufe entspricht ein Pixel am Äquator in etwa 0.5 Metern. Dann ist die gesamte Weltkarte in 262144 x 262144 Kacheln aufgeteilt. Das ist eine äußerst große Datenmenge, die natürlich nicht auf einmal geladen und verarbeitet werden kann.

A.3 Georeferenzierte Rasterbilder

Eine eingescannte Landkarte besitzt zunächst keine digitale Georeferenzierung, womit sie für einen Einsatz in einer Navigationssoftware zunächst nicht verwendbar ist. Auf Landkarten sind allerdings geografische Bezugspunkte eingezeichnet. Mit dem Wissen über die verwendete Projektion, das geodätische Datum und diese Bezugspunkte kann eine Abbildung von den Pixelkoordinaten der Rasterkarte auf die Realweltkoordinaten vorgenommen werden. Dazu ist eine Mindestmenge von georeferenzierten Punkten notwendig.

Die geografischen Koordinaten können, mittels auf Kartenprojektion und Datum angepassten Algorithmen, für jeden Pixel und Subpixel der Rastergrafik ermittelt werden. Für einige Kartenprojektionen, wie zum Beispiel UTM, sind diese Algorithmen relativ einfach über eine Interpolation umzusetzen. Da UTM-Projektionen annähernd längenneutral sind, kann zwischen gegebenen georeferenzierten Punkten über lineare Interpolation für jeden beliebigen Punkt auf der Karte die Realweltkoordinaten ermittelt werden.

B Anhang: TribOSMap Dateiformate

B.1 Dateiformat TOSM für Rastergrafiken

Die Datei TOSM hält eine Menge von JPEG-Bildern vor. Diese Bilder sind Teile eines Gesamtbildes, wobei für jeden Zoomlevel Teile erzeugt wurden. Von allen Teilbildern eines Zoomlevels wird ein gemeinsamer Dateibeginn ermittelt. Also Daten, die sich in allen Teilbildern des Zoomlevels nicht unterscheiden und am Beginn dieser Teilbilder liegen.

Die einzelnen Einträge von A bis D (außer D.5) sind integer (little-endian). Die Datenblöcke D.5 und die Einträge in E sind Teile von JPEG-Bildern. Um ein Teilbild (JPEG) zu rekonstruieren, muss seinem Dateibeginn aus D.5 sein Datenblock aus E angehangen werden (gleicher Zoomlevel).

Im Folgenden die Spezifikationen für dieses Dateiformat. Dabei wiederholen sich die Einträge D.* so oft wie Zoomlevel in der Datei vorhanden sind.

Tabelle B.1: Das Dateiformat TOSM

Name	Bytes	Kommentar
A	4	x-Dimension des Gesamtbildes
B	4	y-Dimension des Gesamtbildes
C	4	Anzahl der gespeicherten Zoomlevel
D.1	4	Anzahl der Teilbilder je Zeile im Zoomlevel
D.2	4	Anzahl der Teilbilder je Spalte im Zoomlevel
D.3	4	Länge gemeinsamer Daten am Beginn der Teilbilder im Zoomlevel
D.4	4*D.1*D.2	Offsets aller Restdaten der Teilbilder des Zoomlevels
D.5	D.3	gemeinsame Daten (Dateibeginn) der Teilbilder des Zoomlevels
D.*		C-1 fache Wiederholung von D.1 bis D.5
E		Alle Restdaten (Teilbilder ohne D.5)

B.2 Dateiformat zur Georeferenzierung von TOSM-Dateien

Die TOSM-Dateien werden in einer separaten XML-Datei (TOSM-XML) georeferenziert. Diese Datei wird für das Verwenden einer TOSM-Datei von der Anwendung TribOSMap benötigt.

Hier die DTD des TOSM-XML Formates:

```
<!-- Georeferenzierung für TOSM Dateien-->
<!-- map : Root -->
<!ELEMENT map ( file, datum, coordsystem,
                projection, size, points ) >
<!-- name : Name der Karte -->
<!ATTLIST map name NMTOKEN #REQUIRED >
<!-- file : Pfad zur TOSM-Datei -->
<!ELEMENT file ( #PCDATA ) >
<!-- datum : Datum (momentan nur WGS84) -->
<!ELEMENT datum ( WGS84 ) >
<!-- coordsystem : Koordinatensystem (momentan nur UTM/WGS84) -->
<!ELEMENT coordsystem ( WGS84 ) >
<!-- projection: Projektion der Karte (momentan nur UTM) -->
<!ELEMENT projection ( UTM ) >
<!-- size: Pixel Ausdehnung -->
<!ELEMENT size EMPTY >
<!-- x: Integer - Pixel x-Ausdehnung -->
<!ATTLIST size x NMTOKEN #REQUIRED >
<!-- y: Integer - Pixel y-Ausdehnung -->
<!ATTLIST size y NMTOKEN #REQUIRED >
<!--points:  Punkte für die Georeferenzierung (min. 2) -->
<!ELEMENT points ( point+ ) >
<!--point:  Punkt für die Georeferenzierung -->
<!ELEMENT point ( utm ) >
<!-- x: Integer - Pixel x-Koordinate -->
<!ATTLIST point x NMTOKEN #REQUIRED >
<!-- y: Integer - Pixel y-Koordinate -->
<!ATTLIST point y NMTOKEN #REQUIRED >
<!--utm: eine UTM-Koordinate, totaler Northing Wert -->
<!ELEMENT utm EMPTY >
<!--hemisphere: UTM-Hemisphäre -->
```

```
<!ATTLIST utm hemisphere (North|South) >
<!-- northing: Float - UTM-Northing (inkl. false Northing)-->
<!ATTLIST utm northing NMTOKEN #REQUIRED >
<!-- easting: Float - UTM-Easting (inkl. false Easting)-->
<!ATTLIST utm easting NMTOKEN #REQUIRED >
<!-- zoneumber: Integer - zwischen 1 und 60 inklusive -->
<!ATTLIST utm zonenumber NMTOKEN #REQUIRED >
```

Ein Beispiel des TOSM-XML Formates:

```
<map name="Hardangerviddawest">
    <file>./hardangerviddaWest.TOSM</file>
    <datum>WGS84</datum>
    <coordsystem>WGS84</coordsystem>
    <projection>UTM</projection>
    <size y="14682" x="10816"></size>
    <points>
        <point y="8512" x="1780">
            <utm northing="6670000" zonenumber="32"
                 easting="372000" hemisphere="North">
            </utm>
        </point>
        <point y="8237" x="8668">
            <utm northing="6670000" zonenumber="32"
                 easting="416000" hemisphere="North">
            </utm>
        </point>
    </points>
</map>
```

C Anhang: UML-Erweiterungen für Scala

Scala besitzt einige Besonderheiten, für die in UML keine sprachlichen Mittel zur Verfügung stehen. Aus diesem Grund war es notwendig, UML zu erweitern. Grundsätzlich orientieren sich die Diagramme in dieser Ausarbeitung an der UML-Spezifikation 2.1. Alle Ergänzungen oder Änderungen werden hier genannt und anhand von Diagrammen erklärt.

C.1 Attribute, Getter und Setter

Für alle veröffentlichten Attribute existieren implizite Getter (*attribute1* und *attribute2*), außer wenn die Sichtbarkeit als "privat" gekennzeichnet wurde (*attribute3*). Wenn veröffentlichte Attribute mit dem Stereotyp "Var" versehen sind, handelt es sich um veränderliche Attribute, für die dann ein impliziter "Setter" existiert (*attribute2*). Der Stereotyp "lazy" kennzeichnet Attribute, die als *lazy*-Variablen definiert sind (attribute4).

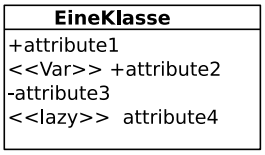

Abbildung C.1: Attribute

C.2 Klassen und Generizität

Generische Klassen werden wie in UML mit einem abstrakten Typ versehen. Zu diesem Typ kann auch eine untere oder obere Grenze angegeben sein. Dabei wird die sprachliche Syntax von Scala verwendet (*EineKlasse*). Es können auch strukturelle Typen als Grenze verwendet werden, dessen Attribute und Methoden in geschweiften Klammern spezifiziert werden(*EineKlasse2*). Generische Typen von polymorphen Methoden werden als zusätzliche Parameterliste in eckigen Klammern angegeben (*operation*).

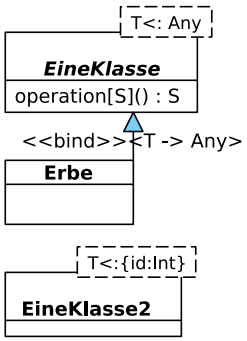

Abbildung C.2: Klassen

C.3 Traits

Traits werden grundsätzlich wie abstrakte Klassen dargestellt, allerdings mit dem Stereotyp "trait" versehen. Sockets, also abstrakte Attribute und Methoden, werden kursiv dargestellt (*attribute2*, *operation2*). Wurden Sockets als abstrakte Attribute spezifiziert, kann ein Abhängigkeitspfeil mit dem Stereotypen "requires" versehen werden (*Trait3*, *attribute3*). Dagegen wird für selbstreferenzierte Typen der Stereotyp "self" verwendet (*Trait4*). Beerbt ein Trait einen anderen Trait, wird die Vererbung mit einem gestrichelten Vererbungspfeil gekennzeichnet (*Trait2*). Dieser Pfeiltyp wird auch verwendet, wenn eine Klasse Traits einmischt(*Klasse*). Um das Einmischen von Traits deutlicher hervorzuheben, wird kann der Abhängigkeitspfeil mit dem Stereotypen "mixin" versehen werden (*Klasse*).

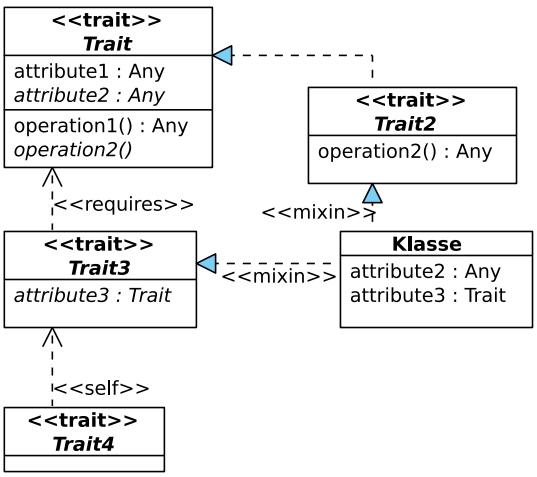

Abbildung C.3: Traits

C.4 Singleton Objekte

Singleton-Objekte werden wie Klassen dargestellt und mit dem Stereotyp "singleton" versehen. Somit kann es vorkommen, dass auf den ersten Blick zwei Klassen mit dem gleichen Namen vorhanden sind. Dann handelt es sich aber um ein Singleton-Objekt und seine zugehörige Kompagnon-Klasse. In diesem Fall wird immer ein Abhängigkeitspfeil mit dem Stereotyp "hasA" verwendet.

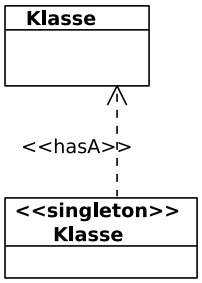

Abbildung C.4: Singleton