Beitrag H: Christine Müller

Entwurfsmuster in Android-Apps am Beispiel von Apps für die Forstwirtschaft

Christine Müller, Inforst UG (haftungsbeschränkt), mueller@inforst.de

Abstract

It is easy to develop an Android-App and even beginners in programming can create an Android-App fast. Therefore, the question arises, it is worth spending time on design and architecture of such lightweight programs. But the frequency with which new Android versions are released and the changing demands of the users makes it just as necessary for apps to be flexible and extendable. Inforst develops apps for forestry that run completely offline. They are just as complex as desktop applications which meet the same tasks of collection, storage and exchange of timber data. The price for apps is far below the price for comparable desktop applications though. This puts even higher pressure on developers to develop reusable software components and reduce the costs for testing and refactoring. To fulfill these demands the use of Design Patterns can be helpful as is shown in this article at the example of the app "WaldFliege"[15] used to collect timber data.

Zusammenfassung

Der Entwicklung von Android-Apps für Smartphones und Tablets unter Verwendung des Android-Frameworks ist leicht zu erlernen. Lauffähige Anwendungen können relativ schnell entwickelt werden. Da stellt sich die Frage, ob Überlegungen zur Architektur und Entwurf überhaupt eine Rolle spielen. In Hinblick auf die kurze Lebensdauer von Android-Versionen und die wandelnden Ansprüche der Nutzer ist es jedoch für

[15] http://www.inforst.de/de/apps/waldfliege/waldfliege-information.html

jede App notwendig, wartbar und erweiterbar zu sein. Inforst entwickelt komplett offline laufende Apps für die Forstwirtschaft. Diese unterscheiden sich in ihrer Komplexität nicht von Desktop-Applikationen mit den gleichen Aufgaben der Holzdatenaufnahmen, -verwaltung und -weitergabe. Die Zahlungsbereitschaft für Apps liegt aber weit unter der für Desktopanwendungen. Umso bedeutender ist es wiederverwertbare App-Bausteine zu entwickeln und den Aufwand für Erweiterungen und Tests so gering wie möglich zu halten. Dabei können die klassischen Entwurfsmuster der objektorientierten Softwareentwicklung eine Hilfe sein. Dies wird in diesem Beitrag anhand der Holzaufnahme-App „WaldFliege“ gezeigt.

1 Rahmenbedingungen für die Softwareentwicklung in der Forstwirtschaft

Die Forstwirtschaft ist eine traditionelle Branche, in der regionale Besonderheiten und überlieferte Geschäftsabläufe eine wichtige Rolle spielen. Die Holzproduktion entzieht sich aus verschiedenen Gründen der in der Industrie verbreiteten Normierung.

Gründe für fehlende Normierung in der Forstwirtschaft nach [OESTEN2001] sind:

- Lange Produktionszeiträume (< 60 Jahre)
- Starke Abhängigkeit vom natürlichen Standort („Das eiserne Gesetz des Örtlichen“)
- Kuppelproduktion: Das Fällen eines Baumes erzeugt mehrere Produkte (z.B. Stammholz und Industrieholz aus der Krone)

Daher verwenden unterschiedliche Betriebe verschiedene Begriffe für dasselbe Objekt (z.B. Kiefer oder Föhre). Neben natürlichen regionalen Unterschieden ergeben sich aufgrund der Einkaufbereiche der europäischen Großsägewerke unterschiedliche Marktverhältnisse, die zum Teil dazu führen können, dass große Kunden die Geschäftsabläufe stark beeinflussen. Daneben kommt es aufrgrund der großen Vielfalt von Forstbetrieben hinsichtlich Betriebsgröße, Eigentumsart und Rechtsform zu unterschiedlichen Geschäftsabläufen [OESTEN2001]. Ein Versuch der Normierung des Rohholzhandels ist die Rahmenvereinbarung für den Rohholzhandel in Deutschland [RVR 2014]. Sie dient als Grundlage für die Geschäftslogik und die Defaulteinstellungen der „WaldFliege“. Da sie jedoch nicht überall umgesetzt wird, sind

die Werte änderbar. Insgesamt gibt es also nur wenig Fakten, die Entwickler von Forstsoftware als allgemein gegeben annehmen können. Daneben gibt es eine Reihe von Datengebilden, die in einer Regeion oder einem Betrieb als statisch angenommen werden können. Diese kann der Anwender der „WaldFliege“.einmal eingeben und dann immer so verwenden. Die Vielzahl von Produkten und Geschäftsabläufen führt dazu, dass Forstsoftware einen hohen Grad an Komplexität hat. Dabei wird der Anteil einer Software für das Erreichen des Betriebsergebnisses als gering eingeschätzt. [LEMM 2002]. Dementsprechend sind Forstbetreibe weniger bereit in Softwareprodukte oder die Entwicklung von individueller Software zu investieren.

1.1 Geschäftsabläufe: Forstbetriebsgemeinschaft versus mittlerer Privatwaldbetrieb

Als Beispiel für die unterschiedlichen Anforderungen an die Funktionsweise eine „WaldFliege“ vergleichen wir eine Forstbetriebsgemeinschaft (nachfolgend FBG genannt) mit einem mittleren Privatwaldbetrieb.

Bedingungen Holzaufnahme in einer FBG:

- Bis zu mehrere tausend Waldbesitzer
- Erstaufnahme durch viele, nicht forstlich ausgebildete Waldbesitzer
- Überprüfung durch Mitarbeiter der FBG
- Weiterleiten an Kunden/Spediteure

Dadurch, dass die Holzaufnahme nicht immer durch Fachleute vorgenommen wird, muss das Formular für die Holzaufnahme sehr viel robuster gestaltet werden. Es werden mehr Plausibilitätsprüfungen notwendig und der Nutzer darf nicht so viele Freiheiten haben, die aufgrund von mangelndem Wissen zu Fehlern führen können. Außerdem darf jedes Mitglied der FBG nur auf seine eigenen Daten Zugriff haben. Die Mitarbeiter der FBG brauchen jedoch Zugriff auf alle Daten innerhalb der FBG. Die Zuordnung des Holzes zu einem bestimmten Waldbesitzer ist von entscheidender Bedeutung. Die Kunden oder Spediteure dürfen jedoch nur bestimmte Informationen einsehen. Demgegenüber stehen die Anforderungen eines Privatwaldbetriebs, der um die langfristige Entwicklung seiner Bestände zu überprüfen eine Zuordnung des Holzes zum Bestand bzw. zur Abteilung benötigt.

Die Bedingungen für die Holzaufnahme in einem mittleren Privatwaldbetrieb sind:

- 1-5 Waldbesitzer, aber Unterscheidung in Abteilungen
- Holzaufnahme durch Fachpersonal (1-5 Personen)
- Überprüfung durch Mitarbeiter der FBG
- Weiterleiten an Kunden/Spediteure

1.2 Holzaufnahme und Holzdatenverarbeitung - Methoden

Zu den wenigen als allgemein angenommenen Voraussetzungen für die Holzaufnahme App gehört die hierarchische Ordnung der Holzdaten. Ein Hieb kann mehrere Lose haben, die aus mehreren Poltern (aufeinander liegende Holzstämme) bestehen können. Je nach Holzaufnahmeverfahren befinden sich auf der Ebene darunter Einzelstammaufnahmen, Sektionsmaße, Strichprobenaufnahmen oder Schätzmaße. Strenggenommen bildet eine Ansammlung von Einzelstämmen nicht unbedingt einen Polter, weil die Stämme auch nebeneinander liegen können. Von der Anwendungslogik her ist das jedoch ohne Bedeutung, denn auch für eine Ansammlung nebeneinander liegender Stämme ist die Angabe eines Waldortes über GPS-Daten erforderlich, um die Abfuhr zu organisieren (s. Abbildung H-1).

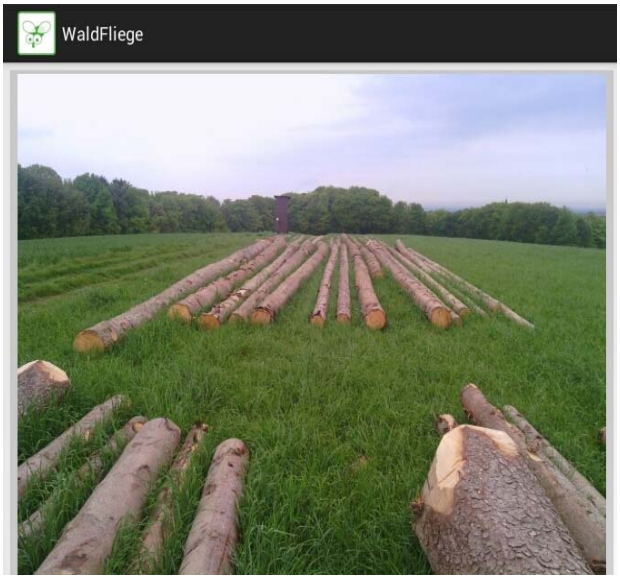

Abbildung H-1: Ansammlung von Einzelstämmen (Foto in der „WaldFliege“), in der Realwelt kein „Polter“ in der Logik der App schon

1.2.1 Einzelstammaufnahme

Bei jedem Stamm werden Länge und Durchmesser gemessen.Das Volumen errechnet sich aus der Formel: $V = (\pi/4) * d^2 * L$ (Hubersche Formel), wobei d der Durchmesser in Meter auf zwei Nachkommastellen genau ist und L die Länge in Metern. Eine Besonderheit bei der Einzelstammaufnahme ist der Klammerstamm, bei dem ein Stamm in zwei Abschnitte mit unterschiedlichen Güteklassen aufgeteilt wird. Dadurch kann die Zahl der Einzelstämme die Zahl der Stämme überschreiten. Der Zusammenhang zwischen den Klammerstammstücken darf bei der Datenverarbeitung nicht verloren gehen.

1.2.2 Sektionsraummaß

Aus den Messgrößen Länge, Höhe und Tiefe des Polters, reduziert um das Raumübermaß (im Regelfall 4%), wird das Raummaß in Raummeter in Rinde ermittelt. Dafür wird der Polter in gleichmäßige Sektionen aufgeteilt und in der Mitte der Sektionen über Höhenmessungen die mittlere Polterhöhe bestimmt. Ungleichmäßige Polter werden dabei in mehrere Teile aufgeteilt (siehe Abbildung H-2).

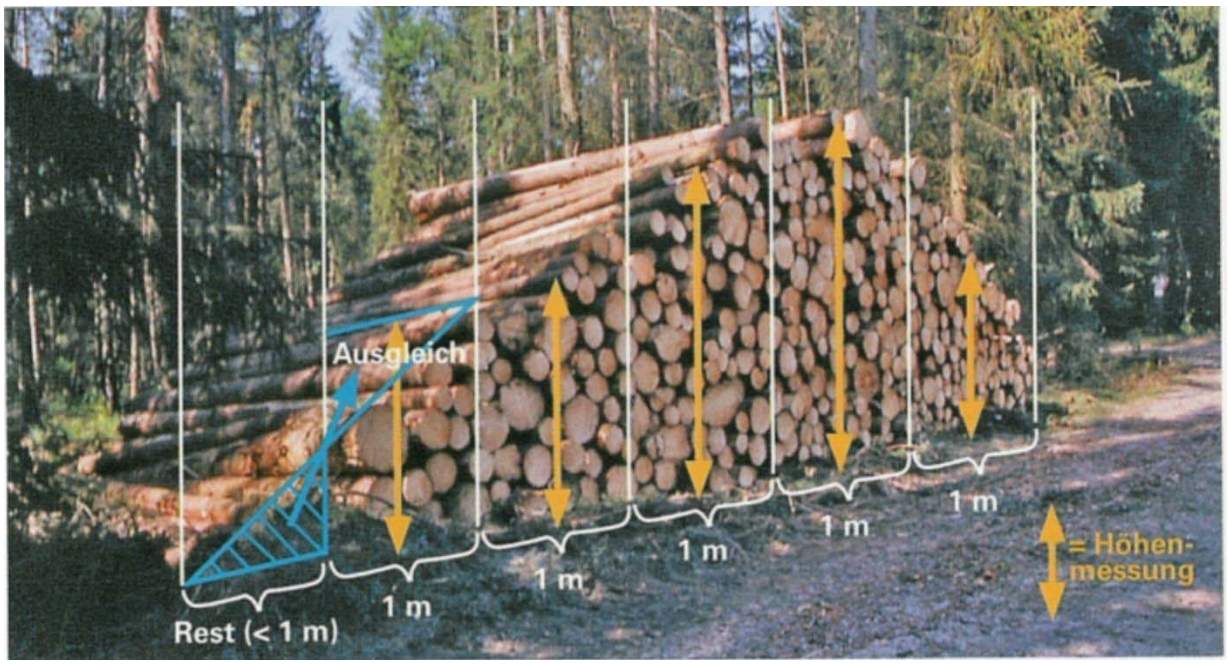

Abbildung H-2: Sektionsverfahren

Bildquelle: Amt für Landwirtschaft und Forsten Landau an der Isar

Die Herausforderung besteht darin, mit Messlatte und Sprühdose in der Hand, die Werte ins Android-Gerät einzugeben. Daher wurde dieses Formular so konzipiert, dass es nur mit einer Hand bedient werden kann (siehe Abbildung H-3).

Abbildung H-3: Einhändig bedienbares Eingabeformular der „WaldFliege“für das Sektionsverfahren

1.2.3 Stichprobenaufnahme

Die Stichprobenaufnahme erfolgt nach dem Stirnflächenverfahren (siehe Abbildung H-4). Dies besagt, dass anstatt jeden Einzelstamm zu messen nur von einem bestimmten Anteil der Stämme (in der Regel 25%) der Durchmesser bestimmt wird. Dafür wird entweder vor bzw., während des Polterns der Mittendurchmesser aufgenommen oder nach dem Poltern vorne und hinten die Stirnflächendurchmesser. Die Güte der gemessenen Stämme wird festgehalten. Da es sich in der Regel um Fixlängen handelt, ist die Länge bekannt. Anhand der Stichprobe wird die Güteklassen und Stärkeklassenverteilung des Polters bestimmt. Man errechnet den Volumenmittelstamm und kann so nach Zählen der Stämme auf den Gesamtpolter hochrechnen.

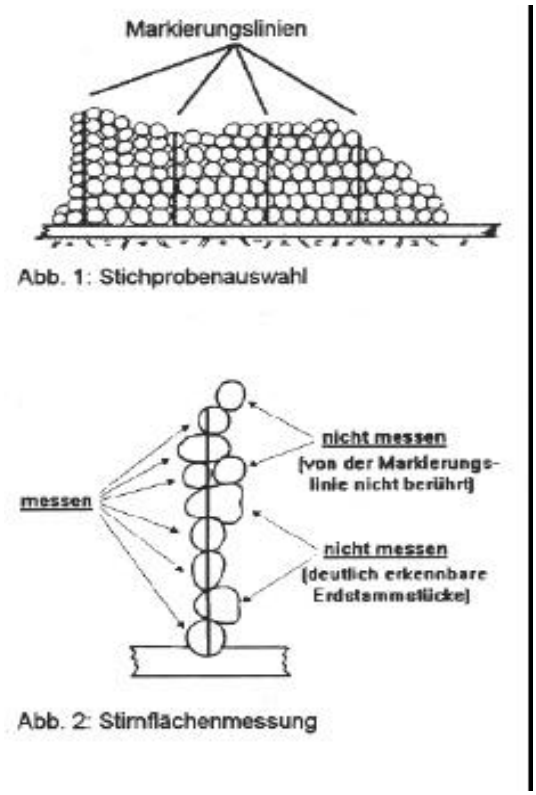

Abbildung H-4: Stirnflächenverfahren

Bildquelle: Merkblätter der Forstlichen Versuchs- und Forschungsanstalt Baden-Württemberg 49/11997

2 Objektorientierte Entwurfsmuster und Ihr Einsatz in der Holzaufnahme-App

2.1 Objekte und die SQLite-Datenbank

Daten für Android-Apps können platzsparend lokal auf dem Gerät in einer sqlite-Datenbank abgelegt werden. Dies ist vor allem aufgrund der benötigten Offline-Lauffähigkeit das Mittel der Wahl. Einige Eigenschaften der SQLite-Datenbank müssen für die Weiterentwicklung der „WaldFliege“ jedoch ausgeglichen werden. So erfordert die Veränderung der Tabellenstruktur innerhalb der Datenbank eine eigene Funktion für den Datenbank Upgrade (und Downgrade). Dabei wird gerüft, ob alle benötigten Tabellen vorhanden sind. Ggf. nicht vorhandene Tabellen werden neu erstellt und falls nötig mit Ausgangswerten ausgestattet. Anhand der Erstellungsbefehle wird geprüft, ob alle benötigten Spalten vorhanden sind. Fehlende Spalten können ergänzt werden. So kann der Nutzer, unabhägig von seiner aktuellen

Datenbankversion immer ohne Datenverlust auf die aktuelle Datenbank upgraden. Leider besteht im Moment innerhalb von SQLite nicht die Möglichkeit, überflüssige Spalten zu löschen. Dies kann gelöst werden, indem in Zukunft beim Upgrade eine Kopie der Datenbank erstellt wird und anschließend nur die benötigten Spalten in die neue Datenbank kopiert werden.

So effizient wie Objekte in der Verwaltung der Nutzerinteraktion und in der Ausführung der Geschäftslogik sind, so ungeschickt lassen sie sich in Datenbanken abbilden. Deshalb erfordert der Weg von der Datenbank zum Nutzer einige architektonische Überlebungen.

2.1.1 Von der Datenbank zum Nutzer

Für die Objekte innerhalb der „WaldFliege“ wurde je eine datenbanknahe Klasse (**Data Transfer Object)** entworfen, die der Tabelle, in der die Daten gespeichert werden, entspricht. Diese Klassen enthalten nur Werte, keine Methoden. Die Implementierung als **Data Transfer Object** ermöglicht auch ein Hochladen der Objekte auf einen Server, sobald die App wieder online ist. Dies ist der ursprüngliche Ansatz für die Verwendung von Data Transfer Objects, den Aufwand von Remote-Zugriffen zu reduzieren. [EILEBRECHT 2010]

id	Hiebnr	Losnr	Polternr	Baumart	Sorte	Guete	Volumen	Volumeneinheit
123	Blauberg	12	1	Fichte	FL	B	25	fm

Tabelle H-1: Ausschnitt aus der Tabelle „Polterliste“ der SQlite-Datenbank

Polter
Hiebnr: String Losnr : String Polternr : int Baumart : String Sorte : String Güte : String Volumenerfassung : String Volumeneinheit : String Volumen : String Stück : String Waldbesitzer : String GPSE : String GPSN : String

Abbildung H-5: Ausschnit aus dem Klassenmodell der Klasse „Polter“ als Data Transfer Object[16].

Für jede datenbanknahe Klasse gibt es ein **Data-Acccess-Object** (nachfolgend DAO genannt), den **Fachgebietsmanager**, der die Speicherung und Verwaltung innerhalb der Datenbank regelt, Abfragen in der Datenbank durchführt und einen gewünschten Datensatz als datenbanknahes Objekt bereitstellt.

Der Manager greift dabei über eine Schnittstelle auf eine einzige Instanz der Klasse **DatenbankHelper** zu, die von der Klasse **SQLiteDatabaseHelper** abgeleitet ist. Die Klasse **SQLiteDatabaseHelper** ist das innerhalb von Android vorgesehene Instrument zur Verwaltung der SQLiteDatenbank(siehe Abbildung H-6).

[16] Alle Attribute, die in der Datenbank vorkommen werden als private Attribute mit Gettern und Settern umgesetzt, so dass hier die Sichtbarkeit weggelassen wird.

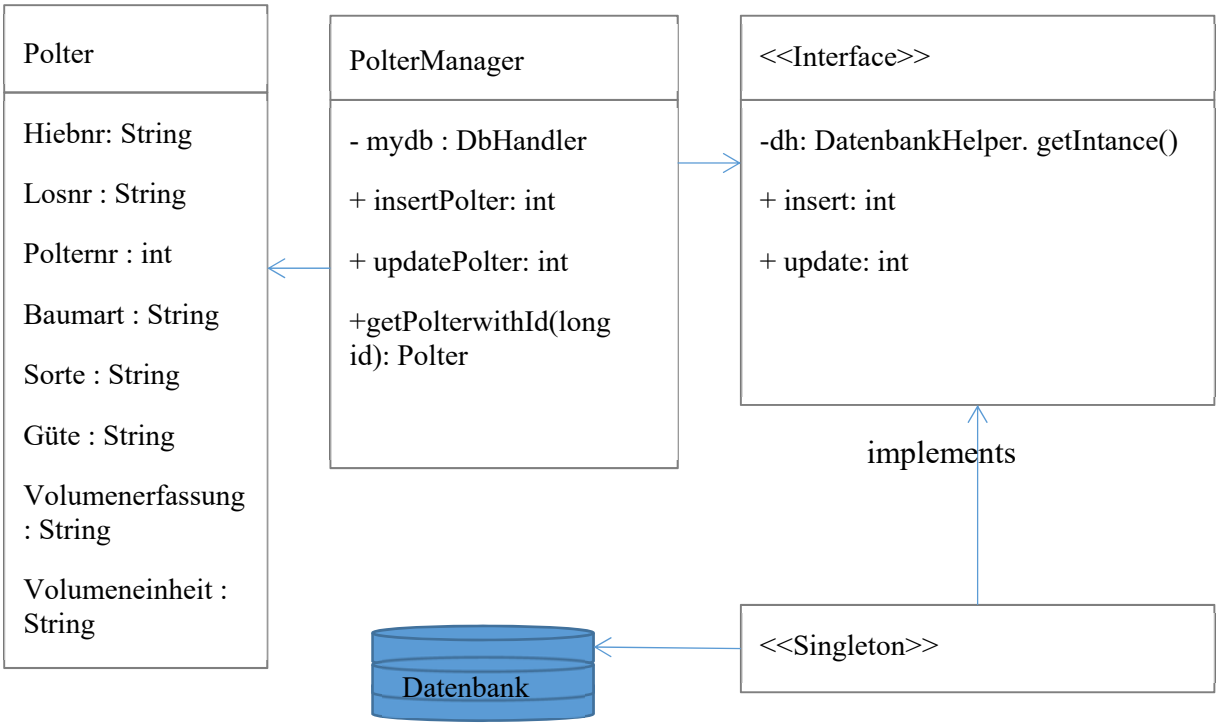

Abbildung H-6: Ausschnitt UML Klassendiagramm für Objektmanagement

Die Managerklasse erzeugt die benötigten Objekte, indem sie deren Daten aus der Datenbank ausliest. Sie steuert die Speicherung und Veränderung der Datensätze über eine Schnittstelle auf die einzige Instanz des DatenbankHelpers, der direkt auf die Datenbank zugreift.

Durch diesen klar definierten Zugriff auf die einzige Instanz kann erreicht werden, dass alle Schreibaktionen hintereinander ausgeführt werden. Daneben kann es noch mehrere Helferklassen für die Durchführung der Geschäftslogik außerhalb der Datenbank (z.B. Berechnungen, Darstellungen) geben. Innerhalb der Activity wird also vom Fachgebietsmanager der gewünschte Datensatz aus der Datenbank als Objekt bereitgestellt. Über die Benutzeroberfläche kann der Nutzer Werte des Objekts verändern und über den Fachgebietsmanager Abfragen ausführen und den Datensatz speichern oder aktualiseren. Der Fachgebeitsmanager übernimmt auch, ggf. mit Hilfe weiterer Klassen, die Plausibititätsprüfung des veränderten Objekts.

2.2 Model-View-Controller

Das Android-Framework bietet in sich schon eine Umsetzung des Model – View-Controller – Prinzips. Die Datenbank oder der ContentProvider sind das „Model“, sie liefern die Daten. Die „Activity“ ist der „Controller“, sie verarbeitet die Nutzer-Interaktion

und ermöglicht das Abrufen, Verändern und Speichern von Informationen. Das xml-Layout ist schließlich die passive „View“ (siehe Abbildung H-7). Diese Realisierung bietet zahlreiche Vorteile, zum Beispiel in der Anpassung der Darstellung auf unterschiedliche Bildschirmgrößen oder Sprachen.

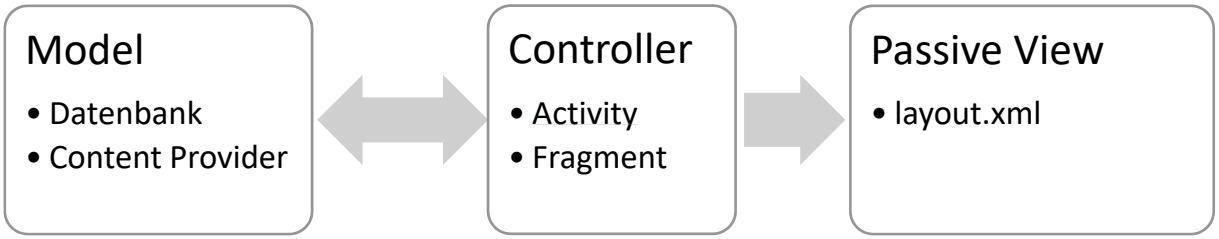

Abbildung H-7: Model-View-Controller in Android

Das Prinzip stößt jedoch bei komplexeren Apps an seine Grenzen, zum einen, wenn Bereiche der Nutzeroberfläche als Fragments gesondert verwaltet werden oder wenn viele Eingaben und daraus resultierende Berechnungen die Actvities sehr groß werden lassen. Dann wird die Activity schnell unübersichtlich. Dasselbe gilt, wenn alle Datenbankzugriffe in einer Klasse behandelt werden.

In der „WaldFliege“ kommen drei **Arten von Activities** vor:

- Formulare für die Dateneingabe,
- Listen für die Darstellung der Datenbankeinträge und
- Ansichten für Fotos oder Karten.

Durch die oben beschriebene Aufteilung von Datenhaltung und Geschäftslogik können die Formulare und Listen immer gleich aufgebaut sein, d.h. alle von einer Grundliste und einem Grundformular abgeleitet sein. Dies entspricht der **Schablonenmethode**. Die im Android-Framework angelegte Klasse „Activity“ wird also nach den Bedürfnissen erweitertert.

Methoden der Klasse „Formular“:

- setUpDefaultValues() : Werte initialisieren, benötigte DAOs schaffen
- gernerateObject(long id): Objekt aus DB holen oder neu generieren
- setUpGUI(): Benutzeroberfläche und Nutzerinteraktion festlegen

- initObject(): Benutzeroberfläche auf das gewünschte Objekt einstellen

Aufbau der Klasse „Liste“:

- verwendet Adapter (eigene Klasse): Der Adapter bereitet Daten aus Datenbank für Listenansicht vor.
- verwendet „ListListener“ (eigene Klasse): Der Listener verarbeitet die Nutzerinteraktion mit Hilfe des DAOS.
- verwendet DAO: Das Data-Access-Object kontrolliert den Datenbankzugriff und führt Änderungen in der Datenbank aus.
- updateList(): Die Methode updateList() paßt unter Verwendung des Adapters die Darstellung der Liste der veränderten Datengrundlage an.

Der **Adapter** bietet eine Anpassung der Daten an die Listenansicht. Die im Android-Framework vorhandenen Apater können hier leicht erweitert werden. Der ListListener ist ein klassischer **Observer**. Er nimmt die Aktionen des Nutzers innerhalb eines Listenelements wahr. Die Activity implementiert den Observer und leitet die entsprechenden Aktionen an das DAO weiter, dass die Änderungen in der Datenbank umsetzt. Anschließend wird mit updateList() die Liste aktualisiert.

2.3 Separation Of Concerns

Das Prinzip der Trennung der Verantwortlichkeiten besagt, dass unterschiedliche Aspekte eines Problems vorneinander getrennt werden sollen und jedes Teilproblem für sich behandelt werden soll. [GHARBI, 2013, S.102]. Oben wurde schon dargestellt wie bei der „WaldFliege“ technische und fachliche Teile getrennt wurden. Die Darstellung für den Nutzer und die Datenhaltung werden von der Geschäftslogik getrennt. Aber auch innerhalb der Fachlogik ist es von Bedeutung, unterschiedliche fachliche Aspekte durch verschiedene „Manager“-Klassen abzubilden. Zum Beispiel erfolgt die Berechnung des Volumens eines Polters nicht innerhalb des „PolterManagers“, sondern innerhalb einer, dem Aufnahmeverfahren entsprechenden „Manager“-Klasse. Beim „Einzelstamm-Manager“ wird ein „RindenabzugsManager“ für die Ermittlung der Rindenabzüge eingesetzt.

3 User Interaction

Im Vergleich zu Destop-Anwendungen müssen sich die Entwickler von Android-Apps aufgrund des Touch-Screen anders mit Fragen der Nutzerinteraktion auseinandersetzen. Sind auf einem Geräte-Bildschirm meherere übereinander liegende Objekte mit Touch-Schnittstellen vorhanden, muss für die korrekte Umsetzung des Touch-Events klar sein, auf welches Objekt der Nutzer gezielt hat und mit welcher Absicht (siehe Abbildung H-9). Innerhalb der Kartenanwendung „WaldKarte“ geschiet dies durch ein **Singleton** „NutzerWunsch“, das die Absicht des Nutzers in Zustandsvariablen speichert. Über alle Klassen und Unterklassen hinweg kann auf „NutzerWunsch“ zugegriffen werden und abhängig von den Variablen dieser Klasse unterschiedliche Interfaces für die Verarbeitung des Touch-Events verwendet werden. Für eine weitere korrekte Anwendungslogik ist es außerdem erforderlich, die Widgets des Android-Frameworks u. a. um ein Attribut zu erweitern, in dem festgehalten wird, ob das Widget vom Nutzer oder vom Programm auf einen bestimmten Wert eingestellt wurde.

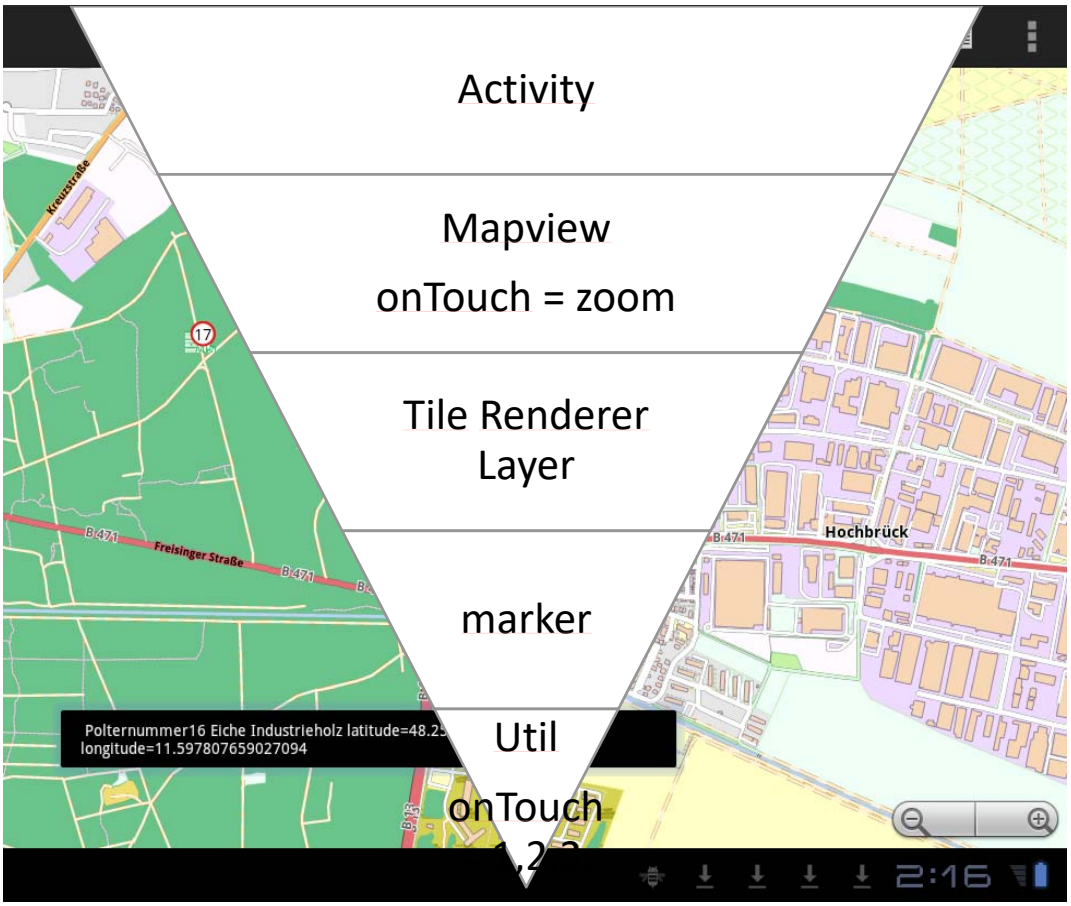

Abbildung H-9: Verarbeitung eines On-Touch-Events

4 Zusammenfassung

Überlegungen zur Software-Architektur und der Einsatz von Entwurfsmustern erhöhen die Wartbarkeit und Robustheit auch von Android-Apps. Viele Entwurfsmuster wie Model-View-Controller, Adapter, Observer, sind in ihren Grundzügen schon im Android-Framework angelegt und müssen für die konkrete Geschäftslogik nur angepasst und entsprechend erweitert werden. Durch die Aufteilung des Datenbankzugriffs nach Fachlogik können Änderungen schnell und ohne Auswirkungen auf andere Bereiche durchgeführt werden. Einmal getestete Datenbankzugriffe können innerhalb verschiedener Data-Access-Objects wiederverwendet werden. Die Datenbank kann durch einen ContentProvider (bei anwendungsübergreifenden Arbeiten) oder durch einen Server-Zugriff ersetzt oder ergänzt werden. Es ist dadurch auch möglich, die Auswirkung von Datenbankveränderungen zuerst an einer Datenbank-Fassade zu testen, bevor sie in der realen Datenbank umgesetzt werden.

5 Literaturverzeichnis

[EILEBRECHT 2010]

Eilebrecht, Karl, Starke Gernot: Patterns kompakt. Heidelberg 2010

[GHARBI, 2013]

Gharbi, Mahbouba; Koschel, Arne; Rausch, Andreas; Starke, Gernot: Basiswissen für Softwarearchitekten. Heidelberg, 2013.

[GOLL, 2013]

Goll, Joachim; Dausmann, Manfred: Architektur- und Entwurfsmuster der Softwaretechnik. Esslingen, 2013.

[LEMM 2002]

Lemm, Renato, Erni , Vinzenz, Thees, Oliver: Komponentenbasierte Softwareentwicklung – neue Perspektiven für forstliche Modellierung und Informationsverarbeitung Schweiz. Z. Forstwes. 153 Birmensdorf (2002) 1: 3 – 9.

[OESTEN 2001]

Oesten, Gerhard, Roeder, Axel: Management von Forstbetrieben, Remagen-Oberwinter 2001

[RVR 2014]

Deutscher Forstwirtschaftsrat e.V., Deutscher Holzwirtschaftsrat e.V.: Rahmenvereinbarung für den Rohholzhandel in Deutschland (RVR). Freiburg, 2004

[SCHMETZSTROFF, 2008]

Schmetzstroff, Martin: Bessere Software kompakt. Heidelberg, 2008.