Projektarbeit Advanced Software Engineering

Android App für Aufgabenmanagement

Studiengang Informatik

Duale Hochschule Baden-Württemberg Karlsruhe

Von

Fabian Braun

Abgabedatum:	31. Mai 2021
Betreuer der DHBW:	Mirko Dostmann
Kurs:	TINF18B4
Matrikelnummer:	2638189

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

Ziel dieser Projektarbeit war die Entwicklung einer Android App für einfaches Aufgabenmanagement in Form von einer verwaltbaren To-do-Liste.

Technologie:

- Entwicklungsumgebun Android Studio
- Programmiersprache Java für Logik, XML für Struktur der Oberfläche
- SQLite als offline Datenbank

Die Hauptfunktionalitäten sind dabei das Erstellen von Aufgaben mit möglichen Unteraufgaben (anstatt Notizen), welche auf der Oberfläche präsentiert werden. Aufgaben können abgeschlossen werden, wobei die benötigt Zeit getrackt werden kann. Die abgeschlossenen Aufgaben werden in einer separaten Liste angezeigt. Mit den Informationen zu Aufgaben und der benötigten Zeit sollen Statistiken generiert werden, welche die Zeit aller Aufgaben mit gleichem Namen aufsummiert anzeigen. Damit soll erkennbar sein, wie viel Zeit für die jeweiligen Aufgaben in einem gewissen Zeitraum aufgewendet wurde.

1.2 Anmerkung

Die Priorität des Projekts lag klar auf dem Einhalten der Vorgaben (1500 Zeilen und 20 Klassen) und dem Anwenden der verschiedenen Architekturprinzipien. Dabei wurden für viele Features zwar die Grundlagen im Backend gelegt, die UI kann diese jedoch nur beschränkt darstellen. Der Zeitliche Umfang für die Implemtierung der UI bzw. die Einarbeitung in das Android-Framework wurde leider unterschätzt. Einige Funktionen der App, wie das Anzeigen der offenen Aufgaben auf dem Sperrbildschirm wurden ausgelassen,

da diese nicht die Kerfunktionalitäten der App darstellen. Statistiken werden noch nicht angezeigt, das die Generierung funktioniert beweisen jedoch eine Vielzahl von Tests. Ein Problem konnte nicht gelöst werden: Manchmal updatet die UI erst nach mehrmaligem Durchwechseln der Tabs, die Änderungen der Aufgaben.

2 Domain Driven Design

2.1 Ubiquitous Language

Die Problemdomäne der entwickelten Software ist überschaubar. Daher wurde nur eine Hauptdomäne mit den folgenden Begriffe definiert.

Begriff	**Erklärung**
Task	Als Task wird eine vom Nutzer definierte Aufgabe verstanden. Diese hat einen Namen und verschiedene Parameter und kann aus bis zu 100 weiteren Unteraufgaben bestehen
SubTask	Als SubTask wird eine Unteraufgabe bezeichnet. Jede Unteraufgabe ist einer konkreten Aufgabe zugeordnet
Frequency	Die Frequency (Frequenz) beschreibt die Häufigkeit in der eine Aufgabe automatisch angelegt werden soll. Dazu gibt es mehrer Auswahlmöglichkeiten bei der Erstellung (Bsp.: täglich, wöchentlich oder in einer selbst definierten Anzahl an Tagen)
Statistic	Eine Statistik fasst alle abgeschlossenen Aufgaben mit identischem Namen zusammen und spiegelt deren Gesamtzeit wieder
MainActivity	Die MainActivity ist die Haupseite der App bestehend aus den drei Tabs: Aufgaben, Abgeschlossen, Statistik
CreateTaskActivity	Als CreateTaskActivity wird die Seite zur Erstellung bzw. Definition von Aufgaben bezeichnet

Tabelle 2.1: Begriffe und Erklärung der Ubiquitous Language

2.2 Taktische Muster

2.2.1 Value Objects

Value Objects sind einfache Objekte ohne eigene Identität. Das bedeutet sie kapseln ein Wertekonzept und werden nur durch ihre Eigenschaften und Werte beschrieben. Zwei *Value Objects* sind gleich, wenn sie dieselben Werte besitzen. In Java werden dazu die Methoden *equals()* und *hashcode()* überschrieben, wobei alle Felder einer Klasse miteinfließen. Gleichzeitig sind alle Felder einer *Value Object*-Klasse mit final gekennzeichnet. *Value Objects* sind also unveränderlich, wodurch keine Seiteneffekte entstehen können. Dadurch sind sie selbst-validieren und leicht testbar und machen das Einhalten von Infarianten im Code sehr einfach.

Statistic

Eine Statistik wird für mehrere Aufgaben generiert, die denselben Namen haben. Die benötigte Zeit für die Aufgaben wird dabei aufsummiert und in einem Verhältnis zu der Gesamtzeit für alle Aufgaben dargestellt. Das VO enthält dazu drei Felder *taskName, timeSpentMinutes, timePercantage*, die nicht verändert werden müssen. Verändern sich die erledigten Aufgaben, so werden alle Statistiken neu generiert. Das ist ein typisches Anwendungsgebiet für eine VO, da für jeden Aufgaben-Namen ein einziges Statistic-Objekt generiert wird und der Zustand nicht mehr verändert werden muss und darf. Da gültige Statistiken einmalig für den aktuellen Zustand der Aufgaben generiert werden und damit nicht verändert oder gespeichert und nur als eine Liste dargestellt werden, wird keine Entität geschweige denn Aggregat benötigt.

```
public class Statistic {
    private final String taskName;
    private final int timeSpentMinutes;
    private final double timePercentage;

    public Statistic(String taskName, int timeSpentMinutes, double
       timePercentage) {
        this.taskName = taskName;
        this.timeSpentMinutes = timeSpentMinutes;
        this.timePercentage = timePercentage;
    }

    public String getTaskName() {
        return taskName;
    }

    public int getTimeSpentMinutes() {
        return timeSpentMinutes;
    }

    public double getTimePercentage() {
        return timePercentage;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Statistic)) return false;
        Statistic statistic = (Statistic) o;
        return Double.compare(statistic.timeSpentMinutes,
           timeSpentMinutes) == 0 &&
                Double.compare(statistic.timePercentage, timePercentage)
                    == 0 &&
                Objects.equals(taskName, statistic.taskName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(taskName, timeSpentMinutes, timePercentage);
    }
}
```

Programmcode 2.1: VO Statistic als Beispiel für VOs im Projekt

TaskBase

Die Klasse *TaskBase* enthält aus 3 Werten die die Basisparameter einer Aufgabe definieren. Diese Werte werden bei der ursprünglichen Definition einmalig gesetzt und nicht mehr verändert. Die Werte repräsentieren den Namen der Aufgabe, das Zieldatum sowie die Frequenz in der die Aufgabe wiederholt werden soll. Somit wurde diese Klasse als VO implementiert und hat denselben Aufbau wie die Klasse *Statistic*

TaskFinish

TaskFinish ist eine Sammlung von Werten die sich direkt auf den Abschluss einer Aufgabe beziehen. Das Objekt enthält dabei Informationen, ob die Aufgabe abgeschlossen ist, zu welchem Zeitpunkt die Aufgabe abgeschlossen wurde und wie viel Zeit für den Abschluss benötigt wurde. Wird eine Aufgabe abgeschlossen ändern sich dabei alle drei Werte wodurch einfach ein neues Vo dieser Klasse erzeugt werden kann. Es ist hier also sinnvoll ein VO einzusetzen, um die oben genannten Vorteile zu nutzen.

SubTask

Jede Aufgabe kann weitere Unteraufgaben, sogenannte *SubTasks*, enthalten. Diese bestehen aus einem Namen und einem Wahrheitswert, der ausdrückt, ob die Aufgabe abgeschlossen ist. SubTasks wurden als einfache VO umgesetzte. Eine Identität wird nicht benötigt, da jede Unteraufgabe mit ihrem Namen nur einmal innerhalb einer Aufgabe vorkommen darf. Wird eine Unteraufgabe abgeschlossen, kann einfach ein neues VO erzeugt werden, welches den Wert Abgeschlossen erhält.

2.2.2 Entities

Entitäten haben eine eindeutige Identität in der Domäne. Sie enthalten veränderliche Eigenschaften und haben damit einen eigenen Lebeszyklus. Entitäten sind verschieden, wenn ihre Identitäten verschiedenen sind. Ihre Eigenschaften sind dabei unerheblich. Entitäten können aus weiteren Entitäten und VOs bestehen.

TaskObject

Die Entität *TaskObject* vereint die VOs TaskBase und TaskFinish und Repräsentiert damit alle Informationen einer Aufgabe. Ausgenommen sind dabei die Unteraufgaben. Es soll möglich sein, mehrere identische Aufgaben in der Domäne zu haben, da eine Aufgabe beispielsweise täglich wiederholt wird. Dazu muss zur Identifizierung eine Identität geschaffen werden. Die Klasse *TaskObject* besitzt dazu ein UUID. Über diese ID wird außerdem das Abspeichern der zusammengehörenden Werte in einer Datenbank vereinfacht. Darüber hinaus enthält die Entität die Logik zum Abschließen der Aufgabe. Dabei wird ein neues passendes VO *TaskFinish* erzeugt.

2.2.3 Aggregates

Aggregate sind Zusammenfassungen von Entitäten und VOs. Mit Aggreagten werden Objektbeziehungen entkoppelt und natürliche Transaktionsgrenzen sichergestellt. Jedes Aggregat bildet eine eigene Einheit und wird immer vollständig geladen oder gespeichert. Es werden Entity- und VO- übergreifende Domänenregeln gesichert. Es darf keine langfristige Referenzen auf innere Elemente geben. Wenn Referenzen auf innere Elemente rausgegeben werden, sind diese unveränderbar oder defensive Kopien.

Task

Das Aggregat *Task* repräsentiert eine vollständige Aufgabe. Hierbei werden zu einem *TaskObject* die zugehörigen SubTasks gespeichert. Jede Aufgabe kann aus bis zu 100 Unteraufgaben bestehen. Das Aggregat bildet diese Beziehung zwischen SubTask und TaskObject ab und sichert dabei die Regel, dass maximal 100 Unteraufgaben pro Aufgabe existieren. Das Aggregat dient damit als Türsteher auch indem es den Zugang zu Subtasks, durch die Ausgabe einer nichtveränderbaren Liste beschränkt. Identifiziert wird das Aggregat *Task* über das Root-Entity *TaskObject*. Eine Aufgabe wird immer vollständig

geladen und gespeichert, da zur Darstellung auf der Oberfläche alle Informationen über die Aufgabe benötigt werden.

2.2.4 Repository

Repositories entkoppelt die Persitenz und machen die Aggregat Roots auffindbar. Pro Aggregat in der Domäne existiert tyischerweise ein Repository. Die Definition ist Teil des Domain Code, die Implementierung findet auf höheren Ebene statt.

TaskRepository

Das Einzige Aggregat der Anwendung ist das einer Aufgabe (*Task*), wofür ein Repository namens *TaskRepository* angelegt wurde. Damit Domain-Services der Domain-Schicht auf das Repository zugreifen können, wurde dieses über ein Interface in der Domain-Schicht definiert. Die Implementierung befindet sich auf der Application-Schicht. Das Repository bezieht seine Daten über den *TaskDAO* (in Kapitel 5 beschrieben) aus der Datenbank und ist Vermittler zwischen Domäne und Datenmodell. Es stellt Aufgaben für Services und die Oberfläche bereit und ermöglicht das Auffinden der Aufgaben über die ID des Aggregat Roots. Darüber hinaus können über das Repository auch neue Aufgaben dem Datenmodell hinzugefügt und veränderte Aufgaben im Datenmodell aktualisiert werden.

```
public interface ITaskRepository {
    public List<Task> getAll();

    public List<Task> getAllOpenTasks();

    public List<Task> getAllFinishedTasks();

    public Optional<Task> find(UUID taskId);

    public void updateTask(Task task);

    public void add(Task task);

    public void deleteTask(UUID taskId);

}
```

Programmcode 2.2: Interface für TaskRepository

2.2.5 Domain Services

In einem Domain Service wird komplexes Verhalten implementiert, das nicht eindeutig einer Entität oder einem VO zugeordnet werden kann. Die Methodensignaturen verwenden die Begriffe des Domänenmodells. Ein- und Ausgabeparameter sind also Entitäten und VOs. Der Service Selbst ist zustandslos, darf aber globale Seiteneffekte haben.

StatisticService

Ein Verhalten das keiner VO oder Entität zugeordnet werden konnte, ist die Generierung der Statistiken. Für die Generierung müssen alle abgeschlossen Aufgaben über das Repository abgeholt werden. Aufgaben mit gleicher Benennung werden dann verschmolzen, wobei die benötigte Zeit aufaddiert wird. Insgesamt ist die Ausgabe eine Liste an *Statistic* VOs welche an der Oberfläche dargestellt werden.

2.2.6 Factory

Factories werden dazu genutzt um Objekte mit komplexen Konstruktionsregeln zu erzeugen und Konstruktoren zu entlasten.

TaskFactory

Als ein relativ komplexes Objekt gilt das Aggregat *Task*. Je nach dem wie viele Informationen zur Verfügung stehen, wird das Objekt anders aufgebaut. Ist eine Aufgabe beispielsweise noch nicht abgeschlossen gibt es kein *TaskFinish* VO. Wird eine neue Aufgabe über die Oberfläche erstellt, hat das Objekt außerdem noch keine UUID zugewiesen. Weiterhin müssen die Subtasks aus Strings generiert und dem Aggregat übergeben werden. Die über eine Aufgabe bekannten Werte werden deshalb einfach der Factory übergeben, welche das Objekt in jedem Fall korrekt über die verfügbaren Konstruktoren aufbaut. Die Factory wurde implementiert, da an verschiedenen Stellen (beim Laden von Aufgaben aus der Datenbank, beim Erzeugen neuer Aufgaben über die UI das Objekt korrekt aufgebaut werden musste. Mit der Factory wurde die Logik dafür in einer Klasse gesammelt. Größere Konstruktoren der beteiligten Klassen konnten damit vermieden werden.

3 Clean Architecture

Clean Architecture definiert ein Konzept zum Entwickeln einer langfristigen Architektur für eine Software. Das Prinzip beruht darauf den Sourcecode in unterschiedliche Schichten einzuteilen und dabei eine strenge Abhängikeitsregel einzuhalten: Abhängigkeiten bestehen immer nur von äußeren Schichten nach innen und nie umgekehrt. Somit bleiben abstrakter Code und Basisfuktionalitäten der Software frei von Abhängigkeiten zu kurzlebigen Plugins. Das bedeutet, dass beispielsweise in Klassen der inneren Schichten nicht erkennbar ist, welche Plugins zum Persistieren oder zur Generierung der Oberfläche verwendet werden. Dazu muss jede Klasse klar im Schichtenmodell positioniert sein. Damit jedoch eine Verbindung zwischen Klassen einer inneren Schicht zu einer äußeren hergestellt werden kann, definieren innere Schichten Interfaces, welche von Klassen aus äußeren Schichten Implementiert werden. Diese Prinzip wird als Dependency Inversion bezeichnet.

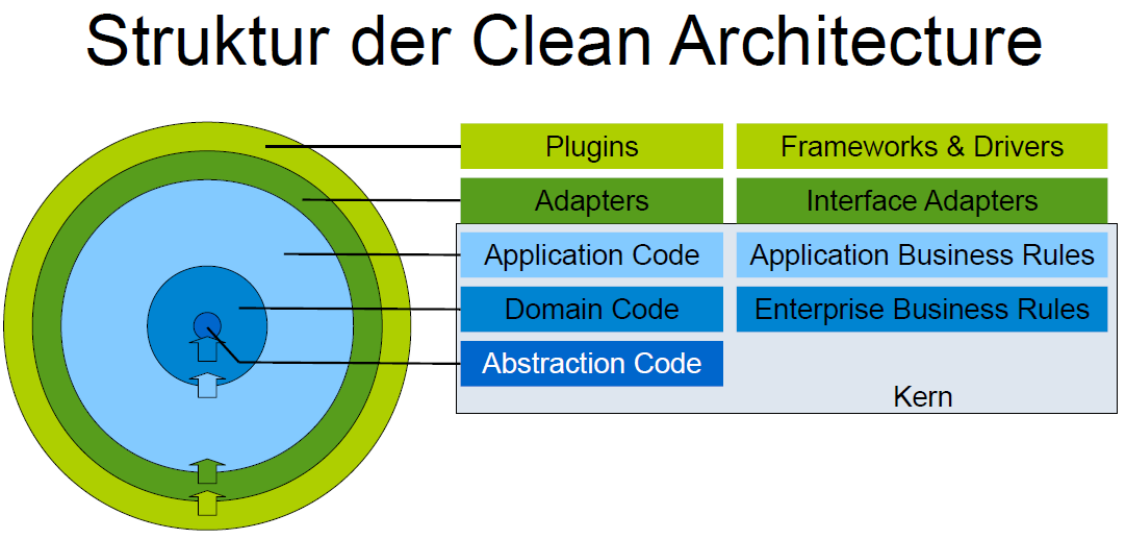

Abbildung 3.1: Schichtenmodell Clean Architecture

3.1 Planung

Im Clean Architecture Modell werden, wie in Abbildung 3.1 zu sehen, prinzipiell fünf Schichten unterschieden. Die Anzahl an Schichten darf jedoch variieren, wenn einer Schicht keine Klassen zugeordnet werden können, solange die genannten Regeln eingehalten werden. Für die entwickelte Anwendung werden die Schichten des Modells übernommen, mit Ausnahme der Schicht *Abstraction Code*. Die Umsetzung der Schichten gelingt über das verwenden von *Packages*, die entsprechend der Schichten benannt werden.

3.2 Begründung

3.2.1 Plugins-Schicht

Die Anwendung verwendet zwei Plugins bzw. Frameworks, die klar zu allen anderen Klassen der Software abgetrennt sein sollen. Das Android Framework wird in sehr kurzen Intervallen aktualisiert, was während der Durchführung des Projekts schon zu Problemen geführt hat. Es ist deshalb sehr wichtig, dass die Basis der Software, bzw. so viele Klassen wie möglich, nicht durch Updates von Android beeinflusst werden und angepasst werden müssens. Selbiges gilt für das Framework *SQLite*, welches zum Persistieren der Aufgaben verwendet wird. Alle Klassen die Abhängigkeiten zu Android und SQLite besitzen müssen, sind deshalb der Schicht *Plugins* zugeordnet.

Die Plugins-Schicht enthält keine Anwendungslogik und greift hauptsächlich auf die Adapter zu. Ein gutes Beispiel des Projekts stellt die Verbindung zwischen *DatabaseHelper* zu *TaskDAO* dar.

Der Plugins Schicht zugehörig sind dabei die *Activities*, welche die Hauptseiten der App darstellen, gefolgt von weiteren UI-Klassen zum Anzeigen der Aufgaben-Liste, Statistiken usw. Die Klasse *DatabaseHelper* stellt die einzige Verbindung zum Framework *SQLite* dar.

3.2.2 Adapters-Schicht

Die Adapters-Schicht vermittelt Aufrufe und Daten an die inneren Schichten. Inhalt ist dabei die Formatkonvertierung mit Mapping-Code. Externe Formate werden dabei für die Applikation angepasst und interne Formate für externe Plugins.

Als sauberes Beispiel kann die Klasse *TaskDAO* angesehen werden. Hier findet das

Mapping zwischen den Domain-Objekten auf einfache Datentypen für die Datenbank statt. In der Implementierten Datenbank (*DatabaseHelper*) werden alle Werte der Aufgaben als Zeichenketten (Strings) abgespeichert. Wird eine Aufgabe aus der Datenbank geladen muss beispielsweise der String für die ein Datum wieder in ein Java Date-Objekt überführt werden. Zum Speichern einer Aufgabe muss dieser Prozess umgekehrt werden. Die UUID einer Aufgabe wird in einen String konvertiert etc. Um die Abhängigkeitsregel einzuhalten, wurde der *DatabaseHelper* über das Interface *IDatabaseHelper* definiert. Zum Entlasten der Konstruktoren und zur vereinfachten Rückgabe der Aufgaben-Parameter aus der Datenbank wird eine Java-Map mit dem Schlüssel *TaskObjectValues* verwendet, durch welche die Zeichenketten klar den Attributen der Aufgabe zugeordnet werden können. Als Verbindung zum Domain Code wird das *TaskRepository* verwendet, welches die Methoden der Klasse *TaskDAO* über das Interface *ITaskDAO* kennt.

3.2.3 Application-Schicht

Die Application-Schicht enthält Anwendungsfälle und implementiert anwendungsspezifische Geschäftslogik. Der Fluss der Daten und Aktionen von und zu den Element des Domain Codes wird gesteuert. Änderungen der Application-Schicht beeinflussen die Domain-Schicht nicht. Gleichzeitig haben die Klassen der Application-Schicht keine Informationen darüber, wie das Ergebnis präsentiert wird (Bsp. mit Android UI) oder von wo die Aufrufe stattfinden. Ändern sich die Anforderungen der Software, hat dies meist Auswirkungen auf diese Schicht.

Die Use-Cases der Anwendung sind:

- Erstellen und Speichern von Aufgaben
- Updaten von Aufgaben (beenden)
- Generieren von Statisitiken

Angesiedelt wurden auf dieser Schicht die Klassen *TaskRepository* (definiert über ein Interface auf Domain-Ebene), der *TaskFactory* sowie dem *StatisticService*. Das TaskRepository kommuniziert durch das *ITaskDAO* Interface schließlich mit der Datenbank zum Abrufen, Speichern oder Ändern von Aufgaben. Der Statistik Service generiert die Statistiken für abgeschlossene Aufgaben. Mit der TaskFactory können aus den einzelnen verfügbaren Parametern für Aufgaben, komplexere Task-Objekte erstellt werden.

3.2.4 Domain-Schicht

Die Domain Schicht bildet den inneren Kern der Anwendung und bildet organisationsweit gültige Geschäftslogik ab. Bei der Anwendung von Domain Driven Design liegen die taktischen Muster in dieser Schicht. Aus den zuvor genannten Gründen wurden einige Klassen der Application-Schicht zugeordnet. In der Domain Schicht befinden sich aktuell alle Aggregate, Entities und ValueObjects sowie weitere Klassen welche für den Aufbau der taktischen Muster verwendet werden (Beispiel das Enum *Frequency*).

3.2.5 Abstraction-Schicht

Diese Schicht enthält domänenübergreifendes Wissen und ist durch sehr abstrakten Code geprägt. Für diese Anwendung wurden alle notwendigen Funktionen durch die Programmiersprache bereitgestellt. Keine Klasse konnte dieser Schicht zugeordnet werden, weshalb sie nicht umgesetzt wurde.

4 Programming Prinziples

4.1 SOLID

Single Responsibility Principle

Jede Klasse einer Software soll eine möglichst geringe Komplexität und Kopplung haben. Dies gelingt indem darauf geachtet wird das eine Klasse nur eine Zuständigkeit hat. Komplexes Verhalten wird durch die Kombination von mehreren Klassen abgebildet.

Durch die Anwendung verschiedener Architekturprinzipien, wie DDD und Clean Architecture, ist dieses Prinzip relativ gut im Projekt umgesetzt, da hier darauf geachtet wird die Logik auf möglichst kleine Klassen aufzuteilen. Außerdem gibt es für viele Aufgaben vordefinierte Konstrukte. Ein Beispiel ist das *TaskRepository*. Dieses verwaltet die definierten Aufgaben. Das Abspeichern und Laden von den Aufgaben gelingt wiederum über die Klasse *TaskDAO*. Diese ist jedoch nur für das Mapping der Objekte zwischen Repository und Datenbank zuständig. Das eigentlich Laden und Speichern wird in der Klasse *DatabaseHelper* ausgeführt. In dieser Kette hat jede Klasse damit genau eine Aufgabe.

Open Closed Principle

Die Elemente einer Software sollen möglichst gut erweiterbar und geschlossen für Änderungen sein. Diese Prinzip wurde im Projekt vor allem durch den Einsatz von Schnittstellen (Interfaces) gelöst. Ein gutes Beispiel stellt das Interface *IDatabaseHelper* dar. Diese Schnittstelle definiert klar, welche Methodensignaturen eine Implementierung der Datenbank bereitstellen muss. Bei den Parametern handelt es sich um einfache Zeichenketten, die also von jeder gewöhnlichen Datenbank verwaltet werden können. Die Klasse *TaskDAO* welche die Methoden des *DatabaseHelper* aufruft kennt dabei nur das Interface. Somit ist es möglich einen weiteren *DatabaseHelper* zu implementieren, der beispielsweise eine anderes Framework als SQLite benutzt, ohne dass weitere Klassen angepasst werden müssen

Liskov Substitution Principle

Das Prinzip setzt voraus, dass Objekte eines abgeleiteten Typs, als Ersatz für die Instanz des Basistyps funktionieren. Dabei darf die Korrektheit des Programms nicht verändert werden.

Das zuvor in *Open Closed Principle* erwähnte Beispiel, ist auch auf dieses Prinzip anwendbar. Ein Interface ist in Java praktisch eine Abstracte Klasse, in der jedoch nur Methodensignaturen definiert werden. Alle Klassen die dieses Interface implementieren halten sich an einen Vertrag, aus den Input-Parametern die vorgegeben Ausgaben zu erzeugen. Wie im Beispiel erwähnt kann damit das zugrundeliegende Datenbanksystem durch ein Anderes ersetzt werden. Die Korrektheit des Programms ist durch den „Vetrag “ der Schnittstelle sichergestellt.

Interface Segregation Principle

Nach diesem Prinzip sollen Interfaces möglichst klein gehalten werden. Ähnlich zu *Single Responsibility* sollte auch bei Interfaces darauf geachtet werden, dass die definierten Methoden nur einer Hauptaufgabe dienen. Die in der Software definierten Interfaces enthalten zwischen 5 und 8 Methoden und dienen einer klaren Aufgabe. Es gibt kein Interface das sinnvoll auf mehrere aufgeteilt werden kann.

Dependcy Inversion Principle

Das Prinzip beschreibt die Richtung der Abhängigkeiten von High-Level-Elementen zu Low-Level-Elementen. Durch die angewendete *Clean Architecture* wurde das Projekt in verschiedene Schichten unterteilt, bei denen nur äußere Schichten Abhängigkeiten zu inneren Schichten haben. Das entspricht dem Dependency Inversion Principle. Innere Schichten sollen möglichst stabil sein und deshalb keine Abhängigkeiten an äußere Schichten haben, die beispielsweise schnelllebige Frameworks verwenden. Für die Umsetzung diese Prinzips werden Interfaces verwendet.

Auch hier greift das Beispiel aus *Open Closed Principle*. Die Klasse *TaskDAO* erwartet ist auf der Adapter-Schicht und muss Anfragen an eine Datenbank weiterleiten, die auf einer äußeren Schicht (Plugins) implementiert ist. Deshalb wird auf der Adapter-Schicht das Interface *IDatabaseHelper* definiert. Somit kann die Implementierung auf der Plugins-Schicht stattfinden und die Klasse *TaskDAO* erwarten ein Objekt das die Funktionen des

Interfaces implementiert. Somit ist die äußere Plugins-Schicht von der inneren Adapter-Schicht abhängig, aber nicht umgekehrt.

4.2 GRASP

Low Coupling

Geringe Kopplung bedeutet, dass Klassen wenig Abhängigkeiten zueinander haben. Das bedeutet, werden in einer Klasse Änderungen vorgenommen, müssen nur wenige andere Klassen angepasst werden. Die Austauschbarkeit von Komponenten ist gegeben. Wie zuvor für das Datenbanksystem erklärt, wurde im Projekt sehr darauf geachtet nur wenige nötige Abhängigkeiten zwischen Klassen zu erzeugen, wodurch viele Komponenten leicht ausgetauscht werden könne.

High Cohesion

Eine hohe Kohäsion bezieht sich auf das Innere von Klassen. Alle enthaltenen Element sollen dabei semantisch nahe zueinander sein. Für den Entwickler wird der Code dadurch verständlicher und Komponenten sind besser wiederverwendbar.

Ein Beispiel ist die Aufteilung einer Aufgabe in verschiedene VOs. Die Entität *TaskObject* repräsentiert die Daten zu einer Aufgabe. Diese Daten sind aufgeteilt in Informationen die direkt mit dem Abschluss einer Aufgabe zusammenhängen, wie Abschlussdatum und benötigte zeit in dem VO *TaskFinish*. Daten die zu Anfang festgelegt werden und damit die Basis einer Aufgabe darstellen, wie den Namen und die Frequenz in welcher die Aufgabe wiederholt wird, werden in einem VO *TaskBase* gehalten.

4.3 DRY

DRY ist eine Abkürzung für don't repeat yourself und bedeutet, das möglichst keine Logik doppelt implementiert werden soll. Nicht nur weil somit während der Entwicklung zeit gespart werden kann, sondern vor allem, weil in späteren Abänderungen der Logik, viele verschiedene Code-Stellen geändert werden müssen. Dabei kommt es häufig zu Fehlern. Innerhalb des Projekts wurde auf diese Prinzip geachtet. Ein Beispiel zeigt die Klasse *ListViewSizeUtil*. Diese enthält die Logik die Größe einer Liste auf der Oberfläche an die

Anzahl der enthaltenen Elemente anzupassen. Verwendet wird diese Logik in mehreren Klassen, wie TaskListAdapter und SubTaskListAdapter. Deshalb wurde die Logik in statischen Methoden in einer eigenen Klasse implementiert. Damit kann die Funktionalität leicht von verschiedenen UI-Klassen genutzt werden.

5 Entwurfsmuster

Entwurfsmuster sind bewährte Vorlagen zum Lösen wiederkehrender Probleme in der Softwarearchitektur. In Dieser Arbeit wurde mit den Entwurfsmustern *DAO* und *Factory* gearbeitet.

5.1 Data Access Object (DAO)

Data Access Objects oft als DAO abgekürzt, werden zum Kapseln von Zugriffen auf verschiedener Arten von Datenquellen verwendet. Mit der korrekte Anwendung des Entwurfsmusters, ist es möglich die Datenquelle auszutauschen ohne den aufrufenden Code zu verändern. Im Projekt wurde dieses Entwurfsmuster mit der Klasse *TaskDAO* und dem Interface *ITaskDAO* umgesetzte. Damit wurde, wie in Kapitel 3.2.2 erklärt, die gesamte Mapping-Logik von einfachen Zeichenketten auf die Domain-Objekte mit komplexeren Datentypen wie Date, UUID, ... in die Adapter-Ebene verschoben. Die Klassen *DatabaseHelper* und *TaskRepository* enthalten dadurch keine Mapping-Logik und es liegt eine saubere Trennung vor.

Als Datenquelle kann jedoch nicht nur die Datenbank angesehen werden, sondern auch die *CreateTaskActivity*, welche die Nutzereingaben beim erstellen neuer Aufgaben an das Repository weiterleitet. Damit überall die gleiche Mapping-Logik verwendet wird, wurde im *TaskDAO* eine Methode *generateTaskFromStringValues(...)* extrahiert, welche zum Überführen der Nutzereingaben in ein Task-Objekt genutzt werden kann. Dieser Schritt wurde in Kapitel 7.1 beschrieben.

Die Klasse *TaskDAO* wurde nicht im Nachhinein eingebaut sondern bei Implementierung von Datenbank und Repository. Ein UML für den Zustand davor kann deshalb nicht gezeigt werden. Der Aktuelle Zustand ist der Abbildung 5.1 zu entnehmen. *CreateTaskActivity* verwendet die Klassen *TaskDao* und *TaskRepository*. Die *use*-Beziehung wurde von dem verwendeten Plugin leider nicht dargestellt.

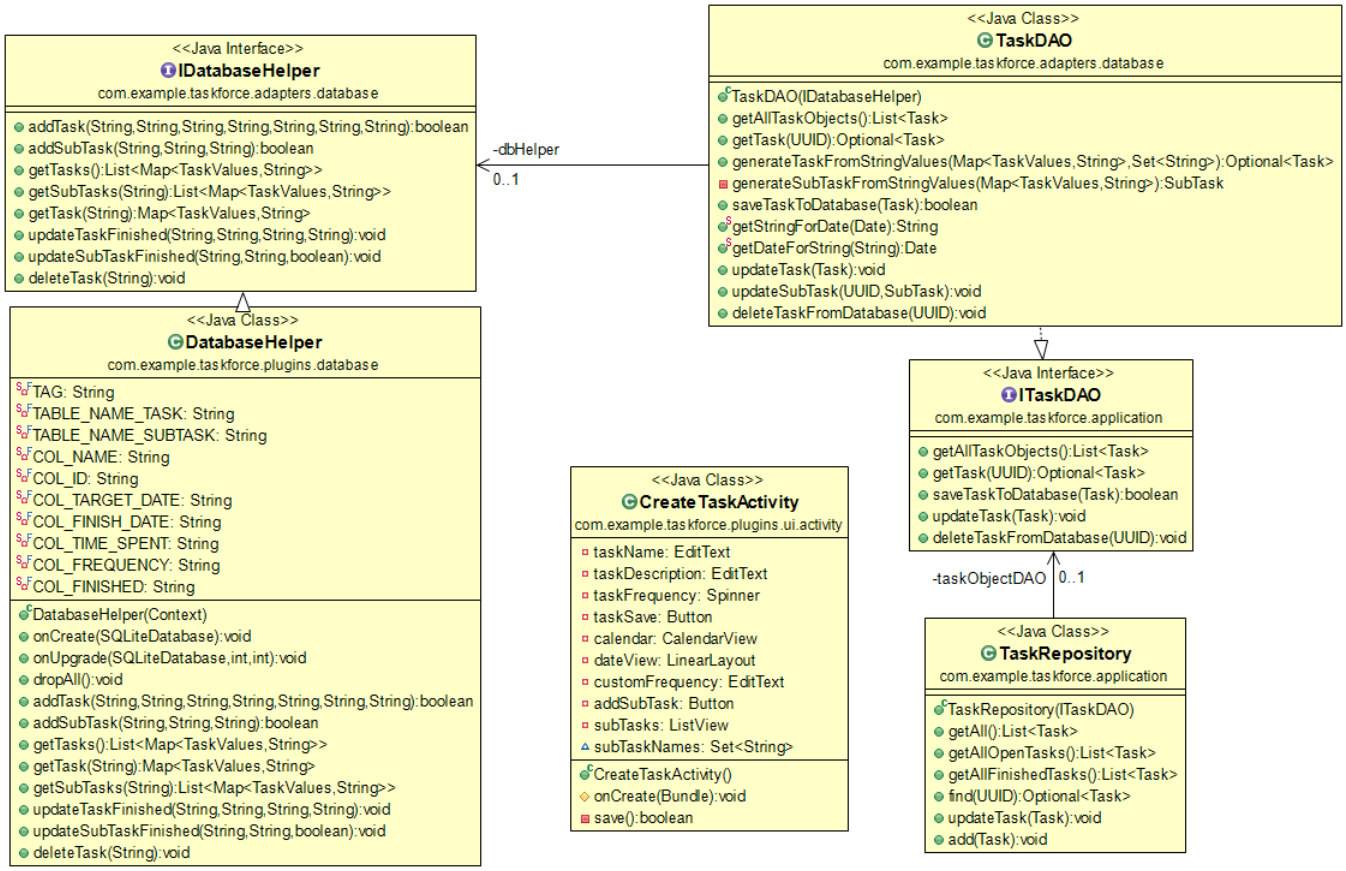

Abbildung 5.1: UML für Entwurfsmuster Data Access Object

5.2 Factory

Auch die *TaskFactory* (in Kapitel 2.2.6 bereits erklärt) stellt ein Entwurfsmuster dar. Die Logik für die Erzeugung eines *Task*s auf verschiedenen Datengrundlagen wurde damit zentralisiert. Somit können bei der Weiterentwicklung weniger Fehler beim erstellen von Aufgaben auftreten.

6 Tests

Mit Unit-Tests werden die kleinst-möglichen Komponenten in einer Software getestet. Das sind oft einzelne Methoden oder Funktionalitäten. Mit Unit-Tests sollten zum einen grundlegende Funktionalitäten getestet werden, aber auch bekannte Randfälle, mit der die Software umgehen muss. Damit sichern solche Tests nicht nur die Funktionalität und Zuverlässigkeit, sondern bilden auch einen wichtigen Teil der Dokumentation. Anhand der Erwartungen (Assertions) in einem Tests sind Entwickler in der Lage zu Erkennen, wie eine Komponente in einer bestimmten Situation reagieren soll. Der Vorteil solcher meist lokalen Tests liegt in der einfachen Entwicklung und Automatisierbarkeit.

Die für das Projekt bestehenden Anforderungen wurden in der Testklasse *StatisticServiceTest*. Durch das mocken des *DatabaseHelpers*, der einen *Context* zur Laufzeit benötigt und somit nicht einfach in einem Unit-Test erzeugt werden kann, konnten durch einfache Tests des *StatisticServices*, abgesehen von der UI, die Funktionalität für viele Klassen abgesichert werden.

7 Refactoring

Wird bereits funktionierender Code verbessert, indem die Lesbarkeit erhöht wird oder beispielsweise auch die Fehleranfälligkeit der Software verringert wird, so spricht man von Refactoring. Während der Projektarbeit wurden viele Refactorings durchgeführt. Auf zwei davon wird im Folgenden eingegangen.

7.1 Mehtod-Extract um Code in anderen Klassen verfügbar zu machen

In der Klasse *TaskDAO* befindet sich Logik erstellen von *Task*-Objekten aus String-Parameter, wie sie in der Datenbank abgelegt werden. Die gleiche Logik wird auch benötigt, wenn der Nutzer eine neue Aufgabe in der Klasse *CreateTaskActivity* definiert. Die Texteingaben müssen in valide Parameter umgewandelt werden. Das Mapping und Aufbauen des *Task*-Objekts mithilfe der *TaskFactory* war hier nochmals implementiert.

7.1.1 Problem

Wird beispielsweise das Format in dem ein Datum gespeichert wird im TaskDAO geändert, müsste es auch in de Klasse *CreateTaskActivity* angepasst werden. Wird dieser Schritt vergessen können Daten eventuell nicht mehr oder nur fehlerhaft umgewandelt werden.

7.1.2 Lösung

Die Generierung der Aufgaben aus den String-Parametern in *TaskDAO* wurde allgemein gehalten und in eine eigene Methode (*generateTaskFromStringValues(taskValues, subTaskNames))* extrahiert. In der *CreateTaskActivity* wird diese Methode verwendet wodurch die Mapping-Logik nur noch einmal existiert und immer identisch ist. Die Parameter wurden

außerdem in einer Map klar zugeordnet, damit der Konstruktor übersichtlich bleibt und dadurch weniger Fehleranfällig ist(siehe Kapitel 7.2).

7.1.3 Vorher

```
private boolean save(){
        TaskFactory fac = new TaskFactory();
        fac.setTaskName(taskName.getText().toString());
        try {
            Frequency freq = Frequency.fromKey((String) taskFrequency.
               getAdapter().getItem(taskFrequency.
               getSelectedItemPosition()));
            fac.setFrequency(freq);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }

        fac.setTargetDate(Date.from(Instant.ofEpochMilli(calendar.
           getDate())));
        fac.setSubTasksFromString(subTaskNames);
        Optional<Task> task = fac.build();
        if(task.isPresent()){
            new TaskRepository(new TaskDAO(new DatabaseHelper(
               getBaseContext()))).add(task.get());
            return true;
        }
        return false;
    }
```

Programmcode 7.1: save() in CreateTaskActivity

7.1.4 Nachher

```
    private boolean save(){
        Map<TaskValues, String> taskValues = new HashMap<>();
        taskValues.put(TaskValues.NAME, taskName.getText().toString());
        taskValues.put(TaskValues.TARGET_DATE, TaskDAO.getStringForDate(
                Date.from(Instant.ofEpochMilli(calendar.getDate()))));
        taskValues.put(TaskValues.FREQUENCY, (String) taskFrequency.
           getAdapter()
                .getItem(taskFrequency.getSelectedItemPosition()));

        TaskDAO dao = new TaskDAO(new DatabaseHelper(getBaseContext()));
        Optional<Task> task = dao.generateTaskFromStringValues(
           taskValues, subTaskNames);
        TaskRepository repository = new TaskRepository(dao);
        if(task.isPresent()){
            repository.add(task.get());
            return true;
        }
        return false;
    }
```

Programmcode 7.2: save() in CreateTaskActivity

7.2 Reduzieren von Konstruktor-Parametern

7.2.1 Problem

Konstruktoren, die viele Parameter des selben Typs entgegennehme, bieten viel Fehlerpotential zum Vertauschen von Parametern. Im *DatabasHelper* wurden zum Speichern von Aufgaben alle Parameter als String entgegen genommen. Den Konstruktor korrekt zu Verwenden erforderte viel Konzentration.

7.2.2 Lösung

Statt den Einzelnen Parametern wird eine Map übergeben in welcher die Zeichenketten über einen Schlüssel (Enum *TaskValues*) den jeweiligen Attributen (Spalten der Datenbank) zugeordnet werden können. Das Enum liegt in der Adapter-Schicht und ist somit für den *DatabaseHelper*, wie auch für den *TaskDAO* ohne Verletzung der Clean-Architecture Regeln zugänglich. Der Code und das Mapping ist somit viel verständlicher und lesbarer, worduch in der Zukunft weniger Fehler auftreten sollten.

7.2.3 Vorher

```
public boolean addTask(String taskObjectId, String taskName, String
    targetDate, String finishDate, String timeSpentMinutes, String
    frequency, String isFinished)

public boolean addSubTask(String taskObjectId, String subTaskName,
    String isSubTaskFinished)
```

Programmcode 7.3: Konstruktoren (DatabaseHelper) zum Speichern von Tasks und SubTasks in der Datenbank

```
public boolean saveTaskToDatabase(Task task){
    TaskBase taskBase = task.getTaskObjectCopy().getTaskBase();
    TaskFinish taskFinish = task.getTaskObjectCopy().getTaskFinish();

    boolean worked = dbHelper.addTask(
            task.getId().toString(),
            taskBase.getName(),
            getStringForDate(taskBase.getTargetDate()),
            getStringForDate(taskFinish.getFinishDate()),
            String.valueOf(taskFinish.getTimeSpentMinutes()),
            taskBase.getFrequency().getKey(),
            String.valueOf(task.isFinished()));

    for(SubTask sub: task.getSubTasks()){
        worked &= dbHelper.addSubTask(task.getId().toString(), sub.
           getTaskName(), String.valueOf(sub.isFinished()));
    }
    return worked;
}
```

Programmcode 7.4: saveTaskToDatabase() in TaskDAO

7.2.4 Nachher

```
public boolean addTask(Map<TaskValues, String> taskValues)

public boolean addSubTask(Map<TaskValues, String> taskValues)
```

Programmcode 7.5: Konstruktoren (DatabaseHelper) zum Speichern von Tasks und SubTasks in der Datenbank

```
public boolean saveTaskToDatabase(Task task){
    TaskBase taskBase = task.getTaskObjectCopy().getTaskBase();
    TaskFinish taskFinish = task.getTaskObjectCopy().getTaskFinish();

    Map<TaskValues, String> taskValues = new HashMap<>();
    taskValues.put(TaskValues.ID,  task.getId().toString());
    taskValues.put(TaskValues.NAME, taskBase.getName());
    taskValues.put(TaskValues.TARGET_DATE, getStringForDate(taskBase.
       getTargetDate()));
    taskValues.put(TaskValues.FINISH_DATE, getStringForDate(taskFinish.
       getFinishDate()));
    taskValues.put(TaskValues.TIME_SPENT, String.valueOf(taskFinish.
       getTimeSpentMinutes()));
    taskValues.put(TaskValues.FREQUENCY, taskBase.getFrequency().getKey
       ());
    taskValues.put(TaskValues.FINISHED, String.valueOf(task.isFinished()
       ));

    boolean worked = dbHelper.addTask(taskValues);

    for(SubTask sub: task.getSubTasks()){
        Map<TaskValues, String> subTaskValues = new HashMap<>();
        subTaskValues.put(TaskValues.ID,  task.getId().toString());
        subTaskValues.put(TaskValues.NAME, sub.getTaskName());
        subTaskValues.put(TaskValues.FINISHED, String.valueOf(sub.
           isFinished()));
        worked &= dbHelper.addSubTask(subTaskValues);
    }
    return worked;
}
```

Programmcode 7.6: saveTaskToDatabase() in TaskDAO

```
public enum TaskValues {
    NAME("name"),
    ID("id"),
    TARGET_DATE("target_date"),
    FINISH_DATE("finish_date"),
    TIME_SPENT("time_spent"),
    FREQUENCY("frequency"),
    FINISHED("finished");

    private final String key;

    private TaskValues(String key){
        this.key = key;
    }

    public String getKey(){
        return this.key;
    }

    public static TaskValues fromKey(String key) throws Exception {
        for(TaskValues value: TaskValues.values()){
            if(value.getKey().equals(key)){
                return value;
            }
        }
        throw new Exception("TaskObjectValue key not found");
    }
}
```

Programmcode 7.7: Enum: TaskValue als Schlüssel für die Zuordnung von String zu Attribut

Abbildungsverzeichnis

Quellcodeverzeichnis