Technische Dokumentation

Im Rahmen der Prüfung zum

Bachelor of Science (B. Sc.)

Des Studiengangs Angewandte Informatik

An der

Dualen Hochschule Baden-Württemberg, Karlsruhe

Autor: Bengt Joachimsohn

Kurs: TINF18-B5

Eidesstattliche Erklärung

Gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Ort, Datum Unterschrift

# Inhaltsverzeichnis

[Inhaltsverzeichnis 2](#_Toc73356495)

[Abbildungsverzeichnis 3](#_Toc73356496)

[1 Einleitung 5](#_Toc73356497)

[2 Unit Tests 5](#_Toc73356498)

[3 Programming Principles 8](#_Toc73356499)

[3.1 SOLID 8](#_Toc73356500)

[3.2 GRASP 12](#_Toc73356501)

[3.3 DRY 13](#_Toc73356502)

[4 Refactoring 14](#_Toc73356503)

[5 Legacycode 16](#_Toc73356504)

[6 Entwurfsmuster 17](#_Toc73356505)

[7 Domain-Driven-Design 17](#_Toc73356506)

# Abbildungsverzeichnis

[Abbildung 1: Tests für die Benutzerverwaltung 5](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356507)

[Abbildung 2: Implementierung der des Benutzerverwaltung Interfaces 6](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356508)

[Abbildung 3: Test für den Passwort-Generator und -Validierer 7](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356509)

[Abbildung 4: Test des Datenbankadapters 8](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356510)

[Abbildung 5: Beispiel für das Single Responsibility Principle anhand vom StorageHelper 9](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356511)

[Abbildung 6: BasicAdapter als Beispiel für eine Klasse die offen für Veränderung ist, aber geschlossen für Veränderungen. 9](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356512)

[Abbildung 7: Beispiel für der Erweiterung der Basisklasse BasicAdapter, die jedoch von ihren Funktionen nicht modifiziert wird. 10](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356513)

[Abbildung 8: Beispiel für das Interface Segregation Principle anhand vom PlaceEditor 11](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356514)

[Abbildung 9: Beispiel für die Verwendung eines Interfaces, das in mehrere kleinere aufgeteilt wurde, sodass Funktionen beschränkt werden können. 11](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356515)

[Abbildung 10: Das Interface Savable 11](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356516)

[Abbildung 11: Save-implementierung aus dem AccountSettingsFragment 12](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356517)

[Abbildung 12: Save-implementierung aus dem TaskEditorFragment 12](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356518)

[Abbildung 13: Basisimplementierung des SavableFragments 12](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356519)

[Abbildung 14: ProfilePrivateFragment das als Controller dient 13](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356520)

[Abbildung 15: Basisklasse die für die Abstrahierung der Sub-Level Fragmente verwendet wird. 14](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356521)

[Abbildung 16: Auswertung des Statistic-Plugins für meine Anwendung 15](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356522)

[Abbildung 17: Report zu Codesmells vom Sonarlint Plugin 15](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356523)

[Abbildung 18: Globale Fields in LocalAuthHelper 16](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356524)

[Abbildung 19: Fields in LocalAuthHelper nach der Modifizierung 16](file:///C:\Users\bebej\Desktop\Advanced%20Software%20Engineering.docx#_Toc73356525)

# Einleitung

In diesem Dokument wird sich mit dem Programmcode der Anwendung TaskMaster auseinandergesetzt. Das Projekt wurde im Rahmen der Studienarbeit entwickelt und größtenteils von mir Programmiert. Es wird erläutert welche Tests ausgeführt werden, welchen Zweck diese erfüllen, welche Designprinzipien angewendet wurden und an welchen Stellen der Programmcode refactored wurde. Die Anwendung wurde in der Programmiersprache Kotlin geschrieben, jedoch wurde auch an einer Stelle Java Code verwendet, da die Konvertierung von Java zu Kotlin nicht erfolgreich durchgeführt werden konnte. Das Repository der Anwendung ist unter: <https://github.com/TheYaINN/TaskMaster-App> zu finden.

# Unit Tests

Unit Tests sind Tests die kleinen Einheiten der Codebase auf Richtigkeit überprüfen. Richtigkeit bedeutet in diesem Fall, dass eine Methode mit einer Eingabe eine erwartete Ausgabe oder Funktion hat, die eine Änderung ausführt, die im Nachhinein durch weiteren Code wieder überprüfbar ist.

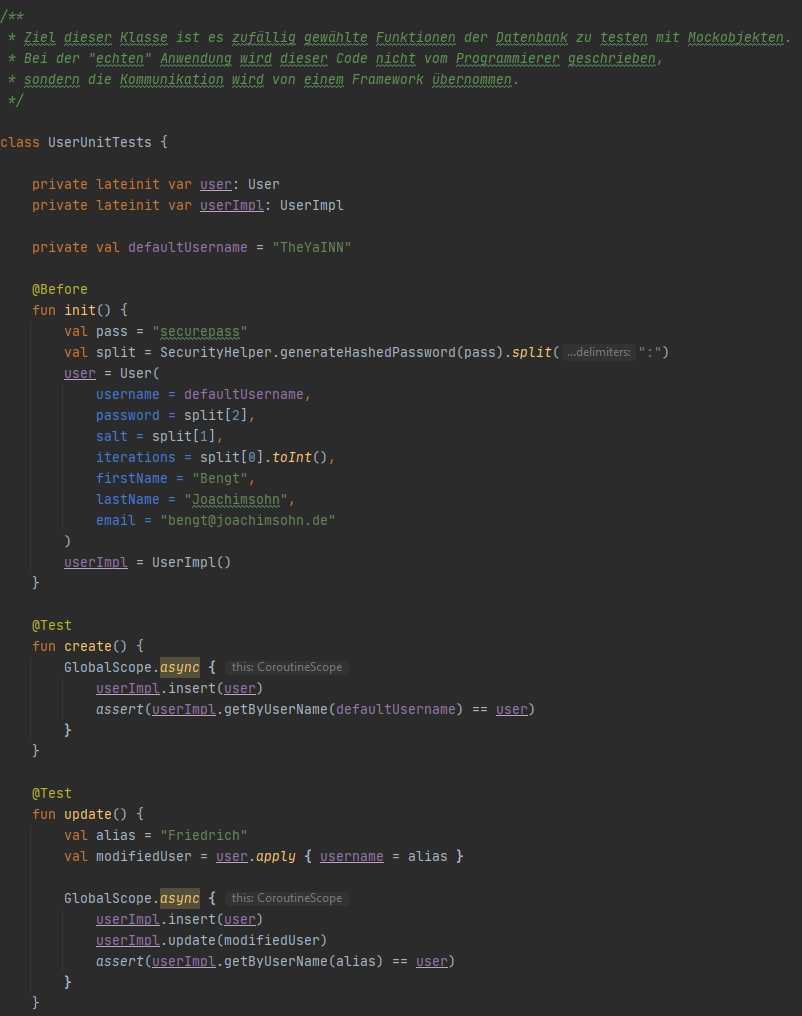


Abbildung 1: Tests für die Benutzerverwaltung

Wenn z.B. Komponenten der Anwendung getestet werde sollen, die auf Datenbankabfragen beruhen können sogenannte Mock-Objekte verwendet werden. Die Mock-Objekte stellen Daten zur Verfügung, die normalerweise über die Datenbank abgerufen werden würden.

Ich habe beispielhaft dargestellt wie solche Tests aussehen können, indem ich mein Interface, dass verwendet wird als Schnittstelle zur Datenbank, selbst implementiert habe. Das Interface beschreibt die Funktionen, die zu Verfügung gestellt werden in der „richtigen“ bzw. LIVE-Datenbank. Die Implementierung wird in der Anwendung generiert durch das Framework Android Room, jedoch habe ich es hier aus demonstrationszwecken selbst nochmal implementiert, um zu testen, dass die Methoden das Abbilden was benötigt wird.

Zum Testen wurden zufällig gewählte Operationen ausgewählt und diese dann getestet anhand der Mock-Implementierung, ob die Funktion dem entspricht, was sich vorgestellt wurde bei der Entwicklung.

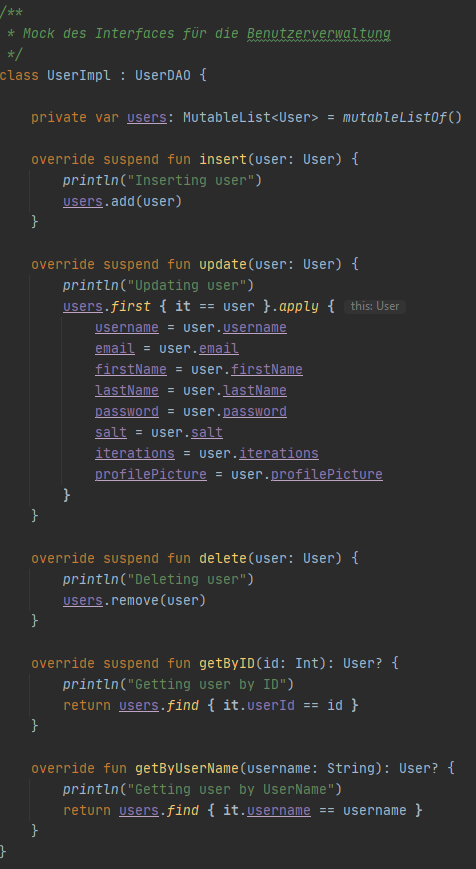


Abbildung 2: Mock-Implementierung des Benutzerverwaltung Interfaces

Die Tests bestehen zum ersten aus einer Initialisierung, die vor jeden Test ausgeführt wird, sodass die getesteten Daten immer gleich sind und somit auch immer die gleichen Bedingungen herrschen.

Die Initialisierung findet in der init-Methode statt, dort werden die Datenbank und ein Benutzer jedes Mal neu generiert, sodass diese immer vor jeden Test gleich sind.

Bei dem Test „Create“ wird getestet, dass das   
Einfügen in die Datenbank und dann das darauffolgende Auslesen, mit nur dem Benutzernamen erfolgreich stattfindet und es der gleiche Benutzer ist.

Bei dem Test „Update“ wird getestet das das Aktualisieren des Benutzernamens korrekt funktioniert. Bei beiden Tests müssen die Datenbankaufrufe in ein sog. „Globalscope“, da diese im Interface als suspend markiert sind, was ein Schlüsselwort in Kotlin für Asynchrone Abfragen ist. Die Kommentare in den Funktionen wurde eingeführt, sodass wie in der LIVE-Datenbank die Abfragen nachvollzogen werden können und man beim Testen immer den Überblick hat welche Funktion explizit getestet wurde.

Leider können keine komplexeren Konstrukte wie Benutzeroberflächen getestet werden, bei denen ein Mock-Objekt für z.B. die Datenbank verdeutlicht werden würde. Das Prinzip der Durchführung wäre jedoch gleich wie es in dem oben dargestellten Test ist. Der Vorteil solcher Mock-Objekte bei z.B. Test von Benutzeroberflächen ist, dass keine Wartezeiten beim Testen oder Entwickeln entstehen, da die Daten sofort erstellt werden und nicht in der Datenbank abgefragt werden. Somit können ebenfalls Testdaten verwendet werden, die man nicht in der Datenbank ablegen möchte, da diese keine Relevanz für die Anwendung haben.

Ebenfalls wurden Helferklassen, die keine Abhängigkeiten zu Android Funktionen wie Fragmente, Kontext, etc. haben, dementsprechend bleiben nicht viele Klassen übrig, getestet.

In der Abbildung 3 ist zu sehen wie das Erstellen und Überprüfen von Passwörtern getestet wird. Es wird zuerst ein Passwort generiert, mit allen benötigten Werten für die DB später, diese werden der Validierungsmethode übergeben und überprüft. Bei der Überprüfung wird ein Boolean zurückgegeben, sodass hier ganz einfach überprüft werden kann, dass wenn man die gleiche Eingabe hat, dass man sich anmelden kann.

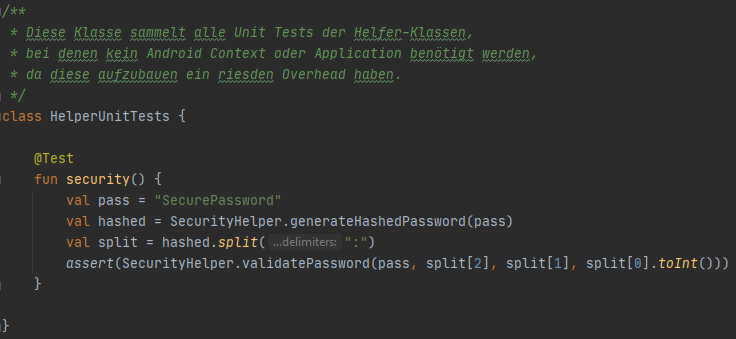


Abbildung 3: Test für den Passwort-Generator und -Validierer

Wichtige Elemente einer Anwendung zu testen und auf Richtigkeit zu überprüfen sind z.B. Adapter bzw. Konvertierer, dessen Aufgabe es ist Objekte von einem System in ein anderes zu Übersetzen. In diesem Fall wird überprüft, dass das Enum Status zu einem String konvertiert wird und so in der Datenbank abgespeichert werden kann. Bei der Konvertierung müssen immer beide Richtungen überprüft werden, da eine Bidirektionale Kommunikation in diesem Fall von hoher Wichtigkeit ist, da die Werte ebenfalls wieder ausgelesen werden müssen, damit diese später in der Benutzeroberfläche angezeigt werden können.



Abbildung : Test des Datenbankadapters

# Programming Principles

In diesem Kapitel werden Programmierprinzipien kurz erläutert und auf den vorhandenen Code angewendet.

## SOLID

* Single Responsibility Principle
  + Eine Klasse / Methode hat nur eine Aufgabe bzw. Aufgabenbereich.

Der StorageHelper verwendet genau dieses Prinzip, da er nur für dafür zuständig ist, dass die Anwendung Zugriff auf den Speicher hat.

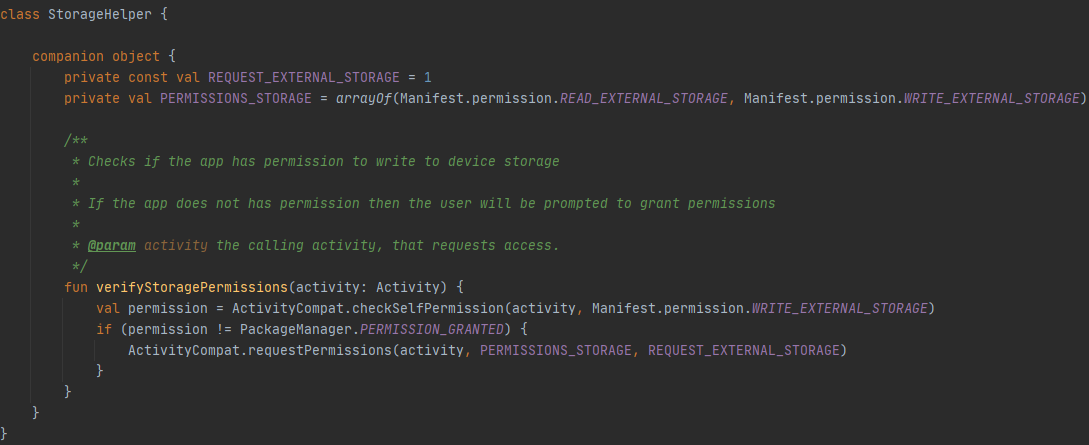


Abbildung : Beispiel für das Single Responsibility Principle anhand vom StorageHelper

* Open- / Closed-Principle
  + Offen für Erweiterungen, geschlossen für Änderungen

Die Klasse BasicAdapter ist eine Klasse die als Grundlage der Adapter, die in Recyclingviews benötigt werden. Die Klasse stellt immer das die gleichen Methoden und das Field datazur Verfügung, da diese immer wieder benötigt werden. Jedoch unterscheiden sich immer die Datentypen, deswegen die Abstrahierung mit dem Generic dargestellt. Ebenfalls fällt die Klasse unter das DRY-Prinzip, da der Code in jedem Adapter immer ähnlich wäre, da sich, wie bereits erwähnt, nur der Datentyp unterscheidet. Die klasse bietet die Möglichkeit sie, in einer Implementierung davon, zu erweitern, sodass weitere benötigte Funktionen implementiert werden können.

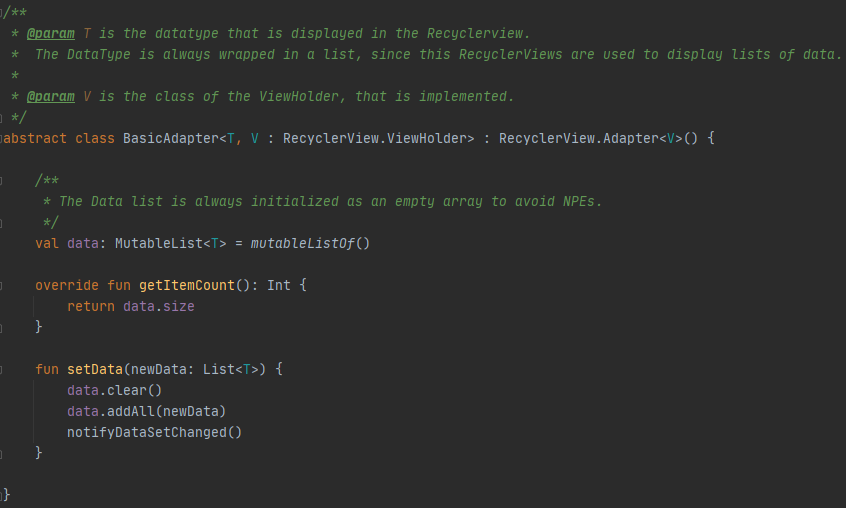


Abbildung : BasicAdapter als Beispiel für eine Klasse die offen für Veränderung ist, aber geschlossen für Veränderungen.

* Liskov Substitution Principle
  + Eine abgeleitete Klasse soll an jeder beliebigen Stelle ihre Basisklasse ersetzen können, ohne, dass es zu unerwünschten Nebeneffekten kommt.

Die Klasse HomeAdapterist eine Erweiterung des BasicAdaptersaus Abbildung 6, dessen Funktionen erweitert werden jedoch nicht modifiziert. Das Liskov Substitution Principle und das Open- / Closed- Principle hängen sehr voneinander ab.

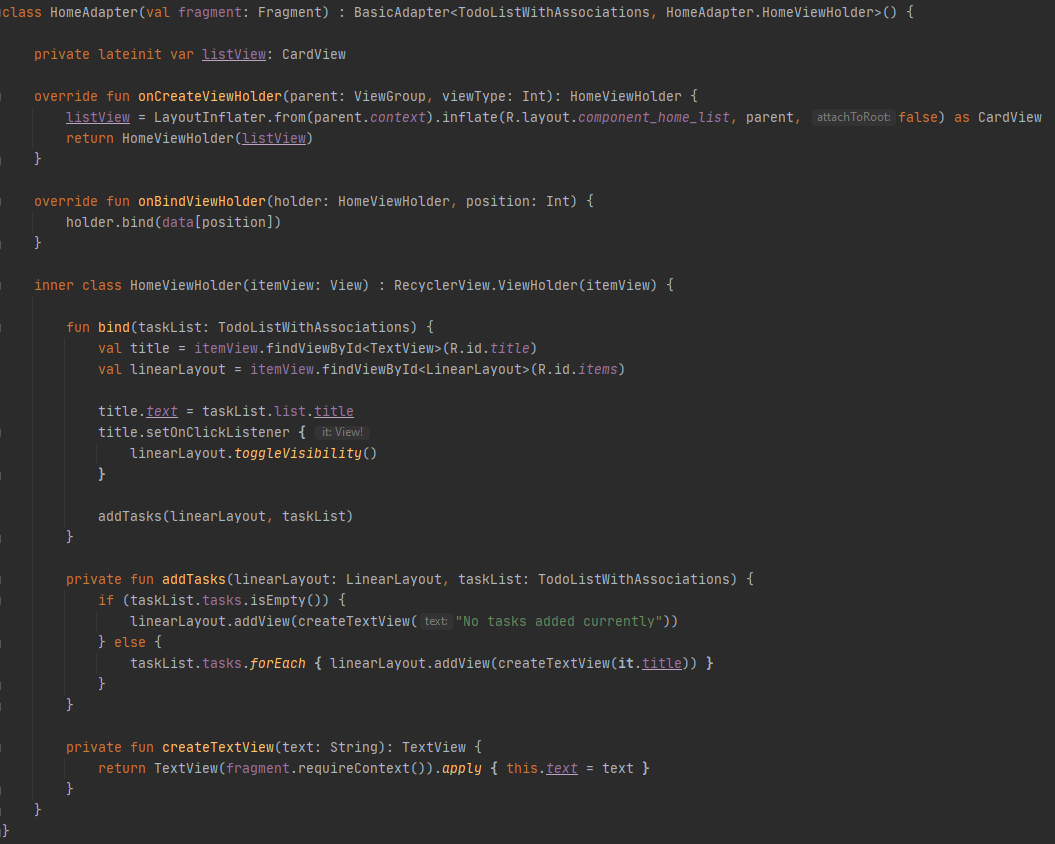


Abbildung : Beispiel für die Erweiterung der Basisklasse BasicAdapter, die jedoch von ihren Funktionen nicht modifiziert wird.

Der Homeadapter benötigt ebenfalls keine Implementierung der Basismethoden, da diese sich nicht verändern von der Basisimplementierung. Sollte sich jedoch das Verhalten verändern, indem z.B. weitere Elemente der Anwendung „benachrichtig“ werden müssen über Veränderung der Liste, dann könnte dies in den Sonderfällen durch eine eigene Implementierung gemacht werden.

* Interface Segregation Principle
  + Kleine Interfaces mit bestimmten anwendungszwecken sind sinnvoller als ein großes Interface, das alles kann.

Das Interface PlaceEditor wurde in mehrere kleineren Interfaces unterteilt, da das Scope der jeweiligen Interfaces begrenzt werden können. Das Scope wurde begrenzt, da bei der Parameterübergabe so z.B. die Berichtigung beschränkt werden kann, damit eine Methode nur bestimmte Funktionen aufrufen kann. Ein Beispiel dafür ist der PlaceViewHolder, der nur ein PlaceRemover entgegennimmt, sodass dieser nur Orte löschen kann.

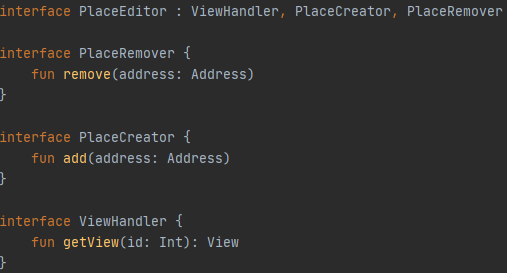


Abbildung : Beispiel für das Interface Segregation Principle anhand vom PlaceEditor

Das entspricht wieder dem Single Responsibility Principle, da dieser nur die Aufgabe hat es anzuzeigen und die Orte zu löschen auf Wunsch und eine andere Komponente den Vorgang des Hinzufügens zuständig ist.

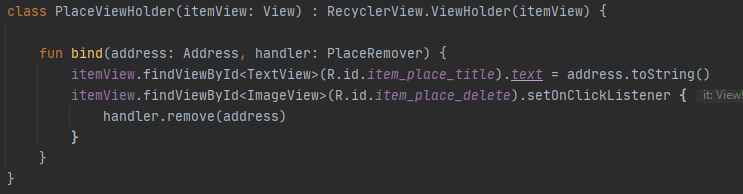


Abbildung : Beispiel für die Verwendung eines Interfaces, das in mehrere kleinere aufgeteilt wurde, sodass Funktionen beschränkt werden können.

* Dependency Inversion Principle
  + Abstraktion sollte nicht von Details abhängen, die Details sollten von der Abstraktion abhängen.

Ein gutes Beispiel, aus meinem Code, für dieses Prinzip ist das Savable-Interface. Das Interface stellt nur eine Methode zur Verfügung, die save heißt. Die Methode liefert einen Boolean zurück, dass in der ersten Basisimplementierung, SavableFragment, verwendet wird. Das SavableFragment ist eine Abstrahierung, damit der code für das Speichern ebenfalls unter das DRY-Prinzip fällt.

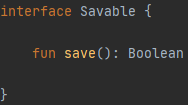


Abbildung : Das Interface Savable

Wenn in einem SavableFragmentauf das Speicherikon gedrückt wird, dann wird immer als Defaultaktion einfach nur zurückgegangen, zum Aufrufer. Dieses Verhalten stammt daraus, dass SavableFragment in der Regel auch SubFragments sind, die zur Bearbeitung von Daten dienen und innerhalb von einem anderen Fragment sind, deswegen wird dann „nach oben“ bzw. zurück gegangen zum Aufrufer. Das einzige Detail, das bis hier berücksichtigt wurde, ist dass die Methode immer ein Booleanzurückliefert, wie es das tut ist dem Interface egal, Hauptsache es passiert. Die Details der Implementierung werden weiterhin verdeutlicht in den spezifischen Fragmentimplementierungen.

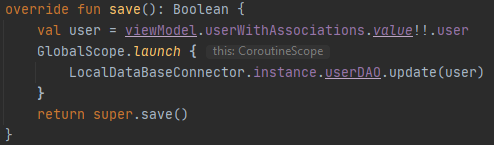


Abbildung : Save-implementierung aus dem AccountSettingsFragment

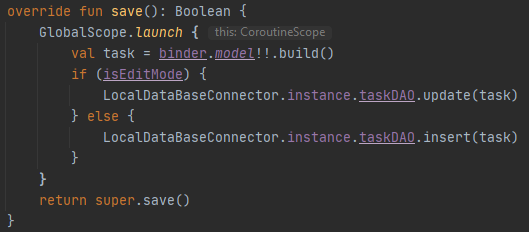


Abbildung : Save-implementierung aus dem TaskEditorFragment

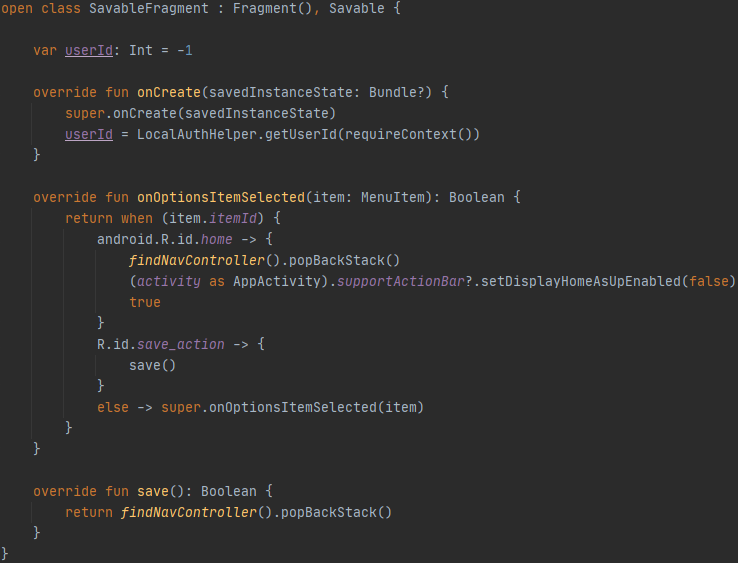


Abbildung : Basisimplementierung des SavableFragments

Die Details der Implementierung, wie z.B. das Aufrufen der Datenbank, sind für das Interface und den Basisklassen nicht relevant.

## GRASP

In diesem Kapitel werden die GRASP-Prinzipien erläutert und anhand von Beispielen dargelegt.

GRASP – Die Zuständigkeit einer Aufgabe wird anhand des vorhandenen Wissens verteilt.

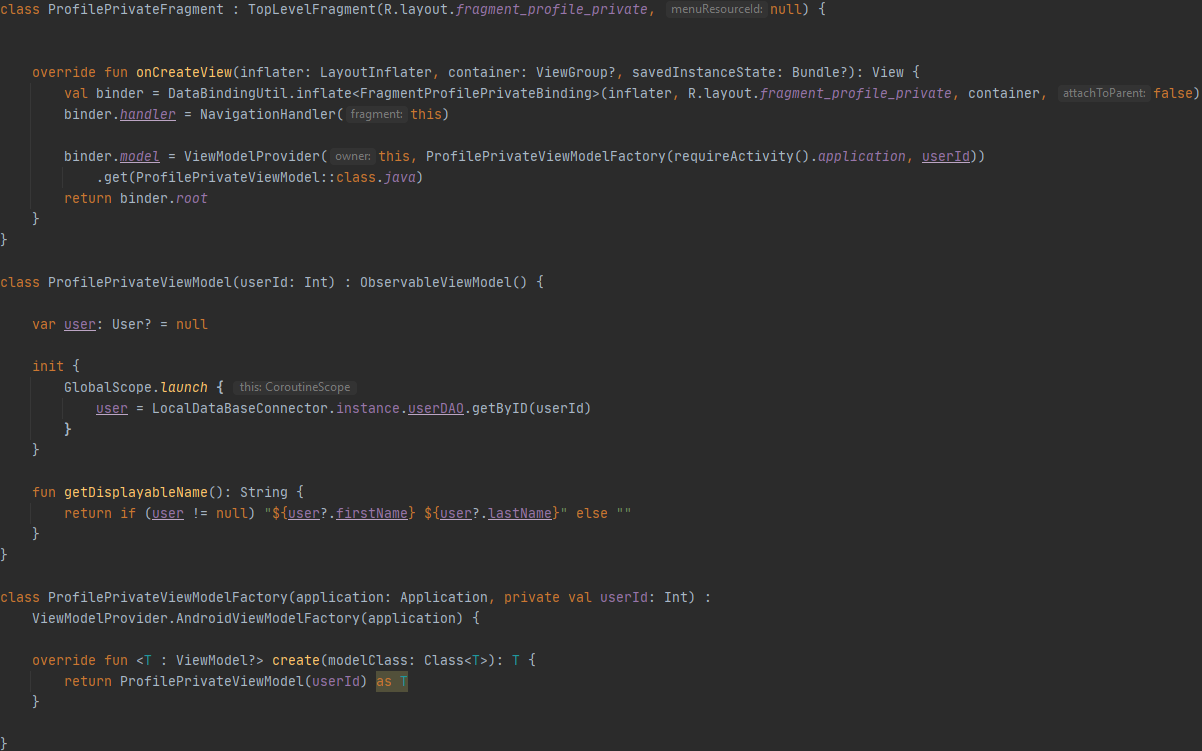


Abbildung : ProfilePrivateFragment das als Controller dient

Das Fragment ProfilePrivateFragment ist der „Controller“ nach dem Model-View-ViewModel- (MVVM) Entwurfsmuster, das häufig verwendet wird bei der der Android Entwicklung. Die klasse sorgt dafür, dass das entsprechende Layout mit den dazugehörigen Daten gefüllt wird und wenn es zu Aktionen kommt, die entsprechende Funktion aufgerufen wird. Der Controller erstellt die Verbindung zur Datenbank, bzw. delegiert diese Verantwortung an das ViewModel, dass das Wissen hat welche Daten wieder benötigt werden. Der Controller weiß nicht um welche Daten es sich im Endeffekt handelt, sondern nur welche Factory benötigt wird, um ein ViewModel zu erstellen, dass die benötigten Daten bereitstellt. Der Controller verbindet die Daten dann mit dem Layout.

## DRY

* Don’t**R**epeat **Y**ourself – Das Prinzip beschreibt das Code der geschrieben wurde nicht wiederholt werden sollte. Wenn Code mehrmals verwendet wird, sollte dieser ausgelagert und wenn notwendig abstrahiert werden, sodass dieser an den benötigten Stellen verwendet werden kann.

Ein sehr gutes Beispiel in meinem Code dafür ist die Erstellung der (Sub Level-) Fragmente. Ein Sub Level Fragment, ist ein Fragment, das eine innerhalb von einem anderen Fragment verwendet wird für eine detaillierte Darstellung. Der Code für die Erstellung eines solches Fragmentes ist immer gleich, es unterscheiden sich nur das Layout und das verwendete Menu. Das Layout und Menu werden im Konstruktor übergeben und diese dann bei der von Android aufgerufenen Erstellungsmethode verwendet zum Aufbauen des Layouts.

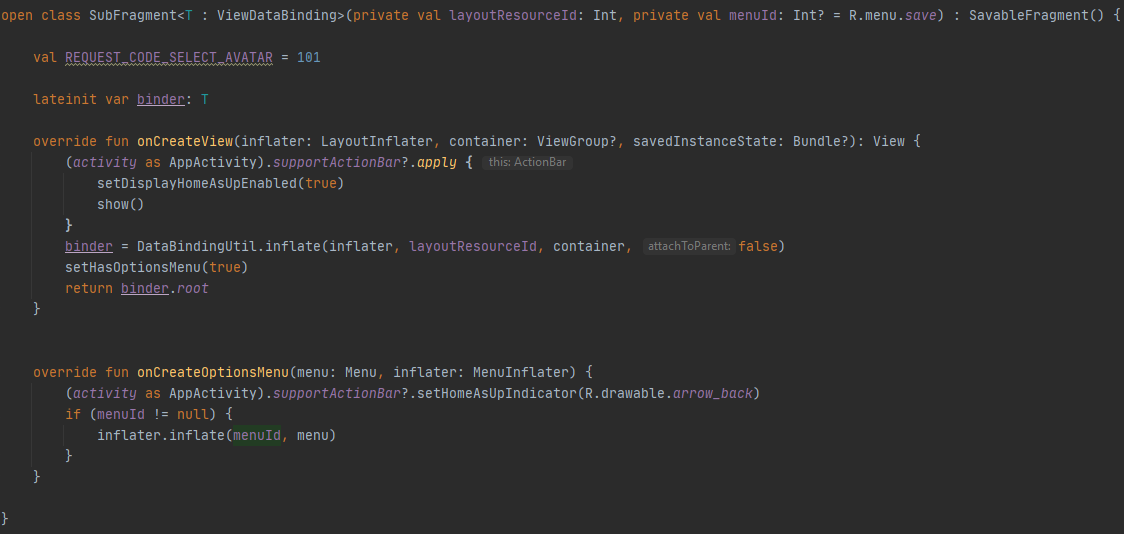


Abbildung : Basisklasse die für die Abstrahierung der Sub-Level Fragmente verwendet wird.

# Refactoring

In diesem Kapitel werden Codesmells erläutert, identifiziert und bereinigt, wenn welche gefunden wurden. Anzeichen von Codesmells sind folgende Faktoren:

* Duplizierte Code
* Lange bzw. Komplexe Methoden / Klassen
* Viele Parameter / Fields
* Lange Zeilen (Nicht immer Notwendigerweise z.B. Chained-calls auf Streams)
* Leere Funktionen

Meine Anwendung bietet nicht viel Code für Refactoring an, da diese meist von mir schon beseitigt werden bei der Implementierung. Einige Codesmells gab es jedoch, die im Commit „6c17e864c64d484d1793ef7dbcf0f3706c500a54“ beseitigt wurden. Hier handelt es sich hauptsächlich darum, dass einige Methoden einen leeren Funktionskörper hatten, sowie das eine Klasse, die nur aus statischen Methoden besteht, kein Private Konstruktor hatte. Die Codesmells wurden entfernt und es bestehen keine weiteren im Code. Damit ich belegen, kann das die Wahrscheinlichkeit für Codesmells gering ist habe ich mit einem Statistic-Plugin meine Anzahl Zeilen an Code in meinen Klassen auswerten lassen. Das Ergebnis ist, das meine (Kotlin-) Klassen im Durchschnitt 46 Zeilen Code haben und maximal 175 Zeilen Code, dementsprechend kann die Schlussfolgerung gezogen werden, dass diese gut gekapselt sind und den vorher präsentierten Programmierprinzipien entsprechen. Ein weiterer Beleg ist das Plugin Sonarlint, dieses Plugin untersucht auf Duplizierten Code, Methoden Komplexität (Zeilen und verschachtelungstiefe), etc.

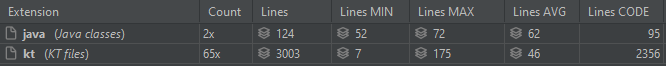


Abbildung : Auswertung des Statistic-Plugins für meine Anwendung

Nach dem Entfernen der ebengenannten Codesmells besteht der Report aus größtenteils entfernen von TODOs. Die Codesmells in der Klasse SecurityHelper sind in dem Fall nicht umsetzbar, da der Code an dieser Stelle von genau dieser Implementierung abhängt und durch die Veränderung zu dem vorgeschlagenen Code die Funktionen nicht mehr richtig funktionieren würden. Der Codesmell aus der Klasse PushNotificationManager, dass die private nicht verwendete Methode entfernt werden kann, kommt daher das es eine Funktion ist die zukünftig unterstütz werden soll und dann diese bereits implementierte Methode verwendet werden kann.

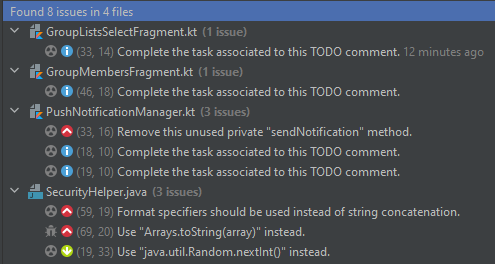


Abbildung : Report zu Codesmells vom Sonarlint Plugin

# Legacycode

Meine Anwendung besitzt nur eine Abhängig, dass ist die Abhängigkeit zur Datenbank. Die Abhängigkeit zur Datenbank wurde bereits in dem Kapitel Unit Tests heruntergebrochen und für Testzwecke entfernt, deswegen werde ich diesen Vorgang nicht erneut erläutern. Des Weiteren besitzt die Anwendung keine Gott-Klassen, da wie schon im Kapitel Refactoring dargelegt wurde keine Klasse mit vielen Zeilen Code oder vielen Methoden vorhanden ist. Es gibt nur eine bekannte Stelle, an der es Globale Konstanten gibt, die angepasst werden können. Der LocalAuthHelper hat 4 Fields, bei denen es sich um Konstante Globale Variablen handelt, die jedoch keine Globale Variable sein müssen. Die Fields können angepasst werden und zu Lokale Variablen gemacht werden, da der Zugriff nur innerhalb der Klasse stattfindet. Damit im Code nur noch Lokal, also innerhalb der Klasse auf die Fields zugegriffen werden kann, wird der Modifier private davorgesetzt. Das Ergebnis ist im Commit „a43e09c2d632d270d528f8df841bb6ce09aed508“ zu sehen.

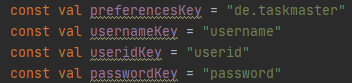


Abbildung : Globale Fields in LocalAuthHelper

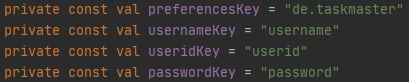


Abbildung : Fields in LocalAuthHelper nach der Modifizierung

# Entwurfsmuster

Das beste Beispiel eines Entwurf Musters in meiner Anwendung ist das MVVM-Muster. Das Muster beschreibt, dass die Anwendung ähnlich wie bei dem Model-View-Controller (MVC)-Muster in 3 Teile aufgeteilt ist.

* Model - das Datenobjekt aus der Datenbank,
* View – Das Design / Layout der Benutzeroberfläche
* ViewModel – Der „Halter“ der Daten (Model) die angezeigt werden in der Benutzeroberfläche (View)

Dieses Entwurfsmuster wird von Google empfohlen für Android Anwendung und ist ebenfalls für diesen Anwendungsfall optimiert. Ein Beispiel aus meiner Anwendung ist die Home-Perspektive. Die Perspektive wird in 2 Dateien beschrieben, erstens die Layout-XML mit dem Namen „fragment\_home.xml“ und die Kotlin-Datei „HomeFragment“. Die Kotlin-Datei beinhaltet mehrere Klassen, da diese Art und Weise des Programmierens in Kotlin kein Codesmell oder schlechter Programmierstil ist. Die Datei beinhaltet das Fragment, das als Kleber für die drei Schichten des MVVM-Musters funktioniert. In dem Fragment wird das Layout angegeben, sowie mit dem ViewModel verbunden und das ViewModel beinhaltet das Model. Das ViewModel ist ein Wrapper um das Model und bietet diverse sinnvolle Funktionen, die für Android Anwendungen sehr relevant sind. Ebenfalls beinhaltet die Datei einen Adapter, der benötigt wird für eine richtige Darstellung der Daten in der Ansicht, da an dieser Stelle Programatisch definiert werden muss, wie die Daten an das Element in der Benutzeroberfläche gebunden werden muss.

# Domain-Driven-Design