

# 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
Abbildungsverzeichnis.....	3
1    Einleitung .....	5
2    Unit Tests .....	6
3    Programming Principles.....	9
3.1    SOLID.....	9
3.2    GRASP .....	13
3.3    DRY.....	14
4    Refactoring.....	15
5    Legacycode .....	17
6    Entwurfsmuster .....	18
7    Domain-Driven-Design .....	18

## Abbildungsverzeichnis

Abbildung 1: Tests für die Benutzerverwaltung.....	6
Abbildung 2: Mock-Implementierung des Benutzerverwaltung Interfaces .....	7
Abbildung 3: Test für den Passwort-Generator und -Validierer .....	8
Abbildung 4: Test des Datenbankadapters .....	9
Abbildung 5: Beispiel für das Single Responsibility Principle anhand vom StorageHelper .....	10
Abbildung 6: BasicAdapter als Beispiel für eine Klasse die offen für Veränderung ist, aber geschlossen für Veränderungen.....	10
Abbildung 7: Beispiel für die Erweiterung der Basisklasse BasicAdapter, die jedoch von ihren Funktionen nicht modifiziert wird.....	11
Abbildung 8: Beispiel für das Interface Segregation Principle anhand vom PlaceEditor .....	12
Abbildung 9: Beispiel für die Verwendung eines Interfaces, das in mehrere kleinere aufgeteilt wurde, sodass Funktionen beschränkt werden können. ....	12
Abbildung 10: Das Interface Savable.....	12
Abbildung 11: Save-implementierung aus dem AccountSettingsFragment.....	13
Abbildung 12: Save-implementierung aus dem TaskEditorFragment .....	13
Abbildung 13: Basisimplementierung des SavableFragments .....	13
Abbildung 14: ProfilePrivateFragment das als Controller dient .....	14
Abbildung 15: Basisklasse die für die Abstrahierung der Sub-Level Fragmente verwendet wird. ....	15

Abbildung 16: Auswertung des Statistic-Plugins für meine Anwendung.....	16
Abbildung 17: Report zu Codesmells vom Sonarlint Plugin .....	16
Abbildung 18: Globale Fields in LocalAuthHelper .....	17
Abbildung 19: Fields in LocalAuthHelper nach der Modifizierung.....	17

## 1 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.

## 2 Unit Tests

Unit Tests sind Tests die kleinen Einheiten der Codebase auf Richtigkeit überprüfen.

```
/**
 * Ziel dieser Klasse ist es zufällig gewählte Funktionen der Datenbank zu testen mit Mockobjekten.
 * Bei der "echten" Anwendung wird dieser Code nicht vom Programmierer geschrieben,
 * sondern die Kommunikation wird von einem Framework übernommen.
 */
class UserUnitTests {

    private lateinit var user: User
    private lateinit var userImpl: UserImpl

    private val defaultUsername = "TheYaINN"

    @Before
    fun init() {
        val pass = "securepass"
        val split = SecurityHelper.generateHashedPassword(pass).split("::")
        user = User(
            username = defaultUsername,
            password = split[2],
            salt = split[1],
            iterations = split[0].toInt(),
            firstName = "Bengt",
            lastName = "Joachimsohn",
            email = "bengt@joachimsohn.de"
        )
        userImpl = UserImpl()
    }

    @Test
    fun create() {
        GlobalScope.async { this CoroutineScope
            userImpl.insert(user)
            assert(userImpl.getUserName(defaultUsername) == user)
        }
    }

    @Test
    fun update() {
        val alias = "Friedrich"
        val modifiedUser = user.apply { username = alias }
        GlobalScope.async { this CoroutineScope
            userImpl.insert(user)
            userImpl.update(modifiedUser)
            assert(userImpl.getUserName(alias) == user)
        }
    }
}
```

Abbildung 1: Tests für die Benutzerverwaltung

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.

Wenn z.B. Komponenten der Anwendung getestet werden 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 demonstrierungszwecken selbst nochmal implementiert, um zu testen, dass die Methoden das Abbilden was benötigt wird.

```

/**
 * Mock des Interfaces für die Benutzerverwaltung
 */
class UserImpl : UserDao {

    private var users: MutableList<User> = mutableListOf()

    override suspend fun insert(user: User) {
        println("Inserting user")
        users.add(user)
    }

    override suspend fun update(user: User) {
        println("Updating user")
        users.first { it == user }.apply {
            username = user.username
            email = user.email
            firstName = user.firstName
            lastName = user.lastName
            password = user.password
            salt = user.salt
            iterations = user.iterations
            profilePicture = user.profilePicture
        }
    }

    override suspend fun delete(user: User) {
        println("Deleting user")
        users.remove(user)
    }

    override suspend fun getById(id: Int): User? {
        println("Getting user by ID")
        return users.find { it.userId == id }
    }

    override fun getByUserName(username: String): User? {
        println("Getting user by UserName")
        return users.find { it.username == username }
    }
}

```

Abbildung 2: Mock-Implementierung des Benutzerverwaltung Interfaces

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.

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.

```
/**
 * Diese Klasse sammelt alle Unit Tests der Helfer-Klassen,
 * bei denen kein Android Context oder Application benötigt werden,
 * da diese aufzubauen ein riesen Overhead haben.
 */
class HelperUnitTests {

    @Test
    fun security() {
        val pass = "SecurePassword"
        val hashed = SecurityHelper.generateHashedPassword(pass)
        val split = hashed.split(":", "")
        assert(SecurityHelper.validatePassword(pass, split[2], split[1], split[0].toInt()))
    }
}
```

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

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.

Wichtige Elemente einer Anwendung zu testen und auf Richtigkeit zu überprüfen sind z.B.

```
class ConverterUnitTests {

    @Test
    fun from_status() {
        val converter = DBConverter()
        val status = Status.OPEN
        assert(converter.fromStatus(status) == "OPEN")
    }

    @Test
    fun to_status() {
        val converter = DBConverter()
        val status = Status.OPEN
        assert(converter.toStatus(status: "OPEN") == status)
    }

}
```

Abbildung 4: Test des Datenbankadapters

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.

### 3 Programming Principles

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

#### 3.1 SOLID

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

```

class StorageHelper {
    companion object {
        private const val REQUEST_EXTERNAL_STORAGE = 1
        private val PERMISSIONS_STORAGE = arrayOf(Manifest.permission.READ_EXTERNAL_STORAGE, Manifest.permission.WRITE_EXTERNAL_STORAGE)

        /**
         * Checks if the app has permission to write to device storage
         *
         * If the app does not has permission then the user will be prompted to grant permissions
         *
         * @param activity the calling activity, that requests access.
         */
        fun verifyStoragePermissions(activity: Activity) {
            val permission = ActivityCompat.checkSelfPermission(activity, Manifest.permission.WRITE_EXTERNAL_STORAGE)
            if (permission != PackageManager.PERMISSION_GRANTED) {
                ActivityCompat.requestPermissions(activity, PERMISSIONS_STORAGE, REQUEST_EXTERNAL_STORAGE)
            }
        }
    }
}

```

Abbildung 5: Beispiel für das Single Responsibility Principle anhand vom StorageHelper

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

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

```

/**
 * @param T is the datatype that is displayed in the RecyclerView.
 * The Data type is always wrapped in a list, since this RecyclerViews are used to display lists of data.
 *
 * @param V is the class of the ViewHolder, that is implemented.
 */
abstract class BasicAdapter<T, V : RecyclerView.ViewHolder> : RecyclerView.Adapter<V>() {

    /**
     * The Data list is always initialized as an empty array to avoid NPEs.
     */
    val data: MutableList<T> = mutableListOf()

    override fun getItemCount(): Int {
        return data.size
    }

    fun setData(newData: List<T>) {
        data.clear()
        data.addAll(newData)
        notifyDataSetChanged()
    }
}

```

Abbildung 6: BasicAdapter als Beispiel für eine Klasse die offen für Veränderung ist, aber geschlossen für Veränderungen.

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 data zur 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.

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

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

```
class HomeAdapter(val fragment: Fragment) : BasicAdapter<TodoListWithAssociations, HomeAdapter.HomeViewHolder>() {
    private lateinit var listView: CardView

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): HomeViewHolder {
        listView = LayoutInflater.from(parent.context).inflate(R.layout.component_home_list, parent, attachToRoot: false) as CardView
        return HomeViewHolder(listView)
    }

    override fun onBindViewHolder(holder: HomeViewHolder, position: Int) {
        holder.bind(data[position])
    }

    inner class HomeViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        fun bind(taskList: TodoListWithAssociations) {
            val title = itemView.findViewById<TextView>(R.id.title)
            val linearLayout = itemView.findViewById<LinearLayout>(R.id.items)

            title.text = taskList.list.title
            title.setOnClickListener { itemView.linearLayout.toggleVisibility() }
        }

        addTasks(linearLayout, taskList)
    }

    private fun addTasks(linearLayout: LinearLayout, taskList: TodoListWithAssociations) {
        if (taskList.tasks.isEmpty()) {
            linearLayout.addView(createTextView(text: "No tasks added currently"))
        } else {
            taskList.tasks.forEach { linearLayout.addView(createTextView(it.title)) }
        }
    }

    private fun createTextView(text: String): TextView {
        return TextView(fragment.requireContext()).apply { this.text = text }
    }
}
```

Abbildung 7: 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 „benachrichtigt“ 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

```
interface PlaceEditor : ViewHandler, PlaceCreator, PlaceRemover {
    interface PlaceRemover {
        fun remove(address: Address)
    }
    interface PlaceCreator {
        fun add(address: Address)
    }
    interface ViewHandler {
        fun getView(id: Int): View
    }
}
```

Abbildung 8: Beispiel für das Interface Segregation Principle anhand vom PlaceEditor

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.

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.

```
class PlaceViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    fun bind(address: Address, handler: PlaceRemover) {
        itemView.findViewById<TextView>(R.id.item_place_title).text = address.toString()
        itemView.findViewById<ImageView>(R.id.item_place_delete).setOnClickListener { it: View
            handler.remove(address)
        }
    }
}
```

Abbildung 9: 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.

```
interface Savable {
    fun save(): Boolean
}
```

Abbildung 10: Das Interface Savable

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.

```

open class SavableFragment : Fragment(), Savable {

    var userId: Int = -1

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        userId = LocalAuthHelper.getUserId(requireContext())
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            android.R.id.home -> {
                findNavController().popBackStack()
                (activity as AppCompatActivity).supportActionBar?.setDisplayHomeAsUpEnabled(false)
                true
            }
            R.id.save_action -> {
                save()
            }
            else -> super.onOptionsItemSelected(item)
        }
    }

    override fun save(): Boolean {
        return findNavController().popBackStack()
    }
}

```

Abbildung 13: Basisimplementierung des SavableFragments

Wenn in einem SavableFragment auf das Speicherkon 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 Boolean zurückliefert, wie es das tut ist dem Interface egal, Hauptsache es passiert. Die Details der Implementierung werden weiterhin verdeutlicht in den spezifischen Fragmentimplementierungen.

```

override fun save(): Boolean {
    val user = viewModel.userWithAssociations.value!!.user
    GlobalScope.launch { this: CoroutineScope
        LocalDataBaseConnector.instance.userDAO.update(user)
    }
    return super.save()
}

```

Abbildung 11: Save-implementierung aus dem AccountSettingsFragment

```

override fun save(): Boolean {
    GlobalScope.launch { this: CoroutineScope
        val task = binder.model!!.build()
        if (isEditMode) {
            LocalDataBaseConnector.instance.taskDAO.update(task)
        } else {
            LocalDataBaseConnector.instance.taskDAO.insert(task)
        }
    }
    return super.save()
}

```

Abbildung 12: Save-implementierung aus dem TaskEditorFragment

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

### 3.2 GRASP

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

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

```
class ProfilePrivateFragment : TopLevelFragment(R.layout.fragment_profile_private, menuResourceId: null) {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View {
        val binder = DataBindingUtil.inflate<FragmentProfilePrivateBinding>(inflater, R.layout.fragment_profile_private, container, attachToParent: false)
        binder.handler = NavigationHandler(fragment: this)

        binder.model = ViewModelProvider(ownen: this, ProfilePrivateViewModelFactory(requireActivity().application, userId))
            .get(ProfilePrivateViewModel::class.java)
        return binder.root
    }
}

class ProfilePrivateViewModel(userId: Int) : ObservableViewModel() {

    var user: User? = null

    init {
        GlobalScope.launch { this CoroutineScope
            user = LocalDataBaseConnector.instance.userDAO.getByID(userId)
        }
    }

    fun getDisplayableName(): String {
        return if (user != null) "${user?.firstName} ${user?.lastName}" else ""
    }
}

class ProfilePrivateViewModelFactory(application: Application, private val userId: Int) :
    ViewModelProvider.AndroidViewModelFactory(application) {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return ProfilePrivateViewModel(userId) as T
    }
}
```

Abbildung 14: 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, das 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, das die benötigten Daten bereitstellt. Der Controller verbindet die Daten dann mit dem Layout.

### 3.3 DRY

- **Don't Repeat Yourself** – 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 solchen 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

```
open class SubFragment<T : ViewDataBinding>(private val layoutResourceId: Int, private val menuId: Int? = R.menu.save) : SavableFragment() {  
  
    val REQUEST_CODE_SELECT_AVATAR = 101  
  
    lateinit var binder: T  
  
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View {  
        (activity as AppCompatActivity).supportActionBar?.apply { this.ActionBar  
            setDisplayHomeAsUpEnabled(true)  
            show()  
        }  
        binder = DataBindingUtil.inflate(inflater, layoutResourceId, container, attachToParent: false)  
        setHasOptionsMenu(true)  
        return binder.root  
    }  
  
    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {  
        (activity as AppCompatActivity).supportActionBar?.setHomeAsUpIndicator(R.drawable.arrow_back)  
        if (menuId != null) {  
            inflater.inflate(menuId, menu)  
        }  
    }  
}
```

Abbildung 15: Basisklasse die für die Abstrahierung der Sub-Level Fragmente verwendet wird.

von Android aufgerufenen Erstellungsmethode verwendet zum Aufbauen des Layouts.

## 4 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,











Extension	Count	Lines	Lines MIN	Lines MAX	Lines AVG	Lines CODE
 <b>java</b> ( <i>Java classes</i> )	2x	 124	 52	 72	 62	95
 <b>kt</b> ( <i>KT files</i> )	65x	 3003	 7	 175	 46	2356

Abbildung 16: Auswertung des Statistic-Plugins für meine Anwendung

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.

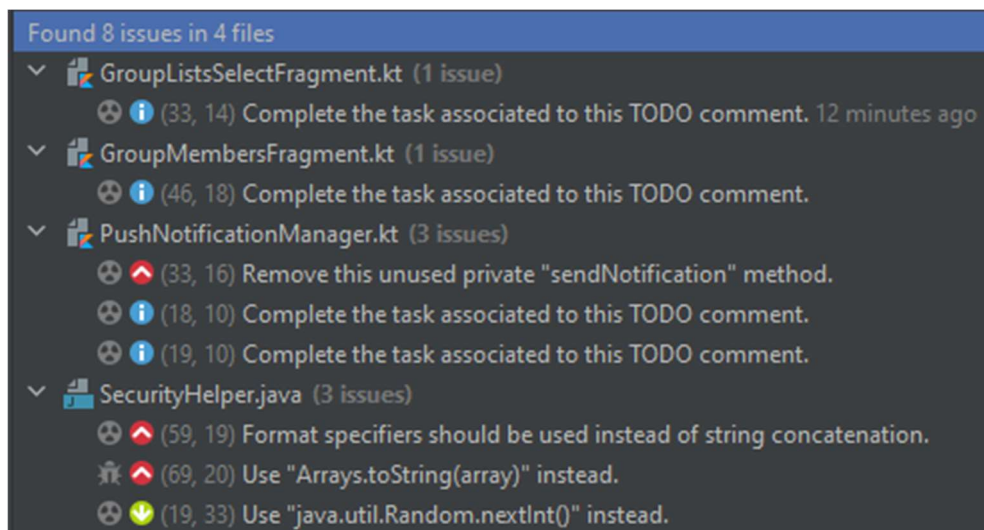


Abbildung 17: Report zu Codesmells vom Sonarlint Plugin

Nach dem Entfernen der eben genannten 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 unterstützt werden soll und dann diese bereits implementierte Methode verwendet werden kann.

## 5 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

```
const val preferencesKey = "de.taskmaster"  
const val usernameKey = "username"  
const val useridKey = "userid"  
const val passwordKey = "password"
```

Abbildung 18: Globale Fields in LocalAuthHelper

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.

```
private const val preferencesKey = "de.taskmaster"  
private const val usernameKey = "username"  
private const val useridKey = "userid"  
private const val passwordKey = "password"
```

Abbildung 19: Fields in LocalAuthHelper nach der Modifizierung

## 6 Entwurfsmuster

Das beste Beispiel eines Entwurfsmusters 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.

## 7 Domain-Driven-Design

- 1) Ubiquitous Language – Domänen (Technisch- & Fachspezifische Begriffe) übergreifende Sprache

Meine Technische Domäne unterscheidet sich nicht großartig von der Fachspezifischen Domäne, jedoch können die Begriffe erläutert werden.

- (1) ToDoListe – Eine Datentyp der eine Liste darstellt, die eine Sammlung von Aufgaben verwaltet, die alle als erledigt markiert werden müssen, damit die Liste als abgearbeitet markiert werden kann.
- (2) Task – Eine Datentyp der eine Aufgabe darstellt, dass eine Tätigkeit ist, bei der dokumentiert werden kann, ob diese bereits erledigt wurde.
- (3) Tag – Ein Datentyp, der ein Schlagwort darstellt, das Suchwort kann für die Suche von ToDoListen und Aufgaben verwendet werden.
- (4) Status – Ein binär definierter Datentyp (Enum), der angibt, ob eine Aufgabe bereits erledigt wurde.
- (5) User – Ein Datentyp der einen Benutzer darstellt, in dem alle relevanten Benutzerinformationen abgelegt sind.

Es wird nur das Android Rooms (und das Mock-Repository zum Testen) Repository verwendet in der Anwendung. Das Repository stellt die gesamte Datenstruktur sowie alle gespeicherten Daten der Anwendung dar. Zukünftig ist für die Anwendung ein weiteres Repository, dass ein Mirror des bereits vorhandenen Repositories darstellt, geplant. Das zweite Repository soll es ermöglichen, dass Daten unabhängig vom Endgerät aufgerufen werden können, wenn der Benutzer sich angemeldet hat.

Die Entities der Domäne sind:

- (1) Address – Ort eine ToDoListe
- (2) Group – Eine Gruppe der Benutzer und Listen hinzugefügt werden können
- (3) Tag – Schlagwort für die Suche
- (4) Task – eine Aufgabe in einer ToDoListe
- (5) ToDoList – eine Sammlung von Aufgaben
- (6) User – ein Benutzer

Ebenfalls sind folgende Value Objects vorhanden

- (1) Deadline – Zeitpunkt zu dem eine ToDoListe erledigt sein muss
- (2) Repeat – Wie oft eine ToDoListe wiederholt werden soll
- (3) Status – Der Status (erledigt / nicht erledigt) einer Aufgabe

Aggregate die in der Anwendung verwendet werden zur Zusammenfassung von mehreren Entities:

- (1) TodoListWithAssociations – Damit einer ToDoListe ihre Aufgaben und Schlagwörter zugeordnet werden kann

- (2) GroupWithToDoLists – Damit eine Gruppe ihre zugehörigen ToDoListen zugeordnet werden kann
- (3) UserGroupCrossRef – Damit ein Benutzer einer Gruppe zugeordnet werden kann
- (4) UserWithAssociations – Damit zu einem Benutzer sein Gruppen, ToDoListen und Orte zugeordnet werden kann