

# Cleaning Plan App

## Advanced Software Engineering

Isabell Reitler

April 2021

### Contents

<b>1</b>	<b>Einführung</b>	<b>3</b>
1.1	Die Anwendung . . . . .	3
1.2	Deployment . . . . .	3
1.2.1	API-Deployment . . . . .	3
1.2.2	App-Deployment . . . . .	4
1.3	Bedienung der Android-App . . . . .	4
1.4	Über die Doku . . . . .	5
<b>2</b>	<b>Domain Driven Design</b>	<b>6</b>
2.1	Ubiquitous Language . . . . .	6
2.2	Verwendete Muster . . . . .	7
2.2.1	Value Objects . . . . .	7
2.2.2	Entites . . . . .	8
2.2.3	Aggregates . . . . .	8
2.2.4	Repositories . . . . .	8
2.2.5	Domain Services . . . . .	9
<b>3</b>	<b>Clean Architecture</b>	<b>10</b>
3.1	Core . . . . .	10
3.2	Domain . . . . .	10
3.3	Application . . . . .	11
3.4	Plugin . . . . .	12

<b>4</b>	<b>Programming Principles</b>	<b>13</b>
4.1	SOLID . . . . .	13
4.1.1	Single-Responsibility-Prinzip . . . . .	13
4.1.2	Open-Closed-Prinzip . . . . .	14
4.1.3	Liskov'sche Substitutionsprinzip . . . . .	15
4.1.4	Interface-Segregation-Prinzip . . . . .	15
4.1.5	Dependency-Inversion-Prinzip . . . . .	15
4.2	GRASP . . . . .	16
4.2.1	Low Coupling . . . . .	16
4.2.2	High Cohesion . . . . .	17
4.2.3	Information Expert . . . . .	17
4.2.4	Creator . . . . .	17
4.2.5	Indirection . . . . .	18
4.2.6	Polymorphism . . . . .	18
4.2.7	Controller . . . . .	18
4.2.8	Pure Fabrication . . . . .	19
4.2.9	Protected Variations . . . . .	19
4.3	DRY . . . . .	20
<b>5</b>	<b>Entwurfsmuster: Singleton-Pattern</b>	<b>21</b>
<b>6</b>	<b>Refactoring</b>	<b>23</b>
6.1	Refactoring 1: Extract Method . . . . .	23
6.2	Refactoring 2: Replace Magic Number with Symbolic Constant . . . . .	24
<b>7</b>	<b>Testing</b>	<b>26</b>

# 1 Einführung

Das Ziel dieses Advanced Software Engineering Projekts ist die Entwicklung einer “Cleaning Plan App”.

## 1.1 Die Anwendung

Das Ziel der App ist es einen Haushalt von mehreren Personen, wie z.B. einer WG oder einer größeren Familie zu organisieren. Dazu bietet die App folgenden Use Cases:

- Authentifizierung bei der Anwendung über Google Firebase (Google Account)
- Anlegen eines eigenen Haushalts
- Beitreten zu einem bestehenden Haushalt
- Aufgaben erstellen
- Aufgaben als erledigt Kennzeichnen
- Aufgaben löschen
- Den Ferienmodus aktivieren, wenn man einige Tage nicht zu Hause ist
- Anzeigen der To-do-Liste des aktuellen Tages
- Anzeigen der To-do-Liste für die nächste Woche

## 1.2 Deployment

Die folgenden Kapitel beschreiben kurz das Deployment der beiden Projekte. Dabei sollte beachtet werden, dass zunächst die API deployed und anschließend erst die App installiert werden sollte, wenn die App auf die selbst-deployte Instanz zugreifen soll.

### 1.2.1 API-Deployment

Für die API wurde für ein einfacheres Deployment in ein Dockercontainer eingerichtet. Daher ist es notwendig, dass auf dem verwendeten Rechner zunächst Docker installiert ist. Der Container lässt sich über den Kommandozeilen-Befehl in Listing 1 starten. Dieser sollte dafür innerhalb des Ordners “/ase-cleaning-plan/src/api/” ausgeführt werden.

Listing 1: Startbefehl des Dockercontainers

```
$ docker-compose up
```

Der Befehl startet den Dockercontainer, welcher die Spring-Boot-Anwendung inklusive der benötigten Postgres-Datenbank enthält. Mit der aktuellen Konfiguration kann die API über den Port 8080 angesprochen werden.

### 1.2.2 App-Deployment

Um die Android-App in Verbindung mit der selbstdeployten Instanz der Api zu nutzen muss vor dem Generieren der APK der Endpunkt geändert werden. Dies ist über die Klasse *Configuration* im Package *config* möglich. In dieser Klasse kann wird der Endpunkt, wie beispielsweise die IP-Adresse, in die statische Variable *BASE\_URL* eingetragen. Anschließend kann die App mittels des Android-APK und der Entwicklungsumgebung *Android-Studio* generiert werden. Dies ist über den Menü-Punkt “Build → Build Bundle(s)/APK(s) → Build APK“ möglich. Anschließend ist die generierte APK bei Standardeinstellungen im Ordner “app/build/outputs/apk/debug/“ zu finden und kann auf dem Android-Gerät installiert werden.

## 1.3 Bedienung der Android-App

Beim ersten Start der App muss eine Anmeldung mittels eines Google-Kontos erfolgen. Anschließend hat der/die Nutzer\*in die Möglichkeit einen neuen Haushalt zu erstellen oder mittels eines Beitrittscodes einem bestehenden Haushalt beizutreten.

Nach diesen ersten Schritten gelangt man zur eigentlichen App. Hier kann nun über den Menüpunkt “Haushalt“ im Sidemenu, den Haushalt zu verwalten. Es kann beispielsweise über den Plus-Button ein neuer Task erstellt, bearbeitet, gelöscht oder der eigene Ferienmodus aktiv werden.

Das Bearbeiten und Löschen von Tasks erfolgt über ein längeres Halten des Tasks und die Auswahl des entsprechenden Icons.

Auf unter dem Menüpunkt “To-do-Listen“ können die To-do-Listen vom aktuellen Tag und von der aktuellen Woche eingesehen und die bearbeiteten Tasks abgehakt werden. Dabei ist zu beachten, dass das “Als erledigt makieren“ der Tasks, sowie auch die Umkehrung nicht über einen einfachen Klick passieren, sondern über ein längeres halten des Feldes erfolgt. Diese Art der Bedienung wurde durch ein Interview mit einigen wenigen Probanden festgelegt.

Schließlich besteht unter dem Menüpunkt “Profil“ die Möglichkeit sein Profil einzusehen und einen Dark-Mode zu aktivieren. Dieser ist jedoch temporär für die aktuelle Sitzung. Die Einstellung wird derzeit nicht gespeichert.

## **1.4 Über die Doku**

Die folgende Dokumentation zeigt ob und wie die Inhalte der Advanced Software Engineering Vorlesung innerhalb des Projekts umgesetzt wurden.

Dabei wird auf die Themen “Domain Driven Design“, “Clean Architecture“, “Programming Principles“, “Entwurfsmuster“, “Refactoring“ und “Testing“ eingegangen. Jedes Thema wird zunächst einmal in wenigen Sätzen erläutert, bevor auf die Analyse und die konkrete Umsetzung im Projekt eingegangen wird.

Daraus soll deutlich werden, dass die Inhalte der Vorlesung verstanden und angewendet werden können.

## 2 Domain Driven Design

### 2.1 Ubiquitous Language

Zu Beginn des Projektes wurde zunächst die Ubiquitous Language der Domäne festgelegt. Sie beschreibt die in der Domäne verwendeten Begriffe. Durch die Festlegung dieser Sprache werden Fehler, die auf missverständlichen oder doppeldeutigen Aussagen oder Anforderungen beruhen, vermieden. Die Begriffe ergeben sich bei der Beschreibung des Projektes insbesondere bei der Definition der einzelnen Use Cases.

Die folgende Auflistung enthält die bei der Analyse herausgearbeiteten Begriffe und ihre Definitionen. Da das Programm in englischer Sprache programmiert wird, sind die Begriffe jeweils einmal auf Deutsch und einmal auf Englisch festgelegt. Die Definition ist in der deutschen Sprache verfasst.

- **Deadline/Frist:** Eine Frist ist ein Datum/Zeitpunkt, bis zu dem eine bestimmte Aufgabe erledigt werden muss.
- **Holiday Mode/Ferienmodus:** Wenn der Ferienmodus aktiv bekommt der/die entsprechende Mitbewohner\*in bzw. die entsprechenden Mitbewohner\*innen, die ihn aktiviert haben keine Aufgaben angezeigt. Er wird benutzt, wenn einer oder mehrere der Roommates nicht Zuhause sind.
- **Household/Haushalt:** Ein Haushalt beinhaltet mehrere Mitbewohner\*innen, die für die Abarbeitung verschiedener Aufgaben zuständig sind.
- **Repetitive Task/ Wiederkehrende Aufgabe:** Eine wiederkehrende Aufgabe ist eine Aufgabe, welche in regelmäßigen Zeitintervallen erledigt werden muss.
- **Roommate/Mitbewohner\*in:** Ein/e Mitbewohner\*in ist eine reale Person, die mit anderen Personen/Mitbewohner\*innen in einem Haushalt zusammenlebt.
- **Task/Aufgabe:** Eine Aufgabe ist etwas, was einer der Mitbewohner\*innen bis zu einem bestimmten Zeitpunkt erledigt haben muss.
- **Time Interval/Zeitintervall:** Ein Zeitintervall ist ein Zeitraum, der in Tagen angegeben wird.
- **To-do list/To-Do-Liste:** Eine To-Do-Liste ist eine Menge von Aufgaben, die ein/e Mitbewohner\*in zu erledigen hat.

## 2.2 Verwendete Muster

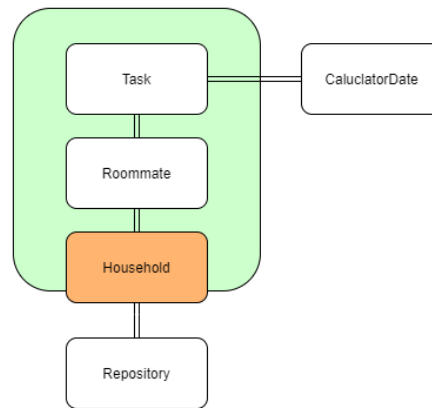


Figure 1: Optimales Domänenmodell

Das taktische Domain Driven Design führt verschiedene Entwurfsmuster ein. Mit Hilfe dieser Entwurfsmuster ist es möglich die komplexen Regeln und Sachverhalte der Domäne in ein verständliches Modell zu bringen. Im den folgenden Abschnitten werden die Entwurfsmuster analysiert und ihre Verwendung im Projekt begründet.

### 2.2.1 Value Objects

Ein Value Object repräsentiert in der Domäne eine Semantik. Das bedeutet, die Instanzen dieser Value Objects sind einzigartig und unveränderlich. Ein Beispiel für ein Value Object ist die Darstellung des Körpergewichts.

Innerhalb der hier vorgestellten Domäne wird ein Value Object eingesetzt. Dies ist das Objekt *Date* (dt.: Datum). Es wird verwendet um die Frist bis zu der eine bestimmte Aufgabe erledigt werden muss, festzulegen, sowie für die Umsetzung des Ferienmodus. Das Objekt *Date* ist ein Value Object, da es seinen Wert nicht ändert. Wird eine Frist geändert, so wird ihr lediglich ein neues Datum zu gewiesen.

Technisch betrachtet wird im Laufe dieses Projektes jedoch kein eigenes Value Object entworfen, da für *Date* bereits ein geeignetes Objekt von der hier verwendeten Programmiersprache Java existiert.

Die Entscheidung keine eigene Klasse zu implementieren begründet sich darauf, dass ein Datum ein sehr komplexes Objekt ist. Dies zeigt u.a. die Entwicklung der unterschiedlichen Klassen, welche für die Darstellung eines Datums in Java über die letzten Jahre implementiert und weiterentwickelt wurden. Java verfügt in verschiedenen Bibliotheken über unterschiedliche Klassen, die alle dazu geeignet sind ein Datum zu

repräsentieren. Beispiele dafür sind das oben erwähnte *Date*, welches in den Bibliotheken *java.util* und *java.sql* zu finden ist, sowie *Calendar* und *LocalDate*. [4] Zudem existieren, wie erwähnt, bereits unterschiedliche Klassen, welche die Einhaltung der komplexen Regeln, denen ein Datum unterliegt, gewährleistet.

### 2.2.2 Entites

Entitäten zeichnen sich dadurch aus, dass sie innerhalb der Domäne eine eigene Identität, veränderliche Eigenschaften, sowie einen Lebenszyklus haben. Sie repräsentieren ein wahrnehmbares Objekt/Ding in der Domäne. Dies trifft in diesem Fall dreimal zu. Das Domänenmodell beinhaltet die *Entitäten*: *Household*, *Roommate* und *Task*. Diese werden im Code durch die gleichnamigen Klassen repräsentiert.

### 2.2.3 Aggregates

Aggregate sind Gruppen von Entitäten und Value Objects, die als eine zusammenhängende Einheit angesehen werden können. Dabei dient eines der Objekte als Wurzel, die die anderen dahinter liegenden Objekte verwaltet. Die anderen Objekte sind dann nur über das Wurzelobjekt erreichbar und es wird beim Zugriff immer das gesamte Aggregat geladen.

Für dieses Projekt wäre es sinnvoll/ideal gewesen, die drei Entitäten in einem Aggregat zusammenzufassen und die Household-Entität als Aggregate Root zu verwenden. Allerdings wurde es praktisch etwas anders umgesetzt. Die optimale Umsetzung ist dem Domänenmodell in Abbildung 1 zu entnehmen.

Innerhalb der Anwendung wurde kein Aggregat aus mehreren Entitäten gebildet, sondern alle drei Entitäten bilden jeweils ein eigenes Aggregat. Diese Entscheidung wurde auf der Grundlage getroffen, dass die Umsetzung von drei getrennten Aggregaten mit dem verwendeten Technologiestack, den vorhandenen Vorkenntnissen mit diesem Technologiestack und dem zeitlichen Rahmen, der für das Projekt vorgesehen ist.

Die Abbildung und die Erläuterung der optimalen Vorgehensweise sind dazu gedacht zu zeigen, dass die Funktionsweise eines Aggregates verstanden wurde, es jedoch bei der praktischen Umsetzung Probleme gegeben hat. Es wurde hier zu Gunsten der Funktionalität der Software entschieden.

### 2.2.4 Repositories

Die Aufgabe der Repositories besteht darin, zwischen der Domäne und dem Datenmodell zu vermitteln. Sie ergeben sich aus der Anzahl der Aggregate. Jedes Aggregat



besitzt eine Wurzel Entität, welche die dahinter liegenden Entitäten verwaltet und vor unbefugtem Zugriff von außen schützt.

Wie bereits im vorangehenden Abschnitt erläutert, ist das Domänenmodell nicht optimal umgesetzt und daher finden sich in der technischen Implementierung drei Repositories anstatt einem. Diese drei Repositories sind den entsprechenden Entitäten zugeordnet und daher auch mit *HouseholdRepository*, *RoommateRepository* und *TaskRepository* benannt.

### 2.2.5 Domain Services

Schließlich beinhaltet das Domänenmodell noch einen Domain Service. Der Service *CalculatorDate* beinhaltet die Funktionalität aus einer Instanzen des Value Objects *Date* und einer Anzahl an Tagen eine neue Instanz zu ermitteln. Das Datum der neuen Instanz liegt dann genau einmal die Anzahl der übergebenen Tage nach dem gegebenen Startdatum.

Dieser Domain Service wird dazu verwendet anhand des Startdatums einer wiederkehrenden Aufgabe und seines Zeitintervalls zu ermitteln, wann die Aufgabe das erste Mal erledigt werden soll.

## 3 Clean Architecture

Robert C. Martin beschreibt in seinem Buch “Clean Architecture - A Craftsman’s Guide to Software Structure and Design“ eine Schichtenarchitektur, welche durch ihre Struktur und ihren Aufbau besonders verständlich und langlebig ist.

Diese “Clean Architecture“ wird in dem hier dokumentierten Projekt verwendet. Dabei stellt jede der Schichten ein eigenes Paket in der Struktur der Spring-Anwendung dar. Dabei wird zwar die korrekte Einhaltung der Abhängigkeiten innerhalb der Anwendung nicht durch den Compiler gewährleistet, aber die Schichten werden sichtbar von einander getrennt (Siehe Abbildung 2).

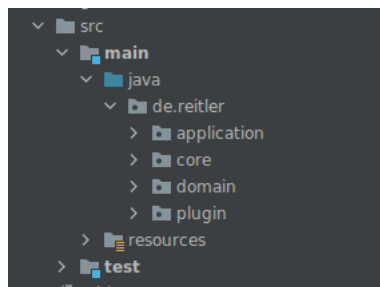


Figure 2: Paketstruktur der API

Die nun folgenden Abschnitte erläutern kurz die Bedeutung der Schicht und wie diese praktisch umgesetzt wurde. Die Schichten werden von innen nach außen betrachtet.

### 3.1 Core

Die innerste Schicht der Architektur ist die Core-Schicht. Diese beinhaltet die mathematischen Grundkonzepte, die unabhängig von der Domäne der Software gültig ist.

In diesem Projekt besteht diese Schicht ausschließlich aus einer Klasse. Die *Date-Calculator*-Klasse hat eine statische Methode. Diese berechnet aus einem Datum und einem Zeitintervall in Tagen ein neues Datum. Diese Berechnung ist allgemein gültig und unabhängig von der Domäne zu betrachten.

### 3.2 Domain

Die Domänenschicht enthält die fachlichen Regeln, die in der Kerndomäne der Anwendung gelten. Diese ergeben sich aus dem in Kapitel 2 beschriebenen Domänenmodell.

Abbildung 3 zeigt, dass sich diese Schicht in drei verschiedene Bereiche einteilen lässt.

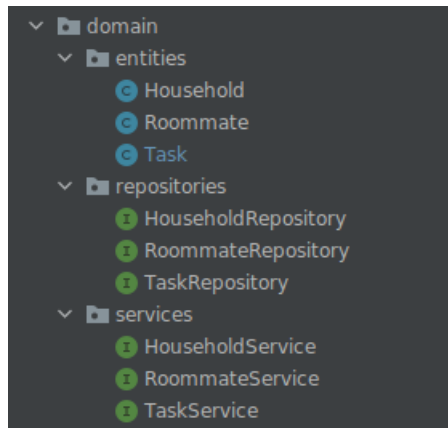


Figure 3: Domänenschicht

Zunächst befinden sich hier die drei Entitäten der Kerndomäne: *Household*, *Roommate* und *Task*. Außerdem befinden sich hier die Interfaces für die Repositories und Services, die für die Persistenz und die Verwaltung der Entitäten benötigt werden.

Für das Speichern der Daten in einer Datenbank wird in diesem Projekt Spring Data JPA verwendet. Spring Data JPA ist ein Teil des Spring Frameworks und ermöglicht es Daten einfach und ohne Boilerplate-Code zu persistieren.

### 3.3 Application

Die letzte Schicht im Kern der Anwendung ist die Applikationsschicht. Im Unterschied zu der Domänenschicht beinhaltet diese Schicht Code, der ausschließlich anwendungsspezifisch ist. Das bedeutet, dass die Regeln, die hier implementiert werden nicht allgemeingültig für die Domäne sind, sondern ausschließlich für die technische Umsetzung der Anwendung benötigt werden.

Auch diese lässt sich wiederum logisch unterteilen. Sie enthält die data-transfer-objects (kurz: dtos), die Händlerklassen und die Implementierungen der Domänenservices.

Die dtos sind pojos (plain old java Objects), die die Struktur für den JSON-Body vorgeben, welcher über die Controller (siehe Abschnitt: Plugin) mittels HTTP an die Android-App übergeben werden.

Die Implementierung der Services stellt die Umsetzung der, in der Domänenschicht liegenden Interfaces, dar. Dies ist nicht direkt domänenspezifisch, aber auch kein technisches Detail. Daher sind diese Klassen in der Applikationsschicht angesiedelt.

Diese Services werden dann von den Händlerklassen verwendet. Die Händlerklassen bereiten die von den Services erhaltenen Daten für die Pluginschicht vor.

### 3.4 Plugin

Die äußerste Klasse in der Architektur ist die Pluginschicht. Diese Schicht enthält die technischen Details der Anwendung, wie z.B. die Schnittstellen, über die die Anwendung angesprochen werden kann. Hier befindet sich keine Businesslogik, da dieser Code sehr kurzlebig ist.

In dem Projekt “cleaning-plan-app“ befinden sich in der Pluginschicht die REST-Controller über die, die Android-App die Spring-API anspricht.

Außerdem beinhaltet die Schicht die Start-Klasse der Spring-Anwendung *ApiApplication*. Diese Klasse befindet sich in dieser Schicht, da sie aufgrund ihres hohen Bezugs zum verwendeten Framework ein technisches Detail darstellt.

## 4 Programming Principles

Programmierprinzipien sind eine Art Leitfaden für die Entwicklung von Software. Diese beschreiben, wie die Verantwortlichkeiten im Programm zu verteilen sind. Dabei sind sie eher allgemein gehalten, was sie von Entwurfsmustern unterscheidet.

In der Vorlesung “Advanced Software Engineering“ wurden drei verschiedene Gruppen dieser Prinzipien unterschieden: SOLID, GRASP und DRY. Diese werden im folgenden erläutert und ihre Verwendung in diesem Projekt analysiert und begründet.

### 4.1 SOLID

“Die *SOLID*-Prinzipien geben vor, wie wir unsere Funktionen und Datenstrukturen in Klassen anordnen und wie diese Klassen miteinander verbunden werden“ [3]

So erklärt Robert C. Martin die Funktionsweise der SOLID-Prinzipien in seinem Buch “Clean Architecture“. SOLID ist ein mnemonisches Akronym, welches 5 Prinzipien unter sich vereint. Im Folgenden werden die Prinzipien erläutert, in Bezug auf das hier betrachtete Projekt analysiert und ihre Verwendung begründet.

#### 4.1.1 Single-Responsibility-Prinzip

Das *Single-Responsibility-Prinzip* (kurz: SRP) besagt, dass jede Klasse nur eine Zuständigkeit haben soll. Daraus folgt, dass es bloß eine Ursache bzw. einen Grund geben darf, der eine Änderung an der Klasse bewirken kann. Dadurch wird die Komplexität der Software, sowie die Kopplung zwischen den Komponenten bzw. Klassen verringert.

Innerhalb des Projekts “cleaning plan app“ ist das SRP besonders gut in den Implementierungen der Services umgesetzt. Jeder der drei Services hat die Aufgabe eines der Repositories zu verwalten und dabei die Einhaltung der Domänenregeln zu gewährleisten. Dadurch ergibt sich für jeden dieser Services lediglich ein Grund zur Änderung. Betrachtet man beispielsweise die Implementierung *HouseholdServiceImpl* (Siehe Abbildung 4). Diese Klasse sollte sich ausschließlich ändern, wenn sich die Domänenregeln für das Verwalten eines Haushalts verändern. Ändern sich die Regeln für die Verwaltung eines einzelnen Nutzers oder eines Tasks, so hat dies keine Auswirkungen auf die Klasse. Auch Erweiterungen der Software, die nicht direkt die *Household*-Klasse und das dazugehörige Repository betreffen, veranlassen keine Änderung der Klasse.

```

@Service
public class HouseholdServiceImpl implements HouseholdService {

    private HouseholdRepository householdRepository;

    private RoommateRepository roommateRepository;

    @Autowired
    public HouseholdServiceImpl(HouseholdRepository householdRepository, RoommateRepository roommateRepository){
        this.householdRepository = householdRepository;
        this.roommateRepository = roommateRepository;
    }

    @Override
    public Household create(Household household){
        householdRepository.save(household);
        return household;
    }

    @Override
    public Household update(Household household){
        Household old = householdRepository.findById(household.getId()).get();
        old.setName(household.getName());

        householdRepository.save(old);
        return old;
    }

    @Override
    public void delete(Household household){
        householdRepository.findById(household.getId()).get().getRoommates()
            .forEach(roommate -> roommate.setHousehold(null));
        householdRepository.delete(household);
    }
}

```

Figure 4: Ausschnitt: HouseholdServiceImpl

#### 4.1.2 Open-Closed-Prinzip

Das *Open-Closed-Prinzip* (kurz: OCP) bezieht sich im Gegensatz zum SRP nicht ausschließlich auf Klassen. Es besagt, dass alle Elemente der Software, also Klassen, Module, Funktionen etc, sollten offen für Erweiterungen und geschlossen für Änderungen sein.

Dieses Prinzip kann besonders gut auf die Entitätsklassen, wie beispielsweise die *Roommate*-Klasse übertragen werden.

Das Prinzip bietet den Vorteil, dass einmal geschriebener und getesteter Code nicht erneut angefasst werden muss. Dadurch wird garantiert, dass korrekt umgesetzte Funktionalitäten, nach der Erweiterung der Anwendung einwandfrei funktionieren. Dies wird häufig durch die Verwendung von Interfaces oder Vererbung umgesetzt.

Die Klasse *Roommate* eignen sich besonders gut als Beispiel für das OCP, da bei Erweiterung des Projekts um etwa einen "Premiumroommate", welcher zusätzliche Funktionen und Einstellungen vornehmen kann, die ursprüngliche Klasse nicht verändert werden müssten. Die Entität "Premiumroommate" kann von *Roommate* erben, so dass alle Funktionalitäten von *Roommate* übernommen werden und ohne Probleme weitere hinzugefügt werden können.

### 4.1.3 Liskov'sche Substitutionsprinzip

Das *Liskov'sche Substitutionsprinzip* (kurz: LSP) schränkt in gewissem Maße die Ableitungsregeln ein. Dabei müssen Objekte eines abgeleiteten Typs als Instanzen ihres Basistyps funktionieren ohne die Korrektheit des Programms zu ändern.

Betrachtet man nach dieser Definition die Umsetzung des hier beschriebenen Projekts, so fällt auf, dass das Projekt keine Vererbung in diesem Sinne beinhaltet. Es werden also keinerlei Unterklassen gebildet. Daraus folgt, dass hier also auch nicht gegen das LSP verstoßen werden kann. Allerdings ist es dadurch auch nicht möglich das LSP anhand eines Beispiels aufzuzeigen.

### 4.1.4 Interface-Segregation-Prinzip

Das *Interface-Segregation-Prinzip* (kurz: ISP) bezieht sich, wie der Name schon vermuten lässt auf Interfaces. Es besagt, dass Interfaces die zu viel Funktionalität in sich bündeln in kleinere Interfaces aufgeteilt werden sollen.

Das hier beschriebene Projekt verwendet sechs Interfaces. Zum einen gibt es drei Interfaces für den Zugriff auf die Datenbank. Diese erben vom Interface *JpaRepository* und sind davon abgesehen leer.

Die anderen Interfaces sind die Service-Interfaces. Sie definieren alle Funktionalitäten, welche innerhalb der Domäne auf der Datenbasis ausgeführt werden können. Da sich die Funktionalitäten der Methoden, welche von den Interfaces vorgegeben werden stark unterscheiden und beispielsweise ein Service, welcher einen *Household* verwaltet keine Methoden für die Erstellung eines *Tasks* benötigt, wurden hier von Beginn an die Methoden auf die drei kleineren Interfaces *HouseholdService*, *RoommateService* und *TaskService* aufgeteilt. Diese Aufteilung verdeutlicht gut das hier verwendete ISP.

### 4.1.5 Dependency-Inversion-Prinzip

Das *Dependency-Inversion-Prinzip* (kurz: DIP) wird in vielen Bereichen der Softwareentwicklung verwendet. Es kann sich genau wie das OCP auf alle Elemente der Software beziehen.

Es besagt, dass High-Level Module nicht von Low-Level Modulen abhängen und Abhängigkeiten nicht auf Objekte, sondern aus Interfaces bestehen sollten.

In diesem Projekt ist die Umsetzung des DIP besonders deutlich in den Handlerklassen und ihren Abhängigkeiten zu den Serviceklassen zu erkennen. Die vier Handlerklassen *HouseholdHandler*, *RoommateHandler*, *TaskHandler* und *HolidayMode*, welche sich alle

```

@Component
public class HolidayModeHandler {

    @Autowired
    RoommateService roommateService;

    @Autowired
    HouseholdService householdService;
}

```

Figure 5: Abhängigkeiten zu Service-Interfaces

in der Applicationschicht befinden, sind abhängig, in den zuvor erwähnten Services. Dabei besteht nie eine Abhängigkeit auf eine konkrete Implementierung, sondern immer nur auf das entsprechende Service-Interface (Siehe Abbildung 5).

Zudem sind die Handler-Klassen aus architektonischer Sicht unterhalb der Service-Interfaces angesiedelt, wodurch die Abhängigkeiten von die Low-Level-Komponente von der High-Level-Komponente abhängig ist.

## 4.2 GRASP

Neben den weitläufig bekannten SOLID-Prinzipien existieren einige weitere Prinzipien, welche unter dem Namen *General Responsibility Assignment Software Patterns*, kurz *GRASP*, zusammengefasst werden. Diese überschneiden sich teilweise mit den zuvor betrachteten SOLID-Prinzipien.

### 4.2.1 Low Coupling

Low Coupling bedeutet, dass zwischen den verschiedenen Komponenten einer Software ausschließlich lose bzw. geringe Kopplungen geben darf. Die Kopplung ist ein Maß für die Abhängigkeit zwischen Objekten. Eine lose Kopplung kann beispielsweise durch die Verwendung von Interfaces umgesetzt werden. Wenn eine Klasse nicht von der konkreten Implementierung, sondern von den entsprechenden Interfaces abhängig.

Ein Beispiel aus diesem Projekt für die Umsetzung einer losen Kopplung mittels Interfaces ist die Abhängigkeit zwischen Serviceklassen und Handlerklassen. Die Klasse *HolidayModeHandler* ist beispielsweise von den beiden Interfaces *HouseholdService* und *RoommateService* abhängig, dadurch ist es leicht möglich die konkrete Implementierung der Services auszutauschen ohne, dass Änderungen an der Handler-Klasse geben muss.



### 4.2.2 High Cohesion

*High Cohesion* oder im deutschen *hohe Kohension* beschreibt die semantische Nähe der Elemente innerhalb einer Klasse. Mit semantischer Nähe ist dabei gemeint, dass alle Klassenvariablen und Methoden sich auf die Erfüllung einer bestimmten Zuständigkeit fokussieren.

Als Beispiel innerhalb dieses Projekts kann hier die erneut die Klasse *RoommateServiceImpl*. Die Klasse beinhaltet die beiden Klassenvariablen *repository* und *taskService*. Beide Variablen sind Objekte, welche für die Verwaltung eines *Roommates* benötigt werden. Zu dem beziehen sich alle Methoden dieser Klasse auf die Verwaltung von *Roommates* und Bereitstellung von Informationen über diese, wie beispielsweise die Methode *getAllTasks*.

Durch die Einhaltung des “high Cohesion“-Prinzips wird sichergestellt, dass die Funktionalitäten innerhalb der Anwendung sinnvoll verteilt werden und der Erstellung von sogenannten *Gottklassen*, welche zu viele verschiedene Funktionalitäten beinhaltet, entgegengewirkt.

### 4.2.3 Information Expert

Ein *Information Expert* ist ein Objekt, welches die Verantwortung für bestimmte Informationen hat. Er ist also allgemein eine Zuweisung einer Zuständigkeit zu einem bestimmten Objekt. Auffällig ist hierbei, die Parallele zwischen diesem Prinzip und dem SRP. Durch diese starke Überschneidung können auch hier als Beispiele die Serviceklassen angebracht werden.

Betrachtet man die Klasse *RoommateServiceImpl*, so fällt auf, dass diese Klasse alle Informationen zu den *Roommate*-Objekten verwaltet. Sie ist für das Speichern, Ändern, Lesen und Löschen der Objekte zuständig.

### 4.2.4 Creator

Das *Creator*-Pattern, wie das *Information Expert*-Pattern Zuständigkeiten fest. Allerdings beschränkt sich bei diesem Pattern der Zuständigkeitsbereich auf die Erzeugung bestimmter Objekte, wie z.B. Objekte der Entitäten. Als Creator eines Objekts kommt allgemein gesagt, dasjenige Objekt in Frage, welches Beziehungen zu allen erstellten Objekten aufweist.

Ein Beispiel für einen Creator in diesem Projekt ist die *RoommateHandler*-Klasse, welche die Creator für die *Roommate*-Objekte. Innerhalb der *RoommateHandler*-Klasse

werden alle Objekte der Entity *Roommate* erzeugt.

#### 4.2.5 Indirection

*Indirection* oder auch Delegation genannt, bezeichnet das Entkoppeln von Systemen oder Teilen von Systemen untereinander. Dadurch entsteht zwischen den beiden Teilsystemen eine lose Kopplung (Siehe auch Kapitel 4.2.1 “Low Coupling“).

Ein Beispiel für eine solche *Indirection*-Klasse ist die Klasse *TaskServiceImpl*. Sie vermittelt zwischen dem *TaskHandler* und dem *TaskRepository*. Dadurch wird das *TaskRepository* vor dem *TaskHandler* verborgen und umgekehrt.

#### 4.2.6 Polymorphism

*Polymorphismus* dient dazu Alternativen eines Objekts abhängig von einem konkreten Typ zu behandeln. Er beruht auf einem der grundlegenden Prinzipien der Objektorientierung, bei dem bestimmte Methoden je nach konkretem Typ eine andere Implementierung haben. Die Methoden des Grundtyps oder der Schnittstelle werden also überschrieben. Durch diese Technik des Überschreibens ist es möglich Konditionalstrukturen, wie If-Else oder Switch-Anweisungen zu vermeiden, da diese eine große Fehlerquelle darstellen.

Im hier beschriebenen Projekt wird das *Polymorphism*-Pattern bei der Umsetzung der Services, wie z.B. dem *TaskService*, angewendet, da für jeden Service ein Interface existiert, welches die benötigten Methoden definiert. Die konkrete Umsetzung dieser Methoden ist dann von der Implementierung dieses Interfaces abhängig. Dieses Projekt beinhaltet ausschließlich eine Implementierung, jedoch ist es durch die Verwendung des Interfaces leicht möglich die konkrete Implementierung ohne Änderung an anderen Klassen durchzuführen, da diese ausschließlich auf das Interface zugreifen.

#### 4.2.7 Controller

Eine Controller-Klasse übernimmt die Verarbeitung von ein kommenden Benutzereingaben und dient der Koordination zwischen Benutzeroberfläche und Businesslogik.

Innerhalb der REST-Applikation befinden sich vier Controller-Klassen. Diese sind die REST-Controller-Klassen, welche sich in der Plugin-Schicht befinden. Sie stellen die Schnittstelle zwischen der REST-Applikation und dem Android-Projekt dar. Daher verwalten und verarbeiten die REST-Controller-Klassen, die im Frontend getätigten Benutzereingaben, wodurch sie die oben genannte Definition erfüllen.

Will man etwas genauer auf Controller eingehen, so kann man Controller in zwei verschiedene Arten von Controllern einteilen: “System Controller“ und “Use Case Controller“.

Ein System Controller verwaltet alle Aktionen eines Programms und ist daher eher für kleinere Anwendungen praktikabel. Ein Use Case Controller verwaltet ausschließlich ein Use Case. Dadurch kommen mehrere kleine Controller zusammen.

Im Projekt “cleaning plan app“ stellen die REST-Controller Use Case Controller dar, da jeder Controller ein eigenes Use Case, wie z.B. die Verwaltung eines *Tasks*, übernimmt.

#### 4.2.8 Pure Fabrication

*Pure Fabrication* bedeutet frei übersetzt soviel wie “reine bzw. völlige Erfindung“. Das Pattern beschreibt eine reine *Verhaltens-* oder *Arbeits-*Klasse, welche keinen Bezug zur Problemdomäne aufweist. Das Muster dient ähnlich, wie die verwendete Architektur der Trennung zwischen der Technologie und der zugrundeliegenden Problemdomäne. Eine *Pure Fabrication*-Klasse kapselt Funktionalitäten, welche auch außerhalb der Domäne leicht wiederverwendbar sind.

Ein Beispiel für eine solche *Pure Fabrication* ist die Klasse *DateCalculator* aus der Core-Schicht. Sie hat keinen konkreten Bezug zur Domäne und stellt eine Methode zur Verfügung, welche problemlos auf andere Domänen übertragen werden kann, ohne dass sie verändert werden muss. Das liegt daran, dass die Klasse keine Abhängigkeiten zu der Domäne hat.

#### 4.2.9 Protected Variations

Das Pattern *Protected Variations*, oder zu deutsch auch “Sicherung vor Variation“, dient dazu den Einfluss den die Variabilität einzelner Komponenten auf das System haben kann durch eine eingezogene Schicht, wie ein Interface, abzusichern. Dadurch wird das Gesamtsystem vor diesen Variationen geschützt, da es ausschließlich die allgemein gültige Schnittstelle kennt und verwendet.

Beispiele aus dem vorgestellten Projekt sind hier erneut die Services. Aber auch die Repository-Interfaces stellen eine solche Schutzschicht dar. Betrachtet man die Interfaces für die Repositories genauer, so wird deutlich, dass sie die konkrete Implementierung der Datenpersistenz vor der Application verstecken, so dass es leicht möglich ist die Persistierung der Daten nachträglich zu ändern ohne dass dies Auswirkungen auf die eigentliche Anwendung hat.

## 4.3 DRY

Das letzte betrachtete Prinzip ist so simple, wie wirkungsvoll. *Don't repeat yourself* oder kurz *DRY* besagt, dass keine Duplikationen im Code geben darf. Doppelter Code birgt immer ein gewisses Risiko, dass Änderungen, welche an einer Stelle vorgenommen werden, an anderen Stellen nicht übernommen werden, obwohl die Logik die dahinter steckt die Gleiche ist. Dies kann dann zu Fehlern führen, die schwer zu finden und leicht zu verhindern sind.

In dem hier betrachteten Projekt ist ein gutes Beispiel für dieses Prinzip in der Klasse *RoommateHandler* zu finden. Dort ist zu erkennen, dass Funktionalitäten, welche an mehreren Stellen innerhalb dieser Klasse benötigt werden, in die beiden Methoden *getStartOfDay* und *getEndOfDay* extrahiert wurden.

Der Vorgang und die Begründung für diese Methodenextraktion werden in Kapitel 6.1 näher erläutert. Dort sind die oben erwähnten Methoden auch in Abbildung 9 abgebildet.

## 5 Entwurfsmuster: Singleton-Pattern

Entwurfsmuster sind Elemente aus leicht wiederverwendbarer objekt-orientierter Software. Sie helfen dabei Probleme zu lösen, in dem sie ein erprobtes Konzept liefern. Dadurch werden komplexe Softwaresysteme, welche solche Entwurfsmuster verwenden, beherrschbarer und wartbarer.

Für die Verwendung in diesem Projekt bot sich besonders das Entwurfsmuster *Singleton* an. Dieses Pattern löst grundsätzlich zwei Probleme.

Zum einen stellt das Entwurfsmuster sicher, dass ausschließlich eine einzige Instanz einer Klasse existiert. Dieses Verhalten ist bei der regulären Verwendung von Konstruktoren unmöglich.

Zum anderen bietet es *einen* globalen Zugangspunkt zu dieser Instanz. Dadurch kann beispielsweise der Zugriff auf geteilte Ressourcen, wie eine Datenbank besser verwaltet werden. [2]

Verwendet wird das Singleton-Pattern innerhalb dieses Projektes für die Repository-Klassen, wie der Klasse *HouseholdRepository*, der Android-App.

Im Folgenden wird die Verwendung dieses Musters in der Klasse *HouseholdRepository* anhand der UML-Diagramme in Abbildung 6 und Abbildung 7 erläutert und begründet. Die UML-Diagramme stellen nicht die vollständige Klasse dar, da viele Methoden für die Verdeutlichung dieses Patterns nicht relevant sind. Die Diagramme zeigen daher nur die für die Elemente, die für die Erstellung bzw. den Zugriff auf Instanzen benötigt werden.

Vor der Anwendung des Singleton wurden Instanzen von *HouseholdRepository* über den öffentlichen Konstruktor erzeugt, wie das Diagramm in Abbildung 6 zeigt.

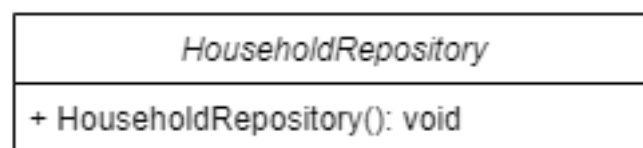


Figure 6: UML-Diagramm (vorher)

Da die Klasse jedoch die globalen Daten der Android-Applikation bzgl. des Haushalts verwaltet, ist es sinnvoll den Zugriff auf diese Klasse einzuschränken, in dem es nicht mehr möglich ist neue Instanzen zu erstellen. Dies wird beim Singleton dadurch umgesetzt, dass der Konstruktor privat ist und die Klasse eine statische private Instanz ihrer

eigenen Klasse enthält. Auf diese Instanz kann dann über die statische öffentliche Methode *getInstance* zugegriffen werden (Vergleiche Abbildung 7).

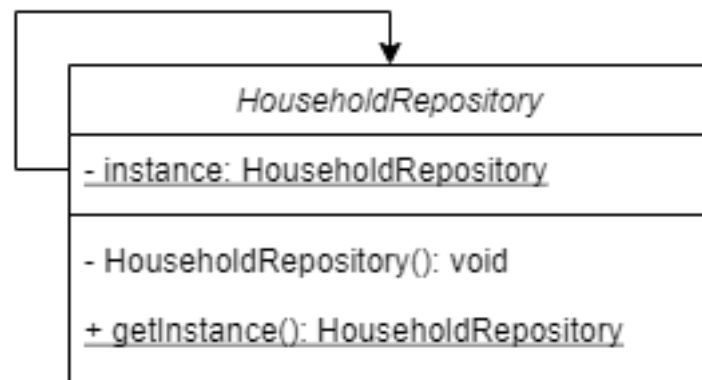


Figure 7: UML-Diagramm (nachher)



*mateHandler* in den Methoden *getAllDailyTasks* und *getAllWeeklyTasks*. Die beiden Methoden filtern die Tasks eines Roommates nach jeweils einer bestimmten Zeitspanne. Zum einen werden die Tasks herausgesucht, die am aktuellen Tag bearbeitet werden sollen. Die andere Methode bestimmt alle Tasks, welche innerhalb der nächsten sieben Tage erledigt werden müssen.

Es ist zu erkennen, dass innerhalb der Methode jeweils ein Startdatum und ein Enddatum ermittelt werden, für die Uhrzeit jeweils auf 00:00:00:000 und 23:59:59:999 gesetzt werden. Dieser Code lässt sich ohne weiteres in eine eigene Methode auslagern. Dieser Vorgang ist eine Refactoring-Methode und wird *Extract Method* genannt. Der Code wird schließlich in die beiden eigenständigen Methoden *getStartOfDay* und *getEndOfDay* ausgelagert. Diese sind in Abbildung 9 zu sehen.

```
private Date getEndOfDay() {
    Calendar endOfWeek = Calendar.getInstance();
    endOfWeek.set(Calendar.HOUR_OF_DAY, 23);
    endOfWeek.set(Calendar.MINUTE, 59);
    endOfWeek.set(Calendar.SECOND, 59);
    endOfWeek.set(Calendar.MILLISECOND, 999);
    return endOfWeek.getTime();
}

private Date getStartOfDay() {
    Calendar startOfDay = Calendar.getInstance();
    startOfDay.set(Calendar.HOUR_OF_DAY, 0);
    startOfDay.set(Calendar.MINUTE, 0);
    startOfDay.set(Calendar.SECOND, 0);
    startOfDay.set(Calendar.MILLISECOND, 0);
    return startOfDay.getTime();
}
```

Figure 9: Extrahierte Methoden

Der gesamte Umfang dieses Refactorings ist im Commit mit der ID “ce6c601“ und der Bezeichnung “Refactoring 1: Duplicate Code mit Extract Method behoben“ dokumentiert.

## 6.2 Refactoring 2: Replace Magic Number with Symbolic Constant

Beim zweiten Refactoring wird kein Code Smell im eigentlichen Sinne behoben, sondern die allgemeine Lesbarkeit des Codes wird verbessert. Sogenannte *Magic Numbers* (engl.: “Magische Zahlen“) sind Zahlen, welche ohne Erklärung oder direkt ersichtlichen Kontext im Quellcode auftauchen. Um diese Zahlen in einen sinnvollen Kontext zu setzen und Code dadurch für Menschen besser lesbar zu machen, sollten solche *Magic Numbers* durch Konstanten mit aussagekräftigen Namen ersetzt werden. Diese Refactoring-Technik wird auch “Replace Magic Number with Symbolic Constant“ genannt. [1]

Im Projekt “ase-cleaning-plan“ wurden mehrere dieser magischen Zahlen identifiziert. Einige Beispiele aus der Klasse *RoommateHandler* sind in Abbildung 10 markiert. Diese sollen nun mit der zuvor beschriebenen Refactoring-Technik in symbolische Konstanten



```

public List<TaskDTO> getAllWeeklyTasks(String id) {
    if(roommateService.getById(id).getHolidayMode()==null||HolidayMode
    {
        return new ArrayList<TaskDTO>();
    }
    List<Task> weeklyTasks = roommateService.getAllTasks(id);

    DateCalculator calculator = new DateCalculator();
    Date endOfWeek = calculator.add(getEndOfDay(), timeInterval: 7);
    return weeklyTasks
        .stream()
        .filter(x -> x.getDeadline().compareTo(endOfWeek) < 0)
        .filter(x->x.getDeadline().compareTo(getStartOfDay())>0)
        .map(x -> new TaskDTO(x.getId().toString(), x.getTitle(), x
        .collect(Collectors.toList());
}

private Date getEndOfDay() {
    Calendar endOfWeek = Calendar.getInstance();
    endOfWeek.set(Calendar.HOUR_OF_DAY, 23);
    endOfWeek.set(Calendar.MINUTE, 59);
    endOfWeek.set(Calendar.SECOND, 59);
    endOfWeek.set(Calendar.MILLISECOND, 999);
    return endOfWeek.getTime();
}

private Date getStartOfDay() {
    Calendar startOfDay = Calendar.getInstance();
    startOfDay.set(Calendar.HOUR_OF_DAY, 0);
    startOfDay.set(Calendar.MINUTE, 0);
    startOfDay.set(Calendar.SECOND, 0);
    startOfDay.set(Calendar.MILLISECOND, 0);
    return startOfDay.getTime();
}

```

Figure 10: “Magische Zahlen“

umgewandelt werden. Dabei wird beispielsweise aus der “7“ in der Methode *getAllWeeklyTasks* die Konstante *WEEK\_DAYS\_COUNT*. Weitere Änderungen durch dieses Refactoring sind in Abbildung 11 visualisiert und der gesamte Umfang des Refactorings ist dem Commit mit der ID “c0689c7“ und der Bezeichnung “Refactoring 2: Replace magic Numbers with symbolic Constants“ zu entnehmen.

```

private final int WEEK_DAYS_COUNT = 7;

private final int FIRST_HOUR_OF_DAY = 0;
private final int FIRST_MINUTE_OF_DAY = 0;
private final int FIRST_SECOND_OF_DAY = 0;
private final int FIRST_MILLISECOND_OF_DAY = 0;

private final int LAST_HOUR_OF_DAY = 23;
private final int LAST_MINUTE_OF_DAY = 59;
private final int LAST_SECOND_OF_DAY = 59;
private final int LAST_MILLISECOND_OF_DAY = 999;

```

Figure 11: Konstanten

Zusätzlich zu dem genannten Vorteil der Lesbarkeit wird durch dieses Refactoring die Wartbarkeit des Codes verbessert, da Änderungen, die diese Konstante betreffen ausschließlich an einer Stelle im Code geändert werden müssen.

## 7 Testing

Bei der Entwicklung von Software entstehen Fehler. Um diesen entgegen zu wirken oder sie gar nicht erst entstehen zu lassen, sollte Software getestet werden. Es gibt verschiedene Arten von Tests, die manuell bis voll automatisiert sein können. Beispiele für Tests, wie sie etwa aus dem Qualitätsmanagement bekannt sind, sind “Smoking Tests“, “Integration Tests“ und “Unit Tests“.

In dem hier dokumentierten Projekt wurden für die Sicherstellung der Korrektheit der Software *Unit Tests* verwendet. Diese testen einzelne Komponenten unabhängig von anderen Komponenten, welche gegebenenfalls mit der zu testenden Komponente interagieren. Bei objektorientierter Programmierung ist eine solche Komponente zumeist eine Klasse.

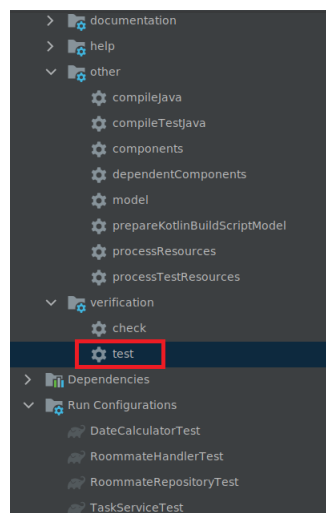


Figure 12: Gradle Tasks

Damit die Komponenten losgelöst von diesen Objekten getestet werden können, werden sogenannte *Mockobjekte* verwendet. Ein Mockobjekt, oder kurz Mock, dient dazu eine künstliche Instanz der Klasse zu erzeugen, welche die Funktionalitäten des originalen Objektes imitiert. Für das Mocken von Objekten, wie z.B. den Repositories wird hier das *Mockito*-Framework verwendet. Dies ist beispielsweise in der Testklasse *RoommateRepositoryTest* zu sehen.

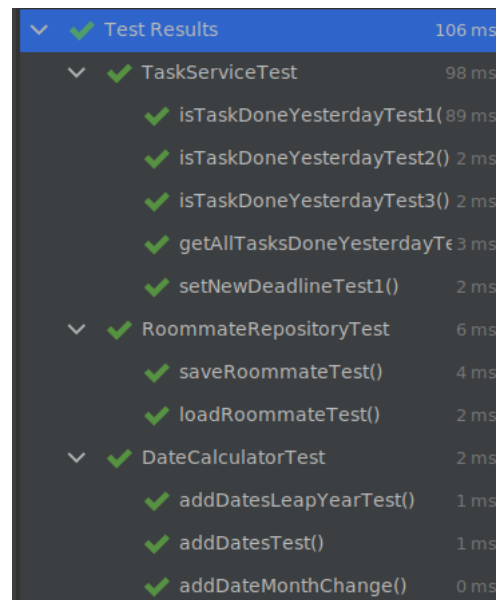
Durch die Verwendung des *Gradle Build Tools* ist es möglich alle Testklassen mit nur einem Befehl bzw. einem Knopfdruck zu starten. In dieser Dokumentation wird ausschließlich der Start der Tests über eine *IDE* erläutert. Hier wird die IDE *IntelliJ Idea Community* verwendet.

Zunächst sollte das Projekt erfolgreich als Gradle-Projekt in die entsprechende IDE importiert worden sein. Ist dies der Fall bringen die meisten IDEs integriert oder als frei erhältliches Plugin die Möglichkeit mit sogenannte Gradle-Tasks direkt über einen Button oder ein Menü zu starten. Aus der Auswahl der unterschiedlichen Gradle-Tasks wählt man dann den Task mit der Bezeichnung “test“. In IntelliJ ist dieser unter dem “verification“-Ordner zu finden (Siehe Abbildung 12).

nach dem der Task gestartet wurde, sollten in einem Fenster die Ergebnisse zu sehen sein. Diese sollten etwa so aussehen, wie in Abbildung 13.

Die Unit Tests befinden sich in den Klassen:

- RoommateRepositoryTest
- TaskServiceTest und
- DateCalculatorTest



✓ Test Results	106 ms
✓ TaskServiceTest	98 ms
✓ isTaskDoneYesterdayTest1()	89 ms
✓ isTaskDoneYesterdayTest2()	2 ms
✓ isTaskDoneYesterdayTest3()	2 ms
✓ getAllTasksDoneYesterdayTest()	3 ms
✓ setNewDeadlineTest1()	2 ms
✓ RoommateRepositoryTest	6 ms
✓ saveRoommateTest()	4 ms
✓ loadRoommateTest()	2 ms
✓ DateCalculatorTest	2 ms
✓ addDatesLeapYearTest()	1 ms
✓ addDatesTest()	1 ms
✓ addDateMonthChange()	0 ms

Figure 13: Testergebnisse

## References

- [1] Refactoring Guru. *Replace Magic Number with Symbolic Constant*. URL: <https://refactoring.guru/replace-magic-number-with-symbolic-constant>.
- [2] Refactoring Guru. *Singleton*. URL: <https://refactoring.guru/design-patterns/singleton>.
- [3] Robert C. Martin. *Clean Architecture*. mitp, 2018. ISBN: 978-3-95845-724-9.
- [4] Pankaj. *Java 8 Date – LocalDate, LocalDateTime, Instant*. URL: <https://www.journaldev.com/2800/java-8-date-localdate-localdatetime-instant>.