Hausmeister-Service Reinigungsobjekt-Verwaltung

Programmentwurf

der Vorlesung „**A**dvanced **S**oftware **E**ngineering“

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

David Huh

Abgabedatum 1. Mai 2022

Matrikelnummer	-
Kurs	TINF19B4
Bearbeitungszeitrum	5. & 6. Semester
Gutachter der Studienakademie	Mirko Dostmann

Inhaltsverzeichnis

1. Setup

1.1 Aufsetzen des Projektes

Folgende Schritte müssen ausgeführt werden, um das Projekt aufzusetzen.

- IntelliJ IDEA starten
- Projekt von GitHub clonen und in IntelliJ öffnen (https://github.com/Huh-David/JanitorManager)
- Unter View > Tool Windows > Gradle aktivieren
- Im Gradle Fenster unter Tasks > compose desktop > run ausführen, um Programm zu starten.
- Im Gradle Fenster unter Tasks > verification > test ausführen, um Tests zu starten.

1.2 Aussehen bei Programmstart

Das Programm sollte nach Programmstart wie folgt (Abbildung 1.1) aussehen. Links ist eine Navigationsleiste. Die einzelnen Reinigungsobjekte sind auf dem Startscreen zu sehen. Wenn man auf die Reinigungsobjekte klickt, kommen weitere Informationen dazu. Aktivitäten zu den Reinigungsobjekten können in Form einer To-do-Liste abgehakt werden. Klickt man in der Navigationsleiste auf alle To-dos, werden alle To-dos unabhängig vom Reinigungsobjekt in chronologischer Reihenfolge angezeigt. Im letzten Abschnitt der Navigationsleiste kann das aktuell ausgewählte Reinigungsobjekt angepasst werden.

1.3 Aufbau des Projektcodes

Der Code ist wie im folgenden Bild (Abbildung 1.2) aufgebaut. Die drei realisierten Clean Architecture Layers sind in Form von Packages implementiert worden.
Sowohl die Domänenlogik als auch die dazugehörigen Tests befinden sicher unter /src.

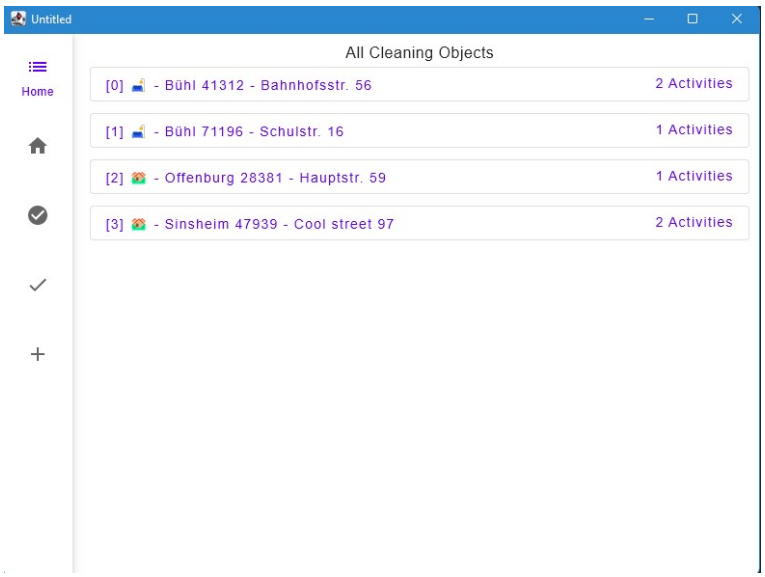


Abbildung 1.1: Aussehen bei Programmstart

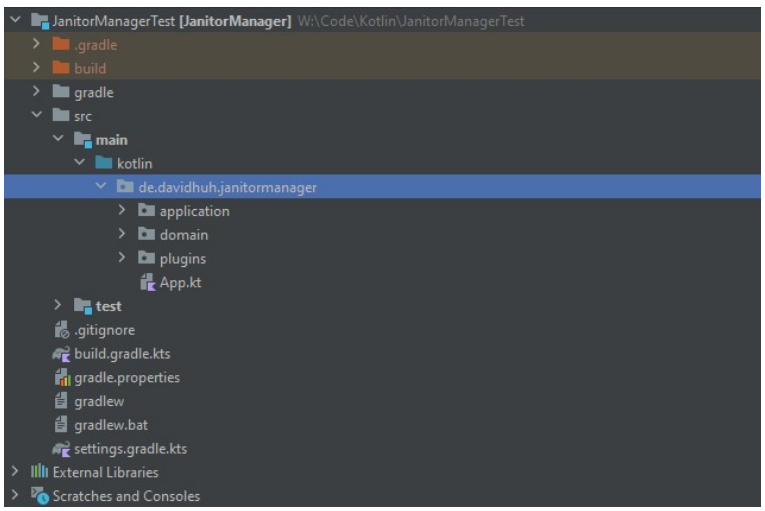


Abbildung 1.2: Aufbau des Projektcodes

2. Domain Driven Design

2.1 Analyse der Ubiquitous Language

ODer Begriff des Reinigungsobjektes wird häufig zu **Objekte** abgekürzt. Objekte sind auch in der Software-Entwicklung ein gängiger Begriff, wobei in folgendem Dokument das Objekt im Sinne eines Reinigungsobjekt interpretiert wird. Ein Reinigungsobjekt beinhaltet eine Adresse, Hausverwaltung, eine Liste von Aktivitäten (ActivityAggregates) und eine Eigenschaft, ob es sich um ein Haus oder eine Wohnung handelt.

2.2 Analyse und Begründung der verwendeten Muster

2.2.1 Value Objects

Value Objects wurden für Adressen, Zeitangaben, Flächenberechnung und Kosten verwendet. Bei diesen Objekten gilt nämlich, dass sie gleich sind, sobald alle Attribute gleich sind, was wiederum der Definition von Value Objects entspricht.

2.2.2 Entities

Entities wurden für Reinigungsobjekte, Angestellte, Fahrzeug verwendet.
Ein Beispiel zu Fahrzeugen:

Es können mehrere Volkswagen Transporter im Fuhrpark sein. Diese können in Erstanmeldung, Kilometerstand, etc. identisch sein, aber sind nicht die gleichen Fahrzeuge. Hierzu wird dann zusätzlich eine Fahrzeug-ID gespeichert.

2.2.3 Aggregates

Aggregates wurden für Aktivitäten verwendet, da Aktivitäten in verschiedenen Zeitintervallen ausgeführt werden, aber dennoch in einer Liste angezeigt werden müssen.
Ein Beispiel hierzu:

Aktivität 1: Treppenhaus reinigen, jeden Montag
Aktivität 2: Treppenhaus reinigen, jeden Freitag

Diese beiden Aktivitäten beinhalten die gleiche Tätigkeit, aber zwei Zeitfenster, die nicht in einem Intervall dargestellt werden könnten. Hierzu wird ein ActivityAggregate angelegt, welches beide Aktivitäten vereint.

2.2.4 Repositories

Repositories wurden für To-dos verwendet, da häufig verschiedene Mengen von To-dos angefordert werden müssen. Beispielsweise müssen alle nicht-erledigten To-dos zu allen Aktivitäten auf einer Übersicht angezeigt werden können. Andererseits muss zu jeder Aktivität auch das letzte To-do, welches noch nicht erledigt wurde, angezeigt werden können.

2.2.5 Domain Services

Domain Services wurden verwendet, um Entities und co. zu persistieren. Angelegte Reinigungsobjekte sollen nicht verloren gehen und erledigte To-dos sollten auch weiterhin abgehakt bleiben. Hierzu wird jeweils eine Service-Klasse angelegt, die die Daten als JSON abspeichern und von der Festplatte wieder lesen kann.

3. Clean Architecture

3.1 Schichtenarchitektur

3.1.1 Plugins

Die äußerste Schicht im Clean Architecture Modell besteht aus Frameworks und Treibern, um beispielsweise Daten aus einer Datenbank oder direkt von der Festplatte zu lesen oder Informationen aus der Anwendungen auch als User-Interface anzeigen zu können. Die Schicht kann jederzeit ausgetauscht werden, sofern nötig, da sie nur auf innere Schichten zugreift und nicht von innen aufgerufen wird. Somit können User-Interface und die Datenbank flexibel ersetzt werden. Sofern später mal eine Datenbank zur Persistierung eingesetzt werden soll, kann dies einfach in der Plugins-Layer umgesetzt werden, ohne dabei irgendwelche anderen Schichten zu verändern.

3.1.2 Adapter

Die dritte Schicht im Clean Architecture Modell besteht aus Interface-Adaptern. Diese ist in meinem Fall leer, da sie nicht notwendig ist.

3.1.3 Application

Die zweite Schicht im Clean Architecture Modell wird dafür verwendet, um die Geschäftsregeln der Anwendung darzustellen. Die Logik und Use-Cases für die Entitäten werden hierbei also repräsentiert. Dies wird hierbei von der Domain-Schicht getrennt, damit die Use-Cases bei Bedarf besser angepasst werden können.

3.1.4 Domain

Die innerste Schicht im Clean Architecture Modell wird dafür verwendet, um die Geschäftsregeln des Unternehmens darzustellen. Zu diesen zählen die Entität-Klassen wie CleaningObject, Activity, Todo, etc. Diese Klassen sind fundamental für die Anwendungen und sind deswegen relativ allgemein gehalten, um Änderungen im Kern der Anwendungen zu vermeiden. Innerhalb dieser Schicht gibt es keinerlei Abhängigkeiten zu Schichten außerhalb, da ansonsten äußere Schichten nicht nach Belieben ausgetauscht werden könnten. Zusätzlich wurden Interfaces für Repository-Klassen angelegt, die von außen belegt werden können.

4. Programming Principles

4.1 SOLID

4.1.1 Single Responsibility

Jede Methode und Funktion verfügt nur über eine Verantwortlichkeit. Das hilft bei Änderungen im Backend. Angenommen es müsste ein Bug gefixt werden, kann der Code effizient von Menschen durchsucht werden, da die Methoden jeweils nur eine Verantwortlichkeit besitzen und danach benannt worden sind.

4.1.2 Open-Closed

Entitäten können problemlos erweitert, aber nicht modifiziert werden. Somit können neue Funktionalitäten implementiert werden, ohne die bestehende Anwendung zu stören. Bereits implementierte Attribute oder Methoden jedoch können nicht ohne weiteres verändert werden, da von außen auf diese zugegriffen wird.

4.1.3 Liskov Substitution

Superklassen können jederzeit durch Subklassen ersetzt werden, ohne dass sich die Korrektheit des Codes ändert. Solch eine Vererbung wurde beim Klassenmodell des Angestellten (**Employee**) angewandt, da es mehrere Arten von Angestellten geben kann, die unterschiedliche Funktionen im weiteren Verlauf ausführen können sollen. Die abstrakte Klasse des Employees dient hierbei als Superklasse und wird an Klassen wie Gärtner oder Reinigungsfachkraft vererbt.

4.1.4 Interface Segregation

Große Interfaces werden in kleinere Interfaces aufgeteilt, da somit leichter Anpassungen durchgeführt werden können. So besteht die Domain Layer beispielsweise getrennt aus Entities und Aggregates, damit man nicht ein Repository Interface über alle Entities und Aggregates auf einmal implementieren muss.

4.1.5 Dependency Inversion

Abhängigkeiten sind stets von konkreteren Modulen niedriger Ebenen zu abstrakten Modulen höherer Ebenen gerichtet. Dies wurde für jede Schicht der Clean Architecture umgesetzt, da

somit Änderungen in äußeren Schichten sehr einfach umsetzbar sind, während der Kern sehr abstrakt und unverändert bestehen bleibt.

4.2 GRASP

4.2.1 Niedrige Kopplung

Es herrscht eine niedrige Abhängigkeit (Kopplung) im Code. So kann zum Beispiel die Implementierung der Persistenzlogik in der Pluginsschicht angepasst werden, ohne dabei andere Geschäftslogik anpassen zu müssen. Das heißt, dass man jederzeit die Speicherung in eine beliebige Datenbank abändern kann. Dazu muss die Implementierung lediglich in der Pluginsschicht angepasst werden.

```
class CleaningObjectPersistenceRepo() : CleaningObjectRepo {
    [...]
    override fun saveCleaningObjectList(cleaningObjectList: MutableList<CleaningObject>) {
        makeDirectories(HelperService.getDirPath())

        val jsonText = Json.encodeToString(cleaningObjectList)
        File(HelperService.getCleaningObjectFilePath()).writeText(jsonText)
    }
    [...]
}
```

4.2.2 Hohe Kohäsion

Es herrscht eine hohe Verantwortlichkeit (Kohäsion) im Code. Beim ActivityAggregateService ist zum Beispiel der Funktion »getFirstTodo« nur bekannt, dass es das erste Todo, welches noch nicht erledigt wurde, aus einer Liste von Todos suchen muss. Ansonsten kümmert sich die Funktion um nichts anderes.

```
class ActivityAggregateService(
    val activityAggregate: ActivityAggregate,
    val cleaningObject: CleaningObject,
) {
    [...]
    fun getFirstTodo(todoList: MutableList<Todo>): Todo {
     val tempTodo = todoList.first() { !it.isOverdue() }
     todoList.removeAll { !it.isOverdue() }
     todoList.add(tempTodo)

     return if (todoList.none() { !it.isDone() }) {
            todoList.last()
     } else {
            todoList.first() { !it.isDone() }
     }
    }
```

```
    [...]
}
```

4.2.3 Polymorphismus

Polymorphismus wurde verwendet, um verschiedene Arten von Angestellten darzustellen und dennoch gleich behandeln zu können. Beispiele hierzu sind Gärtner oder Reinigungsfachkräfte, die von der Klasse »Employee« erben.

4.2.4 Informationen Experte

Komplexere Geschäftslogik wurde stets der Klasse zugewiesen, die bereits am meisten Informationen zum jeweiligen Thema besitzt, damit bereits vorhandene Informationen effizienter weiterverarbeitet werden können. Möchte man sich zum Beispiel alle Angestellten ausgeben lassen, die einer bestimmten Aufgabe zugewiesen wurden, dann nutzt man dazu statt der Employee-Klasse den EmployeeService.

4.2.5 Creator

Das Creator-Prinzip wurde verwendet, um beispielsweise mithilfe einer Activity die dazugehörigen Todos zu generieren. So ist ausschließlich die Activity-Klasse für die Erzeugung von Todos notwendig.

4.2.6 Controller

Der NavigationController ermöglicht die Navigation zwischen den einzelnen Views. So kann man über die Navigationsleiste auf der linken Seite des GUI zum Beispiel zum View von allen ausstehenden To-dos kommen. Dabei nimmt der NavigationController als erste Schnittstelle nach der GUI die Requests entgegen und delegiert das an den verantwortlichen View weiter.

4.2.7 Indirection

Indirection wurde im Projekt nicht verwendet, da es keine Vermittler gibt.

4.2.8 Pure Fabrication

Um die erstellten und bearbeiteten CleaningObjects persistieren zu können, wurden Persistenz-Klassen in Form eines Repos eingeführt, sodass die Kohäsion erhöht wird.

4.2.9 Protected Variation

Das Speichern der CleaningObjects und Todos wurde zunächst durch das direkte Schreiben der serialisierten Daten auf die Festplatte realisiert. Dadurch, dass die Persistenzfunktionen über die Pluginsschicht implementiert wird, können diese Funktionen jederzeit ausgetauscht werden, sodass die Daten zum Beispiel stattdessen in einer Datenbank gespeichert werden können.

4.3 DRY

Don't repeat yourself bedeutet, dass man sich nicht wiederholt. So wird der Code wiederverwendbar. Hierbei wurde beispielsweise die Activity so ausgelagert, dass die Aktivität sowohl in Form eines Aggregates genutzt werden kann, aber auch einzeln innerhalb des TodoServices, um alle anstehenden To-dos zu einer Activity zu generieren. So können Änderungen zentral an der Activity-Klasse durchgeführt werden, ohne dass zusätzlich noch die ActivityAggregate-Klasse verändert werden muss.

```
class TodoService(var activity: Activity) {
    [...]
    fun getTodos(): MutableList<Todo> {
        val todoList = activity.todoList
        todoList.sortBy { it.date }
        return todoList
    }
    [...]
}
```

5. Refactoring

5.1 Identifizieren von Codesmells

5.1.1 Code Smell 1

»Duplicated code: identical or very similar code that exists in more than one location.«
»Shotgun surgery: a single change that needs to be applied to multiple classes at the same time.«
https://github.com/Huh-David/JanitorManager/commit/24f955ec2602df8e91f38bd7fe4b3f29f35eda95

Begründung

In diesem Refactoring wurde die Pfadgenerierung für die Persistierung der Daten vereinheitlicht. Vorher wurde der Code mehrmals geschrieben und musste bei Änderungen auch mehrmals angepasst werden. Das könnte zum Beispiel zu Problemen führen, wenn man den Pfad nur an einer Stelle anpasst und an der anderen nicht.

Fix

Es wurde eine Funktion eingefügt, welche den Pfad passend für das jeweilige cleaningObject und activityType generiert. Somit wird sichergestellt, dass der Pfad zentral angepasst werden kann.

```
private fun getFormattedFilePath(
    activityType: ActivityType,
    cleaningObject: CleaningObject
): String {
    val activityTypeString = activityType.toString()
        .replace(" ", "")
        .replace(".", "")
    val cleaningObjectString = cleaningObject.toSortString()
        .replace(" ", "")
        .replace(".", "")

    return "$TODOSFILEPATH-$cleaningObjectString-$activityTypeString.json"
}
```

5.1.2 Code Smell 2

»Duplicated code: identical or very similar code that exists in more than one location.«
»Long method: a method, function, or procedure that has grown too large.«
https://github.com/Huh-David/JanitorManager/commit/6462209cde7d6362f086c04656ea9838c99abf9f

Begründung

Damit die Anwendung bei erstmaligem Aufruf nicht komplett leer ist, wurde eine Generierung von Demo-Daten eingebaut, um das Prinzip der Anwendung dennoch nachvollziehen zu können. Dies ist zwar kein reeller Use-Case in der Endanwendung, kann dennoch von Code Smells befreit werden. Das Problem bei der Generierung war, dass einzelne Fragmente der gesamten Methode häufiger gebraucht wurden. Zudem wurde es unübersichtlich, welche Generierung wovon wo geschieht.

Fix

Hierzu wurde die große Methode in viele kleinere aufgesplittet. Die kleinen Methoden haben dabei sprechende Namen erhalten, wodurch die Übersichtlichkeit drastisch erhöht wird.

6. Entwurfsmuster

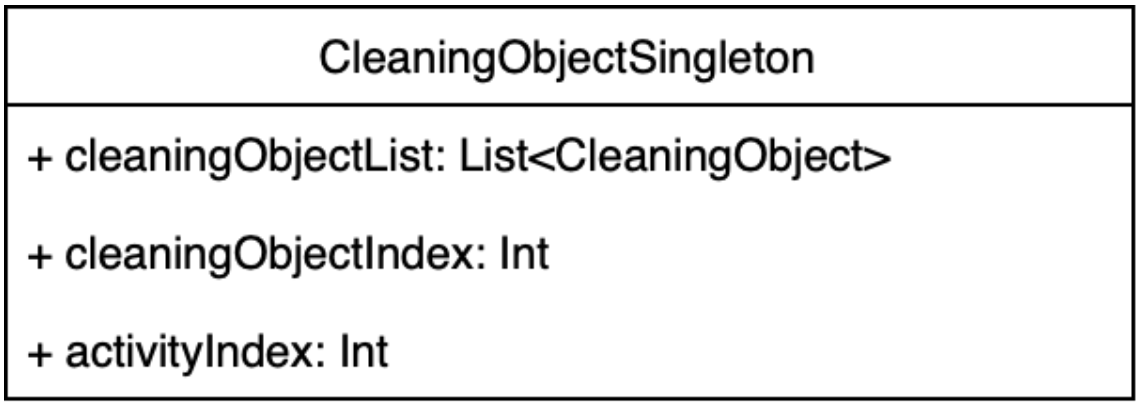
6.1 Begründung des Einsatzes

Um über jede View hinweg die Daten zum jeweiligen CleaningObject beizubehalten, wird ein Singleton Objekt erstellt, welches die Daten zum Start des Programms einmal abruft und von dort an behält. Das heißt, dass nicht zu jeder View eine Datenabfrage von der Festplatte abläuft. Das Singleton-Objekt wird über den NavController, der für die Navigation zuständig ist, übertragen.

6.2 Ohne Singleton

Ohne das Singleton-Objekt musste bei jedem Navigationsschritt, alles neu von der Festplatte geladen werden. Darunter fallen alle Reinigungsobjekte und deren Aktivitäten. Das sorgt natürlich automatisch für eine schlechtere Performance.

6.3 Mit Singleton



Mit dem Singleton-Objekt wird nur einmal zum Start alles aus dem Speicher gelesen und in das Singleton-Objekt gespeichert. Da der NavController für die Navigation zuständig ist, ist dieser über jede View hinweg vorhanden. Dieser trägt nun zusätzlich noch das Singleton-Objekt in sich.