SICK
Sensor Intelligence.

DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

# Advanced Software-Engineering Programmentwurf

# Finanzmanager

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik / Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Niklas Rodenbüsch**

| | |
|---|---|
| Bearbeitungszeitraum | 5. & 6. Semester |
| Matrikelnummer | 6130555 |
| Kurs | TINF21B4 |
| Gutachter der Studienakademie | Mirko Dostmann |

Reposiory: `https://github.com/NI-RO/DH-Software-Engineering-II`

# Inhaltsverzeichnis

**Inhaltsverzeichnis** **I**

**1 Domain Driven Design** **1**

1.1 Analyse der Ubiquitous Language . . . . . 1

1.2 Analyse und Begründung der verwendeten Muster . . . . . 3

**2 Clean Architecture** **5**

**3 Programming Principles** **6**

**4 Refactoring** **7**

**5 Entwurfsmuster** **8**

**Liste der Quellcodes** **II**

# Kapitel 1

# Domain Driven Design

## 1.1 Analyse der Ubiquitous Language

Die Ubiquitous Language des Projektes richtet sich nach der üblichen Sprache der Domäne „Finanzsektor“. Die im Folgenden erläuterten Begriffe sind speziell aus der Subdomäne „Finanzverwaltung“ entnommen und werden im weiteren Verlauf des Projektes verwendet.

**„Konto“ (Account):** Der Begriff bezeichnet eine Einheit, in der finanzielle Transaktionen abgewickelt werden können. Es gibt zwei verschiedene Arten von Konten: Bankkonten und Investmentkonten.

**„Bankkonto“ (Bank account):** Ein Bankkonto ist ein Konto, das für alltägliche Finanztransaktionen wie Einnahmen und Ausgaben genutzt wird. Es gehört zu einer bestimmten Bank und hat einen eindeutigen Namen.

**„Investmentkonto“ (Investment account):** Ein Investmentkonto ist ein Konto, das für den Kauf und Verkauf von Vermögenswerten genutzt wird. Es gehört zu einem bestimmten Broker und hat ebenfalls einen eindeutigen Namen.

**„Transaktion“ (Transaction):** Transaktionen sind Ereignisse, die eine Veränderung des Kontostandes verursachen. Dazu gehören Einnahmen, Ausgaben, sowie Käufe und Verkäufe von Vermögenswerten. Transaktionen gehören immer zu genau einem Konto.

**„Vermögenswert“ (Asset):** Ein Vermögenswert ist eine Ressource, die einen monetären Wert besitzt und von Benutzern gehandelt werden kann, mit der Absicht, finanzielle Gewinne oder langfristige Wertsteigerungen zu erzielen. Vermögenswerte können verschiedene Formen wie Aktien, Fonds, Rohstoffe oder Kryptowährungen annehmen.

**„Einnahme“ (Income):** Einnahmen sind finanzielle Zuflüsse auf ein Bankkonto, die den Kontostand erhöhen.

**„Ausgabe“ (Expense):** Ausgaben sind finanzielle Abflüsse von einem Bankkonto, die den Kontostand verringern.

**„Kauf/Verkauf von Vermögenswerten“ (Purchase/Sale of Assets):** Diese Form von Transaktionen bezieht sich auf den Kauf, bzw. Verkauf von Vermögenswerten. Sie können nur auf einem Investmentkonto angelegt werden.

**„Institution“:** Eine Institution im Kontext dieser Applikation ist eine finanzielle Institution. Es gibt zwei verschiedene Arten von Institutionen: Banken und Broker.

**„Broker“:** Ein Broker ist eine Organisation, die als Vermittler für den Kauf und Verkauf von Vermögenswerten fungiert.

**„Bank“:** Eine Bank ist eine finanzielle Institution, die Dienstleistungen wie z.B. die Verwaltung von Bankkonten, Kreditvergaben, usw. anbietet.

**„Kontostand“ (Balance):** Der Kontostand ist der aktuelle Geldbetrag, der auf einem Konto verfügbar ist. Er errechnet sich aus der Summe aller durchgeführten Transaktionen und kann sowohl positive als auch negative Zahlen annehmen. Im Rahmen der Applikation wird nur der initiale Kontostand, also der Kontostand, der während dem Anlegen des Kontos schon zur Verfügung stand, in der Datenbank gespeichert.

**„Benutzer“ (User):** Der Benutzer der Anwendung, der die Konten erstellt und Transaktionen durchführt.

## 1.2 Analyse und Begründung der verwendeten Muster

### Value Objects

Der aus Vor- und Nachnamen bestehende Name eines Kontoinhabers (Account Owner) ist als Value Object mit dem Namen „AccountOwnerNameValue“ implementiert, da es sich hierbei um ein Objekt ohne Lebenszyklus handelt.

### Entities

Das Objekt „Transaktion“ der Anwendung ist als Entity implementiert. Jede Transaktion lässt sich eindeutig durch eine ID identifizieren und eindeutig genau einem Account zuordnen. Zudem haben Transaktionen einen Lebenszyklus und können sich während iherer Lebenszeit verändern.

### Aggregates

Die Objekte „Institution“ und „Account“ der Anwendung sind jeweils als Aggregates implementiert. Die Institution ist hierbei das Aggregate Root. Jede Institution besitz beliebig viele Accounts und jeder dieser Accounts kann wiederum beliebig viele Transaktionen beinhalten. Transaktionen werden somit über einen Account verwaltet und Accounts über eine Institution.

### Repositories

Repositories sind für alle genannten Aggregates und Entities implementiert. Da die Institution das Aggregate Root darstellt, ist das Repository dieser Klasse das einzige, welches über die Methoden „save“ und „delete“ verfügt. Alle Repositories verfügen über mehrere Methoden zum Finden der Objekte (Institution: findByName, Account: findByInstitutionAndId, Transaktion: findByInstitutionAndAccountAndId).

### Domain Services

Die Anwendung besitzt im Kontext der bei der Themeneinreichung formulierten Use-Cases nur sehr wenig Geschäftslogik. Aus diesem Grund ist nur eine Regel als Domain Service

implementiert: Transaktionen des Typs Kauf/Verkauf können nur zu Accounts von „Brokern“ hinzugefügt werden und Transaktionen des Typs Einnahme/Ausgabe können nur bei „Banken“-Konten angelegt werden.
Diese Regel ist, wie in Abbildung 1.1 abgebildet, in der Klasse „CompatibilityService“ implementiert.

```
@Component  9 usages  Niklas Rodenbüsch *
public class CompatibilityService {

    low complexity (20%)
    public boolean isInstitutionTypeCompatibleWithTransactionList  3 usages  Niklas Rodenbüsch *
            (InstitutionType institutionType, List<Transaction> transactionList){

        if(institutionType == InstitutionType.BANK){
            return !transactionList.stream().anyMatch(
                    t -> t.getTransactionType() == TransactionType.BUY ||
                         t.getTransactionType() == TransactionType.SELL);
        }
        return !transactionList.stream().anyMatch(
                t -> t.getTransactionType() == TransactionType.EXPENSE ||
                     t.getTransactionType() == TransactionType.INCOME);
    }

    low complexity (20%)
    public boolean isInstitutionTypeCompatibleWithTransactionType  10 usages  Niklas Rodenbüsch *
            (InstitutionType institutionType, TransactionType transactionType){

        if(institutionType == InstitutionType.BANK){
            return (transactionType != TransactionType.BUY && transactionType != TransactionType.SELL);
        }
        return (transactionType != TransactionType.INCOME && transactionType != TransactionType.EXPENSE);
    }
}
```

Abbildung 1.1: Domain Service: CompatibilityService-Klasse

# Kapitel 2

# Clean Architecture

Aufgabe: Schichtarchitektur planen und begründen

Abhängigkeiten nur von innen nach außen

4. Abtraction (Abstraktionen, nicht unbedingt nötig, ändert sich nie)
3. Domain (DDD Code: Aggregates, Entities, Values)
2. Application (Enthält Application Use-Cases (nicht Domain use cases))
1. Adapters (Daten-Tausch-Objekte)
0. Plugins (DB/GUI/..., arbeitet nur mit Adaptern)

# Kapitel 3

# Programming Principles

Aufgabe: Analyse und Begründung für 5 der vorgestellten Prinzipien (SOLID/GRASP/DRY/KISS/YAGNI/Conway)

# Kapitel 4

# Refactoring

Aufgabe: 2 Code smells identifizieren und die Durchführung von mindestens zwei Refactorings begründen (irgendwelche raussuchen).

# Kapitel 5

# Entwurfsmuster

Aufgabe: Einsatz von Entwurfsmustern ausführlich begründen.

# Liste der Quellcodes