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

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

Die Anwendung ist in allen fünf Schichten der Clean-Architecture implementiert. Die Implementierung verfolgt dabei stets das Ziel, Äbhängigkeiten im Code immer nur von außen nach innen zu realisieren.

2.1 Abstractions

Die Abstraktionsschicht der Clean Architecture ist in der Anwendung im Modul „4-abstractions“ implementiert. Die Schicht stellt den innersten Kern der Anwendung dar und enthält ausschließlich diejenigen Klassen, die sich während der weiteren Entwicklung der Anwendung nicht mehr verändern werden. Zu diesen Klassen gehören unter anderem eine Klasse zur Definition aller im Programm vorkommenden Konstanten, sowie die Definitionen der verschiedenen Institutions- und Transaktionstypen.

2.2 Domain

Diese zweite Schicht der Clean-Architecture (von innen), ist die Domain-Schicht. In der Anwendung wurde die Schicht im Modul „3-domain“ implementiert. Sie beinhaltet jegliche Geschäftslogik sowie die Implementierung aller in Kapitel 1.2 vorgestellten Domain-Objekte (Aggregates, Entities, ValueObjects, DomainServices). In der Anwendung sind die Domain-Objekte gleichzeitig auch als JPA-Entities implementiert.

Neben den Domain-Objekten befinden sich in dieser Schicht auch die Interfaces der in Kapitel 1.2 erläuterten Repositories.

2.3 Application

Die Applikationsschicht beinhaltet Services („ApplicationServices“) für alle Aggregates und Entities der Domain-Schicht. Während in den DomainServices der Domain-Schicht die Geschäftslogik implementiert ist, ist in den ApplicationServices der Application-Schicht die Applikationslogik implementiert. Innerhalb der Anwendung ist die Application-Schicht im Modul „2-application“ implementiert.
Die Services der Application-Schicht arbeiten mit sowohl Domain-Repositories als auch mit den DTOs der Adapter-Schicht zusammen. Sie können beispielsweise die DTOs zu Aggregates und Entities umwandeln (und umgekehrt) und diese im Anschluss zum Speichern an die Repositories weitergeben.

2.4 Adapters

Die Adapter-Schicht beinhaltet die DTOs (Data Transfer Objects), die beispielsweise als Eingabe für die API genutzt werden können. Die Adapter-Schicht ist in der Anwendung in Modul „1-adapters“ implementiert. Die Schicht enthält neben den DTOs auch die nötigen DTO-Entity-Mapper, die die DTOs in Objekte der Domain-Schicht umwandeln können. Die Adapter-Schicht dient als „Isolationsschicht“ zwischen den Domain-Objekten und der Außenwelt (bspw. der API).

2.5 Plugins

Die Plugin-Schicht ist in der Anwendung aufgeteilt in die Persistence-Schicht und die API-Schicht. Die gesamte Plugin-Schicht ist im Modul „0-plugins“ implementiert.

API

Für die API der Anwendung wurde die REST-Technologie genutzt. Jedes Aggregate und jede Entity der Domain-Schicht besitzt in der API-Schicht eine eigene REST-Controller Klasse, die jeweils die REST-Schnittstellen zum Verwenden der Applikation bereitstellen.

Persistence

Die Persistence-Schicht der Anwendung beinhaltet zum einen die Implementierung der Domain-Repositories und zum anderen die JPA-Repositories, in denen eigene SQL-Abfragen definiert sind und die einen direkten Zugriff auf die Datenbank ermöglichen. Um mit der Datenbank arbeiten zu können, greifen die Domain-Repositories auf die JPA-Repositories zu. Als Datenbank verwendet die Anwendung die in-memory-Datenbank H2.

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