

# Advanced Software-Engineering Programmmentwurf

## 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

Repository: <https://github.com/NI-R0/DH-Software-Engineering-II>

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis</b>	<b>I</b>
<b>1 Domain Driven Design</b>	<b>1</b>
1.1 Analyse der Ubiquitous Language . . . . .	1
1.2 Analyse und Begründung der verwendeten Muster . . . . .	2
<b>2 Clean Architecture</b>	<b>5</b>
2.1 Abstractions . . . . .	5
2.2 Domain . . . . .	5
2.3 Application . . . . .	5
2.4 Adapters . . . . .	6
2.5 Plugins . . . . .	6
<b>3 Programming Principles</b>	<b>7</b>
3.1 Dependency Inversion Principle . . . . .	7
3.2 Low Coupling . . . . .	7
3.3 High Cohesion . . . . .	7
3.4 Indirection . . . . .	8
3.5 Controller . . . . .	8
<b>4 Refactoring</b>	<b>10</b>
4.1 Code-Smells . . . . .	10
4.2 Refactoring . . . . .	11
4.2.1 Magic-Numbers . . . . .	11
4.2.2 Setter-Methoden . . . . .	11
<b>5 Entwurfsmuster</b>	<b>12</b>

---

# 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.

## 1.2 Analyse und Begründung der verwendeten Muster

---

**„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 ihrer Lebenszeit verändern.

### **Aggregates**

Die Objekte „Institution“ und „Account“ der Anwendung sind jeweils als Aggregates implementiert. Die Institution ist hierbei das Aggregate Root. Jede Institution besitzt 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 abgebildet, in der Klasse „CompatibilityService“ implementiert.

## 1.2 Analyse und Begründung der verwendeten Muster

---

```
@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: Domain Service: CompatibilityService-Klasse

---

## 2 Clean Architecture

Die Anwendung ist in allen fünf Schichten der Clean-Architecture implementiert. Die Implementierung verfolgt dabei stets das Ziel, Abhängigkeiten 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.

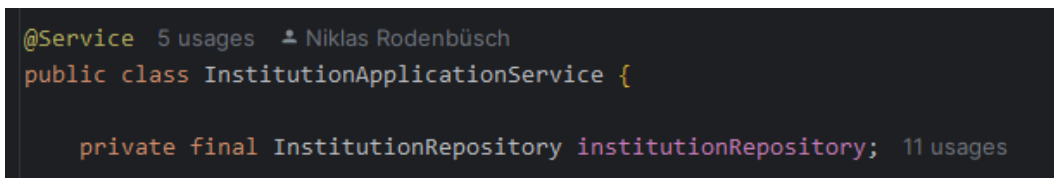


---

## 3 Programming Principles

### 3.1 Dependency Inversion Principle

Das Dependency Inversion Principle (kurz: DIP) besagt, dass Module hoher Ebenen nicht von Modulen niedriger Ebenen abhängen sollten. Stattdessen sollten beide von Abstraktionen abhängen. Weiterhin sollten Abstraktionen nicht von Details abhängen, sondern umgekehrt. Ein Beispiel hierfür sind die `ApplicationServices`, deren einzige Abhängigkeit zur Domain-Schicht in der Verknüpfung mit dem entsprechenden Domain-Repository-Interface (also der Abstraktion des Repositories) liegt. Die eigentliche Verknüpfung erfolgt zur Laufzeit über eine Injection durch die `@Autowired`-Annotation von Spring Boot.

A screenshot of a code editor showing a Java class definition. The code is as follows:

```
@Service 5 usages  Niklas Rodenbüsch
public class InstitutionApplicationService {

    private final InstitutionRepository institutionRepository; 11 usages
```

Abbildung 2: DIP-Beispiel anhand der Klasse `InstitutionApplicationService`

### 3.2 Low Coupling

Der Begriff „Kopplung“ gilt als Maß für die Abhängigkeit einer Klasse von ihrer Umgebung. Das Prinzip „Low Coupling“ besagt also, dass die Kopplung von Klassen so niedrig wie möglich gehalten werden sollte. Ein Beispiel aus der Anwendung ist die Persistence-Schicht. Die einzigen Klassen, die direkt mit dieser Schicht zusammenarbeiten, sind die „RepositoryBridges“ Persistence-Schicht. Somit könnte beispielsweise die Datenbank ausgetauscht werden, ohne dafür große Teile der Anwendung ändern zu müssen.

### 3.3 High Cohesion

Hohe Kohäsion ist ein Maß für den inneren Zusammenhalt einer Klasse, also wie „eng“ Methoden und Attribute einer Klasse zusammenarbeiten. Dieses Prinzip wurde beispielsweise durch die Implementierung des ValueObjects „`AccountOwnerNameValue`“ berücksichtigt, das

den Namen eines Account Besitzers enthält. Da der Name eines Account Besitzers an sich nichts mit den anderen Eigenschaften eines Accounts zu tun hat, erhöht die Einführung des ValueObjects die Kohäsion.

### 3.4 Indirection

Das Prinzip „Indirection“ bezeichnet das Delegieren von Aufgaben an andere Objekte und Klassen. Ein Beispiel aus der Anwendung hierfür sind die Methoden der Domain-Repositories (siehe Abb. 3) die ihre eigentlichen Aufgaben lediglich an die JPA-Repositories weitergeben.

### 3.5 Controller

Der Controller ist das erste Objekt hinter der UI, das Anweisungen annimmt und kontrolliert. Er ist quasi die „Steuereinheit“ des Programms. Der Controller ist in der Regel der einzige Ansprechpartner, den die UI hat. Er empfängt dabei Anweisungen von der UI und delegiert diese, ohne viel eigene Funktionalität zu implementieren, an Domain-Objekte und Services weiter.

Beispiele für Controller in der Anwendung sind die REST-Controller-Klassen der API-Schicht.

```
@Repository  Niklas Rodenbüsch
public class InstitutionRepositoryBridge implements InstitutionRepository {

    private final InstitutionSpringDataRepository springDataRepository; 5 usages

    @Autowired  Niklas Rodenbüsch
    public InstitutionRepositoryBridge(final InstitutionSpringDataRepository springDataRepository) {
        this.springDataRepository = springDataRepository;
    }

    @Override 1 usage  Niklas Rodenbüsch
    public List<Institution> findAllInstitutions(){
        return this.springDataRepository.findAll();
    }

    @Override 18 usages  Niklas Rodenbüsch
    public Optional<Institution> findByName(String institutionName){
        return this.springDataRepository.findById(institutionName);
    }

    low complexity (6%)
    @Override 10 usages  Niklas Rodenbüsch
    public Institution save(Institution newInstitution) {
        return this.springDataRepository.save(newInstitution);
    }

    low complexity (6%)
    @Override 2 usages  Niklas Rodenbüsch
    public void delete(Institution institution) {
        this.springDataRepository.delete(institution);
    }
}
```

Abbildung 3: Indirection principle anhand der Klasse InstitutionRepositoryBridge

---

## 4 Refactoring


### 4.1 Code-Smells

Die Anwendung weist zum aktuellen Zeitpunkt noch die folgenden Code-Smells auf:

#### Duplicate Code/Exception Handling

Nach aktuellem Stand sind viele Methoden der REST-Controller-Klassen gleich aufgebaut: In einem try-Block wird die eigentliche Aufgabe an den entsprechenden ApplicationService delegiert, während in einem catch-Block mögliche dabei auftretende Exceptions abgefangen werden. Der catch-Block besitzt dabei immer den gleichen Inhalt (vgl. Abb. 4). Dieser doppelte Code könnte durch eine ExceptionHandler-Klasse umgangen werden.

**Nachtrag:** Dieser Code-Smell wurde in diesem Commit behoben.



```
catch(Exception e){
    System.out.println("AccountController: " + e.toString());
    return ResponseEntity.badRequest().build();
}
```

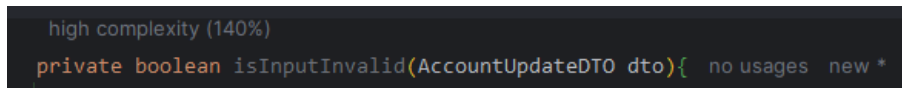
Abbildung 4: Duplicate Code in den catch-Blöcken

#### Long Method

Bis kurz vor Abschluss der Implementierung war die Methode „updateAccount“ der Klasse „AccountApplicationService“ sehr lang und unübersichtlich. Erst durch die Einführung von Validierungs-Annotationen konnte die Methode in diesem Commit etwas besser gestaltet werden.

#### Complex Method

Die für die „updateAccount“ implementierte Methode „isValid“ war eine weitere unschöne Methode. Nach den durch IntelliJ durchgeführten Berechnungen war sie zudem überdurchschnittlich komplex (vgl. siehe Abb. 5). Diese Methode(n) konnte(n) erst durch die Einführung der Validierungs-Annotationen in diesem Commit gänzlich entfernt werden.



```
high complexity (140%)  
private boolean isInputInvalid(AccountUpdateDTO dto){ no usages new *
```

Abbildung 5: Methode mit hoher Komplexität

### Schlechte Fehlermeldungen

In der Anwendung wurde bis kurz vor Abschluss der Implementierung bei fast jedem Fehler die selbe Exception, eine `IllegalArgumentException` ohne genaue Fehlermeldung, geworfen. Diese Fehlermeldung hat weder den Debugging-Prozess erleichtert noch dem API-Nutzer sinnvolle rückmeldungen geliefert. Der Code-Smell wurde mit diesem Commit durch die Implementierung einer eigenen Exception und eines eigenen `ExceptionHandler`s behoben.

## 4.2 Refactoring

### 4.2.1 Magic-Numbers

Um die zahlreichen „Magic-Numbers“ im Programmcode entfernen zu können, wurde die Klasse „Constants“ implementiert. Diese „verwaltet“ alle Magic-Numbers an einem Ort. Um konstante Werte zu verwenden, müssen andere Klassen diese aus der Constants-Klasse auslesen. Wenn sich in Zukunft eine der Konstanten ändern sollte, muss diese durch die Constants-Klasse nur noch an einer Stelle im Code geändert werden.

Commit 1, Commit 2

### 4.2.2 Setter-Methoden

Ein potenzielles Refactoring-Thema, welches zu diesem Zeitpunkt noch nicht verbessert wurde, sind die Setter-Methoden in verschiedenen Klassen (bspw. den Domain-Objekten). Setter-Methoden verletzen das Prinzip der Datenkapselung, wodurch sie theoretisch ermöglichen, Attribute von (Domain-)Objekten auf falsche Werte zu setzen. Anstelle von Setter-Methoden können beispielsweise spezifischere Methoden verwendet werden, die die Parameter auf ihre Korrektheit prüfen, bevor sie das entsprechende Attribut ändern.

---

## 5 Entwurfsmuster

Das Entwurfsmuster „Chain-of-Responsibility“ (kurz: CoR) dient dazu, die Bearbeitung von Anfragen der Reihe nach auf mehrere Objekte zu verteilen. Dabei wird eine Kette von Objekten gebildet, wobei jedes Objekt die Möglichkeit hat, die Anfrage zu bearbeiten und/oder sie an das nächste Objekt weiterzugeben.

Der Einsatz von CoR ermöglicht es, Anfragen zu verarbeiten, ohne dass Sender und Empfänger der Nachricht fest verknüpft sind. Durch diese Art von Entkopplung können neue Handler-Objekte ohne weiteres zur Kette hinzugefügt, geändert oder ganz entfernt werden, ohne dabei das gesamte System anpassen zu müssen. Dies fördert in der Regel sowohl die Wartbarkeit, als auch die Übersichtlichkeit des Codes der gesamten Kette. Durch klar definierte Verantwortlichkeiten der einzelnen Handler-Objekte wird außerdem die Implementierung der Anfragebearbeitung vereinfacht und besser strukturiert. Bei komplexen Ketten mit vielen verschiedenen Handlern kann es allerdings dazu kommen, dass die Wartbarkeit der wieder Kette abnimmt. Des weiteren besteht die Gefahr, dass Anfragen die gesamte Kette durchlaufen, bevor sie bearbeitet werden, was zu einer ineffizienten Verarbeitung führen kann.

In der Anwendung wird das Entwurfsmuster beispielsweise bei der Verarbeitung von REST-Anfragen verwendet: Vom REST-Controller wird eine Anfrage über den `ApplicationService` an das `DomainRepository` weitergeleitet, welches die Anfrage an das `SpringDataRepository` weitergibt. Hier wird die Anfrage abschließend bearbeitet, bevor die Response die gesamte Kette zurückgereicht wird, bis sie wieder beim REST-Controller ankommt. Insgesamt wurde das CoR-Muster drei mal in der Anwendung umgesetzt, einmal für jeden REST-Controller.