

Programmentwurf

TECHNISCHE DOKUMENTATION

für die Vorlesung

Advanced Software Engineering

des Studienganges Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Pascal Stephan

Abgabedatum 31.05.2021

Bearbeitungszeitraum

Matrikelnummer

Kurs

Gutachter der Studienakademie

5. + 6. Semester

3977689

TINF18B4

Mirko Dostmann

Inhaltsverzeichnis

1	Einleitung	2
1.1	Installation	2
1.1.1	Anforderungen	2
1.1.2	Starten der Anwendung mit Visual Studio 2019	3
1.1.3	Starten der Anwendung mit Visual Studio Code	3
1.1.4	Benutzen der Anwendung	3
2	Domain Driven Design	4
2.1	Ubiquitous Language	4
2.2	Analyse und Begründung der verwendeten muster des DDD	4
2.2.1	Value Objects (VO)	4
2.2.2	Entities	4
2.2.3	Aggregates	6
2.2.4	Repositories	7
2.2.5	Domain Services	7
3	Clean Architecture	9
3.1	Plugin-Schicht	10
3.2	Adapter-Schicht	10
3.3	Domain-Schicht	10
3.4	Weitere nicht implementierte Schichten	11
3.4.1	Application-Schicht	11
3.4.2	Abstraction-Schicht	11
4	Programming Principles	12
4.1	SOLID	12
4.1.1	Single Responsibility Principle	12
4.1.2	Open Closed Principle	12
4.1.3	Liskov Substitution Principle	13
4.1.4	Interface Segregation Principle	13
4.1.5	Dependency Inversion Principle	13
4.2	GRASP	13
4.2.1	Low Coupling	14
4.2.2	High Cohesion	14
4.2.3	Information Expert	15
4.2.4	Creator	15

4.2.5	Indirection	15
4.2.6	Polymorphism	15
4.2.7	Controller	16
4.2.8	Pure Fabrication	16
4.2.9	Protected Variations	16
4.3	DRY	16
4.3.1	Anwendung des DRY-Prinzips	17
5	Entwurfsmuster	18
5.1	Singleton-Entwurfsmuster	18
5.2	Einsatz	18
5.3	Vorher-Nachher-Vergleich	19
6	Refactoring	20
6.1	Code Smell 1 - Lange Methode	20
6.2	Code Smell 2 - Duplikate im Code	20
7	Unit Tests	23
7.1	Mocking	23
7.2	Unit Tests	23
7.3	Ausführen der Tests	23

Abbildungsverzeichnis

2.1	AdressVO	5
2.2	KontoEntity	6
2.3	BankAggregate	7
2.4	BankRepository	8
2.5	Ausschnitt aus CredentialsService	8
3.1	Schichten der Clean Architecture	9
5.1	Singleton Vorher	19
5.2	Singleton Nachher	19
6.1	Long Method	20
6.2	Long Method Refactored	21
6.3	Duplicate Code	21
6.4	Duplicate Code Refactored	22

Kapitel 1

Einleitung

In diesem Programmentwurf wurde Finanz/Banking-Anwendung entwickelt. Das projekt besteht aus einem backend und einem Frontend. Für das Backend wurde eine .NET Core WebApi erstellt und für das Frontend ein einfacher Client mit Hilfe von Windows Forms. Der Client dient dazu die Funktionen der API auszuführen und darzustellen. Die WebApi greift auf Anfrage des Clients auf eine InMemory-Datenbank zu und führt auf dieser Operationen aus. Da es sich hier um eine InMemory-Datenbank handelt, die im Speicher der Api läuft, wird die Datenbank resettet, wenn die WebApi gestoppt wird.

Das Frontend und das Backend wurden mit der Visual Studio 2019 Community Edition erstellt. Um das Testen der WebApi durch den Client zu vereinfachen, wird bei Start des Clients automatisch ein Test-Benutzer angelegt. Die Anmeldedaten für diesen Testnutzer sind:

- Email: Test@test.com
- Passwort: Passw0rd

Dieser Benutzer wird als Admin angelegt und kann somit alle Funktionen der Anwendung nutzen. Welche Funktionen nicht genutzt werden können, wird später erklärt.

Weiterhin werden bei Start 3 Banken auf der Datenbank angelegt, um Konten anlegen zu können, ohne dass eine Bank ausgewählt werden muss.

Zum eigentlichen Programmentwurf gehört nur die WebApi. Alle Vorgaben für den Programm-entwurf wurden in dieser Api umgesetzt und nicht im Client-Projekt. Der Client hilft nur beim Testen der Funktionen der Api.

Als Admin-Benutzer können im Client neue Banken angelegt werden, die daraufhin jedem anderen benutzer sichtbar sind. Diese Funktion ist nicht für Nicht-Admin-Benutzer verfügbar, da nicht alle Benutzer die Möglichkeit haben sollten, neue Banken einzutragen, da diese daraufhin in den Clients aller Benutzer erscheinen. So wird verhindert, dass normale Benutzer zu viele Banken eintragen, auf denen Konten angelegt werden können oder Dopplungen entstehen..

1.1 Installation

1.1.1 Anforderungen

Um die WebApi zu starten wird folgende Software benötigt:

- .NET SDK 5.0

- Visual Studio 2019 oder Visual Studio Code

1.1.2 Starten der Anwendung mit Visual Studio 2019

Um die WebApi in Visual Studio 2019 zu starten, wird die gesamte Projektmappe mit Visual Studio geöffnet. Im Solution-Explorer sind daraufhin folgende Projekte zu sehen:

- Programmentwurf_BankingApi
- Programmentwurf_Banking_Client
- Programmentwurf_Mock_Tests
- Programmentwurf_xUnit_Tests

In „Programmentwurf_BankingApi“ befinden sich die einzelnen Projekte der WebApi. Durch Rechtsklick auf die Solution im Solution-Explorer kann unter „Startprojekte festlegen“ ausgewählt werden, welche Projekte durch drücken von F5 gestartet werden sollen. Hier wird für das Projekt „0_Plugin“ die Option „Starten“ ausgewählt. Optional kann auch für das Projekt „Programmentwurf_Banking_Client“ die Option „Starten“ ausgewählt werden, um auch den Client direkt zu starten.

1.1.3 Starten der Anwendung mit Visual Studio Code

Um die WebApi in Visual Studio Code zu starten, muss das Projekt „0_Plugin“ ausgeführt werden. Dafür wird in den Ordner „Programmentwurf_BankingApi“ navigiert und von dort in den Projektordner „0_Plugin“. Dort wird über die Terminal-Konsole folgende Befehle ausgeführt:

- dotnet build
- dotnet run

Mit „dotnet“ build wird versucht die Anwendung zu kompilieren. Dadurch werden auch alle NuGet-Pakete heruntergeladen, die für die Entwicklung des Programms genutzt wurden. Mit „dotnet run“ wird daraufhin die Anwendung gestartet. Wird die Anwendung über Visual Studio Code gestartet, ist sie über den localhost mit Port 5001 erreichbar.

1.1.4 Benutzen der Anwendung

Die WebApi kann entweder durch Postman oder den dazugehörigen Client getestet werden. Im GitHub-Repository befindet sich eine postman-Collection die genutzt werden kann um die WebApi mit Postman zu testen.

Bei Ausführen des Clients wird ein Login-Screen geöffnet. Von diesem aus kann entweder ein neuer Benutzer registriert werden oder es wird der Admin-Benutzer zum einloggen genutzt, dessen Anmeldedaten zuvor schon erwähnt wurden. Daraufhin öffnet sich der Home-Screen von dem aus alle Funktionen erreichbar sind.

Es muss beachtet werden, dass jedesmal wenn etwas erstellt oder hinzugefügt wird über den Client, der Aktualisierungs-Button geklickt werden muss, um die Daten im Client zu aktualisieren.

Kapitel 2

Domain Driven Design

2.1 Ubiquitous Language

In 2.1 wird die Ubiquitous Language der Domäne analysiert.

2.2 Analyse und Begründung der verwendeten muster des DDD

2.2.1 Value Objects (VO)

Value Objects oder auch Wertobjekte sind Objekte, die unveränderbar sind. Diese werden einmal erstellt und sind daraufhin nichtmehr änderbar, da sie spezielle Werte repräsentieren. Sie besitzen keine Methoden, da sie nur auf ihre Werte reduziert werden. Soll der Wert eines Value Objects doch geändert werden, müssen diese neu erstellt werden. In diesem Programmentwurf wurden Value Objects zum einen für die Adresse einer Bank erstellt und zum anderen für die Informationen einer Transaktion. Die Adresse ist ein gutes Beispiel für ein Value Object, da sich die Adresse einer Bank später normalerweise nicht ändert, außer die komplette Bank zieht in ein anderes Gebäude um. Dies geschieht jedoch nicht so häufig, dass die Adresse änderbar sein muss. Auch die Informationen einer Transaktion sollten unveränderbar sein, damit immer nachvollzogen werden kann, welcher Betrag von welchem Konto auf welches Konto überwiesen wurde.

In 2.1 wird ein Ausschnitt aus dem Adressen Value Object gezeigt:

2.2.2 Entities

Entities unterscheiden sich in den folgenden 3 Punkten von Value Objects:

- Entities haben eine eindeutige Id
- Wenn Entities unterschiedliche Ids haben unterscheiden sie sich voneinander
- Eine Entity hat einen Lebenszyklus und verändert sich während diesem Zyklus häufiger

Es wurden in diesem Projekt folgende Entitäten angelegt:

- User
- Konten

Wort	Bedeutung
User	Ein User stellt eine Person dar, die ihre Konten verwalten möchte
Konto	Ein Konto stellt eine Einheit dar, auf der Geld gespeichert wird
Transaktion	Eine Transaktion ist ein Auftrag, der von einem Konto ausgeht oder dieses betrifft. Dies stellt beispielsweise eine Überweisung dar
Bank	Eine Bank ist ein Institu, das Konten von Personen hält und für diese Personen Transaktionen ausführt
Anmelden / Login	Ein bereits registrierter User meldet sich an einem System an
Registrieren	Eine Person erstellt sich einen User in einem System
Adresse	Durch eine Adresse kann eindeutig identifiziert werden, wo sich in dieser Domäne eine Bank befindet oder ein User lebt
Kontostand ändern	Das Geld, das auf einem Konto gespeichert ist, wird entweder erhöht oder verringert
überweisen	Beim Überweisen wird eine transaktion erstellt, die Geld von einem Konto auf ein anderes überträgt

```

4 Verweise
public sealed class TransactionVO
{
    1 Verweis
    public TransactionVO(DateTime date, double betrag, int kontoIdSender, int kontoIdEmpfänger)
    {
        Date = date;
        Betrag = betrag;
        KontoIdSender = kontoIdSender;
        KontoIdEmpfänger = kontoIdEmpfänger;
    }
    2 Verweise
    public DateTime Date { get; set; }
    2 Verweise
    public double Betrag { get; set; }
    2 Verweise
    public int KontoIdSender { get; set; }
    2 Verweise
    public int KontoIdEmpfänger { get; set; }
    3 Verweise
    public DateTime getDate()
    {
        return Date;
    }

    7 Verweise | 2/2 bestanden
    public double getBetrag()
    {
        return Betrag;
    }
}

```

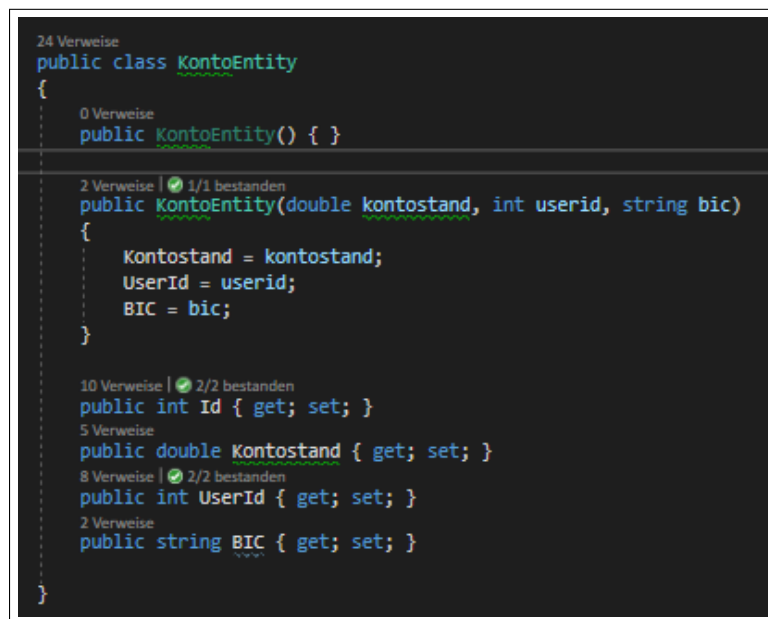
Abbildung 2.1: AdressVO

- Banken

Diese sind eindeutige Objekte die unterschieden werden müssen. Sie können sich von Zeit zu Zeit verändern. Vorallem Konten verändern sich häufig, da immer wieder Geld auf ein Konto eingezahlt und abgebucht wird.

Eine Entity sorgt auch dafür, dass keine ungültigen Werte gesetzt werden, um nicht in einen ungültigen zustand zu gelangen. Außerdem besitzen sie Methoden, die das Verhalten der Entitäten beschreiben.

In 2.2 ist als Beispiel die erstellte KontoEntity zu sehen:



```
24 Verweise
public class KontoEntity
{
    0 Verweise
    public KontoEntity() { }

    2 Verweise | 1/1 bestanden
    public KontoEntity(double kontostand, int userid, string bic)
    {
        Kontostand = kontostand;
        UserId = userid;
        BIC = bic;
    }

    10 Verweise | 2/2 bestanden
    public int Id { get; set; }
    5 Verweise
    public double Kontostand { get; set; }
    8 Verweise | 2/2 bestanden
    public int UserId { get; set; }
    2 Verweise
    public string BIC { get; set; }
}
```

Abbildung 2.2: KontoEntity

2.2.3 Aggregates

Aggregate gruppieren Entities und Value Objects zu gemeinsam verwalteten Einheiten. Aggregate helfen dabei die Komplexität der Beziehungen zwischen Objekten zu reduzieren. Die Entity im Aggregat dient als Aggregate Root Entity.

Folgende Aggregate sind erstellt worden:

- BankAggregate
- TransactionAggregate

Eine außenstehende Klasse muss eine Methode der Aggregate Root Entity aufrufen, sollte der innere Zustand geändert werden sollen. Dadurch wird sichergestellt, dass der Zustand immer den Domänenregeln entspricht.

Das BankAggregate fasst die BankEntity und die dazugehörige Adresse zusammen. Das TransactionAggregate hält das Value Object für die Transaktionsinfos und gibt jeder Transaktion eine Id. Hier könnte für die Transaktion noch eine TransactionEntity angelegt werden, diese würde allerdings nur die Id der Transaktion halten, weshalb sich dagegen entschieden wurde

eine TransactionEntity zu erstellen. Weitere Informationen in der TransactionEntity zu halten macht keinen Sinn, da diese nicht änderbar sein sollen und deshalb in einem Value Object gehalten werden. Der Vollständigkeit halber sollte auch für die übrigen Entities wie UserEntity und KontoEntity ein Aggregat erstellt werden, die nur die jeweilige Entity besitzen. Darauf wurde jedoch verzichtet, um die Übersichtlichkeit des Projektes zu verbessern. Es kam öfters zu Verwirrungen, da bei der Benutzung des Entity Frameworks und der InMemory-Datenbank auch das Aggregat eine Id benötigt, um es zu speichern.

In 2.3 wird ein Ausschnitt aus dem BankAggregate gezeigt:

```
public class BankAggregate
{
    public BankAggregate() { }
    3 Verweise | 1/1 bestanden
    public BankAggregate(string name, string bic, string land, int plz, string straße)
    {
        Bank = new BankEntity(name, bic);
        Adresse = new AdressVO(land, plz, straße);
    }
    0 Verweise
    public BankAggregate(int id, string name, string bic, string land, int plz, string straße)
    {
        Id = id;
        Bank = new BankEntity(name, bic);
        Adresse = new AdressVO(land, plz, straße);
    }
    1 Verweise
    public int Id { get; set; }
    13 Verweise | 2/2 bestanden
    public BankEntity Bank { get; set; }
    14 Verweise | 2/2 bestanden
    public AdressVO Adresse { get; set; }
}
```

Abbildung 2.3: BankAggregate

2.2.4 Repositories

Es wurden Repositories für User, Konto, Bank und Transaktion erstellt. Sie vermitteln zwischen der Domäne und dem Datenmodell. Innerhalb der Repositories werden Methoden zur Verfügung gestellt, um Aggregates aus dem Persistenzspeicher zu lesen, zu speichern oder zu löschen. Der Domain Code erhält dadurch Zugriff auf den persistenten Speicher, deren Implementierung wird der Domäne allerdings verborgen und ist somit flexibler. So wird eine Anti-Corruption-Layer zur Persistenzschicht gebildet. Ein großer Vorteil von Repositories ist, dass zukünftig weitere Datenbanken einfach hinzugefügt oder ausgetauscht werden können, , ohne dass der Domain Code davon beeinflusst wird.

Die wichtigsten Methoden eines Repository's sind diejenigen, die eine Aggregate Root Entity anhand ihrer Eigenschaften finden können. Dazu gehören beispielsweise die Funktionen eine Root Entity über ihre Id zu finden.

Implementiert werden die Repositories in der Plugin-Schicht, da sie, wie bereits erwähnt, die Methoden zur Verfügung stellen um mit dem Persistenzspeicher zu arbeiten.

In 2.4 wird eines der erstellten Repositories gezeigt:

2.2.5 Domain Services

Domain Services dienen zum einen der Abbildung von komplexem Verhalten, das nicht eindeutig einer bestimmten Entity oder Value Object zugeordnet werden können und zum anderen als Definition eines „Erfüllungs-Vertrages“ für externe Dienste, um zu verhindern, dass ein Domänenmodell nicht mit unnötiger „accidental complexity“ belastet wird. Eine weitere wichtige

```
namespace _3_Domain.Domain.Repositories
{
    1 Verweis
    public interface BankRepository
    {
        2 Verweise
        Task<bool> create(BankAggregate bank);
        2 Verweise
        Task<bool> delete(int bankid);
        2 Verweise
        Task<List<BankAggregate>> getAllBanks();
        2 Verweise
        Task<BankAggregate> findById(int bankid);
    }
}
```

Abbildung 2.4: BankRepository

Eigenschaft von Domain Services ist, dass sie selbst zustandslos sind. In diesem Projekt wurde beispielsweise ein Domain Service zur Überprüfung der Namens-, Email- und Passwortkonventionen erstellt. Dieser überprüft mit Hilfe einer Regular Expression bei Erstellung eines Benutzer, ob die übergebenen Parameter korrekt sind. In 2.5 wird ein Ausschnitt des implementierten Domain Service gezeigt:

```

4 Verweise
public class CredentialsService
{
    private static readonly string NAME_REGEX = "(?:[A-Z]|[a-z]|[0-9]|_){4,16}";

    private static readonly string EMAIL_REGEX =
        "^(?:[a-z0-9!#$%&'*/+=?^_`{|}~]+(?:\\.[a-z0-9!#$%&'*/+=?^_`{|}~]+)*|(?:\\\\x01-\\\\x08|\\\\x0A)"+

    private static readonly string PW_REGEX = "^(?=.*?[0-9])(?=.*?[a-zA-Z])(?=.*?[a-zA-Z]).{8,}$";

    4 Verweise
    public bool IsValid(string name)
    {
        Match match = Regex.Match(input: name, pattern: NAME_REGEX);

        if (match.Success)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

Abbildung 2.5: Ausschnitt aus CredentialsService

Kapitel 3

Clean Architecture

Die Architektur der WebApi wurde nach den Regeln der Clean Architecture geplant. Dabei gibt es 5 Schichten die implementiert werden können. Diese Schichten sind in 3.1 zu sehen: Die

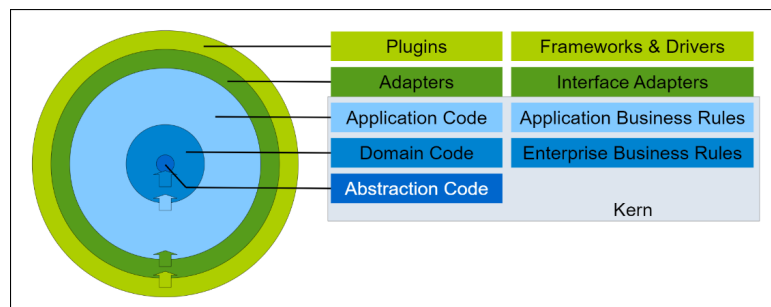


Abbildung 3.1: Schichten der Clean Architecture

Grundregeln der Clean Architecture besagen:

- Der Anwendungs- und Domaincode ist frei von technischen Details
- Innere Schichten definieren Interfaces, äußere Schichten implementieren diese
- Sämtlicher Code kann eigenständig verändert werden
- Sämtlicher Code kann unabhängig von Infrastruktur kompiliert und ausgeführt werden
- Äußere Schichten koppeln sich an die inneren Schichten in richtung Zentrum

Hierbei ist die wichtigste aller Regeln, dass innere Schichten nicht von äußeren Schichten abhängen dürfen. Dadurch können äußere Schichten jederzeit ausgetauscht oder verändert werden, ohne dass die inneren Schichten etwas davon mitbekommen. Dies folgt auch dem Prinzip der Dependency Inversion.

In diesem Projekt wurden 3 der 5 Schichten der Clean Architecture verwendet. Diese Schichten sind:

- Plugin-Schicht
- Adapter-Schicht

- Domain-Schicht

Im Folgenden werden die verwendeten Schichten erklärt.

3.1 Plugin-Schicht

Die Plugin-Schicht ist die äußerste Schicht der Clean Architecture. Sie enthält die Klassen zum Starten der WebApi, stellt die Verbindung zur Datenbank her und führt Operationen auf der Datenbank aus. Außerdem liegen auf dieser Schicht die Controller, die die Anfragen entgegennehmen und Antworten zurück an den Client schicken.

Sie implementiert die Repositories aus der Domain-Schicht und führt aufgrund dieser die Operationen auf der Datenbank aus. Die Adapter-Schicht wird verwendet, um die Ergebnisse aus der Datenbank umzumappen, damit sie an den Client zurückgeschickt werden können.

Die Plugin-Schicht ist die einzige Schicht, die externe Abhängigkeiten haben darf, da sie ganz außen liegt. In diesem Projekt ist dies beispielsweise durch die Abhängigkeit der Plugin-Schicht vom Entity Framework, das für die Datenbank-Operationen zuständig ist, zu sehen.

3.2 Adapter-Schicht

Die Adapter-Schicht konvertiert externe Formate so, dass die Applikation damit zurecht kommt und interne Formate so, dass externe Plugins damit zurecht kommen. Das Ziel dieser Struktur ist die Entkopplung von inneren und äußeren Schichten. Sie dient als Anti-Corruption Layer zwischen den Technischen Schichten und der Geschäftslogik.

Da in diesem Programmentwurf mit einer InMemory-Datenbank, die innerhalb der Api läuft mit Hilfe der DbContext-Library und diese Einträge aufgrund der Entitäten der Anwendung baut, wird die Funktion externe Strukturen auf innere Strukturen abzubilden in der Adapter-Schicht nicht benötigt. Allerdings werden in der Adapter-Schicht in diesem Programmentwurf die inneren Strukturen wie Entitäten in Strukturen konvertiert, die an den Client weitergegeben werden können. Beispielsweise werden UserEntities in User-Objekte konvertiert, die nicht das Passwort des Benutzers enthalten, um eine Liste aller Benutzer in jedem Client darstellen zu können. Dadurch kann ein Benutzer alle anderen Benutzer sehen, um ihnen Geld zu überweisen, ohne dass der Client User-Objekte erhält, die das Passwort des Benutzers enthalten.

3.3 Domain-Schicht

In der Domain-Schicht befinden sich die Value Objects, Entities, Aggregates, Repositories und Domain Services. Die äußeren Schichten greifen auf diese Schicht zu, die Domain-Schicht kann aber auf keine andere Schicht zugreifen. Hier liegt die allgemeine Geschäftslogik. Dies hilft dabei zukünftig die Domain-Schicht auch in anderen Anwendungen verwenden zu können, wenn benötigt, da höchstens Abhängigkeiten von der Abstraction-Schicht bestehen, die sich nur sehr selten ändert. Die Repository-Interfaces, die in den äußeren Schichten implementiert werden, erlauben es der Domain-Schicht oder auch der Application-Schicht, wenn diese genutzt wird, auf die Datenbank zuzugreifen ohne von der Implementierung der Interfaces abhängig zu sein. Dadurch wird eine Inversion of Control erzeugt.

Die Domain-Schicht in diesem Programmentwurf enthält alle der oben genannten Muster des Domain Driven Designs.

3.4 Weitere nicht implementierte Schichten

3.4.1 Application-Schicht

In der Application-Schicht liegt die eigentliche anwendungsspezifische Geschäftslogik und die einzelnen Use Cases der Anwendung. Hier wird der Fluss der Daten von den Elementen der Domain-Schicht ausgehend und zu den Elementen führend gesteuert. Diese Schicht kann nur auf die Domain-Schicht zugreifen und Änderungen auf dieser Schicht beeinflussen die Domain-Schicht nicht. Sie funktioniert isoliert von Änderungen auf der Datenbank oder anderen Plugins, das bedeutet der genutzte Use Case weiß nicht, wer ihn aufgerufen hat oder auf welche Weise das Ergebnis präsentiert wird.

Diese Schicht wurde in diesem Programmentwurf nicht explizit implementiert.

3.4.2 Abstraction-Schicht

Die Abstraction-Schicht enthält Domänen übergreifendes Wissen, wie Grundbausteine, die nicht domänenspezifisch sind, allgemeine Konzepte und Algorithmen oder nachgerüstete Libraries. Der Code auf dieser Schicht ändert sich selten bis nie und ist dadurch sehr stabil. Sie darf von keiner anderen Schicht abhängen, da sie nach dem Prinzip der Clean Architecture ganz innen liegt. In der Praxis muss diese Schicht häufig nicht explizit angelegt werden. Sie kann auch erst nachträglich eingebaut oder auch extrahiert werden. Genauere Beispiele für Code auf dieser Schicht sind beispielsweise Sortier-Algorithmen.

Diese Schicht wurde in diesem Programmentwurf nicht explizit implementiert, da keine der zuvor genannten Bausteine oder Muster in dieser Anwendung verwendet werden.

Kapitel 4

Programming Principles

4.1 SOLID

SOLID setzt sich aus folgenden Prinzipien zusammen:

- S: Single Responsibility Principle
- O: Open Closed Principle
- L: Liskov Substitution Principle
- I: Interface Segregation Principle
- D: Dependency Injection Principle

Für jedes dieser Prinzipien wird das Vorkommen im Code dieses Programmentwurfs analysiert und begründet. Es kann jedoch durchaus vorkommen, dass eines dieser Prinzipien nicht im Code auffindbar ist, da es nicht benötigt wurde.

4.1.1 Single Responsibility Principle

Das Single Responsibility Principle besagt, dass eine Klasse genau eine Zuständigkeit haben sollte. Das bedeutet, dass jede Klasse eine klar definierte Aufgabe hat, wodurch eine niedrige Komplexität des Codes entsteht und eine niedrige Kopplung. Durch die niedrige Komplexität des Codes lässt sich dieser auch einfacher warten und erweitern, da er besser verständlich ist. Angewendet wurde dieses Prinzip beispielsweise in der Adapter-Schicht. Hier wurde für das Konvertieren jedes Objektes eine extra Klasse erstellt. Die Klasse „UserAggregateToUserMapper“ ist nur dafür zuständig ein UserAggregate in ein User-Objekt umzuwandeln und nicht für Objekte anderer Art zuständig. Nach diesem Prinzip gibt es für jedes Aggregat einen Mapper, der einem Aggregat entweder Eigenschaften, die der Client nicht benötigt oder erhalten sollte, entnimmt oder das Aggregat weniger komplex für den Client macht wie zum Beispiel der „TransactionAggregateToTransactionMapper“.

4.1.2 Open Closed Principle

Das Open Closed Principle macht eine Anwendung offen für Erweiterungen aber geschlossen für Änderungen. Das bedeutet, dass der Code nur durch Vererbung oder die Implementierung von

interfaces erweitert wird. Dadurch muss bestehender Code nicht geändert werden. Um dies zu unterstützen, ist es von Vorteil viele Abstraktionen zu nutzen.

In diesem Projekt wird dieses Prinzip vor allem in der Domain-Schicht sichtbar durch die Repository-Interfaces und die Domain Service-Interfaces. Es kann beispielsweise auf einfachste Weise neue Funktionen implementiert werden, indem ein neues Interface erstellt, das die neuen Funktionen nutzt und dieses Interface in der Plugin-Schicht implementiert wird. Daraufhin müssen keine Änderungen an anderen Klassen als dem erstellten Interface und der Klasse die dieses implementiert vorgenommen werden.

4.1.3 Liskov Substitution Principle

Das Liskov Substitution Principle schränkt Ableitungsregeln stark ein, wodurch Invarianzen eingehalten werden. Dabei müssen abgeleitete Typen schwächere Vorbedingungen und stärkere Nachbedingungen besitzen, wodurch in der objektorientierten Programmierung eine „verhält sich wie“ Beziehung entsteht. Wenn nun das Verhalten eines Basistypes bekannt ist, kann sich darauf verlassen werden, dass der abgeleitete Typ dieses Verhalten übernimmt.

Dies ist beispielsweise bei der Implementierung der Repository-Interfaces zu sehen. Diese geben der Implementierung eine „verhält sich wie“ Beziehung.

4.1.4 Interface Segregation Principle

Das Interface Segregation Principle besagt, dass Klassen, die ein Interface implementieren auch genau die Methoden des Interfaces implementieren, die sie benötigen und keine weiteren unnötigen Methoden. Dies wird umgesetzt, indem anstatt einem großen Interface, mehrere kleine Interfaces mit wenigen Funktionen genutzt. Daraufhin werden genau die Interfaces in einer Klasse implementiert, die auch nur genau die Funktionen besitzen, die die Klasse benötigt und keine weiteren Funktionen mitbringen, die nicht benötigt werden.

Da im Programmentwurf keine Interfaces bestehen, die in einer Klasse eine unnötige Methode implementieren, ist dieses Prinzip erfüllt.

4.1.5 Dependency Inversion Principle

Durch das Dependency Inversion Principle wird die klassische Struktur, in der High-Level Module von Low-Level-Modulen abhängig sind umgekehrt. Dies geschieht, da Abstraktionen nicht von Details abhängig sein sollten. High-Level Module geben also die Regeln vor und Low-Level Module implementieren diese. Dadurch wird eine hohe Flexibilität der Software erreicht, da Low-level Module einfach ausgetauscht werden können, ohne dass High-Level Module ausgetauscht werden.

Dieser Programmentwurf setzt dieses Prinzip durch die verwendete Schichtenarchitektur der Clean Architecture um. Dabei werden die Repository- und Domain Service-Interfaces aus der High Level Domain-Schicht in den äußeren Low Level-Modulen implementiert und aufgerufen.

4.2 GRASP

Grasp steht für General Responsibility Assignment Software Patterns. Diese beschreiben Basispattern, auf denen ein Entwurfsmuster aufbaut. Das Ziel dieses Prinzips ist es, die Low

Representational Gap möglichst klein zu halten, was bedeutet, dass die Lücke zwischen dem gedachten Domänenmodell und der eigentlichen Softwareimplementierung möglichst klein gehalten wird.

4.2.1 Low Coupling

Die Klassenkopplung ist ein Maß, das angibt, wie viele Klassen eine einzelne Klasse verwendet. Es wird versucht eine möglichst geringe Kopplung zu erreichen, wodurch eine geringere Abhängigkeit von Änderungen in anderen Teilen des Codes entsteht. Dies macht den Code außerdem einfacher zu testen und wiederverwendbar. Eine niedrige Kopplung macht den Code auch einfacher verständlich, da weniger Wissen über andere Klassen benötigt wird. Eine lose Kopplung macht Komponenten austauschbarer.

Kopplung entsteht beispielsweise durch das Halten von Attributen, deren Typ eine andere Klasse ist, durch das Aufrufen bzw. Besitzen von Methoden mit Referenz auf eine andere Klasse oder wenn Interfaces verwendet werden. Eine Klasse kann an konkrete oder abstrakte Datentypen gekoppelt sein, an Threads, die gemeinsame Sperren besitzen oder auch an Ressourcen, die gemeinsame Dateien nutzen. Dabei ist jedoch die Kopplung an stabilere Komponenten weniger problematisch.

In Visual Studio kann standardmäßig die Klassenkopplung eines Projektes berechnet werden. Dabei kann für jede Klasse des Projektes die Klassenkopplung begutachtet werden.

In diesem Programmentwurf wird durch die Inversion of Control erreicht, dass die Kopplung von Klassen hauptsächlich an stabilere Komponenten besteht. Die meisten Klassen sind hauptsächlich von inneren Klassen, also von stabileren Komponenten abhängig. Diese Komponenten sind stabiler, da sie seltener geändert werden, als die Klassen auf äußeren Schichten. Ein Beispiel hierfür ist, dass die Mapper auf der Adapter-Schicht hauptsächlich von Klassen der Domain-Schicht abhängig sind.

4.2.2 High Cohesion

Die Kohäsion ist ein Maß, das für den Zusammenhalt einer Klasse steht. Es wird die semantische Nähe der Elemente einer Klasse beschrieben.

Ein wichtiges Prinzip ist das „High Cohesion & Low Coupling“-Prinzip. Es ist das Fundament für einen idealen Code, da dieses Prinzip zu einem einfacheren und verständlicherem Design des Codes führt. Dieses Prinzip ist jedoch schwer automatisiert testbar, weshalb es hauptsächlich das menschliche Ermessen benötigt, um zu entscheiden ob es gut umgesetzt ist oder nicht.

Der Zusammenhalt der Klassen in diesem Projekt ist gut. Dies wird vor allem in den Entities oder Value Objects in der Domain-Schicht klar, da hier zu sehen ist, dass diese nur Eigenschaften besitzen, die semantisch zu ihnen passen. Ein Beispiel wäre das „AdressVO“. Es besitzt nur Eigenschaften, die zu einer Adresse passen, wie das Land, die Postleitzahl und die Straße, in der sich das Objekt befindet. Ein weiteres Beispiel ist die „UserEntity“-Klasse. Diese besitzt nur Eigenschaften, die semantisch zu einem angelegten Benutzer passen. Dies sind die Email-Adresse, der Name des Benutzer und das Passwort, das benötigt wird um sich mit diesem Benutzer anzumelden.

4.2.3 Information Expert

Bei diesem Prinzip geht es um die allgemeine Zuweisung einer Zuständigkeit zu einem Projekt. Am einfachsten ist dies umzusetzen, indem einem Objekt, das eine bestimmte Information besitzt, auch die Zuständigkeit für diese Information überreicht wird. Dafür muss allerdings im Designmodell eine passende Klasse existieren. Sollte dies nicht der Fall sein, wird im Domänenmodell eine passende Repräsentation gesucht und dafür eine passende Klasse im Designmodell erstellt. Dieses Objekt ist dann zuständig für die Informationen, die es besitzt. Dadurch entsteht eine Kapselung von Informationen und leichten Klassen.

In diesem Programmentwurf ist dieses Prinzip in den Aggregaten sichtbar. Diese übernehmen die Verantwortung über die enthaltenen Entities und deren Informationen bzw. über die enthaltenen Value Objects. Ein Beispiel hierfür wäre das „BankAggregate“. Dieses übernimmt die Zuständigkeit über die Informationen des „AdressVO“ und die „BankEntity“.

4.2.4 Creator

Das Creator-Prinzip legt fest, wer für die Erzeugung eines Objektes zuständig ist. Ein Objekt kommt als Creator eines anderen Objektes in Frage, wenn es eine Beziehung zu jedem erstellten Objekt gibt. Dadurch wird die Kopplung der Komponenten verringert.

In diesem Programmentwurf sind ein Beispiel für dieses Prinzip die Mapper auf der Adapter-Schicht. Sie konvertieren Objekte und erzeugen dadurch ein neues Objekt einer anderen Klassen. Hier gibt es für jedes Objekt einen eigenen Mapper, weshalb die Beziehung zwischen den Objekten und dem Mapper hergestellt ist. Dadurch wird auch die Kopplung der Klassen verringert.

4.2.5 Indirection

Das Indirection-Prinzip besagt, dass ein System oder die Teile eines Systems voneinander entkoppelt werden. Dies ist allerdings mit viel Aufwand verbunden, bietet jedoch einen höheren Freiheitsgrad als die Nutzung von Vererbung oder Polymorphismus.

In diesem Programmentwurf wurde dies beispielsweise durch die Repository-Schnittstellen erreicht. Diese wurden so designed, dass sie genau ihrem Anwendungszweck angepasst sind und somit eine höhere Flexibilität erreichen.

4.2.6 Polymorphism

Polymorphismus ist ein grundlegendes Prinzip der objektorientierten Programmierung. Dabei erhalten Methoden, je nach Typ eine andere Implementierung. Dadurch werden Fallunterscheidungen vermieden, wie If-Else bzw. Switch-Statements. Es werden dafür abstrakte Klassen oder Interfaces als Basistypen genutzt. Polymorphismus macht eine Anwendung einfacher erweiterbar, da die bestehende Struktur nicht verändert werden muss. Außerdem können Frameworks einfacher extrahiert werden.

Im Programmentwurf wird dies beispielsweise in der „UserRepositoryImpl“ deutlich. Dieses implementiert das zugehörige Repository-Interface und fügt noch spezifische Funktionen hinzu. Das Interface kann somit auch in weiteren Klassen implementiert werden, wo es benötigt wird und daraufhin weitere spezifische Funktionen innerhalb der Klasse hinzufügen.

4.2.7 Controller

Das Controller-Prinzip besagt, dass in diesen Klassen einkommende Benutzereingaben verarbeitet werden. Sie dienen der Koordination zwischen Benutzeroberfläche und Businesslogik. Dabei werden die Benutzereingaben an andere Objekte delegiert, denn im Controller befindet sich keinerlei Businesslogik.

Unterschieden werden Controller in System Controller, der für alle Aktionen des Systems zuständig ist und nur für kleine Anwendungen praktikabel ist und in Use Case Controller. Use Case Controller werden für jeden einzelnen Use Case implementiert.

In diesem Programmentwurf wurden die Use Case Controller genutzt, die auf der Plugin-Schicht implementiert sind. Folgende Use Case Controller wurden implementiert:

- UserController
- KontoController
- TransactionController
- BankController

Diese Controller nehmen einkommende Benutzereingaben an und delegieren sie. Daraufhin erhalten sie ein Ergebnis und senden die zurück an den Client, der die Anfrage geschickt hat.

4.2.8 Pure Fabrication

Dieses Prinzip besagt, dass eine Klasse keinen Bezug zur Problemdomäne besitzt, wodurch eine Trennung von Technologie und Problemdomäne, sowie eine Kapselung von Algorithmen entsteht. Dadurch wird erreicht, dass Softwareteile auch außerhalb der Domäne wiederverwendet werden können. Außerdem wird durch die Kapselung von speziellen Funktionalitäten das High Cohesion-Prinzip begünstigt. Es sollte jedoch möglichst wenig vorkommen.

In diesem Programmentwurf wird dieses Prinzip einmal verwendet durch die Klassen, die eine Email an den Benutzer versendet, die die Transaktionen eines Kontos enthält. Diese Klassen können abgekapselt vom Rest der Problemdomäne und können leicht wiederverwendet werden. Diese Klassen befinden sich auf der Domain-Schicht im Ordner „Others“.

4.2.9 Protected Variations

Dieses Prinzip besagt, dass die Software durch die Kapselung verschiedener Implementierungen hinter einer einheitlichen Schnittstelle vor Variationen gesichert wird. Der Einfluss der Variabilität einer einzelnen Komponente soll nicht das Gesamtsystem betreffen. Gute Schutzmöglichkeiten hierfür sind Polymorphie und Delegation. Weitere Möglichkeiten sind Stylesheets im Webumfeld oder Spezifikationen von Schnittstellen.

Im Programmentwurf wird dies durch die Nutzung der Repository-Interfaces, die abgekapselt implementiert werden können, für neue Datenbanken.

4.3 DRY

DRY steht für „Don't repeat yourself“. Dieses Prinzip kann beispielsweise auf Datenbankschemata, Testpläne, Buildsysteme oder Dokumentationen angewendet werden. Es besagt, dass nur

eine Quelle der Wahrheit bestehen darf und alle anderen Quellen von dieser abgeleitet werden. Dies ist vergleichbar mit den Normalformen bei RDBMS.

Die Auswirkungen einer Modifikation eines Teils haben eine definierte Reichweite. Dabei sind jedoch keine unbeteiligten Teile betroffen und alle relevanten Teile ändern sich automatisch.

Es gibt drei Arten von Duplikationen, die durch das DRY-Prinzip behandelt werden:

Imposed Duplication

Imposed Duplications sind auferlegte Duplikationen. Dabei glaubt der Entwickler die Duplikation ist unumgänglich.

Inadverted Duplication

Inadverted Duplications sind versehentliche Duplikationen, die der Entwickler nicht bemerkt.

Impatient Duplication

Impatient Duplications sind ungeduldige Duplikationen, wobei der Entwickler zu faul ist, um diese zu beseitigen.

4.3.1 Anwendung des DRY-Prinzips

Bei der Erstellung dieses Programmentwurfs wurde von Anfang an darauf geachtet, dass kein duplizierter Code erstellt wird. Es soll eher bestehender Code an verschiedenen Stellen wiederverwendet werden. Duplizierter Code kann zu folgenden Problemen führen:

- Bei Änderungen kann es zu Inkonsistenzen kommen, die möglicherweise ein Sicherheits-Risiko darstellen. Es müssen nämlich daraufhin alle Stellen geändert werden, die den duplizierten Code enthalten, wobei es vorkommen kann, dass eine Stelle vergessen wird.
- Duplizierter Code macht den Programmcode länger und unübersichtlicher.
- Duplizierter Code führt zu duplizierten Bugs

Im Programmentwurf wurde beispielsweise der „CredentialsService“ verwendet, der dafür sorgt, dass Name, Email und Password den Konventionen entsprechen. Dieser Service wird an einer Stelle implementiert und kann daraufhin von mehreren Stellen aufgerufen werden, um Duplikationen zu verhindern. Dadurch sind auch Änderungen am Service an allen Stellen automatisch korrekt. Außerdem gibt es die Mapper der Adapter-Schicht, die an mehreren Stellen aufgerufen werden, um Konvertierungen auszuführen. Änderungen an den Datenstrukturen müssen nur innerhalb der Mapper geändert werden, damit sie überall korrekt sind.

Kapitel 5

Entwurfsmuster

5.1 Singleton-Entwurfsmuster

Das Singleton-Entwurfsmuster besagt, dass von einer Klasse systemweit nur ein Objekt existiert. Dies wird sichergestellt, indem innerhalb der Klasse, in der das Entwurfsmuster angewendet wird, eine `getInstance`-Methode existiert, die anstatt des Konstruktors von externen Klassen aufgerufen wird. Weiterhin existiert innerhalb der Klasse ein Attribut vom Typ dieser Klasse, das als Instanz dient. Die `getInstance`-Methode prüft, ob das instance-Attribut bereits existiert oder nicht. Sollte das instance-Attribut noch nicht existieren wird der Konstruktor aufgerufen und das erstellte Objekt in das instance-Attribut gespeichert und daraufhin zurückgegeben. Sollte das instance-Attribut existieren, wird das Attribut zurückgegeben.

5.2 Einsatz

Im Programmentwurf wurde dieses Entwurfsmuster beispielsweise in den Mappern genutzt. Diese werden nur einmal instanziiert und können daraufhin durch die `getInstance`-Methode von überall aufgerufen werden. So muss nicht jedesmal, wenn ein Mapper benötigt wird, eine neue Instanz des Mappers erzeugt werden, wodurch auch die Systemressourcen geschont werden. Außerdem wird eine Instanz auch nur dann angelegt, wenn sie benötigt wird.

5.3 Vorher-Nachher-Vergleich

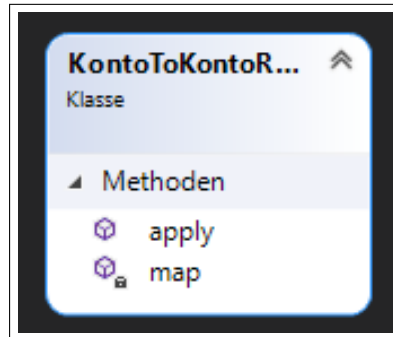


Abbildung 5.1: Singleton Vorher

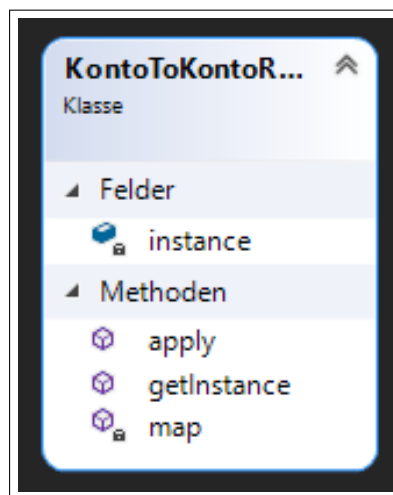


Abbildung 5.2: Singleton Nachher

Kapitel 6

Refactoring

6.1 Code Smell 1 - Lange Methode

Der erste Code Smell, der identifiziert wurde, ist langer übersichtlicher Code, der dafür genutzt wurde, um Transaktionen in einer Mail aufzulisten und an den eingeloggten Benutzer zu versenden. Diese Methode ist die „getTransactionsAsMail“-Methode, die sich in der „TransactionRepositoryImpl“-Klasse befindet. In dieser Methode werden zum einen alle Transaktionen für ein Konto abgerufen und zum anderen ein Mail-Objekt aufgrund der abgefragten Transaktionen erstellt. Es wurde, um die Übersichtlichkeit zu verbessern, die Funktion, die das Mail-Objekt erstellt, in eine neue Methode extrahiert. In 6.1 ist die unübersichtliche Methode zu sehen. In 6.2 ist die Methode zu sehen nach dem Refactoring.

```
2 Verweise
public async Task<ActionResult> getTransactionsAsMail(int kontoid)
{
    var konto = await KontoRepositoryImpl.FindById(kontoid);
    var transactions (List<TransactionAggregate>) = await getAllTransactions(kontoid);
    var owner (UserEntity) = await UserRepositoryImpl.FindById(konto.UserId);

    var mailRequest = new MailRequest();
    mailRequest.ToEmail = owner.Email;
    mailRequest.Subject = "Transactions of konto with id: " + kontoid;
    mailRequest.Body += "Konto: " + konto.Id + " contains following amount of money: " + konto.Kontostand + "\n";
    foreach(var transaction in transactions)
    {
        mailRequest.Body += transaction.Date.ToString() + ": " + transaction.Id.ToString() + ": " +
            transaction.TransactionInfo.getBetrag() + "€ from Konto: " +
            transaction.TransactionInfo.getKontoidSender() +
            " to Konto:" + transaction.TransactionInfo.getKontoIdEmpfänger() + "\n";
    }

    try
    {
        await MailService.SendEmailAsync(mailRequest);
        return new OkObjectResult("Email sent");
    }
    catch(Exception ex)
    {
        throw;
    }
}
```

Abbildung 6.1: Long Method

6.2 Code Smell 2 - Duplikate im Code

Der zweite Code Smell, der identifiziert wurde, ist ein Duplikat im Code. Es wurde zwar versucht nach dem DRY-Prinzip zu arbeiten, dies wurde jedoch nicht hundertprozentig umgesetzt. Später lies sich beispielsweise in der „UserController“-Klasse ein Duplikat finden, in welchem eine Liste

```
2 Verweise
public async Task<IActionResult> getTransactionsAsMail(int kontoid)
{
    var konto = await KontoRepositoryImpl.findById(kontoid);
    var transactions (List<TransactionAggregate>) = await getAllTransactions(kontoid);
    var owner (UserEntity) = await UserRepositoryImpl.findById(konto.UserId);

    var mailRequest = CreateMailRequest(kontoid, owner, konto, transactions);

    try
    {
        await MailService.SendEmailAsync(mailRequest);
        return new OkObjectResult("Email sent");
    }
    catch (Exception ex)
    {
        throw;
    }
}
```

Abbildung 6.2: Long Method Refactored

an UserEntities in eine Liste von User-Objekten durch einen den Mapper konvertiert wurde. Diese Funktion wurde in mehreren Methoden dieser Klasse verwendet.

Um das Duplikat zu verhindern, wurde diese Funktion in eine neue Methode innerhalb des Mappers ausgelagert, durch welche nun eine Liste an UserEntities in eine Liste an User-Objekten umgewandelt wird. Diese Methode wird von mehreren Stellen aufgerufen. In 6.3 wird der Code, der dupliziert vorkam, in einer der Methoden gezeigt. In 6.4 wird gezeigt, wie die Funktion

```
// GET: api/Konto
[HttpGet]
0 Verweise
public async Task<List<KontoResource>> GetKonten()
{
    var konten (List<KontoEntity>) = await _kontoRepositoryImpl.findAllKonten();
    var kontolist = new List<KontoResource>();
    foreach (var konto in konten)
    {
        kontolist.Add(item: new KontoResource(konto.Id, konto.UserId));
    }

    return kontolist;
}
```

Abbildung 6.3: Duplicate Code

ausgelagert wurde und durch den Mapper aufgerufen wird. Diese Funktion wird noch an weiteren Stellen innerhalb des Controllers aufgerufen.


```
// GET: api/Konto
[HttpGet]
// Verweise
public async Task<List<KontoResource>> GetKonten()
{
    var konten (List<KontoEntity>) = await _kontoRepositoryImpl.findAllKonten();
    var kontolist (List<KontoResource>) = KontoToKontoResourceMapper.convertToKontoResourceList(konten);
    return kontolist;
}
```

Abbildung 6.4: Duplicate Code Refactored

Kapitel 7

Unit Tests

In diesem Programmentwurf wurden 11 Unit Tests erstellt.

7.1 Mocking

Im Projekt „Programmentwurf_BankingAp_Mock_Tests“ wurde ein Unit Test erstellt, der die Datenbank mockt. Dafür wurde „IBankingContext“ gemockt und ein Unit Test erstellt, der die Methode „findById“ testet.

7.2 Unit Tests

Weiterhin wurde für die Methoden der verschiedenen Mapper-Klassen Unit tests erstellt, um deren Funktion zu testen.

7.3 Ausführen der Tests

Um die Tests auszuführen, wird in Visual Studio der Test-Explorer geöffnet und von dort aus die Tests ausgeführt.

Um in Visual Studio Code die Tests auszuführen wird in das jeweilige Projekt navigiert und über die Terminal-Konsole der Befehl „dotnet test“ eingegeben.