

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	5
2.2.4	Repositories	6
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 Injection Principle	13

Abbildungsverzeichnis

2.1	AdressVO	5
2.2	KontoEntity	6
2.3	BankAggregate	6
2.4	BankRepository	7
2.5	Ausschnitt aus CredentialsService	8
3.1	Schichten der Clean Architecture	9

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

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:

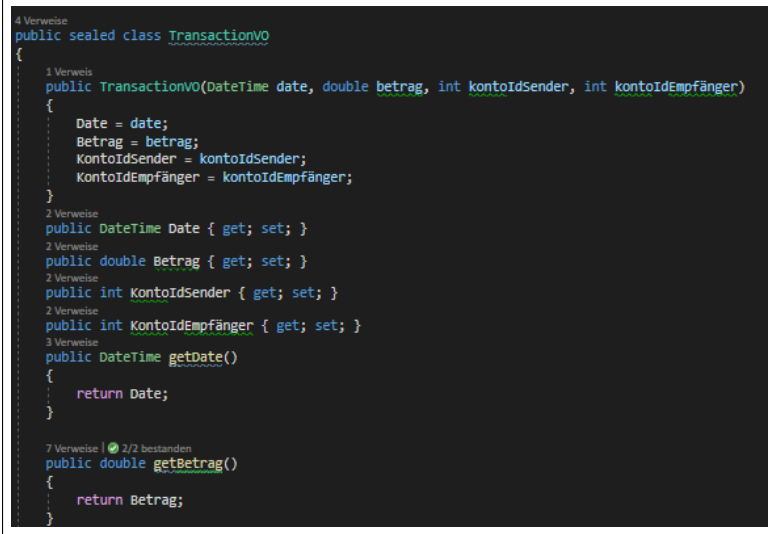
2.2.2 Entities

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
- Banken



```

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

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:

2.2.3 Aggregates

Aggregate gruppieren Entities und Value Objects zu gemeinsam verwalteten Einheiten. Auch wenn ein Aggregat nur aus einer Entity bestehen würde, wird dafür ein Aggregat angelegt. Aggregate helfen dabei die Komplexität der Beziehungen zwischen Objekten zu reduzieren. Die Entity im Aggregat dient als Aggregate Root Entity.

In diesem Projekt wurde neben den Aggregaten für die einzelnen Entitäten auch ein Aggregat erstellt, das das Value Object für eine Transaktion hält. Folgende Aggregate sind erstellt worden:

- UserAggregate
- KontoAggregate
- 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.

UserAggregate und KontoAggregate bestehen nur aus den Entitäten für den User und das Konto. Das BankAggregate fasst die BankEntity und die dazugehörige Adresse zusammen. Das TransactionAggregate hält das Value Object für die Transaktionsinfos und gibt jeder Transaktion

```

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

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.

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

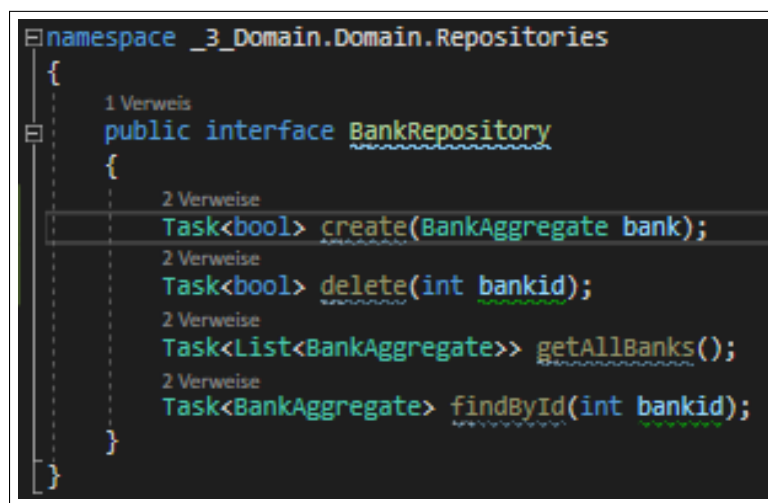
Es wurden Repositories für User, Konto, Bank und Transaktion erstellt. Sie vermitteln zwischen der Domäne und dem Datenmodell. Innerhalb der Repositories werden Methoden zur Verfügung

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:



```
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

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 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\\x0b-\\x0f\\x11-\\x1f\\x21-\\x5a\\x5c-\\x7f]|\\\\[\\x01-\\x0f\\x11-\\x1f\\x21-\\x5a\\x5c-\\x7f])*\")@(?!(?:[a-z0-9]{4,})\\.)(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.){1,63}[a-z0-9](?:[a-z0-9-]*[a-z0-9])?";

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

    4 Verweise
    public bool isNameValid(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

- 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 vorallem 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 Injection Principle

Durch das Dependency Injection 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 werden 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 Klassen, die die Repository-Interfaces und Domain Service-Interfaces implementieren, beim initialen Aufbau der Anwendung instanziiert, wenn sie benötigt werden. In der „Startup.cs“ wird festgelegt, welche Klassen instanziiert werden sollen, wenn sie beim Start benötigt werden.