Advanced Software Engeeniring
Budget Planner App

Imke Schmidt (1799839)

31.05.2021

Inhaltsverzeichnis

1. *Einleitung*

Das Ziel dieses Software Projektes ist die Entwicklung einer “Budget Planner App“.

1.1 Die Budget Planner App

Mit der Zeit des online Bankings wird es immer schwieriger den Überblick über die eigenen Finanzen zu behalten und im Nachhinein noch zu wissen, wie viel man von seinem Geld für Freizeitaktivitäten, Essen oder andere Kategorien ausgegeben hat.

Die Budget Planner App soll einen Überblick über die eigenen Einnahmen und Ausgaben zu behalten. Dabei soll auch ein einfacher Überblick ermöglicht werden wie viel man für welche LLebenskategorienäusgegeben hat.

Die App soll folgende Möglichkeiten bieten:

- Einnahmen eintragen
- Ausgaben eintragen
- Ausgaben in Kategorien einteilen
- Dashboard für Einnahmen und Ausgaben
- Einträge löschen

1.2 GitHub Repository

`https://github.com/FlyingBabYpsilon/AdvancedSE`

1.3 Anwendung laufen lassen

Um die Anwendung auf dem Rechner laufen zu lassen muss folgendes getan werden:

1. Clone Git: `https://github.com/FlyingBabYpsilon/AdvancedSE` / gh repo clone FlyingBabYpsilon/AdvancedSE
2. npm install -g ionic
3. npm install -g angular
4. cd Folder
5. ionic serve –lab

1.4 Über die Dokumentation

Diese Dokumentation befasst sich mit den Programmiertechnischen Themen:

- Domain Driven Design
- Clean Architecture
- Programmin principles
- Refactoring
- Entwurfsmuster

Dabei soll die Dokumentation diese Themen anhand der "Budget Planner Appßeigen und dabei zeigen, inwiefern die Inhalte der Vorlesung Ädvanced Software Engineeringümgesetzt wurden. Dabei soll deutlich werden dass die Inhalte der Vorlesung verstanden wurden und angewendet werden können.

2. *Domain Driven Design*

2.1 Analyse der Ubiquitous Language

Die Ubiquitous Language wird in einem begrenzten Kontext modelliert, in dem die Begriffe und Konzepte der Geschäftsdomäne identifiziert sind und es sollte keine Mehrdeutigkeit geben. Außerdem wird sie verwendet um eine gemeinsame Sprache für das Team, die Entwickler, die Domänenexperten und andere Beteiligte zu schaffen. Die Auflistung enthält die Begriffe der Ubiquitous Language welche bei der Analyse herausgearbeitet wurden:

- **income — einkommen:** Ein Einkommen beschreibt das Geld welches der Nutzer als plus auf seinem Konto oder in Bar bekommt
- **incomeAmt:** IncomeAmt ist die Summe des Einkommens
- **incomeCat:** IncomeCat ist die Kategorie des Einkommens wie z.B.: Gehalt, Taschengeld oder "Schwarzgeld — Sonstige Einnahmen"
- **incomeDate:** Das IncomeDate ist das Datum an dem der Nutzer das Geld erhalten hat.
- **spending — ausgaben:** Die Ausgaben beschreiben das Geld welches der Nutzer als minus auf seinem Konto oder in Bar verbucht
- **spendAmt:** Der spendAmt ist die Summe welche der Nutzer ausgegeben hat
- **spendCat:** Die SpendCat ist die Kategorie für die der Nutzer das Geld ausgegeben hat.
- **spendDesc:** Die SpendDesc bietet dem Nutzer eine Möglichkeit eine kurze Beschreibung zu der Ausgabe hinzuzufügen.
- **spendDate:** Das SpendDate ist das Datum an dem der Nutzer das Geld ausgegeben hat.

2.2 Analyse und Begründung der verwendeten Muster

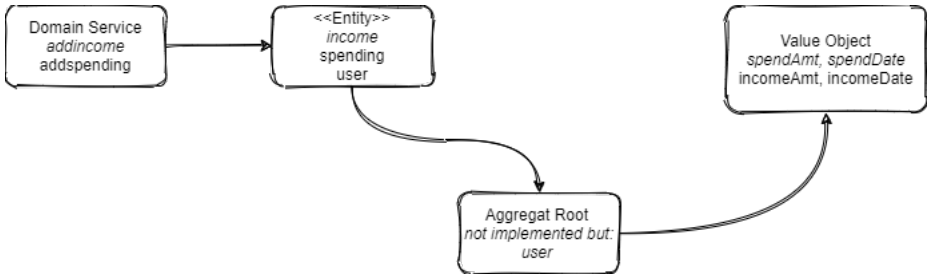

Abbildung 2.1: verwendete Muster

2.2.1 Value Objects

Value Objects sind in dieser Anwendung die Objekte **spendAmt**, **incomeAmt**, **incomeDate** und **spendDate**.

Diese Objekte erfüllen die Definition nach Eric Evans und deren drei Eigenschaften:

- keine Identität
- sind nicht veränderbar
- werden immer in einem gültigen Zustand erzeugt

2.2.2 Entites

Entitäten zeichnen sich dadurch aus, dass sie innerhalb der Domäne eine eigene Identität und veränderliche Eigenschaften haben sowie einen Lebenszyklus haben. Sie repräsentieren ein wahrnehmbares Objekt der Domäne. In der Budget Planner App sind die Entitäten der User, Income und Spending.

2.2.3 Aggregates

Aggregate sind im DDD “zusammenschlüsse“von Entitäten. Dabei dient eine dieser Entitäten als Würzel, als das sogenannte Aggregate Root. Die anderen Entitäten sind dann nur über dieses Wurzelobjekt erreichbar. Als Aggregate Root wäre in der Budget Planner App die Entität user/nutzer und die Entitäten User, Income und Spending zusammenzufassen, so dass die Entitäten Income und Spending nur über den User erreichbar sind.

2.2.4 Repositories

Idealerweise gäbe es für jedes Aggregat ein Respository, da jedoch die Entitäten nicht zu einem Aggregat zusammengefasst wurden gibt es nicht ein Repository, sondern drei. Für jede Entität gibt es ein Repository, da die Entitäten wie ein Aggregat behandelt werden.

2.2.5 Domain Services

Als Domain Services sind in der Applikation zum einen der Income Service und der Spending Service vorhanden.

3. *Clean Architektur*

Die Clean Architektur gibt vorgaben an, wie die Architektur der Anwendung aufgebaut werden muss um “clean“zu sein. Dabei ist vor allem der Grundsatz wichtig, dass alle Abhängigkeiten von außen nach innen zeigen. Dementsprechend wurde in diesem Kapitel die Schichtenarchitektur geplant. Innerhalb des Projektes befinden sich die einzelnen Schichten in Verschiedenen Ordnern.

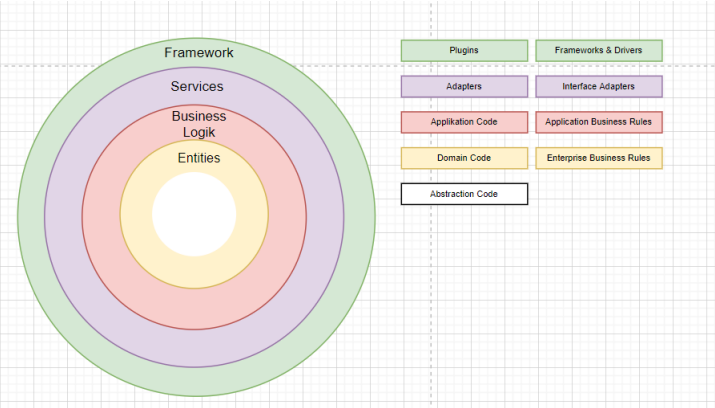

Abbildung 3.1: Clean Architecture Model

3.1 Schichtarchitektur

Auf der Abbildung sind die einzelnen Schichten im Verzeichnis des Projektes zu sehen.

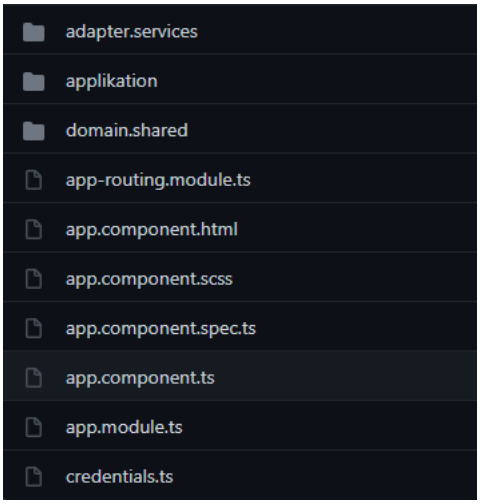

Abbildung 3.2: clean-architecture

3.1.1 Schicht 4 - Abstraction Code

Die Abstraction Code Schicht enthält Domänen übergreifendes Wissen, ändert sich selten bzw. nie, Grundbausteine die nicht domänenspezifisch sind.
Eine explizite Anlage als eigene Schicht ist häufig nicht notwendig, nur anlegen wenn wirklich benötigt.

3.1.2 Schicht 3 - Domain Code

Die Domain Code Schicht enthält v.a. Entities und implementiert die organisationsweite Business Logik. Dabei handelt es sich um den inneren Kern der Anwendung welcher sich am seltensten Ändern sollte. Aggregate(Entities und Value Objects), Domain Services, Repositories und Factories sind typische Strukturen im Domain Code.
In dieser Schicht befinden sich die Entities User, Shared und Income.

3.1.3 Schicht 2 - Applikation Code

Die Applikation Code Schicht enthält die Anwendungsfälle und implementiert die anwendungsspezifische Geschäftslogik. Diese Schicht steuert den Fluss der Daten und Aktionen zu den Elementen des Domain Codes.

In dieser Schicht befinden sich die Business Logik und somit die Formulare und die UI Oberfläche.

3.1.4 Schicht 1 -Adapters

Die Adapter Schicht vermittelt Aufrufe und Daten an die inneren Schichten. In dieser Schicht befinden sich die Services der Anwendung.

3.1.5 Schicht 0 - Plugins

Die äußerste Schicht ist die Plugin Schicht enthält keine Anwendungslogik, sondern greift hauptsächlich auf die Adapter zu und enthält Frameworks, Datentransportmittel und andere Werkzeuge. Diese Schicht wurde nicht in der Anwendung explizit umgesetzt.

4. *Programming Principles*

4.1 SOLID

Solid sind fünf Prinzipien zum Entwurf und Entwicklung einer Software. Dabei sorgen diese Prinzipien für eine einfachere Erweiterbarkeit, eine bessere Wiederverwendbarkeit, eine einfacherer Wartbarkeit und eine bessere Verständlichkeit der Anwendung.

4.1.1 Single Responsbility Principle (SRP)

Bei dem Single Responsibility Principle sollte jede Klasse genau eine Aufgabe erfüllen. In der Anwendung wurde das Single Resposibility Principle z.B.: für den Use Case Income Verwendet. Innerhalb des Use Cases soll der Nutzer in der Lage eine neue Einnahme hinzuzufügen und alle Einnahmen in einer Übersicht zu sehen. Die verschiedenen Use Cases sind in einzelnen Klassen, so dass jede Klasse nur eine Aufgabe hat.

```
export class IncomePage implements OnInit{

  public incomeList: Observable<Income[]>

  constructor(private incomeService : IncomeService) {}

  ngOnInit() {
    this.incomeList= this.incomeService.getIncomeList();
  }
}
```

Listing 4.1: IncomePage

```
export class AddincomePage {
  public createIncomeForm: FormGroup;
  constructor(
    public loadingCtrl: LoadingController,
    public alertCtrl: AlertController,
    private incomeService: IncomeService,
```

```
    formBuilder: FormBuilder,
    private nav: NavController
  ) {
    this.createIncomeForm = formBuilder.group({
      incomeAmt: ['', Validators.required],
      incomeCat: ['', Validators.required],
      incomeDate: ['', Validators.required],
      incomeDesc: ['', Validators.required],
    });
  }


  async createIncome() {
    const loading = await this.loadingCtrl.create();

    const incomeAmt = this.createIncomeForm.value.incomeAmt;
    const incomeCat = this.createIncomeForm.value.incomeCat;
    const incomeDate = this.createIncomeForm.value.incomeDate;
    const incomeDesc = this.createIncomeForm.value.incomeDesc;

    this.incomeService
      .createIncome(incomeAmt, incomeCat, incomeDate, incomeDesc)
      .then(
        () => {
          loading.dismiss().then(() => {
            this.nav.navigateBack(['tabs/income']);
          });
        },
        error => {
          loading.dismiss().then(() => {
            console.error(error);
            this.nav.navigateBack(['tabs/income']);
          });
        }
      );

    return await loading.present();
  }
}
```

Listing 4.2: addIncome

4.1.2 Open Closed

Bei dem Open Closed Principle (OCP) geht es darum, neue Features zu einer Anwendung hinzuzufügen, ohne das System zu verändern. Bei dem Open/Closed Principle geht es darum, dass Module zwar offen für Erweiterung sind aber geschlossen für Änderung. Das OCP wird oftmals durch die Verwendung von Interfaces oder Vererbung umgesetzt. Wenn man sich das OCP im Bezug auf die Anwendung anschaut, dann eignet sich dafür besonders gut die Klasse "User". Denn wenn man die Anwendung erweitern möchte, dann kann ein "PremiumUser"von dem User erben, aber gleichzeitig noch um weitere Funktionalitäten erweitert werden kann ohne das die Ursprüngliche "User"Klasse verändert werden muss. Dadurch könnten einem Premium User weitere Rechte oder Einstellungsmöglichkeiten zugewiesen werden.

4.1.3 Liskov Substitution

Bei der Liskov Substituion (LSP) soll eine abgeleitet Klasse an jeder beliebigen Stelle ihre Basisklasse ersetzen können. Dabei darf es nicht zu ungewünschten Nebeneffekten kommen. Die Voraussetzung für das LSP ist der Einsatz von Vererbung. Wenn man sich das LSP im Zusammenhang mit der Anwendung anschaut, dann fällt auf, dass in diesem Projekt keine Vererbung in diesem Sinne beinhaltet. Dadurch kam das LSP nicht konkret zum Einsatz, allerdings kann dadurch auch nicht dagegen verstoßen werden. Jedoch kann deshalb das LSP auch nicht anhand eines Beispiels aufgezeigt werden.

4.1.4 Interface Segregation

Bei dem Interface Segregation Principle geht es darum, dass ein zu großes Interface die Wahrscheinlichkeit erhöht, dass man bei Änderungen an Schnittstellen auch sehr viel Subklassen anpassen muss. Allgemein lässt sich auch sagen Kundenspezifische Schnittstellen sind besser als eine allgemeine Schnittstelle. Subklassen und Schnittstellen sollten also nur die Methoden beinhalten, welche wirklich benötigt werden. Schaut man sich das Interface Segregation Principle im Bezug auf die Anwendung an, stellt man fest das die Anwendung so aufgebaut wurde, dass kaum Subklassen vorhanden sind und die verschiedenen Interfaces so implementiert worden sind, das sie sich alle nur um ihre Aufgabe kümmern und somit auch nur die Methoden benötigen, welche für die Funktionalität des Interfaces benötigt werden. Um das ganze nochmal an einem Beispiel zu betrachten eignen sich die Interfaces IncomePage, AddIncome und Income-Detail Page. Die IncomePage klasse ist nur für das Anzeigen der verschiedenen Datensätze zuständig.

```
1 export class IncomePage implements OnInit{
```

```
2
3   public incomeList: Observable<Income[]>
4
5   constructor(private incomeService : IncomeService) {}
6
7   ngOnInit() {
8     this.incomeList= this.incomeService.getIncomeList();
9   }
10 }
```

Listing 4.3: income.page.ts

Die AddIncome Klasse ist dafür dann wiederum dafür zuständig, dass ein neues Income als Datensatz erzeugt wird und in die Datenbank geschrieben wird.

```
1  export class AddincomePage {
2    public createIncomeForm: FormGroup;
3    constructor(
4      public loadingCtrl: LoadingController,
5      public alertCtrl: AlertController,
6      private incomeService: IncomeService,
7      formBuilder: FormBuilder,
8      private nav: NavController
9    ) {
10     this.createIncomeForm = formBuilder.group({
11       incomeAmt: ['', Validators.required],
12       incomeCat: ['', Validators.required],
13       incomeDate: ['', Validators.required],
14       incomeDesc: ['', Validators.required],
15     });
16   }
17
18
19   async createIncome() {
20     const loading = await this.loadingCtrl.create();
21
22     const incomeAmt = this.createIncomeForm.value.incomeAmt;
23     const incomeCat = this.createIncomeForm.value.incomeCat;
24     const incomeDate = this.createIncomeForm.value.incomeDate;
25     const incomeDesc = this.createIncomeForm.value.incomeDesc;
26
27     this.incomeService
28       .createIncome(incomeAmt, incomeCat, incomeDate, incomeDesc)
29       .then(
30         () => {
```

```
          loading.dismiss().then(() => {
            this.nav.navigateBack(['tabs/income']);
          });
        },
        error => {
          loading.dismiss().then(() => {
            console.error(error);
            this.nav.navigateBack(['tabs/income']);
          });
        }
      );
    return await loading.present();
  }
}
```

Listing 4.4: addincome.page.ts

Die letzte Klasse aus diesem Beispiel, die Income-Detail Page ist wiederum nur dafür zuständig, die genaueren Informationen zu einem Datensatz von der Income Page anzuzeigen und diesen bei bedarf zu löschen.

```
export class IncomeDetailsPage implements OnInit {
  public income: Income;

  constructor(
    private incomeService: IncomeService,
    private route: ActivatedRoute,
    private alertController: AlertController,
    private nav: NavController
  ) { }

  ngOnInit() {
    const incomeId: string = this.route.snapshot.paramMap.get('incomeId');
    this.incomeService.getIncomeDetail(incomeId).subscribe(income => {
      this.income = income;
    });
  }

  async deleteIncome(incomeId: string): Promise<void> {
    const alert = await this.alertController.create({
      message: `Are you sure you want to delete this Income?`,
      buttons: [
        {
          text: 'Cancel',
          role: 'cancel',
```

```
          handler: blah => {
            console.log('Confirm Cancel: blah');
          },
        },
        {
          text: 'Okay',
          handler: () => {
            this.incomeService.deleteIncome(incomeId).then(() => {
              this.nav.navigateBack(['tabs/income']);
            });
          },
        },
      ],
    });
    await alert.present();
  }
}
```

Listing 4.5: income-detail-page.ts

4.1.5 Dependency Inversion

Das Dependency Inversion Principle (DIP) besagt das High Level Module nicht von Low Level Modulen abhängen sollten und das Abhängigkeiten nicht auf die Objekte sondern auf Interdaces bestehen sollten. Wenn man die Umsetzung das DIP innerhalb der Anwendung betrachtet sieht man, dass die Klassen welche sich in der Applikationsschicht befinden, abhängig von den Serviceklassen sind. Wenn man dies an einem Explitien Beispiel betrachten möchte eignet sich dafür besonders die Abhängigkeitsbeziehung zwischen der "IncomePage"und dem "IncomeService". Die IncomePage befindet sich innerhalb der Applikations Schicht und ist abhängig von dem IncomeService, welcher sich auf der Adapter Schicht befindet.

```
export class IncomePage implements OnInit{

  public incomeList: Observable<Income[]>

  constructor(private incomeService : IncomeService) {}

  ngOnInit() {
    this.incomeList= this.incomeService.getIncomeList();
  }
}
```

Listing 4.6: IncomePage

4.2 GRASP

GRASP bietet weitere Muster/Prinzipien zur Regelung der Zuständigkeit. Grasp bedeutet General Reponsibilty Assignment Software Pattern und überschneidet sich mit den vorherigen SOLID Prinzipien.

4.2.1 Low Coupling

Bei Low Coupling geht es darum eine möglichst geringe Kopplung zu erreichen, da dies eines der Hauptziele eines guten Designs ist. Dabei bezeichnet die Kopplung den Grad der Abhängigkeit zwischen zwei oder mehr Dingen. Dabei geht es um eine leichte Anpassbarkeit, eine gute Verständlichkeit der Klasse da der Kontext nicht betrachtet werden muss. Eine lose Kopplung kann z.B. durch die Verwendung von Interfaces umgesetzt werden. Wenn man das Low Coupling in der Anwendung betrachtet, dann findet man das Low Coupling grad an den Stellen wie zwischen den Klassen der Applikations Schicht und den Services. Die Klasse "IncomePage"ist abhängig von dem IncomeService. Man kann also leicht Änderungen an dem Income Service vornehmen, ohne dass es zu Problemen kommt.

4.2.2 High Cohesion

High Cohesion, bzw zu deutsch "Hohe Kohäsion"begrenzt die Komplexität des Gesamten Systems indem man die Klassen gut überschaubar organisiert und hält. Dabei misst die Kohäsion den inneren Zusammenhalt einer Klasse und wie Eng die Methoden und Attribute einer Klasse zusammenarbeiten. Das bedeutet, das eine Klasse im Idealfall nur Methoden anwendet, welche wirklich benötigt werden. In der Anwendung wurde darauf geachtet, das jede Klasse auch nur die Methoden beinhaltet, die sie auch wirklich benötigt. Um das ganze noch an einem Beispiel zu zeigen eignet sich die Instanz SpendingPage. Innerhalb dieser Instanz soll nur eine Übersicht der Datensätze angezeigt werden, welche in der Datenbank als "spending"gespeichert sind. Das Erstellen eines neuen Datensatzes oder das Anzeigen von Detail Informationen liegt in dem Zuständigkeitsbereich einer anderen Klasse.

```
1 export class SpendingPage {
2
3   public spendingList: Observable<Spending[]>
4
5   constructor(private spendingService : SpendingService) {}
6
7   ngOnInit() {
8     this.spendingList= this.spendingService.getSpendingList();
```

```
9     }
10 }
```

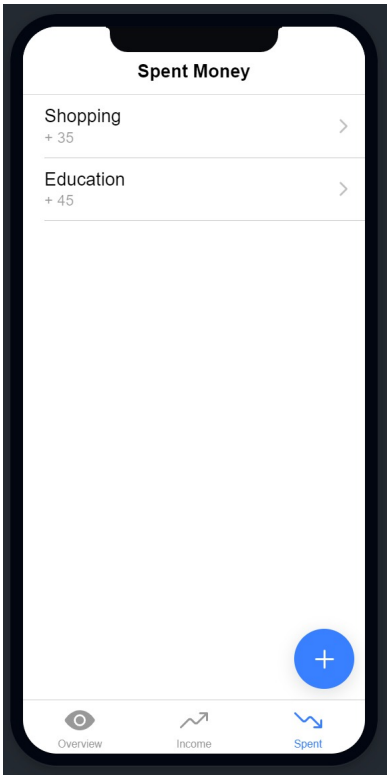

Abbildung 4.1: Spending

4.2.3 Information Expert

Der Information Expert erhält die Verantwortlichkeit für eine Aufgabe über die er das meiste Wissen über die Aufgabe mitbringt. Wenn man sich innerhalb der Anwendung die Aufgabe des “Income“anschaut wird recht schnell klar, dass der Information Expert in diesem Fall der “IncomeService“ist, da er sowohl Informationen über das Speichern, Lesen und Löschen der Income Datensätze hat.

4.2.4 Creator

Der Creator gibt vor, wer für die Erzeugung einer Instanz zuständig ist. Anders als beim Information Expert handelt es sich hier nicht nur um eine Aufgabe sondern um die Erzeugung einer Instanz. Wenn man sich in der Anwendung Beispielhaft die Aufgabe “Income erzeugen“anschaut, dann gibt es zwei verschiedene Klassen, welche dabei beteiligt sind. Dann

stellt sich nur noch die Frage, welche der Creator ist. Die eine beteiligte Klasse ist die Klasse "AddincomePage"

```
export class AddincomePage {
  public createIncomeForm: FormGroup;
  constructor(
    public loadingCtrl: LoadingController,
    public alertCtrl: AlertController,
    private incomeService: IncomeService,
    formBuilder: FormBuilder,
    private nav: NavController
  ) {
    this.createIncomeForm = formBuilder.group({
      incomeAmt: ['', Validators.required],
      incomeCat: ['', Validators.required],
      incomeDate: ['', Validators.required],
      incomeDesc: ['', Validators.required],
    });
  }


  async createIncome() {
    const loading = await this.loadingCtrl.create();

    const incomeAmt = this.createIncomeForm.value.incomeAmt;
    const incomeCat = this.createIncomeForm.value.incomeCat;
    const incomeDate = this.createIncomeForm.value.incomeDate;
    const incomeDesc = this.createIncomeForm.value.incomeDesc;

    this.incomeService
      .createIncome(incomeAmt, incomeCat, incomeDate, incomeDesc)
      .then(
        () => {
          loading.dismiss().then(() => {
            this.nav.navigateBack(['tabs/income']);
          });
        },
        error => {
          loading.dismiss().then(() => {
            console.error(error);
            this.nav.navigateBack(['tabs/income']);
          });
        }
      );
```

```
42
43      return await loading.present();
44    }
45
46  }
```

Listing 4.7: addincome

Diese Klasse überprüft die Daten, welche in das Formular eingegeben wurden und prüft ob diese valide sind. Die zweite Beteiligte Klasse ist die Klasse “IncomeService“

```
1  export class IncomeService {
2
3    constructor(public firestore: AngularFirestore) {}
4
5    createIncome(
6      incomeAmt: string,
7      incomeCat: string,
8      incomeDate: string,
9      incomeDesc: string
10   ): Promise<void> {
11   const incomeId = this.firestore.createId();
12
13   return this.firestore.doc(`incomeList/${incomeId}`).set({
14     incomeId,
15     incomeAmt,
16     incomeCat,
17     incomeDate,
18     incomeDesc,
19   });
20 }
21
22 getIncomeList(): Observable<Income[]> {
23   return this.firestore.collection<Income>(`incomeList`).valueChanges();
24 }
25
26 getIncomeDetail(incomeId: string): Observable<Income> {
27   return this.firestore.collection('incomeList').doc<Income>(String(
       incomeId)).valueChanges();
28 }
29
30 deleteIncome(incomeId: string): Promise<void> {
31   return this.firestore.doc(`incomeList/${incomeId}`).delete();
32 }
```

```
33 }
```

Listing 4.8: IncomeService

Diese Klasse fügt dem Datensatz noch eine ID hinzu und schreibt ihn dann in die Datenbank. Der Creator ist in diesem Fall allerdings die Klasse “AddIncomePage“da sie das Objekt erzeugt.

4.2.5 Indirection

Bei Indirection kommunizieren zwei Einheiten über einen Vermittler, anstatt dass sie direkt miteinander kommunizieren. Dies ist z.B.: bei dem SpendingService und der Spending-detail Page zu sehen. Das UML zeigt, dass sie nicht direkt miteinander kommunizieren sondern das die Detail Seite die spendId zurück gibt und der Spending Service die Methode um das Spending zu löschen.

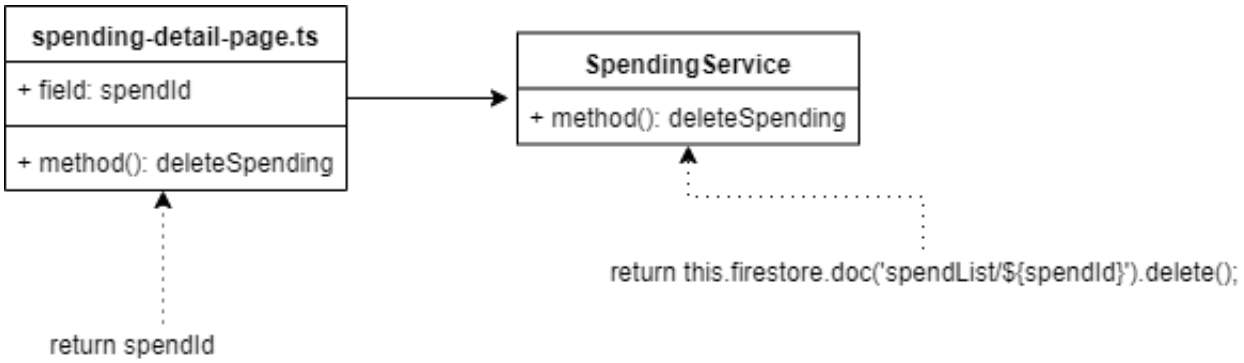

Abbildung 4.2: Indirection Uml

4.2.6 Polymorphism

Nach dem Prinzip der Polymorphie wird die Verantwortung für die Definition der typabhängigen Variation von Verhaltensweisen dem Typ zugewiesen, für den diese Variation erfolgt. Dies wird durch polymorphe Operationen erreicht. Der Benutzer des Typs sollte polymorphe Operationen anstelle von expliziten Verzweigungen basierend auf dem Typ verwenden. Oftmals werden Methoden des Grundtyps oder der Schnittstelle überschrieben werden. In der Anwendung sind keine Methoden vorhanden, welche überschrieben werden.

4.2.7 Controller

Der Controller beinhaltet das Domänenwissen und definiert, wer die für eine nicht benutzeroberflächen klasse bestimmenden Systemereignisse verarbeitet. Bei dem Controller handelt es

sich außerdem um die erste Schnittstelle nach der GUI. Ein Use Case Controller verarbeitet alle Events eines spezifischen Use Cases und ist in der Anwendung z.B. Als der authentication-service vorhanden. Dieser erzeugt einen neuen User und ermöglicht den LogIn / LogOut in der Anwendung.

```
export class AuthenticationService {
  userData: any;

  constructor(
    public afStore: AngularFirestore,
    public ngFireAuth: AngularFireAuth,
    public router: Router,
    public ngZone: NgZone
    ) {
    this.ngFireAuth.authState.subscribe(user => {
      if (user) {
        this.userData = user;
        localStorage.setItem('user', JSON.stringify(this.userData));
        JSON.parse(localStorage.getItem('user'));
      } else {
        localStorage.setItem('user', null);
        JSON.parse(localStorage.getItem('user'));
      }
    })
  }

  // Login in with email/password
  SignIn(email, password) {
    return this.ngFireAuth.signInWithEmailAndPassword(email, password)
  }

  // Register user with email/password
  RegisterUser(email, password) {
    return this.ngFireAuth.createUserWithEmailAndPassword(email, password)
  }

  // Email verification when new user register
  SendVerificationMail() {
    return this.ngFireAuth.currentUser.then(u => u.sendEmailVerification()
   )
    .then(() => {
    this.router.navigate(['verify-email']);
    })
    }
```

```
39
40  // Recover password
41  PasswordRecover(passwordResetEmail) {
42    return this.ngFireAuth.sendPasswordResetEmail(passwordResetEmail)
43    .then(() => {
44      window.alert('Password reset email has been sent, please check your
     inbox.');
45    }).catch((error) => {
46      window.alert(error)
47    })
48  }
49
50  // Returns true when user is looged in
51  get isLoggedIn(): boolean {
52    const user = JSON.parse(localStorage.getItem('user'));
53    return (user !== null && user.emailVerified !== false) ? true : false;
54  }
55
56  // Returns true when user's email is verified
57  get isEmailVerified(): boolean {
58    const user = JSON.parse(localStorage.getItem('user'));
59    return (user.emailVerified !== false) ? true : false;
60  }
61
62
63
64  // Auth providers
65  AuthLogin(provider) {
66    return this.ngFireAuth.signInWithPopup(provider)
67    .then((result) => {
68       this.ngZone.run(() => {
69          this.router.navigate(['dashboard']);
70        })
71      this.SetUserData(result.user);
72    }).catch((error) => {
73      window.alert(error)
74    })
75  }
76
77  // Store user in localStorage
78  SetUserData(user) {
79    const userRef: AngularFirestoreDocument<any> = this.afStore.doc(`users
     /${user.uid}`);
80    const userData: User = {
```

```
81        uid: user.uid,
82        email: user.email,
83        displayName: user.displayName,
84        photoURL: user.photoURL,
85        emailVerified: user.emailVerified
86      }
87      return userRef.set(userData, {
88        merge: true
89      })
90    }
91
92    // Sign-out
93    SignOut() {
94      return this.ngFireAuth.signOut().then(() => {
95        localStorage.removeItem('user');
96        this.router.navigate(['logout']);
97      })
98    }
99
100 }
```

Listing 4.9: authentification.service

4.2.8 Pure Fabrication

Eine Pure Fabrication (reine Erfindung) stellt eine Klasse dar, die so nicht in der Problem Domain existiert sie stellt eine Methode zur Verfügung bei der sie nicht der Experte ist. In der Anwendung ist Pure Fabrication nicht vorhanden, da es sich bei diesen oftmals um Hilfsklassen handelt welche in der Anwendung nicht genutzt wurden.

4.2.9 Protected Variations

Interfaces sollen immer verschiedene konkrete Implementierungen verstecken. Man nutzt also Polymorphismus und Delegation, um zwischen den Implementierungen zu wechseln. Dadurch kann das restliche System vor den Auswirkungen eines Wechsels der Implementierung geschützt werden. Hierfür eignet sich auch wieder als Beispiel einer der Services. Denn durch die Services wird die Applikations Schicht von der Datenbank “getrennt“so kann man im Nachhinein die Speicherung der Daten nachträglich ändern, ohne dass in der Applikation direkte Auswirkungen davon zu spüren sind.

4.3 DRY

DRY “DON’T REPEAT YOUR SELF“Definition:

- Code Duplikationen vermeiden
- kopierter Code sind kopierte Fehler
- kopierter Code deutet auf falsche oder fehlende Abstraktion hin
- gilt auch für Logik, d.h. unterschiedlicher Code löst die selben Probleme

In der Anwendung gibt es einige Klassen, welche an verschiedenen Stellen benötigt werden. Dies Klassen sind alle in eine extra Datei ausgelagert und können dann wenn sie benötigt werden von dort importiert werden. Eine dieser Klassen ist die Klasse Income, wie auf dem Code Snippet zu sehen. Diese Klasse wird sowohl im Income Service, in der Income-Detail Page als auch auf der IncomePage benötigt. Um Code Duplikationen zu vermeiden wurde die Klasse Income also ausgelagert und wird dann an den entsprechenden Stellen importiert.

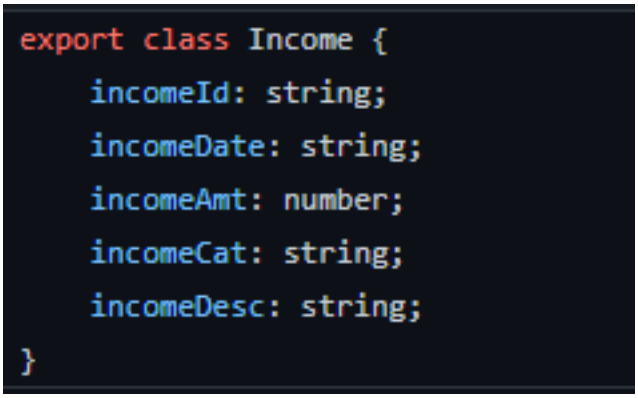

Abbildung 4.3: Income Class

5. *Refactoring*

Beim Refactoring wird schon vorhandener Code erneut durchgegangen und der Code wird umgestaltet. Dabei soll das Gesamt verhalten gleich bleiben, dabei bleiben auch die Schnittstellen nach außen gleich. Das Ziel des Refactorings ist es die Codequalität zu verbessern, sprich, dass der Code leichter lesbar wird und flexibler genutzt werden kann.
Code Smells deuten auf verbesserungswürdige Stellen im Code hin.
Eine Art von Code Smell ist "Duplicated Code". Beim Duplicated Code ist die gleichen Code Struktur mehrfach an unterschiedlichen Stellen im Code vorhanden. Um diesen Code Smell zu lösen wird der gleiche Code ausgelagert und dann von dort aus aufgerufen wenn man ihn benötigt. Aktuell befinden sich die Firebase Credentials an mehreren Orten, wodurch bei Änderungen Fehler entstehen können, wenn an einer Stelle vergessen wird die Credentials zu ändern. Die Credentials befanden sich in diesen beiden Dokumenten. Nach dem Refactoring

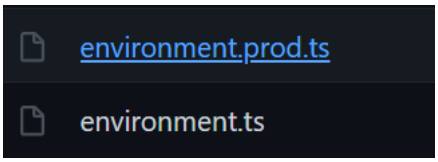

Abbildung 5.1: Duplicated Code

befinden sich die Credentials nur noch in einer Klasse und könne von dort aus Importiert werden. Das Verhindert Fehler, falls sich die Credentials ändern sollten. `https://github.com/FlyingBabYpsilon/AdvancedSE/tree/b86c55e827af85bd3582a628ff5b4a07b14f5116`

Ein weiterer Code Smell ist “Alternative Classes with Different Interfaces“was so viel Bedeutet wie, das der Methodenname nicht zu dem Inhalt der Methode passt das kann. Diesen Code Smell kann man durch die Rename Methode des Refactorings lösen. Dabei werden die Methodennamen so angepasst, dass sie im Nachhinein dann eindeutiger und besser zu der Methode passen. In der Anwendung waren die einzelnen Methoden von Anfang an schlecht benannt, als sie erstellt wurden und im Rahmen des Refactorings wurden die Namen dann an die Tatsächliche Funktionalität der Methoden angepasst. Dadurch wird die Lesbarkeit und Verständlichkeit des Codes erhöht, was eins der Ziele des Refactorings ist.

Dieses Refactoring ist Beispielhaft mit dem Bild aufgezeigt und im GitHub Repository unter dem Commit “Refactoring“updated names““

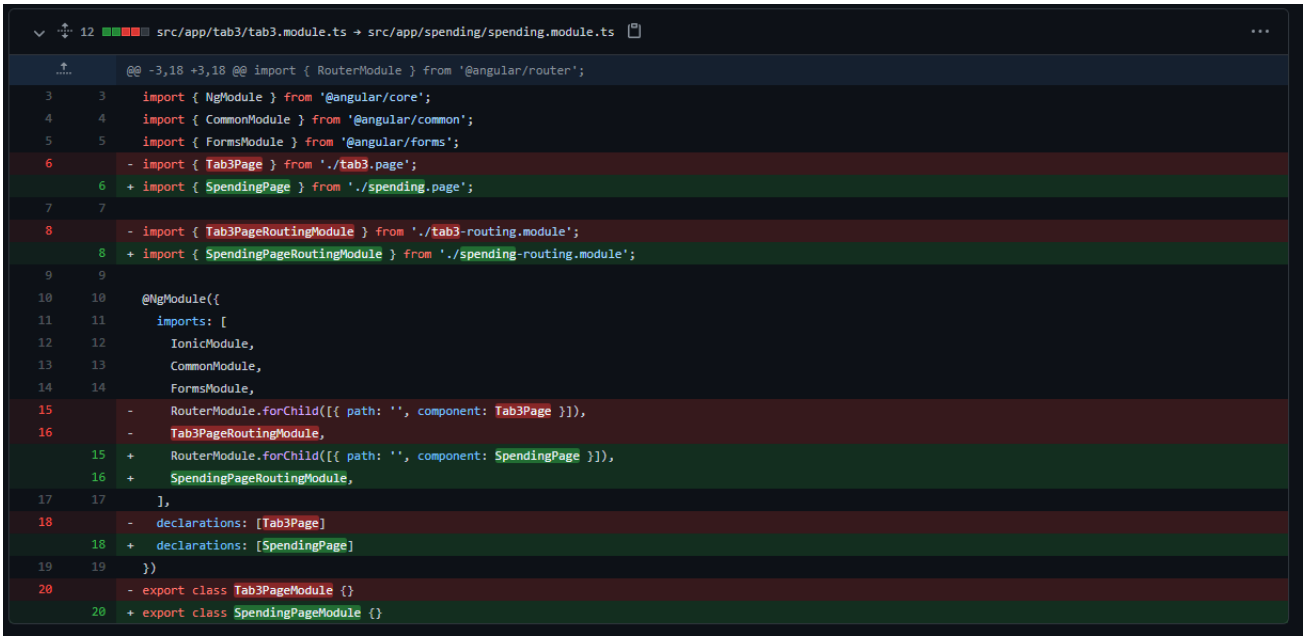

Abbildung 5.2: Umbenennung der Methoden

Auf diesen Zwei Bildern sieht man die Ordner Struktur vorher und nachher mit den Umbenannten Methoden und Klassen. Das erste Bild zeigt den Stand aus dem Git welcher hier zu finden ist `https://github.com/FlyingBabYpsilon/AdvancedSE/tree/624a709e40a10f9edd21093ee2b9` und der Stand nach dem Refactoring ist hier im Git zu finden `https://github.com/FlyingBabYpsilon/AdvancedSE/tree/53363da60b752b9e7664a4c3ebb5e42bf6dc723c`

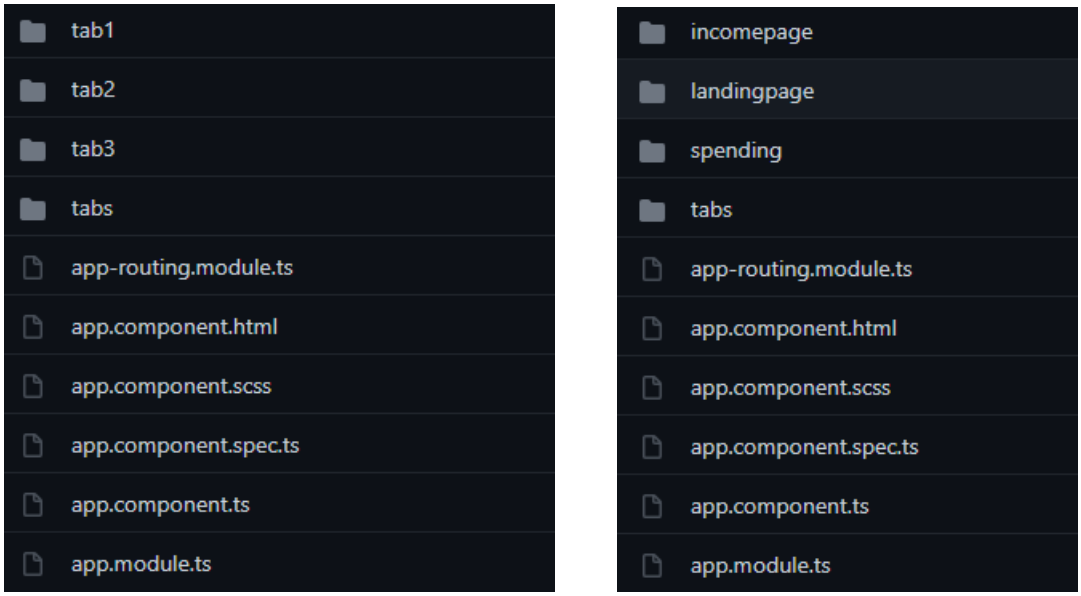

Abbildung 5.3: Bevor and After

6. *Entwurfsmuster*

Der Nutzer erstellt verschiedene Ausgaben und Einnahmen über das entsprechende Formular. Gespeichert werden die Daten dann in einem XML Dokument.

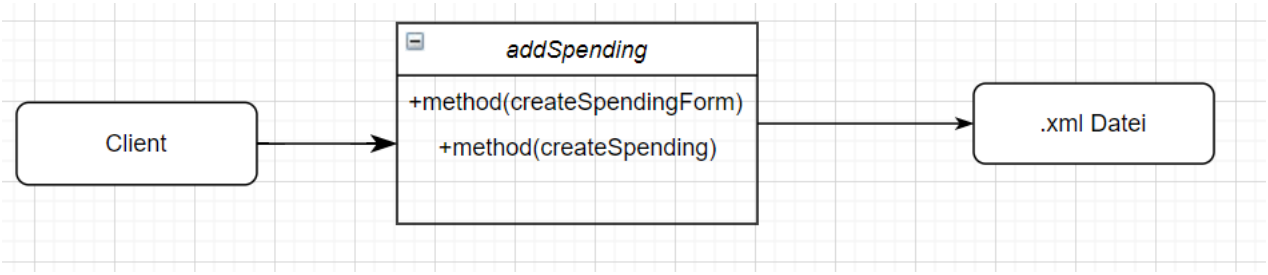

Abbildung 6.1: UML vor Adapter Pattern

Mit Hilfe des Adapter Pattern sollen die Daten aus dem Formular so umgewandelt werden, dass sie ohne Probleme in die NoSQL Datenbank von Firebase gespeichert werden können. Da die Daten in einem Formular mit einer FormGroup eingegeben werden müssen die eingegebenen Daten erst noch so aufbereitet werden, dass sie in die Datenbank geschrieben werden können. Das übernimmt der Adapter mit der CreateSpending() Methode.

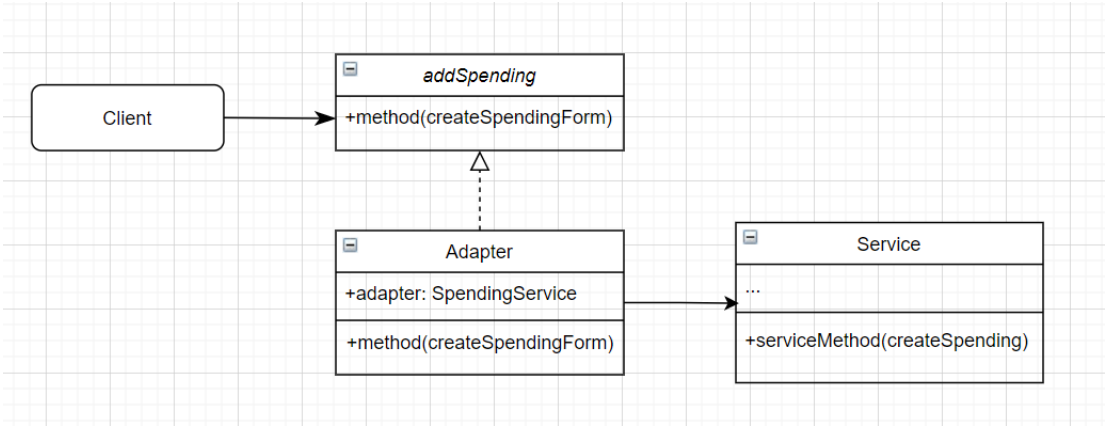

Abbildung 6.2: Adapter Pattern

Das Adapter Pattern bietet den Vorteil des Single Responsibility Principles. Dadurch

kann die Schnittstelle oder der Datenkonvertierungscode von der primären Geschäftslogik getrennt werden. In diesem Fall ist dadurch eine Trennung zwischen dem AddSpending Formular und der Methode möglich, welche die Daten dann in die Firebase Datenbank speichert. Außerdem können durch das Adapter Pattern weitere Adapter hinzugefügt werden, ohne das Änderungen an der Applikations Schicht notwendig sind.

7. *Unit Tests*

to run the test do this: npm run test

Bei den Unit Tests handelt es sich um sehr einfache und simple Tests, das ist mir klar. Allerdings bin ich immer wieder schon alleine bei dem Versuch zu testen, ob die Component erzeugt wurde auf Fehler gestoßen und habe diese nicht gelöst bekommen. Meine Vermutung ist, das es irgendwo Probleme aufgrund der Verwendung von Firebase gab. Dazu habe ich auch noch sehr wenig Erfahrung mit den Unit Tests und deshalb aufgrund der Fehler nicht mehr und komplexere Unit Tests hin bekommen, zumindest nicht mehr Unit Tests, welche funktionieren und nicht fehlschlagen.

In den Folgenden Kapiteln befinden sich die funktionierenden Unit Tests. Alle weiteren Unit Tests sind in dem Ordner der dazugehörigen Instanz zu finden in der Datei “dateiname.page.spec.ts“

7.1 Income Service

```
1  import { inject, TestBed } from '@angular/core/testing';
2
3  import { IncomeService } from './income.service';
4
5  describe('IncomeService', () => {
6    let IncomeServiceSpy;
7
8    beforeEach(() => {
9      IncomeServiceSpy = jasmine.createSpyObj('IncomeService', {
10        incomeId: 0,
11        incomeAmt: 0,
12        incomeCat: 0,
13        incomeDate: 0
14      });
15     TestBed.configureTestingModule({
16       providers: [
```

```
        IncomeService,
        { provide: IncomeService, useValue: IncomeServiceSpy }
      ]
    });
  });

  it('does some test where it is injected',
    inject([IncomeService], (service: IncomeService) => {
      expect(service).toBeTruthy();
    })
  );

  it('does some test where it is manually built', () => {
    const service = new IncomeService(IncomeServiceSpy);
    expect(service).toBeTruthy();
  });
});
```

Listing 7.1: Unit Test IncomeService

7.2 Spending Service

```
import { inject, TestBed } from '@angular/core/testing';
import { SpendingService } from './spending.service.ts.service';

describe('SpendingService', () => {
  let SpendingServiceSpy;

  beforeEach(() => {
    SpendingServiceSpy = jasmine.createSpyObj('SpendingService', {
       spendId: 0,
       spendAmt: 0,
       spendCat: 0,
       spendDesc: 0,
       spendDate: 0
     });
    TestBed.configureTestingModule({
      providers: [
        SpendingService,
        { provide: SpendingService, useValue: SpendingServiceSpy }
      ]
    });
  });

```

```
  it('does some test where it is injected',
    inject([SpendingService], (service: SpendingService) => {
      expect(service).toBeTruthy();
    })
  );

  it('does some test where it is manually built', () => {
    const service = new SpendingService(SpendingServiceSpy);
    expect(service).toBeTruthy();
  });
});
```

Listing 7.2: Unit Test Spending Service

7.3 Logout Page

```
describe('LogoutPage', () => {

    const FirestoreStub = {
        collection: (_user: string) => ({
          doc: () => ({
            valueChanges: () => new BehaviorSubject({ foo: 'bar' }),
            set: (_d: any) => new Promise<void>((resolve, _reject) =>
   resolve()),
          }),
        }),
      };

    beforeEach(waitForAsync(() => {

        TestBed.configureTestingModule({
          declarations: [LogoutPage],
          providers: [
            {provide: UrlSerializer},
            {provide: LogoutPage},
            {provide: AuthenticationService, useValue: FirestoreStub},

          ]
        }).compileComponents();
      }));

      it('should create the page', () => {
        const fixture = TestBed.createComponent(LogoutPage);
        const app = fixture.debugElement.componentInstance;
```

```
        expect(app).toBeTruthy();
      });

});
```

Listing 7.3: Unit Test Logout Page

7.4 Tabs Page

```
describe('TabsPage', () => {
  let component: TabsPage;
  let fixture: ComponentFixture<TabsPage>;

  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [TabsPage],
      schemas: [CUSTOM_ELEMENTS_SCHEMA],
    }).compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(TabsPage);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Listing 7.4: Unit Test Tabs Page

7.5 Verify Email Page

```
describe('VerifyEmailPage', () => {

  const FirestoreStub = {
    collection: (_user: string) => ({
      doc: () => ({
        valueChanges: () => new BehaviorSubject({ foo: 'bar' }),
        set: (_d: any) => new Promise<void>((resolve, _reject) => resolve
   ()),
```

```
      }),
    }),
  };

    beforeEach(waitForAsync(() => {

        TestBed.configureTestingModule({
          declarations: [VerifyEmailPage],
          providers: [
            {provide: UrlSerializer},
            {provide: VerifyEmailPage},

            {provide: AuthenticationService, useValue: FirestoreStub},

          ]
        }).compileComponents();
      }));

      it('should create the page', () => {
        const fixture = TestBed.createComponent(VerifyEmailPage);
        const app = fixture.debugElement.componentInstance;
        expect(app).toBeTruthy();
      });

});
```

Listing 7.5: Unit Test Verify Email

7.6 App Component

```
describe('AppComponent', () => {

  beforeEach(waitForAsync(() => {

    TestBed.configureTestingModule({
      declarations: [AppComponent],
      schemas: [CUSTOM_ELEMENTS_SCHEMA],
    }).compileComponents();
  }));

  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
```

```
15   });
16 });
```

Listing 7.6: Unit Test App Component

7.7 Ergebnisse der Unit Tests

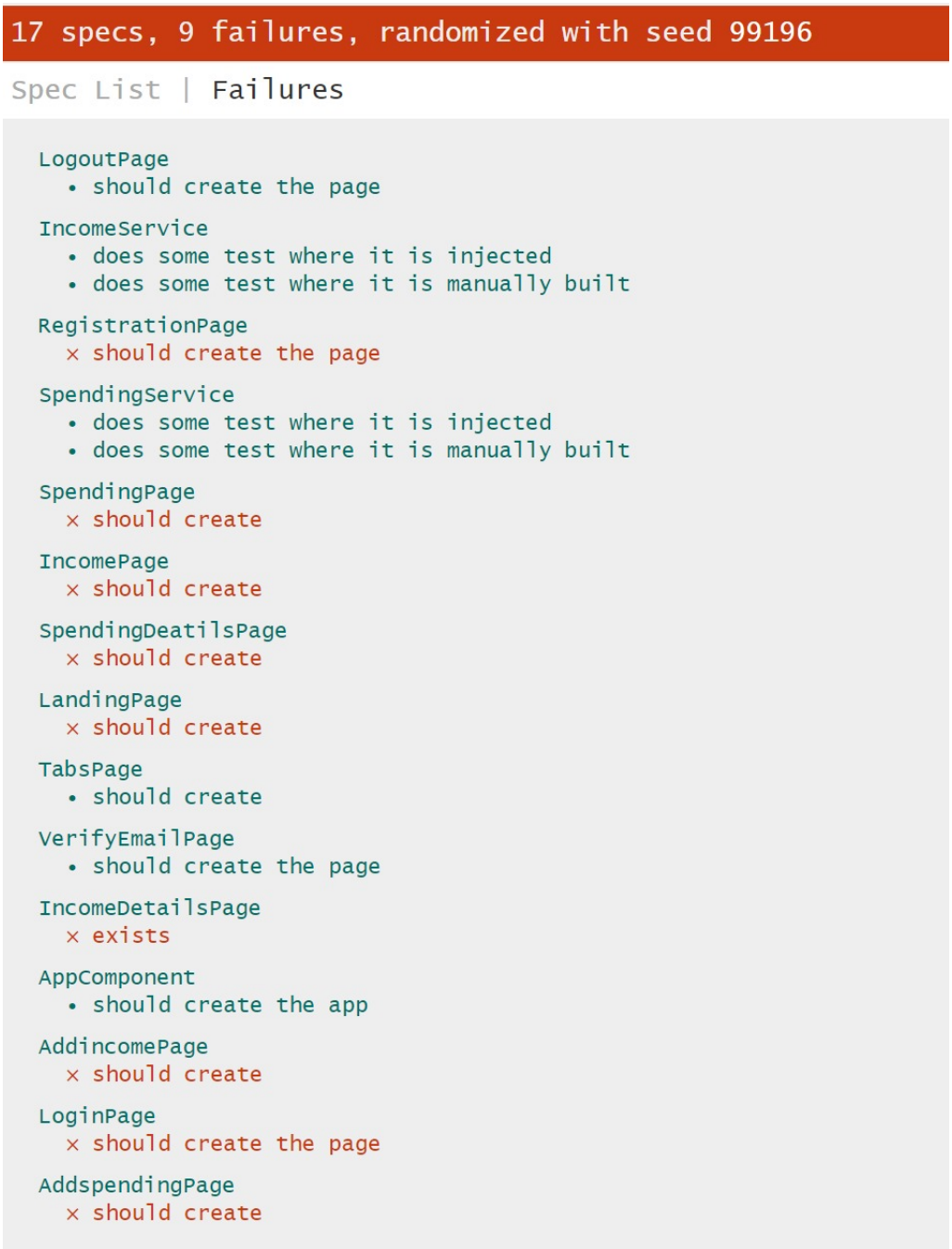

Abbildung 7.1: Unit Test Ergebnisse

Abbildungsverzeichnis

Literaturverzeichnis

[BMBF, 2003] "IT-Ausstattung der allgemein bildenden und berufsbildenden Schulen in Deutschland", http://www.schulen-ans-netz.de/neuemedien/fakten/dokus/it-ausstattung-2003.pdf, 10.03.2005