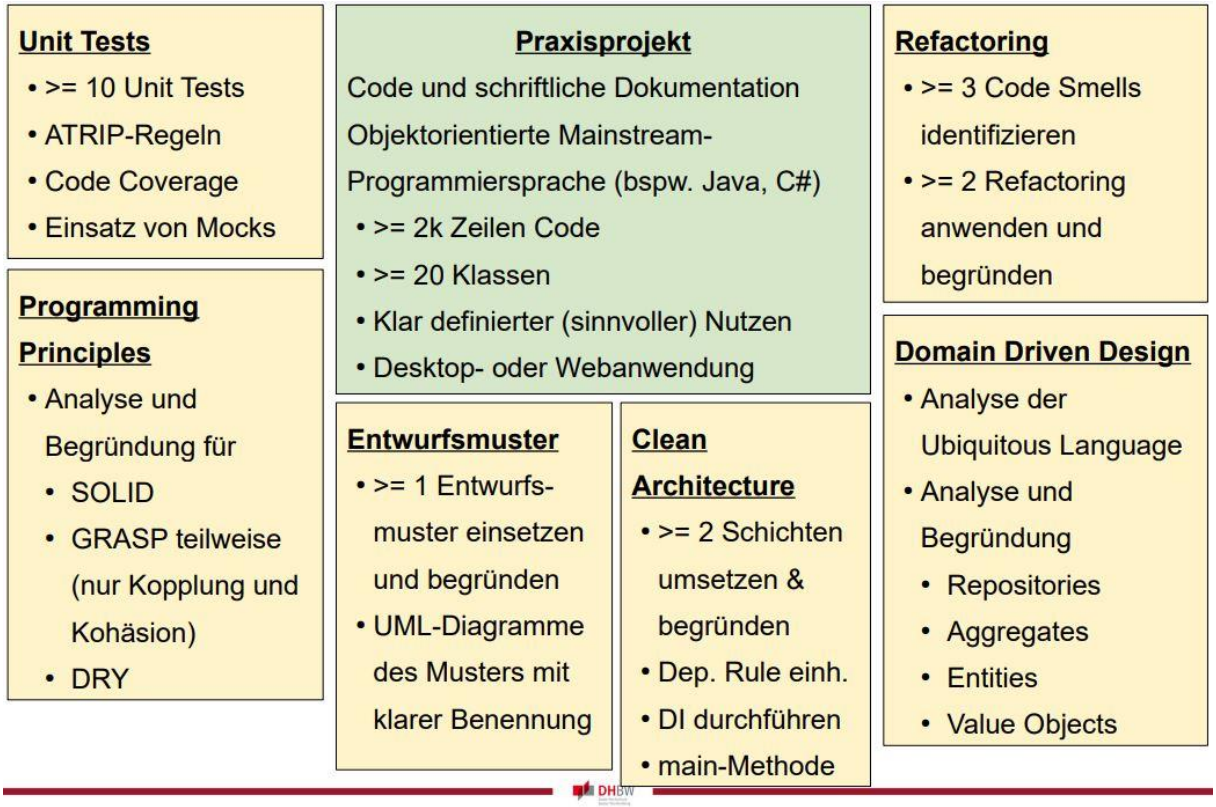
Programmentwurf TINF20B1

Florian Kloss 1844332

Lukas Jaedicke 3275226

Zahra Dehghani 3813855

Github Repository: https://github.com/Eluminix/Praxisprojekt_SWE



1. Unit Tests

1.1 Einführung

Bei Unit-Tests handelt es sich um automatisierte Tests, die von den Entwicklern geschrieben werden, um die Funktionalität einzelner Code-Einheiten (d. h. kleiner Teile des Programmcodes) zu überprüfen. Für das Schreiben und Ausführen dieser Tests werden in der Regel spezielle Frameworks verwendet.

Mit Unit-Tests soll sichergestellt werden, dass der Code für einzelne Komponenten der Software korrekt funktioniert und die erwarteten Ergebnisse liefert. Dadurch können Fehler frühzeitig erkannt und vermieden werden, indem sie frühzeitig im Entwicklungsprozess gefunden und behoben werden.

Insgesamt trägt das Testen von Einheiten zur Verkürzung der Entwicklungszeit und zur Erhöhung der Qualität und Stabilität der Software bei.

1.1.1 ATRIP-Regeln

ATRIP-Regeln

- **Automatic** - Eigenständig
    - Die Tests müssen ihre Ergebnisse selbst überprüfen
- **Thorough** - Gründlich (genug)
    - Jede missionskritische Funktionalität ist getestet
    - Für jeden aufgetretenen Fehler existiert ein Testfall, der ein erneutes Auftreten verhindert
- **Repeatable** - Wiederholbar
    - Jeder Test sollte jederzeit (automatisch) durchführbar sein und immer das gleiche Ergebnis liefern
- **Independent** - Unabhängig voneinander
    - Tests müssen jederzeit in beliebiger Zusammenstellung und Reihenfolge funktionsfähig sein
- **Professional** - Mit Sorgfalt hergestellt
    - Testcode sollte so leicht verständlich wie möglich sein

Angular hat mit Jasmine und Karma sehr gute Testwerkzeuge.

Automatic wird erfüllt, da die Tests ihre Ergebnisse selbst prüfen und bei Fehlern eine Fehlermeldung ausgeben.

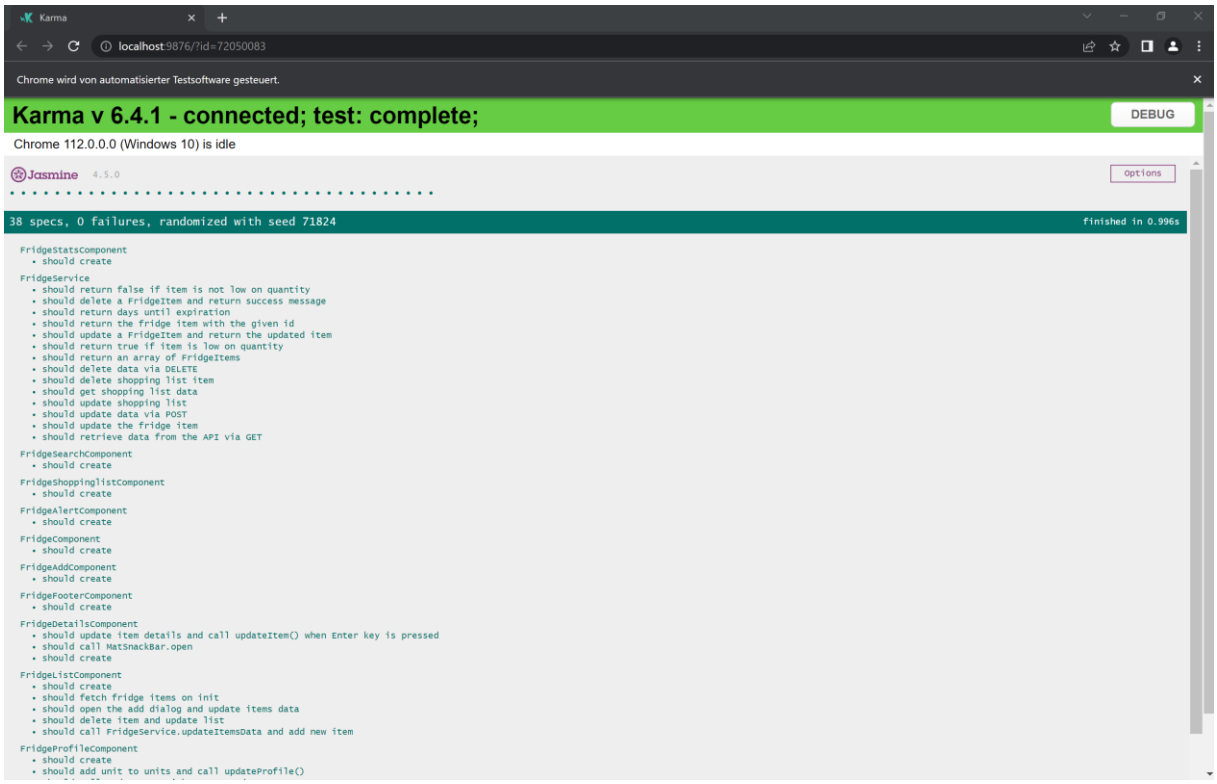
Thorough wird er erfüllt, da jede Komponente eine automatisierte Test-Datei (.spec) beinhaltet. Diese werden mit Tests für kritische Funktionalitäten manuell gefüllt.

Repeatable wird erfüllt, da alle Tests automatisch durchführbar sind und immer das gleiche Ergebnis liefern. Ein Mock wird hierbei beispielsweise benutzt, um Artikeleinträge, welche sich eigentlich auf dem Server befinden zu simulieren. Diese Artikel verändern sich nicht und bleiben bei jedem Test gleich.

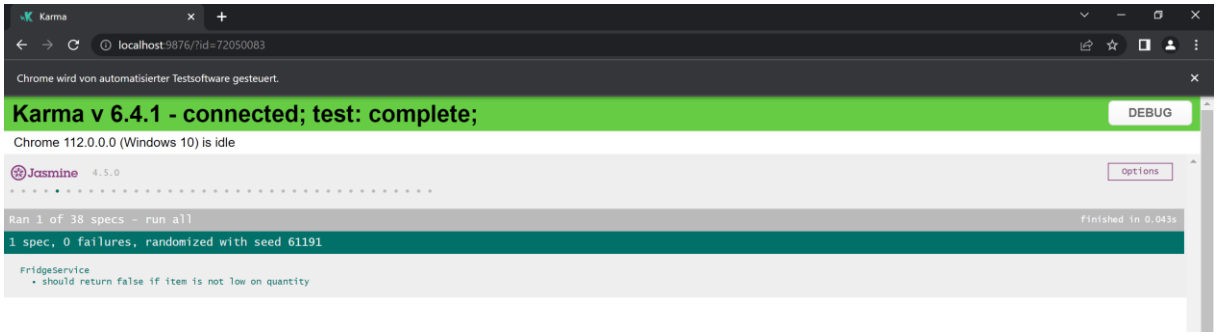
Independent wird erfüllt, da Karma immer einen neuen Seed erzeugt, welcher die Reihenfolge der Tests randomisiert.

Professional wird erfüllt, da jeder Test nur eine Funktion testet. Diese wird so leicht verständlich wie möglich mit aussagestarken Variablennamen bzw. Funktionsnamen umschrieben.

Getestet wird mit Jasmine 4.5.0 Karma v 6.4.1. Jeder SEED ist mit einer zufälligen Reihenfolge generiert.



Komponenten und Unit Tests werden angezeigt und können einzeln ausgewählt werden



1.1.2 Code Coverage

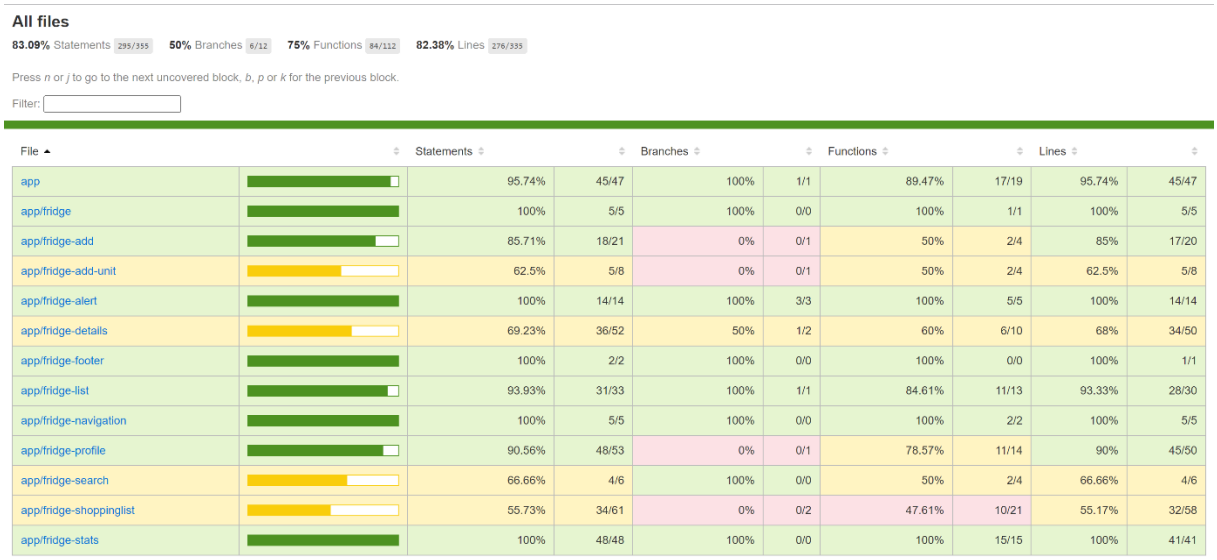
Code Coverage ist ein Maß für den Anteil des während der Testausführung ausgeführten Programmcodes. Sie wird als prozentualer Anteil des während der Testausführung ausgeführten Codes an der Gesamtmenge des Codes angegeben.

Code Coverage ist ein wichtiges Werkzeug, um die Qualität und Vollständigkeit von Tests zu überprüfen und zu verbessern. Es unterstützt Entwickler bei der Überwachung der Testabdeckung und bei der Sicherstellung, dass die Tests effizient sind und alle wichtigen Codepfade abdecken.

In unserer Anwendung kann mit dem Befehl:

ng test --no-watch --code-coverage

ein Ordner erstellt werden, der die Code Coverage beinhaltet. Die index.html muss dafür in einem Browser geöffnet werden.



All files

83.09% Statements 295/355 **50%** Branches 6/12 **75%** Functions 84/112 **82.38%** Lines 276/335

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
app	95.74%	45/47	100%	1/1	89.47%	17/19	95.74%	45/47
app/fridge	100%	5/5	100%	0/0	100%	1/1	100%	5/5
app/fridge-add	85.71%	18/21	0%	0/1	50%	2/4	85%	17/20
app/fridge-add-unit	62.5%	5/8	0%	0/1	50%	2/4	62.5%	5/8
app/fridge-alert	100%	14/14	100%	3/3	100%	5/5	100%	14/14
app/fridge-details	69.23%	36/52	50%	1/2	60%	6/10	68%	34/50
app/fridge-footer	100%	2/2	100%	0/0	100%	0/0	100%	1/1
app/fridge-list	93.93%	31/33	100%	1/1	84.61%	11/13	93.33%	28/30
app/fridge-navigation	100%	5/5	100%	0/0	100%	2/2	100%	5/5
app/fridge-profile	90.56%	48/53	0%	0/1	78.57%	11/14	90%	45/50
app/fridge-search	66.66%	4/6	100%	0/0	50%	2/4	66.66%	4/6
app/fridge-shoppinglist	55.73%	34/61	0%	0/2	47.61%	10/21	55.17%	32/58
app/fridge-stats	100%	48/48	100%	0/0	100%	15/15	100%	41/41

In Angular wird die Code Coverage anhand von Funktionen der jeweiligen Komponenten gemacht.

Im Bild sieht man die einzelnen Komponenten aufgelistet.

70-80 % Codeabdeckung erscheinen uns als angemessen. 50-70% Abdeckung sind noch im guten Bereich, solange die wichtigsten Funktionen abgedeckt sind.

1.1.3 Mocks

Mocks (oder Mock-Objekte) sind spezielle Objekte, die in Unit-Tests zur Simulation von Abhängigkeiten von anderen Komponenten dienen. Sie werden verwendet, um Codeeinheiten zu testen, die von anderen Komponenten wie Datenbanken, Webservices, externen Bibliotheken oder anderen Klassen abhängen.

Mock-Objekte werden eingesetzt, um das Verhalten einer Abhängigkeit zu simulieren und somit zu verhindern, dass der Test auf externe Ressourcen zugreift, die möglicherweise nicht immer verfügbar sind oder ein inkonsistentes Verhalten zeigen.

Im Wesentlichen ersetzt die Verwendung von Mock-Objekten den Code der Abhängigkeit durch einfachen, selbst definierten Code. Auf diese Weise kann das Verhalten der Abhängigkeit im Test kontrolliert und überprüft werden, um sicherzustellen, dass die getestete Code-Einheit korrekt funktioniert.

Mocks sind ein wichtiger Bestandteil von Unit-Tests, da sie helfen, Tests von externen Ressourcen unabhängig zu machen und damit die Testzuverlässigkeit erhöhen.

Ein Beispiel dazu:

```
describe('FridgeService', () => {
  let fridgeService: FridgeService;
  let httpMock: HttpTestingController;

  beforeEach(async () => {
    TestBed.configureTestingModule({
          imports: [HttpClientModule, HttpClientTestingModule],
          providers: [FridgeService]
      });
    fridgeService = TestBed.inject(FridgeService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    httpMock.verify();
  });
```

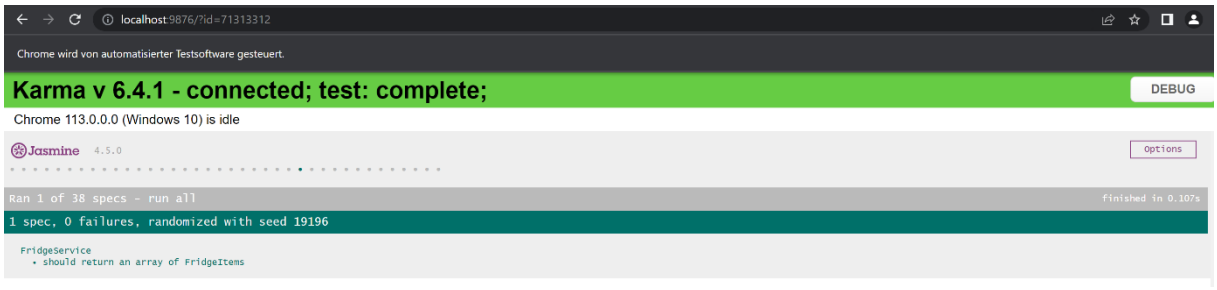
Hierbei wurde das Mock-Objekt “httpMock” erstellt, um HTTP-Anfragen zu simulieren. Die Funktion beforeEach() ermöglicht das injizieren des Mock-Objekts vor jedem Test.

```
it('should return an array of FridgeItems', () => {
    const expectedData: FridgeItem[] = [
        { id: 1, name: 'Milch', quantity: 2, expiryDate: new Date(2023, 4, 1),
category: "Milchprodukte", notes: "test", amount: 12, kcal: 1, sugar: 1,fat:
1,protein: 1, carbs: 1  },
        { id: 2, name: 'Eier', quantity: 6, expiryDate: new Date(2023, 4, 2),
category: "Eier", notes: "test", amount: 12, kcal: 1, sugar: 1,fat: 1,protein:
1, carbs: 1  },
      ];;

    fridgeService.getItemsData().subscribe((data: any) => {
      expect(data).toEqual(expectedData);
    });

    const req = httpMock.expectOne(fridgeService.itemsurl);
    expect(req.request.method).toBe('GET');
    req.flush(expectedData);
  });
```

In diesem Test soll die HTTP-Anfrage “GET” ausgeführt und getestet werden.



Der Test ist abgeschlossen und liefert keinen Fehler. Somit wurde “GET” ohne Probleme aufgerufen.

1.2 Testbereiche

- Fridge Service (Kern der Kommunikation zwischen Server und Komponenten)
- Artikel Liste & Detail Ansicht
    - Verwaltung von Artikeln (Kernfunktion der Software)
    - Kritischste Funktionen
- Profil, Shoppingliste

1.3 Aufbau Tests

Bei Angular werden „.spec“-Dateien (Abkürzung für Specification Files) zur Erstellung von Unit Tests für Komponenten, Dienste oder Direktiven benutzt. Diese Dateien sind Teil des Test-Setups von Angular und enthalten die Testfälle, mit denen die Funktionalität der Komponenten oder Dienste getestet wird. Angular erstellt beim Generieren einer Komponente oder eines Dienstes mit dem Angular CLI automatisch eine „.spec“-Datei für diese Komponente oder diesen Dienst, welche normalerweise den gleichen Namen trägt mit dem Suffix ".spec.ts" angehängt.

Zum Beispiel wird für eine Komponente mit dem Namen "MyComponent" eine „.spec“-Datei mit dem Namen "MyComponent.spec.ts" erstellt.

Die „.spec“-Datei enthält in der Regel eine oder mehrere Testsuiten. Jede Testsuite enthält mehrere Testfälle. Diese Testfälle testen verschiedene Aspekte der Funktionalität der Komponente oder des Dienstes, indem sie bestimmte Methoden oder Eigenschaften aufrufen und die erwarteten Ergebnisse vergleichen.

1.4 Beispiele

Commit: Setup für Unit Tests:

https://github.com/Eluminix/Praxisprojekt_SWE/pull/12

Commit: Erstellung von Unit Tests:

https://github.com/Eluminix/Praxisprojekt_SWE/pull/13

```
it('should update a FridgeItem and return the updated item', () => {
    const newItem: FridgeItem = { id: 1, name: 'Käse', quantity: 1,
expiryDate: new Date(2023, 4, 1), category: 'Milchprodukte', notes: '',
amount: 50, kcal: 200, sugar: 1, fat: 10, protein: 5, carbs: 10 };
    const expectedData: FridgeItem = { ...newItem };

    fridgeService.updateItemsData(newItem).subscribe((data: FridgeItem) => {
      expect(data).toEqual(expectedData);
    });

    const req = httpMock.expectOne(fridgeService.itemsurl);
    expect(req.request.method).toBe('POST');
    req.flush(expectedData);
  });
```

Testet, ob ein Artikel mit der HTTP-Anfrage „POST“ hinzugefügt wurde

```
  it('should delete a FridgeItem and return success message', () => {
    const itemId = 1;
    const expectedData = 'Item deleted successfully';

    fridgeService.deleteItem(itemId).subscribe((data: any) => {
      expect(data).toEqual(expectedData);
    });

    const req = httpMock.expectOne(`${fridgeService.itemsurl}/${itemId}`);
    expect(req.request.method).toBe('DELETE');
    req.flush(expectedData);
  });
```

Testet, ob ein Artikel mit der HTTP-Anfrage „DELETE“ gelöscht wurde.

```
 it('should return the fridge item with the given id', () => {
      const itemId = 1;
      const mockItems = [
        { id: 1, name: 'Milch', quantity: 2, expiryDate: new Date(2023, 4, 1),
category: "Milchprodukte", notes: "test", amount: 12, kcal: 1, sugar: 1,fat:
1,protein: 1, carbs: 1  },
```

```
        { id: 2, name: 'Eier', quantity: 6, expiryDate: new Date(2023, 4, 2),
category: "Eier", notes: "test", amount: 12, kcal: 1, sugar: 1,fat: 1,protein:
1, carbs: 1  },
         ];

      const expectedItem =  { id: 1, name: 'Milch', quantity: 2, expiryDate:
new Date(2023, 4, 1), category: "Milchprodukte", notes: "test", amount: 12,
kcal: 1, sugar: 1,fat: 1,protein: 1, carbs: 1  };
       ;

      fridgeService.getItemById(itemId).subscribe((item: FridgeItem) => {
        expect(item).toEqual(expectedItem);
      });

      const req = httpMock.expectOne(`${fridgeService.itemsurl}`);
      expect(req.request.method).toEqual('GET');
      req.flush(mockItems);
    });
```

Testet, ob der Artikel mit der HTTP-Anfrage „GET“ und der erwarteten ID zurückgegeben wird.

2. Refactoring

Das Refactoring wird in 2 Teile geteilt, Code Smells und Refactoring.

2.1 Code Smells (6)

Code Smells sind in folgende Punkte unterteilt:

Duplicated Code

Long Method

Code Comments

2.1.1 Duplicated Code

Duplicated Code ist Code, der gleich aufgebaut ist und an mehreren Stellen auftritt.

Fridge-add und fridge-add-unit:

```
ngOnInit(): void {

   const backdropClick$ = this.dialogRef.backdropClick();
   if (backdropClick$) {
     backdropClick$.subscribe(() => {
       this.cancel();
     });
   }

 }
```

Dieser Code-Ausschnitt wird bei Fridge-Add und Fridge-Add-Unit benutzt. Er ist gleich aufgebaut und wird für den Dialog des Hinzufügens von Artikeln bzw. Fächer benutzt.

Dieser Ausschnitt kann in einen Service, der für Dialog Funktionen ist, ausgelagert werden.

Fridge-stats:

```
createPieChart() {
   this.chart = new Chart("MyChart", {
     type: 'bar', //this denotes tha type of chart

     data: {// values on X-Axis
       labels: ['Gesamtinhalt', 'Gesamtkapazität', 'Genutzte
Kapazität','Freie Kapazität',
                'Durchschnittliche Anzahl', 'Minimale Anzahl', 'Maximale
Anzahl'],
```

```
         datasets: [
          {
            label: "Anzahl",
            data:
[this.totalItems,  this.totalCapacity,  this.totalUsedCapacity ,
this.totalFreeCapacity , this.avgQuantity,
              this.minQuantity, this.maxQuantity],
            backgroundColor: '#134E5E'
          },

        ]
      },
      options: {
        aspectRatio:2
      }


    });
  }
```

```
createCategoryChart() {
    this.chartCategory = new Chart("CategoryChart", {
      type: 'bar', //this denotes tha type of chart

      data: {// values on X-Axis
        labels: this.labels,
         datasets: [
          {
            label: "Anzahl",
            data: this.data,
            backgroundColor: '#134E5E'
          },

        ]
      },
      options: {
        aspectRatio:2
      }

    });
  }
```

In diesem Code-Ausschnitt werden zwei Diagramme generiert, die bis auf ihren Inhalt gleich aufgebaut sind. Beide Funktionen können zu einer Funktion geändert werden, die den Inhalt unterscheidet und ausgibt.

Die Anwendung beinhaltet viele HTTP-Anfragen, die sehr ähnlich aufgebaut sind. Der größte Unterschied liegt im Zugriff auf die verschiedenen JSON-Dateien. Man könnte eine Funktion für alle „POST“-Anfragen, eine Funktion für alle „GET“-Anfragen und eine Funktion für alle „DELETE“-Anfragen erstellen und in diese, je nach benutzter JSON, unterteilen. Das würde aber zu großen unübersichtlichen Funktionen führen. Deshalb sind einzelne Funktion, für die jeweiligen Anfragen, kürzer und übersichtlicher.

```
  // Profil
  getProfileConfigurationData() {
    return this.http.get(this.dataurl);
  }

  updateData(newData: any): Observable<any> {
    return this.http.post(this.dataurl, newData);
  }

  deleteData(unit: any): Observable<any> {
    const url = `${this.dataurl}/${unit}`;
    return this.http.delete(url);
  }

// Artikelliste
  getItemsData() {
    return this.http.get(this.itemsurl);
  }


  addItem(newData: FridgeItem): Observable<any> {
    return this.http.post(this.itemsurl, newData);
  }


  deleteItem(id: number): Observable<any> {
    const url = `${this.itemsurl}/${id}`;
    re itimurn this.http.delete(url);
  }
```

Die Methode openDialog():

```
  openDialog(): void {
    const dialogRef = this.dialog.open(FridgeAddUnitComponent);
    dialogRef.afterClosed().subscribe(result => {
    if(result != false) {
      this.addItem(result);
    }
  });
```

Bei jeder Komponente, die ein Dialog öffnen kann, wird das Ergebnis nach dem Schließen des Dialogs verarbeitet. Mehrere Komponenten besitzen den Aufruf eines Dialogs. Die Unterschiede sind dabei minimal, da entweder ein Artikel oder ein neues Fach hinzugefügt wird (FridgeAddComponent, FridgeAddUnitComponent). Ein Service, der nur für Dialog Funktionen ist, könnte diese Funktionen aus den Komponenten auslagern.

2.1.2 Long Method

Long Methods sind Funktionen, welche sehr viel Code beinhalten. Diese können gekürzt und in mehrere Funktionen ausgelagert werden.

Fridge-stats.component:

```
 ngOnInit(): void {
    this.getProfileData();
    this.fridgeService.getItemsData().subscribe((data: any) => {
      this.fridgeItems = data;
      this.totalItems = this.fridgeItems.length;

      this.totalUsedCapacity = this.fridgeItems.reduce((total, item) => total
+ item.quantity, 0);
      this.totalFreeCapacity = this.totalCapacity - this.totalUsedCapacity;

  // Calculate average, minimum and maximum quantities of all items
      const totalQuantities = this.fridgeItems.reduce((total, item) => total +
item.quantity, 0);
      this.avgQuantity = totalQuantities / this.totalItems;
      this.minQuantity = Math.min(...this.fridgeItems.map(item =>
item.quantity));
      this.maxQuantity = Math.max(...this.fridgeItems.map(item =>
item.quantity));

 const filteredItems = this.fridgeItems.filter(item => item.category ===
this.fridgeItems[0].category);

    for (let index = 0; index < this.fridgeService.categories.length; index++)
{
      var filter = this.fridgeItems.filter(item => item.category ===
this.fridgeService.categories[index]);
      this.mappedCategory.push({name: this.fridgeService.categories[index],
quantity: filter.length})
    }
    console.log(this.mappedCategory)

    this.mappedCategory.forEach(element => {
      this.labels.push(element.name)
    });
```

```
    this.mappedCategory.forEach(element => {
      this.data.push(element.quantity)
    });

    this.createPieChart();
    this.createCategoryChart();
    });

//    this.fridgeItems = this.fridgeService.getFridgeItems();

  }
```

Hierbei sind viele Funktionalitäten, wie Zuweisungen und Berechnungen in der ngOnInit()-Methode vorhanden. Um den Code verständlicher und umgänglicher zu machen, sollten Aufteilungen und Auslagerungen gemacht werden.

ngOnInit() sollte nur Aufrufe von Methoden haben, die zum Initialisieren der Komponente benötigt werden. Die Zuweisungen und Berechnungen sollten in diesen Methoden erfolgen.

2.1.3 Code Comments

Code Comments befasst sich mit Kommentaren im Code, die keinen Nutzen haben. Sie kommentieren einen alten Stand oder kommentieren Dinge aus, die nicht benutzt werden.

Fridge-profile.component.ts:

```
// this.user = this.authService.getCurrentUser();

/*
  {
    "user": [{"name": "Max"}, {"email": "test@gmx.de"}, {"password":
"wqeqw"}],
    "units": ["Seitenfach", "Gefrierfach", "Hauptfach"],
    "capacity": 150,
  "measurementSetting": "metrisch",
  "languageSetting": "deutsch",
  "localizationSetting": "euro",
  "autoAddSetting": true,
  "alertSetting": true
}
*/
```

Im Code oben wird ein alter Stand auskommentiert. Dieser Kommentar kann entfernt werden.

Das gleiche gilt für die Methode console.log(). Diese wurde zum Testen benutzt und hat in der Anwendung nichts mehr zu suchen.

```
 console.log(this.capacity);
   console.log(this.username);
   console.log(this.localizationSetting);
   console.log(this.languageSetting);
   console.log(this.measurementSetting);
   console.log(this.autoAddSetting);
   console.log(this.alertSetting);
```

Fridge-Service:

```
 // Profil
  getData() {
    return this.http.get(this.dataurl);
  }

  updateData(newData: any): Observable<any> {
    return this.http.post(this.dataurl, newData);
  }

  deleteData(unit: any): Observable<any> {
    const url = `${this.dataurl}/${unit}`;
    return this.http.delete(url);
  }

// Artikelliste
  getItemsData() {
    return this.http.get(this.itemsurl);
  }




  updateItemsData(newData: FridgeItem): Observable<any> {
    return this.http.post(this.itemsurl, newData);
  }


  deleteItem(id: number): Observable<any> {
    const url = `${this.itemsurl}/${id}`;
    return this.http.delete(url);
  }
```

Hier werden Kommentare benutzt, um den Code in Abschnitte zu unterteilen. ABER die Funktionen sollten selbst einen aussagekräftigen Namen haben. Dadurch werden die Kommentare nutzlos und können entfernt werden.

2.2 Anwendung - Refactoring (4)

Commit: Refactoring https://github.com/Eluminix/Praxisprojekt_SWE/pull/14

Das Refactoring wird in folgende Abschnitte unterteilt:

Extract Method

Rename Method

Replace Temp with Query

Replace Error Code with Exception

Methoden, die keinen Zweck erfüllen

2.2.1 Extract Method

Extract Method verkleinert Methoden, indem Teile in andere Methoden ausgelagert werden. Diese Methoden sollten jeweils mit ihrem Namen ihre Funktionalität beschreiben. Dadurch wird der Code übersichtlicher.

Fridge-Add:

```
ngOnInit(): void {
    this.categories = this.fridgeService.categories;
    this.backdropClick$ = this.dialogRef.backdropClick();
  if (this.backdropClick$) {
    this.backdropClick$.subscribe(() => {
      this.cancel();
    });
  }
  }
```

Zu

```
ngOnInit(): void {
    this.categories = this.fridgeService.categories;
    this.getBackdropClick();
  }

getBackdropClick() {
  this.backdropClick$ = this.dialogRef.backdropClick();
  if (this.backdropClick$) {
    this.backdropClick$.subscribe(() => {
      this.cancel();
    });
  }
}
```

Die Methode ngOnInit() wird verkleinert, indem die Funktionalität des Dialogs in eine eigene Methode ausgelagert wird.

Fridge-Add-Unit:

```
ngOnInit(): void {
   const backdropClick$ = this.dialogRef.backdropClick();
   if (backdropClick$) {
     backdropClick$.subscribe(() => {
       this.cancel();
     });
   }
 }
```

Zu

```
backdropClick$: any;

ngOnInit(): void {
   this.getBackdropClick();
   }

 getBackdropClick() {
   this.backdropClick$ = this.dialogRef.backdropClick();
   if (this.backdropClick$) {
     this.backdropClick$.subscribe(() => {
       this.cancel();
     });
   }
 }
```

Hier gilt das gleiche wie zuvor.

Fridge-Alert:

```
 ngOnInit() {
   this.fridgeService.getItemsData().subscribe((data: any) => {
     this.fridgeItems= data;
       const expiringItems = this.fridgeItems.filter((item) => {
         const daysUntilExpiration =
this.fridgeService.getDaysUntilExpiration(item);
         const isLowOnQuantity = this.fridgeService.isLowOnQuantity(item);
         return daysUntilExpiration <= 3 || isLowOnQuantity;
       });

       if (expiringItems.length > 0) {
```

```
          const message = `Folgende Artikel sollten beachtet werden:
${expiringItems
            .map((item) => item.name)
            .join(', ')}`;
          this.snackBar.open(message, '', { duration: 5000, panelClass:
['warn-snackbar'] });
        }
    });

  }
```

Zu

```
  ngOnInit() {
  this.expiringAndLowQuantityAlert();
  }

  expiringAndLowQuantityAlert() {
    this.fridgeService.getItemsData().subscribe((data: any) => {
      this.fridgeItems= data;
        const expiringItems = this.fridgeItems.filter((item) => {
          const daysUntilExpiration =
this.fridgeService.getDaysUntilExpiration(item);
          const isLowOnQuantity = this.fridgeService.isLowOnQuantity(item);
          return daysUntilExpiration <= 3 || isLowOnQuantity;
        });

        if (expiringItems.length > 0) {
          const message = `Folgende Artikel sollten beachtet werden:
${expiringItems
            .map((item) => item.name)
            .join(', ')}`;
          this.snackBar.open(message, '', { duration: 5000, panelClass:
['warn-snackbar'] });
        }
    });
  }
```

Die ngOnInit()-Methode ist mit der Funktionalität des Aufrufens einer Snackbar vollgepackt, welche erscheint, sobald Artikel am Ablaufen oder in geringer Menge sind.

Dies wird in die Methode expiringAndLowQuantityAlert() extrahiert. Das ist der erste Schritt der Auslagerung. Die nächsten Schritte können die Methode in weitere Teile unterteilen. Zum Beispiel einer Methode für Artikel, die Ablaufen und eine Funktion für Artikel, die eine geringe Menge haben.

Fridge-Details:

```
ngOnInit(): void {
   this.route.paramMap.subscribe(params => {
     this.itemId = Number(params!.get('id'));
   //  this.item = this.fridgeService.getFridgeItemById(this.itemId);
     this.fridgeService.getItemById(this.itemId).subscribe(item => {
       this.item = item;
        this.categoryImg.forEach(element => {
         if (item.category == element.category) {
           this.img = element.img;
           this.description = element.description;
           this.chips = element.chips;
         }
       });
        this.quantityValue = item.quantity;
   this.dateValue = item.expiryDate;
   this.nameValue = item.name;
   this.notesValue = item.notes;
   this.amountValue = item.amount;
   this.kcalValue = item.kcal;
   this.sugarValue = item.sugar;
   this.fatValue = item.fat;
   this.proteinValue = item.protein;
   this.carbsValue = item.carbs;
     })
   });
 }
```

Zu

```
 ngOnInit(): void {
   this.setItemDetailData();
 }

 setItemDetailData() {
   this.route.paramMap.subscribe(params => {
     this.itemId = Number(params!.get('id'));
   //  this.item = this.fridgeService.getFridgeItemById(this.itemId);
     this.fridgeService.getItemById(this.itemId).subscribe(item => {
       this.item = item;
        this.categoryImg.forEach(element => {
         if (item.category == element.category) {
           this.img = element.img;
           this.description = element.description;
           this.chips = element.chips;
         }
       });
        this.quantityValue = item.quantity;
```

```
    this.dateValue = item.expiryDate;
    this.nameValue = item.name;
    this.notesValue = item.notes;
    this.amountValue = item.amount;
    this.kcalValue = item.kcal;
    this.sugarValue = item.sugar;
    this.fatValue = item.fat;
    this.proteinValue = item.protein;
    this.carbsValue = item.carbs;
      })
    });
  }
```

Hierbei ist wieder der erste Schritt die ngOnInit()-Methode zu verkleinern. Eine neue Methode setItemDetailData() wird dafür erstellt. Diese beinhaltet alle Informationen zu dem jeweiligen Artikel mit der übergebenen ‘id’. ngOnInit() ruft daraufhin nur noch diese Funktion auf.

Fridge-Shoppinglist:

```
 updateShoppingList(item:any) {
    this.fridgeService.getShoppinglistData().subscribe((data: any) => {
      this.fridgeItems = data;
    //  this.lowQuantityItems = this.fridgeItems.filter(item => item.quantity
< 3);
      this.newItemId = this.fridgeItems.length + 1000;
        this.newFridgeItem = {id: 1000 + this.newItemId, name: item.name,
quantity: item.quantity, expiryDate: item.date, category: item.category,
notes: item.notes, amount: item.amount, kcal: item.kcal, sugar:
item.sugar,fat: item.fat,protein: item.protein, carbs: item.carbs };
        this.fridgeItems.push(this.newFridgeItem);

      this.dataSource3.data = this.fridgeItems;


    this.fridgeService.updateShoppingList(this.fridgeItems).subscribe({
      next: () => {
        console.log('Daten erfolgreich aktualisiert.');
      },
      error: (error) => {
        console.error(error);
      }
    });

    });
```

Zu

```
 updateShoppingList(item:any) {
    this.fridgeService.getShoppinglistData().subscribe((data: any) => {
      this.fridgeItems = data;
      this.addShoppingItem(item);

    this.fridgeService.updateShoppingList(this.fridgeItems).subscribe({
      next: () => {
        console.log('Daten erfolgreich aktualisiert.');
      },
      error: (error) => {
        console.error(error);
      }
    });

    });
  }

  addShoppingItem(item:any) {
    this.newItemId = this.fridgeItems.length + 1000;
    this.newFridgeItem = {id: this.newItemId, name: item.name, quantity:
item.quantity, expiryDate: item.date, category: item.category, notes:
item.notes, amount: item.amount, kcal: item.kcal, sugar: item.sugar,fat:
item.fat,protein: item.protein, carbs: item.carbs };
    this.fridgeItems.push(this.newFridgeItem);
    this.dataSource3.data = this.fridgeItems;
  }
```

Hier wurde die Methode updateShoppingList() um den Teil, welcher dafür zuständig ist, einen neuen Artikel hinzuzufügen, erleichtert. addShoppingItem() übernimmt diese Funktion und wird bei updateShoppingList() aufgerufen.

Fridge-Stats:

```
 ngOnInit(): void {
    this.getProfileData();
    this.fridgeService.getItemsData().subscribe((data: any) => {
      this.fridgeItems = data;
      this.totalItems = this.fridgeItems.length;

      this.totalUsedCapacity = this.fridgeItems.reduce((total, item) => total
+ item.quantity, 0);
      this.totalFreeCapacity = this.totalCapacity - this.totalUsedCapacity;

  // Calculate average, minimum and maximum quantities of all items
      const totalQuantities = this.fridgeItems.reduce((total, item) => total +
item.quantity, 0);
      this.avgQuantity = totalQuantities / this.totalItems;
      this.minQuantity = Math.min(...this.fridgeItems.map(item =>
item.quantity));
```

```
      this.maxQuantity = Math.max(...this.fridgeItems.map(item =>
item.quantity));

 const filteredItems = this.fridgeItems.filter(item => item.category ===
this.fridgeItems[0].category);

    for (let index = 0; index < this.fridgeService.categories.length; index++)
{
      var filter = this.fridgeItems.filter(item => item.category ===
this.fridgeService.categories[index]);
      this.mappedCategory.push({name: this.fridgeService.categories[index],
quantity: filter.length})
    }
    console.log(this.mappedCategory)

    this.mappedCategory.forEach(element => {
      this.labels.push(element.name)
    });

    this.mappedCategory.forEach(element => {
      this.data.push(element.quantity)
    });

    this.createPieChart();
    this.createCategoryChart();
    });

 //   this.fridgeItems = this.fridgeService.getFridgeItems();


  }
```

Zu

```
 ngOnInit(): void {
    this.getProfileData();
    this.createCharts();
  }

  createCharts() {
    this.fridgeService.getItemsData().subscribe((data: any) => {
      this.fridgeItems = data;

      this.setEntireContentData();
      this.setCategoryData();

      this.createPieChart();
      this.createCategoryChart();
    });
  }
```

```
  setEntireContentData() {
    this.totalItems = this.fridgeItems.length;

    this.totalUsedCapacity = this.fridgeItems.reduce((total, item) => total +
item.quantity, 0);
    this.totalFreeCapacity = this.totalCapacity - this.totalUsedCapacity;

// Calculate average, minimum and maximum quantities of all items
    const totalQuantities = this.fridgeItems.reduce((total, item) => total +
item.quantity, 0);
    this.avgQuantity = totalQuantities / this.totalItems;
    this.minQuantity = Math.min(...this.fridgeItems.map(item =>
item.quantity));
    this.maxQuantity = Math.max(...this.fridgeItems.map(item =>
item.quantity));
  }

  setCategoryData() {
    for (let index = 0; index < this.fridgeService.categories.length; index++)
{
      var filter = this.fridgeItems.filter(item => item.category ===
this.fridgeService.categories[index]);
      this.mappedCategory.push({name: this.fridgeService.categories[index],
quantity: filter.length})
    }
    console.log(this.mappedCategory)

    this.mappedCategory.forEach(element => {
      this.labels.push(element.name)
    });

    this.mappedCategory.forEach(element => {
      this.data.push(element.quantity)
    });
  }
```

Der Inhalt von ngOnInit() wurde in mehrere Funktionen aufgeteilt. Die Methode CreateCharts() ist zuständig für die Erstellung der Diagramme. Die Methode setEntireContentData() befüllt das erste Diagramm mit Daten und die Methode setCategoryData() befüllt das zweite Diagramm mit Daten. Der Code ist nun übersichtlicher geworden und jede Funktion erklärt, wofür sie zuständig ist.

2.2.2 Rename Method

Rename Method kümmert sich um die Änderung der Namen von Methoden bzw. Variablen. Diese sollen aussagekräftiger werden.

Fridge-List:

```
  updateItemsData(item:any) {
    this.newItemId = this.fridgeItems.length + 1;
    console.log(this.newItemId)
    this.newFridgeItem = {id: this.newItemId, name: item.name, quantity:
item.quantity, expiryDate: item.date, category: item.category, notes:
item.notes, amount: item.amount, kcal: item.kcal, sugar: item.sugar,fat:
item.fat,protein: item.protein, carbs: item.carbs };
    this.fridgeService.updateItemsData(this.newFridgeItem).subscribe({
      next: () => {
        console.log('Daten erfolgreich aktualisiert.');
      },
      error: (error) => {
        console.error(error);
      }
    });
    this.fridgeItems.push(this.newFridgeItem);
    this.dataSource.data = this.fridgeItems;
  }
```

Zu

```
 addItem(item:any) {
    this.newItemId = this.fridgeItems.length + 1;
    console.log(this.newItemId)
    this.newFridgeItem = {id: this.newItemId, name: item.name, quantity:
item.quantity, expiryDate: item.date, category: item.category, notes:
item.notes, amount: item.amount, kcal: item.kcal, sugar: item.sugar,fat:
item.fat,protein: item.protein, carbs: item.carbs };
    this.fridgeService.addItem(this.newFridgeItem).subscribe({
      next: () => {
        console.log('Daten erfolgreich aktualisiert.');
      },
      error: (error) => {
        console.error(error);
      }
    });
    this.fridgeItems.push(this.newFridgeItem);
    this.dataSource.data = this.fridgeItems;
  }
```

Die Methode updateItemsData() greift auf die HTTP-Anfrage “POST” zu und ist damit eher ein Hinzufügen (add) als ein Updaten. Hinzufügen beschreibt die Funktionalität besser und genauer.

```
openDialog() {
 const dialogRef = this.dialog.open(FridgeAddComponent);

 dialogRef.afterClosed().subscribe(result => {

   if (result != false) {
     this.updateItemsData(result);
   }

 });
}
```

Zu

```
openAddItemDialog() {
 const dialogRef = this.dialog.open(FridgeAddComponent);

 dialogRef.afterClosed().subscribe(result => {

   if (result != false) {
     this.addItem(result);
   }

 });
}
```

Der Name openDialog() ist zu ungenau, da der Dialog für das Hinzufügen von Artikeln aufgerufen wird. Daher wird er zu openAddItemDialog() geändert.

Fridge-Profile:

```
getData() {
    this.fridgeService.getData().subscribe((data: any) => {
      this.data = data;
      this.units = this.data.units;
      console.log(this.units)
      this.capacity = this.data.capacity;
      this.measurementSetting = this.data.measurementSetting;
   this.languageSetting = this.data.languageSetting;
  this.localizationSetting = this.data.localizationSetting;
```

```
  this.autoAddSetting = this.data.autoAddSetting;
  this.alertSetting = this.data.alertSetting;
  this.username = this.data.username
  this.usermail = this.data.usermail
  this.userpassword = this.data.userpassword;

    });
  }
```

Zu

```
 getProfilConfigurationData() {
    this.fridgeService.getProfileConfigurationData().subscribe((data: any) =>
{
      this.data = data;
      this.units = this.data.units;
      console.log(this.units)
      this.capacity = this.data.capacity;
      this.measurementSetting = this.data.measurementSetting;
   this.languageSetting = this.data.languageSetting;
  this.localizationSetting = this.data.localizationSetting;
  this.autoAddSetting = this.data.autoAddSetting;
  this.alertSetting = this.data.alertSetting;
  this.username = this.data.username
  this.usermail = this.data.usermail
  this.userpassword = this.data.userpassword;

    });
  }
```

Der Name getData() ist eine zu ungenaue Beschreibung, da nicht klar wird welche Daten man bekommt. Der Name getProfilConfigurationData() ist eine viel bessere Beschreibung. Nun ist verständlicher, welche Daten man erhält.

Fridge-Shoppinglist:

```
 dataSource = new MatTableDataSource<FridgeItem>();

  dataSource2 = new MatTableDataSource<FridgeItem>();
  dataSource3 = new MatTableDataSource<FridgeItem>();
```

Zu

```
 lowQuantityDataSource = new MatTableDataSource<FridgeItem>();

  soonToExpireDataSource = new MatTableDataSource<FridgeItem>();
  shoppingListDataSource = new MatTableDataSource<FridgeItem>();
```

DataSource ist eine zu ungenaue Beschreibung, welche Datenquelle verwendet wird.

lowQuantityDataSource, soonToExpireDataSource und shoppingDataSource beschreiben die Datenquellen viel besser.

```
 createPieChart() {
    this.chart = new Chart("MyChart", {
      type: 'bar', //this denotes tha type of chart

      data: {// values on X-Axis
        labels: ['Gesamtinhalt', 'Gesamtkapazität', 'Genutzte
Kapazität','Freie Kapazität',
                 'Durchschnittliche Anzahl', 'Minimale Anzahl', 'Maximale
Anzahl'],
         datasets: [
          {
            label: "Anzahl",
            data:
[this.totalItems,  this.totalCapacity,  this.totalUsedCapacity ,
this.totalFreeCapacity , this.avgQuantity,
              this.minQuantity, this.maxQuantity],
            backgroundColor: '#134E5E'
          },

        ]
      },
      options: {
        aspectRatio:2
      }


    });
  }
```

Zu

```
 createEnitreContentChart() {
    this.chart = new Chart("MyChart", {
      type: 'bar', //this denotes tha type of chart

      data: {// values on X-Axis
        labels: ['Gesamtinhalt', 'Gesamtkapazität', 'Genutzte
Kapazität','Freie Kapazität',
                 'Durchschnittliche Anzahl', 'Minimale Anzahl', 'Maximale
Anzahl'],
         datasets: [
          {
            label: "Anzahl",
```

```
            data:
[this.totalItems,  this.totalCapacity,  this.totalUsedCapacity ,
this.totalFreeCapacity , this.avgQuantity,
              this.minQuantity, this.maxQuantity],
            backgroundColor: '#134E5E'
          },

        ]
      },
      options: {
        aspectRatio:2
      }


    });
  }
```

Der Name createPieChart() beschreibt ein Kuchendiagramm, obwohl es ein Balkendiagramm mit den Informationen zu allen Artikeln ist. Der Name createEntireContentChart() beschreibt dies besser.

2.2.3 Replace Temp with Query

“Const” sind temporär und haben keinen spezifischen Datentyp. Deswegen sollten sie in eine eigene Methode ausgelagert werden oder einen Datentyp erhalten.

Fridge-Shoppinglist, Fridge-Stats:

```
openDialog() {
    const dialogRef = this.dialog.open(FridgeAddComponent);
    dialogRef.afterClosed().subscribe(result => {
      if (result != false) {
        this.updateShoppingList(result);
      }
    });

  }
 ngOnInit(): void {
    this.getProfileData();
    this.fridgeService.getItemsData().subscribe((data: any) => {
      this.fridgeItems = data;
      this.totalItems = this.fridgeItems.length;

      this.totalUsedCapacity = this.fridgeItems.reduce((total, item) => total
+ item.quantity, 0);
      this.totalFreeCapacity = this.totalCapacity - this.totalUsedCapacity;
```

```
  // Calculate average, minimum and maximum quantities of all items
      const totalQuantities = this.fridgeItems.reduce((total, item) => total +
item.quantity, 0);
      this.avgQuantity = totalQuantities / this.totalItems;
      this.minQuantity = Math.min(...this.fridgeItems.map(item =>
item.quantity));
      this.maxQuantity = Math.max(...this.fridgeItems.map(item =>
item.quantity));

 const filteredItems = this.fridgeItems.filter(item => item.category ===
this.fridgeItems[0].category);

    for (let index = 0; index < this.fridgeService.categories.length; index++)
{
      var filter = this.fridgeItems.filter(item => item.category ===
this.fridgeService.categories[index]);
      this.mappedCategory.push({name: this.fridgeService.categories[index],
quantity: filter.length})
    }
    console.log(this.mappedCategory)

    this.mappedCategory.forEach(element => {
      this.labels.push(element.name)
    });

    this.mappedCategory.forEach(element => {
      this.data.push(element.quantity)
    });

    this.createPieChart();
    this.createCategoryChart();
    });

 //   this.fridgeItems = this.fridgeService.getFridgeItems();


  }
```

Zu

```
 today: Date = new Date();
  expirationDate: Date = new Date();
  timeDiff: number = 0;
  daysUntilExpiration: number = 0;
  index: number = 0;


soonToExpireList()  {
    this.fridgeService.getItemsData().subscribe((data: any) => {
```

```
    this.fridgeItems = data;
    this.soonToExpireItems = this.fridgeItems.filter(item => {
    this.today = new Date();
    this.expirationDate = new Date(item.expiryDate);
    this.timeDiff = this.expirationDate.getTime() - this.today.getTime();
    this.daysUntilExpiration = Math.ceil(this.timeDiff / (1000 * 3600 * 24));
      return this.daysUntilExpiration <= 3; // Show items that will expire in
the next 3 days
    });
    this.soonToExpireDataSource.data = this.soonToExpireItems;
    });


  }

 totalQuantities: number = 0;


this.totalUsedCapacity = this.fridgeItems.reduce((total, item) => total +
item.quantity, 0);
    this.totalFreeCapacity = this.totalCapacity - this.totalUsedCapacity;
```

Hierbei werden die “const” mit einem Datentyp ersetzt und die zuvor angewandte Auslagerung von ngOnInit() benutzt. Beispielsweise hat “today” den Datentyp “Date”.

FridgeService:

```
isLowOnQuantity(item: FridgeItem): boolean {
    const minQuantity = 2; // hier können Sie die Mindestmenge festlegen
    return item.quantity < minQuantity;
  }

   getDaysUntilExpiration(item: FridgeItem): number {
    const expiryDate = new Date(item.expiryDate);
    const today = new Date();
    const timeDiff = expiryDate.getTime() - today.getTime();
    const daysDiff = Math.ceil(timeDiff / (1000 * 3600 * 24));
    return daysDiff;
  }
```

Zu

```
  minQuantity: number = 0;
  expiryDate: Date = new Date();
  today: Date = new Date();
  timeDiff: number = 0;
```

```
  daysDiff: number = 0;



isLowOnQuantity(item: FridgeItem): boolean {
    this.minQuantity = 2; // hier können Sie die Mindestmenge festlegen
    return item.quantity < this.minQuantity;
  }

   getDaysUntilExpiration(item: FridgeItem): number {
    this.expiryDate = new Date(item.expiryDate);
    this.today = new Date();
    this.timeDiff = this.expiryDate.getTime() - this.today.getTime();
    this.daysDiff = Math.ceil(this.timeDiff / (1000 * 3600 * 24));
    return this.daysDiff;
  }
```

Hier werden die “const” wieder durch spezifische Datentypen ersetzt.

2.2.4 Replace Error Code with Exception

Console.error(error) wird durch eine Exception ersetzt. Diese Exception kann eine eigene Klasse sein und angepasst werden. Die Behandlung von Fehlern wird dadurch spezieller.

Fridge-Shoppinglist:

```
 updateShoppingList(item:any) {
    this.fridgeService.getShoppinglistData().subscribe((data: any) => {
      this.fridgeItems = data;
    //  this.lowQuantityItems = this.fridgeItems.filter(item => item.quantity
< 3);
      this.newItemId = this.fridgeItems.length + 1000;
        this.newFridgeItem = {id: 1000 + this.newItemId, name: item.name,
quantity: item.quantity, expiryDate: item.date, category: item.category,
notes: item.notes, amount: item.amount, kcal: item.kcal, sugar:
item.sugar,fat: item.fat,protein: item.protein, carbs: item.carbs };
        this.fridgeItems.push(this.newFridgeItem);

      this.dataSource3.data = this.fridgeItems;


    this.fridgeService.updateShoppingList(this.fridgeItems).subscribe({
      next: () => {
```

```
        console.log('Daten erfolgreich aktualisiert.');
      },
      error: (error) => {
        console.error(error);
      }
    });

    });
```

Zu

```
  updateShoppingList(item:any) {
    this.fridgeService.getShoppinglistData().subscribe((data: any) => {
      this.fridgeItems = data;
      this.addShoppingItem(item);

    this.fridgeService.updateShoppingList(this.fridgeItems).subscribe({
      next: () => {
        console.log('Daten erfolgreich aktualisiert.');
      },
      error: (error) => {
        throw error;
      }
    });
    });
  }
```

```
addItem(item:any) {
    this.newItemId = this.fridgeItems.length + 1;
    console.log(this.newItemId)
    this.newFridgeItem = {id: this.newItemId, name: item.name, quantity:
item.quantity, expiryDate: item.date, category: item.category, notes:
item.notes, amount: item.amount, kcal: item.kcal, sugar: item.sugar,fat:
item.fat,protein: item.protein, carbs: item.carbs };
    this.fridgeService.addItem(this.newFridgeItem).subscribe({
      next: () => {
        console.log('Daten erfolgreich aktualisiert.');
      },
      error: (error) => {
        throw error
      }
    });
    this.fridgeItems.push(this.newFridgeItem);
    this.dataSource.data = this.fridgeItems;
  }
```

In diesem aktualisierten Code wird die Fehlerbehandlungsroutine geändert, um eine Ausnahme mit dem entsprechenden Fehler zu werfen, anstatt die Fehlermeldung mit "console.error" auszugeben. Dadurch wird der Fehler an die Aufrufe der Methode weitergegeben, die ihn dann entsprechend behandeln können.

2.2.5 Methoden, die keinen Zweck erfüllen

Fridge-List:

```
onSelect(item: FridgeItem): void {
   this.selectedItem = item;
 }
```

Die Methode onSelect() wird nicht mehr verwendet und kann deshalb entfernt werden.

Fridge-Details:

```
updateFridgeItem(item: FridgeItem): void {
   // Implement the update functionality here
 }

 deleteFridgeItem(itemId: number): void {
   // Implement the delete functionality here
 }
```

Die Methoden updateFridgeItem() und deleteFridgeItem() werden nicht mehr verwendet und können deshalb entfernt werden.

Fridge-Shoppinglist:

```
delete(deleteItem:FridgeItem): void {
   console.log(deleteItem);
   const index = this.lowQuantityItems.findIndex(item => item.id ===
deleteItem.id);
    if (index !== -1) {
   this.lowQuantityItems.splice(index, 1);
   this.dataSource.data = this.lowQuantityItems;
     }
  }
```

Die delete() Methode wurde durch eine andere ersetzt und kann deswegen entfernt werden.

3. Entwurfsmuster

3.1 Entwurfsmuster Beobachter

Allgemeine observer-pattern in angular:

```
import { Component } from '@angular/core';
import { Observable, Subject } from 'rxjs';

@Component({
  selector: 'my-component',
  template: `
    <h1>{{title}}</h1>
    <button (click)="updateMessage()">Update Message</button>
    <p>{{message}}</p>
  `,
})
export class MyComponent {
  title = 'Observer Pattern Example';
  message: string;
  messageSubject = new Subject<string>();

  constructor() {
    this.messageSubject.subscribe((message: string) => {
      this.message = message;
    });
  }

  updateMessage() {
    const newMessage = 'New message at ' + new Date().toLocaleTimeString();
    this.messageSubject.next(newMessage);
  }
}
```

In diesem Beispiel wird die MyComponent-Klasse definiert, die ein Subject namens messageSubject als private Instanzvariable hat. Das Subject ist ein Teil des Observable, das als Sender von Benachrichtigungen fungiert, wenn eine Änderung an den Daten erfolgt.

Die MyComponent-Klasse hat auch eine Methode namens updateMessage, die eine neue Nachricht erzeugt und sie an das messageSubject sendet. Wenn eine neue Nachricht empfangen wird, aktualisiert die subscribe-Methode die message-Variable, die dann in der Vorlage gerendert wird.

In unserem Code wird das Entwurfsmuster Beobachter über Observables genutzt.

```
  addItem(newData: FridgeItem): Observable<any> {
    return this.http.post(this.itemsurl, newData);
  }
```

Im Fridge-Service werden bei einigen Methoden Observables zurückgegeben, um diese dann in den jeweiligen Komponenten zu verwenden. Der Zugriff auf die Observables erfolgt über die Methode .subscribe(). Daraufhin kann mit ihnen weitergearbeitet werden.

Hier einige Beispiele:

```
getItemsData(): void {
    this.fridgeService.getItemsData().subscribe((data: any) => {
      console.log(data);
      this.fridgeItems = data;
      this.dataSource.data = this.fridgeItems;
    });
```

Zugriff auf Artikel

```
  deleteItem(id: number): void {
    this.fridgeService.deleteItem(id).subscribe(() => {
      this.getItemsData();
    });
    this.fridgeItems = this.fridgeItems.filter(item => item.id !== id);
    this.dataSource.data = this.fridgeItems;
  }
```

Löschen von Artikel

```
 openAddItemDialog() {
  const dialogRef = this.dialog.open(FridgeAddComponent);

  dialogRef.afterClosed().subscribe(result => {

    if (result != false) {
      this.addItem(result);
    }

  });
}
```

Zugriff auf das Ergebnis des Dialogs

```
setItemDetailData() {
    this.route.paramMap.subscribe(params => {
      this.itemId = Number(params!.get('id'));
    //  this.item = this.fridgeService.getFridgeItemById(this.itemId);
      this.fridgeService.getItemById(this.itemId).subscribe(item => {
        this.item = item;
         this.categoryImg.forEach(element => {
          if (item.category == element.category) {
            this.img = element.img;
            this.description = element.description;
            this.chips = element.chips;
          }
        });
         this.quantityValue = item.quantity;
    this.dateValue = item.expiryDate;
    this.nameValue = item.name;
    this.notesValue = item.notes;
    this.amountValue = item.amount;
    this.kcalValue = item.kcal;
    this.sugarValue = item.sugar;
    this.fatValue = item.fat;
    this.proteinValue = item.protein;
    this.carbsValue = item.carbs;
      })
    });
  }
```

Zugriff auf einen einzelnen Artikel über seine ID und Zuweisung der Informationen des Artikels.

```
soonToExpireList()  {
    this.fridgeService.getItemsData().subscribe((data: any) => {
    this.fridgeItems = data;
    this.soonToExpireItems = this.fridgeItems.filter(item => {
    this.today = new Date();
    this.expirationDate = new Date(item.expiryDate);
    this.timeDiff = this.expirationDate.getTime() - this.today.getTime();
    this.daysUntilExpiration = Math.ceil(this.timeDiff / (1000 * 3600 * 24));
      return this.daysUntilExpiration <= 3; // Show items that will expire in
the next 3 days
    });
    this.soonToExpireDataSource.data = this.soonToExpireItems;
    });


  }
```

Zugriff auf die Artikel und Filterung dieser nach Ablaufdatum

3.2 Entwurfsmuster Dekorierer

Angular bietet eine Vielzahl von Dekoratoren, die zur Definition von Klassen, Methoden und Eigenschaften verwendet werden können. Hier sind einige der wichtigsten Dekoratoren in Angular:

@Component: Definiert eine Komponente in Angular.

@Directive: Definiert eine Direktive in Angular.

@Injectable: Definiert eine Service-Klasse in Angular, die in anderen Komponenten und Diensten verwendet werden kann.

@Input: Definiert eine Eingabeeigenschaft in einer Komponente oder Direktive.

@Output: Definiert eine Ausgabeereignis in einer Komponente oder Direktive.

@ViewChild: Erlaubt den Zugriff auf eine untergeordnete Komponente oder Direktive in einer Vorlage.

@ContentChild: Erlaubt den Zugriff auf eine untergeordnete Komponente oder Direktive in einem Projektionsbereich in der Vorlage.

@HostListener: Bindet einen Ereignishandler an das Host-Element einer Komponente.

@HostBinding: Bindet eine Eigenschaft an das Host-Element einer Komponente.

Diese Dekoratoren sind alle Teile des Angular-Frameworks und bieten eine einfache Möglichkeit, Komponenten, Direktiven und Dienste zu definieren, um komplexe Anwendungen zu erstellen.

Für alle Komponenten wird immer der Dekorator @Component hinzugefügt. Er definiert den selector (wie die Komponente aufgerufen wird), die templateUrl (die View der Komponente) und die styleUrls der Komponente

```
@Component({
  selector: 'app-fridge-list',
  templateUrl: './fridge-list.component.html',
  styleUrls: ['./fridge-list.component.scss']
})
export class FridgeListComponent implements OnInit {
```

Der Dekorator @NgModule importiert und exportiert Module, die in der Anwendung eingesetzt werden.

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Der Dekorator @Injectable definiert eine Service Klasse. Er wird bei FridgeService benutzt und für ‘root’ also die Wurzel der Anwendung bereitgestellt.

```
@Injectable({
  providedIn: 'root'
})
export class FridgeService {
```

3.3 Entwurfsmuster Singleton

In Angular wird ein Singleton-Muster verwendet, um sicherzustellen, dass eine einzige Instanz einer Klasse während der Lebensdauer der Anwendung verwendet wird. Dies kann nützlich sein, wenn eine Ressource oder ein Service erstellet wird, der global in der Anwendung verfügbar sein soll.

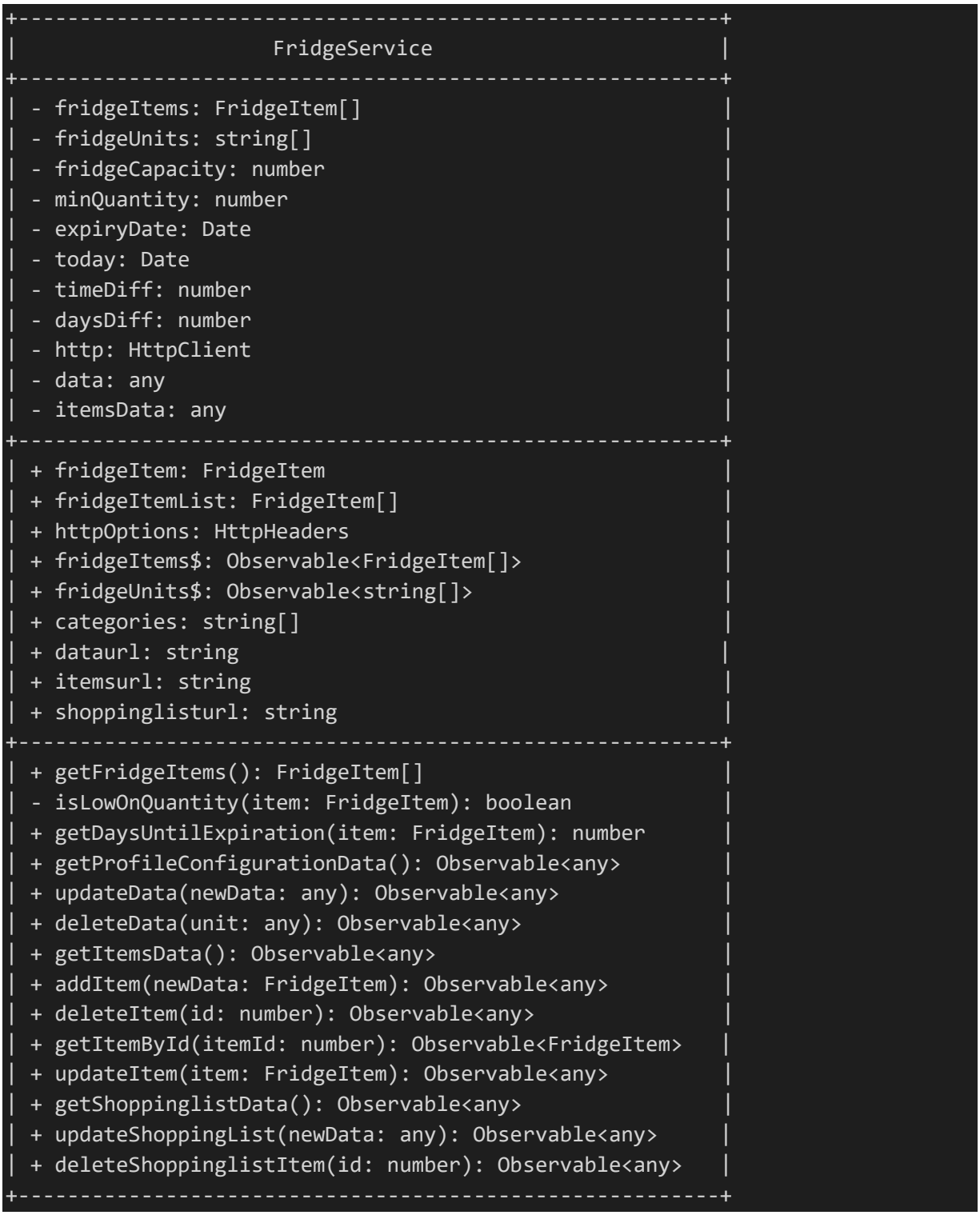
Allgemein:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class MySingletonService {
  private message = 'Hello from MySingletonService';

  getMessage(): string {
    return this.message;
  }
}
```

Fridge-Service:



Das Singleton-Pattern wird in Angular standardmäßig durch die Verwendung des providedIn: 'root'-Dekorators in der @Injectable-Annotation erreicht. Dadurch wird der FridgeService als Singleton-Dienst bereitgestellt und kann überall in der Anwendung injiziert werden. Das UML-Diagramm spiegelt dies wider, indem FridgeService keine öffentlichen Konstruktoren enthält und die Instanz des Dienstes über die Angular-Abhängigkeitsinjektion bereitgestellt wird.

4. Programming Principles

4.1 SOLID

SOLID ist eine Sammlung von Prinzipien, die sich auf die objektorientierte Programmierung anwenden lassen, um einen flexiblen und leicht wartbaren Code zu erstellen. Um eine saubere Architektur zu schaffen und die Wartbarkeit zu verbessern, können diese Prinzipien auch auf Angular-Anwendungen angewendet werden.

Single Responsibility Principle (SRP):

Eine Klasse sollte nur eine einzige Verantwortung haben. In Angular wird dieses Prinzip durch die Erstellung von Services mit spezifischen Aufgaben umgesetzt. Zum Beispiel kann es einen Service geben, der nur für die Verarbeitung von HTTP-Anfragen zuständig ist, während ein anderer Service nur für die Verwaltung des Zustands der Anwendung zuständig ist.

In unserer Anwendung übernimmt das der Service: Fridge-Service. Er ist für die Verarbeitung von HTTP-Anfragen zuständig.

```
@Injectable({
  providedIn: 'root'
})
export class FridgeService {
```

Open-Closed Principle (OCP):

Eine Klasse sollte für Erweiterungen offen, aber für Änderungen geschlossen sein. In Angular wird dieses Prinzip umgesetzt, indem Dependency Injection verwendet und abstrakte Klassen oder Interfaces erstellt werden. Auf diese Weise kann eine Klasse durch Hinzufügen neuer Abhängigkeiten erweitert werden, ohne dass Änderungen am Kerncode der Klasse erforderlich sind.

In unserer Anwendung werden dafür Interfaces verwendet. Das Interface FridgeItem gebt die Struktur der einzelnen Artikel wieder. Die Komponente können mit diesem Interface erweitert werden, ohne dass Änderungen am Kerncode erfolgen müssen.

```
export interface FridgeItem {
    id: number;
    name: string;
    quantity: number;
    expiryDate: Date;
    category: string;
    notes: string;
    amount: number;
```

```
    kcal: number;
    sugar: number;
    fat: number;
    protein: number;
    carbs: number;
  }
```

Liskov Substitution Principle (LSP):

Es ist wichtig, dass eine abgeleitete Klasse jederzeit durch ihre Basisklasse ersetzt werden kann. In Angular lässt sich dieses Prinzip umsetzen, indem die TypeScript-Typisierung verwendet wird, um sicherzustellen, dass die Methoden und Eigenschaften der abgeleiteten Klasse mit den Methoden und Eigenschaften der Basisklasse übereinstimmen.

Beispiel in unserer Anwendung:

Typisierung und ngOnInit()

```
export class FridgeListComponent implements OnInit {
```

FridgeListComponent stimmt mit den Methoden der Klasse OnInit überein. FirdgeListComponent erbt die Methode ngOnInit().

```
 newFridgeItem: FridgeItem = { id: 1, name: 'Milch', quantity: 2, expiryDate:
new Date(2023, 4, 1), category: "Milchprodukte", notes: "test", amount: 12,
kcal: 1, sugar: 1,fat: 1,protein: 1, carbs: 1  };
  newItemId: number = 0;
  displayedColumns: string[] = ['name', 'quantity', 'expiryDate'];
  categories: string[] = ['Milchprodukte','Eier', 'Gemüse', 'Obst', 'Fleisch',
'Fisch', 'Getränke', 'Teigwaren','Soßen & Dressing'];
  dataSource: MatTableDataSource<FridgeItem>  = new
MatTableDataSource<FridgeItem>();

 fridgeItems: FridgeItem[] = [];

  selectedItem: any;

 ngOnInit(): void {
    this.getItemsData();
  }
```

Interface Segregation Principle (ISP):

Eine Klasse sollte nicht auf unnötige Schnittstellen angewiesen sein. In Angular lässt sich dieses Prinzip umsetzen, indem man spezifische Interfaces für Services und Komponenten verwendet, um zu gewährleisten, dass diese nur die Methoden und Eigenschaften haben, die sie brauchen.

Dies wird in unserer Anwendung über den Import der Interfaces geregelt. Nur benötigte Interfaces werden importiert.

```
import { FridgeItem } from './fridge-item.model';
```

Dependency Inversion Principle (DIP):

Statt von konkreten Klassen sollten Abhängigkeiten von abstrakten Klassen oder Interfaces abhängen. In Angular kann dieses Prinzip durch die Verwendung von Dependency Injection umgesetzt werden. Für Abhängigkeiten können abstrakte Klassen oder Interfaces verwendet werden. Dies hat den Vorteil, dass die Anwendung flexibler wird und die Erstellung von Mock-Objekten für Tests einfacher wird.

4.2 GRASP

GRASP (General Responsibility Assignment Software Patterns) ist ein Ansatz zur Verteilung von Aufgaben innerhalb eines objektorientierten Systems, bei dem der Schwerpunkt auf der Definition von Objektverantwortlichkeiten und -zuständigkeiten liegt.

In Angular kann GRASP auf verschiedene Arten implementiert werden:

Controller:
Komponenten in Angular, die für die Verarbeitung von Benutzereingaben und die Aktualisierung der View verantwortlich sind, können als Controller betrachtet werden.

Beispiel Fridge-List Komponente als Controller:

```
@Component({
  selector: 'app-fridge-list',
  templateUrl: './fridge-list.component.html',
  styleUrls: ['./fridge-list.component.scss']
})
export class FridgeListComponent implements OnInit {
```

```
  newFridgeItem: FridgeItem = { id: 1, name: 'Milch', quantity: 2, expiryDate:
new Date(2023, 4, 1), category: "Milchprodukte", notes: "test", amount: 12,
kcal: 1, sugar: 1,fat: 1,protein: 1, carbs: 1  };
  newItemId: number = 0;
  displayedColumns: string[] = ['name', 'quantity', 'expiryDate'];
  categories: string[] = ['Milchprodukte','Eier', 'Gemüse', 'Obst', 'Fleisch',
'Fisch', 'Getränke', 'Teigwaren','Soßen & Dressing'];
  dataSource: MatTableDataSource<FridgeItem>  = new
MatTableDataSource<FridgeItem>();

 fridgeItems: FridgeItem[] = [];

  selectedItem: any;



  constructor(private fridgeService: FridgeService,public dialog: MatDialog,
private cdRef: ChangeDetectorRef) { }

  ngOnInit(): void {
    this.getItemsData();
  }
```

Experten:

Die Verantwortung für eine bestimmte Aufgabe sollte dem Objekt mit entsprechendem Wissen und Erfahrung zugewiesen werden. Beispielsweise sollte eine Validierungsklasse für die Eingabevalidierung zuständig sein.

Ein Beispiel in unserer Anwendung wäre die Fridge-Add Komponente Sie ist der Experte für das Hinzufügen von neuen Artikeln. Die Artikel werden über ein Formular mit Inhalt gefüllt und weitergegeben.

„.ts“-Datei für die Steuerung des Formulars und .html-Datei für die Gestaltung des Formulars:

```
TS:

import { Component, OnInit } from '@angular/core';

import { FridgeService } from '../fridge.service';
import { MatDialogRef } from '@angular/material/dialog';

@Component({
  selector: 'app-fridge-add',
  templateUrl: './fridge-add.component.html',
  styleUrls: ['./fridge-add.component.scss']
})
```

```
export class FridgeAddComponent implements OnInit {


  name: string = "";
  quantity: number = 0;
  date: Date = new Date();
  category: string = "";
  notes: string = "";
  amount: number = 0;
  kcal: number = 0;
  sugar: number = 0;
  fat: number = 0;
  protein: number = 0;
  carbs: number = 0;
  categories: string[] = [];
  backdropClick$: any;



  constructor(private fridgeService: FridgeService,public dialogRef:
MatDialogRef<FridgeAddComponent>) { }



ngOnInit(): void {
    this.categories = this.fridgeService.categories;
    this.getBackdropClick();
  }

getBackdropClick() {
  this.backdropClick$ = this.dialogRef.backdropClick();
  if (this.backdropClick$) {
    this.backdropClick$.subscribe(() => {
      this.cancel();
    });
  }
}
  cancel(): void {
    this.dialogRef.close(false);
  }
}
```

HTML:

```
<h2 mat-dialog-title>Artikel hinzufügen</h2>
<mat-divider></mat-divider>
<mat-dialog-content class="dialogContent">
  <mat-grid-list cols="2" rowHeight="80px" >

    <mat-grid-tile [colspan]=1 [rowspan]=6>
```

```
  <div class="group1">
    <h3>Allgemein</h3>
  <mat-form-field>
    <mat-label>Name</mat-label>
    <input matInput [(ngModel)]="name">
  </mat-form-field>
  <mat-form-field>
    <mat-label>Anzahl</mat-label>
    <input matInput [(ngModel)]="quantity"  type="number" min="1">
  </mat-form-field>
  <mat-form-field appearance="fill">
    <mat-label>Wähle ein Datum</mat-label>
    <input matInput [matDatepicker]="picker" [(ngModel)]="date">
    <mat-hint>MM/DD/YYYY</mat-hint>
    <mat-datepicker-toggle matIconSuffix [for]="picker"></mat-datepicker-
toggle>
    <mat-datepicker #picker></mat-datepicker>
  </mat-form-field>


  <mat-form-field appearance="fill">
    <mat-label>Kategorie</mat-label>
    <mat-select [(ngModel)]="category">
      <mat-option *ngFor="let cat of categories" [value]="cat" >
        {{cat}}
      </mat-option>
    </mat-select>
  </mat-form-field>

  <mat-form-field>
    <mat-label>Notizen</mat-label>
    <textarea matInput [(ngModel)]="notes" placeholder="Notizen"></textarea>
  </mat-form-field>

  <mat-form-field>
    <mat-label>Betrag</mat-label>
    <input matInput [(ngModel)]="amount"  type="number" >
    <span matTextPrefix>€ </span>
  </mat-form-field>
</div>
  </mat-grid-tile>
  <mat-grid-tile [colspan]=1 [rowspan]=5>
<div class="group2">
  <h3>Nährwerte</h3>
  <mat-form-field>
    <mat-label>Kcal</mat-label>
    <input matInput [(ngModel)]="kcal"  type="number" >
    <span matTextSuffix>g</span>
  </mat-form-field>
```

```
  <mat-form-field>
    <mat-label>Zucker</mat-label>
    <input matInput [(ngModel)]="sugar"  type="number" >
    <span matTextSuffix>g</span>
  </mat-form-field>

  <mat-form-field>
    <mat-label>Fett</mat-label>
    <input matInput [(ngModel)]="fat"  type="number" >
    <span matTextSuffix>g</span>
  </mat-form-field>

  <mat-form-field>
    <mat-label>Protein</mat-label>
    <input matInput [(ngModel)]="protein"  type="number" >
    <span matTextSuffix>g</span>
  </mat-form-field>

  <mat-form-field>
    <mat-label>Kohlenhydrate</mat-label>
    <input matInput [(ngModel)]="carbs"  type="number" >
    <span matTextSuffix>g</span>
  </mat-form-field>


</div>
</mat-grid-tile>
</mat-grid-list>
</mat-dialog-content>
<mat-dialog-actions align="end">
  <button mat-button (click)="cancel()">Abbrechen</button>
  <button mat-button [mat-dialog-close]="{name: name, quantity: quantity,date:
date, category: category, notes: notes, amount: amount, kcal: kcal, sugar:
sugar,fat: fat,protein: protein, carbs: carbs}"
cdkFocusInitial>Hinzufügen</button>
</mat-dialog-actions>
```

Low Coupling und High Cohesion:

Die Klassen sollten eng zusammenarbeiten. Sie sollten jedoch nicht voneinander abhängig sein. Auf diese Weise wird ein höheres Maß an Wiederverwendbarkeit und Flexibilität des Codes erreicht.

Die Anwendung ist mit Komponenten strukturiert, also kann jede Komponente wiederverwendet werden. Der Service dient als Schnittstelle.

Creator:

Die Erzeugung von Objekten sollte einem Objekt überlassen werden, das über die notwendigen Informationen und Kenntnisse verfügt. Beispielsweise kann eine Factory-Klasse verwendet werden, um Objekte zu erzeugen.

Der Service erzeugt Observable Objekte aus den JSON-Dateien, also übernimmt er die Aufgabe einer Factory-Klasse.

Polymorphismus:

Unter Polymorphismus versteht man, dass verschiedene Objekte eine gemeinsame Schnittstelle haben und somit untereinander austauschbar sind. In Angular wird dies erreicht, indem Interfaces verwendet werden, die von verschiedenen Klassen implementiert werden können.

```
export interface FridgeItem {
    id: number;
    name: string;
    quantity: number;
    expiryDate: Date;
    category: string;
    notes: string;
    amount: number;
    kcal: number;
    sugar: number;
    fat: number;
    protein: number;
    carbs: number;
  }
```

4.3 DRY

Angular realisiert das DRY-Prinzip durch die Verwendung von Komponenten und Services.

Bei Komponenten handelt es sich um wiederverwendbare Einheiten, die über bestimmte Funktionen und Eigenschaften verfügen. Implementiert ein Entwickler eine Funktionalität in einer Komponente, kann diese Komponente an verschiedenen Stellen im Code verwendet werden, ohne dass an jeder Stelle eine Neuimplementierung der Funktionalität erforderlich ist.

Ein weiteres wichtiges Konzept in Angular sind die so genannten Services. Sie sind ein Mittel zur Organisation des Codes und zur Trennung der Logik von der Komponente. Anstatt dass jede Komponente die API direkt aufruft, kann ein Service Daten von einer API abrufen und diese an eine oder mehrere Komponenten weiterleiten.

Das DRY-Prinzip kann einfach umgesetzt werden, indem Angular Komponenten und Services verwendet. Wird eine Funktionalität in einer Komponente oder einem Service implementiert, kann sie von anderen Codeteilen wiederverwendet werden, ohne sie zu duplizieren. Das Ergebnis ist eine Verbesserung der Wartbarkeit und Skalierbarkeit des Codes sowie eine Reduzierung der Entwicklungskosten und -zeit.

Über Imports geregelt:

```
import { FridgeItem } from '../fridge-item.model';
import { FridgeService } from '../fridge.service';
import { MatTableDataSource } from '@angular/material/table';
import {MatDialog} from '@angular/material/dialog';
import { FridgeAddComponent } from '../fridge-add/fridge-add.component';
```

5. Domain Driven Design

Mit Hilfe von Domain Driven Design (DDD) werden das Kerngeschäft und die Komplexität einer Anwendung in den Vordergrund gestellt. Dadurch können komplexe Softwareprobleme gelöst und domänenzentrierte Lösungen entwickelt werden.

- Analyse der Ubiquitous Language

Ubiquitous Language ist ein zentrales Konzept im DDD. Es ist die Bereitstellung einer gemeinsamen Fachsprache, die von allen am Programm (Projekt) Beteiligten verwendet wird und deren Definition einheitlich ist. Im vorliegenden Programmentwurf wurden folgende Begriffe verwendet:

Wort: … (Begründung)

Fridge: Kühlschrank

Fridge-Management: Kühlschrank-Verwaltung

Fridge-Service: Service

Fridge-Item: Artikel

Fridge-Details: Details zu den Artikeln

Fridge-Profile: Profil des Kühlschranks und des Users

Fridge-Shoppinglist: Einkaufsliste

Die Begriffe starten alle mit Fridge, weil der Kühlschrank der zentrale Mittelpunkt der Anwendung ist. So ist die Fachsprache abgrenzbar von anderen Anwendungen und jeder Beteiligte weiß was gemeint ist.

- Analyse der Repositories

Repositories sind Schnittstellen zwischen der Domänenschicht und der Datenzugriffsschicht. Sie ermöglichen einen zentralen Datenzugriff und abstrahieren die Datenhaltung. Sie erhöhen die Flexibilität und verwalten Aggregatwurzeln. Außerdem trennen sie die Domänenlogik vom Datenzugriff.

In unserem Entwurf wurde dies über den Service Fridge-Service geregelt. Er ist die Schnittstelle zwischen den Komponenten und der Datenbasis. Der Zugriff auf die Daten ist dadurch zentralisiert und flexibel.

- Analyse des Aggregats

Das Aggregat ist eine logische Gruppierung zusammengehöriger Objekte im DDD. Dadurch werden Transaktionsgrenzen definiert und die Konsistenz gefördert. Aggregate sind auch wichtig für die Domänenmodellierung und die Implementierung von DDD-Konzepten.

Konsistenz und Geschäftsregeln:

In einer Angular-Anwendung können Konsistenz und Geschäftsregeln durch die Verwendung von Formularen und Validatoren erreicht werden. Zum Beispiel können Formulare verwendet werden, um sicherzustellen, dass alle erforderlichen Felder ausgefüllt sind und die eingegebenen Daten den definierten Regeln entsprechen. Durch die Verwendung von Validatoren können spezifische Geschäftsregeln überprüft werden, um sicherzustellen, dass die Daten im Einklang mit den Anforderungen der Anwendung stehen.

Beispiel Fridge-Add-Unit Komponente:

```
<h1 mat-dialog-title>Kühlschrankfach hinzufügen</h1>
<div mat-dialog-content>
  <mat-form-field appearance="fill">
    <mat-label>Name</mat-label>
    <input matInput [(ngModel)]="unit">
  </mat-form-field>
</div>
```

```
<div mat-dialog-actions>
  <button mat-button (click)="cancel()" >Schließen</button>
  <button mat-button [mat-dialog-close]="unit" >Hinzufügen</button>
</div>
```

Hier wird ein Formular erzeugt, um sicherzustellen, dass ein neues Kühlschrankfach eingetragen wird. Dem Input-Feld wird dabei „unit“ zugewiesen. „unit“ wird nach Klicken auf Hinzufügen verarbeitet. Weitere Validatoren können hinzugefügt werden, sodass leere Eingaben oder die Anzahl der Zeichen überprüft werden können.

Aggregatwurzel:

In einer Angular-Anwendung kann eine Aggregatwurzel ein zentraler Datenpunkt sein, der andere Objekte und Komponenten umfasst.

In unserer Anwendung übernimmt dies, das Interface FridgeItem. Es enthält detailierte Beschreibungen, wie ein Artikel strukturiert wird und wird in vielen Komponenten gebraucht. Es ist ein zentraler Datenpunkt in der Anwendung.

```
import { FridgeItem } from '../fridge-item.model';
fridgeItems: FridgeItem[] = [];

  getItemsData(): void {

    this.fridgeService.getItemsData().subscribe((data: any) => {
      console.log(data);
      this.fridgeItems = data;
      this.dataSource.data = this.fridgeItems;
    });
  }
```

Transaktionelle Grenzen:

Transaktionelle Grenzen in einer Angular-Anwendung können sich auf den Umfang von Datenänderungen beziehen, die atomar und konsistent behandelt werden müssen. Wenn beispielsweise eine Bestellung aufgegeben wird, müssen möglicherweise mehrere Operationen ausgeführt werden, um die Daten in der Datenbank zu aktualisieren und Bestandsänderungen vorzunehmen. Transaktionelle Grenzen helfen dabei, sicherzustellen, dass diese Operationen entweder vollständig abgeschlossen werden oder fehlschlagen, um eine konsistente Datenintegrität zu gewährleisten.

In unserer Anwendung werden Datenänderungen, wie zum Beispiel einen neuen Artikel in den Datenbestand hinzuzufügen, mit mehreren Schritten abgewickelt.

Die Fridge-List Komponente enthält einen Hinzufügen Button, der die Fridge-Add

Komponente aufruft. Dabei wird ein Formular mit Daten zum Artikel ausgefüllt und diese Informationen weitergegeben. Der Fridge-Service überführt den neuen Artikel in die Datenbasis mit der HTTP-Anfrage “POST”.

Fridge-List Komponente:

```
 addItem(item:any) {
    this.newItemId = this.fridgeItems.length + 1;
    console.log(this.newItemId)
    this.newFridgeItem = {id: this.newItemId, name: item.name, quantity:
item.quantity, expiryDate: item.date, category: item.category, notes:
item.notes, amount: item.amount, kcal: item.kcal, sugar: item.sugar,fat:
item.fat,protein: item.protein, carbs: item.carbs };
    this.fridgeService.addItem(this.newFridgeItem).subscribe({
      next: () => {
        console.log('Daten erfolgreich aktualisiert.');
      },
      error: (error) => {
        throw error
      }
    });
    this.fridgeItems.push(this.newFridgeItem);
    this.dataSource.data = this.fridgeItems;
  }
```

Fridge-Service:

```
  addItem(newData: FridgeItem): Observable<any> {
    return this.http.post(this.itemsurl, newData);
  }
```

Server:

```

  app.post('/items', (req, res) => {
    const newItem = req.body; // Extrahieren Sie die Daten aus der Anforderung

    fs.readFile('artikel.json', (err, data) => {
      if (err) {
        res.status(500).send('Fehler beim Laden der Daten.');
        return;
      }

      const items = JSON.parse(data); // Konvertieren Sie die Daten in ein
Objekt
      items.push(newItem); // Fügen Sie den neuen Eintrag hinzu

      fs.writeFile('artikel.json', JSON.stringify(items), (err) => {
        if (err) {
```

```
          res.status(500).send('Fehler beim Speichern der Daten.');
          return;
        }
        res.send('Eintrag wurde erfolgreich hinzugefügt.');
      });
    });
  });
```

Granularität:

Die Granularität in einer Angular-Anwendung bezieht sich auf die Aufteilung von Komponenten und Funktionen in kleinere, wiederverwendbare Einheiten. Eine feine Granularität bedeutet, dass Komponenten und Funktionen klein und spezialisiert sind, während eine gröbere Granularität bedeutet, dass sie größer und allgemeiner sind. Die Auswahl der Granularität hängt von der spezifischen Anwendung und den Anforderungen ab. Eine gut strukturierte Angular-Anwendung besteht in der Regel aus einer Mischung von fein- und grobkörnigen Komponenten, um eine ausgewogene Wiederverwendbarkeit und Lesbarkeit des Codes zu gewährleisten.

In unserer Anwendung sind feinere Komponenten:

Fridge-Add und Fridge-Add-Unit

Sie sind nur für den Fall, dass Artikel oder neue Fächer hinzugefügt werden.

Gröbere Komponenten sind:

Fridge-List, Fridge-Details, Fridge-Shoppinglist. Fridge-Stats und Fridge-Profile

Sie bilden die Hauptpunkte der Anwendung und haben deshalb mehr Funktionalität.

Grenzen und Beziehungen:

In einer Angular-Anwendung können Grenzen und Beziehungen durch die Verwendung von Modulen, Services und Komponenten definiert werden. Module dienen dazu, den Anwendungsumfang zu organisieren und Grenzen zwischen verschiedenen Funktionalitäten zu ziehen. Services ermöglichen die Kommunikation und den Datenaustausch zwischen Komponenten, während Komponenten die visuelle Darstellung und Interaktion mit den Benutzern steuern. Durch die klare Definition von Grenzen und Beziehungen wird die Modularität gewährleistet.

- Analyse der Entities

Durch Entities werden Objekte im DDD real oder konzeptionell repräsentiert. Die Objekte haben somit eine eindeutige Identität und werden durch Attribute definiert. Sie sind in der Regel persistenzunabhängig.

Identität:

In einer Angular-Anwendung kann Identität bedeuten, dass ein Objekt eine eindeutige Kennung besitzt, um es von anderen Objekten zu unterscheiden.

In unserer Anwendung besitzt jeder Artikel eine ID, um sich von anderen Objekten zu unterscheiden. Das ermöglicht den Zugriff und die Weiterverarbeitung des speziellen Artikels.


```
[
    {
        "id":1,
        "name":"Eier",
        "quantity":14,
        "expiryDate":"11/10/2023",
        "category":"Eier",
        "notes":"test",
        "amount":12,
        "kcal":1,
        "sugar":1,
        "fat":1,
        "protein":1,
        "carbs":1
        },
    {
        "id":2,
        "name":"Brot",
        "quantity":5,
        "expiryDate":"2023-04-20T15:49:55.255Z",
        "category":"Teigwaren",
        "notes":"",
        "amount":0,
        "kcal":0,
        "sugar":0,
        "fat":0,
        "protein":0,
        "carbs":0
        }
]
```


Zustandsverwaltung:

In einer Angular-Anwendung kann die Zustandsverwaltung durch die Verwendung von Techniken wie dem RxJS-basierten Zustandsmanagement mit dem Store-Pattern erfolgen. Der Anwendungsstatus wird in einem zentralen Store gehalten, der von Komponenten abonniert werden kann. Dadurch können Komponenten den Zustand lesen, Änderungen abonnieren und Aktionen auslösen, um den Zustand zu aktualisieren. Der Store kann als Observables implementiert sein, wodurch Komponenten den Zustand abonnieren und auf Änderungen reagieren können.

Die Zustandsverwaltung ermöglicht eine zentrale Verwaltung des Anwendungszustands und eine konsistente Aktualisierung der Benutzeroberfläche.

In unserer Anwendung wird das mit Observable geregelt.

Ein Beispiel:

Die Benutzeroberfläche der Komponente Fridge-ShoppingList wird immer aktualisiert, sodass Artikel, welche in geringer Menge oder am Ablaufen sind in die jeweiligen Tabellen eingetragen werden. Dies geschieht beispielsweise über das Abbonieren des Observables getItemsData()

Sobald neue Artikel hinzugefügt, gelöscht oder verändert werden, wird die Änderung mitgeteilt und die Oberfläche aktualisiert.

```
  lowQuantityList() {
    this.fridgeService.getItemsData().subscribe((data: any) => {
      this.fridgeItems = data;
      this.lowQuantityItems = this.fridgeItems.filter(item => item.quantity <
3);
      this.lowQuantityDataSource.data = this.lowQuantityItems;
    });
  }
```

Persistenzunabhängigkeit:

In einer Angular-Anwendung können Entities persistenzunabhängig sein, indem sie nicht direkt an eine bestimmte Datenbank oder Persistenzschicht gebunden sind. Stattdessen können Entities als reine JavaScript/TypeScript-Objekte modelliert werden, die ihre Attribute und Beziehungen definieren. Die tatsächliche Persistenz kann durch Services oder APIs abstrahiert werden, die mit der Datenbank oder der externen Datenquelle interagieren. Dadurch bleibt die Domänenlogik unabhängig von der spezifischen Persistenztechnologie.

Es kann beispielweise ein weiterer Service erstellt werden, der sich nur um die Dialoge kümmert. Dieser ist unabhängig von der Datenquelle und steuert die Funktionen der Dialoge.

Fridge-Service ist mit der Datenquelle (Server.js) über HTTP-Anfragen verbunden. Somit könnte der Dialog-Service mit dem Fridge-Service kommunizieren, um auf die Datenquelle zuzugreifen.

Grenzen und Beziehungen:

In einer Angular-Anwendung können Grenzen und Beziehungen zwischen verschiedenen Komponenten und Modulen definiert werden. Komponenten repräsentieren abgegrenzte Teile der Benutzeroberfläche und können über Inputs und Outputs Daten austauschen. Durch die Definition von Schnittstellen und Beziehungen zwischen Komponenten können Daten und Ereignisse übermittelt werden. Module dienen als Container für verwandte Komponenten und definieren die Grenzen zwischen verschiedenen Funktionalitäten. Dadurch wird eine klare Struktur und Trennung von Verantwortlichkeiten in der Anwendung erreicht.

Jede Komponente kann über ihren Selector importiert werden, um mit anderen Komponenten in Beziehung zu treten. Das müssen sie aber nicht und können abgegrenzt von anderen Komponenten laufen.

Beispiel Fridge-Stats:

Die Komponente Fridge-Stats greift auf den Service und die Models FridgeItem und Fridge zu. Sie ist abgegrenzt von den anderen Komponenten.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { FridgeService } from '../fridge.service';
import { FridgeItem } from '../fridge-item.model';
import { Fridge } from '../fridge.model';
import { Chart }  from 'chart.js/auto';

@Component({
  selector: 'app-fridge-stats',
  templateUrl: './fridge-stats.component.html',
  styleUrls: ['./fridge-stats.component.scss']
})
export class FridgeStatsComponent implements OnInit {
```

Beispiel Fridge-Profile:

```
import { Component, Inject } from '@angular/core';
import { MatDialog, MatDialogRef, MAT_DIALOG_DATA } from
'@angular/material/dialog';
import { MatSnackBar } from '@angular/material/snack-bar';
import { FridgeAddUnitComponent } from '../fridge-add-unit/fridge-add-
unit.component';
import { FridgeService } from '../fridge.service';

@Component({
  selector: 'app-fridge-profile',
  templateUrl: './fridge-profile.component.html',
```

```
  styleUrls: ['./fridge-profile.component.scss']
})
export class FridgeProfileComponent {
```

Die Komponente Fridge-Profile importiert die Komponente FridgeAddUnit und geht mit ihr eine Beziehung ein.

- Value Objektes

Ein weiteres Konzept im DDD ist das Value-Objekt. Es wird verwendet, um die Repräsentation eines Wertes darzustellen. Value Objects besitzen keine eindeutige Identität. Sie werden als Werte miteinander verglichen und sind in der Regel unveränderlich. Im vorliegenden Entwurf können folgende Werte erfasst werden:

Value Objects stellen sicher, dass Datumswerte als unveränderliche Objekte behandelt werden und Vergleiche oder Operationen mit ihnen korrekt durchgeführt werden können, ohne unerwünschte Nebeneffekte zu haben.

Jeder Artikel hat sein eigenes Verfallsdatum.

```
soonToExpireList()  {
    this.fridgeService.getItemsData().subscribe((data: any) => {
    this.fridgeItems = data;
    this.soonToExpireItems = this.fridgeItems.filter(item => {
    this.today = new Date();
    this.expirationDate = new Date(item.expiryDate);
    this.timeDiff = this.expirationDate.getTime() - this.today.getTime();
    this.daysUntilExpiration = Math.ceil(this.timeDiff / (1000 * 3600 * 24));
      return this.daysUntilExpiration <= 3; // Show items that will expire in
the next 3 days
    });
    this.soonToExpireDataSource.data = this.soonToExpireItems;
    });


  }
```

Hierbei wird das aktuelle Datum mit dem Verfallsdatum des Artikels verglichen und falls der Abstand der beiden geringer als drei Tage ist, wird der Artikel in die Liste der verfallenden Artikel aufgenommen.

Ein weiteres Beispiel wäre die Änderung des Datentyps der Usermail von String zur Klasse E-Mail

```
  usermail: string = "";
```

```
export class Email {
  private readonly value: string;

  constructor(email: string) {
    this.validateEmail(email);
    this.value = email;
  }

  private validateEmail(email: string): void {
    const emailRegex = /^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$/;
    if (!emailRegex.test(email)) {
      throw new Error('Invalid email address.');
    }
  }

  isEqual(other: Email): boolean {
    return this.value === other.value;
  }

  toString(): string {
    return this.value;
  }
}
```

Die Verwendung eines Value Objects wie E-Mail ermöglicht es, die E-Mail-Adresse innerhalb der Anwendung konsistent zu repräsentieren und sicherzustellen, dass nur gültige E-Mail-Adressen verwendet werden. Es bietet eine klare Abgrenzung für den Umgang mit E-Mail-Adressen und erleichtert die Wartung und Erweiterbarkeit der Anwendung im Zusammenhang mit E-Mail-Operationen und -Validierungen.

Eine weitere Änderung wäre eine Klasse Money einzuführen, die für den Preis der Artikel eingesetzt wird. Momentan werden Geldbeträge als number abgewickelt

```
  amountValue: number = 0;
```

```
export class Money {
  private readonly value: number;
  private readonly currency: string;

  constructor(value: number, currency: string) {
    this.value = value;
    this.currency = currency;
```

```
  }

  add(other: Money): Money {
    if (this.currency !== other.currency) {
      throw new Error('Cannot add money with different currencies.');
    }

    const sum = this.value + other.value;
    return new Money(sum, this.currency);
  }
}
```

Durch die Verwendung des Value Objects Money können Geldbeträge in der Anwendung auf eine konsistente und kontrollierte Weise repräsentiert und manipuliert werden. Das Value Object stellt sicher, dass Operationen mit Geldbeträgen korrekt durchgeführt werden und die Währungen korrekt behandelt werden.

6. Clean Architecture

Die Abhängigkeiten zwischen den einzelnen Komponenten werden durch eine „clean architecture“ minimiert. Außerdem wird mit Hilfe dieses Architekturmusters die Codequalität verbessert und die Testbarkeit und Wiederverwendbarkeit erhöht. Diese Architektur besteht aus verschiedenen Schichten, die hierarchisch organisiert sind. Jede Schicht hat eine spezifische Verantwortung. Es ist wichtig anzumerken, dass die konkrete Implementierung der Schichten in Angular nicht immer strikt getrennt ist. Aufgrund der Angular-Struktur und des Frameworks können Komponenten manchmal Elemente aus verschiedenen Schichten enthalten. In diesem Entwurfsprogramm wurden folgende Schichten implementiert:

Präsentationsschicht (Presentation Layer): Diese Schicht ist für die Darstellung der Benutzeroberfläche verantwortlich. Sie umfasst Angular-Komponenten, die für die Präsentation von Daten und die Interaktion mit Benutzern zuständig sind. Hier werden Templates, Styles und Logik für die Anzeige und Interaktion implementiert.

Komponenten in dieser Schicht: Fridge-Navigation, Fridge-List, Fridge-Details, Fridge-Shoppinglist, Fridge-Stats, Fridge-Profile. Diese Komponenten erzeugen jeweils eine View mit ihrer .html Datei. Diese Views werden über die .ts Datei in den Komponenten gesteuert.

```
  <mat-nav-list>
      <a href="#" style="height: 0px"></a>
      <a mat-list-item href="fridge/list"><mat-icon
class="navButton">fastfood</mat-icon>Artikelliste</a>
      <a mat-list-item href="fridge/stats"><mat-icon
class="navButton">bar_chart</mat-icon>Statistik</a>
      <a mat-list-item href="fridge/shoppinglist"><mat-icon
class="navButton">shopping_cart</mat-icon>Shoppingliste</a>
      <a mat-list-item href="fridge/profile"><mat-icon
class="navButton">person</mat-icon>Profil</a>
    </mat-nav-list>
```

Beispielsweise lässt die Fridge-Navigation Komponente den User die anderen Komponenten ansteuern.

Domänenschicht (Domain Layer): Die Domänenschicht enthält die Kernlogik der Anwendung und ist unabhängig von jeglicher technischen Implementierung. Sie repräsentiert das eigentliche Domänenmodell und enthält Entitäten, Value Objects, Aggregate, Services und Domänenregeln. Diese Schicht fokussiert sich auf die spezifischen Geschäftsregeln und -prozesse der Anwendung.

Die Interfaces gehören zu dieser Schicht, wie auch Teile der Komponenten und Teile des Fridge-Service.

Infrastrukturschicht (Infrastructure Layer): Diese Schicht ist für technische Aspekte wie Datenzugriff, externe Schnittstellen, Serverkommunikation und andere Infrastrukturdetails zuständig. Sie umfasst beispielsweise Datenbankzugriff, HTTP-Anfragen, Caching, Authentifizierung und andere Infrastrukturkomponenten. Die Infrastrukturschicht ermöglicht die Zusammenarbeit zwischen der Domänenschicht und externen Systemen oder Datenquellen.

In dieser Schicht befinden sich der Fridge-Service und Server.js. Hierbei werden HTTP-Anfragen mit Zugriff auf die Datenbasis abgehandelt.

- Dependency Rule

Das wichtigste Konzept dieser Architektur ist die Abhängigkeitsregel. Sie besagt, dass Abhängigkeiten immer von außen nach innen verlaufen müssen. Dementsprechend dürfen auch die Abhängigkeitspfeile nur von außen nach innen verlaufen. Das bedeutet, dass jede Schicht nur auf ihre unmittelbare Nachbarschicht zugreifen darf und nicht auf andere Schichten.

Gemäß der Dependency Rule sollten Abhängigkeiten von äußeren Schichten (wie der Präsentationsschicht oder der Infrastrukturschicht) auf innere Schichten (wie die Domänenschicht) zeigen. Das bedeutet, dass die Präsentationsschicht oder die Infrastrukturschicht die Domänenschicht kennen darf, aber nicht umgekehrt. Dies kann durch die Anordnung der Module und die Verwendung von Imports in Angular erreicht werden.

Beispiel dafür:

Fridge-Service kennt das Interface FridgeItem aber FridgeItem kennt den Service nicht.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable, of } from 'rxjs';
import { catchError, map, tap } from 'rxjs/operators';


import { FridgeItem } from './fridge-item.model';

@Injectable({
  providedIn: 'root'
})
export class FridgeService {
```

Der Service kennt auch keine Komponenten. Die Komponenten importieren den Service. Hierbei importiert die Fridge-Alert Komponente den Fridge-Service.

```
import { Component } from '@angular/core';
import { MatSnackBar } from '@angular/material/snack-bar';
```

```
import { FridgeService } from '../fridge.service';
import { Observable } from 'rxjs';
import { FridgeItem } from '../fridge-item.model';

@Component({
  selector: 'app-fridge-alert',
  templateUrl: './fridge-alert.component.html',
  styleUrls: ['./fridge-alert.component.scss']
})
export class FridgeAlertComponent {
```

- Dependency Injection

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable, of } from 'rxjs';
import { catchError, map, tap } from 'rxjs/operators';


import { FridgeItem } from './fridge-item.model';

@Injectable({
  providedIn: 'root'
})
export class FridgeService {

constructor(private http: HttpClient) {

  }
```

In diesem Beispiel wird die HttpClient-Instanz als Parameter des Konstruktors der FridgeService-Klasse definiert. Angular übernimmt die Verantwortung, eine Instanz des HttpClient zu erstellen und ihn dem Konstruktor zur Verfügung zu stellen.

Durch die Verwendung von Dependency Injection wird das Dependency-Inversion-Prinzip umgesetzt. Der FridgeService hängt nicht direkt von einer konkreten Implementierung des HttpClient ab, sondern von einer abstrakten Schnittstelle (HttpClient), die von Angular bereitgestellt wird. Dadurch ist der FridgeService unabhängig von der konkreten Implementierung des HttpClient und ermöglicht es, verschiedene Implementierungen auszutauschen oder zu mocken, was die Testbarkeit und Flexibilität erhöht.

Desweiteren kann der Fridge-Service auch über den Konstrukter von Komponenten verwendet werden.
Angular kümmert sich um die Instanziierung und die Bereitstellung der Abhängigkeiten

```
export class FridgeProfileComponent {
  user: any;
  username: string = "";
  usermail: string = "";
  userpassword: string = "";
  units: string[] = [];
  unit: string = "";
  capacity: number = 0;
  measurementSetting: string = "";
  languageSetting: string = "";
  localizationSetting: string = "";
  autoAddSetting: boolean = true;
  alertSetting: boolean = true;
  data: any;

  constructor(private fridgeService: FridgeService, public dialog: MatDialog,
private snackBar: MatSnackBar) {

   }
```

Angular erstellt bei einer neuen Anwendung automatisch die Datei App.Module.ts. Hier müssen alle Abhängigkeiten, die die Anwendung braucht, registriert werden.

```
@NgModule({
  declarations: [
    AppComponent,
    FridgeListComponent,
    FridgeDetailsComponent,
    FridgeSearchComponent,
    FridgeAlertComponent,
    FridgeStatsComponent,
    FridgeAddComponent,
    FridgeComponent,
    FridgeShoppinglistComponent,
    FridgeNavigationComponent,
    FridgeProfileComponent,
    FridgeFooterComponent,
    FridgeAddUnitComponent,
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    FormsModule,
    NgbModule,
    AppRoutingModule,
    MatToolbarModule,
    MatButtonModule,
```

```
    MatIconModule,
    MatInputModule,
    MatCardModule,
    MatDividerModule,
    MatListModule,
    MatTableModule,
    MatGridListModule,
    MatTabsModule,
    BrowserAnimationsModule,
    MatStepperModule,
    ReactiveFormsModule,
    MatDialogModule,
    MatSnackBarModule,
    MatSidenavModule,
    MatMenuModule,
    LayoutModule,
    MatDatepickerModule,
    MatNativeDateModule,
    MatSelectModule,
    MatExpansionModule,
    MatChipsModule,
    MatCheckboxModule,
    MatRadioModule,
    MatSlideToggleModule,
    ServiceWorkerModule.register('ngsw-worker.js', {
      enabled: !isDevMode(),

      registrationStrategy: 'registerWhenStable:30000'
    })



  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
```

- Main-Methode

In der Clean-Architektur ist die Main-Methode nicht direkt in die Details der inneren Schichten der Architektur integriert. Dies bedeutet, dass die Main-Methode in einer separaten Schicht oder einem separaten Modul untergebracht wird. Dies wird als „Bootstrapping“ bezeichnet. In dieser Schicht werden die erforderlichen Abhängigkeiten konfiguriert, wodurch die entsprechenden Komponenten instanziiert und die eigentliche Anwendung gestartet wird.

In einer Angular-Anwendung startet die Anwendung in der Regel mit der Datei main.ts. Diese Datei wird als Einstiegspunkt für die Anwendung verwendet und enthält den Code, der die Angular-Plattform startet.

In der main.ts-Datei werden verschiedene Aufgaben erledigt:

Importieren des Angular-Moduls: Zuerst wird das Hauptmodul der Angular-Anwendung importiert. Standardmäßig wird das Hauptmodul AppModule genannt.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
```

Starten der Anwendung: Anschließend wird die Angular-Plattform initialisiert und die Anwendung gestartet. Dazu wird die Funktion platformBrowserDynamic().bootstrapModule() verwendet, die das Hauptmodul (AppModule) als Parameter annimmt.

```
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Diese Funktion initialisiert den Angular-Compiler, kompiliert die Angular-Komponenten und startet die Anwendung in einem Browser- oder mobilen Umfeld.

Die main.ts-Datei ist normalerweise im Stammverzeichnis des Angular-Projekts zu finden. Sie ist unter src/main.ts auffindbar

Sie wird automatisch von Angular generiert und konfiguriert, wenn ein neues Angular-Projekt mit dem Angular CLI erstellt wird.

Die main.ts-Datei ist der Startpunkt für die Ausführung der Angular-Anwendung und lädt das Hauptmodul, von dem aus die gesamte Anwendung initialisiert wird.

Hierbei startet AppModule (app.module.ts) die Komponente AppComponent

```
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

AppComponent startet fridge-navigation. Von dort können die anderen Komponenten über das Navigationsmenü erreicht werden.

```
<app-fridge-navigation></app-fridge-navigation>
```