

# Softwareentwurf Advanced SWE

## Softwareentwurf

im Rahmen der Prüfung zum  
**Bachelor of Science (B.Sc.)**

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Lucas Merkel**

April 2022

Abgabedatum:	30. April 2022
Bearbeitungszeitraum:	05.10.2021 - 30.04.2022
Matrikelnummer, Kurs:	4161053, TINF19B1
Gutachter der Dualen Hochschule:	Daniel Lindner

# Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Softwareentwurf mit dem Thema:

*Softwareentwurf Advanced SWE*

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 25. April 2022

---

Merkel, Lucas

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>IV</b>
<b>Abbildungsverzeichnis</b>	<b>V</b>
<b>Tabellenverzeichnis</b>	<b>VI</b>
<b>Quellcodeverzeichnis</b>	<b>VII</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Anforderungen an den Softwareentwurf . . . . .	1
1.2 Anforderungen an die zu entwickelnde Software . . . . .	3
<b>2 Bedienungsanleitung der Anwendung</b>	<b>5</b>
2.1 Aufrufen der Anwendung . . . . .	5
2.2 Konsumgut anlegen . . . . .	6
2.3 Daten eines Konsumsgut ändern . . . . .	8
2.4 Konsumgut löschen . . . . .	8
<b>3 Clean Architecture</b>	<b>10</b>
3.1 Planung der Schichten . . . . .	10
3.2 Anwendung der Schichten . . . . .	12
<b>4 Programming Principles</b>	<b>21</b>
4.1 Zu betrachtende Programming Principles . . . . .	21
4.2 Analyse und Begründung . . . . .	25
<b>5 Domain Driven Design</b>	<b>33</b>
5.1 Analyse und Begründung der <i>Ubiquitous Language</i> . . . . .	34
5.2 Analyse angewandter Value Objects . . . . .	39
5.3 Analyse angewandter Entities . . . . .	40
5.4 Analyse angewandter Aggregates . . . . .	41
5.5 Analyse angewandter Repositories . . . . .	42
<b>6 Unit Tests</b>	<b>44</b>
6.1 ATRIP . . . . .	45
6.2 Testabdeckung . . . . .	45
6.3 Testen mit Mocks . . . . .	46
6.4 Anwendung im Softwareentwurf . . . . .	46

<b>7 Entwurfsmuster</b>	<b>52</b>
7.1 Angewandte Entwurfsmuster . . . . .	52
<b>8 Refactoring</b>	<b>55</b>

# Abkürzungsverzeichnis

<b>API</b>	Application Programming Interface
<b>CRUD</b>	Create, Read, Update, Delete
<b>DDD</b>	Domain Driven Design
<b>DRY</b>	Don't Repeat Yourself
<b>EAN</b>	European Article Number
<b>GRASP</b>	General Responsibility Assignment Software Patterns/Principles
<b>GUI</b>	Graphical User Interface
<b>HATEOAS</b>	Hypermedia as the Engine of Application State
<b>HTTP</b>	Hypertext Transport Transfer Protocol
<b>HTTPS</b>	Hypertext Transport Transfer Protocol Secure
<b>JPA</b>	Java Persistence API
<b>KISS</b>	Keep it Simple, Stupid
<b>REST</b>	Representational State Transfer
<b>URL</b>	Uniform Resource Link
<b>URI</b>	Uniform Resource Identifier
<b>YAGNI</b>	You Ain't Gonna Need it

# Abbildungsverzeichnis

2.1	Grafische Oberfläche des Oberflächenmoduls. . . . .	5
2.2	Suchfunktion der Oberfläche. . . . .	6
2.3	Rückmeldung bei keinem Ergebnis auf Suchanfrage. . . . .	6
2.4	Einlagern eines Konsumguts. . . . .	7
2.5	Darstellung angelegtes Konsumgut. . . . .	7
2.6	Daten eines Konsumguts ändern. . . . .	8
2.7	Auslagern eines Konsumguts. . . . .	9
3.1	Aufteilung der Projektstruktur entsprechend <i>Clean Architecture</i> . . . . .	12
3.2	Ausschnitt aus der pom.xml zur Projektstruktur. . . . .	13
5.1	Wordcloud in Bezug zur Verdeutlichung der Umsetzung Ubiquitous Language. . . . .	35
5.2	Überschrift Eingabemaske zum Einlagern eines Konsumguts. . . . .	37
5.3	Bezeichnung eines Auslagerungsvorgangs. . . . .	37
5.4	Darstellung EAN-Code. . . . .	37
5.5	Button zum Öffnen der Eingabemaske zum Einlagern eines Konsumguts. . . . .	38
5.6	Buttons zum Bestätigen des Einlagerns eines Konsumguts. . . . .	38
5.7	Buttons zum Bestätigen von Änderungen der Attribute eines Konsumguts. . . . .	38
5.8	Buttons zum Bestätigen des Auslagern eines Konsumguts. . . . .	38
7.1	Ausschnitt des UML-Diagramms zur Darstellung des <i>Bridge</i> -Entwurfsmusters. . . . .	52
7.2	Ausschnitt des UML-Diagramms zur Darstellung des <i>Builder</i> -Entwurfsmusters. . . . .	53

# Tabellenverzeichnis

3.1	Gegenüberstellung der in der <i>Adapters</i> -Schicht abgebildeten Klassen aus der <i>Domain</i> -Schicht. . . . .	16
6.1	Code Coverage des gesamten Projekts. . . . .	51

# Quellcodeverzeichnis



# 1 Einleitung

In diesem Kapitel werden die Anforderungen an den abzugebenden Softwareentwurf sowie die Anforderungen an die zu entwickelnde Software beschrieben. Dazu zählt ebenfalls die Protokollierung spezieller Vereinbarungen mit dem Dozenten, der zugleich Gutachter des Softwareentwurfs ist.

## 1.1 Anforderungen an den Softwareentwurf

Dieser Softwareentwurf dient der Lernzielkontrolle der innerhalb der im fünften und sechsten Semester stattfindenden Vorlesung *Advanced Softwareengineering* vermittelnden Theorien. Dozent dieser Vorlesung ist Daniel Lindner. Er ist zugleich der Gutachter des Softwareentwurfs. Die Bearbeitung des Entwurfs startet mit der ersten Vorlesung am 04.10.2021. Der Abgabetermin ist der 30.04.2022. Eine Verlängerung der Abgabefrist ist nur unter bestimmten Voraussetzungen und Gründen möglich und bedarf der Absprache mit dem Dozenten. Hierbei kann die Abgabe maximal um die zum Zeitpunkt der Anfrage restliche Bearbeitungszeit verdoppelt werden. Zu den generellen Anforderungen zählen der zu entwickelnde Code einer Software sowie die schriftliche Dokumentation. Der Code soll dabei auf einem GitHub-Repository verwaltet werden. Der Code des Softwareprojekts muss dabei in einer gebräuchlichen Programmiersprache geschrieben werden. Die Empfehlung liegt dabei auf Java. Die Größe des Softwareprojekts soll über 2000 Zeilen Code liegen und circa mehr als 20 Klassen haben. Die Software soll einen klar definierten und sinnvollen Nutzen haben und eine Desktop- oder Webanwendung sein. In Bezug auf die vermittelten Lehrinhalte der Vorlesung sollen die sechs Schwerpunkte *Clean Architecture*, *Programming Principles*, *Domain Driven Design*, *Unit Tests*, *Entwurfsmuster* sowie *Refactoring* betrachtet und angewandt werden. Die einzelnen Schwerpunkte in Bezug auf die vermittelten Lehrinhalte werden im Folgenden erläutert.

## Clean Architecture

Unter dem Schwerpunkt *Clean Architecture* ist relevant, die Schichtarchitektur in Bezug auf den eigenen Softwareentwurf zu planen und zu begründen. Zudem müssen mindestens zwei Schichten umgesetzt und dokumentiert werden.

## Programming Principles

Bei der Anwendung der *Programming Principles* gilt es, die Anwendung der Prinzipien *SOLID*, *GRASP* und *DRY* auf die eigene Software zu analysieren und zu begründen.

## Domain Driven Design

In Bezug auf *Domain Driven Design* ist es relevant, die Ubiquitous Language zu analysieren. Des Weiteren sollten die Aspekte *Repositories*, *Aggregates*, *Entities* sowie *Value Objects* innerhalb des Softwareentwurfs analysiert und begründet werden.

## Unit Tests

Im Softwareentwurf sollen mindestens 10 Unit Tests angewandt werden. Mindestens einer dieser Unit Tests soll durch den Einsatz von Mocks erweitert werden. Diese sollen dabei entsprechend der *ATRIP*-Regeln begründet werden. Des Weiteren soll die *Code Coverage* betrachtet und deren Aussagekraft beleuchtet werden.

## Entwurfsmuster

Mindestens eines der in der Vorlesung gezeigten Entwurfsmuster soll angewandt und entsprechend begründet werden. Dazu zählt die Erstellung eines UML-Diagramms des Musters mit klarer Benennung.

## Refactoring

Unter dem Themenbereich *Refactoring* sollen mindestens drei Code Smells identifiziert werden. Darauf folgend soll mindestens zwei mal Refactoring unter Begründung angewandt werden.

## 1.2 Anforderungen an die zu entwickelnde Software

Bei der entwickelten Software handelt es sich um einen Lebensmittel-Inventurplaner. Die Bezeichnung des Softwareprojekts lautet `consumergoods-inventory-planner`. Die Anwendung entspricht einer digitalen Auflistung aller im System gepflegten Lebensmittel. Der Nutzer muss diese Angaben selbstständig anpassen. Es soll die Grundlage gegeben sein, die Daten werden auf einer Datenbank zu speichern. Die Realisierung der Speicherung muss zum Zeitpunkt der Abgabe nicht realisiert sein. Die Erreichbarkeit der Anwendung soll als Webapplikation realisiert werden. Die Daten werden durch einen Representational State Transfer (REST)-Service bereitgestellt und dient der Interaktion mit der Webapplikation. Der Nutzen für den Kunden bei dieser Anwendung ist, dass der Lebensmittel-Inventurplaner einen Überblick über alle derzeit verfügbaren beziehungsweise im System gepflegten Lebensmittel gibt, die der Kunde aktuell Zuhause hat. Diese digitale Auflistung kann beim Einkaufen oder der Kochplanung unterstützen. Des Weiteren kann als Attribut der einzelnen Lebensmittel-Objekte ein Mindesthaltbarkeitsdatum angegeben werden. Hierdurch erhält der Kunde einen Überblick über demnächst ablaufende Lebensmittel und kann seine Kochplanung dementsprechend anpassen und letztendlich die Lebensmittelverschwendung reduzieren. Zur Softwarerealisierung verwendete Technologien sind Java sowie SpringBoot für die Realisierung der REST-Schnittstelle. Hibernate für die Interaktionen zwischen Java Anwendung und der Datenbank und MariaDB als Datenbanktechnologie.

Angular in der Version *9.1.12* wird für die Realisierung der Bedienungsoberfläche angewandt.

Als Java-Version wird *Java SE 8* verwendet.

Maven wird in der Version *3.1.0* verwendet.

Die Software wurde auf einem Rechner mit *Windows 10* als Betriebssystem getestet.

Der Webservice ist über den *localhost* erreichbar.

Also Port wurde der Port *8083* angewandt.

Hypertext Transport Transfer Protocol Secure (HTTPS) sollte grundsätzlich für Webanwendungen als Standard angesehen werden, jedoch wird in diesem Umfang zu Demonstrationszwecken zunächst darauf verzichtet.

Somit ist die Uniform Resource Link (URL) des Webservices `http://localhost:8083`.

Die entwickelnde Software wird zudem auf *GitHub* versioniert.

## Abweichung des REST-Paradigmas

Die im Softwareentwurf umgesetzte REST-Schnittstelle hat im Gegensatz zur dem von Roy Fielding definierte R-Paradigma Abweichungen, die erläutert werden. Zum einen weicht die Umsetzung der Repräsentation zur Veränderung von Ressourcen im Softwareentwurf von der Vorgehensweise im Paradigma ab. Die Ressourcen werden zum Einlagern oder Verändern über die Uniform Resource Identifier (URI) in Form von Attributen übertragen. Das Paradigma sieht hierfür jedoch die Übermittlung über den Hypertext Transport Transfer Protocol (HTTP)-Body in einer einheitlichen Sprache vor. Beispiel der Umsetzung Der Vorteil ist, dass das Einlagern eines Konsumguts direkt über die URI erfolgen kann, für die Anbindung weiterer Module an die bestehende Software. Der Nachteil ist, dass die Daten in der URI, selbst bei Nutzung von HTTPS nicht verschlüsselt sind und die URI ab einer bestimmten Länge zu Problemen bei der Verarbeitung durch den Server führen kann. Im Standard RFC 2616, RFC7230 (HTTP/1.1) sowie RFC 7540 (HTTP/2) werden jedoch keine maximalen Zeichenwerte genannt. Des Weiteren ist in REST empfohlen, Ressourcen für den Client durch Verwendung von Hypermedia auffindbar und erkennbar zu machen. Man spricht hierbei auch von Hypermedia as the Engine of Application State (HATEOAS). Darauf wurde in diesem Softwareentwurf ebenfalls verzichtet. Die durch die Anpassungen resultierende Kommunikation wird auch als Home-Made-Messaging bezeichnet.

## 2 Bedienungsanleitung der Anwendung

Der Programmmentwurf besitzt eine mithilfe der Angular-Technologie entwickelten Benutzeroberfläche, deren Anwendung im Folgenden erläutert wird.

### 2.1 Aufrufen der Anwendung

Die Angular-Anwendung lässt sich mittels des Befehls `npm start` im entsprechenden Oberverzeichnis starten. Nach der Initialisierung ist die Anwendung entsprechend Abbildung 2.1 über die Adresse `http://localhost:4200` erreichbar.

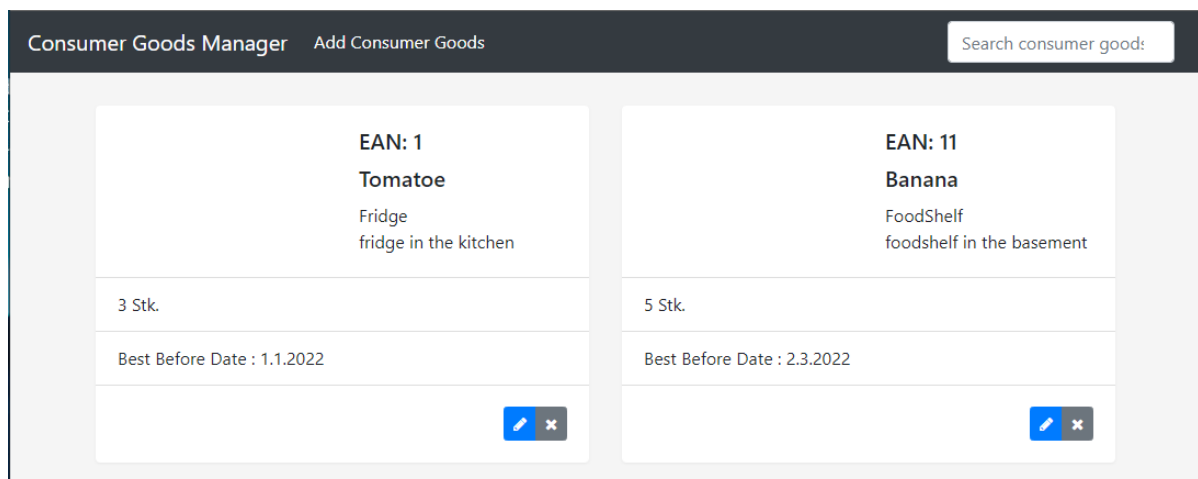
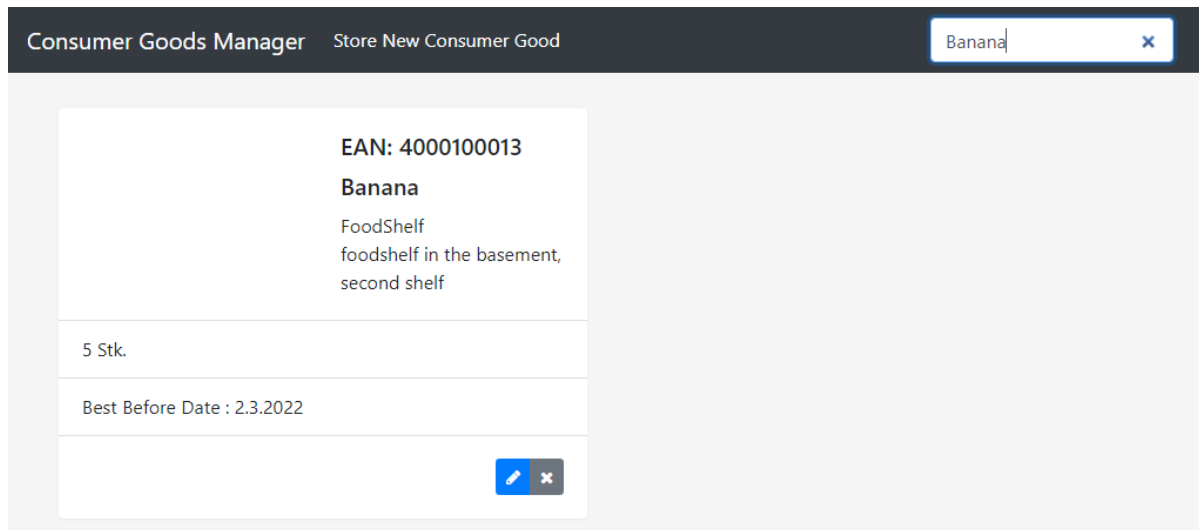


Abbildung 2.1: Die Oberfläche des Oberflächenmoduls zeigt direkt die eingelagerten Konsumgüter samt den jeweiligen Attributen an.

Auf der initialen Darstellung werden bereits verwaltete Konsumgüter dargestellt. Wie in Abbildung 2.2 zu sehen, können über das Eingabefeld an der rechten oberen Seite Konsumgüter entsprechend ihrer Attribute gesucht werden.



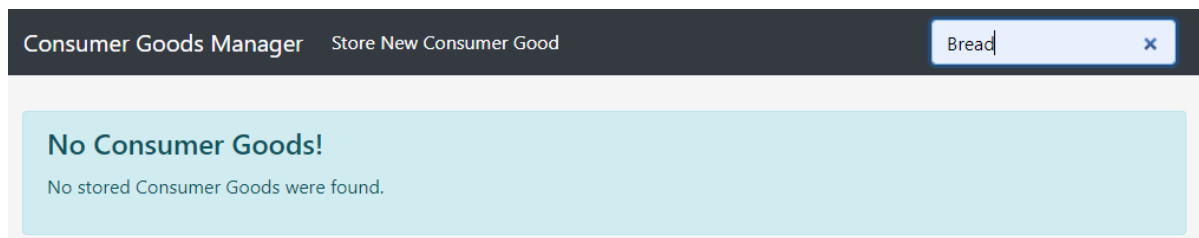
The screenshot shows the 'Consumer Goods Manager' application with the 'Store New Consumer Good' tab active. A search bar at the top right contains the text 'Banana'. Below the search bar, a white card displays the search results for 'Banana'. The card contains the following information:

- EAN:** 4000100013
- Banana**
- FoodShelf
- foodshelf in the basement, second shelf

Below the card, there are two input fields: '5 Stk.' and 'Best Before Date : 2.3.2022'. At the bottom right of the card, there are two buttons: a blue button with a pencil icon and a grey button with an 'X' icon.

Abbildung 2.2: Über das Eingabefeld auf der rechten oberen Seite kann der Suchbefehl eingegeben werden.

Sollte eine Suchanfrage kein Ergebnis liefern, dann erhält der Nutzer eine Fehlermeldung, wie in Abbildung 2.3 dargestellt. Der gesuchte Befehl kann über das X in der Eingabeleiste gelöscht werden.



The screenshot shows the 'Consumer Goods Manager' application with the 'Store New Consumer Good' tab active. A search bar at the top right contains the text 'Bread'. Below the search bar, a light blue message box displays the following text:

**No Consumer Goods!**  
No stored Consumer Goods were found.

Abbildung 2.3: Diese Meldung wird ausgegeben, wenn für den Suchbefehl kein Ergebnis vorliegt.

## 2.2 Konsumgut anlegen

Das Anlegen eines Konsumguts ist über den Button *Add Consumer Goods* möglich. Darauf öffnet sich ein Fenster wie in Abbildung 2.4 zur Eingabe der Daten des neuanzulegenden Konsumguts. Nach Eingabe kann das Konsumgut über den Button *Add Consumer Goods* angelegt werden.

**Store New Consumer Good** ×

EAN  
4 000001 012978

Description  
Tomatoes

Best Before Date - Day  
22

Best Before Date - Month  
02

Best Before Date - Year  
2022

Unit of Measure  
g/Stk./ml

Measured Value  
0.0

Storage Location  
fridge in the basement

Type of Storage  
Fridge

Not store consumer good Store new consumer good

Abbildung 2.4: Eingabemaske zum Eingeben der Attribute eines Konsumguts zum Einlagern.

Nachdem das Konsumgut angelegt ist, wird es ebenfalls auf der Oberfläche entsprechend Abbildung 2.5 dargestellt.

EAN: 4000100014

**Bread**

FoodShelf

First shelf in the basement

100 g

Best Before Date : 30.4.2022

✎ ✕

Abbildung 2.5: Darstellung des Konsumguts nachdem es eingelagert wurde.

Über den blauen Button können die Attribute des Konsumguts geändert werden. Das Vorgehen wird in Abschnitt 2.3 beschrieben. Über den grauen Button kann das Konsumgut gelöscht werden. Das Vorgehen wird in Abschnitt 2.4 beschrieben.

## 2.3 Daten eines Konsumguts ändern

Zum Ändern von Attributen eines Konsumguts muss der blaue Button des gewünschten Konsumguts ausgewählt werden. Daraufhin öffnet sich eine Eingabemaske wie in Abbildung 2.6 gezeigt, in der die aktuellen Werte der Attribute enthalten sind. Die Werte können entsprechend angepasst und durch Auswahl des Buttons *Save changes* bestätigt werden.

Edit Consumer Good Bread with EAN: 4000100014

Description  
Bread

Best Before Date - Day  
30

Best Before Date - Month  
4

Best Before Date - Year  
2022

Unit of Measure  
g

Measured Value  
200

Storage Location  
First shelf in the basement

Type of Storage  
FoodShelf

Make no change Save changes

Abbildung 2.6: Eingabemaske für das Ändern von Attributen eines Konsumguts.

## 2.4 Konsumgut löschen

Das Löschen des Konsumguts erfolgt über den grauen Button. Bei Auswahl öffnet sich ein Dialogfenster entsprechend Abbildung 2.7, das den Nutzer noch einmal auffordert, den Löschvorgang zu bestätigen, um ein unbeabsichtigtes Löschen zu vermeiden.



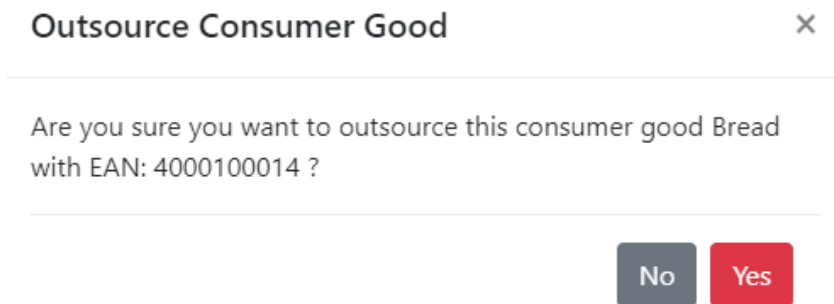


Abbildung 2.7: Bestätigungsmeldung für das Auslagern eines Produkts.

## 3 Clean Architecture

Bei der Umsetzung des Softwareentwurfs wurde sich an der Theorie der Clean Architecture orientiert. Im folgenden Kapitel wird die Planung der einzelnen Schichten beschrieben sowie die Anwendung der Schichten auf den Programmentwurf.

### 3.1 Planung der Schichten

Das Ziel der Clean Architecture ist das Erstellen einer langfristigen Software-Architektur. Schließlich werden während eines Lebenszykluses einer Software unterschiedliche Technologien auf dem Markt angeboten. Diese Technologien können unterschiedliche Anwendungsfälle haben. Dazu zählt zum Beispiel die Datenbanktechnologie sowie die dazugehörige Anbindung an das Softwaresystem oder auch die Technologie der grafischen Oberfläche. Somit bedarf es der Möglichkeit, mit möglichst geringem Aufwand eingesetzte Technologien auszutauschen. Ist die Softwarearchitektur nun an die verwendeten Technologien gebunden, dann ist ein Austauschen der Technologie nur schwer möglich und bedarf großem Aufwand an mitunter vielen Stellen innerhalb der Software.

Zur Minimierung dieses Aufwands durch einen Technologietausch, setzt die Clean Architecture grundsätzlich auf einen bestimmten Aufbau. Zunächst besitzt die Anwendung einen technologieabhängigen Kern, der die eigentlichen Geschäftsregeln enthält. Darauf folgt jede Abhängigkeit als temporäre Lösung, das heißt, dass eine Abhängigkeit von den Geschäftsregeln als temporäre Lösung realisiert ist. Ein Entfernen der Abhängigkeit führt somit nicht dazu, dass das Programm nicht mehr kompiliert werden kann. Man teilt daher auch in zentralen und somit langlebigen Code sowie peripherem, oder auch kurzfristigem, Sourcecode ein. Den innersten Kern bietet der *Abstraction Code*. Er enthält domänenübergreifendes Wissen. Dazu zählen beispielsweise mathematische Konzepte oder physikalische Grundlagen. Eine Änderung dieses Wissens ist nahezu ausgeschlossen. Darauf folgt der *Domain Code*. Er enthält die Entitäten der Anwendungen und somit die eigentliche organisationsweite Geschäftslogik. Der *Domain Code* sollte sich am seltensten ändern. Als nächste Schicht folgt der *Application Code*. Darin ist die Applikationslogik

beziehungsweise die anwendungsspezifische Geschäftslogik enthalten, welche mit den Entitäten aus dem *Domain Code* angewandt wird. Eine Änderung dieses Codes ist möglich, wenn sich die Anforderungen der Software ändern. Auf den *Application Code* folgt die *Adapters*-Schicht. Diese Schicht dient als Interface für Adapter und enthält daher nötige Controller, Presenters oder Gateways. Die Schicht fungiert als Zwischenschicht und vermittelt Aufrufe sowie Daten an die innere Schichten. Das Ziel ist dabei die Entkopplung der inneren und äußeren Schichten, weshalb man auch von einem *Anti Corruption Layer* spricht. Als letzte Schicht folgen die *Plugins*. Darin enthalten sind eingesetzte Frameworks oder Treiber. Diese Schicht ist vor allem für die Anbindung an eine Datenbank, die grafische Oberfläche oder auch Drittsysteme nötig. *Plugins*-Code greift eher nur auf die *Adapters*-Schicht zu und ist leicht auszutauschen.

Der Aufbau der Architektur entspricht somit einer Zwiebel und wird daher auch *Onion Architecture* bezeichnet.

Wichtig ist bei diesem Aufbau, dass jeweilige Abhängigkeiten nur in tieferliegende Schichten bestehen dürfen. Innere Schichten dürfen daher die äußeren Schichten nicht kennen. Man bezeichnet diese Regel auch als *Dependency Rule*. Eine Anwendung der *Dependency Injection* sowie *Dependency Inversion* kann hierbei unterstützen. Das Ziel ist dadurch, dass ein Austausch einer Technologie somit nur die äußerste Schicht betrifft, der Kern der Software hiervon jedoch unberührt ist und somit die Geschäftslogik nicht davon beeinflusst. Somit wird es möglich, dass Code nur von langlebigerem Code als sich selbst abhängig ist.

Die Grenzen der Clean Architecture ist allerdings, dass technische Grundlagen dennoch gegeben und stabil sein müssen. Dazu zählt die verwendete Programmiersprache, der Compiler und die Laufzeitumgebung sowie auch in gewissem Maße das Betriebssystem oder die eingesetzte Hardware. Frameworks können diese Grenzen ebenfalls einschränken, wenn eine zu starke Bindung stattfindet. Dieser Eingrenzung muss man sich bei der Planung der Software-Architektur im Klaren sein und daraus mögliche Folgen erkennen beziehungsweise abwägen.

## 3.2 Anwendung der Schichten

Im Folgenden wird die Umsetzung der *Clean Architecture* innerhalb dieses Softwareentwurfs beschrieben.

Zunächst ist die Projektstruktur entsprechend realisiert worden. Anstatt alle nötigen Klassen in einem Projekt zu erzeugen, ist zur Übersichtlichkeit eine Strukturierung entsprechend der Schichten vorgenommen worden. Die Strukturierung der einzelnen Schichten ist nicht über Packages gemacht worden. Diese Variante ist grundsätzlich möglich, jedoch findet dabei keine Überprüfung durch den Compiler statt und es ist somit schwieriger möglich, einzelne Abhängigkeiten von innen nach außen zu erkennen. Stattdessen bietet sich die Form mehrerer Projekte an. Jedes Projekt bildet dabei eine Schicht ab. Der Compiler erkennt nun nur die im eigenen Projekt sowie in referenzierten Projekten vorhandene Klassen. Diese Möglichkeit unterstützt bei der Umsetzung der tiefergehenden Abhängigkeiten. Darüber hinaus dient ein Überprojekt als Abbild des gesamten Projekts und der Klammerung der einzelnen Schicht. Abbildung 3.1 zeigt die umgesetzte Projektstruktur und Abbildung 3.2 die pom.xml zur Umsetzung der einzelnen Projekte in einem Klammerprojekt. Die einzelnen Projekte, die jeweils eine Schicht der *Clean Architecture* abbilden, sind in dieses globale Projekt als Module eingebunden.

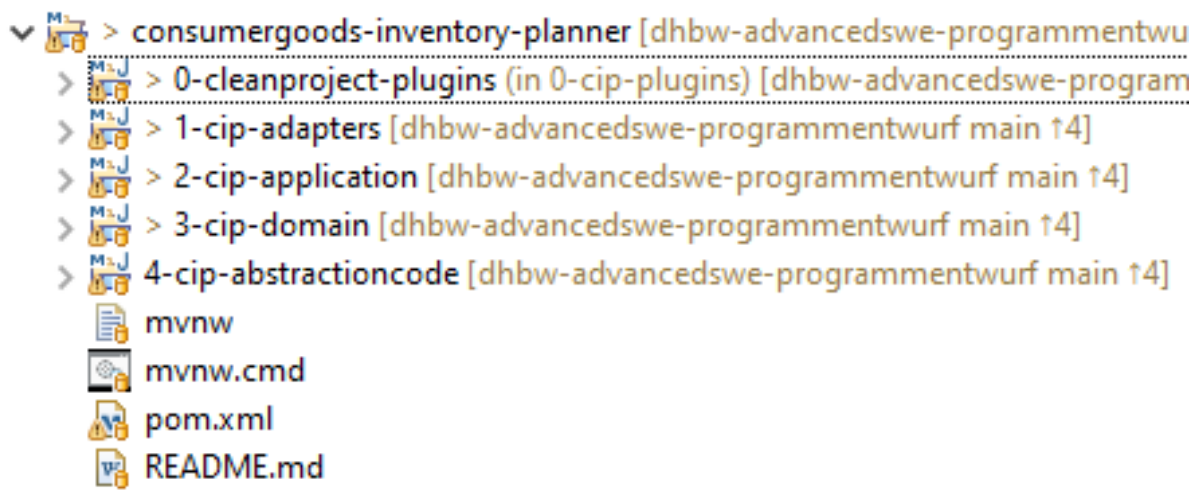


Abbildung 3.1: Die Projektstruktur ist entsprechend der *Clean Architecture* in einzelne Unterprojekte aufgeteilt.

```
<modules>
  <module>4-cip-abstractioncode</module>
  <module>3-cip-domain</module>
  <module>2-cip-application</module>
  <module>1-cip-adapters</module>
  <module>0-cip-plugins</module>
</modules>
```

Abbildung 3.2: Der Ausschnitt zeigt die Umsetzung der Projektstruktur zwischen dem Klammerprojekt und den einzelnen Unterprojekten.

Die Realisierung der einzelnen Schichten bezieht sich auf den Anwendungsbezug und die Anforderungen der Anwendung. Das zum Programmentwurf dazugehörige Klassendiagramm verdeutlicht farblich die Zugehörigkeit der Klassen entsprechend der *Clean-Architecture*. Die farbliche Zuordnung ist wie folgt:

- weiß entspricht der *Abstraction Code*-Schicht,
- dunkelblau der *Domain Code*-Schicht,
- hellblau der *Application Code*-Schicht,
- dunkelgrün der *Adapter Code*-Schicht und
- hellgrün der *Plugin Code*-Schicht.

Die Software dient als REST-Server zur Verwaltung der Lebensmittel Zuhause. Dabei können Lebensmittel sowohl im Kühlschrank, der Gefriertruhe oder in einem Regal gelagert werden. Neue Lebensmittel können angelegt werden. Dabei ist die Bezeichnung des Lebensmittels sowie die entsprechende Menge und das Mindesthaltbarkeitsdatum relevant. Bei fehlerhafter Eingabe oder dem teilweisen Verbrauch können entsprechende Werte angepasst werden. Verbrauchte Lebensmittel können gelöscht werden. Dementsprechend sieht eine Umsetzung der einzelnen Schichten wie folgt aus:

## **Abstraction Code-Schicht**

In der *Abstraction Code-Schicht* befinden sich Klassen für die Verwaltung der Maßeinheit der Lebensmittel. Die Klasse `UnitOfMeasure` ist die Superklasse der Subklassen `Quantity`, `Volume` sowie `Weight`. Diese Klassen repräsentieren die zugehörige Maßeinheit. Der entsprechende Wert wird durch eine Referenz auf ein Objekt der Klasse `Value` gespeichert. Gleiches gilt für die Repräsentation des Datums eines Tages, Monats und Jahres. Dafür dienen die Klassen `Day`, `Month` sowie `Year`. Die Klassen für die Maßeinheit und die Repräsentation eines Tages, eines Monats sowie eines Jahres sind in dieser Schicht festgelegt, da die Informationsrepräsentation der Klassen auf domänenübergreifendes Wissen zurückzuführen ist, dessen Änderung und somit auch das der Klassen ausgeschlossen. Die Klassen benötigen zudem keine Abhängigkeit zu Klassen der äußeren Schichten.

## **Domain Code-Schicht**

Die *Domain Code-Schicht* enthält die für den Anwendungsfall der Software relevanten Domänen. Dazu zählt zum einen die Klasse `ConsumerGoods`. Diese Klasse dient der Repräsentation der Lebensmittel als einer der Domänen. Innerhalb der Klasse `ConsumerGoods` werden dabei die weiteren Klassen `Food` als eigentliches Lebensmittel, `UnitOfMeasure` als Klasse für die Maßeinheit sowie `Storage` als Klasse für den Lagerort verwaltet. Die Klasse `UnitOfMeasure` ist wie bereits erläutert Teil der *Abstraction Code-Schicht* und somit ist die Abhängigkeit möglich. Die Klasse `Storage` ist die Superklasse für die Lagerortsklassen `Fridge` sowie `FoodShelf` und speichern die Beschreibung des Lagerorts. Eine weitere Klasse ist `Food`. Diese Klasse repräsentiert die Bezeichnung des zu speichernden Lebensmittels sowie das dazugehörige Mindesthaltbarkeitsdatum. Das Mindesthaltbarkeitsdatum wird in der Klasse `BestBeforeDate` verwaltet.

Darüber hinaus befinden sich in der *Domain Code-Schicht* die Interfaces `ConsumerGoodsRepository`, `FridgeRepository` und `FoodShelfRepository`. Diese Interfaces dienen im folgenden Verlauf der Anwendung einer *Dependency Inversion*. Die Interfaces beinhalten die abstrakten Methoden zum Finden gezielter oder aller jeweiligen Entitäten sowie dem Löschen und Hinzufügen.

Die Klassen `DayValidator`, `MonthValidator`, `YearValidator` und `ValueValidator`. Diese Klassen dienen der Validierung der Werte für das Tagdatum, Monatsdatum, Jahresdatum und den Wert für die Menge. Die Validierung liegt hierbei nur auf den grundlegenden Kriterien. Dazu zählt, dass die jeweiligen Werte nicht negativ sein dürfen sowie der Wert für ein Tagdatum größer als 31 beziehungsweise das Monatsdatum über 12 ungültig ist. Für die der Gültigkeit des Mindesthaltbarkeitsdatums als ein grundsätzlich gültiges Datum dient die Klasse `DateValidator`.

## **Application Code-Schicht**

Eine weitere Schicht ist die `Application Code-Schicht`. Darin befinden sich die Klassen `ConsumerGoodsManager` sowie `StorageManager`. In diesen Klassen findet die Applikationslogik statt. Innerhalb der Klasse `ConsumerGoodsManager` bedeutet das konkret, das Anlegen neuer `ConsumerGoods`-Objekten, das Ausgeben aller aktuell gespeicherter `ConsumerGoods`-Objekten, Verändern eines `ConsumerGoods`-Objekts sowie das Löschen von `ConsumerGoods`-Objekten.

Die Klasse `ConsumerGoodsRepositoryBridge` aus der *Plugin*-Schicht fungiert als `Repository`-Klasse für die `ConsumerGoods`-Objekte. Die beiden Klassen haben das Interface `ConsumerGoodsRepository` implementiert. Zum Ausführen der Applikationslogik eine Abhängigkeit in eine äußere Schicht nötig. Um dies zu umgehen, wird als Übergabeparameter des Konstruktors ein Objekt, welches das Interface `ConsumerGoodsRepository` implementiert hat, übergeben. Somit ist zur Kompilierungszeit keine Abhängigkeit vorhanden und ein Verwenden der äußeren Klasse zur Laufzeit möglich. Durch das Anwenden der `Dependency Injection` in der Klasse *ConsumerGoodsManager* kann daraus eine *Dependency Inversion* angewandt werden, die das Problem der Abhängigkeit zur Kompilierungszeit in eine äußere Schicht löst. Sollte nun die Klasse `ConsumerGoodsRepositoryBridge` in der *Plugin*-Schicht aufgrund eines Technologiewechsels verändert werden, kann bei Implementierung des Interfaces `ConsumerGoodsRepository` unberührt weiterverwendet werden.

Die Klasse `StorageManager` hat die gleiche Applikationslogik wie die Klasse `ConsumerGoodsManager` jedoch mit dem Unterschied, dass es sich auf `Fridge`-Objekte sowie `FoodShelf`-Objekte bezieht. Analog hier ist ebenfalls ein Anlegen, Ausgeben, Verändern und Löschen eines `Fridge`-Objektes oder `FoodShelf`-Objektes möglich.

Auch in der Klasse `StorageManager` findet eine *Dependency Inversion* für die Klassen der *Repositories* für die einzelnen *Storage*-Ausprägungen statt. Die Klassen `FridgeRepositoryBridge` und `FoodShelfRepositoryBridge` dienen als *Repository*-Klassen für die jeweiligen *Fridge*-Objekte und *FoodShelf*-Objekte. Die beiden Klassen haben das Interface `FridgeRepository` beziehungsweise `FoodShelfRepository` implementiert. Auch hier wäre zum Ausführen der Applikationslogik eine Abhängigkeit in eine äußere Schicht nötig. Um dies zu umgehen, wird auch hierbei als Übergabeparameter des Konstruktors ein Objekt, welches das Interface `FridgeRepository` beziehungsweise `FoodShelfRepository` implementiert hat, übergeben. Somit ist zur Kompilierungszeit keine Abhängigkeit vorhanden und ein Verwenden der äußeren Klasse möglich.

## Adapters-Schicht

Eine weitere Schicht ist die *Adapters-Schicht*. Diese Schicht dient als *Anti Corruption Layer*, indem sie die inneren Schichten von den äußeren Schichten trennt. Dementsprechend sind die Domänen-Klassen aus der *Domain*-Schicht, die zur Kommunikation mit den äußeren Schichten benötigt werden, in dieser Schicht repliziert. Dazu zählen die in Tabelle 3.1 mit den in der *Domain*-Schicht gegenübergestellten Klassen:

Domain-Schicht	Adapters-Schicht
ConsumerGoods	ConsumerGoodsResource
Food	FoodResource
BestBeforeDate	BestBeforeDateResource
Food	FoodResource
Storage	StorageResource
Fridge	FridgeResource
FoodShelf	FoodShelfResource

Tabelle 3.1: Gegenüberstellung der in der *Adapters*-Schicht abgebildeten Klassen aus der *Domain*-Schicht.

Durchgeführt wird diese Trennung der inneren und äußeren Schichten durch die Klassen `ConsumerGoodsToConsumerGoodsResourceMapper`, `FridgeToFridgeResourceMapper`



sowie `FoodShelfToFoodShelfMapper`. Die Erstellung neuer Resource-Objekte beruht auf den Daten der Domänen-Objekte. Die dadurch erzeugten Resource-Objekte können nun von Objekten der Klassen in der *Plugin*-Schicht verwendet werden.

## ***Plugin*-Schicht**

Die äußerste Schicht bildet die *Plugin*-Schicht.

### **Main**

Zum einen enthält die *Plugin*-Schicht die Main-Methode zum Starten des Projekts. Aufgrund der Tatsache, dass die Main-Methode technisch bedingt und somit keinerlei Bezug zur Anwendung und deren Geschäftsprozesse hat, ist sie in der äußersten Schicht anzusetzen. Dementsprechend befindet sich die Main-Methode in der Klasse `ConsumerInventoryPlannerApplication`, die sich im Package `de.dhbw.cip.main` befindet.

### **JPA**

Daneben gibt es in der *Plugin*-Schicht das Package `de.dhbw.cip.plugins.persistence.hibernate`. Es enthält die Klassen `ConsumerGoodsRepositoryBridge`, `FridgeRepositoryBridge` sowie `FoodShelfRepositoryBridge`. Diese Klassen dienen als Repository-Klassen für die Objekte der jeweiligen Klassen `ConsumerGoods`, `Fridge` sowie `FoodShelf`. Die Klassen haben jeweils das aus der *Domain*-Schicht dazugehörige Interface implementiert. Die Bridge dient dabei als Bindeglied zwischen dem internen Repository sowie der durch das Spring Framework bereitgestellten Oberklasse `JpaRepository`, welche für die spätere Anbindung an eine Datenbank zur Persistierung benötigt wird. Dadurch kann eine Trennung zwischen der Geschäftslogik und der Persistierung durch das Spring Framework durchgeführt werden. Die Bridge entspricht zudem einem Entwurfsmuster und wird im Kapitel 7 ebenfalls analysiert. Des Weiteren sind in dem Package die Klassen `PersistenceConsumerGoodsRepository`, `PersistenceFridgeRepository` sowie `PersistenceFoodShelfRepository` enthalten. Wie bereits erwähnt, erben diese Klassen von der Oberklasse `JpaRepository`. Diese Klassen wird durch das

Spring-Framework bereitgestellt und dient der Persistierung auf einer Datenbank. Diese Klassen stellen somit die Möglichkeit bereit, die Anbindung an eine Datenbank zu ermöglichen. Die Implementierung ist jedoch nicht in den Anforderungen festgelegt, weshalb ausschließlich die geforderten Grundlagen zur Umsetzung bereitgestellt sind.

## REST-Controller

Zuletzt findet sich in der *Plugin*-Schicht das Package `de.dhbw.cip.plugins.rest`. Darin befinden sich die Klassen `ConsumerGoodsGuiController` sowie `StorageGuiController`. Bei diesen Klassen handelt es sich um den REST-Controller zur Interaktion über das HTTP-Protokoll mit der Graphical User Interface (GUI). Die GUI ist mit der Technologie *Angular* erstellt worden. Sie dient lediglich der Repräsentation der Funktionalität des REST-Services und wird daher innerhalb des Softwareentwurfs nicht tiefer analysiert. Der REST-Controller dient somit als Schnittstelle zur GUI und befindet sich daher in der *Plugin*-Schicht. Zum Erhalt der darzustellenden Daten werden Instanzen des jeweiligen Application Services sowie des Resource Mappers als Übergabeparameter des Konstruktors übergeben. Dadurch können zum Senden der Entitäten die in der *Adapter*-Schicht erzeugten Resource-Entitäten gestreamt und als Liste gesendet werden. Gleichzeitig können über eine Abhängigkeit zum entsprechenden Application Service Entitäten entsprechend der über die GUI übermittelnden Daten erzeugt, verändert oder gelöscht werden. Die Code-Ausschnitte verdeutlichen die Interaktionen des REST-Controllers am Beispiel der Klasse *ConsumerGoodsGuiController*.

## Testklassen

In der Plugin-Schicht befinden sich ebenfalls die Klassen `AddConsumerGoodsTest`, `DateValidatorTest`, `DayDateTest`, `MonthDateTest`, `RestTest`, `UnitOfMeasureValueTest`, `UpdateConsumerGoodsTest` und `YearDateTest`. Diese Klassen dienen für Unit-Tests und werden im Kapitel 6 ausführlicher beschrieben.

## Frameworks

Während die Verwendung von Bibliotheken Funktionalitäten einer Klasse innerhalb einer Schicht erweitern, versuchen Frameworks Einfluss auf die gesamte Struktur und somit auch schichtübergreifend zu nehmen. Neben der Wahl der Programmiersprache ist das eine weitere grundlegende Entscheidung, die zu Beginn getroffen wird und zu grundlegenden Abhängigkeiten der Software führen kann. Der Nachteil ist, dass die Entwicklung des Frameworks nicht beeinflusst werden kann. Somit können Weiterentwicklungen nicht mehr mit der Konzeption des Softwareentwurfs übereinstimmen und das hätte zur große Wirkungen in Form von Änderungen oder einer Neuentwicklung zur Folge. Deshalb wird die Wahl eines Frameworks als grundlegende Abhängigkeit gesehen und sollte bedacht gewählt werden. Es bietet sich an, soweit das Framework dies zulässt, die Funktionalitäten bestenfalls an die *Plugin*-Schicht zu drängen.

Bei diesem Softwareentwurf wurde das Framework SpringBoot gewählt. SpringBoot vereinfacht die Anwendung von Webanwendungen, wie in diesem Softwareentwurf in Form eines Webservices. Das Framework stellt nötige Webserver bereit und vereinfacht die Anbindung mit Frameworks zur Persistierung wie beispielsweise Hibernate. Das unterstützt bei der Entwicklung eines Webservices.

## Strukturierung

Ein Vorteil ist die Struktur, die SpringBoot vorgibt. Dadurch kann es bei der Umsetzung der definierten Schichten der *Clean Architecture* für die Controller-Klassen und Service-Klassen unterstützen. Es findet jedoch durch die Annotation des Frameworks zugleich eine Abhängigkeit ab. Beispiele sind die Kennzeichnung des Controllers mittels *@RestController* in der Klasse `ConsumerGoodsGuiController` und `StorageGuiController` sowie die Kennzeichnung einer Service-Klasse mit der *@Service* Annotation in den Klassen `ConsumerGoodsManager` und `StorageManager`.

## Dependency Injection durch Inversion of Control

Ein weiterer Vorteil von Spring Boot ist der Spring Inversion of Control Container. Der Inversion of Control Container verwaltet zur Laufzeit die erzeugten Beans und ermöglicht

sehr einfach Dependency Injection, indem der Container die Zuweisung übernimmt, wie es beispielsweise in den Klassen `ConsumerGoodsManager` oder `ConsumerGoodsGuiController` der Fall ist.

Auch hierbei ist zu beachten, dass durch die nötige Annotation `@Autowired` des Frameworks eine gewisse Abhängigkeit stattfindet, wie es beispielsweise in den Klassen `ConsumerGoodsManager` oder `ConsumerGoodsGuiController` der Fall ist.

## Persistierung

Eine weitere Unterstützung bietet Spring Boot bei der Persistierung. Hierbei wurde die Persistierung jedoch mithilfe des *Bridge*-Entwurfsmusters, wie in Kapitel 7 genauer erläutert, in die Plugin-Schicht gedrängt. Beispielklassen `ConsumerGoodsBridge` und `PersistenceConsumerGoodsRepository`

Jedoch befinden sich die nötigen Annotationen in der Domäne, was sehr abhängig ist und im Gegensatz zu den bereits erwähnten Annotationen gewissermaßen stärker in das Gewicht fällt, weil es sich hierbei um die Domänenklassen handelt und diese unabhängig von verwendeten Technologien sein sollten. Ein nichtauszuschließender Technologiewechsel führt somit zu einer Anpassung der Klassen in der *Domain*-Schicht, was die Umsetzung der *Clean Architecture* verletzt. Als Beispiel zählen die nötigen Annotierungen die Kennzeichnung als Entität, zur späteren Bezeichnung von Tabellen, in denen die Attributwerte persistiert werden oder nötige Metainformationen wie beispielsweise die Zuweisung der Funktion eines Attributs als Id. Ein Beispiel sind die Annotationen `@Entity`, `@Id`, `@Column` und `@OneToOne` in der Klasse `ConsumerGoods`.

Ein weiterer Nachteil ist, dass zur Persistierung Attributtypen eindeutig zuweisbar sein müssen. Das bedeutet, dass Attribute, die mit einem Interface als Typ deklariert werden, nicht möglich sind. Stattdessen muss eine Deklaration einer konkreten Klasse oder Superklasse erfolgen. Ein Beispiel ist die Deklaration zur Zuweisung der Menge und des Lagerplatzes in der Klasse `ConsumerGoods` mit den Superklassen `UnitOfMeasure` sowie `Storage` realisiert. Wäre diese Einschränkung für das Persistieren nicht gegeben, könnten `UnitOfMeasure` und `Storage` auch als Interface realisiert sowie das Interface `StorableGood`, dass von der Klasse `Food` daraufhin implementiert werden kann, genutzt werden.

## 4 Programming Principles

Im Folgenden Kapitel werden die *Programming Principles* erläutert und in Bezug auf den Programmmentwurf analysiert sowie Umsetzungen begründet.

### 4.1 Zu betrachtende Programming Principles

Programming Principles dienen als Prinzipien in der Softwareentwicklung. Dementsprechend sind Programming Principles als allgemein anerkannte Regeln zur Begründung und Argumentation von Umsetzungen zu verstehen. Sie entsprechen somit Grundlage für Entscheidungen und betrachten dabei das gesamte Softwareprojekt. Die Programming Principles sind dabei eine kontextlose Idealvorstellung und haben die Funktion als Leitlinien für zielgerichtetes Handeln. Das hat zur Folge, dass kontextabhängig gewisse Abweichungen nötig sein können. Daher gilt es, neben dem Anwenden der Prinzipien auch mögliche Abweichungen zu begründen. Gleichzeitig können sich die jeweiligen Prinzipien gegeneinander widersprechen. Hierbei ist es nötig, kontextabhängig entsprechend abzuwirken und die Entscheidung ebenfalls zu begründen.

Zu den betrachteten Programming Principles zählt SOLID. SOLID setzt sich aus den Prinzipien

- *Single Responsibility Principle* (SRP),
- *Open/Closed Principle* (OCP),
- *Liskov Substitution Principle* (LSP),
- *Interface Segregation Principle* (ISP),
- *Dependency Inversion Principle* (DIP)

zusammen. Die SOLID-Regeln haben das Ziel, unter Anwendung der Regeln wartbare sowie erweiterbare Systeme sowie eine langlebige Codebasis zu schaffen.

Um dies zu gewährleisten, beschreibt die erste Regel, das *Single Responsibility Principle*, das Prinzip der einzigen Zuständigkeit. Die Regel besagt, dass eine Klasse nur einen einzigen Grund haben sollte, sich zu ändern. Eine Klasse sollte somit nur eine Verantwortlichkeit haben. Ein Objekt der Klasse hat somit eine klar definierte Aufgabe und übergeordnetes Verhalten wird durch das Zusammenspiel mehrerer Klassen ermöglicht. Damit soll vermieden werden, dass es zu einer Überdeckung des Sourcecodes bezogen auf dessen Anwendung für die Zuständigkeit kommt. Dieses Problem bezeichnet man als Feature Drift. Eine Änderung an diesem Sourcecode, der für mehrere Zuständigkeiten genutzt wird, hätte zur Folge, dass ungewollt eine anderweitige Zuständigkeit der Klasse verändert wird. Das Problem kann auch visuell mittels eines Koordinatensystems dargestellt werden. Jede Achse beschreibt eine Zuständigkeit. Änderungen entlang der Achse führen zu Codeanpassungen. Im Optimalfall beeinflussen sich die Zuständigkeit nicht gegenseitig. Somit würde die Änderung des gemeinsamen Punktes auf der entsprechenden Zuständigkeitsachse zu keinen Änderungen auf den anderen Achsen und somit auch zu keiner Änderung des Codes der anderen Zuständigkeiten führen. Dieser Fall ist jedoch in der Realität selten anzutreffen. Unter Anwendung der Regel resultiert eine niedrigere Kopplung und Komplexität des Codes. Mehrere Verantwortlichkeiten beziehungsweise die Zuständigkeit einer Klasse sollte dabei zu erkennen sein.

Eine weitere Regel ist das *Open/Closed Principle*. Diese Regel besagt, dass Software-Entitäten, wie beispielsweise Klassen oder Methoden, offen für Erweiterungen aber zeitgleich geschlossen für Veränderungen sein sollen. Zu Veränderungen zählen hierbei Codemodifikationen aufgrund geänderter Anforderungen. Bestehender Code sollte dabei nicht geändert werden müssen und angepasste Anforderungen führen somit nur zu einer Erweiterung des Codes. Eine mögliche Erweiterung kann dabei eine Vererbung darstellen. Die Klasse wird somit durch die Unterklasse erweitert, zeitgleich findet in der Klasse keine Veränderung statt. Dabei ist auch hier zu erkennen, dass kein Programm komplett immun gegen Modifikationen sein kann. Der Entwickler kann jedoch bestimmen, welche Änderungen durch Erweiterungen ermöglicht werden. Hierfür ist jedoch eine Erfahrung in der Domäne und der Umsetzung nötig. Das *Open/Closed Principle* ist ein wichtiges Werkzeug, dass es zu kennen gilt, jedoch sollte es kein beherrschendes Designziel sein, da es sonst zu einer spekulativer Komplexität führen kann.

Die dritte Regel ist das *Liskov Substitution Principle*. Diese Regel legt fest, dass Objekte in einem Programm durch Instanzen ihrer Subtypen ersetzbar sein sollten, ohne die

Korrektheit des Programms zu ändern. Die Regel gibt somit strikte Regeln für Vererbungshierarchien vor und befasst sich daher mit den Varianzregeln. Varianzregeln beschreiben die Ersetzbarkeit eines Objekts durch ein Objekt der Ober- oder Unterklasse. Es gibt dabei drei Arten: Kovarianz besagt, dass die Typhierarchie und die Vererbungshierarchie die gleiche Richtung haben. In der Programmiersprache Java wird Kovarianz präferiert. Kontravarianz beschreibt, dass die Typhierarchie entgegengesetzt zur Vererbungshierarchie ist. Invarianz bedeutet, dass die Typhierarchie unverändert bleibt. Zur Umsetzung des *Liskov Substitution Principle* müssen sich daher Subtypen so verhalten wie ihr Basistyp. Ein Subtyp darf dabei die Funktionalität erweitern, jedoch nicht einschränken. Hierbei zeigen sich Parallelen zum *Open/Closed Principle*. Somit ist das *Liskov Substitution Principle* erfüllt, wenn man jede Spezialisierung einer Generalisierung überall dort einsetzen kann, an den Stellen die Generalisierung verwendet wird.

Eine weitere Regel des SOLIP Prinzips ist das *Interface Segregation Principle*. Diese Regel besagt, dass mehrere spezifische Interface besser sind als ein Allround-Interface. Interface sollen Klient-spezifisch sein. Daraus resultiert eine höhere Kohäsion und somit repräsentieren Klassen oder Interfaces eine Einheit sehr genau. Die Regel unterstützt somit die erste Regel, das *Single Responsibility Principle*. Ein Klient soll dabei nicht von Details abhängig sein, die es gar nicht benötigt. Das würde im Fall eines Interfaces Methoden des Interfaces bezeichnen, die der Klient nicht benötigt. Stattdessen sollten Schnittstellen oder Interfaces möglichst passgenau für den Klienten sein. Daraus resultieren mehrere Schnittstellen für einen Klienten. Die Schnittstellen können dabei domänenspezifisch sowie technischspezifisch sein. Das Ziel ist somit, die Schnittstellen in Nutzergruppen aufzuteilen.

Die letzte Regel des SOLID Prinzips ist das *Dependency Inversion Principle*. Die Regel beschreibt das Prinzip der Entkopplung. Die Intuition des Prinzip ist es, dass Abstraktionen nicht von Details abhängen sollten sondern Details von Abstraktionen. Somit sollten Module höherer Ebenen nicht von Modulen niedriger Ebenen abhängen, stattdessen sollten beide von Abstraktionen abhängen. Die Lösung dieses Problems ist, dass ein höheres Modul eine Schnittstelle definiert und ein niedrigeres Modul diese implementiert. Konkret bedeutet dass, dass Klassen höherer Ebenen nicht von Klassen niedrigerer Ebenen abhängig sein sollen, sondern beide Klassen von Interfaces. Schließlich ist die Abhängigkeit auf eine konkrete Klasse eine starke Kopplung. Methoden zur Auflösung dieser Kopplung ist die *Dependency Injection* sowie *Dependency Inversion*. Aus der Anwendung

des *Dependency Inversion Principle* resultiert eine Entkopplung der Implementierung. Die einzelnen Module können somit flexibler miteinander zusammenarbeiten. Zudem erhält man eine bessere Wiederverwendbarkeit und Schnittstellen werden dadurch zudem klarer und somit wird die Anforderungen deutlicher.

Ein weiteres Programming Principle ist General Responsibility Assignment Software Patterns/Principles (GRASP). GRASP stellt Standardlösungen für typische Fragestellungen der Softwarekonzeption bereit. Innerhalb dieses Softwareentwurfs liegt der Fokus ausschließlich auf den Grundkonzepten *Low Coupling* sowie *High Cohesion*. Darüber hinaus gibt es sieben weitere Werkzeuge, die jedoch nicht Teil der Anforderungen dieses Software-Entwurfs sind.

*Low Coupling* zielt auf eine geringe Kopplung ab. Kopplung bezeichnet das Maß für die Abhängigkeit einer Klasse von ihrer Umgebung, wie zum Beispiel anderer Klassen. Eine geringe Kopplung unterstützt daher eine gute Testbarkeit, leichte Anpassbarkeit, eine bessere Verständlichkeit aufgrund geringeren Kontexts sowie eine erhöhte Wiederverwendbarkeit. Kopplung im Sourcecode kann

Grundsätzlich bietet eine geringere Kopplung eine bessere Austauschbarkeit des nächsten Befehls. jedoch ist dabei wichtig, dass eine minimale Kopplung nicht immer die beste Umsetzung bedeutet. Vielmehr ist eine vernünftige Umsetzung einer geringen Kopplung sinnvoll. Darüber hinaus finden sich auch weitere Kopplungsarten, wie beispielsweise die Kopplung an Datentypen, Kopplung der Threads, Kopplung durch Formate oder Protokolle sowie Kopplung durch Ressourcen.

Ein weiteres Werkzeug ist *High Cohesion*. *High Cohesion* zielt auf eine hohe Kohäsion ab. Kohäsion ist das Maß für den inneren Zusammenhalt einer Klasse und beschreibt somit, wie eng Methoden und Attribute auf semantischer Nähe miteinander zusammenarbeiten. Idealer Code zeichnet sich durch *Low Coupling* in Kombination mit *High Cohesion* aus. Die Kohäsion kann unter anderem durch Kopplung erhöht werden. Dazu zählt beispielsweise das Erstellen von Klassen, die eine Eigenschaft der eigentlichen Klasse beinhalten. Kohäsion ist dabei ein semantisches Maß, somit ist die menschliche Einschätzung darüber entscheidend. Kohäsiver Code bietet zudem den Vorteil zur Analyse. Durch die Aufteilung der Klassen in Eigenschaften tendiert der Code zur Kürze. Zudem kann er als Anfangsverdacht für sogenannte *Code Smells* dienen. Eine ausführlichere Beschreibung zu *Code Smells* ist Teil des Themas *Refactoring*.



Zu guter Letzt wird das Programming Principle Don't Repeat Yourself (DRY) betrachtet. DRY folgt dem Prinzip, dass jeder Wissensaspekt nur eine einzige, eindeutige und verbindliche Replikation in einem System besitzt. Dabei ist eine mechanische Duplikation erlaubt, soweit die Originalquelle klar definiert ist. DRY versucht das Problem zu beheben, dass Quellcode durch mehrfache Wissensaspekte weniger eindeutig wird. Als Beispiel zählt die Größenangabe einer Oberfläche. Eine zentrale Angabe der Maße verhindert beim Anpassen der Größe das Missachten einer Stelle im Code, wodurch es bei einer Änderungen zu ungewollten Fehlern, in Form von unterschiedlich großen Oberflächen, kommen kann. Diese Regel lässt sich sowohl auf Quellcode als auch auf Dokumentationen oder sonstige technische Pläne umsetzen.

Darüber hinaus gibt es mit Keep it Simple, Stupid (KISS), You Ain't Gonna Need it (YAGNI) sowie dem Conway's Law weitere Programming Principles, die jedoch nicht innerhalb dieses Softwareentwurfs betrachtet werden.

## 4.2 Analyse und Begründung

In diesen Abschnitt werden die zu betrachteten Programming Principles in Bezug auf den Softwareentwurf analysiert und Anwendungen entsprechend begründet.

### 4.2.1 SOLID

Im Folgenden werden die einzelnen Prinzipien des SOLID-Prinzips in Bezug auf den Softwareentwurf betrachtet.

#### Single Responsibility Principle

Ein Beispiel in Bezug auf das *Single Responsibility Principle* ist die Klasse `ConsumerGoodsToConsumerGoodsResourceMapper`. Die Klasse hat die einzige Aufgabe, Objekte der Klasse `ConsumerGoods` zu Objekten der Klasse `ConsumerGoodsResource` zu mappen und somit für die Interaktion mit der GUI zu verwenden. Das gleiche gilt für die Klasse `FridgeToFridgeResourceMapper` und `FoodShelfToFoodShelfResourceMapper`, deren Aufgabe das Mappen eines `Fridge`- oder `FoodShelf`-Objekt zu einer entsprechenden Resource ist.

Das *Single Responsibility Principle* wurde an dieser Stelle eingehalten, da auch die gemappten Klassen für die Kommunikation über die Schnittstellen in der Form nötig sind, wie sie in der Domäne abgebildet sind.

Das *Single Responsibility Principle*-Prinzip wurde hingegen bei den Klassen `ConsumerGoodsManager` sowie `StorageManager` verletzt. Die Klassen haben neben dem Suchen eines Objekts auch die Aufgabe der Erstellung sowie das Löschen eines Objekts. In diesem Zusammenhang lässt sich ebenfalls eine Verletzung des *Single Responsibility Principle*-Prinzip bei der Klasse `ConsumerGoodsGuiController` erkennen. Die Interaktion mit der GUI umfasst sowohl das Erstellen, als auch das Ausgeben der verfügbaren Objekte der Klasse `ConsumerGoods`. Gleiches gilt für das Löschen eines Objektes. Hierzu findet die gesamte Interaktion, die verschiedene Aufgaben enthält, in der Klasse `ConsumerGoodsGuiController` statt. Die Verletzung des *Single Responsibility Principle*-Prinzip trifft auch auf die Klasse `StorageGuiController` zu. Auch hier findet die gesamte Interaktion mit der Schnittstelle, worunter das Ausgeben, Erstellen und Löschen von Objekten des implementierten Interfaces `Storage` zählt, in der einen Klasse statt.

Die Verletzung des *Single Responsibility Principle* wurde hierbei in Betracht gezogen, die Klassen die zentrale Möglichkeit des Verwalten der Konsumgüter und Lagerplätze übernehmen und hierbei zum einen übergreifend Löscho- oder Anlegeoperationen beim Aktualisieren eines Konsumguts gemacht werden können und zugleich die zentrale Klasse die Möglichkeit der Integration zukünftiger Business-Logik bietet.

## Open-Closed-Principle

Das *Open-Closed-Principle* wurde bei den Klassen `ConsumerGoodsResource`, `FridgeResource` und `FoodShelfResource` sowie bei den weiteren Ressourcen-Klassen eingehalten. Das Einfügen neuer Funktionen wie beispielsweise weitere abspeichbare Güter oder weitere Lagermöglichkeiten können umgesetzt werden, indem neue Ressourcen-Klassen hinzugefügt werden. Ein Verändern des bestehenden Codes ist hierzu nicht nötig.

Die Einhaltung des *Open-Closed-Principle* ist an dieser Stelle dadurch gegeben, da die Ressourcen-Klassen Klassen aus der Domäne abbilden und somit eine Erweiterung in der Domäne eben zu einer Erweiterung durch Hinzufügen einer neuen Ressourcen-Klasse führt.

Eine Verletzung des *Open-Closed-Principle* stellen die Klassen `ConsumerGoodsGuiController` und `StorageGuiController` dar. Eine Erweiterung der Anwendung um ein weiteres zu verwaltes Gut hätte hierbei zur Folge, dass die Einbindung eines neuen Guts zu einer Veränderung der Schnittstelle für die GUI führt, um das neue Gut in der GUI zu repräsentieren.

Das *Open-Closed-Principle* ist an dieser Stelle verletzt worden, weil eine Abweichung vom REST-Paradigma vorgenommen wurde und stattdessen Home-Made-Messaging umgesetzt wurde. Daraus resultiert diese Verletzung und führt dazu, dass das GUI-Plugin aktuell sehr stark an die Schnittstelle angepasst ist.

### **Liskov Substitution Principle**

Es ist eine Vererbung der Superklasse `UnitOfMeasure` und den Subklassen `Volume`, `Weight` und `Quantity` implementiert. Bei `UnitOfMeasure` handelt es sich um eine abstrakte Klasse. Die Subklassen überschreiben jedoch keine Funktionalität der Superklasse, weshalb das *Liskov Substitution Principle* an dieser Stelle eingehalten wird.

Das Gleiche gilt für die Implementierung der Vererbung der Superklasse `Storage` und den Subklassen `Fridge` und `FoodShelf`. Auch in diesem Fall ist die Klasse `Storage` abstrakt und die Subklassen überschreiben keine Funktionalität der Superklasse, wodurch das *Liskov Substitution Principle* eingehalten wird.

Das *Liskov Substitution Principle* wurde hierbei eingehalten, weil sich die Vererbung stark an der Nutzung von Interfaces orientiert. Diese sind auch an dieser Stelle für die Superklassen `UnitOfMeasure` und `Storage` vorgesehen gewesen, allerdings führt das bei der Persistierung durch Hibernate zu Fehlern, da Hibernate die Referenz auf den direkten Typen benötigt und dieser bei Interfaces für Hibernate nicht ermittelbar ist.

### **Interface Segregation Principle**

Ein Positivbeispiel der Umsetzung des *Interface Segregation Principle* zeigt sich in den Interfaces `FridgeRepository` und `FoodShelfRepository`.

Hierbei wurde in Hinblick auf das *Interface Segregation Principle* bedacht, dass unterschiedliche Lagermöglichkeiten unterschiedliche Eigenschaften und Funktionalitäten

bieten können, die in einem gemeinsamen Repository-Interface nicht abbildbar wären. Diese Entscheidung ist auch im Hinblick der Erweiterbarkeit in Form zukünftiger weiterer Lagermöglichkeiten konzipiert.

Die Interfaces können jedoch zugleich als Negativbeispiel betrachtet werden. Die Repository-Interfaces übernehmen sowohl Lesezugriff als auch die Möglichkeit der Übergabe zur späteren Persistierung. Nach dem *Interface Segregation Principle* wäre eine Trennung nach Leseoperationen und Schreiboperationen in separate Interfaces geeigneter. Im Hinblick auf mögliche Erweiterungen könnte es sonst zu Problemen oder Umständen bei der Implementierung einer Berechtigungskontrolle auf die verwalteten Ressourcen kommen. Diesen Negativpunkt betrifft ebenfalls das Interface `ConsumerGoodsRepository`.

### Dependency Inversion Principle

Die Einhaltung des *Dependency Inversion Principle* war bei den Klassen `ConsumerGoods` sowie `Food` und dem Interface `StorableGood` geplant. Die Klasse `ConsumerGoods` referenziert dabei auf ein Objekt des Typs `StorableGood`. Das hätte zur Folge, dass unter Anwendung der Dependency Injection eine Erweiterung weiterer lagerbarer Objekte neben Essen möglich ist, ohne die Klasse `ConsumerGoods` hierzu anpassen zu müssen. Die Klasse `ConsumerGoods` referenziert dabei auf ein Objekt des Typs `StorableGood`.

Gleiches gilt für die Umsetzung der Superklassen `Storage` und `UnitOfMeasure` als Interfaces in Bezug auf die Klasse `ConsumerGoods`. Auch hierbei könnten neue Lagermöglichkeiten oder Maßeinheiten hinzugefügt werden, ohne dass die Klasse `ConsumerGoods` dafür angepasst werden muss.

Es ist zu erwähnen, dass es sich in beiden Betrachtungen um Planungen handelt, deren resultierender Vorteil beschrieben wurde. Aufgrund der Tatsache, dass die Persistierung mit Hibernate in den Domain-Klassen umgesetzt wurde, kommt es hierbei zu dem Problem, dass Hibernate über den Typen des Interfaces die Referenz nicht auflösen kann. Daher ist zum aktuellen Zeitpunkt noch das referenzierte Gut vom Typ `Food` sowie `Storage` und `UnitOfMeasure` als Superklassen implementiert.

Darüber hinaus sind die Interfaces `ConsumerGoodsRepository`, `FridgeRepository` und `FoodShelfRepository` ein weiteres und implementiertes Beispiel des *Dependency Inversion Principle*. Die Interfaces definieren alle nötigen Methoden und die Services

ConsumerGoodsManager und StorageManager haben zur Laufzeit eine Abhängigkeit auf ein Objekt, welches das entsprechende Interface implementiert hat. Durch die Einhaltung des *Dependency Inversion Principle* resultiert der Vorteil, dass die tatsächlichen Objekte problemlos ausgetauscht werden können und das Einsetzen von Mocks zum Testen vereinfacht wird, indem die entsprechenden Methoden des Interfaces gemockt werden.

Eine Verletzung des *Dependency Inversion Principle* ist in den Mapper Klassen der Ressourcen-Mapper-Klassen zu finden. Die Verwendung der Klassen ConsumerGoodsToConsumerGoodsRepository, FridgeToFridgeResourceMapper sowie FoodShelfToFoodShelfResourceMapper hängt direkt von den jeweiligen Instanzen der Mapper-Klassen ab. Eine Lösung zum Beheben der Verletzung wäre das Nutzen eines Interfaces, das die Ressourcen-Mapper-Klassen implementieren. Dadurch wäre eine stärkere Entkopplung möglich. Dennoch wurde hierbei die Verletzung des *Dependency Inversion Principle* zum besseren Verständnis und einer besseren Lesbarkeit durch eine klarere Zuordnung innerhalb des Programmcodes gemacht.

## 4.2.2 GRASP

### Geringe Kopplung

Eine geringe Kopplung findet sich zwischen der GUI und der Applikationsschicht der Anwendung. Die Interaktion findet über die Klasse ConsumerGoodsGuiController und StorageGuiController statt. Beide Klassen dienen als HTTP-Interface. Durch die Kommunikation über HTTP kennen sich die Anwendung und die GUI nicht und sind somit entkoppelt.

Hierbei wurde auf eine geringe Kopplung geachtet, da GUI-Plugins und die dafür verwendete Technologie tendenziell zu kürzeren Lebenszyklen neigen. Durch die geringe Kopplung ist ein Austausch durch eine andere Technologie für die grafische Oberfläche, die eine Kommunikation über das HTTP-Protokoll ermöglicht, umsetzbar ohne Änderungen an dem Service vornehmen zu müssen. Zudem ist dadurch eine Kommunikation mit weiteren Plugins über das HTTP-Protokoll möglich.

Eine geringe Kopplung wurde bei der Implementierung ebenfalls erzielt, indem die Repository-Klassen ConsumerGoodsRepository, FridgeRepository und FoodShelfRepository

bei der Rückgabe der instanziierten Objekte diese nicht als Liste vom Typ *List* zurückgeben sondern als *Iterable*. Hierdurch wird die Kopplung, welche die Repository-Interfaces sonst durch den Rückgabebetyp erzeugen, reduziert. Einzelne Objekte werden zudem als *Optional* zurückgegeben, was vor allem entkoppelt, da keine Regelungen getroffen werden müssen, falls kein den übergebenen Kriterien entsprechendes Objekt vorhanden ist, das zurückgegeben werden kann.

Es wurde hierbei auf eine geringe Kopplung geachtet, da es die Möglichkeit bietet, in Zukunft fortschrittlichere konkrete Implementierung des Typs *Iterable* zu verwenden und nicht an einen konkreten Typ, der gegenwärtig aktuell ist, gebunden zu sein.

Eine stärkere Kopplung ist in den Serviceklassen `ConsumerGoodsManager` und `StorageManager` zu finden. Zum Löschen, Persistieren oder der Rückgabe initialisierter Objekte der Klassen `ConsumerGoods`, `Fridge` oder `FoodShelf` werden ausschließlich die in den entsprechenden Repository-Interfaces definierten Klassen verwendet. Für den Zugriff auf eine Referenz des jeweiligen implementierten Repository-Typs wird durch eine Dependency Injection in `ConsumerGoodsManager` sowie in `StorageManager` ermöglicht. Somit ist hierbei keine direkte Kopplung an eine konkrete Implementierung eines Repository-Objekts vorhanden.

Es wurde hierbei auf eine ebenfalls geringe, wenn auch stärkere als zuvor, Kopplung geachtet, da somit ein Austausch des konkreten Typs des entsprechenden Repositories möglich ist. Das ist gerade in Hinblick auf den Austausch der Technologie zur Persistierung relevant und entkoppelt die Business-Logik von der konkreten Umsetzung der Persistierung.

Ein Beispiel mit einer starken Kopplung ist der statische Methodenaufruf `validate()` der Klasse `DateValidator`. Auch in diesem Fall entsteht durch den statischen Methodenaufruf eine starke Kopplung an die Klasse `DateValidator`.

Diese starke Kopplung wurde gemacht, da davon auszugehen ist, dass sich die Validierungen an Grundlagen und Rechtmäßigkeiten innerhalb der Domäne orientieren und somit der Eintritt einer Änderung als eher weniger wahrscheinlich erscheint.

## Hohe Kohäsion

Ein Beispiel für eine hohe Kohäsion ist in der Klasse `ConsumerGoods` zu sehen. Die Bestandteile eines Konsumguts sind das Lebensmittel und die dazugehörige Menge samt Maßeinheit. Das Lebensmittel wird dazu in der Klasse `Food` verwaltet, während die Menge in einem Objekt des implementierten Interface-Typs `UnitOfMeasure` verwaltet ist. Die Klasse `Food` hat als Attribut eine Instanz der Klasse `BestBeforeDate` zur Repräsentation des Mindesthaltbarkeitsdatums. In der Klasse `BestBeforeDate` werden auch datumsspezifische Operationen wie zum Beispiel die Datumsüberprüfung mithilfe des `DateValidators` übernommen. Das Objekt der Superklasse `UnitOfMeasure` repräsentiert die Menge in Form der spezialisierte Klassen `Weight`, `Volume` oder `Quantity`. Der Wert der Menge wird in den Klassen durch das Attribut vom Typ `Value` verwaltet.

Das Ziel durch Schaffung einer hohen Kohäsion ist hierdurch, dass die jeweiligen Bestandteile des `ConsumerGoods` in einzelnen Klassen verwaltet werden, dadurch teilt sich der Code auf die entsprechenden Schwerpunkte auf und wird, gerade in Bezug auf Übergabe- und Rückgabeparameter, lesbarer.

Ein weiteres Beispiel für hohe Kohäsion ist die Klasse `BestBeforeDate`. Das Mindesthaltbarkeitsdatum wird durch die Variablen des Typs `DateOfYear` und `Year` repräsentiert. `DateOfYear` verwaltet das Datum eines Jahres durch die Attribute vom Typ `Day` und `Month`.

Der Vorteil durch die Umsetzung der hohen Kohäsion ist auch hierbei, dass die Lesbarkeit bei geforderten Übergabe- und Rückgabeparametern über die genaue Typen-Bezeichnung deutlich lesbarer und verständlicher ist als ein primitiver Datentyp. Darüber hinaus ist nicht die Klasse `BestBeforeDate` für eine genaue Spezifizierung oder Anpassung der einzelnen Attribute verantwortlich sondern die jeweiligen spezifischen Klassen.

### 4.2.3 DRY

Das verwendete Framework Spring Boot bietet den Ansatz *Convention over Configuration*. Dadurch müssen bei Einhaltung von Konventionen keine zusätzlichen Konfigurationen durchgeführt werden. Ein anschauliches Beispiel des *Convention over Configuration*-Ansatzes ist das Erzeugen von Tabellennamen entsprechend der Bezeichnung der Entität.

Hierzu muss die Tabellenbezeichnung nicht zusätzlich konfiguriert werden sondern es wird direkt die Bezeichnung der Entitätsklasse verwendet. Wie im Beispiel `ConsumerGoods` zu erkennen, reicht die Deklaration `@Table` aus.

Das gleiche betrifft die Bezeichnung der Spalten entsprechend der Attributbezeichnung. Ebenfalls im Beispiel `ConsumerGoods` zu erkennen, reicht die Deklaration `@Column` aus.

Der Vorteil ist, dass bei dieser Einhaltung des DRY-Prinzips der Zusammenhang zwischen dem Programmcode und den Tabellen zur Persistierung für den Entwickler leicht zu erkennen ist und Umbenennungen der Klassen direkt zu einer Anpassung der Tabellenbezeichnung sorgt und somit Inkonsistenzen vermieden werden können.

Eine Nichteinhaltung des DRY-Prinzips findet sich in der Ressourcen-Klassen `ConsumerGoodsResource`, `FridgeResource`, `FoodShelfResource`, `FoodResource` und `BestBeforeDateResource` vor. Die Klassen dienen der Repräsentation der gleichnamigen Entitäten für die Kommunikation mit äußeren Anwendungen wie beispielsweise der GUI. Dementsprechend sind sowohl Variablen als auch Methoden dupliziert. Dadurch ist das DRY-Prinzip verletzt und hat den Nachteil, dass eine Anpassung der Domänen dazu führt, dass die entsprechende Ressourcen-Klasse ebenfalls angepasst werden müsste.

Das DRY-Prinzip wurde hierbei bewusst verletzt, weil eine Duplizierung zur Trennung des Domänencodes im Inneren zu einer Repräsentation Interaktion mit äußeren Plugins entsprechend der *Clean Architecture* gewollt ist. Hierbei ist anzumerken, dass Domänen in der *Domain*-Schicht entsprechend der *Clean Architecture* eher seltener verändert werden sollten. Dennoch würde müsste die Änderung auch in den Ressourcen-Klassen berücksichtigt werden und kann somit zu einem Fehlerrisiko durch Nichtberücksichtigung führen.



## 5 Domain Driven Design

*Domain Driven Design* beschreibt eine Form der Herangehensweise an die Modellierung von Software. Dabei wird das Design der Software maßgeblich von der Fachlichkeit der Anwendungsdomäne bestimmt, indem das Domänenmodell die Grundlage für den Entwurf und die Umsetzung der Software ist. Ein Problem stellt dabei die Komplexität dar. Die zwei Formen der Komplexität sind die inhärente Komplexität (die Komplexität der Domäne) sowie die versehentliche Komplexität (zum Beispiel die Komplexität durch Hardware, Framework oder Infrastruktur) und ergeben zusammen die Systemkomplexität. Dabei ist die Komplexität der Domäne fest gegeben. Das Ziel ist das Verhindern, dass durch die technische Umsetzung die Gesamtkomplexität negativ beeinflusst wird.

Möglich wird dies durch die Reduktion des Übersetzungsaufwands. Wichtig ist dabei, das Fachgebiet mit dem Sourcecode zu vereinheitlichen. Dazu zählt, im Sourcecode gleiche Begriffe wie in der Domäne zu verwenden und eine klare Modellierung der Fachlichkeit.

Eine weitere Möglichkeit zur Vermeidung der negativen Beeinflussung der Gesamtkomplexität, ist die Beschreibung von nützlichen Methoden und Muster. Es gibt diesbezüglich zwei Richtungen: das strategische Domain Driven Design (DDD), dass auf die Analyse, die Dokumentation sowie die Abgrenzung der Domäne abzielt und das taktische DDD, dass die Erkenntnisse in Sourcecode umsetzt.

In Bezug auf den Softwareentwurf ist die Analyse der *Ubiquitous Language* relevant. Ubiquitous Language bezeichnet die von Domänenexperten und Entwicklern gemeinsam verwendete Sprache. Schließlich ist die jeweilige Fachsprache des einen nur schwer für die andere Partei zu verstehen. Somit würde sich der Sourcecode von der Sprache der Domäne entfernen und daraus resultiert eine höhere Komplexität und ein schwereres Verständnis der Implementierung. Die entstehende Kluft soll reduziert werden, indem alle relevanten Konzepte, Prozesse und Regeln der Domäne erklärt sind. Zudem werden Zusammenhänge verdeutlicht. Mehrdeutigkeiten und Unklarheiten sollen durch die Definition der *Ubiquitous Language* beseitigt werden und die Domänensprache sollte dabei im Softwaredesign, der Dokumentation und der Bedienungsoberfläche beibehalten werden. Zu beachten ist hierbei, dass sich allerdings auf den Kern des Projekts fokussiert wird.

Darüber hinaus werden in DDD die Grundbausteine eines Modells definiert. Innerhalb dieses Softwareentwurfs werden die Bausteine

- *Repositories*,
- *Aggregates*,
- *Entities* und
- *Value Objects*

betrachtet.

*Repositories* dienen als sogenannte „Vorratsschränke“ des Systems und bieten den Zugriff auf den persistenten Speicher. Dadurch wird der Code der Domäne von den technischen Details der Speicherung getrennt. Ähnlichkeiten sind hierbei zur *Clean Architecture* zu erkennen.

*Entities* bezeichnen Objekte, die entsprechend ihrer Identität modelliert werden. Identitäten gibt es in mindestens drei Formen: der Kombination von Eigenschaften, einem Surrogatschlüssel oder der natürliche Schlüssel. Die Werte der *Entities* sind veränderlich.

*Value Objects* sind einfache Objekte ohne eigene Identität. Die Werte der *Value Objects* sind unveränderlich und *Value Objects* sind gleich, wenn deren Werte gleich sind.

*Aggregates* gruppieren *Entities* sowie *Value Objects* zu gemeinsam verwalteten Einheiten. Bei *Aggregates* übernimmt ein *Aggregate Root* die Zugriffe von außen.

## 5.1 Analyse und Begründung der *Ubiquitous Language*

Zum Verständnis der Software wurde eine Ubiquitous Language festgelegt, indem sich an der Domäne der Anwendung orientiert wurde. Die Domäne bezieht sich auf das Verwalten von Konsumgütern. Die Konsumgüter können in einem Kühlschrank oder einem Regal aufbewahrt werden. Konsumgüter haben eine Gewichtseinheit. Eine Form der Konsumgüter, die derzeit verwaltet werden soll, sind Lebensmittel. Ein Lebensmittel hat ein Mindesthaltbarkeitsdatum.

Diese Informationen über die Domäne wurden in die *Ubiquitous Language* berücksichtigt. Die Abbildung 5.1 des geschriebenen Codes in einer Wordcloud verdeutlicht die Umsetzung der *Ubiquitous Language*.



Abbildung 5.1: Die Wordcloud visualisiert die verwendeten Bezeichnungen im Programmentwurf in Bezug auf die *Ubiquitous Language*. Die Wordcloud verdeutlicht, dass die *Ubiquitous Language* umgesetzt wurde, indem statt rein technischen Begriffen hauptsächlich Bezeichnungen mit Domänenbezug benannt wurde.

Eine Umsetzung der *Ubiquitous Language* ist die Benennung der Klassen. Die Klasse `ConsumerGoods` repräsentiert gleichnamige Konsumgüter. Dazu zählt eine Assoziation auf die Klasse `Food`, die Lebensmittel repräsentieren. Lebensmittel haben ein Mindesthaltbarkeitsdatum, dass in der assoziierten Klasse `BestBeforeDate` verwaltet wird. Das Mindesthaltbarkeitsdatum wird durch die assoziierten Klassen `DateOfYear` und `Year` repräsentiert. Ein *DateOfYear* setzt sich aus den Klassen `Day` und `Month` zusammen, die entsprechend das Datum des Tages und das Datum des Monats repräsentieren. Die Entscheidung für die Repräsentation eines Tages des Jahres durch die Klasse `DateOfYear` ist, dass das MHD oftmals innerhalb eines Jahres liegt und bei der Validierung des Datums nur im Monat Februar das Jahr relevant ist. Die Klasse `DateValidator` übernimmt die entsprechende Aufgabe der Datumsvalidierung. Hinzu kommen die Klassen `DayValidator`, `MonthValidator` und `YearValidator`. Diese Klassen dienen entsprechend ihrer Bezeichnung der Validierung eines Wertes für einen Tag, Monat oder eine Jahreszahl. Die Validierung beschränkt sich insofern auf die

Domäne, dass bei sich bei den gültigen Werten an den gültigen Werten für ein Datum in Deutschland orientiert wird. Daneben haben Konsumgüter eine Menge, die durch die Assoziation auf die Klasse vom Typ `UnitOfMeasure` repräsentiert wird. Als konkrete Maßeinheit können Gewicht, Volumen oder Stückzahl gewählt werden und werden durch die gleichnamigen Klassen `Weight`, `Volume` und `Quantity` repräsentiert. Neben der Maßeinheit dient die Klasse `Value` zur Repräsentation der quantitativen Menge. Die Klasse `ValueValidator` übernimmt die in Bezug auf die Domäne bezogene Überprüfung eines gültigen Werts, der später durch ein Objekt der Klasse `Value`. Bezogen auf die Regeln innerhalb der Domäne darf der Wert nicht negativ sein.

Ein Konsumgut hat zudem einen Lagerort, der durch die Assoziation zu einer Klasse des Typs `Storage` repräsentiert wird. Als mögliche Typen für Lagerorte kommen in der Domäne Kühlschränke oder Lebensmittelregale in Frage, die durch die gleichnamigen Klassen `Fridge` und `FoodShelf` dargestellt sind.

Die Interfaces `ConsumerGoodsRepository`, `FridgeRepository` sowie `FoodShelfRepository` dienen zur Implementierung von, ebenfalls in DDD definierten, Repositories in Bezug auf die Klassen `ConsumerGoods`, `Fridge` und `FoodShelf`. Repositories werden im Abschnitt 5.5 analysiert. Die Bezeichnung *-Repository* ist in diesem Fall möglich, da es sich hierbei um das DDD-Modell *Repository* handelt und dieses somit direkt ersichtlich wird.

Die Klassen `ConsumerGoodsManager` und `StorageManager` repräsentieren die Umsetzung der Business-Logik und diese entspricht dem Managen von Konsumgütern und Lagerorten.

`ConsumerGoodsToConsumerGoodsResourceMapper`, `FoodShelfToFoodShelfResourceMapper` und `FridgeToFridgeResourceMapper` sowie die Klassen `ConsumerGoodsResource`, `FridgeResource` und `FoodShelfResource` sind mit dem Zusatz *-Resource* beschrieben, da es sich hierbei um Ressourcen der Domänenklassen handelt, die zur Verarbeitung mit den äußeren Schnittstellen, wie zum Beispiel der GUI, verwendet werden.

Die Klassen `ConsumerGoodsGuiController` sowie `StorageInteractorGuiController` sind aufgrund ihrer Funktion als Schnittstelle für die Informationsübertragung an die GUI.

ConsumerGoodsRepositoryBridge, FoodShelfRepositoryBridge und FridgeRepository erfüllen die Funktion des *Bridge*-Entwurfsmusters in Bezug auf die Repositories der Domänen *ConsumerGoods*, *Fridge* und *FoodShelf* und haben dementsprechend die Bezeichnung. Bei den Interfaces PersistenceConsumerGoodsRepository, PersistenceFridgeRepository und PersistenceFoodShelfRepository wurde die entsprechende Bezeichnung gewählt, da es sich hierbei um Klassen handelt, die entsprechend dem DDD die Funktion des *Repositories* umsetzen. Die Bezeichnung *Persistence*- wurde gewählt, da die Interfaces zur Implementierung der Persistierung dienen.

## Umsetzung der Ubiquitos Language auf der grafischen Oberfläche

In der GUI wurde ebenfalls die *Ubiquitos Language* angewandt. Beim Anlegen eines Konsumguts wurde die Bezeichnung der Domäne, das Einlagern, sowie beim Löschen, das Auslagern, verwendet und in der Oberfläche beschriftet.

### Store New Consumer Good

Abbildung 5.2: Anstatt der technischen Bezeichnung, dem Speichern eines Konsumguts, wird die domänenspezifische Bezeichnung des Einlagerns verwendet.

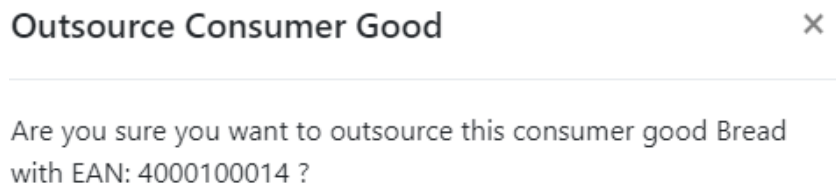


Abbildung 5.3: Beim Auslagern eines Konsumguts erhält der Anwender eine domänenbezogene Rückmeldung und wird nicht gefragt, ob er den technischen Vorgang, das Löschen des Konsumguts, durchführen möchte.

Auch die Identifikationsnummer wurde anstatt *id* der Domäne angepasst und *EAN* bezeichnet.

**EAN: 4000100012**

Abbildung 5.4: Der natürliche Schlüssel der Domäne wird zur eindeutigen Identifikation genutzt und dargestellt.

Ebenfalls wurden Buttons anstatt den technischen Eigenschaften Einfügen und Löschen passend zur Domäne beschriftet.

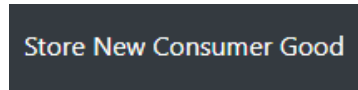


Abbildung 5.5: .



Abbildung 5.6: Die Buttons zum technischen Speichern und Abbrechen sind ebenfalls entsprechend des Domänenkontext beschrieben.



Abbildung 5.7: Die Buttons zum Bestätigen von Änderungen an einem Konsumgut sind ebenfalls entsprechend des Domänenkontext beschrieben.



Abbildung 5.8: Die Buttons zum technischen Löschen sind ebenfalls entsprechend des Domänenkontext beschrieben.

Das Anwenden der *Ubiquitous Language* auf der grafischen Oberfläche hat den allgemeinen Vorteil, dass Abläufe und Funktionen für den Anwender, der mit der Domäne vertraut ist, bei der Anwendung der Software verständlicher sind und Missverständnisse reduziert werden.

Grundsätzlich handelt es sich bei diesem Softwareentwurf um eine Create, Read, Update, Delete (CRUD)-Anwendung. Die Anwendung dient dem Verwalten von Konsumgütern an Lagerorten. Die Komplexität ist dabei relativ gering, weshalb man solche Anwendungen als *Smart UI*-Anwendungen bezeichnet. Dementsprechend wurden nur eine geringe Anzahl an DDD-Modelle angewandt, da das Einfügen zusätzlicher Modelle die Komplexität der Anwendung künstlich steigern würde.

Im Folgenden werden die angewandten DDD-Modelle *Entities*, *Value Objects*, *Aggregates* und *Repositories* aufgezeigt und in ihrer Funktion innerhalb des Softwareentwurfs analysiert.

## 5.2 Analyse angewandter Value Objects

Innerhalb des Softwareentwurfs werden *Value Objects* zur Repräsentation eines Wertes verwendet. Dazu zählen die Klassen *Day*, *Month*, *Year* sowie *Value*. Die Klassen *Day*, *Month* und *Year* dienen dabei dem Speichern des gleichnamigen Teils des Datums. Die Klasse *Value* speichert den Wert der dazugehörigen Einheit innerhalb der Klassen *Volume*, *Quantity* oder *Weight*. Die Werte sind insofern immutable, dass eine Zuweisung des Werts, neben der Initialisierung durch den Konstruktor, nicht möglich ist. Die Variablen innerhalb der Klassen zum Speichern der Methoden sind daher als *final* gekennzeichnet. Die Klassen enthalten zudem keine *setter*-Methode. Darüber hinaus enthalten die Klassen ausschließlich Methoden zur Rückgabe des Werts. Der Wert wird dabei als unveränderliche Objekte zurückgegeben, um ein Verändern des Werts zu unterbinden. Die Klassen selbst sind ebenfalls *final* deklariert, um eine Vererbung auszuschließen. Die Objekte der Klasse dienen schließlich ausschließlich der Werterepräsentation und ein Manipulieren des Wertes, auch durch Subklassen, ist auszuschließen.

Die Implementierung dieser Value Objects wurde durchgeführt, damit Fehler durch Manipulation an mehreren Stellen des Programms reduziert werden und Objekte eine klare Wertzuweisung während ihrer Laufzeit haben sollen.

Darüber hinaus lassen sich in diesem Softwareentwurf weitere *Value Objects* erkennen, die sich aus den grundlegenden *Value Objects* zusammensetzen. Eines stellt die Klasse *DayOfYear* dar. Die Klasse dient der Verwaltung der *Value Objects* *Day* sowie *Month* und repräsentiert somit den Tag eines Jahres. Auch in diesem Fall zeichnet sich die Klasse *DayOfYear* als *Value Object* aus, da es den reinen und unveränderlichen Werten eine Semantik in der Domäne gibt und keine Identität oder Lebenszyklus aufweist. Ein weiteres *Value Object* ist die Klasse *BestBeforeDate*. Die Klasse dient lediglich der Repräsentation eines Datums, in Bezug zur Domäne dem Mindesthaltbarkeitsdatums eines Produkts, und ist somit eine Möglichkeit, dem reinen Datum als Wert eine Semantik in der Domäne zu geben.

Bei allen Ref und Darstellung der Anpassung der hashCode() und equals()-Methode

Ref zu Day, final und immutable (kein set außer Konstruktor) zeigen

Ref zu Month,

Die Umsetzung von *Value Objects*, die auf *Value Objects* basieren, wurde gewählt, da eine Gruppierung für eine bessere Übersichtlichkeit und Lesbarkeit, zum Beispiel bei Übergabeparametern, des Codes sorgt.

## 5.3 Analyse angewandter Entities

Bei Befassung der Domäne, in der sich der Softwareentwurf befindet, fallen die grundlegenden Elemente der Domäne `ConsumerGoods`, `Fridge` sowie `FoodShelf` auf. Es handelt sich bei diesen Klassen entsprechend DDD um Entitäten. Die Klasse `ConsumerGoods` repräsentiert einen Gegenstand der Domäne, ein Konsumgut. Die Klassen `Fridge` und `FoodShelf` repräsentieren ebenfalls Gegenstände der Domäne, nämlich einen Kühlschrank und ein Lebensmittelregal. Eine Eigenschaft von Entitäten ist das Besitzen einer Identität in Form einer Kombination von Eigenschaften oder eines Surrogat- beziehungsweise natürlichen Schlüssels. Ein Konsumgut weist die Eigenschaft auf, dass es in der Domäne einen natürlichen Schlüssel hat. Jedes Produkt, das käuflich im Handel erwerblich ist, hat einen European Article Number (EAN). Dementsprechend hat die Klasse `ConsumerGoods` den natürlichen Schlüssel `eanCode` zur eindeutigen Identifikation implementiert. Die EAN bezeichnet die Nummer unter dem Barcode und dient als eindeutige Produktidentifizierungsnummer. Dabei ist zu beachten, dass die Codezuweisung fremdbestimmt ist und somit auch keine Garantie auf Duplikatfreiheit außerhalb des Kontextes gegeben werden kann. Zudem können Produkte, die keinen EAN haben, zum aktuellen Stand nicht in der Software verwaltet werden, da zunächst ausschließlich im Handel erwerbliche Konsumgüter verwaltet werden. Bei der *Entity* `Fridge` ist jedoch kein eindeutiger Schlüssel in der Domäne enthalten, weshalb ein Surrogatschlüssel angewandt wird. Die Klasse `Fridge` hat hierzu eine Variable `id`, deren Wert automatisch systemseitig zur Laufzeit generiert wird. Der Vorteil ist, dass der Schlüssel jederzeit generierbar ist, jedoch keinen Bezug zur Domäne hat. Bei der *Entity* `FoodShelf` ist ebenfalls kein eindeutiger Schlüssel in der Domäne enthalten, weshalb auch ein Surrogatschlüssel angewandt wird. Die Generierung läuft entsprechend gleich zu der in der Klasse `Fridge` ab.

Die Umsetzung der Entities wurde gewählt, da sie bei Betrachtung der Domäne in der Realität wichtigen Entitäten entsprechen, die für Handlungen in der Domäne benötigt werden und somit in den Programmentwurf übernommen wurden.



## 5.4 Analyse angewandter Aggregates

In diesem Softwareentwurf dienen die Klassen `ConsumerGoods`, `Fridge` und `FoodShelf` als *Aggregates*. Bei Analyse des größten *Aggregates* innerhalb des Softwareentwurfs ist zu erkennen, dass neben der *Entity* `ConsumerGoods` auch eine Implementierung des Interfaces `StorableGoods`, was in dem aktuellen Stand nur auf die Klasse `Food` zutrifft, zu diesem *Aggregate* zählen. Zudem zählen die *Value Objects* `BestBeforeDate` als auch eine Implementierung des Interfaces `UnitOfMeasure` hinzu.

Bei Betrachtung der Visualisierung des *Aggregates* ist zu erkennen, dass ein Zugriff auf die Objekte innerhalb des *Aggregates* über das Objekt der Klasse `ConsumerGoods` erfolgen. Es handelt sich somit beim Objekt der Klasse `ConsumerGoods` um ein *Aggregate Root*. Direkte Referenzen auf Elemente innerhalb des *Aggregates* sind dabei nicht erlaubt. Eine Betrachtung des UML-Diagramms zeigt, dass solch eine direkte Referenz nicht vorliegt und der Zugriff ausschließlich über den *Aggregate Root* funktioniert.

Ein Objekt der Klasse `ConsumerGoods` kann somit den gesamten Zugriff auf das *Aggregate* kontrollieren und die Einhaltung der Domänenregeln gewährleisten. Ein Beispiel ist in diesem Fall die Validierung des eingegebenen Mindesthaltbarkeitsdatums durch die Klasse `DateValidator`. Durch das *Aggregate Root* kann die Einhaltung der Domänenregel, nämlich der Angabe eines gültigen Datums, validiert und gewährleistet werden.

Das *Aggregate Root* `ConsumerGoods` ist über die ID `eanCode` eindeutig identifizierbar. Ein Verlust dieser ID zur Laufzeit führt dazu, dass neben dem Objekt der Klasse `ConsumerGoods` das gesamte *Aggregate* und somit das dazugehörige Objekt der Klasse `Food`, `BestBeforeDate` als auch das Objekt der Spezialisierung der Superklasse `UnitOfMeasure` nicht mehr erreichbar. Die Herausgabe von Referenzen auf innere Objekte werden zur Gewährleistung des den Domänenregeln entsprechenden Zustands Kopien oder Immutable-Dekorierer ausgegeben, wie im Beispiel zu erkennen.

Des Weiteren werden bei CRUD-Anwendungen auf einzelne Objekte des *Aggregates* das gesamte *Aggregate* geladen und entsprechend gespeichert, wie in `ConsumerGoodsApplicationService` zu erkennen. Das Ziel ist die Minimierung des Risikos auftretender Bugs durch ungültige Zustände aufgrund teilweiser Änderungen sowie die Einhaltung der Domänenregeln

Visualisierung  
des Ag-  
gregates

Aktuell  
durch  
Resource  
verletzt,  
entweder  
ändern  
und er-  
läutern  
oder Ver-  
letzung  
erläutern

durch das *Aggregate Root*, über das die CRUD-Anwendungen ausgeführt und an den entsprechenden internen Objekten angewandt werden.

Der Vorteil bei der Bildung dieses *Aggregates* ist, dass durch den zentralen Zugriff über das *Aggregate Root* die Domänenregeln eingehalten werden können. Des Weiteren wird durch die Einteilung in einzelne *Aggregates* Transaktionsgrenzen gebildet und übergreifende Objektbeziehungen ebenfalls durch den zentralen Zugriff entkoppelt.

Bei den *Aggregates* *Fridge* und *FoodShelf* handelt es sich um Spezialisierungen der Superklasse *Storage*. Die *Aggregates* haben zum aktuellen Stand keine Referenzen auf weitere Klassen und sind somit zugleich *Aggregate Root*, auch in Hinblick auf ihre Funktion mit möglichen Erweiterungen.

Die Umsetzung der Aggregate wurde übergreifend bedacht, damit dadurch klare Abgrenzungen und eine bessere Verwaltbarkeit vorliegen, das Bearbeiten und Verändern von Attributen oder Teilen des Aggregates zentral verwaltet und validiert werden kann und daraus resultierende Fehler minimiert werden können.

## 5.5 Analyse angewandter Repositories

Innerhalb dieses Softwareentwurfs übernehmen die Interfaces *ConsumerGoodsRepository*, *FridgeRepository* sowie *FoodShelfRepository* die Funktion des *Repositories* als DDD-Modell. Dabei ist laut DDD vorgesehen, dass *Repositories* direkt mit *Aggregates* zusammenarbeiten und somit für jedes der im Softwareentwurf definierten *Aggregates* *ConsumerGoods*, *Fridge* als auch *FoodShelf* ein *Repository*-Modell vorhanden ist.

Die Bezeichnung der Methoden ist ebenfalls an die Domäne angepasst, wie anhand eines Beispiels an der Klasse *ConsumerGoodsRepository* zu sehen ist.

Die konkrete Implementierung der Persistierung findet in den jeweils entsprechenden Klassen *ConsumerGoodsRepositoryBridge*, *FridgeRepositoryBridge* sowie *FoodShelfRepositoryBridge* statt. Dies ist sowohl für das DDD-Modell *Repository* als auch entsprechend der *Clean Architecture* entsprechend vorgesehen, dass die Implementierung außerhalb stattfindet. Schließlich zählt die Definition des *Repository* zum *Domain Code*, die konkrete Umsetzung ist jedoch domänenunabhängig oder auch *Pure Fabrication* genannt.

Die Implementierung von Repositories wurde gewählt, damit neben der Zusammenarbeit zwischen den *Aggregates* `ConsumerGoods`, `Fridge` als auch `FoodShelf` sowohl ein *Repository*-Modell vorhanden ist als auch zugleich eine zentrale Verwaltungsmöglichkeit.

## 6 Unit Tests

Während eines Entwicklungsprozesses einer Software entstehen Fehler. Fehler kosten jedoch unterschiedliche Ressourcen und steigen mit der Zeit, seit der Fehler existiert. Daher sind Tests ein wichtiges Mittel in der Softwareentwicklung.

Hierfür gibt es folgende Testklassifikationen:

- *Akzeptanztests*,
- *Integrationstests*,
- *Komponententests* sowie
- *Leistungstests*.

In diesem Softwareentwurf werden ausschließlich *Unit Tests* betrachtet. *Unit Tests* zählen zu den *Komponententests* und starten nur den relevanten Teil des Systems. Weitere nötige Systemteile werden durch Stellvertreter, sogenannten *Mock-Ups*, ersetzt. *Mock-Ups* sind einfache Stellvertreter, in Form einer Minimalumsetzung der zum Testen nötigen Funktionalität, für in der späteren Laufzeit der Umgebung „echte“ Objekte. Die Umsetzung der Funktionalitäten wird auch als *Fakes* bezeichnet. Der Vorteil durch den Einsatz von *Mock-Ups* ist, dass Abhängigkeiten während eines Tests ersetzt werden können und somit eine isolierte Betrachtung der Klassen möglich ist. Die Durchführung findet mittels Testframeworks, wie beispielsweise im Java-Umfeld JUnit, statt. *Unit Tests* haben das Ziel, die korrekte Implementierung der Komponente sicherzustellen.

Der Aufbau eines Unit-Tests orientiert sich an der AAA-Regel und steht für:

- *Arrange*,
- *Act* und
- *Assert*.

*Arrange* bezeichnet das Initialisieren der Test-Umgebung. *Act* beschreibt den Teil, der für das Ausführen der zu testenden Applikation nötig ist. *Assert* bezeichnet den Teil, der für das Prüfen der Test-Zusicherung nötig ist.

Mögliche Testergebnisse können wie folgt sein:

- *Success*,
- *Failure* und
- *Error*.

*Success* bedeutet, dass die Testmethode durchläuft und alle Assertions erfüllt sind. *Failure* bedeutet, dass der Test aufgrund einer oder mehrerer Assertions nicht bestanden wurde. *Error* bedeutet, dass der Test aufgrund eines Fehlers nicht bestanden wurde.

## 6.1 ATRIP

Bei *Unit Tests* orientiert man sich zudem an den folgenden *ATRIP-Regeln*:

- *Automatic*,
- *Thorough*,
- *Repeatable*,
- *Independent* und
- *Professional*.

*Automatic* bedeutet, dass die Tests selbstständig ablaufen und Ergebnisse selbst prüfen müssen. *Thorough* besagt, dass gute Tests alle missionskritischen Funktionalitäten testet. *Repeatable* legt fest, dass ein Test jederzeit durchführbar sein soll und immer das gleiche Ergebnis liefert. *Independent* bedeutet, dass jeder Test genau ein Aspekt der Komponente testet. Somit müssen die Tests gewährleisten, dass sie in beliebiger Zusammenstellung und Reihenfolge funktionsfähig sind. *Professional* besagt, dass Testcode auch produktionsrelevanter Code ist und somit leicht verständlich sein sollte.

## 6.2 Testabdeckung

Eine Möglichkeit zur Messung der Testabdeckung ist die *Code Coverage*. Eine Variante ist das Messen der *Line Coverage*, indem die Anzahl der abgedeckten Code-Zeilen bestimmt

wird. Eine weitere Variante ist das Messen der *Branch Coverage*, indem die Anzahl der abgedeckten Pfade im Code bestimmt werden.

## 6.3 Testen mit Mocks

Das Testen mit Mocks erweitert die AAA-Regel beim Durchführen von *Unit Tests*. Der Test wird zu Beginn um die Phasen *Capture* und am Ende um *Verify* erweitert. In *Capture* wird der Mock Initialisiert, dann folgen die bekannten Regeln *Arrange*, *Act* und *Assert* aus den Unit-Tests und am Ende wird in *Verify* die Erwartungen überprüft. Weitere Voraussetzungen für das Testen mit Mocks ist, dass ein Einsatz nur sinnvoll ist, wenn eine Dependency Injection möglich ist. Des Weiteren sind statische Methoden und vergleichbare Konstrukte schwierig zu ersetzen. Zudem sind tiefe Abhängigkeitsstrukturen nur aufwendig nachzubilden. Hierbei sollte sich bei der Entwicklung an das *Law of Demeter* gehalten werden.

## 6.4 Anwendung im Softwareentwurf

Bei den Unit-Tests wurde ein besonderer Fokus auf die Validierungs-Klassen in der *Abstractions*-Schicht sowie das Erzeugen und Ändern einzelner Attribute eines *ConsumerGoods* gelegt. Die Testklassen sind innerhalb des Projekts *0-cleanproject-plugins* im Verzeichnis *src.test.java* gelistet. Bei der Erstellung der Tests wurde eine Einteilung entsprechend *Arrange*, *Act* sowie *Assert* vorgenommen und die Bereiche im Code entsprechend mit Co-dekommentaren gekennzeichnet. Die Testklasse *DayDateValidationTest* überprüft die Funktionalität der Klasse *DayValidator*, indem eine Validierung eines gültigen als auch eines negativen oder eines über 31 liegenden Wertes getestet wird. Die Testklasse *MonthDateValidationTest* überprüft die Funktion der Klasse *MonthValidator*, indem eine Validierung eines gültigen als auch eines negativen oder eines über 12 liegenden Wertes getestet wird. Die Testklasse *YearDateValidationTest* überprüft die Funktion der Klasse *YearValidator*, indem eine Validierung eines gültigen als auch eines negativen Wertes getestet wird. Die Testklasse *DateValidationTest* überprüft die Funktion der Klasse *DateValidator*, indem eine Validierung eines gültigen als auch eines ungültigen Datums im Februar und an einem Monat mit 30 statt 31 Tagen

gestestet wird. Die Tests liegen sehr nahe beieinander, jedoch wurde eine Aufteilung aufgrund der Verständlichkeit (entsprechend der *Professional*-Regel) vorgenommen. Die Testklasse `UnitOfMeasureValueValidationTest` überprüft die Funktion der Klasse `ValueValidator`, indem eine Validierung eines gültigen als auch eines negativen Wertes getestet wird.

Die Testklasse `StoreConsumerGoodsTest` überprüft die Klasse `ConsumerGoodsBuilder` sowie dessen Validierungsmethode, indem unterschiedliche Kombinationen mit invaliden Übergabeparametern getestet werden.

Die Testklasse `ConsumerGoodsGuiControllerTest` dient der Überprüfung der HTTP-Statuscodes für erfolgreiche als auch fehlerhafte Anfragen zum Erhalt, der Erzeugung, dem Aktualisieren oder Löschen von `ConsumerGoods`.

Der Schwerpunkt der genannten Klassen, mit Ausnahme von `ConsumerGoodsGuiControllerTest`, liegt auf der Wertvalidierung als Grundlage zur Vermeidung von auftretenden Fehlern, die zu einem Fehlverhalten der Software führen können, das unter Umständen nicht direkt bemerkt wird oder auch durch den Tausch verschiedener Peripherie auftreten könnte. Neben der *Professional*-Regel wurde auf die weiteren ATRIP-Regeln geachtet.

- Die Tests laufen selbstständig und überprüfen ihr Ergebnis selbst (*Automatic*),
- als Aufgabe des Verwaltens von Konsumgüter testen Sie die Grundlage der ordnungsgemäßen Werteverwaltung der Konsumgüter (*Thorough*),
- die Tests sind jederzeit durchführbar und das Ergebnis reproduzierbar (*Repeatable*),
- die Tests sind unabhängig zueinander und testen jeweils eine Komponente *Independent*.

Als Beispiel für die Erfüllung der *Automatic*-Regel dient das Nutzen von Assertions wie in `DateValidatorTest` zu erkennen. Dieses Mittel wurde auch bei den anderen Tests angewandt, indem das erwartete Ergebnis fest im *Assert*-Teil des Tests programmiert ist und der Test daraufhin die Erfüllung selbst überprüft. Es werden auch keine Informationen zur Laufzeit über den Nutzer abgefragt, um die *Atomic*-Regel zu erfüllen. Die *Thorough*-Regel wurde insofern eingehalten, dass der Schwerpunkt der Tests auf den Validierungsklassen, wie dem `DateValidationTest` oder dem `UnitOfMeasureValueValidationTest` gelegt wurde. Gleiches gilt für `StoreConsumerGoodsTest`, indem der Schwerpunkt auf die Validierung der Attribute zum Erzeugen eines Konsumguts

liegt. Dadurch können Fehler, die aufgrund ungültiger Wertangaben, wie zum Beispiel negativen Werten oder einem falschen Datum auftreten können und für Bugs oder darauf aufbauenden Fehlern führen können, vermieden werden. Ein weiteres Beispiel für das Einhalten der *Repeatable*-Regel ist ebenfalls im *Arrange*-Teil des Tests `DateValidationTest` dargestellt. Zur Erfüllung dieser Regel wurden zu prüfende Daten statisch festgelegt und werden nicht zur Laufzeit, wie beispielsweise die Wahl des aktuellen Datum, bestimmt. Die *Independent*-Regel wurde eingehalten. Dazu wurde darauf geachtet, dass jede Testklasse nur eine Komponente testet. Das wurde erfüllt, indem jede Testklasse eine der implementierten Validierungsklassen testet. Durch die Entkopplung zueinander können die Tests unabhängig und in beliebiger Reihenfolge ausgeführt werden. Alle nötigen Informationen zum unabhängigen Durchführen des Tests sind im *Assert*-Teil des Tests definiert, wie in `DateValidationTest` beispielhaft dargestellt. Zur Erfüllung der *Professional*-Regel wurde sich an der *Ubiquitous Language* aus DDD orientiert. Ein Beispiel hierzu ist die Bezeichnung der Testklassen, indem domänenspezifische Titel wie `DateValidationTest` oder `UnitOfMeasureValueValidationTest` genutzt wurden. Bei den Testmethoden wurde ebenfalls eine klare Bezeichnung gewählt, die das darauf folgende Testergebnis verständlich macht. Ein Beispiel hierfür ist in `DateValidationTest`, indem die Methoden wie beispielsweise `checkValidatorForInvalidFebruaryDate()` oder `checkValidatorForFebruaryDateInLeapYear()` den gezielten Test beschreiben.

## Unit-Tests mit Einsatz von Mocks

Zuzüglich zu den erläuterten Unit Tests testet die Testklasse `UpdateConsumerGoodsTest` das Ändern von Attributen einer Instanz der Klasse `ConsumerGoods` mit Hilfe eines Mocking-Werkzeugs. Der Test bildet den Ablauf der Methode `updateConsumerGoods()` in der Klasse `ConsumerGoodsManager` ab. Dabei wird eine im Test erzeugte Instanz der Klasse `ConsumerGoods` mit den Attributen eines neuen, im Test gemockten, Objekts der Klasse `ConsumerGoods` aktualisiert.

Zu einem älteren Entwicklungsstand wurde in der Testklasse `StoreConsumerGoodsTest` ebenfalls Mocks zum Testen angewandt, da die Überprüfung des EAN-Codes zu dem Zeitpunkt in `ConsumerGoodsBuilder` stattfand und somit eine Instanz des `ConsumerGoodsRepository` übergeben werden musste.



Auch die Unit-Tests mit dem Einsatz von Mocks erfüllen die *ATRIP*-Regeln und erzielen das Testergebnis *Success*.

Als Beispiel für die Erfüllung der *Atomic*-Regel dient ebenfalls der Einsatz von Assertions und das statische Programmieren von Ergebnissen, wie in `UpdateConsumerGoodsTest` dargestellt. Auch bei diesen Tests werden alle Informationen im *Arrange*-Teil des Tests hinterlegt und somit keine Abfragen während der Laufzeit des Tests abgefragt. Die *Thorough*-Regel wurde an diesem Beispiel eingehalten, indem die Tests die Grundfunktionalitäten der Anwendung, wie durch den Test `UpdateConsumerGoodsTest` das Aktualisieren eines Konsumguts, mit `StoreConsumerGoodsTest` das Einlagern eines Konsumguts und mit `ConsumerGoodsGuiControllerTest` die Interaktionsschnittstelle zur GUI testen. Ein weiteres Beispiel für das Einhalten der *Repeatable*-Regel ist ebenfalls im *Arrange*-Teil des Tests `ConsumerGoodsGuiControllerTest` dargestellt. Zur Erfüllung dieser Regel wurden zu prüfende Daten statisch festgelegt und werden nicht zur Laufzeit, wie beispielsweise die Wahl die Zusammenstellung der URI zur Kommunikation mit der Schnittstelle über HTTP, bestimmt. Die *Independent*-Regel wurde eingehalten. Es wurde darauf geachtet, dass jede Testklasse ebenfalls nur eine Komponente testet. Ein Beispiel ist die Testklasse `StoreConsumerGoodsTest`, welche die Validierung der Attribute zum Anlegen eines Konsumguts testet. Durch die Entkopplung zueinander können die Tests ebenfalls unabhängig und in beliebiger Reihenfolge ausgeführt werden. Alle nötigen Informationen zum unabhängigen Durchführen des Tests sind im *Assert*-Teil des Tests definiert, wie in `ConsumerGoodsGuiControllerTest` beispielhaft dargestellt. Zur Erfüllung der *Professional*-Regel wurde sich an der *Ubiquitous Language* aus DDD orientiert. Ein Beispiel hierzu ist die Bezeichnung der Testklassen, indem anstatt der technischen Begriffe *AddConsumerGood* domänenspezifische Titel wie *StoreConsumerGoods* genutzt wurden. Bei den Testmethoden wurde ebenfalls eine klare Bezeichnung gewählt, die das darauf folgende Testergebnis verständlich macht. Ein Beispiel hierzu findet sich in `StoreConsumerGoodsTest`, indem die Methoden `checkStoreValidConsumerGood()` oder `checkStoreConsumerGoodWithoutFoodDescription()` die domänenspezifische Bezeichnung verwenden und somit für Domänenexperten verständlich wird, was der Test überprüft, ohne den Testablauf im Detail zu verstehen.

## Code Coverage

Die Code Coverage-Ergebnisse nach Durchführen der Unit-Tests sind in Tabelle 6.1 dargestellt. Bei Betrachtung der Testabdeckung ist zu erkennen, dass der Schwerpunkt der Tests auf den Klassen in der *Domain*- sowie in der *Abstraction*-Schicht auf den Validierungsklassen sowie der Erzeugung eines Konsumguts durch den `ConsumerGoodsBuilder` befindet. Das entspricht auch den Testbereichen der Testklassen `DateValidationTest`, `DayDateValidationTest`, `MonthDateValidationTest`, `YearDateValidationTest` sowie `UnitOfMeasureValueValidationTest`, die den dazugehörigen Validator testen.

Die Code Coverage verdeutlicht, dass der Fokus auf die Validierungsklassen im Package **de.dhbw.cip.domain** sowie den dazugehörigen domänenspezifischen Klassen in dem sowie im Package **de.dhbw.cip.abstractioncode** liegt. Hinzu kommt das Testen der Attributvalidierung des `ConsumerGoodsBuilder`. Da eine Persistierung noch nicht vorgesehen ist, sind die Klassen im Package **de.dhbw.cip.application** ebenfalls noch nicht in Tests berücksichtigt. Eine Ausnahme stellt die Testklasse `ConsumerGoodsGuiControllerTest`, diese überprüft die HTTP-Schnittstelle und die entsprechenden Rückmeldungen des Servers an die GUI. Hierzu muss der Service jedoch in Betrieb sein und somit ist zu beachten, dass die getestete Code Coverage nicht zur Testinstanz zählt. Gleiches gilt für die Klassen im Package **de.dhbw.cip.adapters**, schließlich handelt es sich hierbei um Klassen zum Trennen der inneren domänenspezifischen Klassen in äußere, für die Schnittstellen relevante Klassen.

Paket-/Klassenname	Code Coverage
<b>de.dhbw.cip</b>	0.0%
ConsumerInventoryPlannerApplication	0.0%
<b>de.dhbw.cip.plugin.rest</b>	0.0%
ConsumerGoodsGuiController	0.0%
StorageGuiController	0.0%
<b>de.dhbw.cip.plugins.persistence.hibernate</b>	0.0%
ConsumerGoodsRepositoryBridge	0.0%
FridgeRepositoryBridge	0.0%
FoodShelfRepositoryBridge	0.0%
PersistenceConsumerGoodsRepository	0.0%
PersistenceFridgeRepository	0.0%
PersistenceFoodShelfRepository	0.0%
<b>de.dhbw.cip.adapters</b>	0.0%
ConsumerGoodsToConsumerGoodsResourceMapper.java	0.0%
ConsumerGoodsResource.java	0.0%
BestBeforeDateResource.java	0.0%
FoodResource.java	0.0%
FoodShelfToFoodShelfResourceMapper.java	0.0%
FridgeToFridgeResourceMapper.java	0.0%
StorageResource.java	0.0%
FoodShelfResource.java	0.0%
FridgeResource.java	0.0%
<b>de.dhbw.cip.application</b>	0.0%
ConsumerGoodsManager.java	0.0%
StorageManager.java	0.0%
<b>de.dhbw.cip.domain</b>	63.5%
ConsumerGoods.java	68.3%
BestBeforeDate.java	48.8%
Food.java	36.6%
Storage.java	46.2%
FoodShelf.java	0.0%
Fridge.java	50.0%
DateValidator.java	71.4%
DayValidator.java	75.0%
MonthValidator.java	75.0%
ValueValidator.java	66.7%
YearValidator.java	66.7%
<b>de.dhbw.cip.abstractioncode</b>	57.4%
DayOfYear.java	53.1%
Volume.java	50.0%
Weight.java	50.0%
DayOfYear.java	53.1%
Quantity.java	61.5%
Day.java	60.0%
Month.java	60.0%
Value.java	60.0%
Year.java	60.0%
UnitOfMeasure.java	100.0%

## 7 Entwurfsmuster

Innerhalb dieses Kapitels werden die Umsetzung von *Entwurfsmustern* innerhalb des Softwareentwurfs analysiert und deren Verwendung begründet.

### 7.1 Angewandte Entwurfsmuster

Bei Betrachtung der Ausschnitte UML-Diagramms ist der *Builder* 7.2 sowie die *Bridge* 7.1 als Entwurfsmuster zu erkennen, die in diesem Softwareentwurf angewandt wurden.

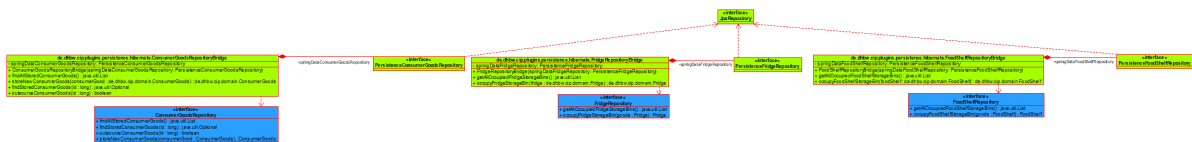


Abbildung 7.1: Der Ausschnitt des UML-Diagramms verdeutlicht die Verwendung des *Bridge*-Entwurfsmusters.

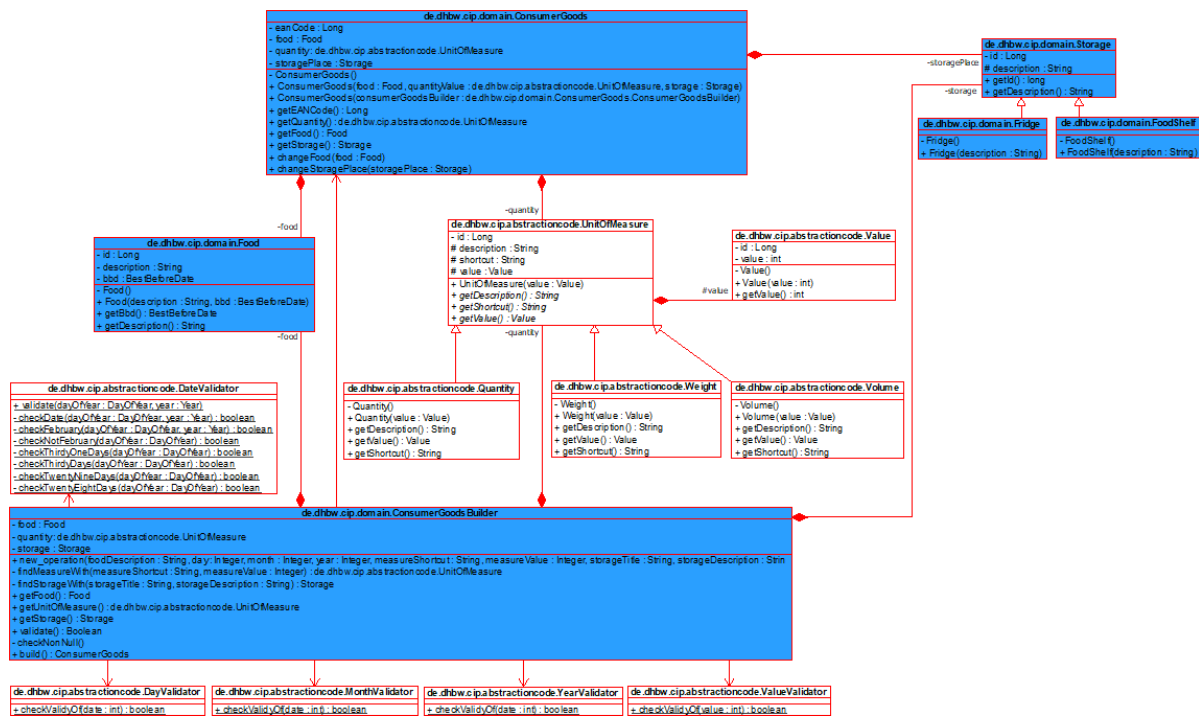


Abbildung 7.2: Der Ausschnitt des UML-Diagramms verdeutlicht die Verwendung des *Builder*-Entwurfsmusters.

## Builder-Entwurfsmuster

Das Builder-Entwurfsmuster zählt zu den *Erzeugungsmustern* und ist in der Klasse `ConsumerGoodsBuilder` umgesetzt. An dieser Stelle wurde ein Builder-Entwurfsmuster eingesetzt, da es die Möglichkeit gibt, nötige Attribute zu sammeln und, im Gegensatz zum direkten Erzeugen eines Objekts, die Möglichkeit der Überprüfung auf Gültigkeit der Attribute vor dem Erzeugen des Objekts ermöglicht. Der Einsatz ist in diesem Fall sinnvoll, da neben dem Abfangen von Fehlern bei der fehlenden Übertragung von Attributen, Datumseingaben oder Werteeingaben auf Gültigkeit überprüfen zu sind.

Das Bridge-Entwurfsmuster zählt zu den *Strukturmustern* und ist in den Klassen `ConsumerGoodsRepositoryBridge`, `FridgeRepositoryBridge` sowie `FoodShelfRepository` umgesetzt. Das Bridge-Entwurfsmuster wurde eingesetzt, da es die Trennung der Domänenlogik von der Pluginlogik ermöglicht. Während die Repository-Interfaces `ConsumerGoodsRepository`, `FridgeRepository` und `FoodShelfRepository` zur Domäne zählen, findet die Persistierung der Objekte über ein Plugin mit der Implementierung der Interfaces

PersistenceConsumerGoodsRepository, PersistenceFridgeRepository sowie PersistenceFoodShelfRepository statt. Durch Anwenden des Bridge-Entwurfsmuster ist es nun möglich, auf einen implementierten Typ des entsprechenden Repository-Interfaces für die Interaktion mit der Entitätsverwaltung zuzugreifen. Das Bridge-Entwurfsmuster hat das entsprechende Repository-Interface implementiert und übernimmt die Kommunikation mit dem Persistierungs-Plugin. Somit ist die Aufteilung entsprechend der *Clean Architecture* gewährleistet und bei einem Austausch des Persistierungs-Plugins bedarf nur Änderungen in der *Plugin*-Schicht, während das Repository-Interface in der *Domänen*-Schicht sowie alle darauf zugreifenden Instanzen davon unberührt sind.

## **8 Refactoring**