

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 10. April 2022

Merkel, Lucas

Inhaltsverzeichnis

Abkürzungsverzeichnis	III
Abbildungsverzeichnis	IV
Tabellenverzeichnis	V
Quellcodeverzeichnis	VI
1 Bedienungsanleitung der Anwendung	1
1.1 Aufrufen der Anwendung	1
1.2 Konsumgut anlegen	2
1.3 Daten eines Konsumsgut ändern	4
1.4 Konsumgut löschen	4
2 Clean Architecture	6
2.1 Planung der Schichten	6
2.2 Anwendung der Schichten	8
3 Programming Principles	14
3.1 Zu betrachtende Programming Principles	14
3.2 Analyse und Begründung	18
4 Domain Driven Design	23
4.1 Analyse und Begründung der <i>Ubiquitous Language</i>	24
5 Unit Tests	30
5.1 Anwendung im Softwareentwurf	32
6 Entwurfsmuster	35
6.1 Angewandte Entwurfsmuster	35
7 Refactoring	37

Abkürzungsverzeichnis

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DDD	Domain Driven Design
DRY	Don't Repeat Yourself
EAN	European Article Number
GRASP	General Responsibility Assignment Software Patterns/Principles
GUI	Graphical User Interface
HTTP	Hypertext Transport Transfer Protocol
JPA	Java Persistence API
KISS	Keep it Simple, Stupid
REST	Representational State Transfer
YAGNI	You Ain't Gonna Need it

Abbildungsverzeichnis

1.3	3
1.4	3
1.5	4
1.6	5
2.1	Aufteilung der Projektstruktur entsprechend <i>Clean Architecture</i>	8
2.2	Ausschnitt aus der pom.xml zur Projektstruktur.	9
4.1	Wordcloud in Bezug zu Domain Driven Design.	25

Tabellenverzeichnis

5.1	Code Coverage des gesamten Projekts.	34
-----	--	----

Quellcodeverzeichnis

1 Bedienungsanleitung der Anwendung

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

1.1 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 1.1 über die Adresse `http://localhost:4200` erreichbar.

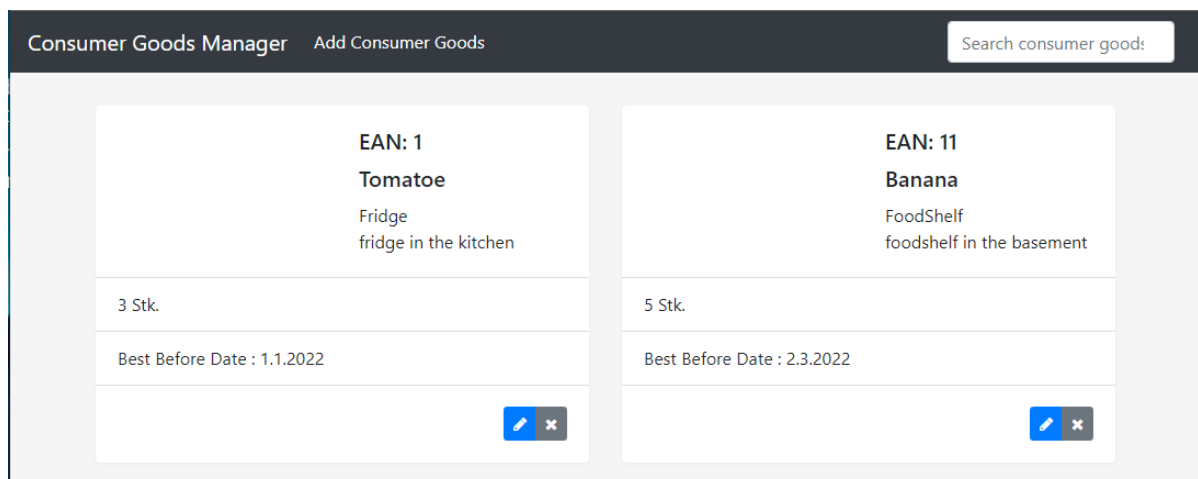
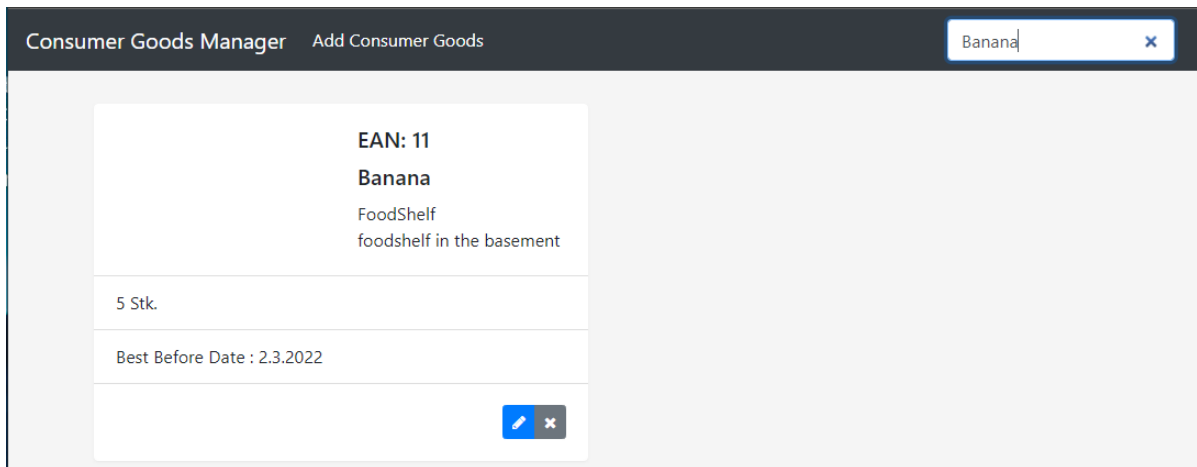


Abbildung 1.1

Auf der initialen Darstellung werden bereits verwaltete Konsumgüter dargestellt. Über das Eingabefeld an der rechten oberen Seite können Konsumgüter entsprechend ihrer Attribute gesucht werden.

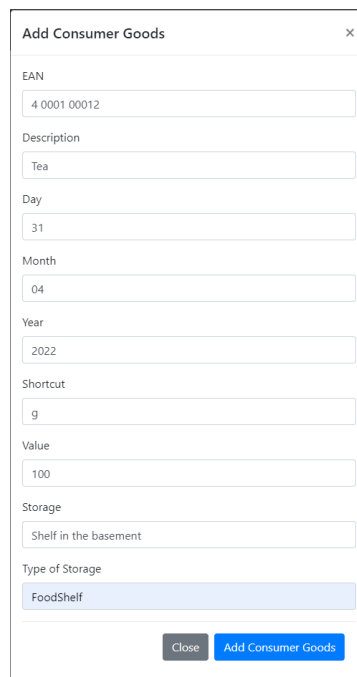


The screenshot shows a web application titled "Consumer Goods Manager" with a sub-header "Add Consumer Goods". A search bar in the top right contains the text "Banana". The main content area displays a form for a new consumer good. The form is divided into sections: the top section contains "EAN: 11", "Banana", "FoodShelf", and "foodshelf in the basement"; the middle section contains "5 Stk."; the bottom section contains "Best Before Date : 2.3.2022". At the bottom right of the form is a blue button with a pencil icon and a grey button with an 'x' icon.

Abbildung 1.2

1.2 Konsumgut anlegen

Das Anlegen eines Konsumguts ist über den Button *Add Consumer Goods* möglich. Darauf öffnet sich ein Fenster wie in Abbildung 1.3 zur Eingabe der Daten des neuanzulegenden Konsumguts. Nach Eingabe kann das Konsumgut über den Button *Add Consumer Goods* angelegt werden.



The screenshot shows a dialog box titled "Add Consumer Goods" with a close button (X) in the top right corner. The dialog contains several input fields and a dropdown menu:

- EAN:** 4 0001 00012
- Description:** Tea
- Day:** 31
- Month:** 04
- Year:** 2022
- Shortcut:** g
- Value:** 100
- Storage:** Shelf in the basement
- Type of Storage:** FoodShelf (highlighted in blue)

At the bottom of the dialog, there are two buttons: "Close" (grey) and "Add Consumer Goods" (blue).

Abbildung 1.3

Nachdem das Konsumgut angelegt ist, wird es ebenfalls auf der Oberfläche entsprechend Abbildung 1.4 dargestellt.



The screenshot shows a card-like display for the consumer good "Tea". The information is organized as follows:

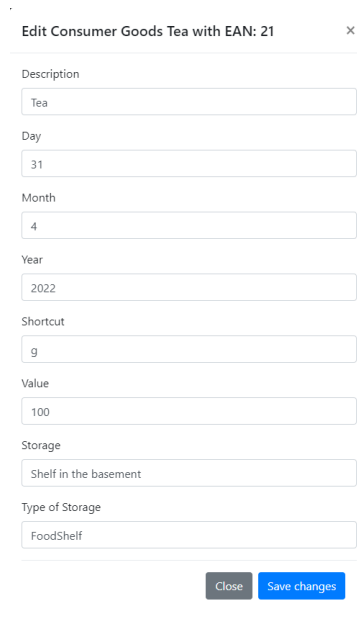
- Top section:** EAN: 21, Tea, FoodShelf, Shelf in the basement
- Value section:** 100 g
- Best Before Date section:** Best Before Date : 31.4.2022
- Bottom section:** Two buttons: a blue button with a pencil icon (edit) and a grey button with an X icon (delete).

Abbildung 1.4: .

Über den blauen Button können die Attribute des Konsumguts geändert werden. Das Vorgehen wird in Abschnitt 1.3 beschrieben. Über den grauen Button kann das Konsumgut gelöscht werden. Das Vorgehen wird in Abschnitt 1.4 beschrieben.

1.3 Daten eines Konsumsgut ändern

Zum Ändern von Attributen eines Konsumguts muss der blaue Button des gewünschten Konsumguts ausgewählt werden. Daraufhin öffnet sich eine Eingabemaske wie in Abbildung 1.5 gezeigt, in der die aktuellen Werte der Attribute enthalten sind. Die Werte können entsprechend angepasst und durch Auswahl des Buttons *Save changes* bestätigt werden.



The image shows a dialog box titled "Edit Consumer Goods Tea with EAN: 21". It contains several input fields with the following values: Description: Tea, Day: 31, Month: 4, Year: 2022, Shortcut: g, Value: 100, Storage: Shelf in the basement, and Type of Storage: FoodShelf. At the bottom right, there are two buttons: "Close" (grey) and "Save changes" (blue).

Abbildung 1.5: .

1.4 Konsumgut löschen

Das Löschen des Konsumguts erfolgt über den grauen Button. Bei Auswahl öffnet sich ein Dialogfenster entsprechend Abbildung 1.6, dass den Nutzer noch einmal auffordert, den Löschvorgang zu bestätigen, um ein unbeabsichtigtes Löschen zu vermeiden.

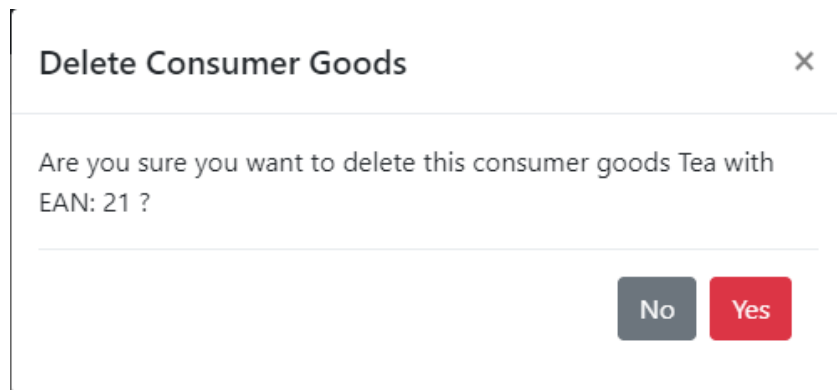


Abbildung 1.6: .

2 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.

2.1 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 Technologeaustausch, 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.

2.2 Anwendung der Schichten

Im Folgenden wird die Umsetzung der *Clean Architecture* innerhalb dieses Softwareentwurfs beschrieben. Zunächst ist die Projektstruktur entsprechend realisiert worden.

Die Projektstruktur ist ebenfalls angepasst. Anstatt alle nötigen Klassen in einem Projekt zu erzeugen, ist zur Übersichtlichkeit eine Strukturierung entsprechend der Schichten vorgenommen worden. Dabei ist keine Strukturierung der einzelnen Schichten 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 2.1 zeigt die umgesetzte Projektstruktur und Abbildung 2.2 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.

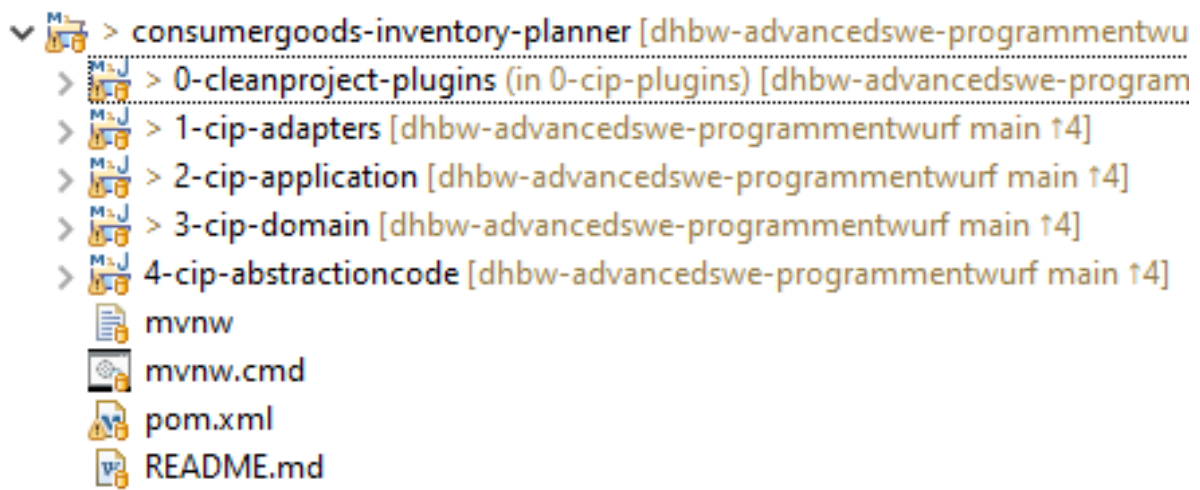


Abbildung 2.1: Die Projektstruktur ist entsprechend der *Clean Architecture* in einzelne Unterprojekte aufgeteilt.

```
<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 2.2: 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 UML-Diagramm in [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.

Perma-
Link
UML

Die Software dient als Representational State Transfer (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*. Innerhalb dieser Klassen wird die zugehörige Maßeinheit sowie der entsprechende Wert gespeichert. In dieser Schicht befindet sich zudem die Klasse *DateValidator*. Die Klasse dient der Prüfung der Gültigkeit des Mindesthaltbarkeitsdatums als ein grundsätzlich gültiges Format. Sowohl die Klassen für die Maßeinheit als auch die Klasse zur Gültigkeitsprüfung des Mindesthaltbarkeitsdatums sind in dieser Schicht festgelegt, da sowohl die Maßeinheiten als auch die Gültigkeit des Datums auf domänenübergreifendes Wissen zurückzuführen sind, 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 lediglich die Beschreibung des Lagerorts. Eine weitere Klasse ist *Food*. Diese 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.

Gefrierschrank
nicht
vergessen

Codeausschnitt
eines
Interfaces als
Beispiel

Application Code-Schicht

Eine weitere Schicht ist die *Application Code*-Schicht. Darin befinden sich die Klassen *ConsumerGoodsApplicationService* sowie *StorageApplicationService*. In diesen Klassen findet die Applikationslogik statt. Innerhalb des *ConsumerGoodsApplicationService* 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. Es findet hierbei eine *Dependency Inversion* statt. Die Klasse *ConsumerGoodsRepositoryBridge* 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. Sollte nun die Klasse *ConsumerGoodsRepositoryBridge* in der *Plugin*-Schicht aufgrund eines Technologiewechsels verändert werden, kann bei Implementierung des Interfaces *ConsumerGoodsRepository* unberührt weiterverwendet werden.

Code-
Ausschnitt

Die Klasse *StorageApplicationService* hat die gleiche Applikationslogik wie der *ConsumerGoodsApplicationService*, 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*-Objekts oder *FoodShelf*-Objekts möglich. Auch in der Klasse *StorageApplicationService* 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.

Code-
Ausschnitt
zu
Depen-
dency
Inversion

Gefriertru

Code-
Ausschnitt

Code-
Ausschnitt
zu
Depen-
dency
Inversion

eventuell
Render-
Modell

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 folgenden Klassen:

Durchgeführt wird diese Trennung der inneren und äußeren Schichten durch die Klassen *ConsumerGoodsToConsumerGoodsRessourceMapper*, *FridgeToFridgeRessourceMapper* sowie *FoodShelfToFoodShelfMapper*. Die Abläufe sind sehr identisch und werden beispielhaft durch einen Ausschnitt der Klasse *ConsumerGoodsToConsumerGoodsRessourceMapper* dargestellt. Der Code-Ausschnitt verdeutlicht die Trennung durch die Erstellung neuer Resource-Objekte entsprechend den Daten der Domänen-Objekte. Die dadurch erzeugten Resource-Objekte können nun von Objekten der Klassen in der *Plugin*-Schicht verwendet werden.

Tabelle
mit Ge-
genüber-
stellun-
gen

Code-
Ausschnitt

Plugin-Schicht

Die äußerste Schicht bildet die *Plugin*-Schicht. 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. 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 dem durch das Spring Framework bereitgestellte Java Persistence API (JPA)-Repository, dass 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

Code-
Ausschnitt
der
Main-
Methode

entspricht zudem einem Entwurfsmuster und wird im Kapitel 6 ebenfalls analysiert. Des Weiteren sind in dem Package die Klassen *SpringDataConsumerGoodsRepository*, *SpringDataFridgeRepository* sowie *SpringDataFoodShelfRepository* 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. Zuletzt findet sich in der *Plugin*-Schicht das Package *de.dhbw.cip.plugins.rest*. Darin befinden sich die Klassen *ConsumerGoodsController* sowie *StorageController*. 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 *ConsumerGoodsController*.

Frameworks

Code-
Ausschnitt
Get-
Stream

Code-
Ausschnitt
put,
post,
delete

Bezug zu
Render-
Model

Besonderh
durch
Verwen-
dung
von
Spring

3 Programming Principles

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

3.1 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.

3.2 Analyse und Begründung

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

3.2.1 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.

Zeigen
anhand
von kla-
ren Bei-
spiele,
was gut
ist und
was nicht

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 `ConsumerGoodsInteraction` 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 `ConsumerGoodsGuiInterface` statt. Gleiches gilt für die Klasse `StorageGuiInterface`. 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.

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 möglich.

Eine Verletzung des *Open-Closed-Principle* stellen die Klassen `ConsumerGoodsGuiInterface` und `StorageGuiInterface` dar. Eine Erweiterung einer Anwendung um ein weiteres zu verwaltendes 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.

Liskov Substitution Principle

Die Analyse des Liskov Substitution Principles ist nur in einem älteren Stand des Softwareentwurfs zu analysieren, da zum aktuellen Stand keine Vererbung implementiert ist. Bei Betrachtung des Standes entsprechend 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 überschrei-

oder alte
Versi-
on von
`UnitOf-
Measure`
und `Sto-
rage`

aktuell
keine
Klassen,
eventu-
ell nur
durch

ben 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.

Interface Segregation Principle

Ein Positivbeispiel der Umsetzung des *Interface Segregation Principle* findet sich _____

[ausführen](#)

Als Negativbeispiel kann das Interface betrachtet werden. _____

[ausführen](#)

Dependency Inversion Principle

Das *Dependency Inversion Principle* wurde bei den Klassen ConsumerGoods sowie Food und dem Interface StorableGood umgesetzt. Die Klasse ConsumerGoods referenziert dabei auf ein Objekt des Typs StorableGood. Das hat 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. Gleiches gilt für die Interfaces Storage und UnitOfMeasure in Bezug auf die Klasse ConsumerGoods. Auch hierbei können neue Lagermöglichkeiten oder Maßeinheiten hinzugefügt werden, ohne dass die Klasse ConsumerGoods dafür angepasst werden muss.

Darüber hinaus _____

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.

Weiteres Bsp vlt. in Be-
zug auf
Repository
Klassen
bzgl.
Testbar-
keit

3.2.2 GRASP

Geringe Kopplung

Ein Beispiel zur Umsetzung einer losen Kopplung ist Eine geringe Kopplung findet sich zwischen der GUI und der Applikationsschicht der Anwendung. Die Interaktion findet über die Klasse *ConsumerGoodsGuiInterface* und *StorageGuiInterface* statt. Beide Klassen dienen als Hypertext Transport Transfer Protocol (HTTP)-Interface. Durch die Kommunikation über HTTP kennen sich die Anwendung und die GUI nicht und sind somit entkoppelt.

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. Zur Zugriff auf eine Referenz des jeweiligen implementierten Repository-Typs wird durch eine Dependency Injection ermöglicht. Somit ist hierbei keine direkte Kopplung an eine konkrete Implementierung eines Repository-Objekts vorhanden.

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*.

Low
Coupling, lose
Kopplung

Persistieren

Interfaces
statt
Vererbung

Iterable
statt
List?

Optional?

Ref

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 des implementierten Interfaces-Typs *UnitOfMeasure* repräsentiert die Menge in Form der Klassen *Weight*, *Volume* oder *Quantity*. Der Wert der Menge wird in den Klassen durch das Attribut vom Typ *Value*

High
Cohesion

Wie
„eng“
arbeiten
Methoden und
Attribute einer
Klasse zusammen
(semantische
Nähe!)

Consumer
=>
Food,

verwaltet. Das Ziel 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 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.

3.2.3 DRY

Eine Nichteinhaltung des DRY-Prinzip 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. 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.

Wissen sollte sich nur an einem Ort befinden, z.B. Window-Sizes

Positivbeis

JPA?

4 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 (die Komplexität durch Hardware, Framework, 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 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.

Ubiquitous
Language

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.

4.1 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 4.1 des geschriebenen Codes in einer Wordcloud verdeutlicht die Umsetzung der *Ubiquitous Language*.



Abbildung 4.1: Die Wordcloud visualisiert die verwendeten Bezeichnungen im Programm-entwurf in Bezug auf die *Ubiquitous Language*.

Eine Umsetzung der *Ubiquitous Language* ist die Benennung der Klassen. Die Klasse *ConsumerGoods* repräsentiert gleichnamige Konsumgüter. Dazu zählt eine Assoziation auf die Klasse vom Typ *StorableGoods*, wozu auch die Klasse *Food* zählt, 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 Klasse *DateValidator* übernimmt die entsprechende Aufgabe der Datumsvalidierung. 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. 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

DateOfYear da MHD oftmals innerhalb eines Jahres ist und bei Validierung nur im Februar das Jahr relevant ist

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 4.1 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, *StorageToStorageResourceMapper* und die Klassen *ConsumerGoodsResource* und *StorageResource* sind mit dem Zusatz *-Resource*, 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 *ConsumerGoodsGuiInterface* sowie *StorageInteractorGuiInterface* sind aufgrund ihrer Funktion als Schnittstelle für die Informationsübertragung an die GUI.

ConsumerGoodsBridge, *FoodShelfRepositoryBridge* und *FridgeRepositoryBridge* 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.

In der GUI wurde ebenfalls die *Ubiquitous Language* angewandt.

Grundsätzlich handelt es sich bei diesem Softwareentwurf um eine Create, Read, Update, Delete (CRUD)-Anwendung. Die Anwendung dient dem Verwalten 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.

Stufe
2, Stufe
2+ und
Stufe 3

Methoden,
Varia-
blen
Stufe 1

Auch
Bezug
zur GUI

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

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.

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
Darstel-
lung der
Anpas-
sung der
hashCode()
und
equals()-
Methode

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

Ref zu
Day, fi-
nal und
immuta-
ble (kein
set au-
ßer Kon-
struktor)
zeigen

Ref zu
Month,
final und
immuta-
ble (kein
set au-

natürlichen Schlüssels. das *Entity* 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). 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. Bei der *Entity Fridge* ist jedoch kein eindeutiger Schlüssel in der Domäne enthalten, weshalb ein Surrogatschlüssel angewandt. 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.

In diesem Softwareentwurf dienen die Klassen *ConsumerGoods*, *Fridge* und *FoodShelf* als *Aggregates*-Modell. 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 im Kapitel zeigt jedoch, 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 Implementierung des Interfaces *UnitOf-*

Consumer
= Na-
türlicher
Schlüssel

Was
ist mit
Produk-
ten, die
keinen
EAN
haben?

Fridge =
Surro-
gatschlüs-
sel

FoodShelf
= Surro-
gatschlüs-
sel

Aggregates

Visualisier
des Ag-
gregates

ref zu
UML

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

gleiches
bei Map-

Measure nicht mehr erreichbar. Die Herausgabe von Referenzen auf innere Objekte werden zur Gewährleistung des den Domänenregeln entsprechenden Zustands Kopien oder Immutable-Dekorierer ausgegeben, wie in diesem Beispiel zu erkennen .

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

Innerhalb dieses Softwareentwurfs übernehmen die Interfaces *ConsumerGoodsRepository*, *FridgeRepository* sowie 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.

Ref ein-
fügen

Jedes
Aggregat
bildet ei-
ne eigene
Einheit
(auch für
Create,
Read,
Update,
Delete –
CRUD)
Wird im-
mer voll-
ständig
geladen
und ge-
speichert

Ausschnitt
aus Con-
sumer-
Goods-
Applica-
tionSer-
vice

Repository

FoodShelf

Methoden
passen
zur
Domäne
entspre-
chend
den Bei-
spielen
in der

5 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*.

Test
first und
TDD er-
wähnen?

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.

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 getestet. *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.

Testabdeck

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.

Testen
mit
Mocks

Das Testen mit Mocks erweitert die AAA-Regel beim Durchführen von *Unit Tests*. Der Test wird zu Beginn um *Capture* und am Ende um *Verify* erweitert. 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

erläutern

aufwendig nachzubilden. Hierbei sollte sich bei der Entwicklung an das *Law of Demeter* gehalten werden.

5.1 Anwendung im Softwareentwurf

Bei den Unit-Tests wurde ein besonderer Fokus auf die Validierungs-Klassen in der Abstraktions-Schicht sowie das Erzeugen und Ändern einzelner Attribute eines `ConsumerGoods` gelegt. Die Testklassen sind im Projekt im Verzeichnis `Test` gelistet. Bei der Erstellung der Tests wurde eine Einteilung *Arrange*, *Act* sowie *Assert* vorgenommen. Die Testklasse `DayDateTest` 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 `MonthDateTest` 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 `YearDateTest` überprüft die Funktion der Klasse `YearValidator`, indem eine Validierung eines gültigen als auch eines negativen Wertes getestet wird. Die Testklasse `DateValidatorTest` 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 getestet wird. Die Tests liegen sehr nahe beieinander, jedoch wurde eine Aufteilung aufgrund der Verständlichkeit (*Professional*) vorgenommen. Die Testklasse `UnitOfMeasureValueTest` überprüft die Funktion der Klasse `ValueValidator`, indem eine Validierung eines gültigen als auch eines negativen Wertes getestet wird.

Die Testklasse `AddConsumerGoodsTest` 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 *Professional* 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*.

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.

Bei Betrachtung der Code Coverage-Ergebnisse in Tabelle 5.1 wird der Schwerpunkt auf das Testen der Validierungsklassen sowie dem `ConsumerGoodsBuilder` verdeutlicht.

Einsatz
von
Mocks

Consumer

REST?

ATRIP-
Regeln

in Bezug
auf den
jeweili-
gen Unit-
Test

Code Co-
verage
(mindestens 1,
mehr
gibt
Bonus)
der ge-
samten
Schich-
ten

Über
alle Unit-
Tests,
Line und
Branch

Paket-/Klassenname	Code Coverage
de.dhbw.cip	0.0%
ConsumerInventoryPlannerApplication	0.0%
de.dhbw.cip.plugin.rest	0.0%
ConsumerInventoryPlannerApplication	0.0%
ConsumerInventoryPlannerApplication	0.0%
de.dhbw.cip.plugins.persistence.hibernate	0.0%
ConsumerInventoryPlannerApplication	0.0%
ConsumerInventoryPlannerApplication	0.0%
ConsumerInventoryPlannerApplication	0.0%
de.dhbw.cip.adapters	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%
de.dhbw.cip.application	0.0%
ConsumerGoodsManager.java	0.0%
StorageManager.java	0.0%
de.dhbw.cip.domain	64.9%
ConsumerGoods.java	75.4%
BestBeforeDate.java	48.1%
Food.java	36.6%
Storage.java	30.0%
FoodShelf.java	0.0%
Fridge.java	100.0%
de.dhbw.cip.abstractioncode	59.1%
Volume.java	0.0%
Weight.java	0.0%
DateValidator.java	85.3%
DayOfYear.java	53.1%
Quantity.java	50.0%
Day.java	60.0%
Month.java	60.0%
Value.java	60.0%
Year.java	60.0%
DayValidator.java	75.0%
MonthValidator.java	75.0%
ValueValidator.java	66.7%
YearValidator.java	66.7%
UnitOfMeasure.java	100.0%

Tabelle 5.1: Code Coverage des gesamten Projekts.

6 Entwurfsmuster

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

6.1 Angewandte Entwurfsmuster

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

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-Entwurfsmusters 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

Perma-
Link
UML

Perma-
Link
UML

bedarf nur Änderungen in der *Plugin*-Schicht, während das Repository-Interface in der *Domänen*-Schicht sowie alle darauf zugreifenden Instanzen davon unberührt sind.

7 Refactoring