HAW HAMBURG

# Bachelorarbeit

Kai Meier

## Evaluierung des jMolecules Framework zur architektonischen Abstraktion im Kontext des Domain-Driven Design

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department Computer Science*

Kai Meier

# Evaluierung des jMolecules Framework zur architektonischen Abstraktion im Kontext des Domain-Driven Design

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Wirtschaftsinformatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 17. November 2022

**Kai Meier**

**Thema der Arbeit**

Evaluierung des jMolecules Framework zur architektonischen Abstraktion im Kontext des Domain-Driven Design

**Stichworte**

Domain-Driven Design, Software-Architektur, jMolecules-Framework, ArchUnit, jQAssistant

**Kurzzusammenfassung**

In der vorliegenden Bachelorarbeit wird das jMolecules Framework für Java zur architektonischen Abstraktion im Kontext des Domain-Driven Design evaluiert. Ziel ist die Lösung des Problems der Überprüfbarkeit bezüglich des korrekten Einsatzes der architektonischen Konzepte des Domain-Driven Designs in Softwareprojekten. Das jMolecules Framework bietet zur Lösung dieses Problems Funktionen zur Abbildung, Verifizierung und Dokumentation dieser architektonischen Konzepte. Darüber hinaus liefert das Framework Funktionen zu Reduzierung von technischem Boilerplate Code.
Zur Bewertung dieser Funktionen wird zunächst eine Anforderungsanalyse durchgeführt und anschließend eine Fallstudie erstellt, anhand derer die Erfüllung der Anforderungen ausgewertet wird.
Die Ergebnisse dieser Auswertung zeigen, dass das jMolecules Framework einen Großteil architektonischen Konzepte abbilden kann. Es können jedoch nur zwei Drittel der erforderlichen Designregeln automatisiert verifiziert werden. Die Dokumentationsfunktionen können aufgrund von Problemen mit der Einbindung nicht bewertet werden. Des Weiteren liefert das Framework sehr gute Ergebnisse bei der Reduzierung von Boilerplate Code und eignet sich für den kombinierten Einsatz mit dem Spring Framework.
Alles in allem bietet das jMolecules Framework trotz vorhandener Schwachpunkte einen signifikanten Mehrwert für die Softwareentwicklung im Kontext des Domain-Driven Design.

**Kai Meier**

**Title of Thesis**

Evaluation of the jMolecules Framework for Architectural Abstraction in the Context of Domain-Driven Design

**Keywords**

Domain-Driven Design, Software architecture, jMolecules framework, ArchUnit, jQAssistant

**Abstract**

In this bachelor thesis, the jMolecules framework for Java is evaluated for architectural abstraction in the context of Domain-Driven Design. The goal is to solve the problem of verifiability regarding the correct use of architectural concepts of Domain-Driven Design in software projects. To solve this problem, the jMolecules framework provides functions for mapping, verifying and documenting these architectural concepts. In addition, the framework provides functions to reduce technical boilerplate code.
To evaluate these functions, a requirements analysis is first performed and then a case study is created, which is used to evaluate the fulfillment of the requirements.
The results of this evaluation show that the jMolecules framework can represent a large part of architectural concepts. However, only two-thirds of the required design rules can be verified automatically. The documentation functions cannot be evaluated due to problems with the integration. Furthermore, the framework delivers very good results in the reduction of boilerplate code and is suitable for combined use with the Spring Framework.
All in all, the jMolecules framework offers significant added value for software development in the context of Domain-Driven Design, despite existing weaknesses.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings

# 1 Einleitung

Zum Einsteig wird in diesem Kapitel zunächst die grundlegende Motivation und das Ziel dieser Arbeit vorgestellt. Des Weiteren wird das Thema abgegrenzt und genau definiert, was in dieser Arbeit behandelt wird. Schlussendlich wird die Struktur der Arbeit und damit auch die Vorgehensweise der Evaluierung erläutert.

## 1.1 Motivation und Ziel

Diese Bachelorarbeit beschäftigt sich mit dem Problem von zuverlässiger Überprüfbarkeit der korrekten Verwendung von Domain-Driven Design in Softwareprojekten. Domain-Driven Design bietet einen umfangreichen Werkzeugkasten an Techniken um Software mit gutem und effektivem Design zu entwerfen. Hierfür wird ein besonderes Augenmerk auf die Domäne gelegt, die eine Software abbilden soll. Die domänenspezifische Sprache, „Ubiquitous Language“ genannt, wird bis in den Code hinein verwendet. Auch der generelle Aufbau des Programmes soll ein konsequentes Modell der Domäne widerspiegeln. Um dieses Vorhaben umzusetzen, arbeitet das Domain-Driven Design mit verschiedenen Konzepten wie die „Bounded Contexts“, „Subdomains“ und „Aggregates“. Diese Bausteine des Softwaremodells werden auch als „Building Blocks“ bezeichnet und haben spezifische Regeln sowohl in Bezug auf ihren Aufbau als auch auf die Kommunikation untereinander.

Die Einhaltung genau dieser Regeln in der Implementierung ist meist den Entwicklern selbst überlassen und schwer überprüfbar. Gleichzeitig sorgt die eindeutige Deklaration der einzelnen Bausteine im Code für ein gewisses Problempotential. Die Kennzeichnung einer dieser Bausteine im Code, zum Beispiel einer Aggregate Root, über den Klassennamen widerspricht dem Konzept des konsequenten Einsatzes der „Ubiquitous Language“. Aufgrund der Verwendung technisch abstrakter Begriffe und entfernt die Implementierung weiter vom eigentlichen Domänen-Modell. Diese Diskrepanz zwischen Modell und

Code wird auch als „Model-code gap“ bezeichnet. Ziel ist es den „Model-code gap“ möglichst gering zu halten, da dessen Reduzierung einen besseren Umgang mit Komplexität und Skalierung ermöglicht [5, S. 167-171].

Als ein möglicher Lösungsansatz dieser Probleme wird eine Evaluation des jMolecules Framework für Java durchgeführt, welches Teil eines Projektes namens xMolecules ist. Ziel dieses Projektes ist es architektonische Konzepte im Code zu abstrahieren sowie die Einhaltung der Regeln dieser Konzepte überprüfbar machen und damit das Programmieren erleichtern und Fehler vermeiden. Das jMolecules Framework ermöglicht hier die Deklaration architektonischer Konzepte des Domain-Driven Designs mittels Annotationen oder einer typbasierten Methode. Zur Verifizierung der korrekten Einhaltung der Regeln dieser Konzepte werden verschiedene Bibliotheken unterstützt. Darüber hinaus bietet jMolecules Funktionen zur Erzeugung automatischer Architekturdokumentation und technischem Boilerplate Code.

## 1.2 Themenabgrenzung

In dieser Arbeit werden ausschließlich Möglichkeiten zur architektonischen Abstraktion der Konzepte des Domain-Driven Designs evaluiert. Hierzu gehört sowohl die Markierung der architektonischen Konzepte des Domain-Driven Designs sowie die Verifizierung der spezifischen Designregeln dieser Konzepte. Darüber hinaus wird auf die Möglichkeiten des jMolecules Frameworks zur automatischen Generierung von Architekturdokumentation und technischem Boilerplate Code eingegangen. Um die Komplexität dieser Arbeit in einem angemessenen Rahmen zu halten werden bewusst weitere Funktionen des jMolecules Frameworks ausgeklammert, die sich nicht auf Domain-Driven Design beziehen. Bei diesen Funktionen handelt es sich um die Markierung und Verifizierung von architektonischen Designpatterns der *Command and Query Responsibility Segregation (CQRS), Layered, Onion* und *hexagonalen* Software-Architektur.

## 1.3 Struktur

Zu Beginn werden zunächst die nötigen Grundlagen erläutert, die benötigt werden um die im weiteren Verlauf genutzten Konzepte richtig einordnen zu können. Anschließend wird

eine Anforderungsanalyse, basierend auf den Zielen des Frameworks und der Fachliteratur durchgeführt. Mit dessen Hilfe wird festgelegt, welche Voraussetzungen zur Erfüllung der, vom jMolecules Framework definierten, Ziele gestellt werden. Als Basis zur Bewertung dieser Anforderungen wird im Anschluss an die Anforderungsanalyse eine Fallstudie durchgeführt. In dieser Fallstudie werden unter Kombination der verschiedenen Technologien des jMolecules Frameworks, diverse Versionen des Fallszenarios umgesetzt. Nach erfolgreicher Durchführung der Fallstudie werden die zuvor definierten Anforderungen auf diese angewendet und die Ergebnisse umfassend erläutert. Letztendlich wird zum Abschluss der Arbeit ein Fazit gezogen und ein kurzer Ausblick gegeben.

# 2 Grundlagen

Dieses Kapitel soll dazu dienen, die Grundkonzepte von Domain-Driven Design kurz und erläutern, um die nötige Wissensbasis zum Verständnis der weiteren Arbeit und der darin enthaltenen Terminologie zu schaffen. Darüber hinaus wird das Kernthema dieser Arbeit, das jMolecules Framework, mit seinen Funktionen und Zielen vorgestellt.

## 2.1 Domain-Driven Design (DDD)

### 2.1.1 Überblick DDD

„DDD ist ein Satz von Werkzeugen, die beim Entwerfen und Implementieren von hochwertiger Software helfen, und das sowohl auf strategischer als auch auf taktischer Ebene“[11, S. 1]. Mit diesem Satz lässt sich DDD wohl am besten zusammenfassen. Dennoch steckt hinter dem Begriff weitaus mehr als nur ein Werkzeugkasten. DDD legt den Fokus der Softwareentwicklung auf die Anwendungsdomäne, die von der Software abgebildet wird, und setzt es sich zum Ziel, diese bis hinein in den Code möglichst genau widerzuspiegeln ohne sich dabei in zu technischen Abstraktionen zu verlieren. Hierzu ist die Einbindung der Fachexperten oder auch Domain Experts in den Design- und Entwurfsprozess unerlässlich. Statt über Umwege sollen bei DDD die Entwickler und Domain Experts direkt zusammenarbeiten, miteinander sowie voneinander lernen und sogar dieselbe Sprache sprechen [10, S. 9]. Diese Sprache, „Ubiquitous Language“ genannt, dient nicht nur zur Kommunikation sondern ist auch ein zentrales Element des strategischen und taktischen Designs. Im Allgemeinen ist das Design einer der wichtigsten Aspekte von DDD. Oft spielt das Design in Softwareprojekten aufgrund von Geldeinsparungen eine untergeordnete Rolle oder findet ausschließlich in den Köpfen der Entwickler statt. Bei DDD wird der Fokus auf effizientes Design gelegt, welches die notwendigen Anforderungen in dem

Grad erfüllt, der benötigt wird um die Wettbewerber abzuhängen sowie die Kernkompetenzen hervorzuheben. Kein Design ist hier gleichzusetzen mit schlechtem Design [11, S. 3-7].

### 2.1.2 Strategisches Design

Beim strategischen Design geht es um die strategische Analyse und konzeptionelle Modellierung des Designs innerhalb des Problemraums, der anschließend in den Lösungsraum überführt wird. Hier findet die tatsächliche Implementierung statt. Das strategische Design findet generell auf einer höheren Abstraktionsebene statt und setzt somit den Rahmen für die praktische Umsetzung ohne dabei zu tief in implementierungstechnische Details des taktischen Designs vorzudringen [11, S. 7f].

#### Bounded Contexts

Bei Bounded Contexts (begrenzte Kontexte) handelt es sich um ein konzeptionelles Instrument, über das die Komponenten des Softwaremodells kontextspezifisch und semantisch unter Einbeziehung von sprachlichen Grenzen voneinander trennt werden [11, S. 11f]. Dies hilft dabei die Komplexität zu verringern, die wichtigsten Kernbereiche des Geschäftsmodells zu identifizieren und dementsprechend Prioritäten zu setzen für die Verteilung der verfügbaren Ressourcen und Arbeitskräfte. An jedem Bounded Context arbeitet maximal ein Team. Jedoch kann ein Team für mehrere Bounded Contexts verantwortlich sein. Des Weiteren ist es üblich für jeden Bounded Context ein eigenes Quellcode-Repository anzulegen und ebenfalls die Datenbank getrennt von anderen Bounded Contexts anzulegen. Auf diese Weise wird die Entstehung von großen Software-Monolithen oder auch „Big Ball of Muds“ vermieden. So wird versucht, die gesamte Geschäftslogik in einem einzigen Quellcode-Repository abzubilden [11, S. 12-16].

#### Ubiquitous Language

Die Ubiquitous Language oder übersetzt die allgegenwärtige Sprache nimmt im DDD eine wichtige Rolle ein. Sie spiegelt die Sprache der Fachexperten wider und ist Basis der Kommunikation des gesamten Teams einschließlich der Entwickler. Gleichzeitig beeinflusst die Ubiquitous Language die Grenzen der Bounded Contexts und wirkt sich

auf die Architektur der Software aus [11, S. 12]. Entwickler haben die Aufgabe sich mithilfe der Fachexperten Stück für Stück deren Sprache anzueignen und diese konsequent zu verwenden, ihre Kenntnisse auszubauen sowie die Sprache bis hin in den Code zu tragen. Ein wichtiges Konzept von DDD ist, dass der Code die Domäne widerspiegelt. Konsequenterweise wird der Code frei von technisch abstrakten Begriffen gehalten, indem durchgehend die Ubiquitous Language genutzt wird [11, S. 27]. Die Ubiquitous Language beeinflusst die Grenzen der Bounded Contexts in der Form, dass pro Bounded Context nur eine eindeutige Ubiquitous Language existieren darf. Wird festgestellt, dass innerhalb des festgelegten Bounded Contexts verschiedene Definitionen für selbe Begriffe existieren oder Begriffe auftauchen, die außerhalb des Kerns der Geschäftsstrategie liegen, ist dies ein Zeichen dafür, dass es innerhalb des Contexts Konzepte gibt, die in einen eigenen Bounded Context ausgelagert werden sollten. So wird Fehlern entgegnet, die aufgrund von Missverständnissen entstehen sowie die Komplexität in einem geeigneten Rahmen gehalten [10, S. 68ff].

#### Subdomains

Eine Subdomain ist ein Teilbereich einer übergeordneten Domain. Die Domain ist im DDD die gesamte Domain des Geschäfts, in dem die Software entwickelt wird. Diese wird wiederum in verschiedene Subdomains unterteilt, welche die Bereiche der Geschäftsdomain logisch voneinander abgrenzen [10, S. 44]. Unterschieden wird bei Subdomains zwischen der „Core Domain“, „Supporting Subdomains“ und „Generic Subdomains“. Die Core Domain spiegelt das Kerngeschäft des Unternehmens wider und ist der Bereich, auf den der Hauptfokus liegt. Hier werden die besten Entwickler eingesetzt und das größte Budget bereitgestellt. Supporting Subdomains unterstützen oder ergänzen die Core Domain ohne ein Teil davon zu sein. Für Supporting Subdomains werden geringere Budgets bereitgestellt oder ggf. Standard-Lösungen verwendet. Bei Generic Subdomains handelt es sich um weniger wichtige Teilbereiche, die nicht in direktem Bezug zum Kerngeschäft stehen wie z. B. die Benutzeranmeldung. Es empfiehlt sich hier die Arbeit mit geringeren Budgets oder ggf. die Verwendung von Standard-Lösungen oder Outsourcing. So lassen sich möglichst viele Ressourcen für die Core Domain bereitstellen [10, S. 50ff].

#### Context Mapping

Der Schritt des Context Mappings regelt die Kommunikation und damit den Informationsaustausch zwischen den verschiedenen Bounded Contexts. Ergebnis des Context Mappings ist eine sogenannte Conext Map, die die Arten von Beziehungen zwischen den Bounded Contexts widerspiegelt. Das Context Mapping regelt hier die Form der Übersetzung in der Kommunikation zweier Bounded Contexts und ihrer spezifischen Ubiquitous Languages anhand der Art der Beziehung zwischen diesen. Unterschieden wird die Art der Beziehung in folgende Varianten: Partnership, Shared Kernel, Customer-Supplier, Conformist, Anticorruption Layer, Open Host Service Published Language und Speperate Ways. Einen tieferen Einblick ins Context Mapping liefert Kapitel drei in Implementing Domain-Driven Design [10].

### 2.1.3 Taktisches Design

Beim taktischen Design wird das zuvor ausgearbeitete Domänen-Modell des strategischen Designs vertieft und auf feingranularer Ebene weiter ausgearbeitet. Hier werden nun auch implementierungsspezifische Themen angesprochen und die vorher definierten Bounded Contexts mithilfe der sogennanten Building Blocks ausmodelliert. Zur Kategorie der Building Blocks gehören unter anderem Aggregates, welche die transaktionalen Konsistenzgrenzen festlegen. Darüber hinaus die, in den Aggregates enthaltenen, Entities und Value Objects sowie Domain Events, mit deren Hilfe sich Ereignisse innerhalb der Domain signalisieren und behandeln lassen [11, S. 8f].

#### Entities, Value Objects und Aggregates

Entities oder auch Entitäten bilden Konzepte im Domänen-Modell ab, welche eine einzigartige Identität sowie einen Lebenszyklus haben. Entities werden zu einem bestimmten Zeitpunkt erstellt, im weiteren Verlauf verwendet und irgendwann wieder gelöscht [4, S. 89-93]. Ein Beispiel für eine Entity ist ein Kunde mit einer einzigartigen Kundennummer. Dieser Kunde wird zu einem bestimmten Zeitpunkt im System angelegt, bestellt anschließend Produkte und wird eventuell ggf. auf eigenen Wunsch wieder aus dem System entfernt.

Im Gegensatz zu Entities besitzen Value Objects keine individuelle Identität und keinen Lebenszyklus. Darüber hinaus sind Value Objects im Gegensatz zu Entities immer unveränderbar. Value Objects werden verwendet, um Entitäten in Form von fachlichen Werten zu beschreiben [4, S. 97ff]. Beispielsweise ließen sich bei einem Fahrrad die einzelnen Teile aus denen es gefertigt wurde als Value Objects definieren. Jedes Teil hat einen Hersteller, eine Modellbezeichnung und einen Typen, jedoch keine eigene Identität, da es nur zur Beschreibung des Fahrrads dient und die generelle Art des Bauteils widerspiegelt, nicht das einzelne Teil selbst.

Aggregates wiederum fassen ein oder mehrere Entities und Value Objects in sich zusammen. Jedes Aggregate besteht aus einer Root Entity, also einer Wurzel Entity und allen weiteren Entities und Value Objects, die in dieser Root Entity zusammengefasst sind. Sie modellieren die transaktionalen Konsistenzgrenzen innerhalb des Domänen-Modells und legen damit fest, welche Entitäten und Value Objects sich in einem konsistenten Zustand befinden müssen, wenn eine Transaktion in die Datenbank schreibt [4, S. 125-129]. Aggregates sind das zentrale Objekt der Building Blocks. Ihre korrekte Modellierung nimmt große Abschnitte in der Fachliteratur ein.

## 2.2 jMolecules Framework

Beim jMolecules Java Framework handelt es sich um den Java-spezifischen Teil des Open-Source-Projekts xMolecules. Ziel des xMolecules Projekts ist es die abstrakten architektonischen Konzepte des DDD im Code der jeweiligen Programmiersprache einzubetten, ohne dabei das Konzept der Ubiquitous Language zu verletzen. Auf diese Weise soll sowohl die Verständlichkeit verbessert, die Dokumentation durch Automatisierung erleichtert, die abstrakten Konzepte der Architektur mit Ihren spezifischen Regeln testbar gemacht und darüber hinaus auch mittels Technologieintegration Boilerplate Code reduziert werden können [1].

### 2.2.1 Aufbau xMolecules Projekt

Das xMolecules Projekt besteht zurzeit aus drei Teilbereichen: phpMolecules ist spezialisiert auf die Sprache PHP, jMolecules richtet sich an Java Entwickler und nMolecules ist in der Sprache C# umgesetzt. Aktuell nimmt jMolecules den größten Umfang innerhalb

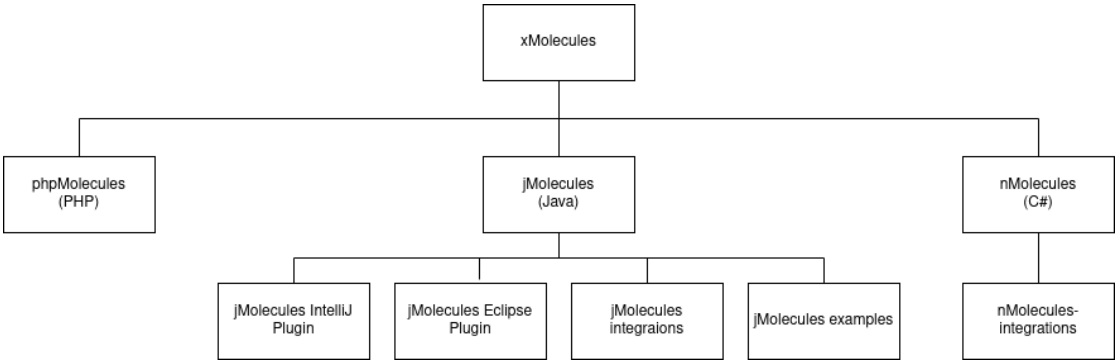


Abbildung 2.1: Aufbau xMolecules Projekt.

des Projekts ein und ist der am aktivsten weiterentwickelte Bereich. jMolecules selbst wird wiederum durch verschiedene Unterprojekte erweitert. Diese sind zum einen die jMolecules IntelliJ und Eclipse Plugins, jMolecules Integrations zur Technologieintegration und ein Beispiel-Repository namens jMolecules examples, in dem die verschiedenen Funktionen des Frameworks veranschaulicht werden. nMolecules besitzt ebenfalls ein Unterprojekt namens nMolecules Integrations [9].

### 2.2.2 DDD-Konzepte ausdrücken

Das jMolecules Framework bietet zwei verschiedene Möglichkeiten die Konzepte der Building Blocks im Code auszudrücken. Zum einen kann die annotationsbasierte Methode gewählt oder auf die typbasierte Methode zurückgegriffen werden. Beide Methoden ermöglichen es architektonische Konzepte auszudrücken, ohne dabei die Regeln die Ubiquitous Language zu verletzen [1].

#### Annotationen

Eine Möglichkeit die Architektur innerhalb des Codes zu abstrahieren bietet jMolecules über Java Annotationen. Hier werden Klassen, Attribute oder Methoden mithilfe der Annotationen als bestimmte Elemente der Building Blocks gekennzeichnet. Auf diese Annotationen wird anschließend zugegriffen, wenn die Architekturregeln mithilfe von ArchUnit oder jQAssistatnt verifiziert werden sollen [1].

#### Typen

Bei der typbasierten Methode werden die Konzepte der Building Blocks über Marker Interfaces ausgedrückt, die von den jeweiligen Klassen implementiert werden. Bei den Interfaces handelt es sich zumeist um generische Datentypen, mit deren Hilfe nicht nur die Zugehörigkeit der Klasse ausgedrückt wird, sondern auch auf welchem Wege sie eindeutig identifiziert werden kann [1].

### 2.2.3 Architektonische Konzepte ausdrücken

Über die Konzepte des Domain-Driven Design hinaus bietet jMolecules die Möglichkeit verschiedene allgemeine architektonische Konzepte in Form von Annotationen auszudrücken und diese verifizierbar zu machen. Unterstützt werden hier die Schichten-, Zwiebel-, Hexagonal-Architektur sowie CQRS (Command Query Responsibility Segregation) [1]. Da sich diese Arbeit auf den Kontext des Domain-Driven Design ausrichtet, werden diese Themen nicht tiefergehend behandelt.

### 2.2.4 jMolecules Technology Integrations

Die jMolecules Technology Integrations haben den Zweck Boilerplate-Code zu reduzieren, indem basierend auf den jMolecules Annotationen und Typen, automatisch Boilerplate Code generiert wird. Dieser Boilerplate Code übersetzt die Konzepte von jMolecules in die Konzepte der gewünschten Zieltechnologie. Unterstützte Technologien sind Spring, Data JPA, Data MongoDB, Data JDBC und Jackson. Zusätzlich mit enthalten in den jMolecules Technology Integrations ist ein Satz von ArchUnit Regeln, der genutzt werden kann um Aggregatstrukturen innerhalb des Models zu verifizieren [8]. Bei ArchUnit handelt es sich um eine Bibliothek zur Überprüfung der Architektur von Java Code unter Verwendung eines Java-Unit-Test-Frameworks. Auf diese Weise lassen sich zum Beispiel Abhängigkeiten zwischen Paketen, Klassen, Schichten sowie zyklische Abhängigkeiten prüfen [7].

### 2.2.5 jQAssistant Plugin

Bei jQAssistant handelt es sich um ein Open-Source-Plugin, mit dessen Hilfe sich die Struktur der Software auf einer abstrakteren Ebene darstellen und Architektur- sowie

Designregeln verifizieren lassen. Um dies zu realisieren, arbeitet jQAssistant im Hintergrund mit der Neo4j Graph-Datenbank, auf der diverse Abfragen in der Abfragesprache Cypher gestellt werden können. jQAssistant scannt die Softwarestrukturen und speichert diese als Graphen in der Datenbank ab, wobei die einzelnen Bausteine die Knoten des Graphen und ihre Beziehungen zueinander die Kanten abbilden. Auf diese Weise lassen sich mit einfachen Cypher-Abfragen Verstöße gegen die Architektur- und Designregeln aufdecken [6].

# 3 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an das jMolecules Framework erläutert. Zur Ermittlung der Anforderungen wurde zunächst die Dokumentation des Frameworks [1] analysiert und vier Hauptziele des Frameworks identifiziert. Diese Ziele dienen als grundlegende Anforderungen:

1. DDD Konzepte explizit im Code abbilden
2. DDD Designregeln verifizieren
3. Automatische Generierung von Architekturdokumentation
4. Reduzierung von technischem Boilerplate Code

Basierend auf diesen vier Hauptanforderungen wurde mit Hilfe von *Implementing Domain-Driven Design* [10] und *Domain-Driven Design - Tackling Complexity in the Heart of Software* [4] auf feingranularer Ebene ausgearbeitet, welche Voraussetzungen erfüllt werden müssen, damit die Hauptanforderungen ebenfalls als erfüllt angesehen werden können. Auf diese Voraussetzungen wird in den folgenden Unterpunkten genauer eingegangen.

## 3.1 A.1 DDD Konzepte explizit im Code abbilden

Eine der wichtigsten Anforderungen ist es, die Konzepte des Domain-Driven Design explizit im Code abbilden zu können. Hierbei muss gleichzeitig gewährleistet werden können, dass das Konzept der Ubiquitous Language, welches besagt, dass die Software bis in den Code hinein ein genaues Modell der realen Anwendungsdomäne darstellen soll, nicht verletzt wird. Dies bedeutet, dass im Code keine abstrakten, technischen Begriffe auftauchen, sondern ausschließlich die Ubiquitous Language verwendet wird. Um dies gewährleisten zu können muss das Framework eine Möglichkeit bieten die Designelemente des

DDD zu kennzeichnen ohne direkten Einfluss auf deren Benennung zu nehmen. Hierdurch soll es Entwicklern und Architekten leichter gemacht werden die Struktur der Software auf abstrakterer Ebene analysieren zu können. Diese abstraktere Betrachtungsweise der Struktur erleichtert das allgemeine Verständnis über den Aufbau und die Funktion der Software, indem es Komplexität herausnimmt.
Basierend auf dieser Anforderung wurde mithilfe des Buchs *Implementing Domain-Driven Design* [10] eine Liste der Kernkonzepte des DDD erarbeitet, die im Code abbildbar sein sollen:

A.1.1 Aggregates

A.1.2 Aggregate Roots

A.1.3 Entities

A.1.4 Entity Identitäten

A.1.5 Mit der Entity verbundene Aggregate Root

A.1.6 Value Objetcs

A.1.7 Mit dem Value Object verbundene Entity

A.1.8 Bounded Contexts

A.1.9 Subdomain allgemein

A.1.10 Core Subdomain

A.1.11 Supporting Subdomain

A.1.12 Generic Subdomain

A.1.13 Module

Außerhalb dieses Kerns liegen weitere Konzepte, die allerdings zunächst außen vor gelassen werden, da es sich hier um bereits allgemein bekannte und etablierte Designkonzepte handelt. Diese tauchen in der DDD spezifischen Fachliteratur nur in der Hinsicht auf, dass Wege aufgezeigt werden, wie sie zusammen mit DDD eingesetzt werden. Zu diesen Konzepten gehören Domain Events, Domain Services, Domain Factories und Domain Repositories. Die Möglichkeit diese Konzepte abzubilden, wird hier als eine Zusatzanforderung (A.1.14) betrachtet, deren Erfüllung nicht als zwingend erforderlich angesehen wird.

## 3.2 A.2 DDD Desingregeln verifizieren

Über die Funktion des reinen Markierens von DDD-Designelementen hinaus, ist die Verifizierung der Designregeln dieser Elemte eine äußerst wichtige Anforderung. Die Markierung soll hierzu die nötige Grundlage schaffen, um auf die Elemente und ihr Verhalten zugreifen zu können und dieses Verhalten wiederum mit spezifischen Designregeln abgzugleichen. Auf diese Weise sollen Fehler in der Umsetzung frühzeitig erkannt und vermieden werden. Häufig entstehen solche Umsetzungsfehler aufgrund von Zeitdruck, Flüchtigkeitsfehlern oder unzureichendes Verständnis der Entwickler in Bezug auf die verwendeten Designelemente des Domain-Driven Design. Eine automatische Verifizierung dieser Designregeln kann ggf. viel Zeit und Ressourcen einsparen.
Gleichzeitig muss berücksichtigt werden, dass nicht jede Designregel einfach automatisiert prüfbar ist. So lässt sich zum Beispiel schwer kontrollieren, ob die Ubiquitous Language konstant beachtet wurde, da sowohl die Sprache selbst als auch ihre korrekte Verwendung bekannt sein muss. Dies durch ein Framework zu automatisieren scheint, in Anbetracht der Komplexität und Individualität eines Projektes, zum aktuellen Zeitpunkt unrealistisch. Aus diesem Grund werden in den Anforderungen nicht sämtliche Designregeln aufgeführt, sondern ausschließlich Regeln in Bezug auf die Syntax der Modellierung.

In der Tabelle 3.1 wurden, aus den oben genannten Gründen, Regeln zur Ubiquitous Language nicht berücksichtigt. Darüber hinaus wurden Regeln zur Dimensionierung und Strukturierung innerhalb von Bounded Contexts, Subdomains und Modules nicht mit aufgenommen, da diese Regeln stark abhängig vom fachlichen Kontext der Anwendung und ihrer Domäne sind. Sie beinhalten Richtlinien und Anhaltspunkte, wie diese Konzepte in der Anwendungsdomäne ausfindig gemacht und darauf basierend modelliert werden. Hierbei gibt es keine festen Kennzahlen oder bestimmte Verhaltensweisen, die allgemeingültig als richtig oder falsch definierbar wären, da diese stets die individuelle Anwendungsdomäne widerspiegeln.

| **Anforderungs- / Designregel-Kategorie** | **Nr.** | **Beschreibung** |
|---|---|---|
| A.2.1 Bounded Contexts | A.2.1.1 | Beziehungen zwischen Bounded Contexts sind nur erlaubt wenn diese zuvor explizit definiert wurden. |
| A.2.2 Aggregates | A.2.2.1 | Jedes Aggregate bildet eine transaktionale Konsistenzgrenze. |
| | A.2.2.2 | Referenzen auf andere Aggregates dürfen nur über die Identität erfolgen. |
| | A.2.2.3 | Ein Aggregate benötigt ein Aggregate Root. |
| A.2.3 Entities | A.2.3.1 | Entities benötigen eine zugehörige Root. |
| | A.2.3.2 | Entities benötigen eine Identität über die sie identifizierbar sind. |
| | A.2.3.3 | Identitäten dürfen nicht veränderbar sein. |
| | A.2.3.4 | Entities dürfen nur innerhalb des Aggregate referenziert werden in dem sie definiert sind. |
| A.2.4 Value Objects | A.2.4.1 | Value Objects dürfen nicht veränderbar sein. |
| | A.2.4.2 | Value Objects besitzen keine Identität. |
| | A.2.4.3 | Keine Referenzen auf Entities oder Aggregate Roots. |
| | A.2.4.4 | Value Objects dürfen nur innerhalb des Aggregate referenziert werden in dem sie definiert sind. |

Tabelle 3.1: Anforderungen DDD Designregeln

## 3.3 A.3 Dokumentation

Basierend auf den, im Code abgebildeten, Designkonzepten sollte das Framework die Möglichkeit bieten automatische Dokumentationen über die Struktur der Software auf abstrakter Designebene zu generieren.
Hierzu gehören die einzelnen Klassen, Packages und Interfaces sowie Ihre Beziehungen zueinander und die Domain-Driven Design Konzepte, die sie repräsentieren.

## 3.4 A.4 Boilerplate Code reduzieren

Um Boilerplate Code zu reduzieren, soll das Framework die Möglichkeit bieten diesen anhand der verwendeten DDD-Konzepte automatisch zu generieren, so weit es möglich ist, um die Verwendung anderer Technologien zusammen mit dem jMolecules Framework zu vereinfachen. Besonders bei Anwendungen in Kombination mit dem Spring Framework oder der Java Persistence API (JPA) kann es zu begrifflichen Redundanzen kommen. Diese Redundanzen wiederum könnten für Verständnisprobleme und Fehler aufgrund von Verwechslungen sorgen.
Darüber hinaus dient jede Reduzierung von technischem Boilerplate Code dem Ziel der konsequenten Verwendung der Ubiquitous Language und den Verzicht auf abstrakte technische Begrifflichkeiten, die nicht in direktem Bezug zum Domänenmodell stehen.

# 4 Fallstudie

Als Bewertungsgrundlage der Evaluation wird im folgenden Kapitel eine Fallstudie durchgeführt. In dieser Fallstudie wird ein Beispielszenario erstellt, welches auf einem Online-Shop für Mountainbikes basiert. Dieses Szenario wird anschließend auf verschiedene Weisen implementiert, um die Vor- und Nachteile der verschiedenen Technologieoptionen, die das jMolecules Framework anbietet, erörtern zu können.

## 4.1 Szenario

Das fiktive Unternehmen „BestBikes“ plant die Erstellung eines Shops für den Verkauf von Mountainbikes. Da die Entwicklungsabteilung von BestBikes bereits gute Erfahrungen mit dem Einsatz von Domain-Driven Design in Ihren Softwareprojekten gemacht hat, soll dieser Designansatz zum Einsatz kommen.
Trotz der allgemein guten Erfahrung mit Domain-Driven Design, sind in Vergangenheit wiederholte Probleme in der Praxis aufgetaucht. Im Entwicklungsprozess ist es immer wieder vorgekommen, dass die reale Architektur und die vorher geplante Architektur der Software voneinander abgewichen sind, sowie architekturspezifische Designregeln verletzt wurden. Als Gründe hierfür wurden Probleme mit der Übersicht über die Struktur und Verständnisprobleme in Bezug auf die architektonischen Designregeln genannt. Darüber hinaus kam es vermehrt dazu, dass die vorher angelegten Entwürfe und Dokumentationen zur Architektur im Rahmen des Entwicklungsprozesses aufgrund von Zeitmangel nicht konsequent gepflegt wurden.
Als potenzielle Lösung für diese Probleme ist das Unternehmen BestBikes nun auf das jMolecules Framework gestoßen. Um herauszufinden, ob das Framework die vorhandenen Probleme lösen kann, werden unter Kombination der verschiedenen Technologieoptionen

des Frameworks diverse Implementationen eines stark vereinfachten Modells Ihres geplanten Shops umgesetzt und anschließend bewertet.
Ziel ist die Klärung folgender Fragestellungen:

1. Wie lassen sich die Konzepte des DDD am besten im Code abbilden?
2. Wie lassen sich die Designregeln am besten verifizieren?
3. Wie gut funktioniert die Technologieintegration zur Reduzierung von Boilerplate-Code?
4. Wie gut funktioniert die automatische Generierung von Software-Dokumentation?

## 4.2 Umsetzung

Zur Umsetzung des Szenarios wurde zunächst mit Event Storming begonnen, einer typischen Designtechnik im Umfeld des Domain-Driven Designs. Die Ergebnisse des Event Stormings beschreiben bereits die Bounded Contexts, Subdomains und Aggregates. Darauf basierend wurde die grundlegende Struktur der Software abstrahiert und in Form eines UML-Klassendiagramms festgehalten, sowie der erste Testfall, ohne den Einsatz des jMolecules Frameworks, umgesetzt. Anschließend wurden die weiteren Testfälle definiert und ebenfalls implementiert.

### 4.2.1 Even Storming

#### Event Storming allgmein

Beim Event Storming handelt es sich um eine leichtgewichtige, iterative Designtechnik in der Softwareentwicklung mit Fokus auf Geschäftsprozesse. Domain Experts sowie Entwickler modellieren zusammen mithilfe von verschiedenfarbigen Klebezetteln und Stiften die grundlegenden Prozesse, welche die geplante Software umsetzen soll. Die Klebezettel repräsentieren sowohl Elemente der Geschäftsprozesse als auch Designelemente des DDD wie Bounded Contexts, Subdomains und Aggregates. Ziel ist es sowohl eine die Basis für die spätere Implementierung zu schaffen, als auch einen Lernprozess für Domain Experts und Entwickler anzuregen, mit dem ein tieferes Verständnis über die Prozesse und ihre Abläufe generiert werden soll. Für tiefergehende Informationen zum Event Storming siehe *Introducing EventStorming* [3].

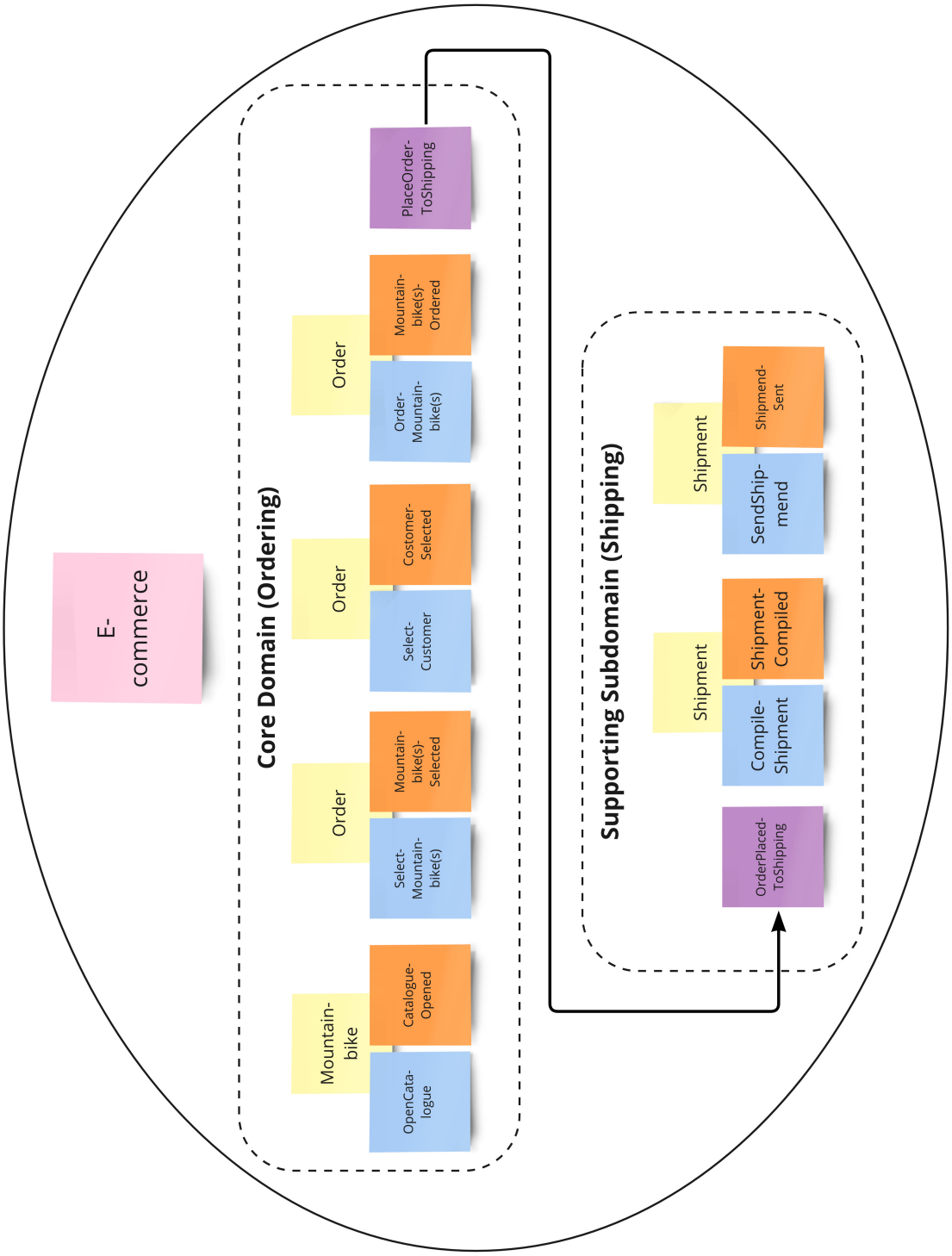


Abbildung 4.1: Geschäftsprozess des Mountainbike-Shops als Ergebnis des Event Stormings.

#### Ergebnis Event Storming

Da es sich hierbei um ein stark vereinfachtes Beispiel zur Veranschaulichung handelt, fällt das Ergebnis des Event Stormings entsprechend simpel aus.
Die Abbildung 4.1 zeigt zwei der grundlegenden Prozesse des geplanten Online-Shops in chronologischer Abfolge von links nach rechts und von oben nach unten. Der große rosa Klebezettel markiert zusammen mit dem äußeren Kreis den Bounded Context „E-commerce“. Die gestrichelten Kästen separieren die beiden Subdomains, in Form der Core Domain „Ordering“ und der Supporting Subdomain „Shipping“, voneinander. Ob die Subdomain Shipping wirklich als eigenständige Subdomain modelliert werden sollte oder ob es Sinn ergeben würde diese in die Core Domain mit einzubetten, bietet durchaus Raum für Diskussionen. In diesem Fall wurde sich jedoch für eine separate Subdomain entschieden, um herauszufinden wie das jMolecules Framework diesen Fall abbildet.
Die blauen, orangen und violetten Klebezettel repräsentieren die Vorgänge der eigentlichen Geschäftsprozesse. Die blauen Klebezettel stehen für sogenannte Commands oder Befehle, auf die Events oder Ereignisse folgen. Diese Events wiederum sind durch die orangen Klebezettel abgebildet. Diese Vorgehensweise erinnert, wenn auch verkehrter Reihenfolge, entfernt an ereignisgesteuerte Prozessketten zur Modellierung von Geschäftsprozessen. Der violetten Klebezettel am Ende des Bestellprozesses in der Ordering Subdomain markiert einen Command, der einen neuen Prozess auslöst. Der Beginn dieses Prozesses ist durch den zweiten violetten Klebezettel, dem zugehörigem Event, in der Shipping Subdomain gekennzeichnet.
Die gelben Klebezettel repräsentieren die Aggregates auf denen die Commands angewendet werden.

### 4.2.2 UML-Diagramm

Wie vielleicht bereits aufgefallen ist, lassen sich aus dem Ergebnis des Event Stormings für Entwickler schon leicht erste Packages, Klassen und Methoden für die spätere Implementierung ableiten. Unter Einbindung dieser Informationen und basierend auf den Designregeln des Domain-Driven Design, wurde UML-Klassendiagramm in Abbildung 4.2 für den Shop von BestBikes entworfen.
Da es sich hier um eine Arbeit mit Fokus auf Architekturkonzepte und Verifizierung von

Designregeln im Kontext des Domain-Driven Design handelt, spielt der Entwurf der Anwendung und die dort getroffenen Entscheidungen bezüglich der DDD-Designelemente eine besonders wichtige Rolle. Aus diesem Grund werden im Folgenden einige wichtige Designentscheidungen genauer betrachtet und ihre Korrektheit anhand von Fachliteratur belegt.

### Package-Struktur

Da Java-Packages ein guter Weg sind um DDD-Module abzubilden [10, S. 333] und eine wichtige Rolle in der Architektur einnehmen, wurden die Packages ab der Ebene des Bounded Contexts E-commerce mit in das Klassendiagramm aufgenommen. Wie in Abbildung 4.2 zu sehen ist, hat der Bounded Context E-commerce ein eigenes Package. Zusätzlich befindet sich auf der Ebene von E-commerce noch ein Package namens Infrastruktur, auf das im weiteren Verlauf noch genauer eingegangen wird.
Innerhalb von E-commerce werden die Subdomains Ordering und Shipping mit jeweils einem Package repräsentiert. Die Entscheidung, die Subdomains mithilfe von Packages zu separieren, wurde basierend auf der Designempfehlung für den Fall von mehreren Subdomains innerhalb von einem Bounded Context in [11, S. 47] getroffen.
Innerhalb der Subdomains befinden sich wiederum weitere Packages, die die einzelnen Aggregates, ebenfalls nach DDD-Vorgaben [10, S. 334f], repräsentieren.
Diese Package-Struktur spiegelt den DDD-Ansatz wider, dass eine Anwendung stets ein Modell der Fachdomäne darstellen soll.

### Entity-Identitäten

Wie im Klassendiagramm zu sehen, sind die Identitäten der Entities in Form von Value Objects umgesetzt worden. Dieser Designansatz basiert auf einer Empfehlung in *Implementing Domain-Driven Design* [10]. Vorteil ist die Möglichkeit den Identitäten selbst seiteneffektfreie Funktionen hinzuzufügen, mit denen sich ggf. nur über die Identität bereits bestimmte Informationen zur Entity abrufen lassen. Dies wäre zum Beispiel der Fall, wenn die Identität über einen String bestimmt wird, welcher bereits Informationen zum Erstelldatum dieser Entity enthält [10, S.177f].
Auch wenn dies in der vorliegenden vereinfachten Anwendung nicht der Fall ist, wurde dieses Designkonzept übernommen, da es unter Umständen in der realen Version des Shops benötigt werden könnte.

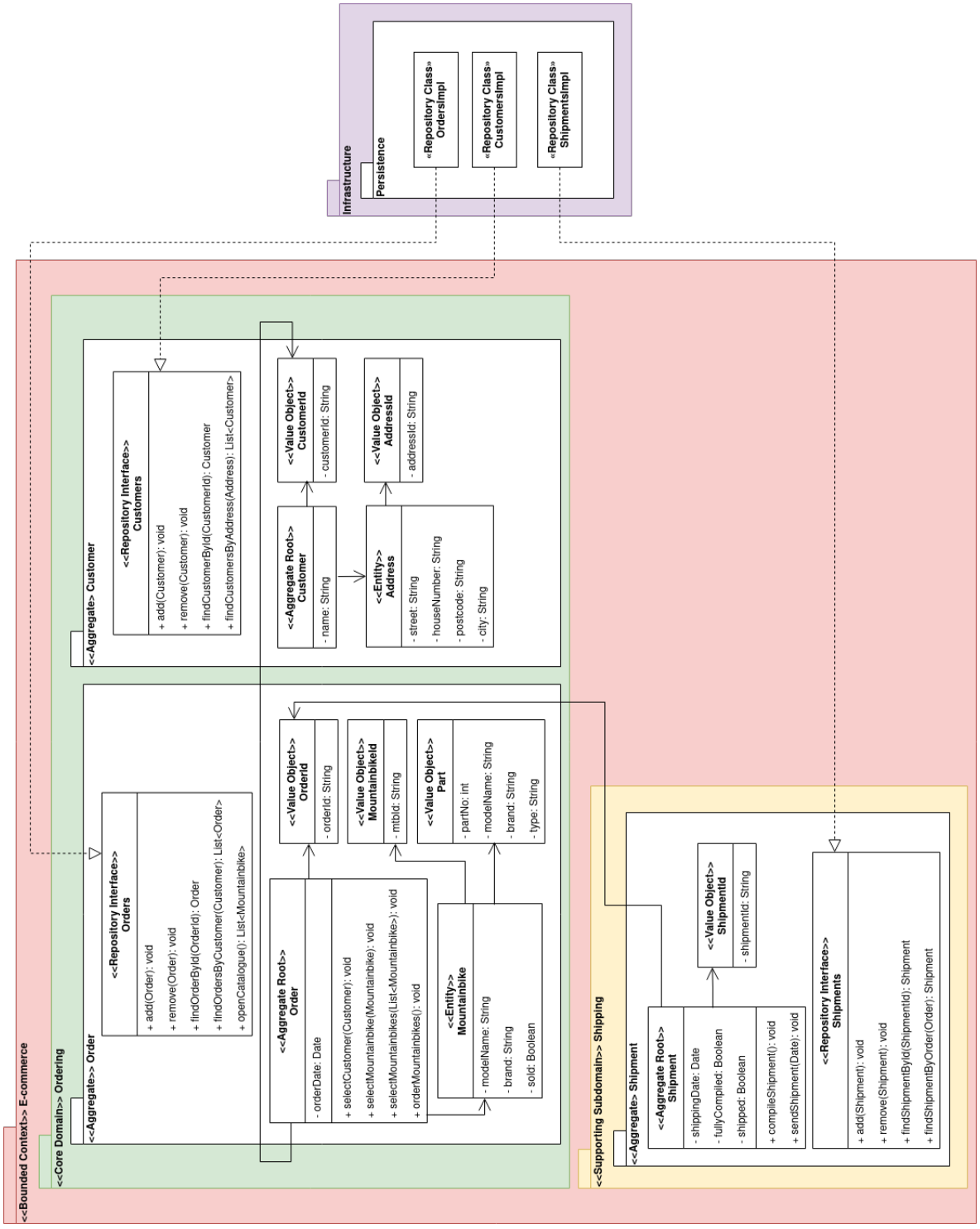


Abbildung 4.2: UML-Klassendiagramm des Mountainbike-Shops basierend auf Event Storming.

### Referenzen über Aggregate-Grenzen

Referenzen von Entities über Aggegrate-Grenzen hinaus finden ausschließlich über die Identitäten der außerhalb liegenden Entities statt. Dieses Konzept dient unter anderem der Regel zum Schutz von fachlichen Invarianten innerhalb von Aggregate-Grenzen mit denen festgelegt wird, welche Entities und Value Objects zum Zeitpunkt des Commits einer Transaktion konsistent mit der Aggregate-Root sein müssen [10, S. 353f]. Werden außenstehende Entitäten nur über die Identität referenziert, wird es durch die losere Kopplung erschwert, mehrere Aggregates innerhalb einer Transaktion anzupassen.
Darüber hinaus halten Referenzen über die Identität das Aggregate-Design leichtgewichtig und klein. Dies entspricht den Designregeln für Aggregates. [10, S. 355-358].
Schlussendlich bietet die Referenzierung über Identitäten den Vorteil verbesserter Skalierbarkeit aufgrund klarer Transaktionsgrenzen mit loser Kopplung unter den Aggregates und der daraus entstehenden Möglichkeit, verteilte Speichersysteme einzusetzen [10, S. 363f].

### Repositories

Bezüglich der Persistenz wurde sich für den Einsatz sogenannter DDD-Repositories entschieden. Repositories bieten die Möglichkeit das Domänen-Modell frei von den rein technischen Details zur Implementierung der Persistenz zu halten. Dies ist im Sinne der Ubiquitous Language. Repositories imitieren nach Außen ein ähnliches Verhalten wie Collections, während sämtliche Details zur Persistierung im Hintergrund, innerhalb der Methoden des Repositories, festgelegt werden. Pro Aggregate gibt es immer ein Repository [10, S. 401f]. Im eigentlichen Domänen-Modell finden sich die Interface-Definitionen der Repositories innerhalb der Aggregate-Packages [10, S. 408]. Die Implementierung dieser Interfaces kann, in Anlehnung an eine Schichten-Architektur, in einem eigenen Package außerhalb des Domänen-Modells ausgelagert werden. Gleichzeitig wäre es hier erlaubt innerhalb der Aggregate-Packages weitere Packages zu erstellen, die jeweils die Implementierung enthalten [10, S. 410f]. Um eine klarere Trennung zwischen dem Domänen-Modell und der technischen Implementierung der Persistenzmechanismen zu schaffen, wurde die Lösung in Anlehnung an die Schichten-Architektur ausgewählt. Widergespiegelt wird dies durch die Packages Infrastructure und Persistence außerhalb des E-commerce Contexts und den dort enthaltenen Implementierungen der Repository-Interfaces.

### 4.2.3 Implementationen

Nach dem Event Storming und der Erstellung des UML-Klassendiagramms kann bereits die eigentliche Umsetzung des Shops, in Form von Programmcode, begonnen werden. Zuvor wurde, aufgrund von vorhandener Erfahrung und der weiten Verbreitung, beschlossen den Microsoft MySQL-Server zur Persistenz einzusetzen.
Insgesamt sollen sechs verschiedene Varianten des Shops umgesetzt werden. Hiervon wird erhofft die bestmögliche Abdeckung von Kombinationen der verschiedenen Technologieoptionen des jMolecules Frameworks zu erreichen, ohne dabei die Komplexität aus dem Rahmen laufen zu lassen, da nicht für jede nur denkbare Kombination eine eigene Version des Shops erstellt wird.
Aufgrund der umfangreichen Integrationsmöglichkeiten des Spring-Frameworks in Kombination mit jMolecules soll der eventuelle Einsatz des Spring Frameworks mit in den Testfällen berücksichtigt werden.

#### Traditionell ohne Spring

Hierbei handelt es sich um die traditionelle Umsetzung, völlig ohne das jMolecules- oder Spring-Framework. Es wurde, wie bei allen sechs Umsetzungen, das Lombok-Framework zur Reduzierung von technischem Boilerplate Code verwendet. Darüber hinaus wurde die Java Persistence API (JPA) in Kombination mit Hibernate für die Persistenz eingesetzt.

#### jMolecules Annotations mit ArchUnit und ohne Spring

Die Einbindung von jMolecules und ArchUnit verlief problemlos über das Build-Tool Maven. Zusätzlich wurden die jMolecules Integrations zur automatischen Übersetzung der jMolecules Annotationen in die JPA und Hibernate Äquivalente mittels Codegeneration verwendet. Auf diese Weise lassen sich Redundanzen in den Annotationen vermeiden und Verwechslungen bzw. Missverständnisse vorbeugen. Eine genaue Auflistung der verschiedenen Übersetzungsmöglichkeiten ist im Anhang A in der Tabelle A.1 zu finden.
Bei der Einbindung der jMolecules Integrations kam es zu dem Problem, dass die automatische Codegenerierung beim Ausführen des Programms nicht funktioniert hat. Gelöst werden konnte dies mit der Unterstützung von Herrn Oliver Drotbohm, einem der Hauptentwickler des jMolecules Frameworks. Zur Codegenerierung wird im Hintergrund

das ByteBuddy Plugin genutzt. Es gibt jedoch einen Bug der bei Verwendung der IntelliJ Entwicklungsumgebung dafür sorgt, dass nach jeder Änderung vom Code ein manueller Build-Vorgang der Anwendung über die Kommandozeile vorgenommen werden muss. Dies geschieht über den Befehl „mvn clean install“. Die Anweisung „clean“ innerhalb des Befehls ist hier besonders wichtig, da hiermit das Projekt bereinigt und alle Dateien, die vom vorherigen Build erzeugt wurden, entfernt werden.
Die jMolecules Integrations stellen, über die Möglichkeit der Codegenerierung hinaus, eine Reihe von ArchUnit Designregeln zur Verifizierung der Architektur, in Bezug auf Domain-Driven Design, bereit. Diese lassen sich folgendermaßen verwenden:

```
1 @AnalyzeClasses(packages = {"example.casestudymtbshop.standard.
      annotations_archunit"})
2 public class ArchitectureTests {
3     @ArchTest public static final ArchRule ddd = JMoleculesDddRules.all();
4 }
```

Listing 4.1: ArchUnit Test der DDD Designregeln

Die Ausführung dieser Designregeln geschieht nach demselben Prinzip, nach dem Unit Tests, z. B. mit JUnit, ausgeführt werden. Sollte eine der Regeln verletzt werden, schlägt der Test fehl.

### jMolecules Types mit jQAssistant und ohne Spring

Wie bereits bei den Annotationen verlief die Einbindung der typbasierten Variante von jMolecules problemlos. Da die Markierung von Bounded Contexts und Subdomains, in Form von DDD-Modules, nur über Package-Annotationen möglich ist, wurden diese bei allen typbasierten Umsetzungen verwendet.
Im Vergleich zu den Annotationen, ermöglichen die Interfaces mehr Informationen über die Architektur zu hinterlegen. Dies geschieht mit der Hilfe von generischen Datentypen in Java. Auf diese Weise lässt sich nicht nur festhalten, dass eine Klasse zum Beispiel eine Entity implementiert, sondern zu welchem Aggregate Root diese Entity gehört und über welchen Typen diese Entity identifiziert wird.
Die jMolecules Integrations benötigen hier, wie bei allen Varianten, ebenfalls ein manuelles Kompilieren nach jeder Codeänderung, jedoch funktioniert die eigentliche Übersetzung der jMolecules Konzepte und die damit verbundene Codegenerierung einwandfrei.
Die Einbindung des jQAssistant Plugins erwies sich als problematisch, da in diesem Projekt JPA 3 und Hibernate 6 zum Einsatz gekommen sind. Beim Ausführen des Plugins

kam es zu einer NullpointerException während des Scanvorgangs mit dem jQAssistant die Programmstruktur analysiert um diese mittels der, im Plugin integrierten, Graphdatenbank abzubilden. Nach kurzer Recherche stellte sich heraus, dass die Exception stets beim Scan der „persistence.xml“ geworfen wurde. Die „persistence.xml“ dient zur Konfiguration von JPA, hier werden z. B. die Informationen hinterlegt, die benötigt werden um eine Verbindung zur Datenbank aufzubauen. Weitere Recherchen ergaben, dass das jQAssistant Plugin zum aktuellen Zeitpunkt nicht Kompatibel mit JPA 3 und der dazugehörigen „persistence.xml“ der neuen Version ist, was wiederum zu der Exception im Scan geführt hat. Gelöst wurde dieses Problem durch ein Downgrade auf JPA 2 und Hibernate 5. Dieses Downgrade führte zu einem neuen Problem, welches sich auf Hibernate 5 in Verbindung mit den Value Objects bezieht, die genutzt werden, um die Identität einer Entity zu bestimmen. Hier hat der Datentyp „UUID“, der zu Beginn verwendet wurde um die Id der Entities zu speichern, zu einem Problem mit der Persistenz geführt. Dieses Problem trat beim Mapping der Java-Datentypen mit den passenden Datenbank-Datentypen in der Form auf, dass es zwar möglich war Entities zu persistieren, es jedoch zu Exceptions beim Löschen der Entities gekommen ist. Auslöser hierfür war, das diese nicht anhand ihrer Id wiedergefunden wurden. Eine Änderung des Datentyps in den Value Objects von „UUID“ zu „String“ hat dieses Problem behoben.
Unabhängig der anfänglichen Probleme mit der Einbindung, verlief die Verwendung von jQAssistant mit den, von den jMolecules Integrations mitgelieferten, Designregeln problemlos.

### Traditionell mit Spring

Bei der Umsetzung der traditionellen Version unter Einsatz des Spring Frameworks kam es zu keinen Problemen. Durch die Verwendung der Spring-Repositories unterscheiden sich die Versionen mit Spring von denen ohne Spring darin, dass das Package „Infrastructure“ mit den Implementationen der Repository-Interfaces wegfällt, da der Spring Container dies übernimmt. Die Spring-Repositories liefern bereits vorgefertigte Methoden zum Speichern, Löschen und Auffinden von Entities anhand ihrer Ids oder Instanzen mit. Speziellere Abfragen, wie zum Beispiel die Abfrage aller Bestellungen eines Kunden werden hier über Query Annotationen der Methoden des Interfaces festgelegt. Diese Abfragen werden entweder mittels der Java Persistence Query Language (JPQL) oder wahlweise nativem SQL definiert, wie im folgenden Codeausschnitt zu sehen:

```
1 public interface Orders extends JpaRepository<Order, Order.OrderId> {
2     @Query("SELECT o FROM Order o WHERE o.customerId = :#{#customer.id}")
3     List<Order> findOrdersByCustomer(@Param("customer")Customer customer);
4
5     @Query("SELECT m FROM Mountainbike m WHERE m.sold = false")
6     List<Mountainbike> openCatalogue();
7 }
```

Listing 4.2: Order Repository mit Spring

### jMolecules Annotations mit jQAssistant und Spring

Unter Berücksichtigung der Erfahrungen aus den vorherigen Versionen kam es bei der Umsetzung mit den jMolecules Annotationen in Kombination mit jQAssistant und dem Spring Framework zu keinen Problemen. Die Integration der verschiedenen Technologien funktionierte einwandfrei.

### jMolecules Types mit ArchUnit und Spring

Die Umsetzung der typbasierten Variante von jMolecules in Kombination mit ArchUnit und dem Spring-Framework verlief weitestgehend problemlos.
Da bei dieser Variante aggregateübergreifende Referenzen zwischen Aggregate Roots mithilfe des generischen Typs Association realisiert werden, kam es zu kleineren Komplikationen bei der Umsetzung der Queries in den Repositories. Veranschaulicht werden kann dies am Beispiel des Repositories Orders und der Aggregate Root Order. Die Referenz auf den Customer innerhalb einer Order geschieht hier über das Feld „customer“ des Typs Association<Customer, CustomerId>. Wird hier nun im Repository Orders die JPQL-Query zur Methode „findOrdersByCustomer(Customer customer)“ definiert. Wie bereits im Abschnitt zur traditionellen Version mit Spring zu sehen ist, kommt es zu einer Diskrepanz zwischen den Datentypen der Id des, in der Methode als Parameter übergebenen, Customers und des Datentyps der Referenz auf diesen in der Order. Eine Association kann nicht mit einer Id verglichen werden, um die Orders des Customers zu finden.
Hier ist die Lösung die Verwendung einer nativen SQL-Abfrage, mit der direkt auf den Id-String in der Datenbank zugegriffen wird anstatt auf die Association im Order Objekt wie bei der objektorientierten Abfrage in JPQL. Dieser String wird wiederum mit dem

String innerhalb des Value Objects der Id des Customers verglichen, wie im Folgenden zu sehen ist:

```
1  public interface Orders extends Repository<Order, Order.OrderId>,
       AssociationResolver<Order, Order.OrderId> {
2
3      ...
4
5      @Query(value = "SELECT * FROM sample_order o WHERE o.customer = :#{#
           customer.id.customerId}",
6              nativeQuery = true)
7      List<Order> findOrdersByCustomer(@Param("customer")Customer customer);
8
9      ...
10 }
```

Listing 4.3: Query im Order Repository mit nativem SQL

### Automatische Generierung der Architekturdokumentation

Bei der Umsetzung der automatischen Generierung der Architekturdokumentation kam es zu diversen Problemen die letztendlich dazu geführt haben, dass diese nicht überprüft werden kann. Im jMolecules Projekt wird auf ein Repository zu einem Tool namens Moduliths [2] verwiesen.
Dieses Tool wurde im Laufe des Jahres in das Spring-Framework als experimentelles Feature integriert. Zur Verwendung der noch verfügbaren Version wird das Spring-Framework benötigt. Aus diesem Grund lassen sich für die Versionen der Fallstudie ohne Spring keine automatischen Dokumentationen generieren. Darüber hinaus kam es während der Einbindung des Tools bei den Umsetzungen mit Spring zu Kompatibilitätsproblemen einer Klasse der Moduliths-Bibliothek, die im Zeitrahmen dieser Arbeit nicht gelöst werden konnte. Es trat folgende Fehlermeldung auf:

```
1  java: cannot access org.springframework.modulith.docs.Documenter
2    bad class file: /home/kai/.m2/repository/org/springframework/experimental/
         spring-modulith-docs/0.1.0-SNAPSHOT/spring-modulith-docs-0.1.0-SNAPSHOT
         .jar!/org/springframework/modulith/docs/Documenter.class
3      class file has wrong version 61.0, should be 55.0
4      Please remove or make sure it appears in the correct subdirectory of the
           classpath.
```

Listing 4.4: Exception beim Generieren der Architekturdokumentation

# 5 Ergebnisse

Nachdem die Anforderungen definiert und die Fallstudie durchgeführt wurde, können die Ergebnisse ausgewertet werden. Hierzu wird die Fallstudie anhand der verschiedenen Anforderungen aus Kapitel 3 analysiert und bewertet.
Die traditionellen Umsetzungen mit und ohne dem Spring-Framework werden nicht einzeln betrachtet, sondern ausschließlich zu Vergleichszwecken herangezogen.
Aufgrund der, in Kapitel 4 Abschnitt 4.2.3, bereits behandelten Problematik mit der automatischen Generierung von Architekturdokumentationen, wird die Anforderung A.3 im Weiteren nicht weiter evaluiert und als nicht erfüllt betrachtet.

## 5.1 Umsetzungen ohne Spring

### 5.1.1 jMolecules Annotations mit ArchUnit

#### A.1 DDD Konzepte explizit im Code abbilden

Die Abbildung der DDD Konzepte mittels Annotationen, sofern vorhanden, hat in der Umsetzung einwandfrei funktioniert.
Wie der Tabelle 5.1 entnommen werden kann, bietet das jMolecules Framework über Annotationen die Möglichkeit Aggregate Roots sowie Entities mit ihren Identitäten und Value Objects abzubilden. Jedoch fehlen Informationen zu den Aggregate-Grenzen im Allgemeinen. Darüber hinaus lässt sich nicht abbilden, zu welchem Aggregate Root eine Entity oder zu welcher Entity ein Value Object gehört. Hier wäre für Aggregates eine Package Annotation wünschenswert. Die Informationen zu den Aggregate Roots der Entities und Entities der Value Objects könnten den Annotationen über Parameterwerte mitgeliefert werden.
Für Bounded Contexts gibt eine eigene Package Annotation. Das Gleiche gilt für Modules.
Für Subdomains gibt es keine eigene Annotation. Es bietet sich lediglich die Möglichkeit

| Nr. | Anforderung | Ergebnis |
|---|---|---|
| A.1.1 | Aggregates | Nicht vorhanden |
| A.1.2 | Aggregate Roots | Vorhanden |
| A.1.3 | Entities | Vorhanden |
| A.1.4 | Entity Identitäten | Vorhanden |
| A.1.5 | Mit der Entity verbundenen Aggregate Root | Nicht vorhanden |
| A.1.6 | Value Objetcs | Vorhanden |
| A.1.7 | Mit dem Value Object verbundene Entity | Nicht vorhanden |
| A.1.8 | Bounded Contexts | Vorhanden(Package Annotation) |
| A.1.9 | Subdomain allgemein | Nur über Module (siehe A.1.13) |
| A.1.10 | Core Subdomain | Nicht vorhanden |
| A1.1.11 | Supporting Subdomain | Nicht vorhanden |
| A.1.12 | Generic Subdomain | Nicht vorhanden |
| A.1.13 | Module | Vorhanden (Package Annotation) |
| A.1.14 (Zusatz) | Events, Repository, Factory, Service | Vorhanden |

Tabelle 5.1: jMolecules Annotationen Ergebnisse Anforderung A.1

Subdomains über die Module Annotation zu kennzeichnen, dies entspricht der Handlungsempfehlung in [11, S. 47] diese als Modules zu modellieren, im Falle von mehreren Subdomains innerhalb eines Bounded Contexts. Über eigene Package-Annotationen wäre jedoch eine präzisere Kennzeichnung von Subdomains sowie deren speziellen Ausprägungen möglich gewesen.
Die Zusatzanforderung A.1.14 erfüllt das jMolecules Framework vollständig. Es ist sowohl möglich Repositories, Factories und Services über Annotationen abzubilden, als auch den Einsatz von Events inklusive Event Handlern und Event Publishern.

### A.2 DDD Desingregeln verifizieren

Bei der Auswertung der mitgelieferten ArchUnit Designregeln kam es zu folgenden Ergebnissen:

| **Anforderungs- / Designregel-Kategorie** | **Nr.** | **Ergebnis** |
|---|---|---|
| A.2.1 Bounded Contexts | A.2.1.1 | Nicht erfüllt |
| A.2.2 Aggregates | A.2.2.1 | Nicht erfüllt |
| | A.2.2.2 | Erfüllt |
| | A.2.2.3 | Nicht erfüllt |
| A.2.3 Entities | A.2.3.1 | Nicht erfüllt |
| | A.2.3.2 | Erfüllt |
| | A.2.3.3 | Nicht erfüllt |
| | A.2.3.4 | Nicht erfüllt |
| A.2.4 Value Objects | A.2.4.1 | Nicht erfüllt |
| | A.2.4.2 | Erfüllt |
| | A.2.4.3 | Nicht erfüllt |
| | A.2.4.4 | Nicht erfüllt |

Tabelle 5.2: jMolecules Annotations mit ArchUnit Designregeln Ergebnisse Anforderung A.2

Im Zusammenspiel mit den jMolecules Annotations und den ArchUnit Designregeln werden nur insgesamt drei der zwölf Designregeln erfüllt. Es wird geprüft, ob Referenzen von Aggregate Roots auf andere Aggregate Roots nur über die Identität erfolgen, Entitites eine Identität besitzen und Value Objects keine Identitäten besitzen.

### A.4 Boilerplate Code reduzieren

Zu erwähnen ist bei der automatischen Generierung von Boilerplate Code die Problematik, wie in der Fallstudie bereits beschrieben, bezüglich des Bugs im ByteBuddy Plugin. Durch diesen ist es nötig nach jeder Codeänderung einen manuellen Buildvorgang über Maven durchzuführen. Diese Problematik bezieht sich auf sämtliche Umsetzungen.
Die vorhandenen Möglichkeiten zur automatischen Generierung von technischem Boilerplate Code sind im Anhang A in Tabelle A.1 zu finden. Zur Veranschaulichung dieser Funktion wird im Folgenden die traditionelle Umsetzung in Listing 5.1 herangezogen.

```
1  @Entity
2  @Getter
3  @Table(name = "SampleOrder")
4  public class Order {
5
6      @EmbeddedId
7      private final OrderId id;
8      private LocalDate orderDate;
9      private Customer.CustomerId customerId;
10     @OneToMany(cascade = CascadeType.ALL)
11     private List<Mountainbike> orderedBikes;
12
13     ...
14
15     @Value
16     @Embeddable
17     @RequiredArgsConstructor(staticName = "of")
18     @NoArgsConstructor(force = true)
19     public static class OrderId implements Serializable {
20         private static final long serialVersionUID = -8782038556781615559L;
21         String orderId;
22     }
23
24     ...
25 }
```

Listing 5.1: Order Klasse traditionell ohne Spring

Zu sehen ist hier die Klasse Order aus der traditionellen Umsetzung ohne Spring. Die verwendeten Annotationen stammen sowohl von der JPA (Entity, Table, EmbeddedId, OneToMany und Embedabble) als auch vom Lombok Framework (Getter, Value, RequiredArgsConstructor und NoArgsConstructor).
Dem gegenüber steht die Order Klasse unter Verwendung der jMolecules Annotationen in Kombination mit der automatischen Boilerplate Code Generierung in Listing 5.2.

```
1  @AggregateRoot
2  @Getter
3  @Table(name = "SampleOrder")
4  public class Order {
5
6      @Identity
7      private final OrderId id;
8      private LocalDate orderDate;
9      @Association(aggregateType = Customer.class)
10     private Customer.CustomerId customerId;
11     private List<Mountainbike> orderedBikes;
12
13     ...
14
15     @ValueObject
16     @Value(staticConstructor = "of")
17     public static class OrderId implements Serializable{
18         private static final long serialVersionUID = -4782250954947621 83L;
19         String orderId;
20     }
21
22     ...
23 }
```

Listing 5.2: Order Klasse jMolecules Annotations ohne Spring

Die Annotationen der JPA werden hier durch die jMolecules Annotationen ersetzt, während die passenden JPA Annotationen über die Code Generierung nicht sichtbar im Hintergrund automatisch wieder hinzugefügt werden. Übrig bleibt ausschließlich die Table Annotation, welche jedoch nur genutzt wird, weil es durch den Klassennamen Order zu Komplikationen mit dem verwendeten MySQL Server kommt. Bei Order handelt es sich ein Keyword des Servers, welches nicht als Tabellenname verwendet werden darf. Die Liste der Mountainbikes wird automatisch, auch ohne Annotation, mit der passenden OneToMany Annotation versehen.
Darüber hinaus kann auf die Lombok Annotationen NoArgsConstructor und RequiredArgsConstructor verzichtet werden.

### 5.1.2 jMolecules Types mit jQAssistant

#### A.1 DDD Konzepte explizit im Code abbilden

Wie bereits bei den Annotationen, hat die Umsetzung mit den vorhandenen Marker Interfaces problemlos funktioniert.

| Nr. | Anforderung | Ergebnis |
|---|---|---|
| A.1.1 | Aggregates | Nicht vorhanden |
| A.1.2 | Aggregate Roots | Vorhanden |
| A.1.3 | Entities | Vorhanden |
| A.1.4 | Entity Identitäten | Vorhanden |
| A.1.5 | Mit der Entity verbundenen Aggregate Root | Vorhanden |
| A.1.6 | Value Objetcs | Vorhanden |
| A.1.7 | Mit dem Value Object verbundene Entity | Vorhanden |
| A.1.8 | Bounded Contexts | Nicht vorhanden |
| A.1.9 | Subdomain allgemein | Nicht vorhanden |
| A.1.10 | Core Subdomain | Nicht vorhanden |
| A.1.11 | Supporting Subdomain | Nicht vorhanden |
| A.1.12 | Generic Subdomain | Nicht vorhanden |
| A.1.13 | Module | Nicht vorhanden |
| A.1.14 (Zusatz) | Events, Repository, Factory, Service | Teils vorhanden |

Tabelle 5.3: jMolecules Typen Ergebnisse Anforderung A.1

Es ist möglich über die jMolecules Typen Aggregate Roots, Entities und Value Objects abzubilden, jedoch keine Aggregates. Da es in Java keine Package Interfaces gibt, ist hier keine akkurate Möglichkeit gegeben Aggregates sinnvoll über Typen abzubilden. Selbiges gilt für die fehlenden Möglichkeiten zur Abbildung von Bounded Contexts, Subdomains inklusive ihrer Ausprägungen sowie Modules. Am Sinnvollsten ist es hier eine Kombination aus dem typbasierten Ansatz und den Annotationen zu wählen.
Im Gegensatz zu den Annotationen bieten die Typen die Möglichkeit zu definieren, zu welchen Aggregate Root eine Entity gehört beziehungsweise zu welcher Entity ein Value Object gehört. Darüber hinaus wird über den Typen der Datentyp der verwendeten

Identität abgebildet. Beides ist mittels generischer Datentypen umgesetzt worden, dessen Parameter die besagten Informationen enthalten.
Die Zusatzanforderung A.1.14 erfüllt der typbasierte Ansatz teilweise. Es werden zwar keine Interfaces für Factories und Services mitgeliefert, jedoch ein Repository Interface, welches als generischer Datentyp weitere Informationen enthält. Hier wird die Klasse des Aggregate Roots und dessen Datentyp zur Identitätsbestimmung festgehalten. Es gibt darüber hinaus ein Interface für Events, jedoch nicht für Event Handler oder Event Publisher.
Zusätzlich liefert der typbasierte Ansatz ein Interface namens Association. Dieses Interface dient dazu, Aggregate übergreifende Referenzen von Aggregate Root zu Aggregate Root zu definieren. Bei Association handelt es sich ebenfalls um einen generischen Datentypen über dessen Parameter die Klasse des referenzierten Aggregate Root sowie der Datentyp des Identifiers festgehalten wird. Das Interface Association enthält eine statische Methode namens „forAggregate(T aggregate)“. Diese Methode erzeugt eine Instanz der Klasse SimpleAssociation, in der der Typ der referenzierten Aggregate Root sowie ihre Identität hinterlegt ist.

### A.2 DDD Desingregeln verifizieren

Bei der Variante ohne Spring in Kombination mit jQAssistant ist es bei der Umsetzung zu der, in der Fallstudie bereits beschriebenen, Problematik bezüglich des Plugins und der JPA Version 3 gekommen. Diese konnte nur durch ein Downgrade auf JPA 2 behoben werden. Über die mitgelieferten Designregeln des jQAssistant Plugins lassen sich insgesamt vier der zwölf Designregeln verifizieren. Hier ist es möglich die Beziehungen zwischen Bounded Contexts auf ungewünschtes Verhalten zu überprüfen, sowie Entity Identitäten und Value Objects auf ihre Unveränderbarkeit zu prüfen. Darüber hinaus lässt sich ebenfalls prüfen, ob Value Objects verbotenerweise auf Entities oder Aggregate Roots referenzieren. Die beiden Anforderungen 2.3.4 und 2.4.4 welche besagen, dass Entities und Value Objects nur innerhalb des Aggregates referenziert werden dürfen, in dem sie definiert sind, wird nicht vollständig überprüft. Es ist jeweils eine Regel vorhanden die prüft, ob ein Typ nur innerhalb des Bounded Contexts sowie nur innerhalb des Modules referenziert wird, in dem dieser definiert ist.

| Anforderungs- / Designregel-Kategorie | Nr. | Ergebnisse |
|---|---|---|
| A.2.1 Bounded Contexts | A.2.1.1 | Erfüllt |
| A.2.2 Aggregates | A.2.2.1 | Nicht erfüllt |
| | A.2.2.2 | Nicht erfüllt |
| | A.2.2.3 | Nicht erfüllt |
| A.2.3 Entities | A.2.3.1 | Nicht erfüllt |
| | A.2.3.2 | Nicht erfüllt |
| | A.2.3.3 | Erfüllt |
| | A.2.3.4 | Teils erfüllt |
| A.2.4 Value Objects | A.2.4.1 | Erfüllt |
| | A.2.4.2 | Nicht erfüllt |
| | A.2.4.3 | Erfüllt |
| | A.2.4.4 | Teils erfüllt |

Tabelle 5.4: jMolecules Types mit jQAssistant Ergebnisse Anforderung A.2

### A.4 Boilerplate Code reduzieren

Wie bei den Annotationen, funktioniert die automatische Generierung ebenfalls mit der typbasierten Variante, wie der Tabelle A.1 im Anhang A zu entnehmen ist.
Zur Veranschaulichung wird wieder die Klasse Order herangezogen.

```
1  @Getter
2  @Table(name = "SampleOrder")
3  public class Order implements AggregateRoot<Order, Order.OrderId> {
4
5      private final OrderId id;
6      private LocalDate orderDate;
7      private Association<Customer, Customer.CustomerId> customer;
8      private List<Mountainbike> orderedBikes;
9
10     ...
11
12     @Value(staticConstructor = "of")
13     public static class OrderId implements Identifier {
14         String orderId;
15     }
16
17    ...
18 }
```

Listing 5.3: Order Klasse jMolecules Types ohne Spring

Im direkten Vergleich zur annotationsbasierten Variante lassen sich hier die Designelemente auf schlankere Weise in gleichem Detailgrad abbilden und automatisch in die Konzepte der JPA übersetzen. Die Informationen bezüglich der Identität werden über den generischen Datentypen AggregateRoot definiert, wodurch das Feld id nicht explizit markiert werden muss. Die Referenz auf den Customer der Order wird als Association abgebildet und die eingebettete Klasse des Value Objects OrderId wird mittels der Implementation des Identifier Interfaces mit den nötigen Konzepten zur Persistenz versehen, zu denen die Implementation von Serializable gehört.

## 5.2 Umsetzungen mit Spring

An dieser Stelle ist zu erwähnen, dass der bloße Einsatz des Spring-Frameworks bereits einen positiven Einfluss auf die Erreichung der Ziele des Domain-Driven Design nimmt. Durch den Einsatz von Spring Repositories fallen die Implementationen der DDD Repositories weg, da diese im Hintergrund über den Spring Container automatisiert umgesetzt werden. Bei den Implementationen der Repositories handelt es sich um rein technischen Code, welcher ausschließlich Persistenzzwecken dient. Der Wegfall dieses Codes unterstützt damit die Erreichung des Ziels, dass die Software ein möglichst genaues Abbild des Domänen-Modells widerspiegeln soll.

### 5.2.1 jMolecules Annotations mit jQAssistant

#### A.1 DDD Konzepte explizit im Code abbilden

Da die Möglichkeiten zur Abbildung der DDD Konzepte über Annotationen bereits ausführlich evaluiert wurden, werden diese hier nicht weiter ausgeführt sondern lediglich auf Abschnitt 5.1.1 verwiesen.

#### A.2 DDD Desingregeln verifizieren

Die Ergebnisse in Bezug auf die Verifizierung von Designregeln mit jQAssistant und Annotationen unterscheiden sich nicht von denen mit der jQAssistant und der typbasierten Variante von jMolecules. Aus diesem Grund wird hier auf Tabelle 5.4 verwiesen.
Bei der Umsetzung mit Spring ist es jedoch nicht zu Kompatibilitätsproblemen mit der

JPA Version 3 gekommen, da es hier keine „persistence.xml“ Datei gibt und somit die, in der Fallstudie beschriebene, Exception nicht aufgekommen ist. Aus diesem Grund wird empfohlen, beim Einsatz von jQAssistant ebenfalls das Spring Framework für die Persistenz zu verwenden.

#### A.4 Boilerplate Code reduzieren

Bei den Annotationen in Kombination mit dem Spring Framework kommt es in Bezug auf die Generierung von technischem Boilerplate Code nur zu kleinen Unterschieden. Die jMolecules Factory, Service und Repository Annotationen werden zu den Spring Component, Service sowie Repository Annotationen übersetzt. Es ist jedoch weiterhin nötig manuell das entsprechende Spring Repository Interface zu implementieren, da der Annotation die Informationen bezüglich des verwendeten Aggregate Roots und dessen Identifier fehlen. Hier wäre es empfehlenswert, wenn der Repository Annotation diese Werte mittels Parameter übergeben werden könnten.

```
1 @Repository
2 public interface Orders extends JpaRepository<Order, Order.OrderId> {
3     ...
4 }
```

Listing 5.4: Orders Repository jMolecules Annotations mit Spring

### 5.2.2 jMolecules Types mit ArchUnit

#### A.1 DDD Konzepte explizit im Code abbilden

Wie bereits bei den Annotationen wird auf den vorherigen Abschnitt, in diesem Fall 5.1.2, verwiesen. Der Einsatz des Spring Frameworks führt zu keinen Unterschieden bezüglich der Abbildung von DDD Konzepten im Code.

#### A.2 DDD Desingregeln verifizieren

Das Ergebnis der Kombination des typbasierten Ansatzes mit den mitgelieferten Archunit Designregeln weicht leicht ab von dem der Annotationen mit Archunit. Über den typbasierten Ansatz wird zusätzlich geprüft, ob Entities nur innerhalb des Aggregates

| **Anforderungs- / Designregel-Kategorie** | **Nr.** | **Ergebnis** |
|---|---|---|
| A.2.1 Bounded Contexts | A.2.1.1 | Nicht erfüllt |
| A.2.2 Aggregates | A.2.2.1 | Nicht erfüllt |
| | A.2.2.2 | Erfüllt |
| | A.2.2.3 | Nicht erfüllt |
| A.2.3 Entities | A.2.3.1 | Nicht erfüllt |
| | A.2.3.2 | Erfüllt |
| | A.2.3.3 | Nicht erfüllt |
| | A.2.3.4 | Erfüllt |
| A.2.4 Value Objects | A.2.4.1 | Nicht erfüllt |
| | A.2.4.2 | Erfüllt |
| | A.2.4.3 | Nicht erfüllt |
| | A.2.4.4 | Nicht erfüllt |

Tabelle 5.5: jMolecules Types mit ArchUnit Designregeln Ergebnisse Anforderung A.2

referenziert werden, in dem sie definiert sind. Umgesetzt wird dies über die generischen Datentypen.

### A.4 Boilerplate Code reduzieren

Die typbasierte Methode unterscheidet sich in Kombination mit dem Spring Framework nicht von der Version ohne Spring, da das jMolecules Repository Interface durch die Generierung des Boilerplate Codes automatisch in das passende Spring Data Repository übersetzt wird. Dies hebt die typbasierte Version von der annotationsbasierten Variante ab, da hier bereits die nötigen Informationen für das Spring Framework durch den generischen Datentypen bereitgestellt werden. Gleichzeitig sind hier jedoch keine Interfaces für Services und Factories vorhanden.

```
1 public interface Orders extends Repository<Order, Order.OrderId> {
2     ...
3 }
```

Listing 5.5: Orders Repository jMolecules Types mit Spring

# 6 Schluss

Zum Abschluss wird in diesem Kapitel zunächst der Verlauf sowie die Ziele dieser Thesis reflektiert und ein Fazit der Ergebnisse gezogen. Des Weiteren wird ein kurzer Ausblick inklusive Handlungsempfehlungen und Möglichkeiten zum weiteren Vorgehen gegeben.

## 6.1 Fazit

Im Rahmen dieser Arbeit wurde das jMolecules Framework analysiert um vorrangig zu evaluieren, ob es geeignet ist, sowohl die architektonischen Konzepte des Domain-Driven Designs innerhalb des Codes zu abstrahieren als auch die korrekte Verwendung dieser Konzepte und ihrer Designregeln zu verifizieren. Darüber hinaus wurden ebenfalls die Funktionen des Frameworks zur automatischen Generierung von Architekturdokumentation sowie technischem Boilerplate Code bewertet.
Hierzu wurden zunächst die vier Hauptziele des Frameworks analysiert und basierend auf Fachliteratur Anforderungen zur Einhaltung dieser Ziele festgelegt. Anschließend wurde eine Fallstudie durchgeführt auf deren Grundlage die Erfüllung der, zuvor definierten, Anforderungen und somit die Einhaltung der Ziele bewertet werden konnte.
Zur Einhaltung der Ziele kann abschließend festgehalten werden:

**1. DDD Konzepte explizit im Code abbilden:**

Sowohl die annotationsbasierte Variante als auch die typbasierte Variante konnte einzeln betrachtet nicht den gesamten Anforderungskatalog an Designkonzepten im Code abbilden. Es empfiehlt sich hier eine Kombination aus beiden Varianten zu wählen, wobei die typbasierte Variante aufgrund des höheren Informationsgehalts der generischen Datentypen hier zu bevorzugen und die annotationsbasierte Variante zum Ausgleichen der nicht vorhandenen Designkonzepte zu nutzen ist. Beide Varianten bieten jedoch nicht die Möglichkeit Aggregates und Subdomains inklusive ihrer Ausprägungen explizit abzubilden.
Dieses Ziel wird gilt somit größtenteils als erfüllt.

**2. DDD Designregeln verifizieren:**

An dieser Stelle ist noch einmal die Problematik bezüglich der Kompatibilität im Zusammenspiel zwischen dem jQAssistant Plugin und der Java Persistence API (JPA) Version 3 zu erwähnen. Sowohl die ArchUnit Bibliothek als auch das jQAssistant Plugin erfüllt über die mitgelieferten Designregeln, jeweils nur ein Drittel der Designregeln ganz oder teilweise. Da es hierbei zu keinen Überschneidungen kommt, wird ebenfalls für ein bestmögliches Ergebnis eine Kombination aus beiden Technologien empfohlen. Zur Vermeidung von Kompatibilitätsprobleme mit jQAssistant und der JPA wird darüber hinaus der Einsatz des Spring Frameworks empfohlen.

Aufgrund der maximalen Erreichung von zwei Dritteln der Anforderungen gilt dieses Ziel als Teilweise erfüllt.

**3. Automatische Generierung von Architekturdokumentation:**

Wie bereits in den Ergebnissen festgehalten, wird diese Anforderungen aufgrund der, im Rahmen dieser Arbeit, nicht behebbaren Probleme mit der Einbindung der Moduliths Bibliothek als nicht erfüllt angesehen.

**4. Reduzierung von technischem Boilerplate Code:**

Die in der Fallstudie beschriebene Problematik bezüglich des Bugs des ByteBuddy Plugins, welcher zur Folge hat, dass nach jeder Codeänderung ein manueller Build Vorgang notwendig ist, führt zwar zu gewissen zu Einschränkungen des Nutzungskomforts, wirkt sich jedoch nicht auf die eigentliche Funktionalität der Generierung von technischem Boilerplate Code aus. Die Generierung hat im Test einwandfrei funktioniert. Sowohl in Kombination mit der annotationsbasierten als auch mit der typbasierten Variante konnte eine deutliche Reduzierung an Boilerplate Code und damit übersichtlicherer Code erzielt werden. Daraus resultiert zusätzlich ein geringerer Model-Code-Gap. Besonders in Kombination mit der typbasierten Variante ließen sich sehr gute Ergebnisse erzielen.

Der Einsatz des Spring Frameworks unterstützt die Erfüllung dieses Ziels ebenfalls aufgrund der wegfallenden Implementationen der Repositories, sofern diese im Projekt zur Anwendung kommen.

Dieses Ziel gilt somit als vollständig erfüllt.

Alles in allem lässt sich somit sagen, dass er Einsatz des jMolecules Framework zur architektonischen Abstraktion und Verifikation im Kontext des Domain-Driven Design, trotz vorhandener Schwächen, einen durchaus signifikanten Mehrwert für die Softwareentwick-

lung bietet. Lohnenswert ist ebenso der Einsatz des Spring Frameworks in Kombination mit jMolecules, da viele der Funktionen bereits hierauf ausgelegt sind. Lediglich zur Dokumentation sollten hier Alternativen in Betracht gezogen werden.

## 6.2 Ausblick

In Bezug auf die vier Hauptziele dieser Arbeit lassen sich folgende Ausblicke geben:

**1. DDD Konzepte explizit im Code abbilden:**
Auch wenn die Anforderungen an die Abbildung der Designkonzepte im Code zum größten Teil erfüllt wurden, würden sich weitere Package Annotationen für Aggregates und Subdomains inklusive ihrer Ausprägungen als nützlich erweisen. Auf diese Weise könnten sämtliche Designkonzepte hinreichend im Code abgebildet werden. Da es sich beim jMolecules Framework um ein Open-Source-Projekt handelt, könnten diese wahrscheinlich in Eigenleistung eingebaut werden.

**2. DDD Designregeln verifizieren:**
Im Rahmen dieser Arbeit war es nicht möglich, die Optionen zur Erzeugung eigener ArchUnit Tests sowie jQAssistant Concepts und Constraints mittels Cypher-Queries genauer zu betrachten. Hierdurch könnten möglicherweise die nicht erfüllten Anforderungen bezüglich der Designregeln noch umgesetzt werden. Eine weitere Betrachtung scheint empfehlenswert.

**3. Automatische Generierung von Architekturdokumentation:**
Zur Lösung der Kompatibilitätsprobleme mit der Documenter Klasse würde es sich empfehlen Kontakt zu Herrn Oliver Drotbohm, dem Entwickler der Moduliths Bibliothek, oder dem Team des Spring Frameworks aufzunehmen, zu dem Herr Drotbohm ebenfalls gehört.

**4. Reduzierung von technischem Boilerplate Code:**
Weitere Recherchen bezüglich des ByteBuddy Bugs führen möglicherweise zu einer besseren Lösung als einen manuellen Build Vorgang nach jeder Codeänderung durchzuführen.

Über den Rahmen dieser Bachelorarbeit hinaus bietet das xMolecules Projekt neben den jMolecules für Java noch Repositories für die Sprachen PHP und C# dessen Funktionalitäten jedoch zum aktuellen Zeitpunkt deutlich eingeschränkter sind. Eine weitere

Beobachtung dieser Teile des Projekts könnte vielversprechend sein.
Des Weiteren bietet das jMolecules Framework für Java noch weitere Funktionalitäten zur Abstraktion und Verifizierung von Architekturstilen, unabhängig von Domain-Driven Design, auf die in dieser Arbeit nicht tiefergehend eingegangen wurde.

# Literaturverzeichnis


[1] : *jMolecules – Architectural abstractions for Java*. Juni 2022. – URL https://github.com/xmolecules/jmolecules

[2] : *Moduliths - A playground to build technology supporting the development of modular monolithic (modulithic) Java applications*. Mai 2022. – URL https://github.com/moduliths/moduliths

[3] Brandolini, Alberto: *Introducing EventStorming*. URL https://leanpub.com/introducing_eventstorming, 2021

[4] Evans, Eric: *Domain-driven design*. Upper Saddle River, NJ [u.a.] : Addison-Wesley, 2004. – +++Achtung+++Hier auch später erschienene, unveränderte Nachdrucke!. – ISBN 9780321125217

[5] Fairbanks, George: *Just enough software architecture*. Marshall and Brainerd, 2010. – ISBN 9780984618101

[6] Mahler, Dirk: *jQAssistant - Verify Your Design And Architecture*. Mai 2016. – URL https://vimeo.com/170797227?signup=true

[7] Open-source: *ArchUnit User Guide*. November 2022. – URL https://www.archunit.org/userguide/html/000_Index.html

[8] Open-source: *Technology integration for jMolecules*. November 2022. – URL https://github.com/xmolecules/jmolecules-integrations

[9] Schwentner, Oliver Drotbohm; H.: *xMolecules - Architectural abstractions in code*. November 2022. – URL https://github.com/xmolecules

[10] Vernon, Vaughn: *Implementing domain-driven design*. Upper Saddle River, NJ [u.a.] : Addison-Wesley, 2013. – ISBN 9780321834577

[11] Vernon, Vaughn ; Lilienthal, Carola (Hrsg.) ; Schwentner, Henning (Hrsg.): *Domain-Driven Design kompakt*. 1. Auflage. Heidelberg : dpunkt.verlag, 2017. – Literaturverzeichnis: Seite 135-136. – ISBN 9783864904394

# A Anhang

**Tabelle A.1 aus Platzmangel auf der nächsten Seite zu finden.**

Tabelle A.1: Liste der automatischen Übersetzungen von jMolecules Konzepten in Konzepte anderer Technologien zur Integration.

| **Art** | **jMolecules** | **JPA / Hibernate / Spring** |
|---|---|---|
| Annotation | AggregateRoot | @Entity + Default Constructor |
| Annotation | Association | - |
| Annotation | BoundedContext | - |
| Annotation | Entity | @Entity + Default Constructor |
| Annotation | Factory | Spring @Component |
| Annotation | Identity | @Id, @EmbeddedId (Abhängig vom Feldtypen) |
| Annotation | Module | - |
| Annotation | Repository | Spring @Repository wenn Spring im Classpath vorhanden. Fehlende Informationen zu Typen. |
| Annotation | Service | Spring @Service |
| Annotation | ValueObject | @Embeddable + Default Constructor |
| Interface | AggregateRoot | @Entity + Default Constructor |
| Interface | Association (Verwendung als Feldtyp mit Generics) | AttributeConverter um Assoziation aufzulösen und und nur die Id zu persistieren. |
| Interface | AssociationResolver (Für Repositories) | Bereitstellung von API zur Auflösung von Association-Instanzen in ihr Zielaggregat. |
| Interface | Entity | @Entity + Default Constructor |
| Interface | Identifier | @Embeddable und implementierung vom interface Serializable. |
| Interface | Repository | Implementierung von Spring Data Repository Interface wenn Spring im Classpath vorhanden ist. |
| Interface | ValueObject | @Embeddable + Default Constructor |
| Feld | Nicht annotierte Referenz von Entity zu Entity | @OnoToOne |
| Feld | Nicht annotierte Referenz von Entity zu Entity über Collection. | @OneToMany |
| Feld | Nicht annotierte Referenz von Entity auf ValueObject. | - |
| Feld | Feld eines Typen, der Identifier implementiert. | @EmbeddedId |

# B Anhang

## B.1 Inhalte der beiliegenden CD

Auf der beiliegenden CD sind folgende Dateien und Verzeichnisse zu finden:

**\thesis-kai-meier.pdf**
Diese Thesis als PDF-Datei
**\CaseStudyMtbShop\case-study-mtb-shop-main.zip**
Gesamtes Projekt der Fallstudie als komprimierter Zip-Ordner. Dieses Projekt wurde mit dem Maven-Build-Tool in der IntelliJ-IDE erstellt.

## Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

| Ort | Datum | Unterschrift im Original |
| --- | --- | --- |