Aggregationsschnitt einer Checkout-Software auf Basis einer Hexagonalen Architektur mit Domain-Driven Design

Bachelor-Arbeit

Simon Thalmaier

Erstprüfer	-
Zweitprüfer	-
Betreuer	Sebastian Apel
Ausgabedatum	-
Abgabedatum	-

Angaben zum Autor oder Vergleichbares.

Dokumenteninformation, Veröffentlichung, Rahmen, bibliographisches Angaben

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Zusammenfassung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Danksagung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Inhaltsverzeichnis

Darstellungsverzeichnis

Listingsverzeichnis

Akronyme

CQRS	Command and Query Responsibility Segregation.
CRUD	Create Update Delete.
DDD	Domain-Driven Design.
DIP	Dependency-Inversion-Prinzip.
DTO	Data-Transfer-Object.
HTTP	Hypertext Transfer Protocol.
ISP	Interface-Segregation-Prinzip.
LSP	Liskovsches Substitutionsprinzip.
OCP	Open-Closed-Prinzip.
POC	Proof-of-Concept.
REST	Representational State Transfer.
SRP	Single-Responsibility-Prinzip.

Glossar

Boilerplate	Ein Teil einer Software, welcher viele Zeilen an Code einnimmt, obwohl dadurch nur wenig bis gar keine Funktion bereitgestellt wird.
Dependency Injection	Eine erweiterte Form des Dependency-Inversion-Prinzip, welches Abhängigkeiten erst zur Laufzeit des Programmes hinzufügt.
Enumeration	Eine Auflistung von konstanten, unveränderlichen Werten.
immutable	Die Unveränderlichkeit von Werten bzw. Variablen innerhalb einer Klasse.
Information-Expert-Prinzip	Die Verantwortung eines Anliegens liegt bei der Komponente, welche die notwendigen Informationen zur Erfüllen besitzt.
Invariante	Bedinung, welche auch nach Datenanpassungen jederzeit erfüllt sein muss.
Kohäsion	Grad der logischen inneren Zusammengehörigkeit einer Komponente. Komponente, welche nur eng beinaheliegende Aufgaben erfüllen, haben einen hohen Grad an Kohäsion.
Lost Update	Phänomen, welches bei zeitgleichen Operationen auf den gleichen Datensätzen auftreten kann. Die angepassten Datensätze einer Transaktion gehen verloren, da sie direkt von einer zweiten Transaktion überschrieben werden, welche jedoch als Ausgangspunkt noch auf den alten Stand durchgeführt worden ist.
Product Owner	Eine Scrum-Rolle, welche die wirtschaftliche Ziele und Prioritäten der Aufgabenpakete bestimmt.
Scrum	Eine agile Entwicklungsmethode, welches hohen Fokus auf kontinuierliche Verbesserung in einem geregelten Zyklus legt.
Sprint	Ein wiederkehrender festgelegter Zeitraum, indem eine vorher definierter Umfang an Arbeitspakten abgearbeitet wird.
Stakeholder	Eine Gruppe von Personen mit relevanten Interesse und Einfluss auf eine Sache bzw. Projekt.

1 Einleitung

Der Schwerpunkt dieses Projekts und somit zugleich dieser Bachelorarbeit ist die Entwicklung eines Proof-of-Concepts (POC) einer Checkout-Software unter Einsatz von Domain-Driven Design und Hexagonaler Architektur mit speziellen Fokus auf den Aggregationsschnitt des Datenmodells. In der folgenden Einleitung wird diese Problemstellung detaillierter beschrieben, sowie die dahinterliegende Motivation und Ziele erläutert. Hierdurch wird ein grundlegendes Verständnis der Hintergründe dieses Projektes geschaffen.

1.1 Problemstellung

Ein elementarer Bestandteil der Funktionsweisen eines Onlineshops erfüllt der sogenannte Warenkorb. In diesem können, unter anderem, Waren abgelegt werden, um sie zu einem späteren Zeitpunkt zu erwerben oder weitere Service hinzuzufügen. Im Verlauf des Kaufprozesses sollte es zudem möglich sein eine Versandart einzustellen, Kundendaten zu hinterlegen und gewünschte Zahlungsarten auszuwählen. Nach erfolgreicher Überprüfung von Sicherheitsrichtlinien findet die Kaufabwicklung statt, der sogenannte 'Checkout'. Um die hier beschriebenen Anwendungsfälle zu verwirklichen, wird eine eigens dafür designierte Software benötigt. In diesem Projekt wird eine solche Anwendung vereinfacht implementiert und als 'Checkout-Software' bezeichnet.

Eine solche Applikation stellt das Rückgrat des Onlineshops dar. Sie erfährt stetige Änderungen, besitzt eine Vielzahl von Businesslogik und ihre Einbindung in das Frontend beeinflusst weitergehend auch das Kundenerlebnis. Dadurch liegt ein hoher Fokus auf die Erfüllung von Qualitätsmerkmalen, wie Stabilität, Testbarkeit und Wartbarkeit. Der Checkout-Prozess, welcher durch diese Software abgewickelt wird, muss für alle relevanten Länder und ihre individuellen gesetzlichen Voraussetzungen fiskalisch korrekt ausgeführt werden. Jederzeit können neue Businessanforderungen entstehen, wodurch weitere länderspezifische Richtlinien in den Zuständigkeitsbereich der Anwendung fallen oder eine Anpassung des Datenmodells erfordern. Dementsprechend wird ein flexibles Datenkonstrukt benötigt, welches zugleich performant und übersichtlich bleibt. Das Datenmodell bestimmt weitestgehend wie die externen Systeme mit den internen Softwarekomponenten interagieren. Eine Umfeldanalyse und die klare Definition von Anwendungsfällen besitzen dadurch einen hohen Einfluss auf die resultierende Qualität des Entwicklungsprozesses. Die verwendete Architektur und das Softwaredesign sollte Entwicklern bei der Erfüllung dieser Kriterien unterstützen.

Zur Erfüllung dieser Kriterien existieren etablierte Vorgehensmodelle und Architekturen. Ein Teilaspekt der Problemstellung besteht in der Auswahl eines geeigneten Ansatzes zur Realisierung der Software. Hierbei wird auf Basis der nachfolgenden Kapiteln die Verwendung von Hexagonaler Architektur inklusive Domain-Driven Design für diese Arbeit argumentativ begründet. Die Projektdurchführung orientiert sich an den empfohlenen Entwicklungsprozess innerhalb eines Domain-Driven Kontextes. Im Zentrum der Bachelorarbeit stehen die möglichen Aggregationsschnitte des Datenmodells. Als Forschungsfrage bildet sich heraus, welche Auswirkungen unterschiedliche Aggregate-Designs auf die

Software und ihre Funktionalität besitzen. Zur Veranschaulichung und praktischen Umsetzung der zugehörigen Antwort wird eine konkrete Implementierung in Form eines Proof-of-Concepts umgesetzt.

//Kommentar: Ist die Problemstellung klar und ihre Darstellung so akzeptabel?
//Kommentar: Hinleitung zum Projektumfeld sinnvoll?

1.2 Das Unternehmen MediaMarktSaturn

Dieses Projekt wurde in dankbarer Zusammenarbeit mit dem Unternehmen *MediaMarktSaturn Retail Group*, kurz *MediaMarktSaturn*, erarbeitet.

Als größte Elektronik-Fachmarktkette Europas bietet MediaMarktSaturn in über 1023 Märkten und 13 Ländern den Kunden die Erwerbsmöglichkeit einer Vielzahl unterschiedlicher Artikel. Dabei wird ein großer Wert auf ein technologisch neuartiges Kundenerlebnis gelegt, um ein positives Einkaufserlebnis zu garantieren. Die Zugehörigkeiten der Märkte ist in den Marken *Media Markt* und *Saturn* unterteilt.

Über die Jahre gewann der Onlineshop für Media Markt und Saturn zunehmend an Bedeutung, da die prozentuale Verteilung des jährlichen Gewinns in den Märkten zurückgegangen und im Onlineshop gestiegen ist. Als Folge wurden die Unternehmensziele dementsprechend auf die Entwicklung von komplexer Software zur Unterstützung des Onlineshops neu ausgelegt. *MediaMarktSaturn Technology* ist eine Tochtergesellschaft der MediaMarktSaturn Retail Group und zuständig für alle Entwicklungstätigen des Unternehmens. Dank den 705 internen Mitarbeiter am Standort Ingolstadt kann eine einwandfreie Benutzererfahrung der Kunden erzielt werden.

Die Durchführung und Implementierung des Projektes bzw. Proof-of-Concepts geschah in Kooperation mit dem Team *Chechkout & Payment*. Die sechs zugehörigen Teammitglieder sind zuständig einen unternehmensweiten universellen Checkout für alle Länder bereitzustellen, sowohl für den Onlineshops als auch im Markt oder per Handyapp. Durch den Einsatz von Scrum wird eine konstante Verbesserung der Applikation und des Arbeitsprozesses erzielt. In kontinuierlichen Sprints wird zusätzlich die Checkout-Software auf Basis von hinzukommende Anwendungsfälle stetig expandiert. Dieses Projekt soll dem Team als Revision dienen und Aufschlüsse über mögliche architektonische Designansätze darbieten.

1.3 Motivation

Durch den fortlaufend Anstieg der Komplexität von Softwareprojekten haben sich gängige Designprinzipien und Architekturstile für den Entwicklungsprozess etabliert, sodass auch weiterhin die Businessanforderungen in einem zukunftssicheren Ansatz erfüllt und die multiple Prozessabläufen jederzeit angepasst und erweitert werden können. Dadurch ist zur Gewährleistung der Langlebigkeit einer solchen Software eine flexible Grundstruktur entscheidend. Da der Checkout ein wichtiger Bestandteil eines jeden Onlineshops ist, besitzt die Software für MediaMarktSaturn eine zentraler Bedeutung. Folglich ist eine sorgfältige Projektplanung und stetige Revision der bestehenden Software relevant, um auch weiterhin einen reibungslosen Ablauf der Geschäftsprozesse zu ermöglichen. Zur Erreichung dieses Ziels verwendet die zum aktuellen Zeitpunkt bestehende Anwendung des Checkout-Teams eine Hexagonale Architektur und Domain-Driven Design.

Dieses Projekt hilft somit bei der Überprüfung der bestehenden Architektur auf Verbesserungsmöglichkeiten und eventuelle Schwachstellen. Zudem existieren aufgrund des jetzigen zugrundeliegenden Aggregationsschnitts Performance-Einbusen. In diesem Projekt wird analysiert, ob die Performance durch einen andere Aufteilung des Datenmodells und einem damit verbundenen vertretbaren Aufwand gesteigert werden kann. Dies dient ebenfalls als nützliche Untersuchung der bestehenden Anwendung und kann als Reverenz für zukünftige Softwareprojekte verwendet werden, da viele Projekte mit ähnlichen Problemstellungen konfrontiert sind.

1.4 Ziele

Aus den vorhergehenden Motivationen lassen sich folgenden Projektziele ableiten. Grundlegend stellt diese Arbeit einen Anhaltspunkt für neue Softwareprojekte und Mitarbeiter dar. Dies kann zu einem erhöhten Grad an Softwarequalität im Unternehmen beitragen. Zugleich wird durch die Analyse und Durchführung des Proof-of-Concepts das bestehende Softwaredesign überprüft und herausgefordert. Dadurch können konkrete Verbesserungsvorschläge ein mögliches Fazit der Arbeit sein. Womöglich ergeben sich jedoch keine sinnvollen Änderungen der Produktivanwendung. Letzteres stellt dennoch eine wichtige Erkenntnis für das Team und Unternehmen dar, denn zukünftige Projekte können mithilfe der verwendeten Vorgehensmodelle ähnliche Ergebnisse erzielen. Sollten sich durch einen anderen Aggregationsschnitt erhebliche Vorteile bilden, kann ein Resultat dieses Projektes dem Umbau der Software entsprechen.

2 Grundlagen

Zur erfolgreichen Durchführung dieses Projekts werden Kernkompetenzen der Softwareentwicklung vorausgesetzt. Diese beschäftigen sich weitestgehend mit Softwaredesign und Architekturstilen. Um zu verstehen, wie eine Architektur die Programmierer bei der Entwicklungsphase unterstützt, muss zunächst festgelegt werden, welche Eigenschaften der Quellcode erfüllen soll, damit dieser positive Qualitätsmerkmale widerspiegelt. Hierzu wurden gängige Designprinzipien über die Jahre festgelegt. Unter anderem die sogenannten 'SOLID'-Prinzipien, welche im nächsten Abschnitt erläutert werden. Sie tragen bei Architekturansätze miteinander zu vergleichen und bewerten.

2.1 SOLID-Prinzipien

Das weitverbreitete Akronym 'SOLID' steht hierbei für eine Ansammlung von fünf Designprinzipien, namentlich das Single-Responsibility-Prinzip (SRP), Open-Closed-Prinzip (OCP), Liskovsches Substitutionsprinzip (LSP), Interface-Segregation-Prinzip (ISP) und das Dependency-Inversion-Prinzip (DIP). Sie sollen sicherstellen, dass Software auch bei Expansion weiterhin testbar, anpassbar und fehlerfrei bleibt. Die grundlegenden Definitionen hinter den Begriffen lauten wie folgt:

- **Single-Responsibility-Prinzip:** Jede Softwarekomponente darf laut SRP maximal eine zugehörige Aufgabe erfüllen. Eine Änderung in den Anforderungen erfordert somit die Anpassung in genau einer einzelnen Komponente. Dies erhöht stark die Kohäsion der Komponente und senkt die Wahrscheinlichkeit von unerwünschten Nebeneffekten bei Codeanpassungen.
- **Open-Closed-Prinzip:** Zur Sicherstellung, dass eine Änderung in einer Komponente keine Auswirkung auf eine andere besitzt, werden Komponente als 'geschlossen' gegenüber Veränderungen aber 'offen' für Erweiterungen anhand des OCPs definiert. Der erste Teil des Prinzips kann durch ein Interface realisiert werden, welches als geschlossen gilt, da die abstrakten Methoden keine Anpassungen ihrer Signatur erfahren dürfen, sonst müsste der darauf basierender Code ebenfalls bearbeitet werden. Dennoch können weiterhin Modifikationen durch das Vererben von Klassen oder die Einbindung von neuen Interfaces stattfinden. Dies wird als 'offen' im Sinne des OCPs angesehen.
- **Liskovsches Substitutionsprinzip:** Eine wünschenswerte Eigenschaft der Vererbung ist, dass eine Unterklasse S einer Oberklasse T die Korrektheit einer Anwendung nicht beeinflusst, wenn ein Objekt vom Typ T durch ein Objekt vom Typ S ersetzt wird. Dadurch wird die Fehleranfälligkeit bei einer Substitution im Code erheblich gesenkt und der Client kann sichergehen, dass die Funktionalität auch weiterhin den erwarteten Effekt birgt. Da das LSP sich mit der Komposition von Klassen beschäftigt, ist es für die nachfolgende Analyse vernachlässigbar.
- **Interface-Segregation-Prinzip:** Der Schnitt von Interfaces sollte so spezifisch und klein wie möglich gehalten werden, damit Clients nur Abhängigkeiten zu Funktionalitäten besitzen,

welche sie wirklich benötigen. Dadurch wird die Wiederverwendbarkeit und Austauschbarkeit der Komponente gewährleistet.

- **Dependency-Inversion-Prinzip:** Module sollten so unabhängig wie möglich agieren können. Dadurch wird eine erhöhte Testbarkeit und Wiederverwendbarkeit ermöglicht. Das zweiteilige DIP ist von zentraler Bedeutung für eine stabile und flexible Software. Der erste Abschnitt besagt, dass konzeptionell höherliegende Komponente nicht direkt auf darunterliegenden Komponenten angewiesen sein sollen, sondern die Kommunikation zwischen ihnen über eine Schnittstelle geschieht. Dies ermöglicht die Abstraktion von Funktionsweisen und löst die direkte Abhängigkeit zwischen Modulen auf. Weiterhin wird festgelegt, dass Interfaces nicht an die Implementierung gekoppelt werden sollten, sondern die Implementierung auf die Abstraktion beruht. Dadurch werden die Abhängigkeiten invertiert und ermöglicht beispielhaft die Anwendung von Dependency Injection.

Architekturen und Kompositionen können anhand dieser Prinzipien bewertet werden. Diese Vorgehensweise wird ebenfalls verwendet, um die nachfolgenden Architekturstile miteinander zu vergleichen.

2.2 Architekturmuster

Eine Softwarearchitektur beschreibt die grundlegende Struktur der Module, ihre Relationen zueinander und den Kommunikationsstil unter ihnen. Die Wahl der verwendeten Architektur beeinflusst somit die komplette Applikation und ihre Qualitätsmerkmale. Das zu bevorzugende Design einer Anwendung ist stark gekoppelt an die Anwendungsfällen und ihren Anforderungen.

In diesem Projekt soll ein Backend-Service erstellt werden, welcher mit den vorgelagerten Systemen über HTTP und REST kommuniziert. Dadurch wird die Auswahl der optimalen Architektur beschränkt, da beispielsweise Ansätze wie Model-View-Controller oder Peer-To-Peer für dieses Projektumfeld generell kaum Anwendung finden. Ein Pipe-Filter Architektur eignet sich für die Verarbeitung von einer Vielzahl an Daten, jedoch ist das Abbilden von Entscheidungsstränge und Businessrichtlinien nur umständlich verwirklichbar. Etablierte Architekturen für Backend-Software, welche die Businessprozesse als Kern der Applikation halten, müssen hingegen genauer untersucht werden. Die Schichtenarchitektur und Hexagonale Architektur erfüllen hierbei alle notwendige Bedingungen und bieten ein solides Fundament für das Projekt. Trotz ihrer ähnlichen Ziele, unterscheidet sich der Aufbau auf ersten Blick stark voneinander. In den folgenden Abschnitt werden beide Stile untersucht und anhand ihrer Tauglichkeit für eine Checkout-Software bewertet. Diese Analyse beinhaltet ebenfalls die nativ erhaltene Unterstützung der Entwickler durch die Architekturen zur Umsetzung von Designprinzipien, sodass die generelle Softwarequalität gewährleistet werden kann.

2.2.1 Schichtenarchitektur

Durch die Einteilung der Softwarekomponenten in einzelne Schichten wird eine fundamentale Trennung der Verantwortlichkeiten und ihren Aufgaben erzwungen. Die Anzahl der Schichten kann je nach Anwendungsfall variieren, liegt jedoch meist zwischen drei und vier Ebenen. Die meistverbreitete Variante beinhaltet die Präsentations-, Business- und Datenzugriffsschicht. Der Kontrollfluss der Anwendung fließt hierbei stets von einer höheren Schicht in eine tiefere gelegene oder innerhalb einer Ebene zwischen einzelnen Komponenten. Ohne eine konkrete Umkehrung der Abhängigkeiten ist

der Abhängigkeitsgraph gleichgerichtet zum Kontrollflussgraph. Hierbei dient Abbildung 2.1 als eine beispielhafte Darstellung einer solchen Architektur.

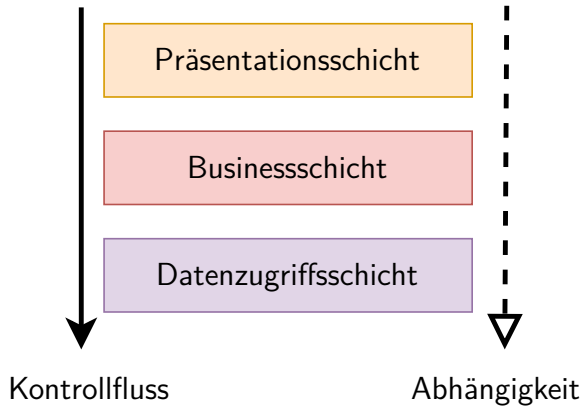

Abbildung 2.1: Beispielhafte Darstellung einer Drei-Schichtenarchitektur

Das Ziel einer Schichtenarchitektur ist die Entkopplung der einzelnen Schichten voneinander und das Erreichen von geringen Abhängigkeiten zwischen den Komponenten. Dadurch sollen Qualitätseigenschaften wie Testbarkeit, Erweiterbarkeit und Flexibilität erhöht werden. Dank dem simplen Aufbau gewann dieser Architekturstil an großer Beliebtheit, jedoch aufgrund der fehlenden Restriktionen erhalten Entwicklern nur geringe Beihilfe zur korrekten Umsetzung des Softwaredesigns.

Beispielsweise sind die SOLID-Prinzipien nur minimal im Grundaufbau verankert. Das Single-Responsibility-Prinzip wird durch die Schichteneinteilung unterstützt, da Komponente zum Beispiel keinen Zugriff auf die Datenbank und Businesslogik gleichzeitig beinhalten kann. Nichtsdestotrotz ist eine vertikale Trennung innerhalb einer Schicht nicht gegeben, daher können weiterhin Komponenten mehrere, konzeptionell verschiedene Aufgaben entgegen des SRPs erfüllen. Um die einzelnen Schichten zu entkoppelt, kann die Kommunikation zwischen den Ebenen durch Schnittstellen geschehen. Das Open-Closed-Prinzip soll hierbei helfen, dass Änderungen an den Schnittstellen und ihren Implementierungen die Funktionsweise, worauf tieferliegende Schichten basieren, nicht brechen. Die logische Zuteilung dieser Interfaces ist entscheidend, um eine korrekte Anwendung des Dependency-Inversion-Prinzips zu gewährleisten. Meist wird bei sogenannten CRUD-Applikationen eine Schichtenarchitektur verwendet. CRUD steht im Softwarekontext für '**Cr**eate **U**pdate **D**elete', somit sind Anwendungen gemeint, welche Daten mit geringer bis keiner Geschäftslogik erzeugen, bearbeiten und löschen. Im Kern einer solchen Software liegen die Daten selbst, dabei werden Module und die umliegende Architektur angepasst, um die Datenverarbeitung zu vereinfachen. Dadurch richten sich oft die Abhängigkeiten in einer Schichtenarchitektur von der Businessschicht zur Datenzugriffsschicht. Bei einer Anwendung, welche der Hauptbestandteil aus Businesslogik besteht, sollte hingegen die Abhängigkeiten stets zur Businessschicht fließen. Daher muss während des Entwicklungsprozesses stets die konkrete Einhaltung des DIPs beachtet werden, da entgegen der intuitiven Denkweise einer Schichtenarchitektur gearbeitet wird. Folglich bietet dieser Architekturansatz zwar einerseits einen hohen Grad an Simplizität, jedoch andererseits sind die SOLID-Prinzipien nur gering in dem Grundaufbau wiederzuerkennen.

2.2.2 Hexagonale Architektur

Durch weitere architektonische Einschränkungen können Entwickler zu besseren Softwaredesign gezwungen werden, ohne dabei die Implementierungsmöglichkeiten einzuengen. Dieser Denkansatz wird

in der von Alistair Cockburn geprägten Hexagonalen Architektur angewandt, indem eine klare Struktur der Softwarekomposition vorgegeben wird. Hierbei existieren drei Bereiche in denen die Komponenten angesiedelt sein können, namentlich die primären Adapter, der Applikationskern und die sekundären Adapter.

Die gesamte Kommunikation zwischen den Adaptern und dem Applikationskern findet über sogenannte Ports statt. Diese dienen als Abstraktionsschicht, sorgen für Stabilität und schützen den Kern vor Codeänderungen. Realisiert werden Ports meist durch Interfaces, welche hierarchisch dem Zentrum zugeteilt und deren Design durch diesen maßgeblich bestimmt werden. Somit erfolgt eine erzwungene Einhaltung des *Dependency-Inversion-Prinzip*, wodurch die Applikationslogik von externen Systemen und deren konkreten Implementierungen unabhängig wird. Dies erhöht drastisch Qualitätsmerkmale der Anwendung, wie beispielsweise geringe Kopplung zwischen Komponenten und Testbarkeit.

Unter den Adaptern fallen jegliche Komponenten, welche als Schnittstellen zwischen externen Systemen und der Geschäftslogik dienen. Dabei werden die primären Adapter von außerhalb angestoßen und tragen hierbei den Steuerfluss durch einen wohldefinierten Port in den Applikationskern. Zu diesen externen Systemen zählen Benutzerinterfaces, Kommandokonsolen sowie Testfälle. Andererseits bilden alle Komponenten, bei denen der Steuerfluss von dem Applikationskern zu den externen Systemen gerichtet ist, die Gruppe der sekundären Adapter. Hierbei entsteht der Impuls im Vergleich zu den primären Adaptern nicht außerhalb der Applikation sondern innerhalb. Die, von den sekundären Adaptern angesprochenen Systemen, können beispielsweise Datenbanken, Massage-Broker und jegliche Nachbarsysteme sein.

Letztendlich werden alle übrigen Module im Applikationskern erschlossen. Diese beinhalten Businesslogik und sind komplett von äußeren Einflussfaktoren entkoppelt. Der beschriebene Aufbau wird in Grafik 2.2 veranschaulicht.

Das Speichern von Daten in einer hexagonalen Applikation ist ein simpler Anwendungsfall, welcher im Folgenden beispielhaft dargestellt wird.

Ein Webclient überträgt an eine Schnittstelle des Systems Daten, wodurch er den Steuerfluss in einem sogenannten Controller initiiert. Dieser ist den primären Adaptern zugeteilt und erledigt Aufgaben, wie Authentifizierung, Datenumwandlung und erste Fehlerbehandlungen. Über einen entsprechenden Port wird der Kern mit den übergebenen Daten angesprochen. Innerhalb des Applikationszentrums werden alle business-relevanten Aufgaben erfüllt. Darunter fallen das logische Überprüfen der Daten anhand von Businessrichtlinien, Erstellen neuer Daten und die Steuerung des Entscheidungsflusses. In diesem Anwendungsfall sollen die Daten in einer Datenbank abgespeichert werden. Dementsprechend wird aus dem Anwendungskern über einen weiteren Port ein sekundäre Adapter aufgerufen, welcher für das persistieren von Daten in der Datenbank zuständig ist. //Kommentar: Abruptes Ende. Hinleitung zur Bewertung?

Anhand des Aufbaus einer Hexagonalen Architektur kann hinsichtlich der SOLID-Prinzipien im Vergleich zur Schichtenarchitektur folgendes Fazit formuliert werden:

- **SRP:** Durch den Aufbau wird eine strengere konzeptionelle Trennung der Verantwortlichkeiten erzwungen. Dies wirkt sich positiv auf die Einhaltung des Single-Responsibility-Prinzips aus.
- **OCP & ISP:** Als Folge der Einführung von Ports zwischen den Applikationskern und den business-irrelevanten Komponenten ist die Anwendung der beiden Prinzipien erleichtert und teilweise vorausgesetzt. Die Applikation profitiert von erhöhter Stabilität und Kohäsion.

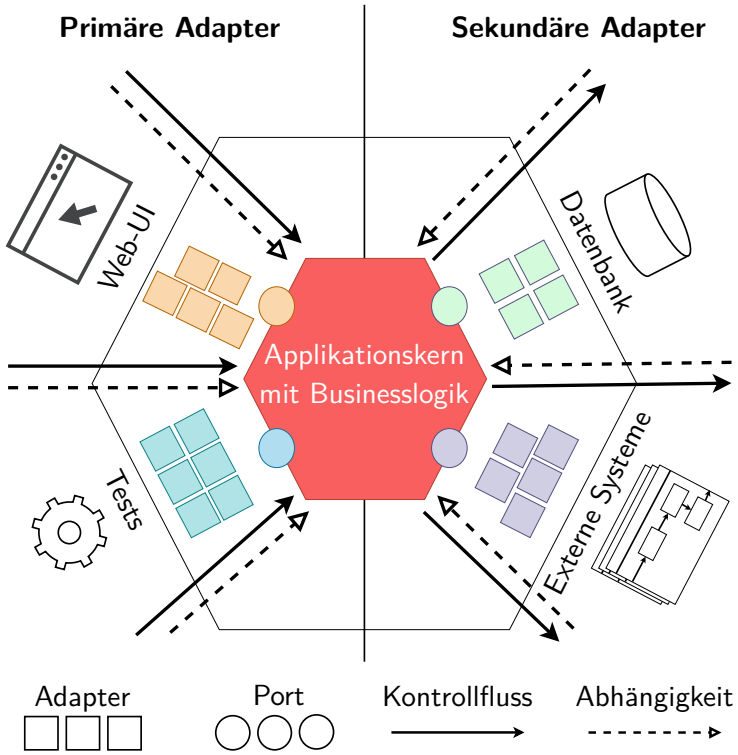

Abbildung 2.2: Grundstruktur einer Hexagonalen Architektur

- **DIP:** Mithilfe des Dependency-Inversion-Prinzips ist das Austauschen von Komponenten möglich, ohne dabei den Businesskern verändern zu müssen. Dies entkoppelt nicht nur den wichtigsten Bestandteil der Applikation, sondern fördert auch die Testbarkeit enorm. Durch eine native Invertierung der Abhängigkeiten bei korrekter Umsetzung der Hexagonalen Architektur gewinnt die Software vielen positiven Qualitätsmerkmalen.

Abschließend lässt sich die Schlussfolgerung bilden, dass für eine Checkout-Software mit intensiver Businesslogik der Einsatz einer Hexagonalen Architektur zu empfehlen ist. Nicht nur ergibt sich eine natürlichere Einhaltung der SOLID-Prinzipien, sondern der Applikationskern wird ebenfalls in den Vordergrund gerückt. Anzumerken ist, dass erfahrene Entwickler jedoch ebenfalls mit einer Schichtenarchitektur ein gleiches Maß an Softwarequalität erzielen können, sofern die Designprinzipien diszipliniert eingehalten werden, da bei genauer Betrachtung eine Hexagonale Architektur nur eine umgestellte Schichtenarchitektur mit erzwungenem Dependency-Inversion-Prinzip ist.

2.3 Domain-Driven Design

In der Entwicklungsphase von komplexer Software besteht stets die Gefahr bei steigender Anzahl von Anforderungen und Codeänderungen zu einem sogenannten 'Big Ball of Mud' zu verschmelzen. Die bestehende Architektur wird undurchschaubar, Entstehungschancen für Bugs erhöhen sich und die Businessanforderungen sind überall in der Anwendung verteilt wiederzufinden. Somit kann die Wartbarkeit der Software nicht mehr gewährleistet werden und ihre Langlebigkeit ist stark eingeschränkt. Die oben analysierten Architekturstile können bei strikter Umsetzung diese Risiken einschränken,

jedoch bestimmen sie nur begrenzt wie das zugrundeliegende Datenmodell und die damit verbundenen Komponente designt werden sollen.

In dem Buch *Domain-driven design: Tackling complexity in the heart of software* entwickelte Eric Evans im Jahre 2003 zu diesem Zweck Domain-Driven Design, kurz DDD. Der Hauptgedanke hinter Domain-Driven Design ist, dass Applikationen primär zur Bewältigung eines konkreten Problems entwickelt werden und sollten deswegen die Anforderungen durch ein strukturiertes Design aus dem Quelltext herausheben. Dadurch werden die Softwarekomponenten um die Businesslogik herum geschnitten und ihre Realisierung erleichtert. Vor allem Anwendungen mit komplexen Entscheidungssträngen und vielen, jederzeit gültigen Konditionen können dadurch übersichtlich implementiert werden. Aus diesem Grund definiert Domain-Driven Design einige Vorgehensweisen, Richtlinien und Entwurfsmuster, welche in diesem Kapitel erläutert werden.

2.3.1 Unterteilung der Problemebene in Domains und Subdomains

In einem neuen Projekt, welches Domain-Driven Design einsetzten will, sollte zu Beginn eine ausführliche Umfeldanalyse mitsamt allen relevanten Systemen durchgeführt werden, um festzulegen welche Verantwortungen in den zu bestimmenden Bereich fallen. Der Problemraum des Projekts wird dadurch als eine *Domain* aufgespannt. Hierbei ist der Domainumfang entscheidend, da darauf basierend die dazugehörigen *Subdomains* und ihre *Bounded Contexts* bestimmt werden. Eine Subdomain repräsentiert einen kleineren, spezifischeren Bereich der Domain, welcher logisch zusammenhängende Problemstellungen löst. Zur Bestimmung der Subdomains werden die Verantwortlichkeiten stets aus Businesssicht betrachtet und technische Aspekte vernachlässigt. Sollte die Domain zu groß geschnitten sein, sind dementsprechend die Subdomains ebenfalls zu umfangreich. Dadurch ist die Kohäsion der Software gefährdet und führt über den Verlauf der Entwicklungsphase zu architektonischen Konflikten. Sollte eine Subdomain mehrere logisch unabhängige Aufgaben enthalten, kann diese weiter in kleinere Subdomains unterteilt werden. Für einen Domain-Driven Ansatz ist es entscheidend die Definitionsphase gewissenhaft durchzuführen, damit eine stabile Grundlage für die Projektdurchführung geboten werden kann.

2.3.2 Bounded-Contexts und ihre Ubiquitous Language

Als Ausgangspunkt für die Bestimmung der Lösungsebene dienen die sogenannte Bounded-Contexts, welche eine oder mehrere Subdomains umfassen und ihre zugehörigen Verantwortlichkeiten bündeln. Wie es in der Praxis häufig der Fall ist, können Subdomains und Bounded-Context durchaus identisch sein. In jedem Bounded-Context sollte maximal ein Team agieren, um Kommunikationsprobleme zu vermeiden und eine klare Zuteilung der Kompetenzen zu gewährleisten. Andernfalls kann dies ein Indiz sein, dass die Subdomains zu groß geschnitten worden sind. Jeder Bounded-Context besitzt zudem eine zugehörige *Ubiquitous Language*. Die Festlegung der *Ubiquitous Language* stellt einen wichtigen Schritt der Projektphase dar. Diese definiert die Bedeutung von Begriffen, welche durch die Stakeholder und das Business verwendet werden, eindeutig. Dadurch können Missverständnisse in der Kommunikation zwischen dem Business und den Entwicklern vorgebeugt und eventuelle Inkonsistenzen aufgedeckt werden. Der größte Vorteile ergibt sich allerdings, sobald auch das Datenmodell diese Sprache widerspiegelt. Entities können Nomen darstellen, Funktionen können Verben realisieren und Aktionen können als Event verwirklicht werden. Somit sind Businessprozesse auch im Quelltext wiederzufinden. Folglich steigert dies die Verständlichkeit und Wartbarkeit der Software. Zudem lassen sich Testfälle und Anwendungsfälle leichter definieren und umsetzten. Zu beachten ist, dass diese

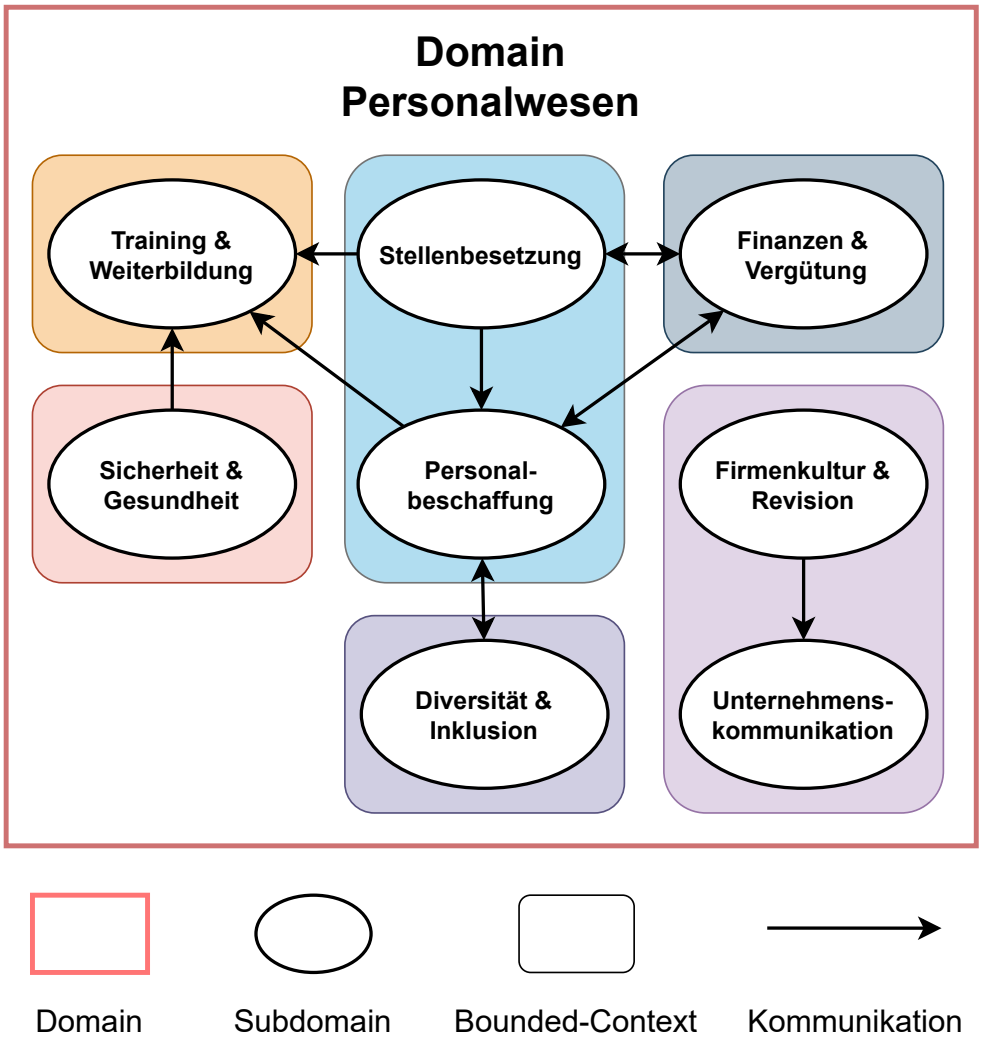

Abbildung 2.3: Beispiel einer Context-Map anhand des Personalwesens einer Firma

Sprache nur innerhalb eines Bounded-Context Gültigkeit hat. Beispielhaft kann der Begriff 'Kunde' in einem Onlineshop einen zivilen Endkunden, jedoch im Wareneingang eine Lieferfirma beschreiben. Daher ist bei der Kommunikation zwischen Teams unterschiedlicher Subdomains zu berücksichtigen, dass Begriffe eventuell unterschiedliche Bedeutung besitzen.

Die Domains, Subdomains, Bounded-Contexts und ihre Kommunikation zueinander wird durch eine Context-Map dargestellt. Diese stellt ein wichtiges Artefakt der Definitionsphase dar und kann als Werkzeug zur Bestimmung von Verantwortlichkeiten und Einteilung neuer Anforderungen benutzt werden. Sollte eine eindeutige Zuteilung nicht möglich sein, spricht dies für eine Entstehung eines neuen Bounded-Contexts und eventuell einer neuen Subdomain. Gleichermaßen wie eine Software Anpassungen erlebt, entwickelt sich die Context-Map ebenfalls stetig weiter. Zur Veranschaulichung wurde in Abbildung 2.3 das Personalwesens eines Unternehmens als Domain ausgewählt und in Subdomains bzw. Bounded-Contexts aufgeteilt. Abhängig von der Unternehmensgröße und -strategie kann der Schnitt der Bounded-Contexts auch umfassender oder detaillierter ausfallen.

2.3.3 Kombination von Domain-Driven Design und Hexagonale Architektur

Innerhalb eines Bounded-Contexts wird die grundlegende Architektur durch das zugehörige Team bestimmt. Je nach Sachverhalt des jeweiligen Kontexts kann sich diese stark von zwischen Bounded-Contexts unterscheiden. Beliebte Modellierungs- und Designstile in Verbindung mit DDD sind unter anderem Microservices, CQRS, Event-Driven Design, Schichtenarchitektur und Hexagonale Architektur. In den vorhergehenden Unterkapiteln wurden bereits die Vorzüge und Nachteile der zwei zuletzt genannten Architekturen erläutert. Auf Basis dieser Analyse wird generell für komplexere Software eine Hexagonale Architektur bevorzugt. Zudem verfolgen Domain-Driven Design und Hexagonale Architektur ähnliche Ziele, wodurch die Software natürlich an Kohäsion und Stabilität gewinnt. Im Zentrum der beiden steht das Domain-Modell, welches ohne Abhängigkeiten zu externen Modulen arbeitet. Primäre und Sekundäre Adapter sind hierzu technisch notwendige Komponente, welche durch fest definierte Ports auf den Applikationskern zugreifen können. Somit ermöglicht die Kombination aus Domain-Driven Design und Hexagonaler Architektur in Zeiten von häufigen technischen Neuheiten und komplexen Businessanforderungen weiterhin eine anpassbare, testbare und übersichtliche Software zu verwirklichen. //Kommentar: Eventuell weiter ausführen, verdeutlichen. Ist noch etwas vage.

Auf ein solches solides Grundgerüst wird mithilfe der Kenntnisse über den Bounded-Context das Domain-Modell gesetzt. Es umfasst sowohl die Datenhaltung als auch das zugehörige Verhalten, wie zum Beispiel die Überprüfung von Richtlinien, die Modifikation von Attributen oder die dauerhafte Speicherung. Für diesen Zweck existieren in Domain-Driven Design mehrere Arten von Komponente, welche anhand ihrer Verantwortlichkeiten zugeordnet werden. Die korrekte Zuordnung der Klassen und ihrer Rollen in DDD ist entscheidend für einen skalierbaren Aufbau, daher wird in den folgenden Unterkapiteln ein zentraler Überblick über die einzelnen Bestandteile ausgeführt.

2.3.4 Value Object

Die Value Objects bilden eine Möglichkeit zusammengehörige Daten zu gruppieren. Entscheidend ist hierbei die Frage, durch welche Eigenschaft der Zusammenschluss identifiziert wird. Die Identität eines Value Object wird alleinig durch die Gesamtheit ihrer Attribute bestimmt. Somit sind zwei Value Objects mit gleichen Werten auch identisch und miteinander austauschbar ohne die Funktionalität der Software zu beeinflussen.

Ein konkretes Beispiel wäre eine Klasse *Preis*, welche die Attribute für Bruttobetrag, Nettobetrag und Mehrwertsteuer enthält. In den meisten Bounded-Contexts sind alleinig die konkreten Beträge von Interesse. Sollte ein Preis die gleichen Wertebelegung besitzen, gelten sie dementsprechend als identisch. Bei einer Aktualisierung eines Preises, kann das vorgehende Objekt gelöscht und durch einen neuen Preis mit den neuen Werten ersetzt werden.

Aus diesen Grund gelten Value Objects als immutable, da sie selbst keinen Werteverlauf besitzen. Eine Neuzuweisung der Attribute ist somit nicht möglich und stattdessen werden Referenzen auf eine angepasste Instanz der Klasse gelegt. Dies gilt als ein positives Designmuster, da unveränderbare Objekte eine erhöhte Wiederverwendbarkeit genießen und unerwünschte Seiteneffekte unterdrücken. Weiterhin kann dadurch abgeleitet werden, dass sie selbst keinen eigenen Lebenszyklus besitzen, jegliche eine Momentaufnahme des Applikationszustandes darstellen. Folglich können sie nur in Zusammenhand mit Entities existieren.

2.3.5 Entity

Im Gegenzug zu einem Value Object wird eine Entity nicht durch den Zugsamenschluss ihrer Werte identifiziert, sondern enthalten ein vordefiniertes Set an unveränderlichen Attributen, welche ihre Identität bestimmen. Auch nach dem Aktualisieren ihrer Informationen bleibt ihre Identität unverändert. Demzufolge gelten die Attribute einer Entity als veränderlich und besitzen ihren eigenen Lebenszyklus, auch wenn dieser nicht explizit abgespeichert werden muss.

Ein *Kunde* in einem Domainmodell stellt einen guten Vertreter dieser Kategorie dar. In vielen Bounded-Contexts wird ein Kunde durch eine eindeutige Id ausgewiesen. Somit sind zwei Kunden mit identischen Namen dennoch nicht die gleichen Personen. Sollte der Name einer Person angepasst werden, bleibt dennoch ihre Identität bestehen.

In einer Entity werden Businessanforderungen, welche sich auf die enthaltenen Daten beziehen, direkt implementiert und ihre Invarianten sichergestellt. Dadurch wird eine hohe Kohäsion erzeugt und entsprechend des Information-Expert-Prinzips korrekt verankert.

//Kommentar: Tabelle zum Vergleich von Entities und Value Objects hier oder erst im späteren Kapitel?

2.3.6 Aggregate

Innerhalb des Bounded-Context ist ein Aggregate der Verbund aus Entities und Value Objects, welcher von außen als eine einzige Einheit wahrgenommen wird. Hierbei findet die Gruppierung anhand ihrer logischen Zusammengehörigkeit und Verantwortungen statt. Externe Komponente dürfen bei Aufruf eines Aggregates nur auf das sogenannte Aggregate Root zugreifen und nicht direkt enthaltene Objekte referenzieren. Die Root-Klasse stellt somit eine Schnittstelle zwischen dem Aggregate und der Außenwelt dar.

Ein mögliches Aggregat im Bereich des Personalmanagements ist ein *Mitarbeiter*. Das Aggregat Root ist die Klasse *Mitarbeiter* selbst. Diese beinhaltet Value Objects, wie *Gehalt* und *Abteilung*. Bei Gehaltsanpassungen wird eine Funktion auf der Mitarbeiter-Klasse aufgerufen, welche den neuen Wert durch Austausch des Value Objects einträgt. Hierbei können Invarianten überprüft werden, sodass ein neues Gehalt nicht negativ oder niedriger als das vorgehende ausfallen darf. Zu beachten ist, dass abhängig vom jeweiligen Bounded-Context zum Beispiel der Werteverlauf des Gehalt dieses Mitarbeiters vielleicht relevant ist und dementsprechend als eine Entity realisiert werden kann.

Um einen effektiven Aggregationsschnitt zu gewährleisten, wurden einige Einschränkungen und Richtlinien von Aggregates beschlossen.

Anhand des vorherigen Beispiels kann abgeleitet werden, dass Businessanforderungen bzw. Invarianten der enthaltenen Objekte stets vor und nach einer Transaktion erfüllt sein müssen. Dadurch sind die Grenzen der Aggregates durch den minimalen Umfang der transaktionalen Konsistenz ihrer Komponente gesetzt. Zur Folge dessen, wird immer das komplette Aggregat aus der Datenbank geladen und abgespeichert, sonst könnten die vorgehenden Anforderungen nicht erfüllt werden. Große Aggregates leiden aus diesem Grund an eingeschränkter Skalierbarkeit und Performance, da die Datenmenge und notwendige Operationen auf Seiten der Datenbank an Last gewinnt. Weiterhin sollte pro Transaktion jeweils nur ein Aggregat bearbeitet werden. Dies schränkt umfangreichere Aggregates durch fehlende Parallelität weiter ein. Bei Anwendung der letzteren Regel anhand der Mitarbeiter-Klasse, wäre es nicht möglich das Gehalt und die Abteilung durch zwei unterschiedliche Personalmitarbeiter anzupassen, da

eine der beiden Transaktion auf einen veralteten Stand operieren würde und zur Vermeidung eines Lost Updates zurückgerollt werden muss.

Im Falle, dass ein Anwendungsfall die Anpassung zweier Aggregates benötigt, kann dies durch eventuelle Konsistenz ermöglicht werden. Dadurch entsteht kurzzeitig ein inkonsistenter Stand der Daten, da zwei Transaktionen zeitversetzt gestartet werden. In vielen Fällen ist ein Verzug der Konsistenz aus Sicht der Businessanforderungen akzeptabel und stellt eine mögliche Alternative zur Zusammenführung der beiden Aggregates dar.

//Kommentar: Ausarbeiten und Ergänzen weil dieser Abschnitt relevant ist für das Thema?

2.3.7 Applicationservice

Aufgaben, welche kein Domainwissen erfordert, werden in den Applicationservices realisiert. Entgegen der Entities und Value Objects ist ihre Aufgabe die Bereitstellung von notwendigen Dienstleistungen. Dazu gehören das Management von Transaktionen, simple Ablaufsteuerung und Aufrufe anderer Service oder Aggregate Roots. Ihre Namensgebung und Funktionen stammt meist aus Begriffen der Ubiquitous Language.

Um Nebeneffekte ausschließen zu können und Parallelität zu ermöglichen, müssen die Applicationservice ohne Zustand designt werden. Innerhalb von Applicationservices dürfen keine Businessanforderungen enthalten sein oder Invarianten überprüft werden.

2.3.8 Domainservice

Soweit anwendbar, werden meist alle Businessanforderungen direkt in den zuständigen Entities oder Value Objects realisiert. Allerdings existieren Fälle, in denen keine klare Zuteilung der Aufgaben möglich ist. Dies kann beispielsweise auftreten, wenn sich der Prozess über zwei oder mehr Aggregates spannt. In diesem Fall wird die Funktionalität in einem Domainservice ausgelagert. Ein weiterer Grund für die Anwendung eines Domainservices kann sein, dass die auszuführende Logik Abhängigkeiten zu anderen Services besitzt oder die Kohäsion der Entity bzw. des Value Object verringert.

Analog zu den Applicationservices werden Domainservice zustandslos implementiert und finden ihre Funktionsweise aus der Ubiquitous Language. Lediglich ist der Unterschied, dass es Domainservices erlaubt ist Businesslogik umzusetzen und Invarianten zu beinhalten.

2.3.9 Factory

Die wiederholten Erstellung von komplexen Objekten kann unnötigen Platz im Quelltext einnehmen und die Übersichtlichkeit einschränken, vor allem wenn zusätzliche Services benötigt werden. Dieser Effekt wird vervielfacht, sollte das Codefragment an verschiedenen Stellen auftreten. Zur Auslagerung der Objekterzeugung sind sogenannte Factories gedacht. Sie nehmen alle nötigen Daten entgegen und geben das gefragte Objekt zurück.

2.3.10 Repository

Eine Grundfunktion von allen Anwendungen stellt die Speicherung und das Laden von Daten dar. Mithilfe von Repositories wird der Datenbankzugriff ermöglicht und orchestriert. In Domain-Driven Design benötigt jedes Aggregate ihr eigenes Repository, da sie unabhängig voneinander geladen werden müssen. Durch diese Zuordnung der Zuständigkeiten wird die konzeptionelle Abhängigkeit der Domain von den Datenbanken getrennt. Die Kommunikation mit einem Repository sollte stets über ein fest definiertes Interface geschehen, damit bei Änderungen der darunterliegenden Datenbanktechnologie der Domainkern unbetroffen bleibt.

Mithilfe des, in diesem Kapitel erarbeiteten, Wissen wurden die Grundgedanken hinter Designprinzipien, Hexagonaler Architektur und Domain-Driven Design verdeutlicht und bildet somit ein solides Fundament für die Durchführung dieses Projekts. Im folgendem wird die Planungsphase des Proof-of-Concepts erläutert.

3 Planungs- und Analysephase

Der erfolgreiche Abschluss eines Projektes mit Domain-Driven Design erfordert die sorgfältige Analyse der Domain und des Bounded-Contexts. Basierend auf diesen Erkenntnissen können erst das Domain-Modell und die Ubiquitous Language vollständig definiert werden. Besonders ist der Aggregationsschnitt stark von den Anwendungsfällen abhängig. Aus diesen Gründen wird im folgenden Kapitel eine umfassende Untersuchung des Bounded-Contexts stattfinden.

3.1 Ausschlaggebende Anwendungsfallbeschreibungen

Um den Kunden im Onlineshop oder den Mitarbeitern in den Märkten eine einwandfreie Benutzererfahrung zu gewährleisten, soll die Checkout-Software alle Prozesse vom Erstellen eines Warenkorbs bis hin zum Kaufabschluss verwalten können. Dadurch entstehen eine Vielzahl von relevanten Anwendungsfällen, welche alle korrekt und möglichst performant abgearbeitet werden müssen. Zur Dokumentation dieser Vorgänge empfiehlt es sich in der Entwicklungsphase die Prozesse in einem Diagramm abzubilden. Zusätzlich zu den Dokumentationszwecken kommt noch hinzu, dass eventuelle Unklarheiten aufgedeckt, Bedingungen an den Daten oder Programmfluss geklärt und sich eine natürliche Benutzung der Ubiquitous Language etabliert. Auf Basis dieser Anwendungsfälle ist es später möglich, Artefakte des Domain-Driven Designs leichter zu definieren. Vor allem die entscheidenden Invarianten bilden sich heraus und das Datenmodell kann klarer in Aggregates unterteilt werden. Die wichtigsten Anwendungsfälle für den Proof-of-Concept sind in diesem Kapitel vereinfacht beschrieben. Zu beachten war bei der Reduzierung der Prozesse, dass Bedingungen zwischen Datenstrukturen weiterhin unverändert sind, damit die Aggregationsschnitte im Zentrum dieser Arbeit nicht von den möglichen Varianten der Produktivanwendung abweicht. Dieses Kapitel dient somit als Grundlage für das Design der Software und wird in späteren Kapiteln referenziert. Die folgenden Prozesse wurden in Zusammenarbeit zwischen den Teams des Onlineshops, dem Checkout-Team und zuständigen Stakeholdern erarbeitet. Als Darstellungsmethode wurde sich auf Swimlane Diagramme geeinigt, damit die Interaktion zwischen den Systemen ebenfalls abgebildet werden kann.

3.1.1 Erstellung eines neuen, leeren Baskets

Ein Basket, also ein Warenkorb, stellt das grundlegendste Konstrukt des Checkouts dar. Die Anfrage auf Erzeugung eines Warenkorbs kann aus verschiedenen Gründen geschehen. Sollte ein nicht eingeloggter Kunde zum ersten Mal den Onlineshop aufrufen, wird ein Warenkorb mit der Session-ID als Kundeninformationen angelegt. Sobald dieser Kunde sich einloggt wird eine Anfrage zum Ersetzen der Session-ID durch die konkrete Kundendaten gesendet. Ein Warenkorb kann auch entstehen, wenn ein Mitarbeiter beispielsweise eine physikalische Kasse im Markt bedient und einen Kauf abschließt. Die verschiedenen Zugriffsmethoden, wie Onlineshop oder Handyapp, sind unter den Begriff 'Touchpoint' zusammengefasst. Durch diese Beschreibung ergeben sich mindestens zwei Anwendungsfälle: Die Erstellung eines Baskets und das Setzten von Kundendaten.

Die Warenkorberstellung besteht hauptsächlich aus dem Empfangen der Kundendaten, die Identifikation des zugehörigen Marktes, hier als Outlet-ID bezeichnet, und dem permanente Speicherns des neuen Warenkorbs. Dem Touchpoint wird das komplette Basket-Objekt zurückgegeben inklusive einer Basket-ID zur Referenz für spätere Zugriffe. Dieser Prozess ist in Abbildung 3.1 verdeutlicht. Es sind keine Invarianten zu prüfen, außer dem korrekten Format der Empfangsdaten. Dies gilt ebenfalls für das Aktualisieren der Kundendaten. //Kommentar: Erst Diagramm oder erst Beschreibung? Legende von Nöte für das Swimlane?

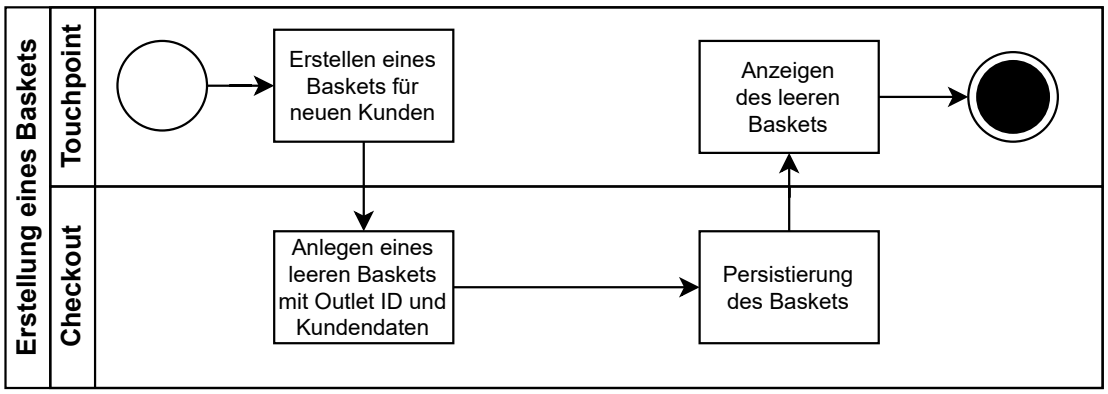

Abbildung 3.1: Swimlane Diagramm für die Erstellung eines Baskets

3.1.2 Abruf eines Warenkorbs anhand der Basket-ID

Anhand der Basket-ID kann nun der Warenkorb jederzeit durch den Touchpoint abgefragt werden. Die wichtigste Bedingung für diesen Anwendungsfall besagt, dass der Warenkorb stets mit aktuellen Daten befüllt sein muss. Dies stellt eine Herausforderung dar, da sich Preise und Artikeldetails mit dem Verlauf der Zeit ändern. Um das Problem möglichst performant zu lösen, werden die Informationen auf begrenzte Dauer zwischengespeichert. Dadurch wird nicht jedes Mal das externe System aufgerufen sondern die Werte aus dem Cache geladen. Genaue Zeitspannen wurden durch die verantwortlichen Teams festgelegt. Dies bedeutet jedoch, dass die Software das Alter der im Warenkorb enthaltenen Informationen analysieren und gegebenenfalls aktualisieren muss. Auf Authentifizierung und Autorisierung wurde in dem POC und den Diagrammen verzichtet, da es sich rein um eine technische Funktion handelt und keine Relevanz für die Projektumsetzung besitzt. Es verbleiben Aufgaben, wie die Suche des Warenkorbs innerhalb der Datenbank, die De- und Serialisierung der Objekte und das Zurückgeben von Fehlern, falls der Warenkorb nicht gefunden werden konnte. Das Schaubild 3.2 stellt das zugehörige Swimlane Diagramm für diesen Anwendungsfall dar.

3.1.3 Stornierung eines offenen Baskets

Der Warenkorb kann sich in verschiedenen Zuständen befinden. Hauptsächlich wird unterschieden zwischen 'Open', 'Freeze', 'Finalized' und 'Canceled'. Sollte beispielsweise ein Kunde beim Bezahlen an der Kasse im Markt nicht ausreichend Geld bei sich haben, muss der Warenkorb geschlossen werden. Die tatsächliche Löschung eines Baskets ist aus rechtlichen Gründen strengstens untersagt. Für diesen Prozess ist es notwendig vorher zu prüfen, ob der Warenkorb sich im Zustand 'Open' befindet, da ein

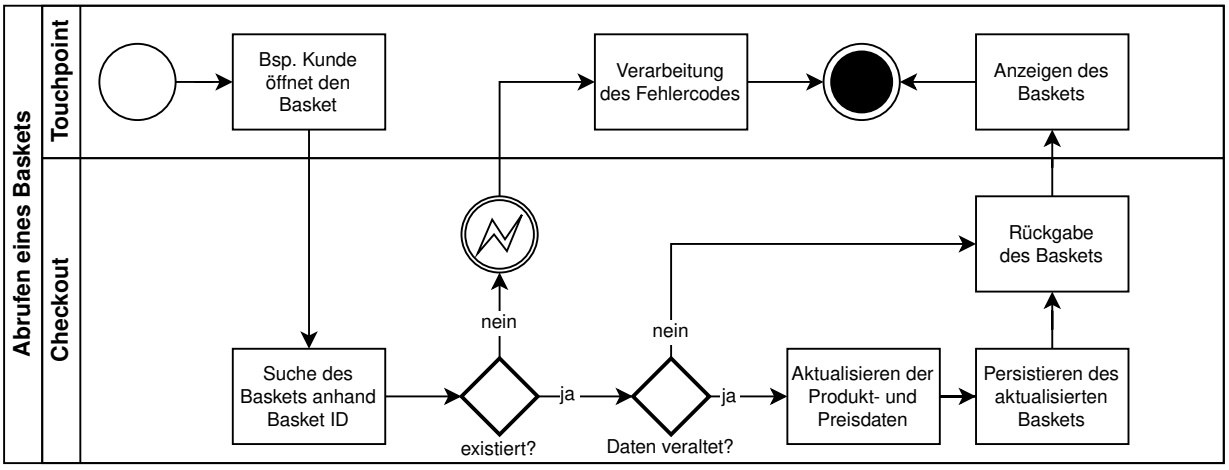

Abbildung 3.2: Swimlane Diagramm für den Abruf eines Baskets

eingefrorener, abgeschlossener oder bereits stornierter Warenkorb zur Wahrung des Zustandsverlaufes nicht storniert werden kann. Dies stellt eine Invariante dar, welche in der Software sichergestellt werden muss. Im Diagramm 3.3 ist dieser Vorgang zusammengefasst abgebildet.

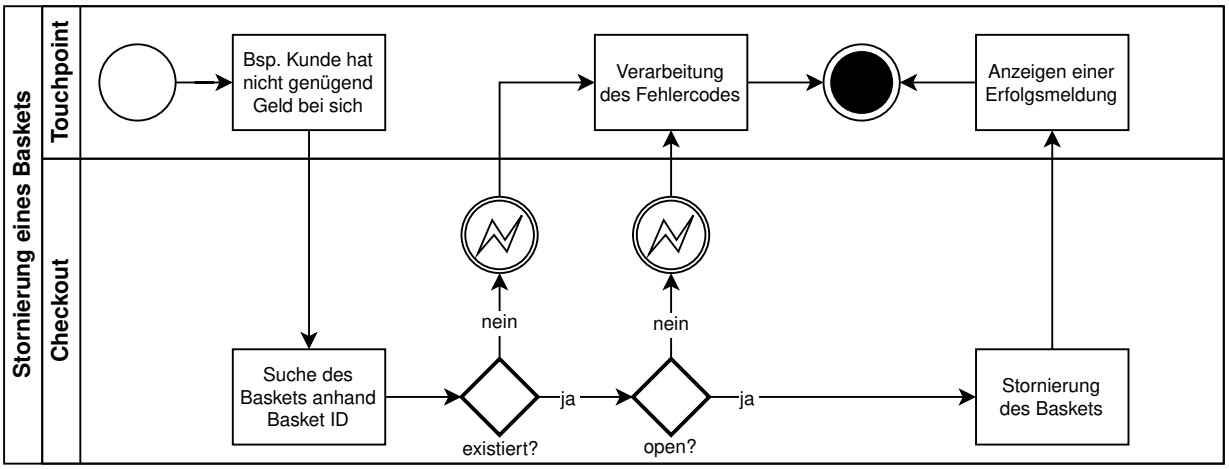

Abbildung 3.3: Swimlane Diagramm für die Stornierung eines Baskets

3.1.4 Aktualisieren der Checkout Daten des Baskets

Ein Warenkorb besitzt eine große Menge an Attributen. Einige dieser werden implizit durch einen Prozess innerhalb der Software gesetzt, andere durch Empfangen der Daten von einem externen System. Beim sogenannten 'Checkout-Prozess' werden einige dieser Daten vom Touchpoint an den Warenkorb gesendet, unter anderem Kundendaten, Bezahlmethoden und Zustellungsart. Da diese Daten im gleichen Schritt durch das vorgelagerte System gesetzt werden, bietet es sich an, diese Schnittstelle so zu designen, dass all diese Informationen gleichzeitig angepasst werden können. Eine Überprüfung der Daten erfolgt in diesem Schritt dabei nicht, mit der Ausnahme, dass die gewählte Zustellungsart, in diesem Bounded-Context 'Fulfillment' genannt, für den ausgewählten Warenkorb und dessen Produkte überhaupt verfügbar sein muss. Zudem ist es notwendig eine Neuberechnung der

Geldbeträge, wie Gesamtpreis, Mehrwertsteuer usw., durchzuführen, falls eine neue Bezahlmethode hinzugefügt worden ist. Diese Bedingungen und der dazugehörige Prozess sind im Swimlane Diagramm 3.4 veranschaulicht worden.

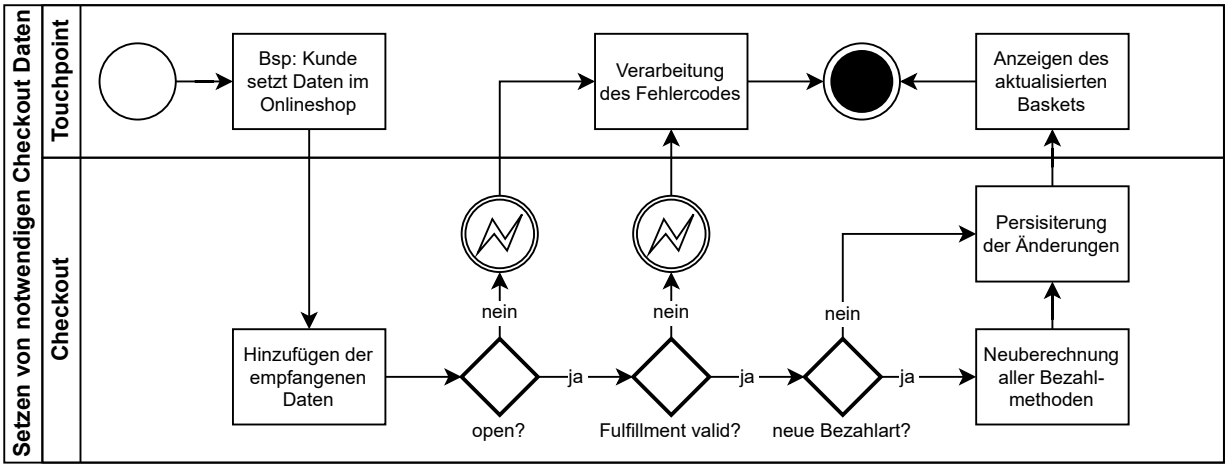

Abbildung 3.4: Swimlane Diagramm für das Setzen der Checkout Daten

3.1.5 Hinzufügen eines Produktes anhand einer Produkt-ID

Der Warenkorb fungiert ebenfalls als ein Speicher einer Liste von Artikeln, deren Quantität, Produktbeschreibung und ausgewählte Service bzw. Garantien. Der aufwändigste und deswegen in Grafik 3.5 abgebildete Prozess ist hierbei das Hinzufügen eines neuen Produktes zum Basket. Dabei sendet der Client lediglich die zugehörige Produkt-ID, weswegen externe System von der Checkout-Software aufgerufen werden müssen. Dazu gehört die Product-API, welche alle notwendigen Produktdetails liefert. Der Artikelpreis selbst ist hierbei nicht in den Produktinformationen zu finden, da dieser von Markt zu Markt unterschiedlich sein kann. Daher ist ein weiterer Aufruf einer API benötigt, welche zu der ID des Produktes und zugehörigen Outlet-ID den jetzigen Preis zurückschickt. Diese zwei APIs müssen ebenfalls bei der Aktualisierung des Warenkorbs in anderen Anwendungsfällen aufgerufen werden, sofern die zwischengespeicherten Werte im Cache nicht mehr gültig sind. Zusätzlich folgt eine Validierung des Warenkorbs auf verschiedene Parameter. Unter anderem darf die Gesamtanzahl der Artikel im Warenkorb keinen festen Wert überschreiten. Der aktualisierte Basket wird beim erfolgreichen Abschluss der Operation dem Touchpoint zurückgegeben.

3.1.6 Hinzufügen einer Bezahlungsmethode

Eine weiter essentielle Funktion ist das Hinzufügen von Bezahlarten, damit der Kauf erfolgreich initiiert werden kann. Da es sich nur um das ungeprüfte Anhängen der Beizahlinformationen handelt, müssen keine strengen Validierungen vorgenommen werden, da diese Aufgabe durch ein externes System in einem späteren Schritt des Checkouts erledigt wird. Dennoch werden logische Überprüfungen durchgeführt, wie zum Beispiel, dass der Warenkorb nicht leer oder bereits bezahlt sein darf. Ebenfalls ist eine Neuberechnung aller Bezahlinformationen notwendig. Analog zu den vorgehenden Fällen wurde das Diagramm 3.6 designt.

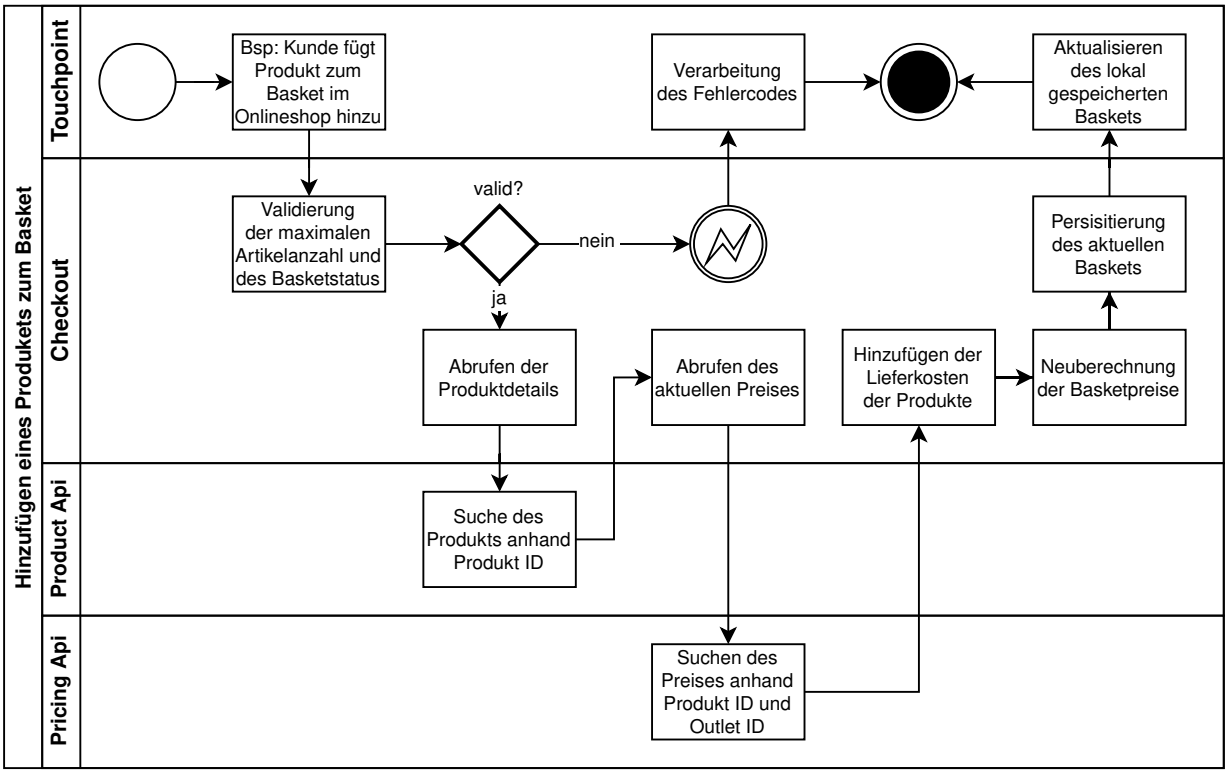

Abbildung 3.5: Swimlane Diagramm für das Hinzufügen eines Produktes

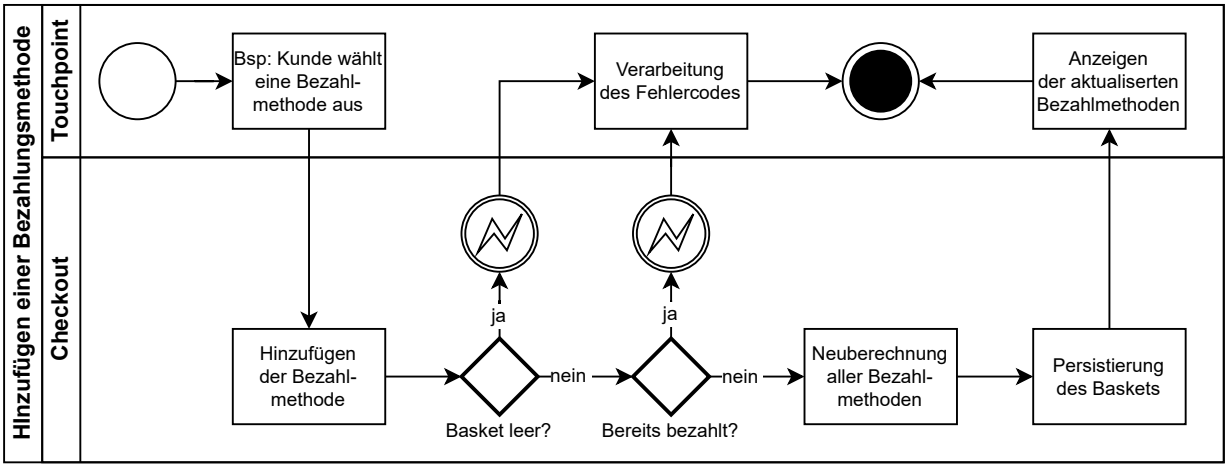

Abbildung 3.6: Swimlane Diagramm für das Hinzufügen einer Bezahlmethode

3.1.7 Initiierung des Bezahlprozesses und einfrieren des Baskets

Nachdem die Bearbeitung des Baskets abgeschlossen ist, kann der Bezahlprozess gestartet werden. Hierbei muss der Warenkorb einen konsistenten und validen Stand besitzen. Sollte dies der Fall sein, wird der Basket in den Status 'Freeze' gestellt und jegliche weitere Datenänderungen verhindert. Der Bazahlprozess wird von einer externen Software abgewickelt. Alleinig eine Referenz auf diesem Prozess wird im Warenkorb gespeichert. Eine vereinfachte Darstellung des Prozesses bietet Abbildung 3.7

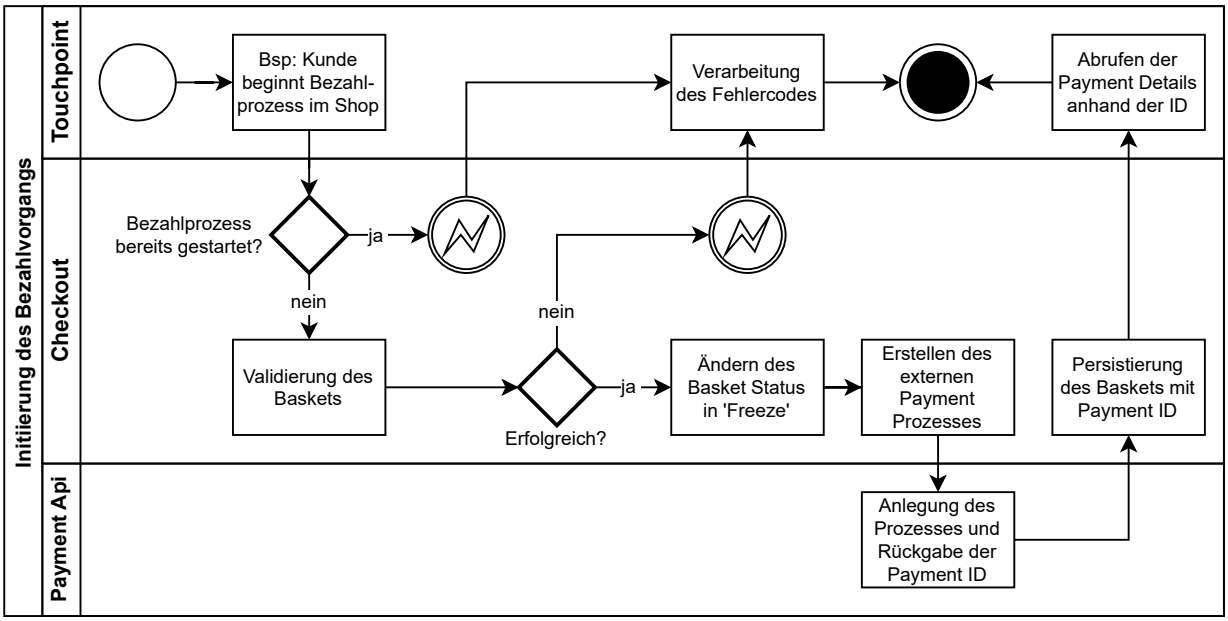

Abbildung 3.7: Swimlane Diagramm für das Initiieren des Bezahlvorgangs

3.1.8 Ausführung des Bezahlprozesses und Finalisierung des Warenkorbs

Als letzter, höchst relevanter Anwendungsfall befindet sich der Abschluss des Bezahlungsvorgangs, abgebildet in Figur 3.8. Die Checkout-Software dient hierbei als Proxy zwischen Touchpoint und Payment-API. Sofern die Bezahlung erfolgreich war, wird der Status des Baskets auf 'Finalized' gestellt. Zugleich wird eine Bestellung durch die Order-API angelegt und im Basket durch eine Referenz verlinkt.

Es existieren noch weiter simplere Prozesse, jedoch auf genaue Ausführung wurde verzichtet, um den Fokus der Arbeit beizubehalten. Durch die Anforderungen an der Checkout-Software kann auch bestimmt werden, welche Systeme als Kommunikationspartner benötigt werden. Dies ergib das Umfeld des Projekts.

3.1.9 Resultierende API-Schnittstellen aus den Anwendungsfällen

Anhand dieser Anwendungsfälle wird es möglich eine klare Schnittstellendefinition für die Checkout-Applikation zu erstellen. Hierbei beinhaltet diese alle benötigten Operationen zum erfolgreichen Bewältigen der Anforderungen aus Sicht des Touchpoints. Die Kommunikation der Systeme geschieht

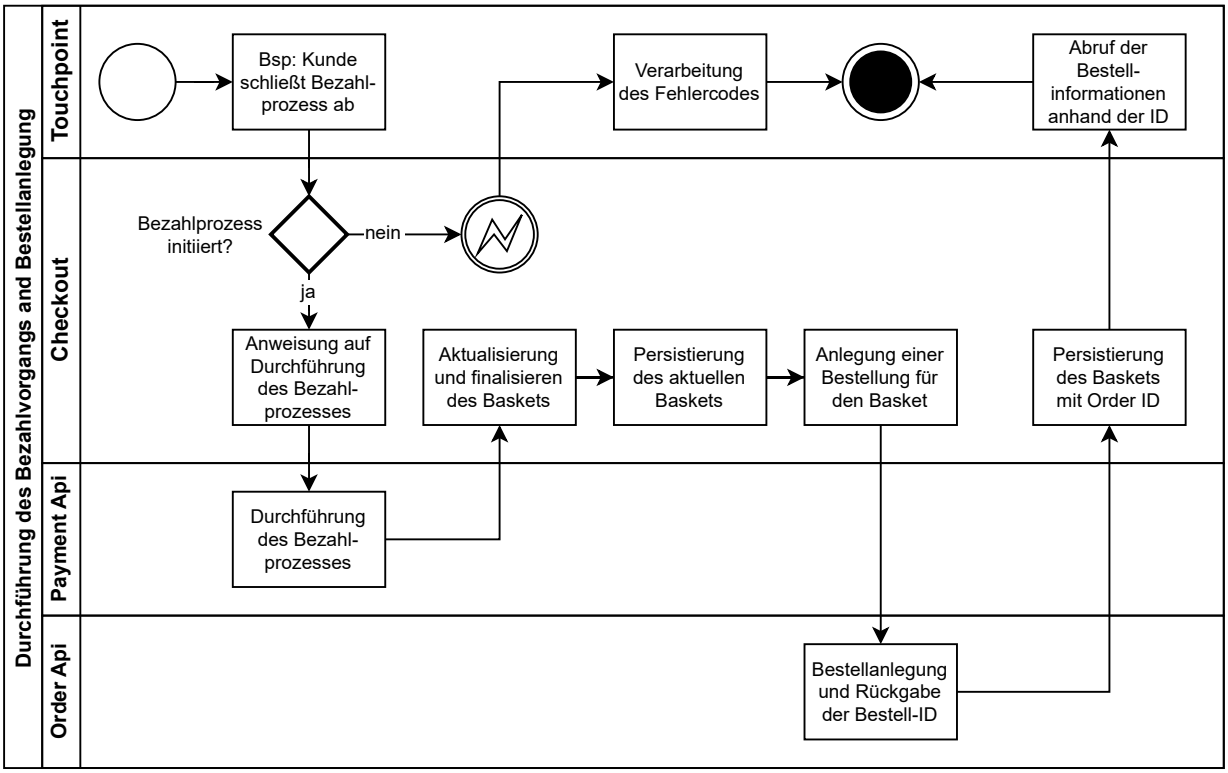

Abbildung 3.8: Swimlane Diagramm für das Ausführen des Bezahlvorgangs

über eine REST-API und somit auf Basis des HTTP-Protokolls. In folgender Grafik 3.9 sind alle relevanten Endpunkte enthalten mitsamt ihrer HTTP-Methode, Parameter und gesendeten bzw. empfangenen Datensätze.

3.2 Projektumfeld und technologische Vorschläge

Die komplette Systemumgebung von MediaMarktSaturn ist eine komplexe Struktur mit zahlreichen Abhängigkeiten zwischen Teams und ihren betreuten Applikationen. Es ist unmöglich ein solches Konstrukt aufzubauen ohne die Kommunikation der einzelnen Systeme untereinander zu definieren. Als Leitfaden für dieses Projektumfeld dienen die Anwendungsfälle des vorgehenden Unterkapitels. In dem vereinfachten Checkout-Prozess werden sechs verschiedene Schnittstellen aufgerufen. Damit plötzliche Systemänderungen keine Auswirkung auf die Funktionsweise der abhängigen Clients haben, wird eine verpflichtende API-Vereinbarung beschlossen, welcher die benötigten Informationen, mögliche Fehlerfälle und die zurückgelieferten Daten festlegt. Der Proof-Of-Concept orientiert sich an diese Vereinbarungen, erleichtert allerdings die Kommunikationsbedingungen, um unnötigen Boilerplate-Code zu unterdrücken. Als Ergebnis stellt die Grafik 3.10 ein Context-Diagramm der Umgebung dar.

Zusätzlich bestehen noch firmen- bzw. teaminterne Vorbedingungen. Wo sinnvoll anwendbar, wird in der Firma Java als Entwicklungssprache angewandt. Über die letzten Jahre gewann Kotlin an Beliebtheit und wird seitdem ebenfalls in MediaMarktSaturn eingesetzt. Die aktuelle Live-Umgebung nutzt zu dem jetzigen Stand noch Java, jedoch werden Codeanpassungen zukünftig in Kotlin vorgenommen, um eine langsame Migration zu gewährleisten. Aus diesen Grund wird ebenfalls der Proof-Of-Concept in

HTTP Methode	URL-Pfad	Antwort Daten	Anfrage Daten
	Beschreibung		
POST	**/basket**	Basket	OutletId + Customer
	Erstellung eines neuen Warenkorbs		
GET	**/basket/{basketId}**	Basket	
	Abrufen eines existierenden Warenkorbs		
DELETE	**/basket/{basketId}**	Basket	
	Stornierung eines existierenden Warenkorbs		
PUT	**/basket/{basketId}/customer**	Basket	Customer
	Abrufen eines existierenden Warenkorbs		
GET	**/basket/{basketId}/available-fulfillment**	Liste<Fulfillment>	
	Abrufen aller verfügbaren Fulfillment Methoden für diesen Warenkorb		
PUT	**/basket/{basketId}/fulfillment**	Basket	Fulfillment
	Setzen einer neuen Fulfillment Methode für diesen Basket		
PUT	**/basket/{basketId}/shipping-address**	Basket	ShippingAddress
	Setzen einer neuen Lieferadresse		
PUT	**/basket/{basketId}/billing-address**	Basket	BillingAddress
	Setzen einer neuen Rechnungsadresse		
PUT	**/basket/{basketId}/checkout-data**	Basket	Checkout Data
	Setzen von Customer, ShippingAddress, BillingAddress, Fulfillemnt und Payment		
Basis-Pfad der BasketItem API: /basket/{basketId}			
POST	**/item/{productId}**	Basket	
	Hinzufügen eines neuen Produktes an dem Warenkorb		
DELETE	**/item/{itemId}**	Basket	
	Löschen eines Eintrags im Warenkorb		
PUT	**/item/{itemId}/quantity**	Basket	
	Setzen einer konkreten Quantität für ein Warenkorb Item		
Basis-Pfad der Payment API: /basket/{basketId}			
GET	**/payment/available-payment-method**	List<PaymentMethod>	
	Abruf einer Liste an verfügbaren Zahlungsmethoden für diesen Warenkorb		
POST	**/payment**	Basket	Payment
	Hinzufügen einer Bezahlung mit optionalen konkreten Betrag		
DELETE	**/payment/{paymentId}**	Basket	
	Stornierung einer Bezahlung		
POST	**/payment/{paymentId}/initialize**	Basket	
	Initiierung des Bezahlprozesses		
POST	**/payment/{paymentId}/execute**	Basket	
	Ausführung des Bezahlprozesses		
DELETE	**/payment/{paymentId}/cancel**	Basket	
	Stornierung des Bezahlprozesses		

Abbildung 3.9: REST-API der Checkout-Software für diesen Proof-of-Concept

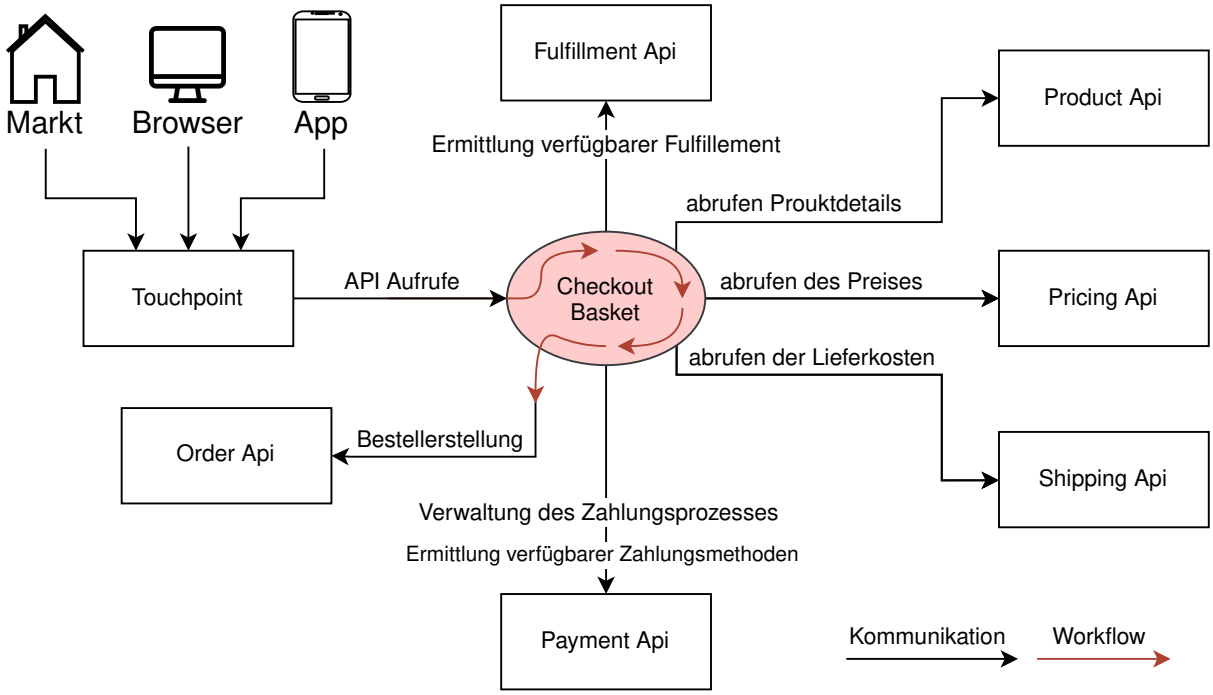

Abbildung 3.10: Context Diagramm der produktiven Checkout-Umgebung

Kotlin umgesetzt. Zudem ist die Technologie der systemübergreifenden Kommunikation auf REST-APIs festgelegt. Dies kommt mit einigen Einschränkungen und muss in der Entwicklung der primären und sekundären Adapter beachtet werden. Die Auswahl der Datenbank ist grundsätzlich nicht vorgegeben. Da die Applikation eine Vielzahl an Leseoperationen durchführt und somit einen hohen Nutzen aus der erhöhten Performance von No-SQL Datenbanken zieht, wurde die Benutzung einer MongoDB beschlossen.

4 Festlegung des Datenmodells durch Domain-Driven Design

Durch die Schaffung eines grundlegenden Verständnis für Designprinzipien, Hexagonaler Architektur und Domain-Driven Design kann auf Basis der vorgehenden Analysen ein Proof of Concept entwickelt werden. Hierzu wird weiterhin das typisches Vorgehen von DDD verfolgt und zunächst die Domain und Ubiquitous Language definiert, gefolgt vom Erstellen des zentralen Domain-Modell.

Die Wahl der Programmiersprache ist aufgrund der Teamexpertisen und dem vorliegenden Anwendungsfall auf Java oder Kotlin beschränkt. Kotlin ist eine, auf Java basierende, Programmiersprache, welche in ihrem Design einen hohen Fokus auf Lesbarkeit und Flexibilität legt. Aufgrund einer Vielzahl von Vorteile, welche durch den Einsatz von Kotlin erzielt werden können, wurde diese als die zu verwendende Sprache festgelegt. Zugleich wird nach Projektabschluss jegliche Codeanpassung an der produktiven Checkout-Software in Kotlin vorgenommen und somit ein langsame Migration des Sourcecodes erreicht.

4.1 Abgrenzung der Domain und Bounded Contexts mithilfe der Planungsphase

Aufgrund der ausführlichen Vorbereitung wurde die Domain bereits passiv festgelegt und analysiert. Das Context-Diagramm 3.10 beschreibt hierbei unsere Domaingrenzen. Eine Domain und die Subdomains spannen den Problemraum auf. Dazu zählen alle definierten Anwendungsfälle und Businessanforderungen. Die Größe der Domain ist entscheidend für die Bestimmung der Subdomains und Bounded Contexts. Den Checkout als eine Domain anzusehen würde resultieren, dass entweder sich nur ein Bounded Context ergibt, welcher sich über die ganze Domain erstreckt, oder der Checkout selbst eine weitere Unterteilung erfahren muss. Anhand der Richtlinie, dass ein Team zuständig sein soll für ein Bounded Context kann abgeleitet werden, dass dieser Domainschnitt zu klein ausfällt. Folglich kann der Checkout und alle abhängigen Systeme als nächstmögliche Domain angesehen werden. Zu Beachten ist hierbei sich nicht auf die Systeme zu fixieren, dass sie eher der Lösungsebene und somit Bounded Contexts zuweisbar sind. In den Zuständigkeitsbereich der zuständigen Domain fallen unter anderem Anforderungen an der Abwicklung des Zahlungs- und Bestellungsprozesses. Zusätzlich müssen Produkt- und Preisinformationen abrufbar sein. Hierfür muss eine Verwaltungsmöglichkeit für diese Daten bereitgestellt werden. Die Abgrenzung der Bounded Context ist durch die jetzigen Überlegungen und die bereits bestehenden Architektur vorgegeben. Eine weitere Unterteilung des Checkouts in mehrere kleiner Bounded Contexts ist wieder durch die Einschränkung auf einem Team per Bounded Context sinnfrei. Ebenfalls würde dies eine Verteilung der Verantwortlichkeiten bedeuten, was entgegen dem Ziel ein einzelnes, universelles Produkt für den Checkout anzubieten. Aus diesen Überlegungen kann folgende Context-Map erstellt werden.

4.2 Festlegen einer Ubiquitous Language

In der Kommunikation zwischen den Business und Entwicklern kann es oft zu Missverständnisse kommen. Womöglich werden Informationen, Einschränkungen oder Prozesse ausgelassen oder für selbstverständlich erachtet. Durch die klare Definition von gemeinsam verwendeten Begriffen und ihren Bedeutungen wird implizit notwendiges Wissen über die Domain und ihre Prozesse geschaffen. Viele dieser Fachbegriffe können für notwendige Anwendungsfälle wiederverwendet werden und machen die Personen, welche die Businessanforderungen umsetzten sollen, mit der Domain vertraut. Da mit geringem Domainwissen die Korrektheit der Software gefährdet wird, stellt die Ubiquitous Language in Domain-Driven Design ein wichtiger Teilaspekt dar.

In Zusammenarbeit mit den Lead-Developer und Product Owner des Teams wird im folgenden Abschnitt die Ubiquitous Language definiert, um ein solches Verständnis über den Checkout Bounded-Context zu gewährleisten. Hierbei wurde sich auf die, für dieses Projekt, relevanten Terme beschränkt und stellt lediglich eine mögliche Umsetzung einer Ubiquitous Language dar. Dank der Planungsphase wurde eine Vielzahl von Begriffen bereits definiert und hilft bei der Erstellung einer solchen Dokumentation. Eingeklammerte Wörter beschreiben Synonyme zu dem vorangestellten Ausdruck.

Domain-Modell:

- **Basket (Warenkorb):** Die zentrale Datenklasse, welche den Warenkorb darstellt.
- **Basket Status:** Stellt den aktuellen Zustand des Baskets dar und orientiert sich an den Prozessen. Beinhaltet 'open', 'frozen', 'finalized' und 'canceled'. Der Startzustand ist 'open'.
- **Customer (Kunde):** Ein Endkunde des Onlineshops oder im Markt. Kann eine zivile Person sein oder eine Firma. Ebenfalls kann ein nicht eingeloggter Kunde im Onlineshop durch seine eindeutige Session-ID einen Warenkorb besitzen.
- **Product (Artikel, Ware):** Ein Artikel aus dem Warenbestand, welcher zu Verkauf steht. Kann ebenfalls für eine Gruppierung von mehreren Artikeln stehen.
- **Outlet:** Repräsentiert einen Markt oder den länderbezogenen Onlineshop, welche durch eine einzigartige Outlet Nummer referenziert werden können.
- **BasketItemId:** Eine, innerhalb eines Baskets, eindeutige Referenz auf ein Artikel des Warenkorbs. Wird aus technischen Gründen benötigt, um beispielsweise die Quantität eines Artikels anzupassen oder ihn zu entfernen.
- **Net Amount (Nettobetrag):** Ein Nettobetrag mit dazugehöriger Währung.
- **VAT (Steuersatz):** Der Steuersatz zu einem bestimmten Nettobetrag.
- **Gross Amount (Bruttobetrag):** Der Bruttobetrag eines Preises errechnet aus dem Steuersatz und einem Nettobetrag. Die Währung gleicht die des Nettobetrags.
- **Fulfillment:** Zustellungsart der Waren seitens der Firma.
  - *Pickup:* Warenabholung in einem ausgewählten Markt durch den Kunden. Nur möglich sofern Artikel im Markt auf Lager ist.
  - *Delivery:* Zustellung der Ware an den Kunden durch einen Vertragspartner.
  - *Packstation:* Lieferung der Ware an eine ausgewählte Packstation durch einen Vertragspartner.

- **Payment Process (Bezahlungsprozess):** Alle relevanten Informationen der aktuelle Status des Bestellungsprozesses und alle zugewiesenen Payments.

- **Payment:** Zahlung des Kunden inklusive Beträge und Zahlungsmethode, wie Barzahlung oder Paypal.

- **Order:** Bestellung eines Warenkorbs, welche durch nachfolgende Systeme angelegt und verwaltet wird.

Prozesse:

- **Basket Validation:** Durchführung einer Validierung des Baskets auf Inkonsistenzen oder fehlenden, notwendigen Werten.

- **Basket Finalization:** Überprüfung der Korrektheit des Baskets durch die Basket Validation, Sperrung des Baskets durch Setzen des Zustands 'finalized' vor weiteren Abänderungen, Reservierung der Produkte im Basket und Anlegen einer Bestellung. Wird nach erfolgreichem Zahlvorgang durchgeführt. Beinhaltet den Prozess der Initiierung und Durchführung des Bezahlugnsprozesses.

- **Basket Cancellation:** Stornieren des zugehörigen Baskets. Kann nur auf einen offenen Basket ausgeführt werden und setzt den Zustand auf 'canceled'. Nach Cancellation dürfen keine weiteren Änderungen an den Basket durchgeführt werden.

- **Basket Creation:** Explizite oder Implizite Erstellung eines neuen Baskets. Findet automatisch statt sofern noch kein Basket für den Customer existiert. Dies ist ebenfalls der Fall nach einer Basket Finalization.

- **Basket Calculation:** Das Berechnen der Geldbeträge des Baskets. Beinhaltet die Auswertung aller Products mitsamt ihren Net Amounts und VATs. Der resultierende Betrag aus unterschiedlichen VATs muss weiterhin aus rechtlichen Gründen einzeln verwiesen werden können.

- **High Volume Ordering:** Die Bestellung von Artikeln in hoher Stückzahl. Aufgrund von Businessanforderungen soll es nur begrenzt möglich sein, dass ein Kunde in einem Basket oder wiederholt das gleiche Produkt mehrfach kauft.

- **Payment Initialization:** Start des Zahlungsvorgangs. Nur möglich auf einen offenen Basket, welcher Produkte und Payments enthält. Nach Validierung der Basket Inhalte wird der Zustand auf 'frozen' gesetzt. Keine weiteren Inhaltsänderungen an den einzelnen Basket Items dürfen durchgeführt werden.

- **Payment Execution:** Durchführung des Zahlungsvorgangs. Nur möglich auf einen gefrorenen Basket. Setzt den Basket bei Erfolg auf 'finalized' und legt eine Order für diesen Basket an.

Im Verlaufe der Definitionsphase der Ubiquitous Language wurden die Prozesse näher beleuchtet, Benamungen von Datenobjekten aufdeckt und Businessanforderungen vorgegeben. Ein gutes Datenmodell spiegelt hierbei ebenfalls die Sprache des Bounded-Contexts wieder, daher wird auf Basis dieses Unterkapitels und den vorgehenden Analysen anschließend das Datenmodell designt.

4.3 Definition der Value Objects

Aufgrund der Simplizität und klaren Zuteilung vom Modell zu Value Objects lassen sich diese als leichtes bestimmen. Value Objects kapseln sowohl Daten als auch das dazugehörige Verhalten. Wobei ihre Funktionalitäten keine Änderungen an den beinhalteten Attributen durchführen dürfen, da sie als immutable gelten. Daher wird, anstatt einer direkten Wertanpassung, in der Regel eine Kopie des aktuellen Objektes mitsamt der aktualisierten Werte erstellt und zurückgegeben. Zudem gilt diese Kopie als ein neues Value Object, dank der Eigenschaft, dass die Identität eines Value Objects alleinig von ihren Attributen und den zugeteilten Werten abhängt. Anfangs sollte jede Datenstruktur des Domain-Modells als ein Value Object definiert und erst nach gründlicher Überlegung, falls die Notwendigkeit besteht, zu einer Entity umgeschrieben werden.

Anfangs wird das Daten-Modell mithilfe dem erlangten Domainwissen grundlegend aufgebaut. Die Ubiquitous Language unterstützt bei der richtigen Benamung der Klassen. Der Basket stellt den Ausgangspunkt der Modellierung dar. Bei der Definition der Attribute wurde auf eine schlankes Datenmodell geachtet, ohne dabei den Aggregationsschnitt zu beeinflussen. Folgende Attribute werden benötigt:

Basket:

- **BasketId:** Eindeutige ID des Baskets zur Referenzierung durch externe System, wie den Touchpoints.
- **OutletId:** Eine Referenz zugehörig zu dem Markt oder Onlineshop, durch welchen Warenkorb angelegt wurde. Unerlässlich für die Bestimmung von unter anderem Lagerbeständen, Lieferzeiten, Fulfillment-Optionen und Versandkosten. Wir benötigen nur die Id des Outlets und nicht alle Daten, da nur die Id für unsere Kommunikationspartner relevant ist.
- **BasketStatus:** Repräsentiert den aktuellen Zustands des Warenkorbs. Mögliche Werte sind OPEN, FROZEN, FINALIZED und CANCELED.
- **Customer:** Speichert Kundendaten (IdentifiedCustomer) oder Session-Informationen (Session-Customer).
- **FulfillmentType:** Lieferart, wie PICKUP oder DELIVERY.
- **BillingAddress:** Adresse für die Rechnungserstellung.
- **ShippingAddress:** Adresse für die Warenlieferung.
- **BasketItems:** Liste aller Produkten und ihren zugehörigen Informationen aktuell im Warenkorb.
- **BasketCalculationResult:** Beinhaltet die berechneten Werte des Warenkorbs, wie Nettobetrag, Bruttobetrag und VAT. Die Speicherung dieser Werte wäre technisch nicht notwendig, spart aber an Rechenzeit ein, da nicht bei jeder Abfrage des Basket dieser Wert neu berechnet werden muss.
- **PaymentProcess:** Beinhaltet alle Informationen zur erfolgreichen Abwicklung des Zahlungsprozesses.
- **Order:** Speichert Referenz zur zugehörigen Bestellung des Baskets.

Anschließend werden die im Basket enthaltenen Klassen ebenfalls mit der gleichen Vorgehensweise definiert, sofern sie nicht durch einfache Strings oder Enumerations darstellbar sind:

SessionCustomer:

- **SessionID:** Eindeutige ID zur Zuweisung einer Session im Onlineshop zum zugehörigen Basket. Notwendig, um eine Einkaufmöglichkeit für anonyme Kunden zu bieten.

IdentifiedCustomer:

- **Name:** Enthält den Vor- und Nachnamen als eigenes Datenkonstrukt.
  - *FirstName:* Vorname des Kunden.
  - *LastName:* Nachname des Kunden.
- **Email:** Email des Kunden.
- **CustomerTaxId:** Die Steuernummer des Kunden. Relevant aus Sicht der Rechnungsabwicklung und für den Ausdruck der Rechnung, da diese ausgewiesen werden muss
- **BusinessType:** Bestimmt ob Kunde als B2C oder B2B gilt.
- **CompanyName:** Firmenname des Kunden. Kann optional angegeben werden oder ist verpflichtend bei einem B2B-Kunden
- **CompanyTaxId:** Steuernummer zugehörig zur Firma des Kunden, falls eine Firma angegeben wurde.

Address:

- **Country:** Land der Adresse.
- **City:** Stadt der Adresse.
- **ZipCode:** Postleitzahl der Adresse.
- **Street:** Straße der Adresse.
- **HouseNumber:** Hausnummer der zugehörigen Straße.

BasketItem:

- **Id:** Eindeutige Nummer des Items in diesem Basket.
- **Product:** Beinhaltet alle Produktinformationen, welche durch die Touchpoints benötigt werden.
- **Price:** Aktueller Preis des zugehörigen Products. Kann sich zeitlich ändern, muss daher durch eine Businessfunktion aktualisiert werden.
- **ShippingCost:** Betrag der Lieferkosten des Items.
- **BasketItemCalculationResult:** Speichert die Bruttokosten des Produktes, die errechneten Nettokosten, Lieferkosten und den Gesamtpreis.

Product:

- **Id:** Eindeutige Referenz für dieses Product im externen System.
- **Name:** Textuelle Produktbezeichnung des Products.
- **Vat:** Mehrwertsteuerinformationen des Products.
- **UpdatedAt:** Zeitstempel notwendig für die Aktualisierungsfunktion der Artikelinformationen.

Vat:

- **Sign:** Identifizierung des Steuertyps, abhängig von genauen Prozentsatz und zugehörigen Land.
- **Rate:** Prozentualer Wert der Mehrwertsteuer, wie beispielsweise '19%'.

Price:

- **PriceId:** Setzt sich zusammen aus der ProductId und der OutletId.
- **GrossAmount:** Bruttobetrag mit Währung.
- **UpdatedAt:** Zeitstempel notwendig für die Aktualisierungsfunktion des Preises.

BasketItemCalculationResult:

- **ItemCost:** Beinhaltet Netto, Brutto und VAT Informationen in Form eines CalculationResults.

- **ShippingCost:** Betrag der Lieferkosten.
- **TotalCost:** Zusammengerechnete Werte der einzelnen Preise im Form eines CalculationResults.

CalculationResult:

- **GrossAmount:** Bruttobetrag mit Währung.
- **NetAmount:** Nettobetrag mit Währung.
- **VatAmounts:** Eine zusammengebautes Set aus VatAmounts der Preise der BasketItems. Benötigt, da Vats mit unterschiedlichen Prozentbeträgen rechtlich nicht kombiniert werden dürfen.

VatAmount:

- **Sign:** Identifizierung des Steuertyps, abhängig von genauen Prozentsatz und zugehörigen Land.
- **Rate:** Prozentualer Wert der Mehrwertsteuer. item **Amount:** Berechneter Betrag der Mehrwertsteuer zugehörig zu einem Bruttobetrag.

BasketCalculationResult:

- **GrandTotal:** Betrag der finalen Gesamtkosten des ganzen Baskets.
- **NetTotal:** Fasst alle Nettobeträge zusammen in einem einzelnen Betrag.
- **ShippingTotal:** Fasst alle Lieferkosten zusammen in einem einzelnen Betrag.
- **VatAmount:** Rechnet alle Vats zusammen, welche das gleiche Sign besitzen.

PaymentProcess:

- **BasketId:** Id des zugehörigen Baskets.
- **ExternalPaymentRef:** Referenz zu dem zugehörigen Payment Prozess des externen Systems. Anfangs leer bis zur Initiierung des Payments.
- **AmountToPay:** Betrag der insgesamt bezahlt werden muss. Entspricht dem GrandTotal des Baskets.
- **AmountPayed:** Rechnet alle Payments zusammen und bestimmt in welchem Maße der Basket bereits bezahlt ist.
- **AmountToReturn:** Falls der bezahlte Betrag größer ist als gefordert, wird dieser Wert berechnet. Repräsentiert den Betrag, welcher durch das System zurückgegeben werden muss.
- **PaymentProcessStatus:** Status wieweit der der AmountToPay bezahlt ist. Kann die Werte TO_PAY, PARTIALLY_PAID und PAID annehmen.
- **Payment:** Liste aller Payments zugehörig zu diesem Prozess.

Payment:

- **PaymentId:** Die Id des Payments.
- **PaymentMethod:** Bezahlungsart, wie Gutschein oder Barzahlung.
- **PaymentStatus:** Aktueller Zustand des Payments. Mögliche Werte entsprechen SELECTED, INITIALIZED, EXECUTED, CANCELED. Ein Payment ist bei Hinzufügung im Status SELECTED.
- **AmountSelected:** Betrag welcher durch dieses Payment bezahlt werden soll. Falls dieser Wert leer ist, wird der gesamte Warenkorb durch dieses Payment bezahlt.
- **AmountUsed:** Betrag wie viel insgesamt durch dieses Payment abgedeckt wurde, falls nur ein Bruchteil des AmountSelectes benötigt wird.
- **AmountOverpaid:** Berechnet durch Abziehen des AmountSelecteds und AmountUsed.

Order:

- **OrderRef:** Referenz auf die Bestellung des Warenkorbs. Wird bei Abschluss des Zahlungsprozesses gesetzt.

Durch diese Datenstruktur ist es möglich, alle geforderten Anwendungsfälle korrekt abzuarbeiten. Um eine klare Gesamtübersicht zu bieten, wurde das Model in mehrere Klassendiagramme unterteilt. In Figur 4.1 ist das Ergebnis dieses Unterkapitels abgebildet. Der Customer und PaymentProcess wurden aus Platzgründen in separate Klassendiagramme, Abbildung 4.2 und 4.3, verlagert. Anzumerken ist, dass bei Fehlender Multiplizität eine Eins-zu-Eins Beziehung vorliegt. Der Kunde ist in diesem Anwendungsfall genau einem Warenkorb zugewiesen, da alleinig die Daten abgespeichert werden, nicht aber seine genaue Kundennummer. Daher existiert keine Zuweisung zwischen Kunden und mehreren Warenkörben.

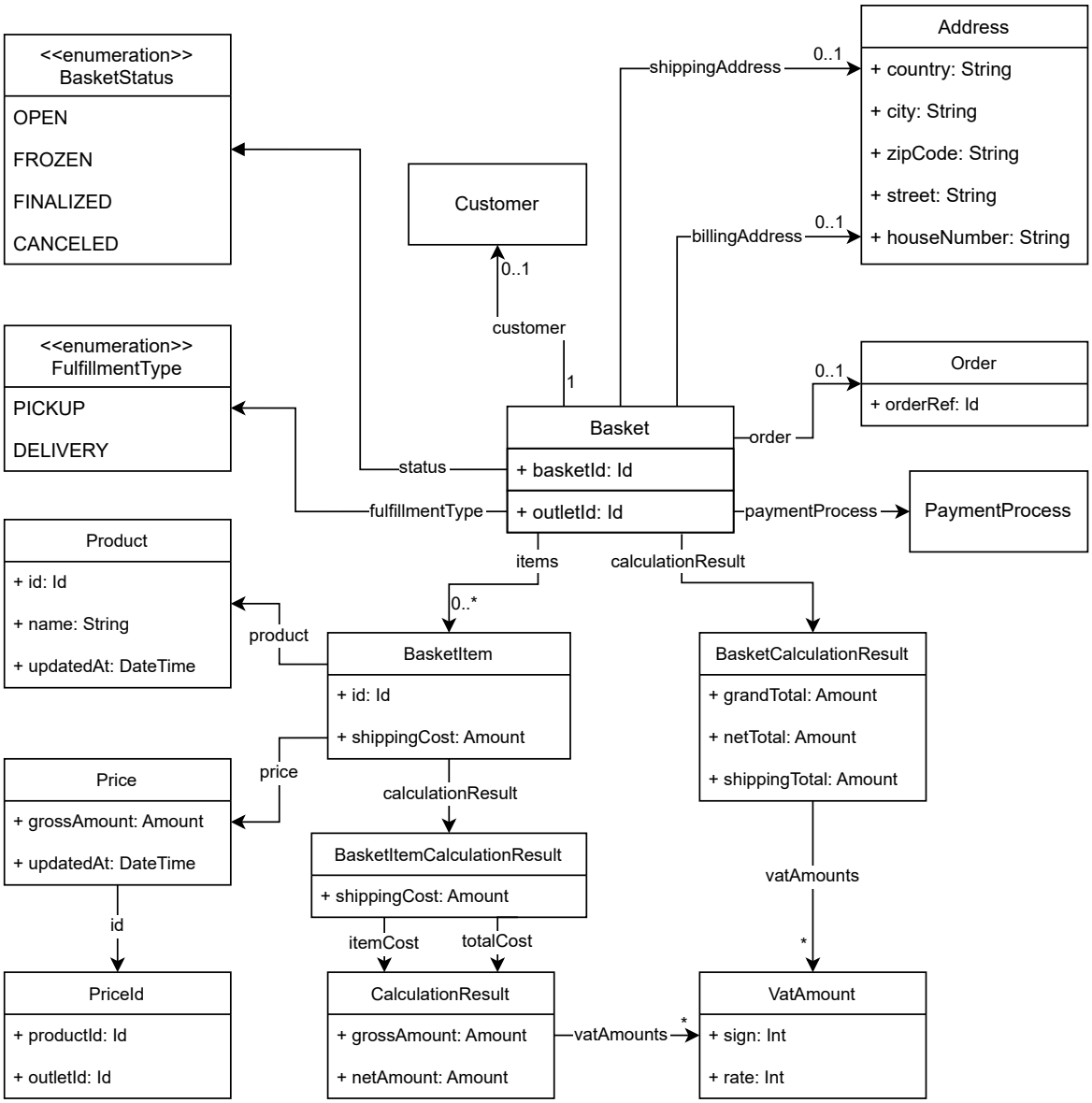

Abbildung 4.1: Klassendiagramm eines Baskets

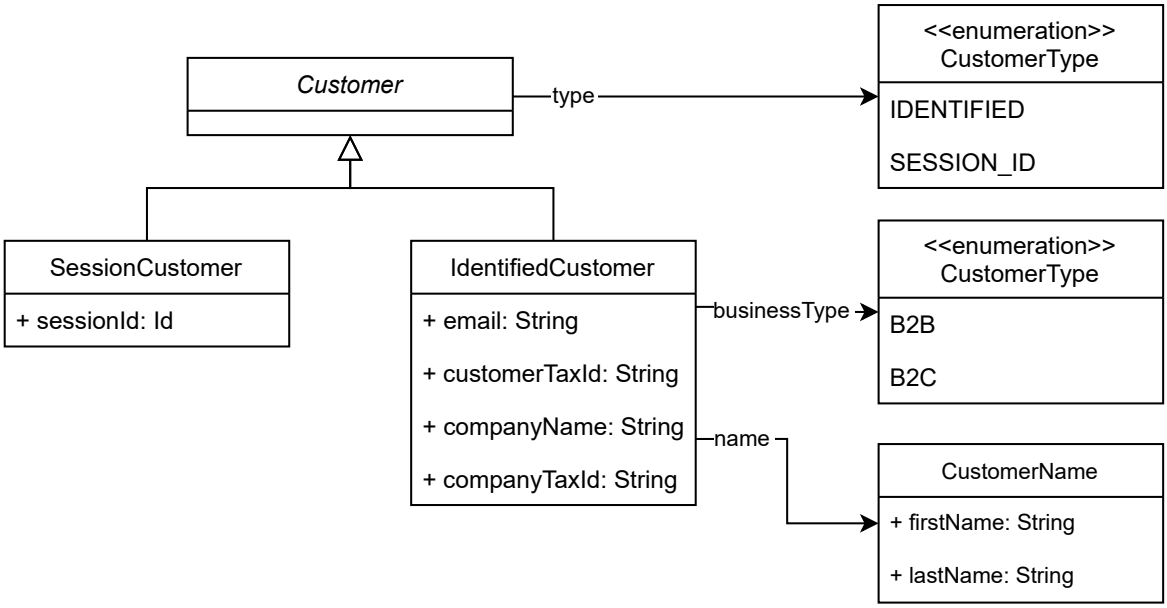

Abbildung 4.2: Zugehöriges Klassendiagramm des Customer Value Objects

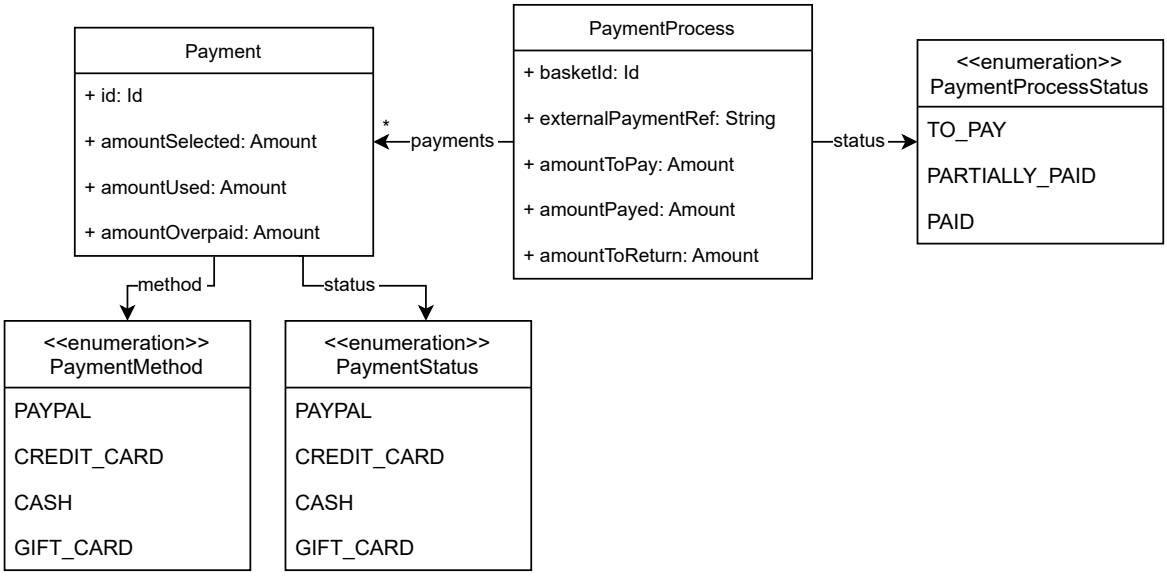

Abbildung 4.3: Darstellung des Payment Process als Klassendiagramm

	Value Object	**Entity**
Identität	Summe der Werte der Objekte. Objekte mit gleichen Werten besitzten gleiche Identität	Bestimmt anhand eines Identifikator, zum Beispiel einer Datenbank-Id. Objekte mit gleichen Werten sind ungleich, außer ihre Identifikatoren sind identisch.
Lebenszyklus	Stellt nur eine Momentaufnahme des Applikationszustands dar.	Werden zu einem bestimmten Zeitpunkt erstellt, bearbeitet, gespeichert oder gelöscht und besitzen somit einen Verlauf ihrer Wertänderungen.
Veränderbarkeit	Durch einen fehlenden Lebenszyklus gelten Value Objects als immutable.	Aufgrund ihrer Eigenschaften sind Entities mutable.
Abhängigkeit	Können nur als Unterobjekt einer oder mehrerer Entities existieren.	Um einen eigenen Lebenszyklus zu besitzen, können sie unabhängig von anderen Objekten erstellt werden.
Zugriffsmethode	Auf Daten und Funktionen wird mithilfe einer Entität zugegriffen.	Können als Aggregate Root direkt Zugriff erfahren.

4.4 Bestimmung der Entities anhand ihrer Identität und Lebenszyklus

Anhand der vorgehenden Sektion ist das Datenmodell nun vollständig. Jedoch besteht weiterhin die Frage, ob die jeweiligen Klassen eine eigene Identität besitzen und somit als Entity designt werden müssen. Es existiert in DDD kein objektive Verfahren zur Bestimmung der Entities, da je nach Bounded-Context Datengruppierungen unterschiedliche Eigenschaften besitzen. Als Hilfestellung für diese Entscheidung existieren folgende grundlegende Unterscheidungsmerkmale und Richtlinien:

Anhand dieser Eigenschaften können die Value Objects untersucht und daraufhin alle Entities bestimmt werden.

- **Basket:** Als zentrales Datenobjekt besitzt ein Basket zur eindeutigen Identifikation durch den Touchpoint eine Id. Diese Eigenschaft spricht stark für eine Entity. Zusätzlich bestimmen nicht die enthaltenen Attribute wie Products oder der zugehörige Kunde die Identität des Baskets, sondern alleinig die Id. Aufgrund der geforderten Anwendungsfälle entsteht zugleich ein Lebenszyklus für die Instanzen eines Baskets und er durchgeht verschiedene Zustände. Folglich ist ein Basket eine *Entity*.
- **IdentifiedCustomer:** In Bounded-Contexts, welche mit den Kundendaten operieren, kann diese Klasse durchaus eine Entity darstellen. In unserer Domain finden keine Operationen auf diesen Informationen statt und die vorangestellten Systeme senden bei Änderungen der Kundendaten diese zu. Folglich besitzen sie keinen eigenen Lebenszyklus, werden ebenfalls nicht separat gespeichert und kann weiterhin als *Value Object* designt werden.
- **Address:** Auf Basis der Begründungen für die IdentifiedCustomer-Klasse kann eine analoge Schlussfolgerung für alle Addressen getroffen werden und weiterhin ihren Status als *Value Object* beibehalten.
- **SessionCustomer:** Die Identifikation dieses Objekts geschieht über die SessionId. Dadurch ist ein SessionCustomer die Klasse der Entities zuzuweisen.
- **BasketItem:** Auf erstem Blick ist es eindeutig ein BasketItem als Entity zu designen. Es besitzt eine eigene Id und wird durch das Aktualisieren der Preise und Produktdaten bearbeitet. Somit entsteht ebenfalls ein Lebenszyklus. Jedoch lassen sich auch Argumente finden, warum ein

BasketItem durchaus ein Value Object sein kann. Die Identifikation erfolgt zwar durch eine Id, allerdings kann dies durch folgenden Anwendungsfall hinterfragt werden. Wenn das gleiche Produkt mehrmals sich im Basket befindet, existieren auch mehrere zugehörige BasketItems. Bei der Anpassung der Quantität beispielsweise von vier auf eins, werden anhand der Id alle Items gesucht, welche das gleiche Produkt besitzen und aus dieser Liste werden drei gelöscht. Dies würde aber bedeuten, dass ein BasketItem zusätzlich anhand seines Produktes identifiziert wird. Sollte die ProduktId die Identität des Baskets ausmachen, dann wären alle BasketItems mit dem gleiche Produkt auch identisch. Dies stimmt allerdings nur bedingt, da sie sich theoretisch durch unterschiedliche Preise und Lieferkosten unterscheiden können, falls die Aktualisierung der Preise noch nicht durchgeführt wurde. Als Folgerung kann geschlossen werden, dass ein BasketItem lediglich eine Momentaufnahme darstellt, wodurch das Design als Value Object berechtigt wäre. Letztendlich kann das BasketItem in diesem Bounded Context sowohl als Entity sowie als Value Object definiert werden. Für den Proof-Of-Concept wurde das BasketItem als Entity festgelegt. Die Begründung hierfür ist, dass die schiere Anzahl von Datenanpassungen und Operationen auf einem BasketItem als *Entity* natürlicher bewältigt werden können.

- **Product und Price:** Die vorgehende Analyse des BasketItems kann auch auf das Product und den Price angewandt werden. Beide besitzen eine Id zur Identifikation und werden stetig aktualisiert. Allerdings ist ein Price bzw. Product mit unterschiedlichen Daten aber gleicher Id in unserem Kontext auch unterschiedliche Objekte. Theoretisch ist auch hier eine Entscheidung für beide Möglichkeiten vertretbar. Jedoch ist das Datenkonstrukt beider Klassen relativ klein und Anpassungen betreffen nahezu alle Attribute, wodurch ein unveränderliches Design natürlich ausfällt. Als Folge dessen sind beide Klassen als *Value Object* umgesetzt worden.
- **CalculationResult:** Als Datenstruktur, welche bei jeder Neuberechnung aktualisiert wird, könnte die Eigenschaft eines Lebenszyklus erfüllt sein. Dennoch stellt die Klasse einzig ein Zwischenspeicher der Ergebnisse dar und ohne den Kontext eines darüberlegenden, zugehörigen Objekt besitzen diese Daten keine Aussagekraft. Weiterhin sind die gleichen Ergebnisse unterschiedlicher Baskets im Sinne der Identität äquivalent. Dadurch überwiegen die Argumente für ein *Value Object*.
- **PaymentProcess:** Der Bezahlungsprozess besitzt zur korrekter Ausführung ein Feld zum Speichern des aktuellen Status. Somit ist ein Lebenszyklus zuweisbar. Die Identität eine Payment Process ist gleich mit der BasketId, da eine Eins-zu-Eins Relation zwischen ihnen existiert. Die Lebensdauer des Objektes ist somit auch an die des Baskets gebunden. Weiterhin verwaltet ein PaymentProcess alle darunterliegenden Payments. Diese Eigenschaft erfüllen fast nur Entities. Dementsprechend ist ein PaymentProcess eine *Entity*.
- **Payment:** Ein Payment hat eine eindeutige Id, welche eine hohe Wichtigkeit trägt für den Ablauf des Bezahlprozesses und alle folgenden rechtlichen Prozesse. Dadurch ist weder der konkrete Betrag noch die Bezahlmethode von Relevanz bei der Identifikation. Ähnlich zum PaymentProcess ist auch hier ein Lebenszyklus im Form eines Statusfeldes vertreten und eine Verwirklichung als *Entity* ist zu empfehlen.

5 Design möglicher Aggregationsschnitte

Ein Domainmodell kann unterschiedlich gestaltet werden. Ebenfalls betrifft dies den Schnitt der Aggregates. Verschiedene Schnitte genießen verschiedene Vor- und Nachteile. In diesem Kapitel werden mehrere mögliche Aufteilungen untersucht und durch verschiedene Kriterien bewertet. Diese Kriterien umfassen unter anderem Performance, Komplexität, Konsistenz, Parallelität, Relevanz und Anwendbarkeit im Bezug auf den Bounded Contexts des Checkouts.

5.1 Ein zusammengehöriges Basket-Aggregate als initiales Design

Das Design eines großen Aggregats fällt Entwicklern meist natürlich leichter. Eine einzelne zusammenhängende Struktur, durch welche Daten einfach bearbeitet und Invarianten überprüft werden können. Daher hält sich die Komplexität bei diesem Aggregationsschnitt weitestgehend in Grenzen. Vor allem müssen kaum Überlegungen über transaktionale Konsistenz getroffen werden, da die Grenzen der Transaktion all umspannend ist. Die erste Variante des Proof-Of-Concepts wurde mit diesem Design im Gedanken entwickelt und stellt das Grundgerüst für alle folgenden Konzepte dar. Zur vereinfachten Referenz auf die einzelnen Umsetzungsvarianten wird das Design, welches einen großen Basket zugrundeliegend hat, als '*Variante A*' betitelt.

5.1.1 Performance von unterschiedlich großen Aggregates im Vergleich

Als Konsequenz eines großen Aggregates ergibt sich, dass immer das ganze Aggregate geladen werden muss, da auf darunterliegende Datenstrukturen nur durch das Aggregate Root zugegriffen werden darf. Für die Aktualisierung einer Entity tief im Aggregat müssen alle anderen Daten ebenfalls geladen werden. Die Auswirkungen dieser Aussage kann auf Datenbankebene anhand von Lazy-Loading oder durch Einsatz einer dokumentbasierten Datenbank eingeschränkt werden. Generell gilt, dass große Aggregates aus Sicht der Performance langsamer arbeiten als kleinere. Diese Aussage ist allerdings mit Vorsicht zu genießen und kann je nach Anwendungsgebiet sogar gegensätzlich ausfallen. In diesem Unterkapitel wird die diese Richtlinie anhand unseres Bounded Contexts analysiert.

Wie oben erwähnt, ermöglicht ein kleinerer Aggregationsschnitt das unabhängige laden und bearbeiten der Aggregates. In Betrachtung der Anwendungsfälle können die Operationen in drei große Gruppen umfasst werden. Anfragen, welche Value Objects im Basket bearbeiten stellt die erste Ansammlung dar. Diese würden unnötige Daten laden, da nur der Basket mit möglichst wenig Value Objects relevant ist. Ebenfalls gilt dies für die Anfragen, welche BasketItems oder den PaymentProcess anpassen, weil sie den ganzen Basket laden anstatt nur die wirklich benötigten Daten.

Wird die *Variante A* unterteilt in Basket-, Payment und BasketItem-Aggregate (*Variante B*) können diese somit unabhängig voneinander geladen und bearbeitet werden. Ohne Beachtung, welche Auswirkungen dieses neu überlegte Design auf andere Faktoren hat, ist eine Performanceverbesserung anhand ersten Überlegungen zu erwarten. Die vermeintlich gewonnene Performance wird allerdings aufgrund mehrerer

Umständen verringert. Eine kurze Übersicht über diesen neuen Aggregationsschnitt bietet Abbildung 5.1.

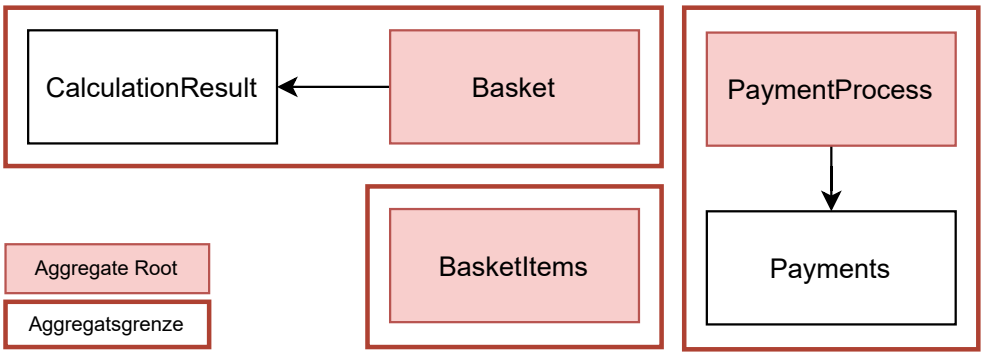

Abbildung 5.1: Aggregationsschnitt der Variante B

Invarianten über Aggregatsgrenzen hinweg

Wird dem Warenkorb ein neuer Artikel hinzugefügt hat dies zu Folge, dass nicht nur ein neues Item angelegt, sondern auch der Baskets neu berechnet werden muss. Sofern eine Aggregatstrennung zwischen Basket und BasketItem vorliegt, muss dennoch der ganze Basket geladen werden, andernfalls ist es nicht möglich den Gesamtpreis zu ermittelt. Innerhalb eines Checkout-Contexts existieren viele dieser Businessanforderungen, welche mehrere Objekte benötigen, um angewandt zu werden. Zudem ist es gegen die grundlegenden Bestimmungen eines Aggregates, dass eine Anfrage bzw. Transaktion zwei oder mehrere Aggregate bearbeiten. Dieser Anwendungsfall hätte eine Speicherung eines neuen BasketItems und die Aktualisierung des CalculationResults des Baskets zufolge. Schlussfolgernd wäre eine solche Operation nicht erlaubt ohne Verwendung von eventueller Konsistenz. Um genauer zu bestimmen, welche tatsächliche Performance erzielt werden kann durch kleinere Aggregates am konkreten Design der *Variante B* müssen die Anzahl der tatsächlich unabhängig abschließbaren Operationen untersucht werden. Die vorher festgelegten Swimlane Diagramme helfen bei dieser Aufgabe.

Ein Abruf eines Baskets kommt einher mit dem Laden aller anderen Aggregates, da die Touchpoints den gesamten Basket benötigen. Dies gilt ebenfalls für alle anderen Anwendungsfälle, welche als Antwort auf ihre Anfrage einen kompletten Basket erwarten. Das Stornieren des Baskets und Setzten der Checkout-Daten ist positiv von dieser Anpassung betroffen. Jedoch ist die Initiierung und Durchführung des Bezahlvorgangs nicht komplett unabhängig abschließbar, da im Nachhinein der Status des Baskets angepasst werden muss. Dadurch entsteht eine Abhängigkeit, welche im Nachhinein erfüllt werden muss und somit zur eventuellen Konsistenz führt. Die größte Anzahl an API-Anfragen beziehen sich auf das Hinzufügen und Bearbeiten von BasketItems und die Verwaltung des Bezahlvorgangs. Letztendlich sind nur eine Bruchzahl der Anwendungsfälle durch einen Umbau der Architektur von Variante A auf Variante B wirklich positiv betroffen.

Einfluss des Datenbankmodells auf den Aggregationsschnitt

Das Design einer Software soll stets so weit wie möglich unabhängig von verwendeten Technologien sein. Technologien entwickeln sich weiter, werden durch neuer ersetzt und bringen unnötige Abhängigkeiten in den Quelltext. Theoretisch hat somit eine Beeinflussung der Architektur durch eine externe Komponente

einen negativen Effekt auf die Qualität der Software und ihre Wartbarkeit. Dennoch kann in der Praxis dieser Gedanke durchaus Vorteile bergen, welche dieses Vorgehen gerechtfertigt. Folglich wird der Aggregatsschnitt aus Sicht der Datenbank bewertet.

Laut Definition erhält jedes Aggregate seine eigene Tabelle in der darunterliegenden Datenbank. Daher existieren in *Variante B* mindestens die Tabelle für den Basket, BasketItems und PaymentProcess. In *Variante A* kann der PaymentProcess in die gleiche Tabelle wie der Basket geschrieben werden, da eine Eins-zu-Eins Relation vorliegt. Obwohl es nur ein Aggregate gibt, sind dennoch zwei Tabellen für Basket und BasketItem notwendig, da in einer Relationalen Datenbank eine Eins-zu-N Beziehung nicht anders dargestellt werden kann. Demnach muss beim Abruf eines kompletten Baskets in *Variante A* N Joins durchgeführt werden, wobei *Variante B* N+1 Joins benötigt, wobei N die Anzahl der BasketItems entspricht. Dies resultiert in eine schlechtere Performance der *Variante B* zumindest in Bezug der Datenbankabfrage. Wie bereits erwähnt, erwarten eine Vielzahl von Anfragen alle Daten des Baskets zurück und dadurch erhalten wir eine erhöhte Bearbeitungszeit der Anfragen.

Sollte die verwendete Datenbank jedoch einen dokumentbasierten Ansatz verfolgen, gilt vorheriger Absatz nur noch bedingt. Hierbei benötigt *Variante B* weiterhin drei eigene Collections, jedoch kann *Variante A* in einem einzigen Eintrag gespeichert und geladen werden. An sich besitzt der einzelne Datensatz zwar mehr Daten, allerdings hat dies keine Auswirkung auf die Gesamtperformance. Daher würde bei einem Umbau der Aggregate für die meisten Anwendungsfälle eine einzelne Suchanfrage in N+1 Suchanfragen abgewandelt werden. Dies kann bemerkbare Auswirkungen auf die Antwortzeit unserer Applikation haben.

Im ersten Absatz wurde beschrieben, dass eine Technologie keine Auswirkung auf die Architektur haben sollte, konträr dazu ist bei der Betrachtung der Applikationsperformance dies eventuell sinnvoll. Sollte eine relationale Datenbank verwendet werden, hält sich die Performance Einbußen in Grenzen. Hingegen bei einer dokumentbasierten Datenbank ist dieser Effekt vervielfacht. Daher kann die Entscheidung über den verwendeten Aggregationsschnitt durchaus von einer konkreten Technologie beeinflusst werden.

5.1.2 Parallele Bearbeitung eines großen Aggregates

Aufgrund dessen, dass ein Aggregat eine transaktionale Grenze abbildet, wirkt sich der Aggregationsschnitt auf die mögliche Parallelität der Anfragen aus. Ausgehend von *Variante A* resultiert eine Bearbeitung der Daten des Baskets in der Speicherung des kompletten Warenkorbs in der Datenbank. Bei Schreibprozessen können Anomalien auftreten wodurch die Transaktion bzw. der Schreibvorgang abgebrochen werden muss. Eine mögliche Anomalie ist das sogenannte 'Lost Update'-Problem. Es kann auftreten wenn zwei Transaktionen den gleichen Datensatz zeitgleich bearbeiten. Sie beginnen ihre Operationen auf den selben Startzustand und schreien ihre Ergebnisse zurück in die Datenbank. Dabei kann die Änderungen einer Transaktion sofort durch eine andere überschrieben werden und dessen Datenänderungen gehen verloren.

Konkretisiert kann dieses Problem anhand des Baskets. Zwei Personen fügen beispielsweise einen neuen Artikel zu ein und demselben Warenkorb hinzu. Dies kann zum Beispiel bei einer Wishlist auftreten, welche von einer Personengruppe bearbeitet werden kann. Der erste Kunde legt hierbei ein neues Handy in den Warenkorb, wohingegen zeitgleich ein anderer Nutzer einen Fernseher hinzufügt. Beide Transaktionen starten mit einem leeren Warenkorb und schreiben in die Datenbank einen aktualisierten Datensatz mit nur einem Artikel. Daher geht die Änderung eines der zwei Personen verloren.

Eine Lösung dieses Problems kann durch optimistischen oder pessimistischen Verfahren erzielt werden. Entweder wird im optimistischen Ansatz anhand eines Versionsfeldes überprüft, ob der Datensatz in der Zwischenzeit geändert worden ist und im Falle dessen zurückgewiesen, oder der Datensatz wird durch die pessimistischen Methode gesperrt und die zweite Transaktion muss warten bis die erste den Schreibprozess abschließt. Dementsprechend ist die zeitgleiche Bearbeitung eines Aggregats eingeschränkt. Existieren Anwendungsfälle in denen eine hohe Parallelität notwendig ist, sollten die Aggregates sofern realisierbar voneinander getrennt werden.

5.1.3 Bewertung des großen Aggregationsschnitts

Abschließend kann anhand der vorhergehenden Überlegungen der Aggregationsschnitt von *Variante A* bewertet werden.

- **Komplexität:** Die Umsetzung der Businessanforderungen in der Applikation kann übersichtlich erfolgen. Der Sourcecode muss durch keine komplizierteren Vorgänge ergänzt werden.
- **Performance:** Jede Anfrage benötigt das Auslesen des ganzen Baskets. Jedoch ist dies in den meisten Anwendungsfällen ohnehin von Nöten. Ein kleinere Aggregationsschnitts kann die Performance sogar verschlechtern.
- **Parallelität:** Eine zeitgleiche Bearbeitung des Baskets oder eines Objektes innerhalb ist nicht möglich.
- **Client-Freundlichkeit:** In Hinsicht auf die wichtigsten Anwendungsfälle erfährt der Client keine Einschränkungen und alle Businessanforderungen können erfüllt werden.

5.2 Trennung der Zahlungsinformationen von dem Basket-Aggregate

Damit ein neues Aggregat aus der großen Basket-Klasse herausgeschnitten werden kann, wird ein Root Aggregate benötigt. Aus diesem Grund ist der erste Ansatz eine Entity vom Basket abzutrennen. Hierbei ist der PaymentProcess nur schwach an den eigentlichen Basket gebunden und mögliche Anwendungsfälle beziehen sich meist alleinig auf entweder den Warenkorb und seine Items oder den Bezahlprozess. Daher wird in der *Variante C* die Aggregate in Basket und PaymentProcess, wie in Grafik 5.2, aufgeteilt.

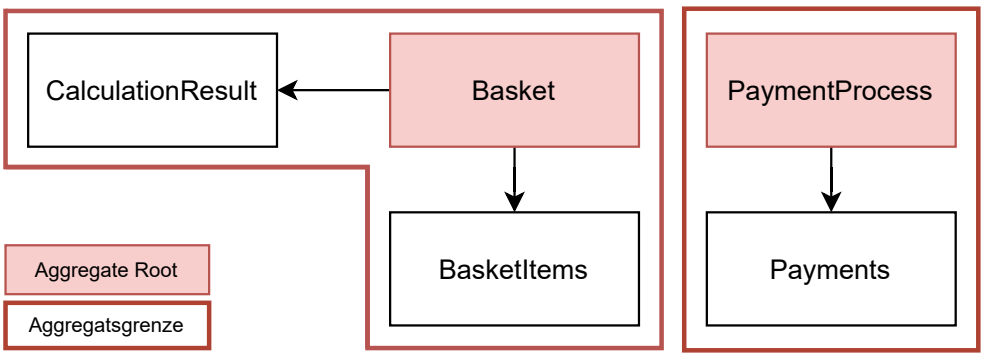

Abbildung 5.2: Aggregationsschnitt der Variante C

Da die Aggregates unabhängig voneinander agieren, müssen sie auch getrennt geladen und abgespeichert werden können. Folglich wird ein neues Repository benötigt, welches diese Operationen für das neue

Aggregate absolviert. Die Referenz auf den PaymentProcess wird aus dem Basket entfernt und alle Funktionen, welche Aufrufe auf dieses Objekt benötigt haben, müssen entweder in die PaymentProcess-Klasse verlagert oder durch einen Domainservice durchgeführt werden. Es entstehen Auswirkungen auf die bisherige Funktionsweise und den Datenfluss der Applikation, welche in den kommenden Unterkapiteln anhand der Initiierung des Bezahlvorgangs genauer untersucht werden.

Kurze Zusammenfassung des Anwendungsfalles: Bevor ein Bezahlprozess gestartet werden darf, muss eine Evaluierung des Warenkorbzustands stattfinden, wie die Überprüfung der hinzugefügten Zahlungsarten, Kundendaten und errechneten Geldbeträge. Nach erfolgreicher Validierung wird der Warenkorbzustand auf 'freeze' abgeändert und der PaymentProcess kann eingeleitet werden.

5.2.1 Eventuelle Konsistenz zwischen Aggregates

Bei der Implementierung von Variante A ist es möglich as Einfrieren des Warenkorbs und die Initiierung innerhalb einer Transaktion erfolgt. Aufgrund der Richtlinie, dass jede Transaktion nur ein Aggregate bearbeiten darf, müssen somit diese Funktionalitäten aufgespalten werden. Das konkrete Problem, welches sich dabei ergibt, ist im Codebeispiel 5.1 veranschaulicht.

```
1        fun initializePaymentProcess(basket: Basket, paymentProcess: PaymentProcess) {
2            basket.validate()
3            basket.freeze()
4            basketRepository.save(basket)
5            // Mögliches Interleaving anderer Operation, welche zur Abänderung des
6            // Baskets und des Validierungsergebnisses führt.
7            paymentProcess.initialize()
8            paymentProcessRepository.save(paymentProcess)
9        }
```

Listing 5.1: Getrennte Transaktionen für die Initiierung des Bezahlvorgangs

Zwischen dem Persistieren des eingefrorenen Warenkorb und der Initiierung des PaymentProcesses können aufgrund von Parallelität der Warenkorb durch separate Anfragen weiterhin bearbeitet werden. Konkret könnte eine Bedingung für die Validierung sein, dass die Kundendaten nicht leer sein dürfen. Jedoch wird zwischen Zeile 4 und Zeile 7 in einem parallel laufenden Thread durch einen externen API-Aufruf die Kundendaten entfernt. Dadurch wäre die eigentliche Validierung in Zeile 2 fehlgeschlagen und die Initiierung darf nicht durchgeführt werden, jedoch wird diese dennoch in unseren Beispiel angestoßen, da davon ausgegangen wird, dass sich der Basket in der Zwischenzeit nicht verändert hat. Die diese beiden Operationen nicht als eine atomare Einheit umgesetzt ist, entsteht immer eine Möglichkeit des Interleavings zwischen den einzelnen Codezeilen.

Verdeutlicht wird dieser Effekt bei der Benutzung eines Message Brokers, falls exemplarisch die einzelnen Prozesse in verschiedenen Microservices ablaufen. Hierbei wird die konkrete Initiierung in einem anderen Service durchgeführt, welcher bei Empfang eines bestimmten Events ausgeführt wird. Folglich feuert die Funktion nach Einfrieren des Warenkorbs ein solches Event ab. Ein möglicher Programmablauf ist in Diagramm 5.3 zu sehen.

Der rot markierte Bereich stellt die Zeitspanne dar, in welcher andere Prozesse den Warenkorb weiterhin bearbeiten können und dadurch die eigentliche Initiierung invalide, jedoch weiterhin durchgeführt wird. Als Folge dessen entstehen weitere Herausforderungen für das Domainmodell und die Interleavingsmöglichkeiten müssen stets in Betracht gezogen werden. Diese Art von Konsistenz wird als 'Eventuelle Konsistenz' betitelt, da es einen Zeitpunkt gibt in dem die Businessanforderungen temporär nur *eventuell* erfüllt sind. Generell kann in vielen Anwendungsfällen eine kurzzeitige Abweichung der

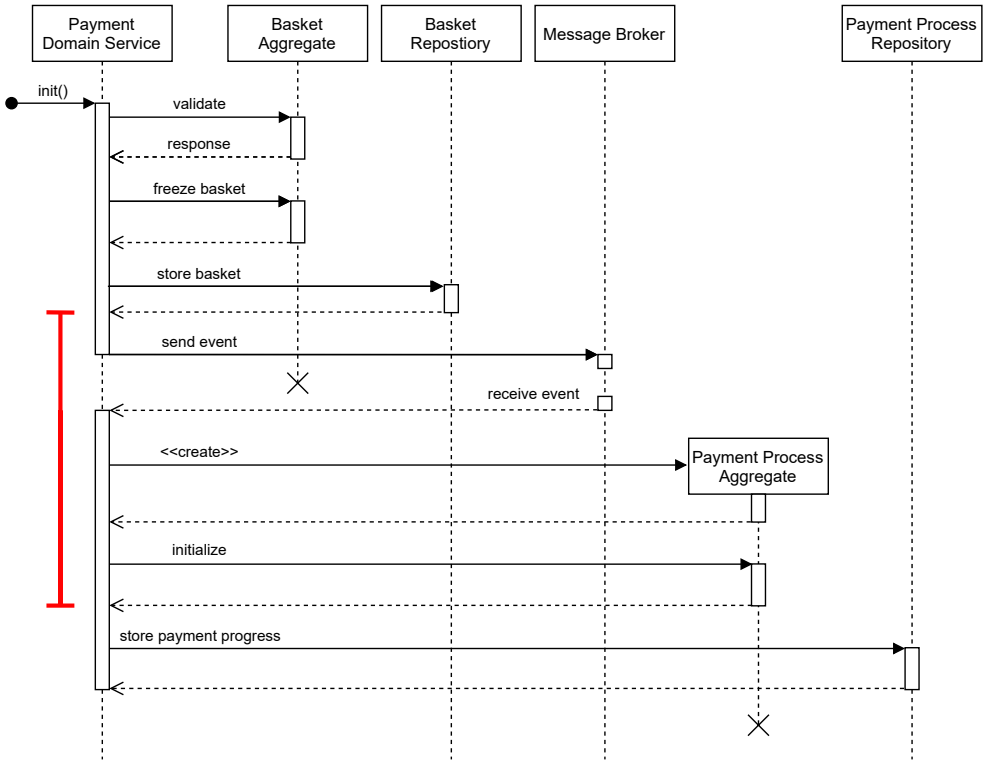

Abbildung 5.3: Vereinfachtes Sequenzdiagramm zur Initiierung des Bezahlvorgangs in Variante C

Voraussetzungen akzeptiert werden, zum Beispiel in einem Gruppenchat hat ein kurzer verzögerter Empfang der Nachrichten unter den Teilnehmern keinen großen Einfluss auf die Nutzererfahrung oder korrekte Funktionsweise der Applikation. Hingegen gilt bei der Initiierung des Zahlungsprozesses ein hoher Fokus auf fiskalisch korrekte Abarbeitung des Prozesses. Clients, welche den aktuellen Stand des Bestellprozess abfragen, erhalten bis zur Speicherung des Payment Aggregates einen veralteten Stand. Dies stellt einen bedeutenden Bruch der Invarianten dar. Folglich kann das Zusammenspiel zwischen Basket und PaymentProcess nicht durch eventuelle Konsistenz gelöst werden ohne ein hohes Risiko einzugehen.

5.2.2 Atomare Transaktionen über mehrere Aggregates

Ein weiterer Ansatz zum Bewältigen des Problems ist der Einsatz einer Transaktion über mehrere Aggregates hinweg, obwohl dies die vorher definierte Richtlinie bricht. Argumentativ muss hierzu zuerst die Frage beantwortet werden, aus welchem Grund überhaupt eine Transaktion nicht mehrere Aggregates bearbeiten darf. Durch Anpassen der Fragestellung wird dieser Aspekt klarer.

Der Hintergedanke des Einsatzes von Aggregates ist eine Gruppierung von Klassen, welche vor und nach einer Transaktion stets konsistent sein müssen. Dies schützt die Applikation vor invaliden Zuständen und erlaubt die Annahme, dass alle Businessvoraussetzungen erfüllt sind. Kann diese Eigenschaft nicht garantiert werden, muss die Software und ihre Clients eventueller Konsistenz handhaben können. Die Aggregationsgrenzen sind somit mit der transaktionalen Konsistenz äquivalent. Dadurch entspricht die Speicherung zweier Aggregates innerhalb einer einzelnen Transaktion eine Überschreitung dieser Grenzen und lässt vermuten, dass der Aggregationsschnitt neu eingeteilt werden muss. Die Richtlinie stellt folglich

lediglich ein Indiz für korrektes bzw. inkorrektes Design der Aggregates dar. Bedenklich ist alleinig, wenn die zugehörigen Aggregates auf unterschiedlichen Datenbankhosts liegen. Eine atomare Transaktion ist dadurch unmöglich, wodurch eine solche Architektur die Probleme einer eventuell konsistenten Applikation entwickelt und Interleavings auftreten können. Bei einer Microservice Architektur tritt dieser Fall oft ein, da jede einzelne Applikation meist eine eigene Datenbank auf unterschiedlichen Hosts besitzen.

Eventuelle Konsistenz ist aus den Gründen des vorgehenden Unterkapitels nicht anwendbar und für die Checkout-Software existiert aktuell kein Anlass zur Annahme, dass zukünftig mehrere Datenbankhosts benötigt werden. Dadurch kann die Überlegung entstehen, das Einfrieren des Warenkorbs und die Initiierung des Bezahlprozesses innerhalb einer Transaktion auszuführen. Dazu ist erforderlich, dass die darunterliegende Datenbank eine Transaktion über mehrere Tabellen beziehungsweise Einträge erlaubt. In diesem Projekt wird eine MongoDB zur Datenspeicherung verwendet, welche seit Version 4 atomare Operationen auf mehrere Dokumente und Collections unterstützt. Collections sind hierbei grundlegend gleichbedeutend mit Tabellen aus relationalen Datenbanken. Unsere Applikation erfüllt dementsprechend diese Voraussetzung und eine Implementierung dieses Lösungsansatzes ist technisch möglich. Die Funktion aus dem Beispiel 5.3 wird ergänzt, indem ein optimistisches Sperrverfahren auf die betroffenen Einträge in der Datenbank angewandt wird. Dadurch werden andere Zugriffe auf diesen Basket bzw. PaymentProcess erst ausgeführt, wenn die Initiierung vollständig abgeschlossen ist. Eine Möglichkeit zum Interleaving wird dadurch verhindert. Diese Vorgehensweise kann analog in ähnlichen Szenarien verwendet werden.

Weiterhin besteht die Frage, ob *Variante C* ein valides Aggregationsdesign darstellt, da in vielen Funktionen eine Überschreitung der transaktionalen Grenzen stattfindet. Begründet kann diese Entscheidung dadurch, dass der Bezahlvorgang eine kritische Operation aus rechtlicher Sicht darstellt. Sobald dieser gestartet wird, muss das ganze Datenmodell stets konsistent sein. Wenn es uns nicht erlaubt sein sollte, eine Transaktion über mehrere Aggregates durchzuführen, sind somit alle Aggregationsschnitte, abgesehen von *Variante A*, nicht zulässig. Eventuell kann diese Erkenntnis bereits eine vorläufige Antwort für die initiale Forschungsfrage dieser Arbeit darstellen.

Letztendlich soll die Architektur und das Datenmodell die Businessprozesse optimal unterstützen, während die Software weiterhin flexibel und performant bleibt. Die Richtlinie stellt einzig einen Leitfaden dar, um ein korrektes Design zu erleichtern, jedoch keine absolute Regel. Der Anreiz für eine genauer Unterteilung der Aggregates ist das separate Laden und zeitgleiche Bearbeiten unterschiedlicher Aggregates, welches weiterhin in *Variante C* entlang der Richtlinie möglich ist. Anhand dieser Begründung wird für den Proof-of-Concept angenommen, dass ein solcher Aggregationsschnitt als Designentscheidung vertretbar ist.

5.3 Verkleinerung der Aggregates durch Analyse existierender Businessanforderungen

Die ideale Gruppierung von Klassen zu einem Aggregate hängt stark von den Invarianten ab, welche sie zusammenbinden. Anhand einer Untersuchung von Anwendungsfällen ist es möglich, das Zusammenspiel von Entities und Value Objects innerhalb eines Aggregates klarer herauszukristallisieren und neue Ansätze zur Neuverteilung zu finden.

5.3.1 Herausschneiden der Berechnungsergebnisse aus dem Basket-Aggregate

Zum jetzigen Zeitpunkt existieren einige Performance-Einbußen, welche eventuell durch einen verbesserten Aggregationsschnitt verhindert werden können. Viele Anwendungsfälle erfordern ein Neukalkulation des Baskets, ansonsten wäre sein Zustand inkonsistent, wodurch auch die transaktionalen Grenzen des Aggregates ungültig werden würden. In den meisten User Stories fügt der Kunde erst die gewünschten Artikel zum Warenkorb hinzu und öffnet danach Basket, sobald der Kauf abgeschlossen werden sollte. Dadurch ist es möglich die Neuberechnung der Preise erst bei Abruf explizit durchzuführen und Berechnungszeit einzusparen. Zudem werden weniger Daten zwischen Client und Checkout-Software gesendet, wodurch die Netzwerklast vor allem bei mobilen Touchpoints verringert wird. Dieses Design kommt allerdings mit einigen Fragen, welche zuerst beantwortet werden müssen.

Ist die Trennung der Kalkulation vom Warenkorb überhaupt möglich aus Sicht des Business?

Bevor Überlegungen über die Umsetzung des neuen Aggregationsschnittes stattfinden können, müssen es die Businessanforderungen zulassen. Aus technischen Gründen hätte die Abspaltung positive Auswirkungen, jedoch kann es vorkommen, dass die Clients bei jedem Aufruf der API auch ein CalculationResult erwarten. Ist dies immer oder in den meisten Anforderungen der Fall, kann die Trennung nicht sinngemäß durchgeführt werden, ohne auf die Probleme der bisherigen Aggregationsschnitte zu stoßen. Zum Zwecken der Analyse wird angenommen, dass eine solche Änderung für die Touchpoints akzeptabel ist.

Wann muss der Basket sowohl als auch das CalculationResult bearbeitet werden?

Als Folge der gewonnen Erkenntnisse kann es problematisch sein, zwei Aggregates gleichzeitig anzupassen. Das Ergebnis der Preisberechnung wird nur bei der Anzeige des ganzen Warenkorbs benötigt, daher sind alle Operationen, welche den Gesamtpreis während der Einsicht des Baskets manipulieren bedenklich. Um diese Situationen ausfindig machen zu können, muss der Checkout-Prozess genauer untersucht werden. In der Abbildung 5.4 sind die zwei relevanten Seiten des Checkouts dargestellt. Die roten Pfeile zeigen auf wichtige Stellen des Warenkorbs für dieses Abschnitt.

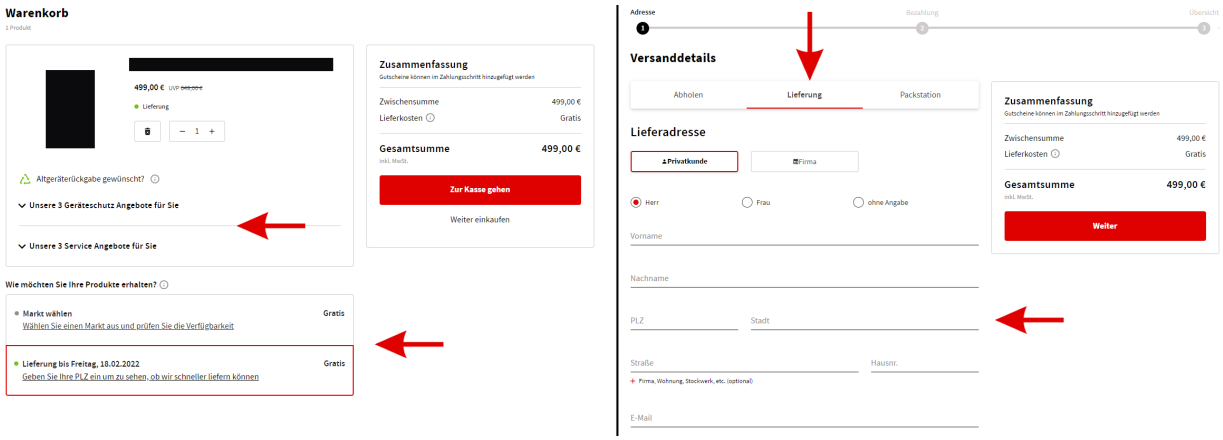

Abbildung 5.4: Aktueller Checkout-Prozess des Onlineshops von MediaMarkt.de

Eine Neukalkulation findet statt sofern die Anzahl der Produkte im Warenkorb oder die Lieferkosten manipuliert werden. Ersteres kann außerhalb des Warenkorbs geschehen oder durch Hinzufügen von Services während der Anzeige des Baskets. Ebenfalls kann die Fulfillment-Methode und Lieferadresse

angepasst werden, wodurch sich die Lieferkosten ändern können. Somit existieren einige Anwendungsfälle in denen eine Transaktion über beide Aggregate hinweg notwendig ist.

Gibt es Invarianten zwischen den Warenkorb und den CalculationResult?

Die Businessanforderung, dass das Berechnungsergebnis stets aktuell sein muss, wurde bereits gelockert. Verbleibend ist es zudem Notwendig aus rechtlichen Gründen, dass sich der Preis nach Initiierung des Bezahlvorgangs nicht ändert. Dieses Problem kann allerdings nicht auftreten, da der Warenkorb selbst nicht mehr manipuliert werden darf, somit bleibt der Gesamtpreis ebenfalls unberührt. Weitere Voraussetzungen existieren zum jetzigen Zeitpunkt nicht, jedoch müssen eventuelle zukünftige Anwendungsfälle berücksichtigt werden, ansonsten kann die Flexibilität der Anwendung gefährdet sein.

Zur Veranschaulichung kann Auswirkungen einer neuen Regelung untersucht werden. Beispielsweise wird angenommen, dass der Gesamtpreis eines Warenkorbs nicht über 20.000€ liegen darf. In diesem Fall würde eine Anpassungen des Basket auch eine Neukalkulation benötigen, wodurch die Abtrennung des CalculationResults sinnfrei wird. Allerdings kann eine mildere Form dieser Richtlinie keine negativen Effekt besitzen, indem die Prüfung erst bei der Initiierung des Bezahlvorgangs ausgeführt wird, da zu diesem Zeitpunkt beide Aggregates immer synchron sind und keine Änderungen mehr erfahren können. Weitere erdenkbaren zukünftige Businessanforderungen sollten berücksichtigt werden, bevor ein Neudesign der Applikation durchgeführt wird.

Grundsätzlich scheint eine Abspaltung der Berechnungsergebnisse vom Warenkorb durchaus plausibel zu sein. Diese Vorgehensweise der Analyse kann analog auf verschiedene Anwendungsfälle durchgeführt werden, um weitere Teile des Baskets zu finden, welche separat agieren können.

5.3.2 Herausschneiden der Checkout-Daten aus dem Basket-Aggregate

In dem Swimlane Diagramm 3.4 wurde das Hinzufügen der Kundendaten, Bezahlungsmethode und Fulfillment beschrieben. Dieser Prozess ist ein Hinweise darauf, dass diese Daten eventuell aus dem Basket-Aggregat genommen werden können. Die Summe der Attribute wird als 'Checkout-Daten' betitelt und beinhalten Kundendaten, Fulfillment, Rechnungs- und Lieferadresse. Nachteilig ist jedoch, dass dadurch die Value Objects zu einer Entity zusammengefasst werden, da die Rolle des Aggregate Root nur durch Entities verfüllt werden darf. Ähnlich zum vorgehenden Unterkapitel ist eine Untersuchung der Implikationen eines solches Aggregationsschnittes notwendig.

Anpassungen an diesen Daten dürfen nur durchgeführt werden, wenn der Basket im Status 'open' ist. Dadurch muss bei jedem API-Aufruf zuvor der Zustand überprüft und der Warenkorb-Datenbankeintrag gesperrt werden. Aufgrund von möglichen Interleavings kann ansonsten die Initiierung des Bezahlvorgang auf invalide Checkout-Data stattfinden. Weiterhin wird durch das Setzten eines neuen Fulfillments die Neuberechnung des Warenkorbs angestoßen und bei Übergabe eines Payments muss der PaymentProcess aktualisiert werden.

Vorteilhaft an dieser Variante ist, dass der Basket leichtgewichtiger wird und geringere Datenmengen transportiert werden müssen. Zusätzlich bleiben die negativen Auswirkungen der getrennten Collections in der Datenbank gering, da die Anpassungen der Checkout-Daten nur selten in den User Stories vorkommt.

5.4 Zusammenführung der vorgehenden Domain-Modelle

Anhand der vorgehenden Analyse kann ein neuer, zusammengefasster Aggregationsschnitt gebildet werden. Dieser hebt das CalculationResult, die CheckoutData und den PaymentProcess aus dem Basket heraus in ihre eigenen Aggregates, wie in Figur 5.5 dargestellt. Eine Abspaltung der BasketItems erfordert zu viele zusätzliche Datenbankoperationen, wodurch diese weiterhin unter dem Basket aufgehängt sind. Falls in zukünftigen Szenarien die parallele Bearbeitung der Warenkorbinhalte unerlässlich ist, kann weiterhin die Trennung der beiden Klassen voneinander in Betracht gezogen werden.

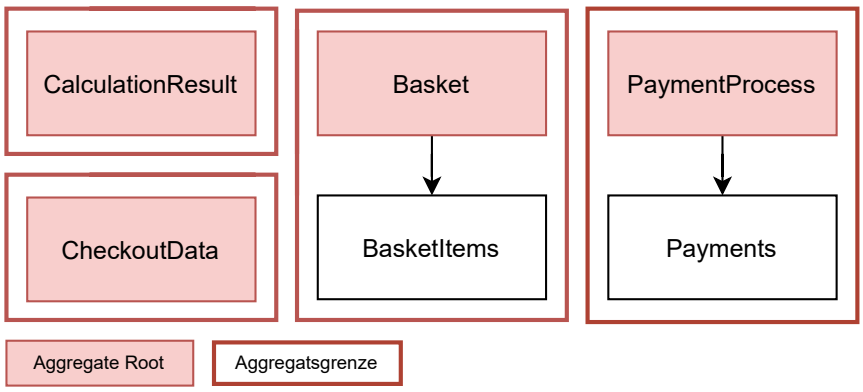

Abbildung 5.5: Aggregationsschnitt der Variante D

5.4.1 Aktualisieren von veralteten Datenständen

Ähnlich zu anderen Aggregationsschnitte benötigen viele Anwendungsfälle das Laden und Bearbeiten mehrerer Aggregates. Beispielsweise erfordert das Hinzufügen eines neuen BasketItems die Neuberechnung des PaymentProcesses und CalculationResults. Gleichermaßen muss dieser Prozess angestoßen werden, wenn das Fulfillment in der CheckoutData bearbeitet und die Versandkosten des Warenkorbs sich dadurch ändern. In einer solchen Situation sollte die Neuberechnung allerdings nicht sofort stattfinden, ansonsten gehen die Performance-Verbesserungen durch eine Kalkulation erst bei Bedarf verloren. Aus diesem Grund wird eine Zusatzinformation benötigt, um zu indizieren, dass der aktuelle Berechnungswert veraltet ist. In der konkreten Implementierung wurde zu diesem Zweck ein 'Outdated'-Wahrheitswert in dem Basket- und CheckoutData-Aggregate hinterlegt. Bei Zustandsänderungen, welche den Gesamtwert des Warenkorbs beeinflussen wird dieser auf 'wahr' gesetzt. Sobald ein CalculationResult des dazugehörigen Baskets aus der Datenbank geladen wird, findet auch eine Neuberechnung statt, sofern der 'Outdated'-Wert wahr ist, welche zugleich auch den PaymentProcess aktualisiert.

5.4.2 Dependency Injection von Services in Domain-Driven Design

Zur Durchführung der Applikationslogik werden Services und Repositories benötigt. Dies stellt die Frage, an welchen Stellen die Funktionsaufrufe im Code stattfinden und wie die Services an das Objekt weitergegeben wird. In diesem Proof-of-Concept wird für die Initiierung von Objekten das Prinzip der Dependency Injection verwendet. Ein Framework erstellt erforderlichen Objekte durch ihren

Konstruktor indem die Parameter ebenfalls injiziert werden. Dadurch entsteht ein rekursives Muster bis alle notwendigen Objekte erzeugt sind.

Injektion des Services in ein Aggregate durch das Repository

Eine Realisierungsmöglichkeit ist die Haltung und Aufrufen einer Referenz im Aggregate selbst. Folglich muss beim Lesen der Datensätze aus der Datenbank alle relevanten Service durch das Repository in das Aggregate injiziert werden. Im kurzen Beispielcode 5.2 wird dieser Gedanke verdeutlicht.

```
class Aggregate(val someData: Data) {
    lateinit var domainService: SomeService

    fun inject(domainService: SomeService) {
        this.domainService = domainService
    }

    fun executeBusinessLogic() {
        domainService.doStuff(someData)
    }
}
class AggregateRepository(val someService: SomeService) {
    fun load(id: Id): Aggregate {
        val aggregate = searchInDatabase(id)
        aggregate.inject(someService)
        return aggregate
    }
}
```

Listing 5.2: Injektion des Services in ein Aggregate durch das Repository

Jedoch hat sich diese Methodik bei der Implementierung als problematisch erwiesen. Einerseits ist es fragwürdig, ob Datenklassen aus Programmiersicht überhaupt Referenzen auf Services halten sollten, da äußere Abhängigkeiten in das innere Datenmodell getragen werden. Davon abgesehen erhöht sich stark die Wahrscheinlichkeit auf eine 'Circular Dependency' (dt. Zirkelbezug) innerhalb der Repositories, weil sie selbst alle Services besitzen müssen, um diese den Aggregate zu injizieren. Das bedeutet konkret, dass die Repositories so miteinander in Abhängigkeit stehen können, sodass ein endloser Zyklus während der Dependency Injection entsteht. Verdeutlicht wird dieser Effekt anhand eines sachbezogenen Anwendungsfalles in Abbildung 5.6. In dem Beispiel benötigt das 'CheckoutDataRepository' das 'BasketDataRepository' zur Initiierung und anders herum erfordert das 'BasketDataRepository' das 'CheckoutDataRepository'. Dadurch ist es unmöglich ein Objekt der jeweiligen Klasse zu erzeugen und die Applikation ist nicht mehr ausführbar. In der Checkout-Software stehen die Aggregates in enger Bindung zueinander und benötigen zur Validierung somit ihre gegenseitigen Repositories. In Variante A kann ein solcher Zyklus nicht auftreten, da nur ein Repository existiert.

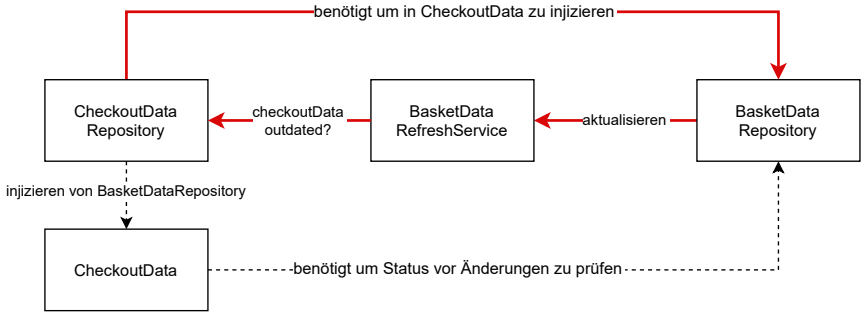

Abbildung 5.6: Beispiel einer Circular Dependency

Auslagerung der Referenz in einen Domainservice

Ein anderer Lösungsansatz kann das Extrahieren einer Teilfunktionalität aus dem Aggregat in einen dafür vorgesehenen Domainservice darstellen. Das Listing 5.3 implementiert einen Service zur Validierung einer Invariante.

```
class DomainService(someService: SomeService) {

    // Kann unabsichtlich umgangen werden, durch direkten Aufruf von aggregate.doStuff()
    fun doStuff(aggregate: Aggregate) {
        if (someService.isActionValid()) {
            aggregate.doStuff()
        }
    }

    // Kann nicht direkt umgangen werden, da doSomeOtherStuff einen Parameter erwartet
    fun doSomeOtherStuff(aggregate: Aggregate) {
        val data = someService.getData()
        aggregate.doSomeOtherStuff(data)
    }
}
```

Listing 5.3: Auslagerung der Referenz in einen Domainservice

Die Wahrscheinlichkeit auf eine Circular Dependency wird hierbei gesenkt, da die Injektionen in verschiedenen Klassen verteilt stattfindet. Nachteilig kann weiterhin aus einem anderen Codeabschnitt 'aggregate.doStuff()' aufgerufen werden ohne Durchführung der Validierung. Die Konsistenz des Aggregatzustandes ist somit gefährdet und anhand einer unvorsichtigen Änderung des Quellcodes können neue Bugs entstehen. Unproblematisch sind Serviceaufrufe, welche lediglich Daten erzeugen und diese den Aggregate weitergeben.

Übergabe der Referenz an das Aggregate als Parameter

Ein Aggregate sollte stets selbst alle wesentlichen Invarianten überprüfen, sodass die Ausführung von invaliden Aktionen unterbunden ist. Um dies zu gewährleisten, muss der Service direkt aus dem Aggregate aufgerufen werden, weswegen dieser eine Referenz auf das gefragte Objekt besitzen muss. Als letzte Realisierungsmöglichkeit in Figur 5.4 kann der Service als Parameter übergeben werden.

```
class DomainService(someService: SomeService) {
    fun doStuff(aggregate: Aggregate) {
        aggregate.doStuff(someService)
    }
}
class Aggregate {
    fun doStuff(someService: SomeService) {
        if (someService.isActionValid()) {
            ...
        }
    }
}
```

Listing 5.4: Übergabe der Referenz an das Aggregate als Parameter

Dies ermöglicht die Kombination der Vorteile der beiden anderen Varianten, jedoch werden die Funktionssignaturen aufgebläht. Aus Sicht der Skalierbarkeit und Fehleranfälligkeit ist generell dieser Ansatz zu bevorzugen, aus welchem Grund auch die Implementierung des Proof-of-Concepts die Domainservices als Parameter übergibt.

5.4.3 Bewertung des verkleinerten Aggregationsschnitt

Die finale Evaluation ist ein Resultat der vorgehenden theoretischen Überlegungen, ein Vergleich mit anderen Varianten des Aggregationsschnitts und ihren tatsächlichen Implementierungen.

Innerhalb eines solchen Domain-Modells ist es möglich, die Kalkulation des Warenkorbs erst bei expliziter Abfrage der Ergebnisse erfolgen, wodurch insgesamt unnötige Berechnungszeit eingespart werden kann. Ein solches Vorgehen erlaubt es bei einer hohen Auslastung der Systeme multiplikativ an Performance zu gewinnen. Zusätzlich müssen weniger Nutzlast zwischen den Clients und den Server gesendet werden, da die einzelnen Aggregates eine geringere Anzahl an Attributen beinhalten. Es muss allerdings genauer Untersucht werden, ob die zusätzlichen Datenbankabfragen diesen Effekt negieren oder dennoch insgesamt ein positiver Einfluss auf die Applikationsgeschwindigkeit zu verzeichnen ist.

- **Komplexität:** Viele Prozesse benötigen zum Abfragen von externen Informationen oder für die Einhaltung von Invarianten mehr als ein Aggregat, wodurch der Quelltext an Übersichtlichkeit verliert aufgrund der zusätzlichen Funktionsaufrufe. Dabei müssen die Referenzen auf die Repositories oder Service an die Aggregates weitergegeben werden, wodurch weiter die Übersichtlichkeit leidet. Das Transaktionsmanagement gewinnt ebenfalls an Relevanz, sodass Datensätze so kurz wie möglich gesperrt sind. Weiterhin muss für die Neuberechnung der Preise Nebeneffekte von Funktionen eingeführt werden zum Setzen des 'Outdated'-Wertes. *Insgesamt gewinnt die Applikation stark an Komplexität.*
- **Performance:**
- **Parallelität:**
- **Client-Freundlichkeit:**

6 Implementierung des Proof-of-Concepts

Im Verlaufe dieses Projekts wurde ebenfalls das Datenmodell mitsamt der Hexagonalen Architektur und Anwendungsfällen in einem Proof-of-Concept implementiert. Das Ziel dieser Software ist die Unterstreichung einer möglichen praktischen Umsetzung der, in dieser Arbeit beschriebenen, Anforderungen unter Anwendung der gewonnen Erkenntnisse.

Zu Beginn wurde ein neues Projekt aufgesetzt, welches benötigte Frameworks und Abhängigkeiten importiert, um Boilerplate-Code weitestgehend zu vermeiden. Die gesamte Software wurde mithilfe von Kotlin entwickelt und Komponenten wurden analog zu einer Hexagonalen Architektur in die drei Bereiche primäre Adapter, Applikationskern und ekundäre Adapter aufgeteilt. Innerhalb des Applikationskerns befindet sich jegliche Businesslogik, sowie die notwendigen Applicationservices, Domainservices und das Domainmodell entsprechend des Domain-Driven Designs. Der verwendete Aggregationsschnitt des POCs ist äquivalent zu einem einzelnen, großen Aggregate mit den Basket als Aggregate Root.

6.1 Design der primären Adapter

Primäre Adapter sind Komponente, welche den Datenfluss aufgrund von einem Signal eines externen Systems initiieren. Grundlegend sind sie die Kommunikationsschnittstelle zwischen Clients und der Software. Im Proof-of-Concept fallen hauptsächlich in diesen Bereich sogenannte Controller, welche zuständig sind für den Empfang von REST-API Anfragen und der Deserialisierung übergebener Daten, sowie der Serialisierung des Antwortinhaltes. Zu Beginn jedes Anwendungsfalles wird ein Controller durch den Touchpoint angesprochen. Die zuständige Komponente wird aus einen Zusammenschluss von aufgerufener URL und HTTP-Methode bestimmt.

Zur Implementierung dieser Funktionalität wurde das Framework 'Ktor' verwendet. Ein Controller beinhaltet lediglich Logik für den Empfang von Daten und der Formulierung zugehöriger Antworten. Alle Informationen, welche von außen stammen, müssen vor der Weitergabe an den Applikationskern in Datenmodelle des Applikationskerns umgewandelt werden. Ist dies nicht der Fall, besitzt der zentrale Teil der Software eine Abhängigkeit von außen. Dies würde sonst einen Bruch des Dependency-Inversion-Prinzips darstellen. Die Verwendung von sogenannten Data-Transfer-Object (DTO) ist jedoch gestattet, sofern diese in einem Modul des Applikationskern liegen und auch von diesen verwaltet werden. Ein konkretes Beispiel für einen Controller ist in Listing 6.1 abgebildet.

```
1 put("/basket/{id}/customer") {
2     val basketId = BasketId(parseUUIDFromParameter("id"))
3     val customer = call.receive<Customer>()
4     val basket = basketApiPort.setCustomer(basketId, customer)
5     call.respond(HttpStatusCode.OK, basket)
6 }
```

Listing 6.1: Beispiel eines Controllers zum aktualiseren von Kundendaten

- Zeile 1: Definiert die HTTP-Methode PUT und die URL für diesen Endpunkt
- Zeile 2: Auslesen der BasketId aus der URL als Pfadparameters
- Zeile 3: Deserialisierung des Payloads zu einem Customer-Objekt
- Zeile 4: Weitergabe der Daten an den zuständigen Applicationservice durch ein Interface
- Zeile 5: Antwort an die Anfrage mit HTTP-Status 200 und den Basket als Daten

Für jeden definierten Anwendungsfall wurde eine entsprechende Schnittstelle und Controller definiert. Im Kontextes dieses POCs umfassen die primären Adapter alleinig die Controller, da keine komplexe Datenumwandlung oder Validierung durchgeführt werden muss. Für jede Funktion, welche durch die Adapter aufgerufen werden, existiert eine zugehörige Funktion in einem Port. Die tatsächliche Implementierung der Schnittstelle wird mittels durch das Framework 'Koin' bestimmt und geladen. Dadurch bleiben Abhängigkeiten jederzeit austauschbar und unabhängig testbar. Beispielsweise kann die korrekte Funktionsweise eines Controllers überprüft werden, indem der Applicationservice durch ein Mock-Objekt ausgetauscht und der Aufruf seiner Funktionen ausschließlich simuliert wird. Somit erfahren die einzelnen Komponenten in Testfällen keine Beeinflussung durch die falsche Funktionsweise anderer Teile der Applikation und Fehlschläge können eindeutig einem einzelnen Codeabschnitt zugeschrieben werden.

6.2 Realisierung des Applikationskerns

Der Applikationskern stellt das Herz der Anwendung dar. Das maßgeblich Ziel einer Hexagonalen Architektur ist es, dass Zentrum komplett von äußeren Modulen zu entkoppelt. Somit können Adapter jederzeit ausgetauscht werden ohne einer Notwendigkeit die Businesslogik anpassen zu müssen. Die Kommunikation mit dem Kern geschieht ausschließlich über Interfaces, den Ports.

6.2.1 Definition von Applicationservices anhand ihrer Aufgaben

Entsprechend der Definition im Domain-Driven Design liegen Applicationservice eine logische Ebene höher als die Domainservice. Der Unterschied zwischen den beiden ist ihre Einsichten in die Domain. Ein Applicationservice darf kein direktes Domainwissen verwirklichen. Dazu zählen Invarianten oder die Umsetzung von Businessanforderungen. Meist beinhalten diese Service folgende Anweisungen:

- Starten und schließen einer Transaktion
- Laden von Aggregates aus Repositories
- Einfache Ablaufsteuerung
- Aufruf von Funktionen auf den Aggregates oder Domainservices
- Umsetzung von technologischen Komponenten wie Event-Listener oder Caching

Ein simples Beispiel bietet uns der BasketItem-Applicationservice. Dieser wird aufgerufen, sofern Änderungen an den BasketItems durchgeführt werden sollen. Der in Figur 6.2 dargestellte Code behandelt die Entfernung eines BasketItem aus dem Warenkorb.

```
1 override fun removeBasketItem(basketId: BasketId, basketItemId: BasketItemId): Basket {
2     return basketStorageService.findById(basketId).also { basket ->
3         logger.info { "Remove item $basketItemId from basket $basketId" }
4         basket.removeBasketItem(basketItemId)
5         basketRefreshService.recalculateBasket(basket)
6         basketStorageService.save(basket)
7     }
8 }
```

Listing 6.2: Eine Beispielsfunktion des BasketItem-Applikationservice

- Zeile 2: Laden des Baskets durch ein Domainservice. Theoretisch könnte hier auch das Repository direkt angesporchen werden, jedoch behandelt dieser Domainservice auch die Aktualisierung veralteter Produkt- und Preisdaten.
- Zeile 4: Aufruf einer Funktion auf den Basket zum Entfernen des übergebenen Eintrags
- Zeile 5: Aktualisierung des CalculationResults durch einen Domainservice
- Zeile 6: Speichern des Warenkorbs mit den abgeänderten Daten.

Theoretisch könnte argumentiert werden, dass die Kondition der Neuberechnung des Warenkorbs als Businessanforderung und somit als Domainwissen zählt. Jedoch wurde in diesem Fall der Funktionsaufruf eher dem Ziele der Ablaufsteuerung zugewiesen. Hingegen ist der BasketStorageService ein Domainservice, da nach dem Laden des Baskets aus dem Repository eine Aktualisierungsfunktion initiiert wird. Dies stellt mehr eine Anforderung im Sinne von 'Jeder geladene Basket wird aktualisiert sofern die enthaltenen Daten veraltet sind' dar.

6.2.2 Aufteilen der Businesslogik zwischen Domainservices und Datenmodell

Der verbleibende Codeanteil im Applikationskern wird auf das Datenmodell verteilt. Aufgrund der Tatsache, dass wir nur ein Aggregat und somit auch nur ein Aggregate Root besitzen, müssen jegliche Operationen auf dem Daten des Baskets und seinen Attributen durch eine Funktion im Basket selber geschehen. Zur Trennung der Funktionalität und Datenhaltung implementiert der Basket ein Interface, welche alle nach außen hin notwendigen Funktionen beinhaltet. Dadurch kann das konkrete darunterliegende Datenmodell ausgetauscht oder auch in Tests simuliert werden. Operationen, welche es erfordern tiefer gelegene Objekte anzupassen, werden durch eine Kette von Funktionsaufrufen umgesetzt. Beispielsweise führt der Warenkorb eine 'cancelPayment'-Methode auf den PaymentProcess aus, welcher wiederum die Anfrage an den betroffenen Payment-Eintrag weitergibt. Hierbei können alle notwendigen Validierungen durchgeführt werden, da das komplette Datenmodell geladen ist. Ein veranschaulichender Codeauszug des Proof-of-Concepts ist das Abändern der Fulfillment Methode in Listing 6.3.

```
1 fun setFulfillment(fulfillment: FulfillmentType, fulfillmentPort: FulfillmentPort) {
2     validateIfModificationIsAllowed()
3     val availableFulfillment = fulfillmentPort.getPossibleFulfillment(outletId)
4     throwIf(!availableFulfillment.contains(fulfillment)) {
5         IllegalModificationError("cannot select $fulfillment for outlet $outletId")
6     }
7     this.fulfillment = fulfillment
8 }
```

Listing 6.3: Setzen der Fulfillment Methode im Basket Aggregate

- Zeile 2: Überprüfung, ob der Basket aktuell Änderungen zulässt anhand des Status. Stellt eine Invariante des Datenmodells dar.

- Zeile 3: Laden aller verfügbarer Fulfillment Methoden für diesen Basket durch einen sekundären Adapter. Die Kommunikation mit dem Adapter erfolgt über einen Port.
- Zeile 4-6: Falls der neue Wert nicht unter den verfügbaren Fulfillment ist, wird der Aufruf zurückgewiesen und eine entsprechende Fehlermeldung an den Client durch den Controller geliefert.
- Zeile 7: Überschreiben des alten Wertes. Dieser Punkt wird nicht erreicht, wenn zuvor eine Businessanforderung gescheitert ist.

Funktionalitäten, welche nicht direkt einem Objekt zugewiesen werden können oder mehrere Aggregates betrifft sind in Domainservice zu implementieren. Sollte ein Prozess Abhängigkeiten zu anderen Services oder Ports besitzen, werden diese ebenfalls in Domainservice ausgelagert. Die Domainservices liegen auf der gleichen konzeptionellen Ebene wie das Datenmodell und dürfen Businesslogik enthalten. Beispielsweise existiert im Proof-of-Concept ein Domainservice für die Abwicklung des Bezahlverfahren. Der Code-Ausschnitt 6.4 enthält den Algorithmus zum Durchführen des Prozesses.

```
1  override fun executePaymentProcessAndFinalizeBasket(basketId: BasketId): Basket {
2      return basketStorageService.findById(basketId).also { basket ->
3          throwIf(!basket.isFrozen() || !basket.isPaymentInitialized()) {
4              IllegalModificationError("cannot cancel payment process if not initialized")
5          }
6          val externalPaymentRef = basket.getExternalPaymentRef()
7          paymentPort.executePayment(externalPaymentRef)
8          basket.executePayments() and basket.finalize()
9          basketStorageService.save(basket)
10         logger.info { "Basket payment was executed, saved and finalized" }
11         createOrderAfterFinalization(basket)
12     }
13 }
```

Listing 6.4: Ausführung des Bezahlvorgangs in einem Domainservice

- Zeile 2: Laden des aktualisierten Baskets aus dem BasketStorageService.
- Zeile 3-5: Weist die Durchführung zurück sofern der Basket nicht in den erwarteten Zustand ist. Dies kann auftreten, wenn die REST-API aufgerufen worden ist, ohne dass ein Bezahlprozess zuvor gestartet wurde.
- Zeile 4-6: Falls der neue Wert nicht unter den verfügbaren Fulfillment ist, wird der Aufruf zurückgewiesen und eine entsprechende Fehlermeldung an den Client durch den Controller geliefert.
- Zeile 7: Überschreiben des alten Wertes. Dieser Punkt wird nicht erreicht, wenn zuvor eine Businessanforderung gescheitert ist.

6.3 Anbinden externer Systeme und Datenbanken durch sekundäre Adapter

In dem in der Planungsphase erstellten Context-Diagramm 3.10 wurden verschiedenste Systeme aufgezeigt mit welchen die Anwendung zum Erfüllen ihrer Aufgaben kommunizieren muss. Für diesen Zweck wurden zwei Gruppen von Komponenten eingeführt. Die API-Service, welche einen Aufruf der externen Schnittstellen simulieren, und die Sekundären Adapter. Letztere implementieren eine vom Applikationskern definierte Schnittstelle, um den notwendigen Dienst bereitzustellen. Analog zu den primären Adapter, existieren im Anwendungskern keine direkten Abhängigkeiten zu diesem Teil der Software.

Ein Spezialfall sind hierbei die Adapter für das Erfragen der aktuellen Preis- bzw. Artikelinformationen. Aufgrund von Performance Verbesserungen wurden zwei Arten von Adapter implementiert. Einerseits der normal fungierende Service, welcher den dazugehörigen API-Service aufruft und anderseits dein Adapter mit Caching-Funktion, welcher zuerst den Preis bzw. Artikel aus dem Cache lädt und zurückgibt. Sollte der Cache-Eintrag veraltet sein, wird der eigentliche Adapter angesprochen, um ein aktuelles Ergebnis aus dem externen System zu erfragen. In Listing 6.5 ist die Klasse zuständig für das Aktualisieren des Preises abgebildet.

```
1  class CachedPriceAdapter(priceApiService: PriceApiService,
2                           private val priceRepository: CachedPriceRepository
3                           ) : PriceAdapter(priceApiService) {
4
5      override fun fetchPrice(priceId: PriceId): Price {
6          return priceRepository.getAndUpdateIfInvalid(priceId, fallback = {
7              super.fetchPrice(priceId)
8          })
9      }
10 }
```

Listing 6.5: Preisadapter mit Caching-Funktion

- Zeile 1-2: Der Cache-Preisadapter hat eine Abhängigkeit zum normalen Preisadapter und zu einem Repository zum abrufen des zwischengespeicherten Preises
- Zeile 6: Abfragen des Preises aus dem Cache-Repository. Sollte der Preis invalide sein wird Zeile 7 ausgeführt.
- Zeile 7: Weiterleitung der Anfrage an den eigentlichen Adapter. Das Ergebnis wird wieder im Cache abgelegt mit dem aktuellen Zeitstempel.

Zusätzlich zu diesen Adaptern gehören ebenfalls die Repositories in diesem Bereich der Hexagonalen Architektur. Sie managen den Zugriff auf die Datenbank und alle Funktionalitäten, welche in dieses Aufgabengebiet fallen. In dem Proof-of-Concept wird das gesamte Basket-Aggregate abgespeichert. Aus diesem Grund wird nur ein Repository benötigt, welches eine grundlegende Suchfunktion mithilfe der BasketId und eine Speicherfunktion anbietet. Abgesehen davon gehören noch die zwei Caching-Repositories zu dieser Gruppe. Zur Veranschaulichung wird in dem Quelltext-Block 6.6 die verwendete Funktion 'getAndUpdateIfInvalide' aus Zeile 6 des Listings 6.5 ebenfalls vorgezeigt.

```
1  override fun getAndUpdateIfInvalid(key: K, fallback: () -> E): E {
2      val cachedEntity = getCacheable(key)
3      return when (cachedEntity != null && isValid(cachedEntity)) {
4          true -> cachedEntity.payload
5          false -> fallback().also { this.put(mapping(it)) }
6      }
7  }
```

Listing 6.6: Abstraktes Repository zum Caching von Daten

- Zeile 2: Laden des Objektes aus dem Cache anhand ihrer Id.
- Zeile 3: Sollte das Objekt für die übergebende Id vorhanden und noch nicht veraltet sein, wird es dem Aufrufer zurückgegeben
- Zeile 5: Ist diese Bedingung nicht erfüllt, wird die übergebene Aktualisierungsfunktion ausgeführt und das Ergebnis zugleich mit den aktuellen Zeitstempel in den Cache geschrieben.

7 Fazit und Empfehlungen

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Literatur

[1] Eric Evans. *Domain-driven design: Tackling complexity in the heart of software*. Upper Saddle River, NJ: Addison-Wesley, 2011. ISBN: 9780321125217.