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

Codebeispiel-Verzeichnis

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.
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.
Lazy Loading	Das Laden von Daten aus einem Datenspeicher oder sonstigen Quellen wird erst durchgeführt, sobald diese benötigt werden, wodurch unnötiges Laden minimiert wird..
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.

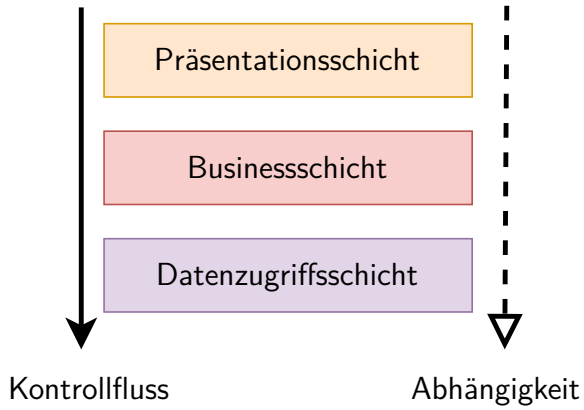

Abbildung 2.1: Beispielhafte Darstellung einer Drei-Schichtenarchitektur

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.

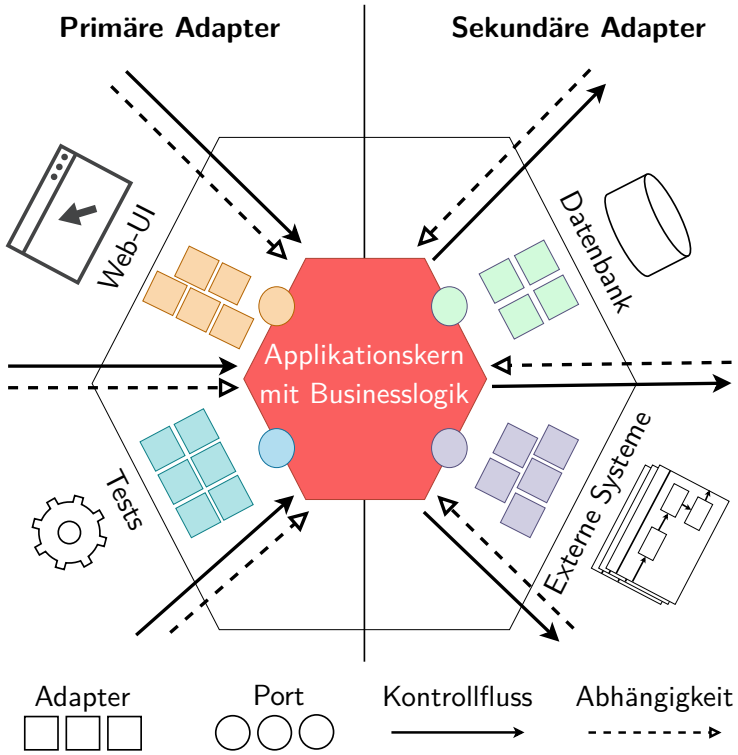

Abbildung 2.2: Grundstruktur einer Hexagonalen Architektur

- **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

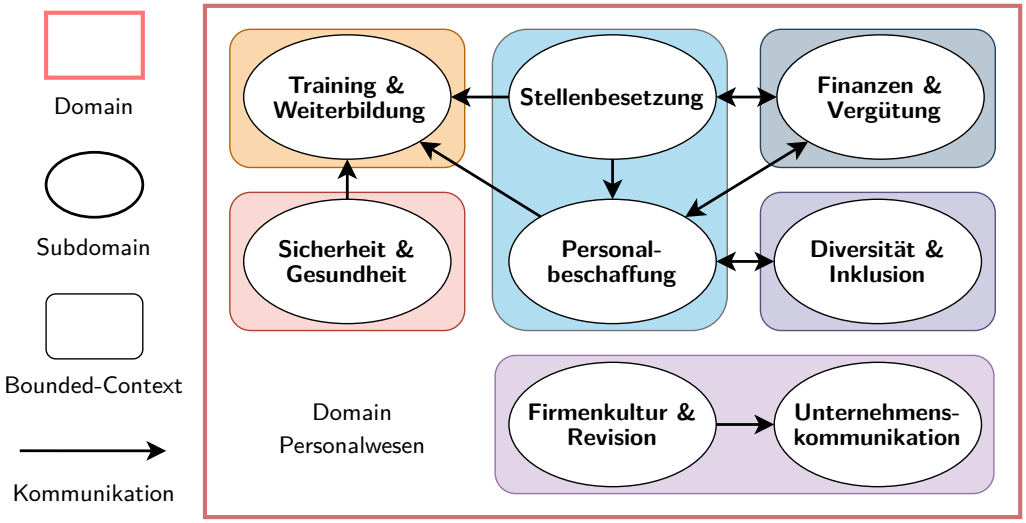

Abbildung 2.3: Beispiel einer Context-Map anhand des Personalwesens einer Firma

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 Aktivitätsdiagramme 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.

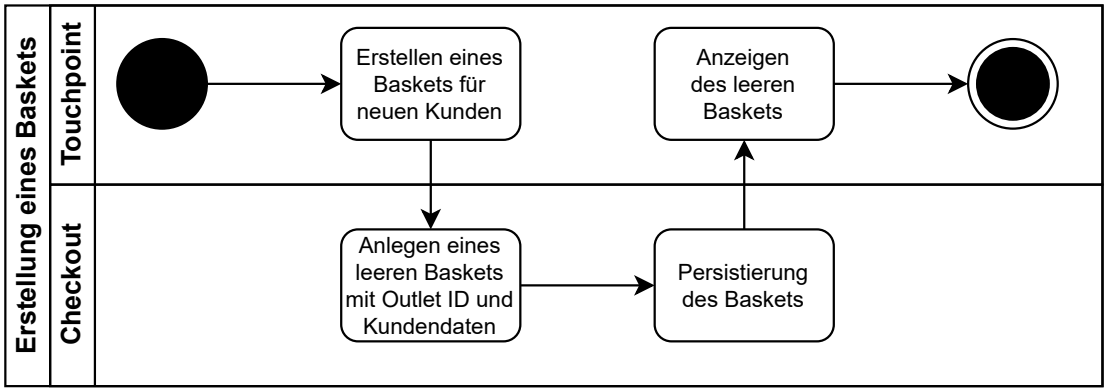

Abbildung 3.1: Aktivitätsdiagramm für die Erstellung eines Baskets

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 Aktivitätsdiagramm 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 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.

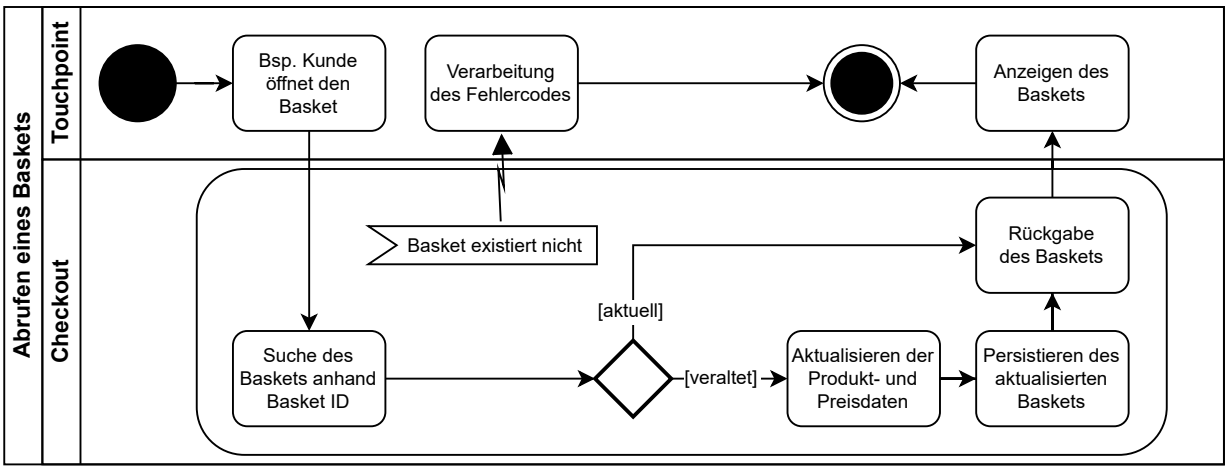

Abbildung 3.2: Aktivitätsdiagramm für den Abruf eines Baskets

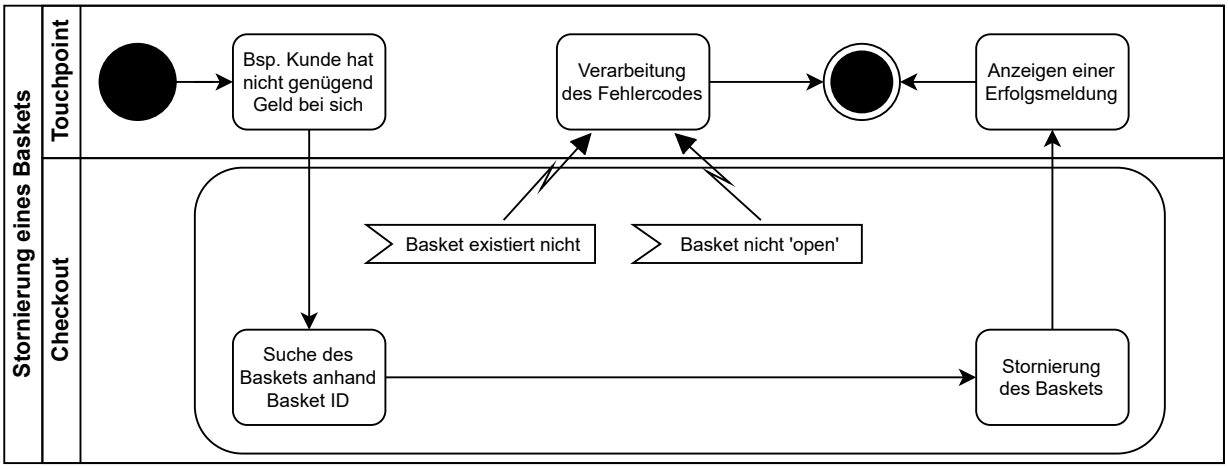

Abbildung 3.3: Aktivitätsdiagramm für die Stornierung eines Baskets

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 Aktivitätsdiagramm 3.4 veranschaulicht worden.

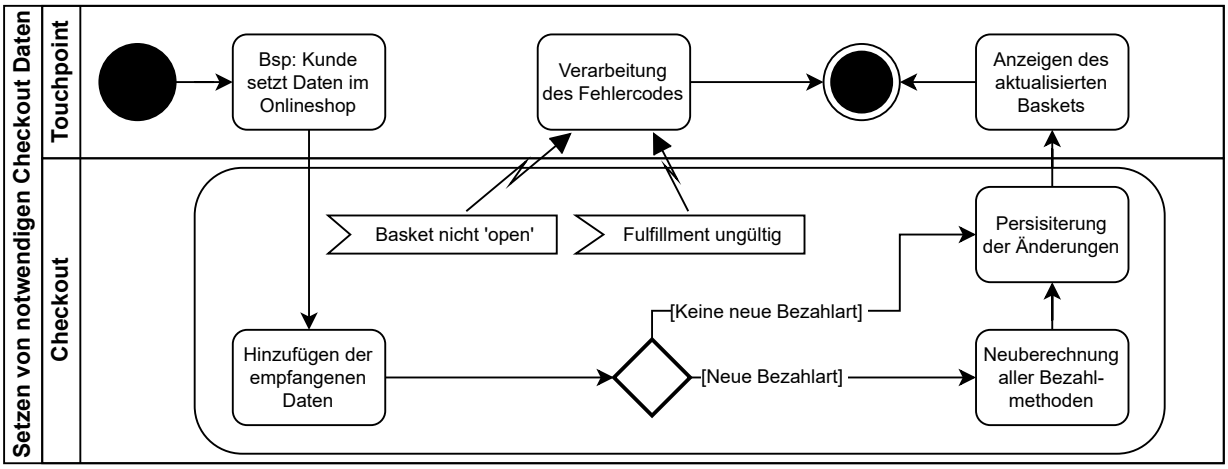

Abbildung 3.4: Aktivitätsdiagramm für das Setzen der Checkout Daten

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.

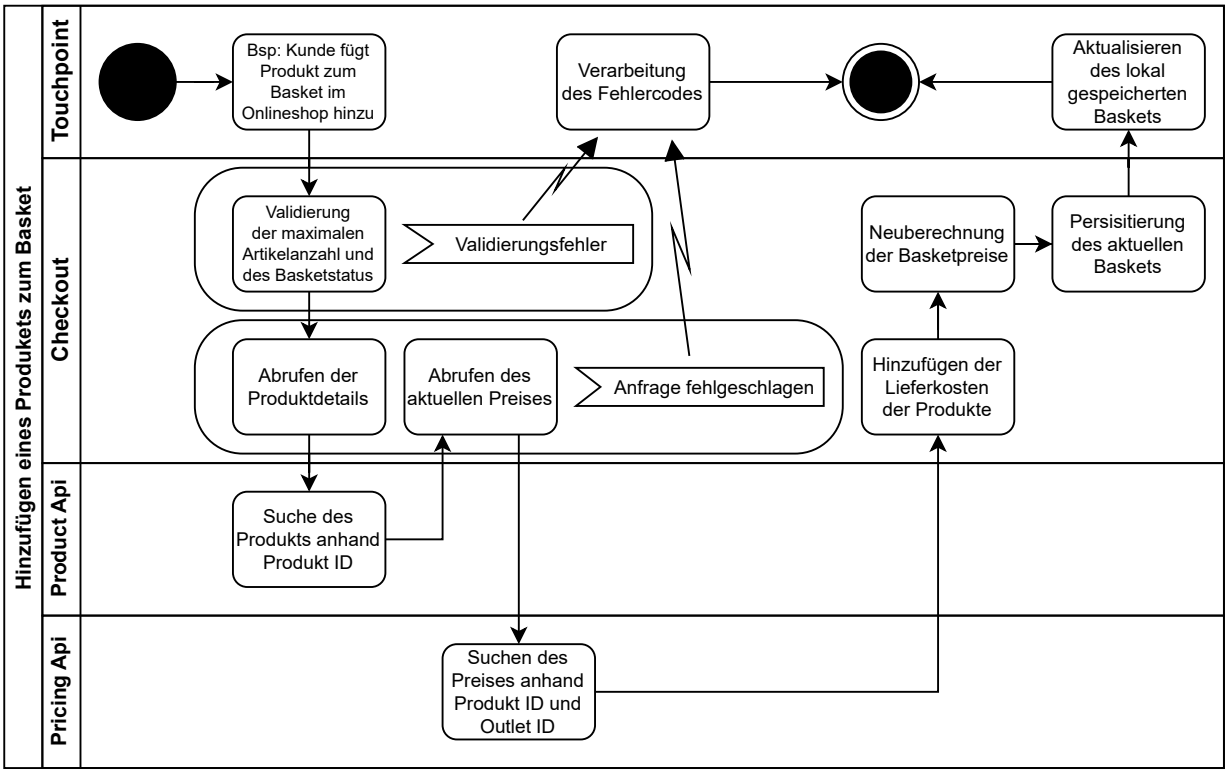

Abbildung 3.5: Aktivitätsdiagramm für das Hinzufügen eines Produktes

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.

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

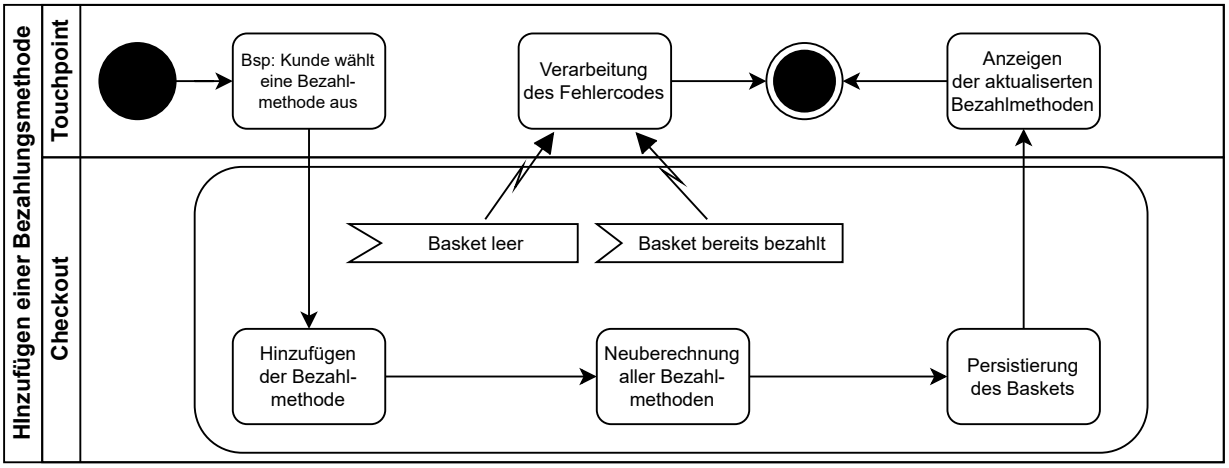

Abbildung 3.6: Aktivitätsdiagramm für das Hinzufügen einer Bezahlmethode

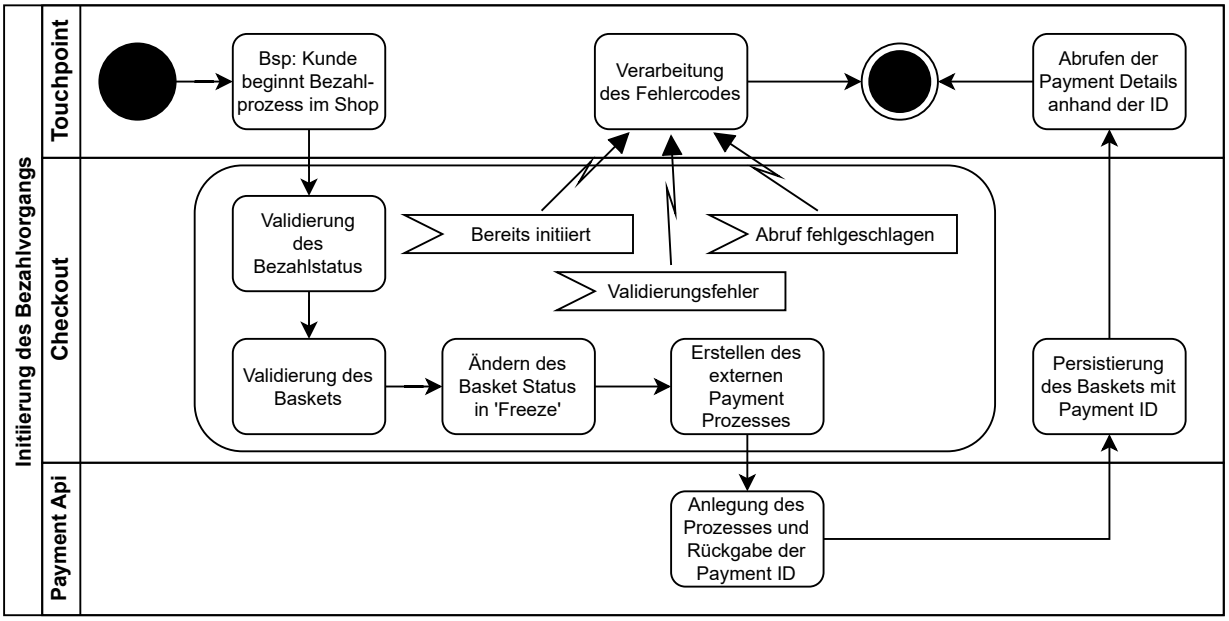

Abbildung 3.7: Aktivitätsdiagramm für das Initiieren des Bezahlvorgangs

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.

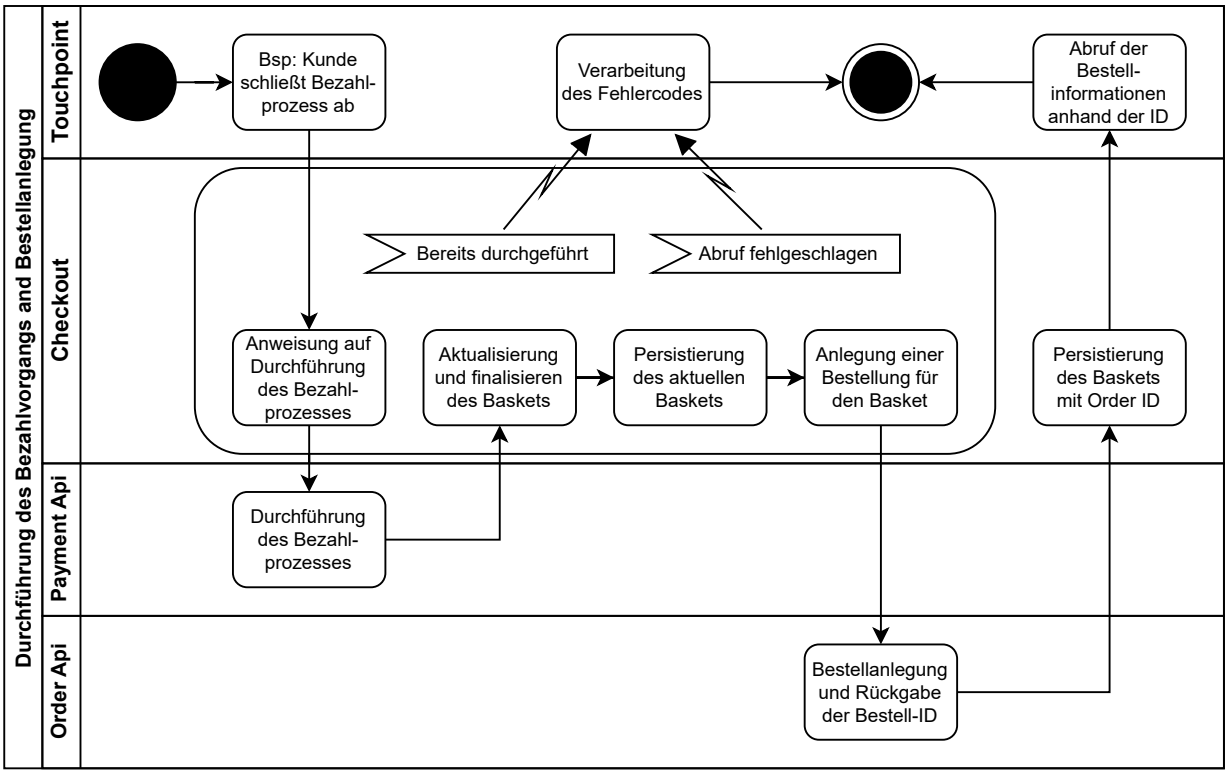

Abbildung 3.8: Aktivitätsdiagramm für das Ausführen des Bezahlvorgangs

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 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.

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

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.

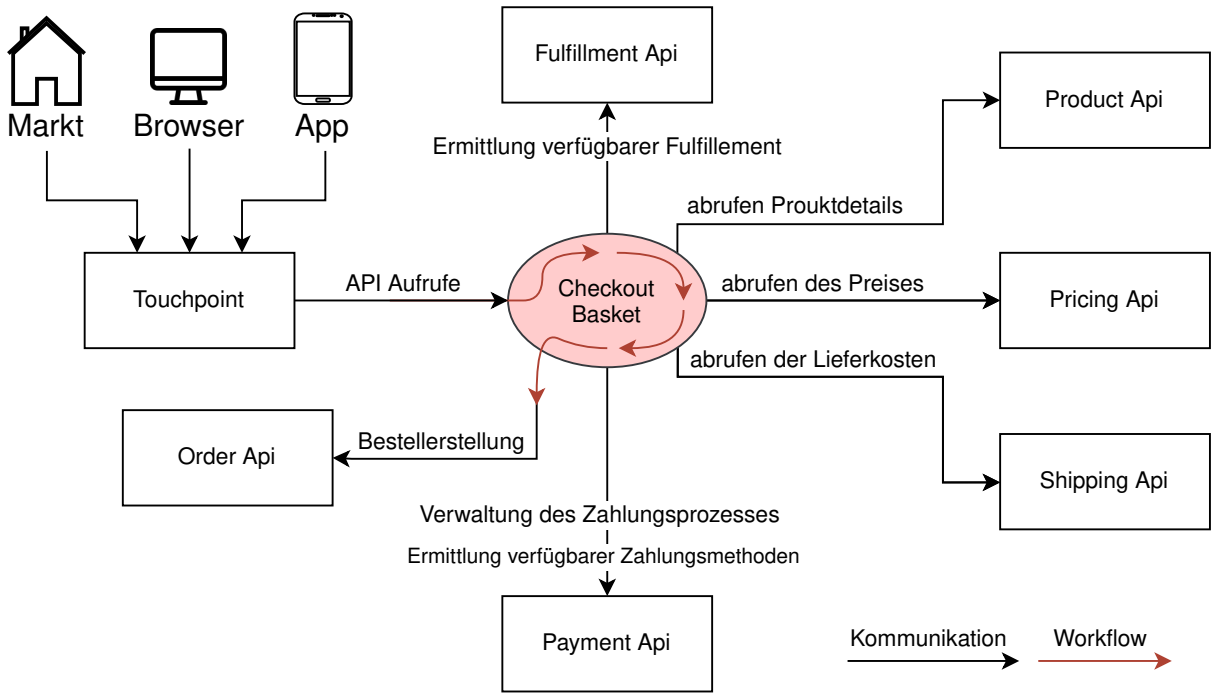

Abbildung 3.10: Context Diagramm der produktiven Checkout-Umgebung

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 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 zusätzlicher Basis der vorgehenden Analysen ein Proof-of-Concept der Checkout-Software entwickelt werden. Hierzu wird weiterhin das typisches Vorgehen von Domain-Driven Design verfolgt und zunächst der Domainumfang und die Ubiquitous Language definiert, gefolgt vom Erstellen des zentralen Domain-Modells.

4.1 Abgrenzung der Domain und Bounded Contexts mithilfe der Planungsphase

Aufgrund der ausführlichen Vorbereitung wurde die Domain bereits passiv festgelegt und analysiert. Beispielsweise beschreibt das Context-Diagramm 3.10 hierbei unsere Domaingrenzen. Eine Domain und die dazugehörigen Subdomains spannen den Problemraum über alle definierten Anwendungsfälle und Businessanforderungen auf. Die Größe der Domain ist entscheidend für die Bestimmung der Subdomains und des Bounded-Contexts. Den Checkout beispielsweise als eine zusammengehörige Domain anzusehen, würde darin resultieren, dass sich entweder nur ein Bounded-Context ergibt oder der Checkout selbst eine weitere Unterteilung erfahren muss. Diese Aufteilung fällt dementsprechend zu klein aus, da grundlegend je Bounded-Context nur maximal ein Team zuständig sein sollte. Folglich kann als nächstmögliche Eingrenzung der Checkout und alle abhängigen Systeme angesehen werden. Zu Beachten ist hierbei, sich nicht auf die konkreten Systeme zu fixieren, da sie eher der Lösungsebene zuweisbar sind, sondern logisch naheliegende Aufgaben in eine Gruppe zusammenzufassen. In den Zuständigkeitsbereich der Domain fallen unter anderem Anforderungen an der Abwicklung des Zahlungs- und Bestellprozesses, sowie Bereitstellung von Preis- bzw. Artikelinformationen. Hierfür muss ebenfalls eine Verwaltungsmöglichkeit für diese Daten bereitgestellt werden. Die Abgrenzungen der Bounded-Contexts ist durch die jetzigen Überlegungen und die bereits bestehenden Architektur vorgegeben, wodurch das vorgehende Context-Diagramm 3.10 ebenfalls als Context-Map fungieren kann.

4.2 Festlegen einer Ubiquitous Language

In der Kommunikation zwischen den Business und Entwicklern kann es oft zu Missverständnisse kommen. Womöglich weil Informationen, Einschränkungen oder Prozesse ausgelassen bzw. für selbstverständlich erachtet werden. Durch die klare Definition von gemeinsam verwendeten Begriffen und ihren Bedeutungen wird implizit notwendiges Wissen über die Domain und ihre Eigenschaften geschaffen. Viele dieser Fachbegriffe können für notwendige Anwendungsfälle wiederverwendet werden und machen die Personen, welche letztendlich die Businessanforderungen umsetzten sollen, mit der Domain vertraut. Da ein

Team mit geringem Domainwissen auch die Korrektheit der Software gefährdet, stellt die Ubiquitous Language ein wichtiger Meilenstein in Domain-Driven Design dar.

In Zusammenarbeit mit dem 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 sind zahlreiche Begriffen bereits definiert und helfen bei der Erstellung einer solchen Dokumentation. Eingeklammerte Wörter beschreiben Synonyme zu dem vorangestellten Ausdruck.

Ubiquitous Language des Domain-Modells:

- **Basket (Warenkorb):** Repräsentiert den gesamten Warenkorb mit allen Artikel, Kundeninformationen, Preisen etc.
- **Basket Status:** Stellt den aktuellen Zustand des Baskets dar, welcher sich an den Prozessen orientiert. Kann die Werte 'open', 'frozen', 'finalized' und 'canceled' annehmen. Der Startzustand ist hierbei 'open'.
- **Customer (Kunde):** Ein Endkunde des Onlineshops oder im Markt. Kann eine zivile Person oder Firma sein. Ein nicht-eingeloggter Kunde im Onlineshop wird der zugehörige Warenkorb durch seine eindeutige Session-ID zugewiesen.
- **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.
- **BasketItemId:** Eine, innerhalb eines Baskets, eindeutige Referenz auf ein Artikel des Warenkorbs. Wird aus technischen Gründen benötigt, um einen Eintrag zu bearbeiten oder entfernen.
- **Net Amount (Nettobetrag):** Ein Nettobetrag mit Währung.
- **VAT (Steuersatz):** Der Steuersatz eines zugehörigen Nettobetrag.
- **Gross Amount (Bruttobetrag):** Der Bruttobetrag eines Preises errechnet aus dem Steuersatz und 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 (Zahlungsprozess):** Beinhaltet alle relevanten Informationen für das Bezahlen eines Warenkorbs, wie ausgewählte Zahlungsarten und Beträge.
- **Payment:** Zahlung des Kunden inklusive Betrag und Zahlungsmethode, wie Barzahlung oder Paypal.
- **Order:** Bestellung eines Warenkorbs nach Abschluss des Zahlvorgangs, welche durch nachfolgende Systeme angelegt und verwaltet wird.

Ubiquitous Language der Businessprozesse:

- **Touchpoint:** Eine Komponente, welche mit der Checkout-Software interagiert, wie Kassensysteme im Markt, die Onlineshop-Seite oder Handy-App.
- **Basket Validation:** Durchführung einer Validierung des Baskets auf Inkonsistenzen oder fehlende jedoch notwendige Werte.
- **Basket Finalization:** Erfolgt automatisch nach erfolgreichem Zahlvorgang und setzt den Warenkorb in den Zustand 'finalized'. Danach folgt die Reservierung der Produkte im Basket und das Anlegen einer neuen Bestellung.
- **Basket Cancellation:** Stornieren des zugehörigen Baskets mithilfe des Zustandswechsel auf 'canceled'. Nach Cancellation dürfen keine weiteren Änderungen an den Basket durchgeführt werden. Der Warenkorb muss sich zuvor im Zustand 'open' befinden.
- **Basket Creation:** Explizite oder Implizite Erstellung eines neuen Baskets. Findet automatisch statt sofern noch kein Basket für den Customer existiert, sowie nach einer Basket Finalization.
- **Basket Calculation:** Das Berechnen der Summe von beinhalteten Preisen des Baskets, sowie die Kalkulation von Bruttobeträgen für alle Artikel. Beträge aus unterschiedlichen Mehrwertsteuern müssen 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, nachdem der Warenkorb auf invalide Zustände überprüft wird. Nur möglich auf einen offenen Basket, welcher Produkte und Payments enthält. Resultiert in den Zustand 'frozen', wodurch keine weiteren Inhaltsänderungen an den Warenkorb durchgeführt werden dürfen.
- **Payment Execution:** Durchführung des Zahlungsvorgangs eines gefrorenen Basket. Anschließend findet die Basket Finalization statt.

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 die Klassen designt.

4.3 Definition der Value Objects

Aufgrund der Simplizität und klaren Zuteilung der Value Objects lassen sich diese als leichtes bestimmen. 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. Der Basket stellt hierbei den Ausgangspunkt der Modellierung dar und die Ubiquitous Language unterstützt bei der richtigen Klassen-Benamung. Es wird auf eine schlankes Design im Vergleich zur Produktivanwendung geachtet, ohne dabei mögliche Aggregationsschnitte zu beeinflussen.

Basket:

- **BasketId:** Eindeutige Identifikation des Baskets zur Referenzierung durch die Touchpoints.
- **OutletId:** Eine Referenz zugehörig zu dem Markt oder Onlineshop, durch welchen der Warenkorb angelegt wurde. Unerlässlich für die Bestimmung von unter anderem Lagerbeständen, Lieferzeiten, Fulfillment-Optionen und Versandkosten.
- **BasketStatus:** Repräsentiert den aktuellen Zustands des Warenkorbs. Mögliche Werte sind OPEN, FROZEN, FINALIZED und CANCELED.
- **Customer:** Speichert Kundendaten (IdentifiedCustomer) oder Session-Informationen (SessionCustomer).
- **FulfillmentType:** Lieferart, wie PICKUP oder DELIVERY.
- **BillingAddress:** Adresse für die Rechnungserstellung.
- **ShippingAddress:** Adresse für die Warenlieferung.
- **BasketItems:** Liste aller enthaltenen Produkten im Warenkorb und ihren zugehörigen Informationen.
- **BasketCalculationResult:** Beinhaltet die berechneten Werte des Warenkorbs, wie Nettobetrag, Bruttobetrag und Mehrwertsteuer. 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:** Bindet alle Informationen zur erfolgreichen Abwicklung des Zahlungsprozesses.
- **Order:** Speichert eine Referenz auf die Bestellung für einen Basket. Wird erst nach Zahlungsabschluss befüllt.

Anschließend werden die im Basket enthaltenen Klassen ebenfalls mit der gleichen Vorgehensweise definiert, sofern sie nicht durch einen einfachen Text oder Aufzählungen darstellbar sind, bzw. ihre Inhalte keine nähere Erklärung erfordern:

//Kommentar: Nachfolgenden Abschnitt verkürzen? Auslassen von selbsterklärenden Werten?

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.
- **BusinessType:** Bestimmt ob Kunde als Business-to-Customer (B2C) oder Business-to-Business (B2B) gilt.

- **CompanyName:** Firmenname des Kunden. Kann optional angegeben werden oder ist verpflichtend für einen B2B-Kunden.
- **CompanyTaxId:** Steuernummer der Firma eines B2B-Kunden.

BasketItem:

- **Id:** Eindeutige Referenz auf den Warenkorbeintrag.
- **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 des Products 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 jeweiligen Prozentsatz und 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.
- **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 auf den Bezahlvorgangs im externen System. 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 der Zahlung.
- **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 Subtraktion von AmountSelected 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 Modell in mehrere Klassendiagramme unterteilt. In Figur 4.1 ist das Ergebnis dieses Unterkapitels abgebildet. Der *Customer* und *PaymentProcess* wurden aus Platzgründen in separate Klassendiagramme 4.2 und 4.3 verlagert. Anzumerken ist, dass bei fehlender Multiplizität eine Eins-zu-Eins Beziehung vorliegt. Speziell, ist der Kunde in diesem Kontext genau einem Warenkorb zugewiesen, da alleinig die Daten abgespeichert werden, nicht aber seine Kundennummer, wodurch keine Zuweisung zwischen Kunde und mehreren Warenkörben existiert.

//Kommentar: Im Anhang oder direkt darunter hinzufügen?

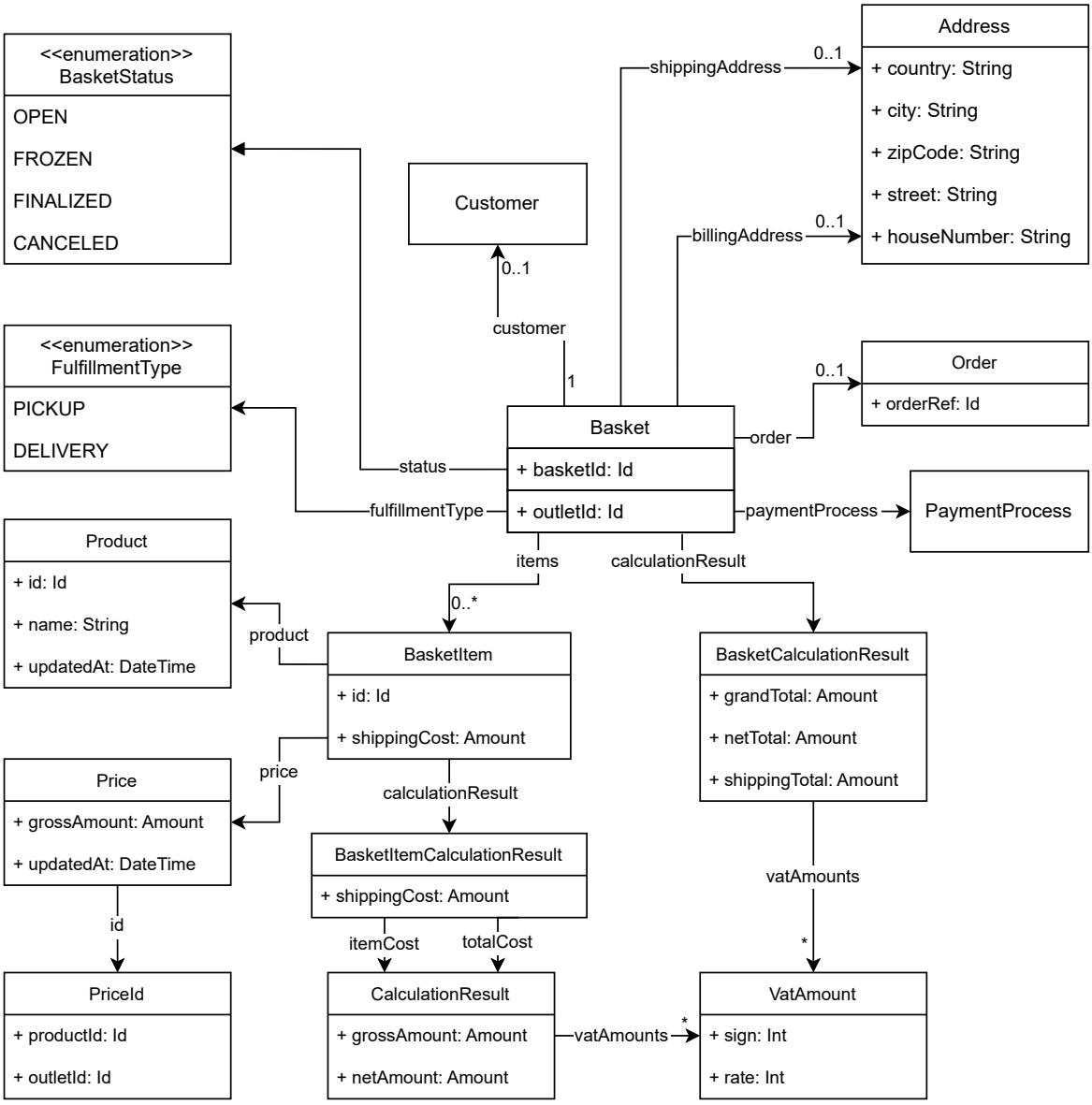

Abbildung 4.1: Klassendiagramm eines Baskets

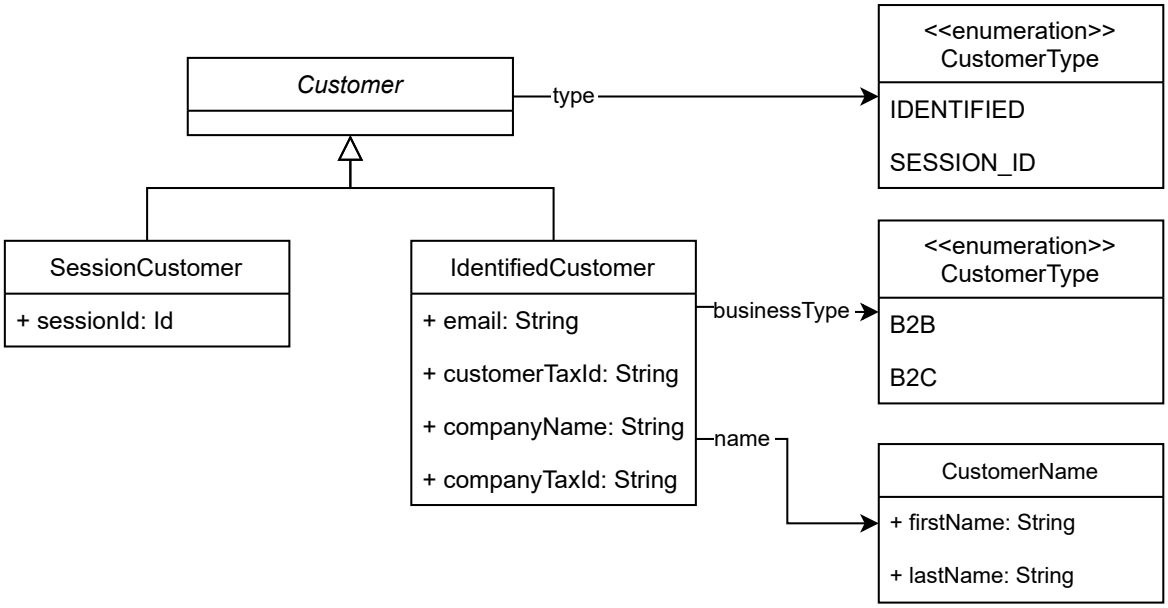

Abbildung 4.2: Zugehöriges Klassendiagramm des Customer Value Objects

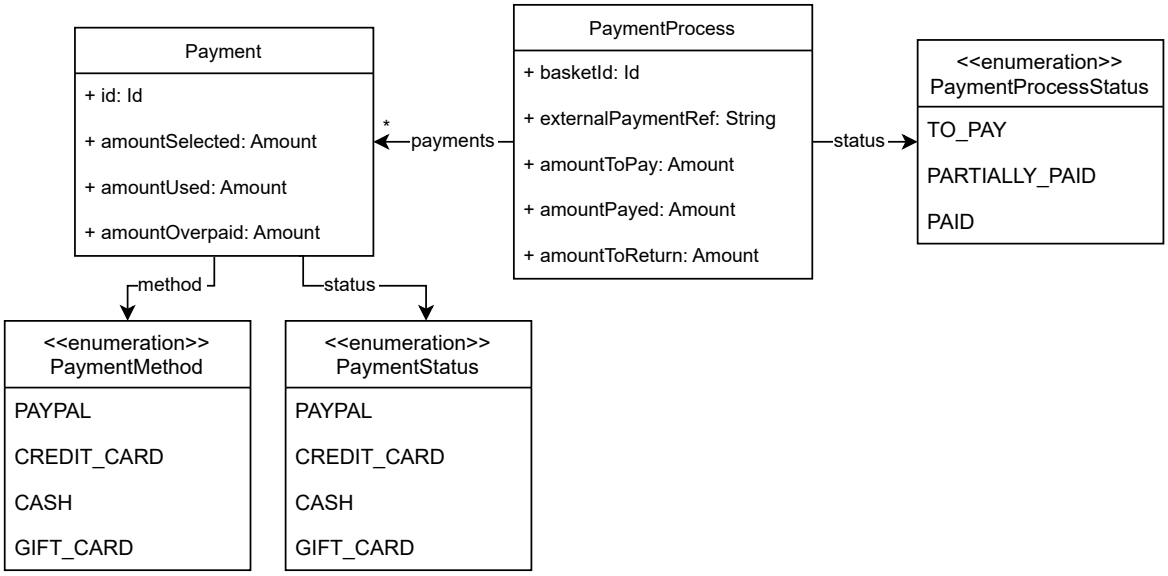

Abbildung 4.3: Darstellung des Payment Process als Klassendiagramm

4.4 Bestimmung der Entities anhand ihrer Identität und Lebenszyklus

Anhand der vorgehenden Sektion ist das Datenmodell nun vollständig definiert. Jedoch besteht weiterhin die Frage, ob die jeweiligen Klassen eine eigene Identität besitzen und somit als Entity designt werden müssen. Es existiert in Domain-Driven Design kein objektives Verfahren zur Bestimmung der Entities, da je nach Bounded-Context Datengruppierungen unterschiedliche Eigenschaften besitzen. Als Hilfestellung für diese Entscheidung können folgende grundlegende Unterscheidungsmerkmale und Richtlinien in Tabelle 4.4 verwendet werden.

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

Tabelle 4.4: Vergleich zwischen Value Object und Entity

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 Referenznummer. 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 diese Id. Aufgrund der geforderten Anwendungsfälle entsteht zugleich ein Lebenszyklus für die Instanzen eines Baskets und er durchgeht verschiedene Zustandsänderungen. 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 und können weiterhin als *Value Object* designt werden.
- **SessionCustomer:** Die Identifikation dieses Objekts geschieht über die SessionId. Dadurch ist ein SessionCustomer in der Gruppe der *Entities* aufzuhängen.

- **BasketItem:** Auf erstem Blick ist 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 Serviceangebote (in der Produktivumgebung) unterscheiden können. 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 als Entity oder Value Object definiert werden. Für den Proof-Of-Concept wurde das BasketItem als Entity festgelegt. Die Begründung hierfür ist die schiere Anzahl von Datenanpassungen und Operationen auf einem BasketItem, welche 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 diesem 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 eines Payment Processes 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. Zusammenfassend spricht jede Eigenschaft für ein Design als *Entity*.
- **Payment:** Ein Payment hat eine eindeutige Id, welche eine hohe Relevanz für den Ablauf des Bezahlprozesses und alle folgenden rechtlichen Prozesse hat. Dadurch ist weder der konkrete Betrag noch die Bezahlmethode bei der Identifikation wichtig. 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 auf unterschiedliche Weisen realisiert werden. Dies gilt ebenfalls für den Schnitt der Aggregates. Verschiedene Schnitte genießen zugleich andere Vor- und Nachteile. In diesem Kapitel werden mehrere mögliche Aggregationsaufteilungen untersucht und anhand von Kriterien, wie unter anderem Performance, Komplexität, Parallelität, Anwendbarkeit und Client-Freundlichkeit, bewertet.

5.1 Ein zusammengehöriges Basket-Aggregate als initiales Design

Das Design eines großen Aggregats fällt Entwicklern meist einfacher. Bei einer einzelnen, zusammenhängenden Struktur, durch welche unmittelbar Daten bearbeitet und Invarianten überprüft werden können, hält sich die Komplexität weitestgehend in Grenzen. Vor allem müssen kaum Überlegungen über die transaktionale Konsistenz getroffen werden, da die Transaktion das gesamte Datenmodell umspannend. Die erste Variante des Proof-Of-Concepts wurde mit diesem Design in Gedanken entwickelt und stellt das Grundgerüst für alle folgenden Konzepte dar. Folglich ist der Basket hier das Aggregate Root und alle verbleibenden Klassen sind diesem unterteilt. Zur vereinfachten Referenz auf die einzelnen Umsetzungsmöglichkeiten wird das Design als '*Variante A*' betitelt.

5.1.1 Performance von unterschiedlich großen Aggregates im Vergleich

Als Konsequenz eines großen Aggregates muss immer das ganze Aggregate geladen werden, da auf darunterliegende Datenstrukturen nur durch das Aggregate Root zugegriffen werden darf. Für die Aktualisierung von Daten tief im Aggregat müssen alle anderen Objekte ebenfalls aus der Datenbank gelesen werden. Die Auswirkungen dieser Tatsache kann mithilfe von Lazy Loading oder durch Einsatz einer dokumentenorientierten 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 diese Richtlinie anhand des vorliegenden Bounded-Contexts analysiert.

Untergliederung des Baskets in kleinere Bestandteile

//Kommentar: Obige Überschrift lassen oder entfernen?

Wie oben erwähnt, ermöglicht ein kleinerer Aggregationsschnitt das unabhängige Laden und Bearbeiten der Aggregates. In Betrachtung der Anwendungsfälle beziehen sich Aktionen meist auf einzelne Value Objects, BasketItems oder PaymentProcesses. Dementsprechend wäre das explizite Abfragen dieser Daten aus der Datenquelle effektiver. Dies ist in Domain-Driven Design jedoch nur möglich,

sofern diese Klassen die Rolle eines Aggregate Root einnehmen. Wird die *Variante A* unterteilt in ein Basket-, Payment und BasketItem-Aggregate (*'Variante B'*) können sie somit unabhängig voneinander gehandhabt werden. Eine kurze Übersicht über diesen neuen Aggregationsschnitt bietet Abbildung 5.1. Ohne Beachtung, welche Auswirkungen dieses neu überlegte Design auf andere Faktoren hat, ist eine erhöhte Performance anhand ersten Überlegungen zu erwarten. Die vermeintlich gewonnene Leistungsverbesserung wird allerdings aufgrund folgender Umstände minimiert.

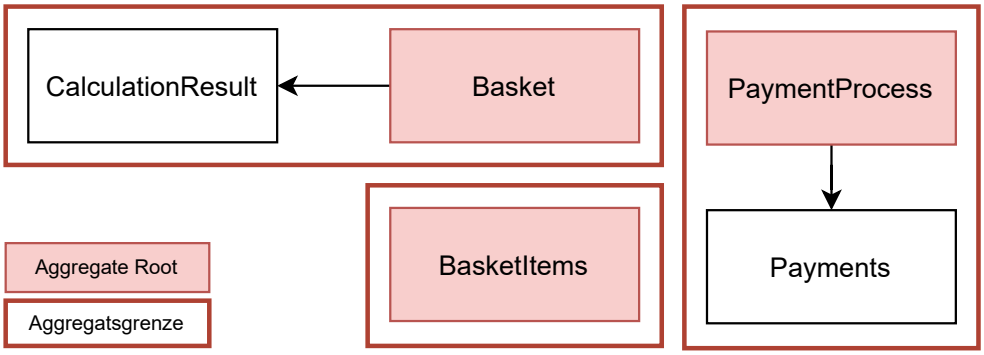

Abbildung 5.1: Aggregationsschnitt der Variante B

Invarianten über Aggregationsgrenzen hinweg

Wird dem Warenkorb ein neuer Artikel hinzugefügt, hat dies nicht nur das Anlegen eines Items, sondern auch die Neukalkulation des Baskets zu folge. Sofern eine Trennung zwischen Basket und BasketItem vorliegt, muss dennoch der ganze Warenkorb aus der Datenbank geladen werden, andernfalls ist es nicht möglich den neuen Gesamtpreis zu ermittelt. Innerhalb eines Checkout-Kontextes existieren viele dieser klassenübergreifenden Businessanforderungen, weshalb die Kopplung des Datenmodells oftmals eine wirklich unabhängige Datenanpassung eines Aggregates verhindert.

Durch eine genauere Untersuchung der Anwendungsfälle kann eine Abwägung getroffen werden, wie viele Prozesse tatsächlich unabhängig abzuarbeiten sind. 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, wodurch die Effektivität des Aggregationsdesigns ebenfalls an die Bedürfnisse der Clients gekoppelt sind. Das Stornieren des Baskets und Setzten der Checkout-Daten ist hingegen positiv von dieser Anpassung betroffen, denn diese verwaltet alleinig der Warenkorb. Die meisten API-Anfragen beziehen sich allerdings auf beinhaltete BasketItems und die Verwaltung des Bezahlvorgangs, welche Wechselwirkungen mit dem Warenkorb oder sein zugehöriger Gesamtpreis besitzen. Letztendlich sind nur eine Bruchzahl der Anwendungsfälle mithilfe von Variante B isoliert abschließbar, somit steigt die Anzahl der benötigten Datenbankoperationen auch bei kleineren Aggregates und die damit verbundene Bearbeitungszeit wieder an.

In dem Buch '*Domain-driven design: Tackling complexity in the heart of software*' wird zusätzlich auf eine Richtlinie hingewiesen, dass bei korrekten Aggregationsdesign eine Transaktion maximal ein Aggregate bearbeiten sollte [1, S. 354]. Dies führt zu einem Konflikt mit den vorgehenden Businessanforderungen. Beispielsweise wäre es nicht möglich bei der Initiierung des Zahlungsvorgangs zeitgleich den Warenkorb einzufrieren. Ein möglicher Lösungsansatz stellt hierbei das Verwenden von eventueller Konsistenz dar. Konkret findet die Bearbeitung der Aggregates zeitversetzt voneinander statt,

weshalb eine Zeitspanne existiert in welcher der Datensatz einen invaliden Stand besitzt. Angewandt an den vorgehenden Anwendungsfall bedeutet dies, dass zwar der Warenkorb eingefroren, allerdings der Bezahlvorgang noch nicht gestartet ist. Verbundene Implikationen mit eventueller Konsistenz werden in einem folgenden Kapitel besprochen.

Einfluss des verwendeten Datenbanksystems auf den Aggregationsschnitt

Das Design einer Software soll stets, so weit wie möglich, isoliert 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 rechtfertigt. Folglich wird das Aggregationsdesign aus Sicht der Datenbank bewertet.

Laut Definition erhält jedes Aggregate bzw. Aggregate Root seine eigene Tabelle in der darunterliegenden, relationalen Datenbank. Deshalb existiert in *Variante B* mindestens eine Tabelle für den Basket, BasketItems und PaymentProcess. Hingegen kann bei *Variante A* der PaymentProcess in die Basket-Tabelle hinzugefügt werden, da eine Eins-zu-Eins Relation vorliegt. Weitergehend benötigen beide Aggregationsschnitte aufgrund der Eins-zu-N Beziehung zwei Tabellen für Basket und BasketItem. Dementsprechend muss beim Abruf eines kompletten Baskets in *Variante A* im Vergleich zu *Variante B* weniger Ladevorgänge durchgeführt werden. Das kleiner Aggregationsdesign ist bei isolierten Modifikationen von Aggregates aus Sicht der Datenbankoperationen jedoch vorteilhafter. Je nach Businessanforderungen können sich diese Aspekte ausgleichen und lediglich einen geringen Effekt auf die generelle Performance der Anwendung besitzen.

Sollte die verwendete Datenbank allerdings einen dokumentenorientierten Ansatz verfolgen, gilt vorheriger Absatz nur noch bedingt. Hierbei benötigt *Variante B* weiterhin pro Aggregate eine eigene Collection. *Variante A* kann das komplette Datenmodell hingegen in einem einzigen Eintrag persistierten. An sich ist der einzelne Datensatz umfangreicher, allerdings ist dies durch die eingesparten Datenbankoperationen dennoch effektiver. Daher würde bei einem Umbau der Aggregates für die meisten Anwendungsfälle eine einzelne Suchanfrage in mehrere abgewandelt werden, wodurch bemerkbare Auswirkungen auf die Antwortzeit unserer Applikation entstehen können.

Anfangs wurde beschrieben, dass eine Technologie idealerweise 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-Unterschiede in Grenzen, wohingegen dies bei einen umfangreicher Aggregationsschnitt mit einer dokumentenorientierten Datenbank nicht gewährleistet werden kann. Anhand eines konkreten Lasttests wird in einem späteren Kapitel dieser Effekt genauer untersucht und bewertet.

5.1.2 Parallele Bearbeitung eines großen Aggregates

Ausgehend von *Variante A* resultiert eine Bearbeitung des Baskets in der Speicherung des kompletten Warenkorbs in der Datenbank. Bei Schreibprozessen können Anomalien auftreten, wodurch die Operation abgebrochen werden muss. Eine mögliche Anomalie ist das sogenannte 'Lost Update'-Problem. Es kann auftreten, wenn zwei Transaktionen den gleichen Datensatz zeitgleich bearbeiten. Anfangs besitzen beide den selben Startzustand, jedoch beim Zurückschreiben ihrer Ergebnisse stößt der letztere von beiden Transaktionen auf einen nun neueren Stand. Hierbei muss diese Transaktion erkannt und

abgebrochen werden, da sonst die zuvor geschehene Datenänderung der ersteren Transaktion verloren gehen würde.

Konkretisiert kann dieses Problem auftreten, wenn beispielsweise zwei Personen einen neuen Artikel dem selben Warenkorb hinzufügen. 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 Datensatz mit nur einem Artikel. Das System aktualisiert den Eintrag in der Datenbank zuerst mit dem Handy. Aufgrund dessen, dass die zweite Anfrage mit einem leeren Warenkorb initiiert worden ist, wird der Datensatz ebenfalls mit nur einen Artikel persistiert. Letztendlich enthält der Basket nun nur einen Fernseher statt den eigentlichen zwei Artikeln. Alternativ kann die letztere Operation anhand des neuen Zustandes erneut durchgeführt werden oder der Datensatz wird beim Laden für weitere Anfragen gesperrt, sodass ein solches Problem nicht auftreten kann. Diese Lösungsansätze nennen sich optimistisches bzw. pessimistisches Sperrverfahren.

Innerhalb eines großen Aggregates kann es bei zeitgleichen Aktionen vermehrt zu Schreibanomalien kommen, wodurch sie sich für parallele Bearbeitung nicht eigenen. In *Variante B* ist es möglich BasketItems zum selben Zeitpunkt anzupassen, da sie in der Datenbank unabhängig voneinander persistiert werden. Existieren Businessanforderungen für Ressourcen, welche mehrere Nutzer gleichzeitig verwalten, ist ein umfassenderer Aggregationsschnitt nur noch bedingt möglich. Deswegen muss vor dem Entwicklungsprozess bei der Aufnahme des Funktionsumfanges auf solche Anforderungen geachtet werden. Zum jetzigen Zeitpunkt ist eine parallele Bearbeitung eines Warenkorbs nicht geplant.

5.1.3 Bewertung des großen Aggregationsschnitts

- **Komplexität:**
  - Die Umsetzung der Businessanforderungen in der Applikation kann übersichtlich erfolgen und eventuelle Konsistenz ist nicht von Nöten.
  - Der Sourcecode muss durch keine komplizierteren Verfahren ergänzt werden.
  - Invarianten können direkt geprüft werden, da stets alle Informationen geladen sind.
- **Performance:**
  - Jede Anfrage benötigt das Auslesen des ganzen Baskets. Da eine dokumentenorientierte Datenbank verwendet wird, beläuft sich dies auf einen einzelnen Suchvorgang.
  - Sofern eine relationale Datenbank eingesetzt wird, leidet die Software an eventuell unnötigen Datenbankoperationen, wodurch eine Verkleinerung des Aggregationsschnittes zu Performance-Verbesserungen führen kann.
- **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 und somit eine Entity benötigt. 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 Zahlvorgang. Daher wird in der *Variante C* das Datenmodell in Basket und PaymentProcess, entlang der Grafik 5.2, aufgeteilt.

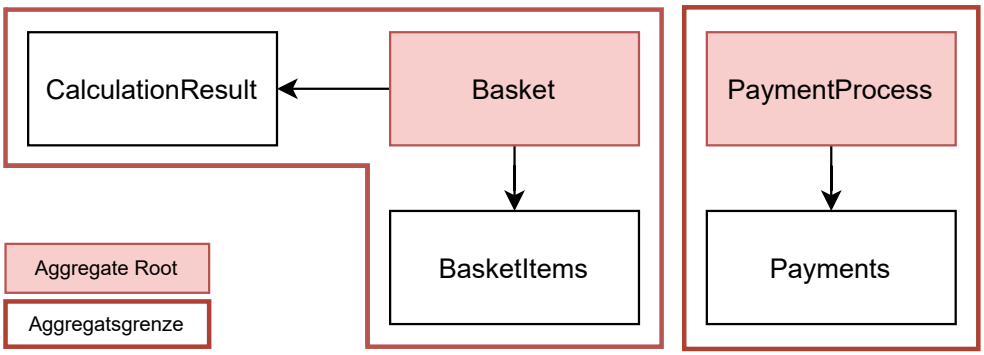

Abbildung 5.2: Aggregationsschnitt der Variante C

Da die Aggregates unabhängig voneinander agieren, müssen sie auch getrennt geladen und abgespeichert werden können. Folglich wird ein Repository angelegt, 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 realisiert 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. Eine kurze Zusammenfassung des Anwendungsfalles lautet wie folgt:

Bevor ein Bezahlvorgang gestartet werden darf, muss eine Evaluierung des Warenkorbzustands stattfinden, wie beispielsweise die Überprüfung der hinzugefügten Zahlungsarten, Kundendaten und errechneten Geldbeträge. Nach erfolgreicher Validierung wird der Warenkorbzustand auf 'freeze' abgeändert und der Zahlungsprozess kann eingeleitet werden.

5.2.1 Eventuelle Konsistenz zwischen Aggregates

Bei der Implementierung von Variante A ist es möglich das Einfrieren des Warenkorbs und die Initiierung des Zahlungsprozesses innerhalb einer einzelnen Transaktion durchzuführen. Aufgrund der Richtlinie, dass jede Transaktion nur maximal ein Aggregate bearbeiten darf, müssen somit diese Funktionalitäten hier aufgespalten werden. Zwischen den beiden Aktionen existiert deswegen eine Zeitspanne, in welcher der Warenkorb zwar eingefroren, allerdings der Bezahlvorgang noch nicht gestartet wurde. Der Warenkorb hat hierbei temporär einen inkonsistenten Zustand, weshalb diese Art von Konsistenz auch als 'eventuelle Konsistenz' referenziert wird. Generell ist in vielen Anwendungsfällen eine kurzzeitige Abweichung der Voraussetzungen akzeptierbar. 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. Das resultierende Problem der eventuellen Konsistenz ist im Codebeispiel 5.1 veranschaulicht.

```
1 function initializePayment(Basket basket, PaymentProcess paymentProcess) {
2     basket.validate()
3     basket.freeze()
4     basketRepository.store(basket)
5     // Mögliches Zwischenschalten anderer Operation, welche zur Abänderung des
6     // Baskets und des Validierungsergebnisses führen kann.
7     paymentProcess.initialize()
8     paymentProcessRepository.store(paymentProcess)
9 }
```

Codebeispiel 5.1: Getrennte Transaktionen für die Initiierung des Bezahlvorgangs

Zwischen dem Persistieren des eingefrorenen Warenkorb und dem Starten des PaymentProcesses kann, aufgrund von parallel ablaufenden Anfragen, der Warenkorb weiterhin bearbeitet werden. Diese Race Condition kann Businessvoraussetzungen, welche zuvor explizit überprüft und erfüllt waren, nun als invalide gestalten. Beispielsweise kann in einem anderen Thread, zeitlich zwischen Zeile 4 und 7, ein externer API-Aufruf die einzige Zahlungsmethode entfernen, wodurch die Initiierung eigentlich Fehlschlagen sollte, jedoch trotzdem ausgeführt wird. Verdeutlicht ist dieser Effekt im Sequenzdiagramm 5.3. Beide Anfragen richten sich an den gleichen Warenkorb. Der rot markierte Bereich stellt die Zeitspanne dar, in welcher andere Prozesse den Warenkorb weiterhin bearbeiten können, obwohl dies nicht vorgesehen ist. Weil die Baskets vor den Modifikationen der anderen Threads geladen werden, ist das Abfangen eines solchen Fehlerzustandes erst beim Zurückschreiben in die Datenbank möglich. Hierbei helfen die vorher erwähnten Sperrverfahren. Das Auftreten eines Fehlers bei der zweiten Transaktion ist besonders problematisch, da die vorgehende Transaktion zurückgerollt werden müsste, diese aber bereits durch einen Commit festgeschrieben ist. Weiterhin entstehen bei Verwendung einer Microservice-Architektur weitere Herausforderungen, denn jede Applikation besitzt ihre eigene Datenbank und ein Lösungsansatz mithilfe von Sperrverfahren ist wegen den verteilten Datensätzen nicht mehr möglich.

Ein valider Zustand des Warenkorbs besitzt in der Checkout-Domain, vor allem aus rechtlichen Gründen, extreme Wichtigkeit. Mithilfe der aktuellen Architektur kann zwar ein solcher gewährleistet werden, erfordert allerdings eine sorgfältige Überprüfung von strengen Invarianten und kann zu einer fehleranfälligen Software führen. Deswegen ist dieses Vorgehen innerhalb einer Checkout-Software nicht empfehlenswert.

5.2.2 Atomare Transaktionen über mehrere Aggregates

Ein weiterer Ansatz zum Bewältigen der Problemstellung 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 von Aggregates ist eine Gruppierung von Klassen, welche vor und nach einer Transaktion stets zusammen 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 eventuelle Konsistenz handhaben können. Die Aggregationsgrenzen sind folglich mit der transaktionalen Konsistenz äquivalent. Dadurch entspricht die Speicherung zweier Aggregates innerhalb einer einzelnen Transaktion eine Überschreitung

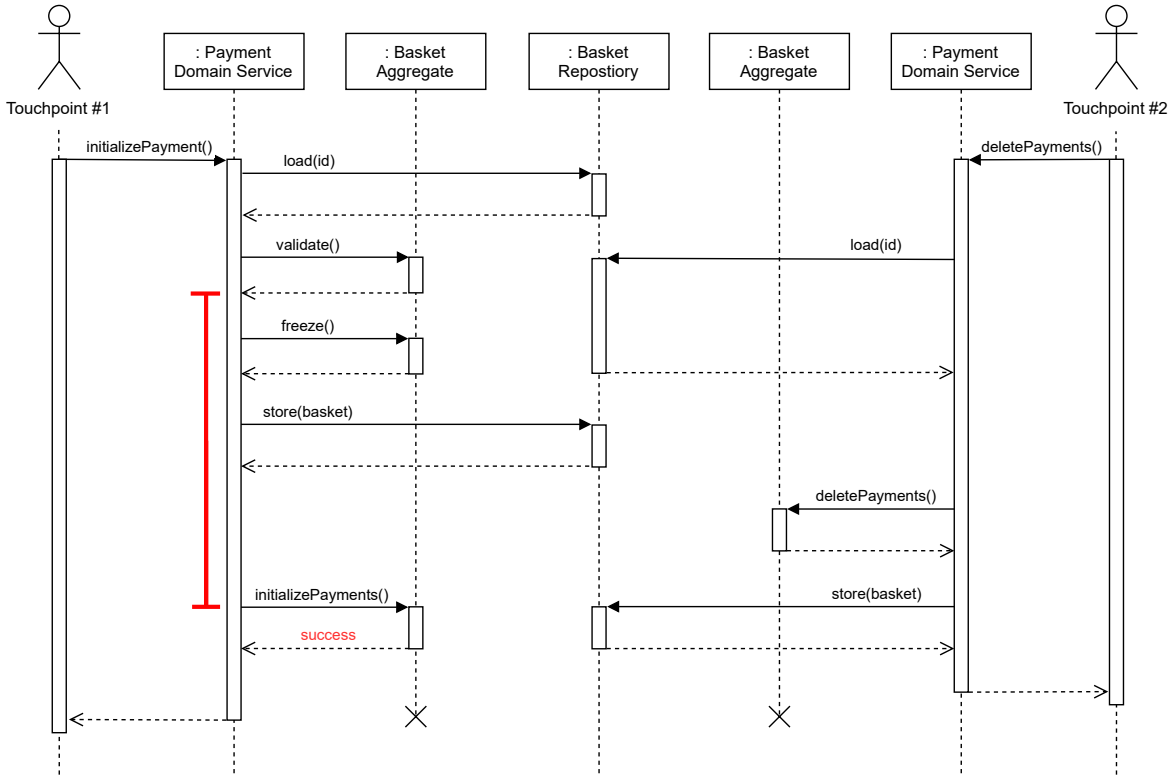

Abbildung 5.3: Vereinfachtes Sequenzdiagramm zur Initiierung des Bezahlvorgangs in Variante C

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, wie bei einer Microservice-Architektur üblich. Eine atomare Transaktion ist dementsprechend unmöglich, wodurch wiederum Race Conditions auftreten können.

Aktuell besteht kein Anlass zur Annahme, dass zukünftig mehrere Datenbankhosts benötigt werden, weshalb die Überlegung entstehen kann, das Einfrieren des Warenkorbs und die Initiierung des Zahlungsvorgangs 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. Demzufolge ist eine Implementierung dieses Lösungsansatzes durch ein Sperrverfahren technisch anwendbar. Mithilfe eines pessimistischen Ansatzes werden beispielsweise andere Zugriffe auf diesen Basket bzw. PaymentProcess erst ausgeführt, wenn die Initiierung vollständig abgeschlossen ist. Das Auftreten von Race Conditions wird dadurch verhindert. Diese Vorgehensweise kann analog in ähnlichen Szenarien verwendet werden. Im Vergleich zum Einsatz von eventueller Konsistenz wird diese Vorgehensweise aus bevorzugt.

5.2.3 Bewertung des Aggregationsschnittes

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, aus rechtlicher Sicht, kritische Operation 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 nahezu alle Aggregationsschnitte, abgesehen von *Variante A*, unzulässig.

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* möglich ist. Anhand dieser Begründung wird für den Proof-of-Concept angenommen, dass ein solcher Aggregationsschnitt als Designentscheidung vertretbar ist.

- **Komplexität:**
  - Unter Einsatz von aggregatsübergreifende Transaktionen bleibt die gewonnene Komplexität überschaubar.
  - Sofern eventuelle Konsistenz eingesetzt wird, entstehen die negativen Eigenschaften einer asynchronen Verarbeitung und die Wechselwirkungen zwischen Anwendungsfällen müssen stets berücksichtigt werden.
- **Performance:**
  - Bei Verwendung einer dokumentenorientierten Datenbank wird die Bearbeitungszeiten von Anfragen anhand der theoretischen Überlegungen nur minimal beeinflusst.
  - Eine relationale Datenbank als Datenspeicher muss weniger Operationen bewältigen, vor allem weil das Payment-Aggregate relativ gesehen selten bearbeitet wird. Daher sollte insgesamt ein positiver Performance-Gewinn entstehen.
- **Parallelität:**
  - Da der Zahlungsvorgang eigentlich nie gleichzeitig mit dem Warenkorb bearbeitet wird, ist dieser Aspekt unberührt.
- **Client-Freundlichkeit:**
  - Die Trennung der Warenkorb- bzw. Zahlungsinformationen betrifft die Clients nicht bemerkbar, da innerhalb eines Anwendungsfalles die Touchpoints lediglich Interesse an eines der beiden Aggregates besitzen.

5.3 Verkleinerung der Aggregates durch Analyse existierender Businessanforderungen

Die ideale Aggregatsgruppierung von Klassen hängt stark von den Invarianten ab, welche sie zusammenbinden. Anhand einer Untersuchung der Anwendungsfällen ist es möglich, das Zusammenspiel von Entities und Value Objects innerhalb eines Aggregates herauszukristallisieren und weitere Ansätze für eine Neuverteilung zu finden.

5.3.1 Herausschneiden der Berechnungsergebnisse aus dem Basket-Aggregate

Zum jetzigen Zeitpunkt existieren in der Produktivanwendung durch den großen Aggregationsschnitt Performance-Einbußen, welche eventuell durch ein verbessertes Design verhindert werden können.

Viele Anwendungsfälle erfordern ein Neukalkulation des Baskets, ansonsten wäre sein Zustand und dementsprechend auch die transaktionalen Grenzen des Aggregates ungültig. In den meisten User Stories fügt der Kunde die gewünschten Artikel zum Basket hinzu und öffnet erst vor Abschluss des Kaufes den Warenkorb. Dadurch ist es möglich, den Zeitpunkt der Gesamtpreiskalkulation bis zu einem explizierten Abruf zu verzögern, um Berechnungszeit einzusparen. Zudem werden weniger Daten zwischen Client und Checkout-Software gesendet, weshalb die Netzwerklast vor allem bei mobilen Touchpoints verringert wird. Aus diesem Grund kann das *CalculationResult* getrennt vom Warenkorb verwaltet werden. 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 CalculationResult wird nur bei der Anzeige des Warenkorbs benötigt, daher ist eine Operation, welche den Gesamtpreis während der Einsichtnahme beeinflusst, bedenklich. Um diese Situationen ausfindig machen zu können, muss der Checkout-Prozess genauer untersucht werden. In der Abbildung 5.4 sind die zwei hierfür relevanten Webseiten des Onlineshops dargestellt. Die roten Pfeile zeigen auf wichtige Stellen des Warenkorbs für dieses Abschnitt.

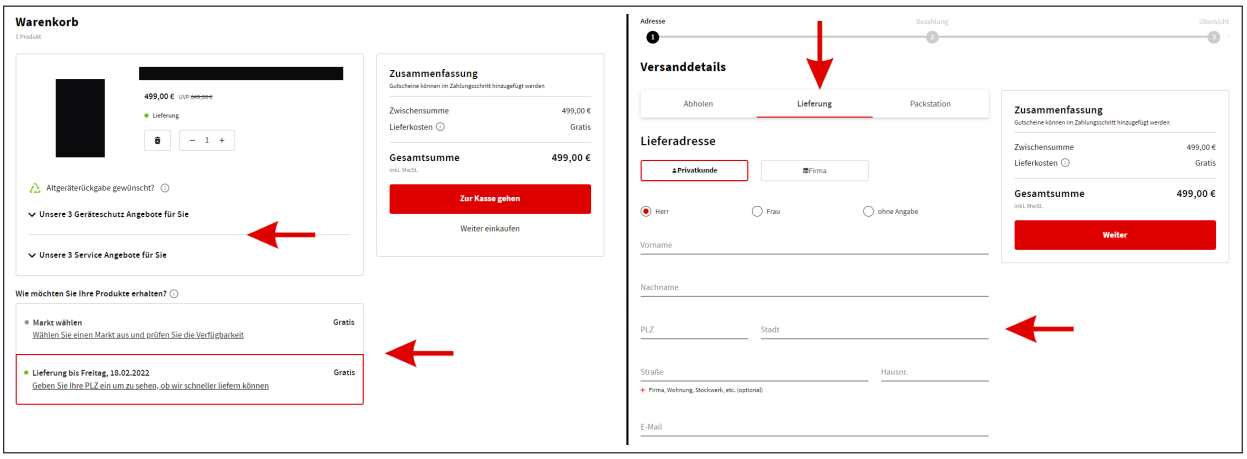

Abbildung 5.4: Aktueller Checkout-Prozess des Onlineshops von MediaMarkt.de

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. Schlussfolgernd 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 aus rechtlichen Gründe notwendig, eine Anpassung des Preises nach Initiierung des Zahlungsvorgangs zu verhindern. Dieses Problem kann allerdings nicht auftreten, da der Warenkorb selbst nicht mehr manipuliert werden kann, dementsprechend 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 dieses Aspektes wird die Auswirkungen einer neuen Regelung untersucht. Exemplarisch kann angenommen werden, dass der Gesamtpreis eines Warenkorbs nicht über 20.000€ liegen darf. In diesem Fall würde eine Manipulation der Elemente im Warenkorb auch eine Neukalkulation benötigen, wodurch die Abtrennung des CalculationResults den Vorteil der verzögerten Berechnung verliert. Allerdings kann eine mildere Form dieser Richtlinie keine negativen Effekt besitzen. Indem zum Beispiel die Prüfung erst beim Start des Bezahlvorgangs ausgeführt wird, da zu diesem Zeitpunkt beide Aggregates immer synchron sind und nachher keine Änderungen mehr erfahren können. Weitere erdenkbaren Businessanforderungen sollten berücksichtigt werden, bevor ein Neudesign der Applikation durchgeführt wird.

Vorläufige Analyse der Bewertungskriterien

- **Komplexität:** Die Implementierung kann ohne umfassendere Codeanpassungen realisiert werden.
- **Performance:** Zwar ist die Anzahl der Kalkulationen minimiert, jedoch steigen die Datenbankoperationen an, weil zuvor beide Objekte innerhalb einer Tabelle persistiert werden konnten, wodurch die Performance zweiseitig beeinflusst ist.
- **Parallelität:** Der Gesamtpreis wird nur als Wechselwirkung von Aktionen angepasst, sodass dieser Gesichtspunkt nicht bewertbar ist.
- **Client-Freundlichkeit:** Anhand der analysierten Fragestellungen ist die Anwenderfreundlichkeit davon abhängig, ob die Berechnungsergebnisse als Antwort von API-Aufrufen erwartet werden. Die Verwendung von Technologien wie GraphQL ermögliche die Rückgabe beider Aggregates und neutralisieren dieses Argument, jedoch beeinflussen wiederum wegen zusätzlichen Datenbankoperationen und Berechnungen die Performance.

Grundsätzlich ist eine Abspaltung der Berechnungsergebnisse vom Warenkorb durchaus plausibel. 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 Aktivitätsdiagramm 3.4 wurde das Hinzufügen der Kundendaten, Zahlungsmethode und des Fulfillments beschrieben. Innerhalb eines API-Aufrufs soll eine Anpassung dieses Datenumfangs möglich sein. Das resultierende Webshop-Design in Bild 5.4 ist ein Hinweise darauf, dass diese Attribute eventuell aus dem Basket-Aggregat genommen werden können. Sie sind in der Ubiquitous Language als *'Checkout-Daten'* betitelt und beinhalten Kundendaten, Fulfillment, Rechnungs- und Lieferadresse. Ähnlich zum vorgehenden Unterkapitel ist eine Untersuchung der Implikationen eines solches Aggregationsschnittes notwendig.

Nachteilig ist hier, dass dadurch die Value Objects zu einer Entity zusammengefasst werden müssen, da die Rolle des Aggregate Root nur durch Entities erfüllt werden darf. Deshalb wird die ursprüngliche eins-zu-eins Relation wiederum in der Datenbank als zwei separate Tabelle designt und zieht somit Performance-Einflüsse bei Abfragen beider Datensätze mit sich. Jedoch werden die Checkout-Daten in nahezu allen User Stories nur einmalig bearbeiten, sodass dieser Effekt minimal bleibt.

Anpassungen an den Daten dürfen nur durchgeführt werden, wenn der Basket im Status 'open' ist. Aus diesem Grund muss bei jedem API-Aufruf zuvor der Zustand überprüft und der Warenkorb-Datenbankeintrag bis zum Abschluss der Bearbeitung gesperrt werden. Aufgrund von möglichen Race Conditions kann ansonsten beispielsweise die Initiierung des Bezahlvorgang auf invalide Checkout-Data stattfinden. Weiterhin können sich bei der Auswahl einer anderen Fulfillment-Methode die Lieferkosten ändern und folglich müssen die aktualisierten Preise des Warenkorbs persistiert werden. Hierbei stößt die Applikation auf eine Transaktion über zwei Aggregates und zugleich auf die vorher untersuchte Problemstellung.

Mithilfe dieses Designs ist der Basket leichtgewichtiger, geringere Datenmengen müssen bei Abruf transportiert werden und die Aufteilung der Aggregates spiegelt genauer die betroffenen Anwendungsfälle wider. Die genauere Bewertung der Performance und Komplexität findet in Verbund mit den vergangen Aggregaten im nächsten Kapitel statt.

//Kommentar: Bewertung hier? Unterkapitel ist bisschen kurz und der Aggregationsschnitt klingt sehr negativ belastet. Allerdings ist es schwer mehr zu schreiben als 'leichtgewichtiger' ohne sich im Kreis zu drehen...

5.4 Zusammenführung der vorgehenden Domain-Modelle

Um einen möglichst unterteilten Aggregationsschnitt zu gewährleisten, wird auf Basis der vorgehenden Analysen ein kombiniertes Design erstellt, welches das *CalculationResult*, die *CheckoutData* und der *PaymentProcess* aus dem Basket in ihre eigenen Aggregates heraus hebt. Abbildung 5.5 zeigt das resultierende Datenmodell '*Variante D*'. 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 wird, kann weiterhin die Trennung der beiden Klassen voneinander in Betracht gezogen werden.

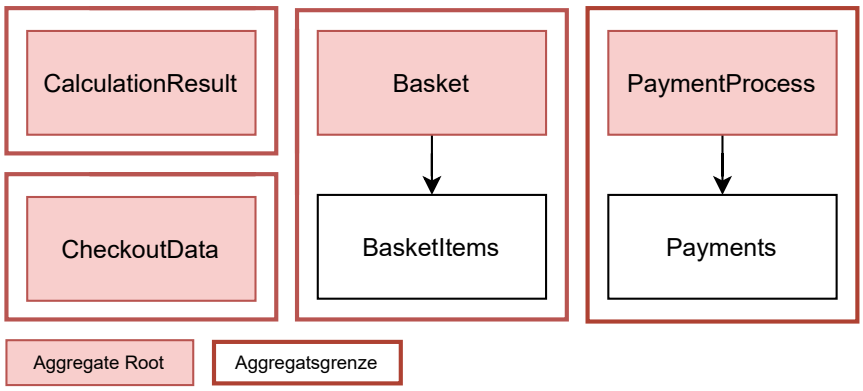

Abbildung 5.5: Aggregationsschnitt der Variante D

5.4.1 Aktualisieren von veralteten Datenständen

Ähnlich zu anderen Designvariationen benötigen viele Anwendungsfälle das gleichzeitige Laden und Bearbeiten mehrerer Aggregates. Beispielsweise beim Hinzufügen eines neuen BasketItems ist die Neuberechnung des PaymentProcesses und CalculationResults notwendig. Gleichermaßen muss dieser Prozess angestoßen werden, wenn das Fulfillment in den Checkout-Daten bearbeitet und die Versandkosten des Warenkorbs sich dadurch ändern. Zur Abwicklung einer solchen Abhängigkeit sind mehrere Implementierungsmöglichkeiten denkbar.

Sollte die Kalkulationen erst bei Bedarf stattfinden, werden Zusatzinformation benötigt, um zu indizieren, dass der aktuelle Berechnungswert veraltet ist. Zu diesem Zweck wird ein Wahrheitswert (auch 'Flag' genannt) in den Basket und Checkout-Daten 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 Wert 'wahr' ist, welche zugleich auch den PaymentProcess aktualisiert. Dieses Vorgehen spart Berechnungszeit auf Kosten von zusätzlichen Datenbankoperationen ein. Die Komplexität und Fehleranfälligkeit der Software steigt hierbei an.

Andernfalls kann das CalculationResult bei jedem Abruf aus der Datenbank aktualisiert werden. Die Vor- und Nachteile sind im Vergleich zur vorherigen Lösung invertiert. Es sind mehr Kalkulationen notwendig, jedoch wird die Datenquelle entlastet. Sofern die Preisberechnung sich umfangreich gestaltet, sind Performance-Einbußen zu erwarten.

Alternativ kann unverändert zu *Variante A* die Kalkulation sofort bei Datenänderungen geschehen. Allerdings werden die Aggregationsgrenzen überschritten und innerhalb einer Transaktion mehrere Aggregates bearbeitet. Dies stellt einen Zwischenweg der beiden vorgehenden Implementierungen dar.

In dem Proof-of-Concept sind erstere und letztere Vorgehensweise implementiert, um eine exemplarische Realisierung aufzuzeigen.

5.4.2 Dependency Injection von Services in Domain-Driven Design

Umfangreiche oder nicht klar zuordenbare Funktionalitäten werde in Services ausgelagert. Die Reihenfolge der Serviceaufrufe ist durch den Anwendungsfall bereits vorgegeben, jedoch nicht in welcher Klasse diese stattfindet und wie die notwendigen Komponente an die Objekte übergeben wird. Verschiedene Implementierungen besitzen auch unterschiedliche Vor- bzw. Nachteile. Dieses Unterkapitel geht unter Beachtung von Dependency Injection auf einige Realisierungsmöglichkeiten genauer ein.

Dependency Injection ist eine spezielle Art des Dependency-Inversion-Prinzips. Hierbei initialisiert ein Framework erforderlichen Klassen durch ihren Konstruktor, indem die obligatorisch Parameter ebenfalls injiziert werden. Dadurch entsteht ein rekursives Muster bis schließlich alle Objekte erzeugt worden sind. Dadurch wird eine lose Kopplung der Module erreicht und eine Möglichkeit geschaffen, die konkreten Implementierungen anhand von Konfigurationsdateien auszutauschen. Eine Problematik dieser Vorgehensweise, welche auf inkorrektes Klassendesign hinweisen kann, ist die Bildung einer 'Circular Dependency' (dt. Zirkelbezug). Entsprechend der Grafik 5.6 tritt dies auf, sofern zwei Komponente voneinander abhängig sind. Das Framework versucht eine der beiden Klassen zu initialisieren, wobei im Konstruktor die verbleibende Klasse benötigt wird. Da diese jedoch ein Objekt der noch nicht erzeugen ersteren Klasse erwartet, entsteht ein ewiger Kreislauf von Konstruktoraufrufen.

Eine Applikation mit einer Circular Dependency ist folglich nicht mehr ausführbar und die Eliminierung einer der Abhängigkeiten ist erforderlich.

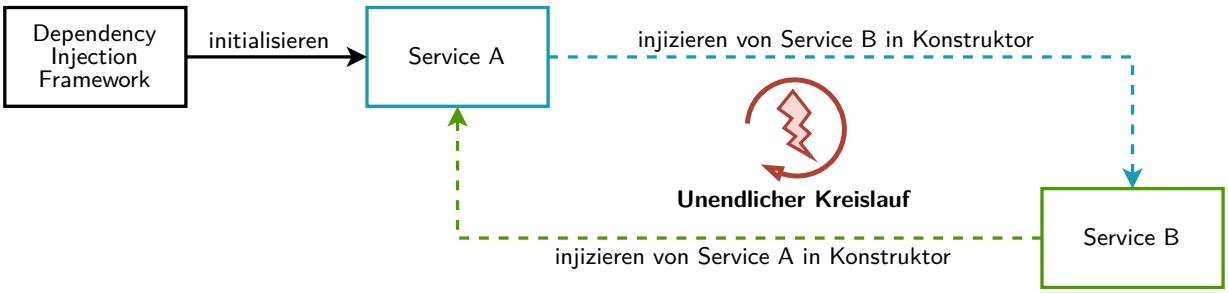

Abbildung 5.6: Darstellung einer Circular Dependency

Je mehr Abhängigkeiten eine Klasse besitzt, desto höher ist die Wahrscheinlichkeit des Auftretens einer Circular Dependency. Bei der konkreten Implementierung des Proof-of-Concepts stellte die Vermeidung einer solchen Situation teilweise eine Herausforderung dar.

Steuerung des Programmablauf innerhalb von Domainservices

Sofern mehrere Komponente für die Durchführung eines Anwendungsfalles notwendig sind, fällt das Wissen über deren Aufrufreihenfolge in den Kontext der Domain. Deshalb sind Klassen außerhalb des Datenmodells mit mehreren Serviceaufrufen den Domainservices zuzuordnen. Im ersten Lösungsansatz wird der Programmablauf außerhalb des Aggregates durch einen Domainservice bestimmt. Das Codebeispiel 5.2 implementiert beispielsweise einen Service zur Validierung einer Invariante. Dass zuvor eine Überprüfung der Validität einer Aktion stattfinden muss, stellt hier das Domainwissen dar.

```
1 class DomainService {
2
3     function doStuff(Aggregate aggregate) {
4         if (someService.isActionValid()) {
5             aggregate.doStuff()
6         }
7     }
8
9 }
```

Codebeispiel 5.2: Bestimmung des Steuerflusses durch einen Domainservice

Die Eintrittswahrscheinlichkeit einer Circular Dependency ist gesenkt, da normalerweise die Domainservices sich nicht gegenseitig benötigen. Der größte Nachteil dieses Ansatzes liegt jedoch in der eingeführten Fehleranfälligkeit des Programms. Bei unvorsichtigen Codeanpassungen kann der Aufruf von 'aggregate.doStuff()' von anderen Abschnitten der Software ohne die vorgehende Validierung geschehen. Die Konsistenz des Aggregatzustandes ist somit gefährdet und damit verbundenen Qualitätsmerkmale werden geschwächt. Zur Verhinderung einer solchen Situation ist eine Verlagerung des Funktionsaufrufs in das Aggregate selber denkbar.

Übergabe der Referenz an das Aggregate als Parameter

Ein Aggregate sollte stets direkt alle wesentlichen Invarianten überprüfen, sodass die Ausführung von invaliden Aktionen unterbunden ist. Um dies zu gewährleisten, muss der Service innerhalb des Aggregate aufgerufen werden, weswegen dieser eine Referenz auf das gefragte Objekt benötigt. Dementsprechend wird in Figur 5.3 der Service als Parameter an das Aggregate übergeben.

```
class DomainService {

    variable SomeService someService

    function doStuff(Aggregate aggregate) {
        aggregate.doStuff(someService)
    }
}
class Aggregate {
    // Kann nicht umgegangen werden, da Validierung direkt in der Funktion geschieht
    function doStuff(SomeService someService) {
        if (someService.isActionValid()) {
            ...
        }
    }
}
```

Codebeispiel 5.3: Übergabe der Referenz an das Aggregate als Parameter

Dies löst das vorgehende Problem der Fehleranfälligkeit, jedoch wird die Funktionssignatur und das Aggregate aufgebläht. Grundlegend ist eine Abwägung notwendig, wann es sinnvoll ist, den Funktionsaufruf in das Aggregate mitaufzunehmen. Bei wichtigen Validierungen sollte diese Variante bevorzugt werden. Die Bewertung der Dependency Injection bleibt hier unverändert im Vergleich zur vorherigen Implementierung. Der Proof-of-Concept verwendet diesen Ansatz, um Abhängigkeiten zu realisieren.

Injektion der Service in ein Aggregate durch das Repository

Weiterhin können auch zur Minimierung der Funktionsparameter die Service in das Aggregate hinein injiziert werden. Folglich halten die Aggregates selbst eine Referenz auf die jeweilig Klassen und rufen diese bei Notwendigkeit auf. Die Injektion muss bei Objekterzeugung geschehen, weshalb die Verantwortung bei den Repositories liegt. Im kurzen Beispielcode 5.4 wird dieser Gedanke verdeutlicht.

```
class Aggregate {

    variable SomeService domainService

    // Setzen des konkreten Service
    function inject(SomeService domainService) {
        this.domainService = domainService
    }
}

class AggregateRepository {

    function load(Id id) returns Aggregate {
        variable aggregate = searchInDatabase(id)
        aggregate.inject(someService)
        return aggregate
    }

}
```

Codebeispiel 5.4: Injektion des Services in ein Aggregate durch das Repository

Diese Methodik hat sich bei der Implementierung allerdings als problematisch erwiesen. Einerseits ist es fragwürdig, ob Datenklassen aus Entwicklersicht überhaupt Referenzen auf Services halten sollten, da weiter Abhängigkeiten erzeugt werden. Davon abgesehen erhöht sich stark die Wahrscheinlichkeit auf eine Circular Dependency innerhalb der Repositories, weil sie alle Services besitzen müssen, um diese den Aggregate zu übergeben. In *Variante D* der Checkout-Software stehen die Aggregates in

enger Bindung zueinander und benötigen zum Laden dieser somit ihre gegenseitigen Repositories. Deshalb kann es bei Wechselwirkungen zwischen den Aggregates vorkommen , dass zwei Repositories sich gegenseitig benötigen.

5.4.3 Performance-Analyse der Aggregationsschnitte unter Einsatz von Lasttests

Damit eine genaueren Aussage über die Performance von Variante A und D möglich ist, wird ein Lasttest mithilfe der Open-Source-Software 'JMeter' realisiert. Zuerst wurde hierfür ein gängiger Anwendungsfall definiert, welcher folgende Aktionen umfasst: Erstellen eines Baskets, dreimaliges Hinzufügen von Artikeln, Setzen der Checkout-Daten, zweimaliges Abrufen des Warenkorbs, Hinzufügen eines Payments und das Initiieren inklusive Durchführen des Bezahlvorgangs. Nach dem Vorbefüllen der Datenbank mit 1000 Datensätze wird anschließend dieser Prozessablauf in zehn parallelen Threads 'X'-mal durch die verschiedenen Aggregationsschnitte abgearbeitet. Zum Ausschließen von abweichenden Ergebnissen wird dies drei mal wiederholt. Die Auswertungen dieses Kapitels ist im Anhang zu finden.

Bemerkungen zum Performance-Test

Die Performance-Analyse dient lediglich als genereller Vergleich und weicht je nach Anwendungsfall, Hardware, Netzwerk und Softwareimplementierung ab. Zudem können weiterhin Optimierungsoptionen vorgenommen werden, wie die angepasste Einstellung des Connection-Pools, das Einsparen von ganzen Datenbankoperationen durch Caching oder schnellere Kommunikation der Systeme untereinander. Anhand der Analyse sollen vorgehende Aussagen bewiesen und Argumente für das Fazit gebildet werden.

Zur simpleren Referenz der getesteten Varianten wird die Verwendung von MongoDB mit 'M' und von PostgreSQL mit 'P' abgekürzt. Weiterhin ist der Aggregationsschnitt D in die Untertypen 'F' bzw. 'C' unterteilt. Hierbei finden in der Abwandlung 'C' Nebeneffekte, wie die Neuberechnung des Gesamtpreises sofort statt, wohingegen in 'F' erst bei Abruf der jeweiligen Daten anhand von Flags erkennt wird, wann eine Neukalkulation notwendig ist. Folglich steht beispielsweise 'Variante D-PF' für den Aggregationsschnitt D mit einer PostgreSQL-Datenbank und eine Implementierung von Flags.

Erster Testdurchlauf und hieraus ableitbare Aussagen

Zu Beginn wurde der Proof-of-Concept für Variante A und D mitsamt ihren Abwandlungen implementiert und sachbezogene KPIs (Key-Performance-Indicator) anhand des obigen Testaufbaus ausgewertet. Das Ergebnis kann dem Anhang unter 8.1 entnommen werden. Um die Konsequenz bei Verwendung einer relational Datenbank zu beobachten, wurde ebenfalls der Versuch mit einer PostgreSQL-Datenbank wiederholt. Die Auswertungen sind in Tabelle 8.2 hinterlegt. Zum Präsentieren einer schnellen Übersicht wurde die durchschnittlichen 'Abläufe pro Sekunde' über einen Gesamtumfang von 10.000 Abläufen in der Grafik 5.7 visualisiert.

Mithilfe von Flags werden Neukalkulationen im Austausch für zusätzliche Datenbankoperationen eingespart, jedoch sind die eigentlichen Berechnungen des Warenkorbs in dem verkleinerten Proof-of-Concept trivial, weshalb zu beobachten ist, dass ein solches Vorgehen insgesamt zu einer geringeren Performance führt. In der aktuellen Live-Applikation kann aufgrund der komplexeren Berechnungsgrundlage dieser Ansatz allerdings weitaus performanter ausfallen. Negativ ist anzumerken, dass die Nutzung von Flags mit einem komplexeren Programmablauf einhergeht, da die Aktualisierung von Daten asynchron stattfinden. Bei der Implementierung von neuen Funktionen müssen stets die Wechselwirkungen zwischen den Wahrheitswerten und Businessprozessen beachtet werden, wodurch die Fehleranfälligkeit

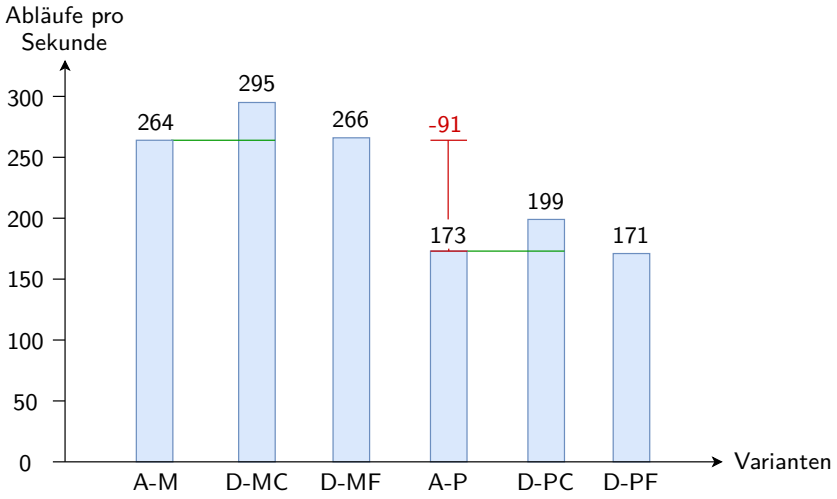

Abbildung 5.7: Performance-Ergebnisse aus dem ersten Testdurchlauf

Variante	A-M	D-MC	D-MF	A-P	D-PC	D-MF
Median-Dauer	35ms	31ms	35ms	56ms	49ms	56ms
Leseoperationen	15	43	55	84	137	170
Schreiboperationen	15	27	28	108	98	127

Tabelle 5.8: Median der Ausführungszeit und Datenbankoperationen eines Durchlaufes

der Software steigt. Folglich sollten Flags nur in Betracht gezogen werden, sofern relevante Performance-Verbesserungen vernehmbar sind. Schlussendlich kann eine finale Aussage zwischen Variante D-C und D-F für die Produktivanwendung aufgrund der fehlenden Komplexität nicht getroffen werden.

In einem früheren Kapitel wurde die These aufgestellt, dass die Unterteilung von großen Aggregaten in kleinere unter Verwendung einer dokument-orientierten Datenbank in geringeren oder sogar negativen Performance-Gewinnen resultieren kann, als beim Einsatz einer relationale Datenbank. Diese Aussage ruht auf der Basis, dass mehr Datenbankoperationen bei einer MongoDB benötigt werden, sobald die Anzahl der Aggregates steigen, da zuvor alle Daten in einer einzelnen Collection abgespeichert werden konnten. Hingegen ist in relationalen Datenbanksystemen, aufgrund von Mehrfachbeziehungen und der Einhaltung von Normalformen, ein Aggregat bereits in mehrere Tabellen unterteilt und die weitere Abspaltung von Objekten führt zu insgesamt weniger neuen Relationen. In Tabelle 5.8 wurde der Median über die Ausführungsdauer eines Anwendungsfalles pro Aggregationsschnitt gebildet und die Gesamtzahl der hierfür benötigten Datenbankoperationen aufgelistet.

Der Performance-Unterschied beläuft sich auf 4 Millisekunden zwischen A-M und D-MC, sofern eine MongoDB als Datenbankmanagementsystem genutzt wird, wohingegen ein Einsparung von 6 Millisekunden bei einer PostgreSQL möglich ist. Dieser Effekt kann weiterhin deutlicher in kommende Testaufbauten beobachtet werden, worauf die These gestützt wird. Aus den Daten ergibt sich allerdings die Frage, warum der Umbau zu Variante D überhaupt einen Performance-Gewinn mit sich bringt, obwohl dies mit mehr Datenbankoperationen einhergeht.

Zum Speichern oder Laden von Datensätzen muss eine sogenannte De- bzw. Serialisierung stattfinden, damit die Daten richtig interpretiert werden können. Bei genauer Untersuchung der Datenbankzugriffe stellt sich heraus, dass dieser Vorgang aktuell den Großteil der Bearbeitungszeit in Anspruch nimmt,

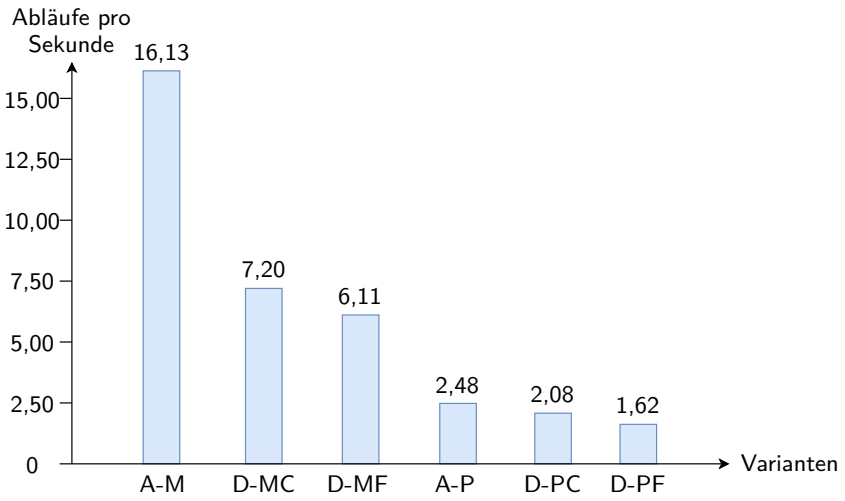

Abbildung 5.9: Performance-Ergebnisse bei Auslagerung der Datenbanksysteme

nicht wie zu vermuten die eigentliche Datenbankoperation. Deshalb sind die Variationen A insgesamt langsamer, da mehr Daten verarbeitet werden und somit zugleich die De-/Serialisierungsdauer erhöht wird. Diese Situation entsteht, da die Datenbank auf der gleichen Maschine wie die eigentliche Software ausgeführt wird. Die Kommunikation zwischen den Systemen ist nicht vom Netzwerk abhängig und ein Datenbankzugriff beläuft sich auf circa 1,5 Millisekunden. In den meisten Produktivumgebungen ist die Datenbank allerdings physisch getrennt, woraus Verzögerungen durch den Datentransport über das Netzwerk entstehen. Zur Simulation dieses Phänomens wird ein weiterer Performance-Test durchgeführt.

Aushängen der Datenbank in eine Cloud-Umgebung

Damit eine ähnlichere Situation zur Produktivanwendungen geboten werden kann, wurde bei einem Cloud-Anbieter jeweils eine MongoDB und PostgreSQL in einem frankfurter Datenzentrum mit 8 Gigabyte Arbeitsspeicher und 4 virtuellen Kernen angemietet. Der vorherige Testaufbau wird unter Verwendung dieser Datenspeichern wiederholt und tabellarisch dem Anhand unter 8.3 hinzugefügt. Das Diagramm 5.9 bietet die dazugehörige grafische Auswertung.

Die zusätzlichen Datenbankoperationen wirken sich nun stark auf die Performance des Proof-of-Concepts aus. Der mögliche Durchsatz von Variante A ist höher als der verkleinerte Aggregationsschnitt, da die De-/Serialisierung nur noch ein Bruchstück der Gesamtdauer ausmacht und die eigentliche Datenbankverbindung hingegen mehr Zeit in Anspruch nimmt. Die Differenz zwischen Variante A-M und D-MC beträgt hierbei durch das Netzwerk durchschnittlich 751 Millisekunden. Es muss beachtet werden, dass die Kommunikation mit den Datenspeicher aktuell die einzige Wartezeit der Applikation darstellt, wodurch der Einfluss auf die Performance unnatürlich hoch ausfällt. Zudem kann die Einbußen verringert werden, indem beispielsweise die Systeme physisch näher zusammenliegen oder eine dedizierte Verbindung mit der Datenbank über das Netzwerk hergestellt wird.

Simulation von Wartezeiten bei Aufrufen von externen APIs

Die Tatsache, dass Aufrufe an externe Systeme in zusätzlicher Wartezeit resultiert, wurde bis jetzt in den Performance-Test vernachlässigt. Aus der Analyse von Monitoring-Systemen der aktuellen Produktivanwendung konnten durchschnittliche API-Abrufzeiten gebildet und in die Applikation

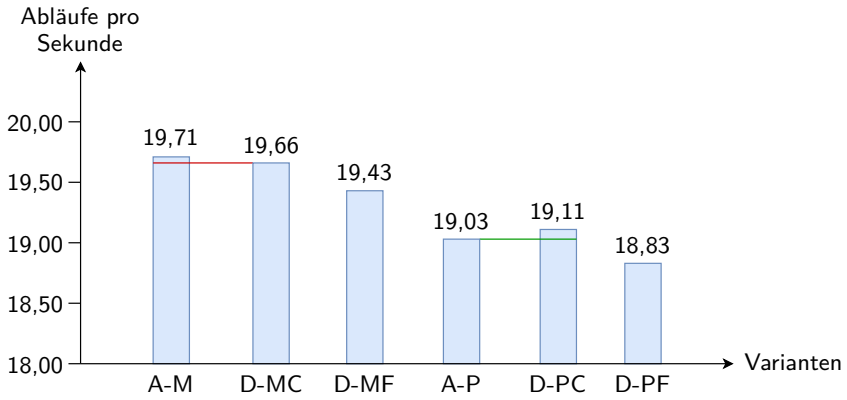

Abbildung 5.10: Performance-Ergebnisse unter Beachtung der Wartezeiten bei API-Aufrufe

eingebaut werden. Im Mittel verbringt die Software 465 Millisekunden mit dem Warten auf Antworten von abhängigen Systemen. Der Testablauf wird auf Basis des ersten Tests inklusive der Wartezeiten-Simulation dementsprechend erneut durchgeführt . Die gemessenen Werte befinden sich in Tabelle 8.4 und im Balkendiagramm 5.10.

Der mögliche Durchsatz sinkt bemerkbar und Unterschiede zwischen den Aggregationsdesigns verwaschen. Alle Aggregationsschnitte erfordern im Proof-of-Concept die gleiche Anzahl an API-Aufrufe. Sollte dies nicht der Fall sein, muss ein solcher Fakt ebenfalls mit in die Bewertung der Aggregates mit einfließen, weil sich unterschiedliche Wartezeiten und folglich Performance-Unterschiede ergeben.

Fazit aus dem Performance-Test

Jegliche Art von Kommunikation mit externen Systemen gehen einher mit erheblichen Performance-Verluste der Software, da die eigentliche Datenverarbeitung in dem Proof-of-Concept minimal gehalten wurde. Der eventuelle Verlust von Performance muss anhand des konkreten Anwendungsfalles bewertet werden. In Systemen, welche zusätzliche 100 Millisekunden keine relevanten Auswirkungen auf die Businessprozesse haben, kann der Einfluss des Datenmodells auf die Performance in den meisten Fällen vernachlässigt werden. In produktiven Umgebungen fallen die Performance-Verluste zudem weitaus geringer aus. Letztendlich besteht jedoch die Tendenz, dass ein kleinerer Aggregationsschnitt im Kontext einer Checkout-Software in Kombination mit einer dokumentenorientierten Datenbank negative Auswirkungen besitzt.

5.4.4 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.

- **Komplexität:** Viele Prozesse benötigen zum Abfragen von externen Informationen oder für die Einhaltung von Invarianten mehr als ein Aggregat, wodurch zusätzlichen Funktionsaufrufe und Übergabeparameter eingeführt werden müssen. Dabei leidet die Übersichtlichkeit des Quellcodes. Dennoch ist es beispielsweise möglich durch Verwendung eines Kontextobjekts, auf welches überall in der Applikation Zugriff besteht, diese Auswirkungen einzudämmen, indem geladene Aggregate dort eingespeichert und abgerufen werden.

Das Transaktionsmanagement gewinnt ebenfalls an Relevanz, sodass Datensätze so kurz wie möglich gesperrt sind. Sofern keine zeitgleichen Aufrufe auf dieselben Aggregates stattfinden, wirkt sich dies jedoch nicht bemerkbar auf die Performance aus.

Weiterhin muss für die Neuberechnung der Preise zum Abarbeiten von Nebeneffekten Flags eingeführt werden, sofern eine Vermeidung von extra Schreibprozesse angestrebt wird.

Insgesamt gewinnt die Applikation generell an Komplexität, welche je nach Realisierungsmethode stärker oder schwächer ausfallen kann.

- **Performance:** *Basierend auf die vorgehende Analyse ist ein Verlust von Performance vorhanden, da in der Live-Umgebung eine MongoDB verwendet wird.*
- **Parallelität:** Generell können zwei unterschiedliche Aggregates unabhängig voneinander bearbeitet werden. Als konkretes Beispiel ist die Bearbeitung von enthaltenen Artikeln und das setzen von Checkout-Informationen gleichzeitig möglich. Eine tatsächliche Anwendung dieses Falles ist allerdings kaum denkbar. Sofern zusätzlich die Abspaltung der BasketItems hinzukommt, gewinnt dieser Aspekt an Bedeutung. *Die gewonnene Parallelität ist folglich nahezu vernachlässigbar bzw. nicht anwendbar.*
- **Client-Freundlichkeit:** Touchpoints erhalten als Antwort auf API-Anfragen in dieser Variante nur einen Teilaspekt des Baskets, denn andererseits müssten wieder alle Aggregate geladen werden, wodurch der Sinn einer Trennung verloren geht. Im Vorhinein findet mit den Clients eine Klärung über die erwarteten Antwortdaten mithilfe einer API-Vereinbarung statt, sodass diese ohne zusätzliche Anfragen weiterhin ihre Arbeitsprozesse abwickeln können. *Letztendlich sind Clients bestenfalls nur neutral von der neuen Architektur betroffen.*

Zusammenfassend existiert nur sehr beschränkt ein wirklicher Grund für die Anwendung von Variante D oder ähnlichen Aggregationsschnitten.

6 Implementierung des Proof-of-Concepts

Im Verlaufe des Projekts wurde das Datenmodell mitsamt der Hexagonalen Architektur und allen relevanten Anwendungsfällen in einem Proof-of-Concept implementiert. Das Ziel dieser Software ist die Unterstreichung einer möglichen praktischen Umsetzung der Businessanforderungen 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 Komponente sind analog zu einer Hexagonalen Architektur in die drei Bereiche primäre Adapter, Applikationskern und sekundäre Adapter aufgeteilt. Innerhalb des Applikationskerns befindet sich jegliche Businesslogik, sowie die notwendigen Applicationservices, Domainservices und das Domainmodell entsprechend des Domain-Driven Designs. Die umgesetzten Aggregationsschnitte belaufen sich auf die Variante A und D, sowie ihren Abwandlungen. Dieses Kapitel geht auf die Entwicklung des ersteren Ansatz mit einem einzelnen, großen Basket-Aggregate tiefer ein.

//Kommentar: Bild der Architektur einfügen!

6.1 Design der primären Adapter

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

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 das Domainmodell umgewandelt werden. Ist dies nicht der Fall, hält der zentrale Teil der Software eine Abhängigkeit nach außen und das Dependency-Inversion-Prinzip wird verletzt. Der Einsatz von sogenannten Data-Transfer-Objects (DTO) ist gestattet, sofern diese in einem Modul des Applikationskern liegen, sodass dieser volle Kontrolle über ihre Struktur besitzt. Ein konkretes Beispiel für einen Controller ist im Codebeispiel 6.1 abgebildet. Zur Implementierung dieser Funktionalität wurde das Framework 'Ktor' verwendet.

```
1 put("/basket/{id}/customer") {
2     variable basketId = parseIdFromUrl("{id}")
3     variable customer = request.body<Customer>()
4     variable basket = basketApiPort.setCustomer(basketId, customer)
5     request.respond(HttpStatusCode.OK, basket)
6 }
```

Codebeispiel 6.1: Beispiel eines Controllers zum aktualiseren von Kundendaten

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

Für jeden definierten Anwendungsfall ist ein entsprechender Port und Controller zuständig. Die tatsächliche Implementierung der Schnittstelle wird mittels Dependency Injection 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 Test-Objekt ausgetauscht und Aufrufe des Objektes ausschließlich simuliert werden. Somit erfahren die einzelnen Komponenten in Testfällen keine Beeinflussung durch eventuell inkorrekt implementierten Code und Test-Fehlschläge können eindeutig einem bestimmten Abschnitt der Software 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 hierbei 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 function removeBasketItem(BasketId basketId, BasketItemId basketItemId) {
2     variable basket = basketRepository.findById(basketId)
3     basket.removeBasketItem(basketItemId)
4     basketStorageService.store(basket)
5 }
```

Codebeispiel 6.2: Eine Beispielsfunktion des BasketItem-Applikationservice

- Zeile 2: Laden eines Baskets anhand seiner Id durch einen Domainservice.
- Zeile 3: Aufruf des Basket-Aggregate-Root zum Entfernen des übergebenen Items. Innerhalb dieser Funktion werden zusätzlich Aufgaben erledigt, wie das Neuberechnen des Gesamtpreises. Sollte die Kalkulation zu komplex ausfallen, kann hier ein Berechnungsservice als Parameter übergeben werden. Somit ist der Basket für seine eigene Konsistenz verantwortlich.
- Zeile 4: Speichern des Warenkorbs mit den abgeänderten Daten.

6.2.2 Aufteilen der Businesslogik zwischen Domainservices und Datenmodell

Aufgrund der Tatsache, dass die Software lediglich ein Aggregat und somit auch nur ein Aggregate-Root besitzt, müssen alle Operationen auf den Attributen des Baskets durch eine Methode im Basket selbst geschehen. Anwendungsfälle, die es erfordern tiefer gelegene Objekte anzupassen, werden durch eine Kette von Funktionsaufrufen umgesetzt. Hierbei können alle notwendigen Invarianten überprüft werden, da das komplette Datenmodell geladen ist. Zur Trennung der Datenhaltung von den Funktionalitäten implementiert der Warenkorb ein Interface, welches jegliche von außen notwendigen Schnittstellen beinhaltet. Dadurch kann das darunterliegende Datenmodell ausgetauscht oder in Tests simuliert werden. Als Veranschaulichung für das Abändern der Fulfillment Methode dient der Codeauszug 6.3.

```
1 function setFulfillment(Fulfillment fulfillment, FulfillmentPort fulfillmentPort) {
2     validateIfModificationIsAllowed()
3     variable availableFulfillment = fulfillmentPort.getAvailableFulfillment(outletId)
4     throwIf(availableFulfillment.doesNotContain(fulfillment)) {
5         IllegalModificationError("$fulfillment is not avaiable")
6     }
7     this.fulfillment = fulfillment
8 }
```

Codebeispiel 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.

Aufgaben, welche nicht direkt einem Objekt zugewiesen werden können oder mehrere Aggregates betreffen sind in Domainservices zu implementieren. Diese 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, wie in Code-Ausschnitt 6.4 aufgeführt.

```
1  function executePaymentProcessAndFinalizeBasket(BasketId basketId) {
2      variable basket = basketRepository.findById(basketId)
3      throwIf(basket.isNotFrozen() or basket.paymentIsNotInitialized()) {
4          IllegalModificationError("cannot cancel payment process")
5      }
6      variable externalPaymentRef = basket.getExternalPaymentRef()
7      paymentPort.executePayment(externalPaymentRef)
8      basket.executePayments() and basket.finalize()
9      basketStorageService.store(basket)
10     createOrderAfterFinalization(basket)
11 }
```

Codebeispiel 6.4: Ausführung des Bezahlvorgangs in einem Domainservice

- Zeile 2: Laden des aktualisierten Baskets aus dem Repository.
- Zeile 3-5: Weist die Durchführung zurück, sofern der Basket sich nicht in den erwarteten Zustand befindet. Dies kann auftreten, wenn die REST-API aufgerufen worden ist, ohne dass ein Zahlungsprozess zuvor gestartet wurde.
- Zeile 6-8: Durchführung des Bezahlvorgangs. Die erforderliche Aufrufreihenfolge stellt einen Teil des Domainwissens dar und begründet die Zuteilung der Klasse in die Gruppe der Domainservices.
- Zeile 9: Speichern des angepassten Warenkorbs.
- Zeile 10: Erstellen eines Bestellvorgangs durch einen separaten Domainservice.

6.3 Anbinden externer Systeme und Datenbanken durch sekundäre Adapter

Das, in der Planungsphase erstellte, Context-Diagramm 3.10 zeigt verschiedenste Systeme mit denen die Anwendung zum Erfüllen ihrer Aufgaben kommunizieren muss. Für diesen Zweck wurden zwei Gruppen von Komponente eingeführt. Die Service, welche Aufrufe von externen API-Schnittstellen simulieren, und die sekundären Adapter. Letztere implementieren ein vom Applikationskern definiertes Interface, damit eine Brücke zwischen dem Domainkern und den APIs der Nachbarsysteme gewährleistet ist. Analog zu den primären Adapter, existieren im Anwendungskern keine direkten Abhängigkeiten zu diesem Teil der Software.

Ein Spezialfall sind hierbei die Service für das Erfragen der aktuellen Preis- bzw. Artikelinformationen. Aufgrund von Performance-Verbesserungen wurde eine Abwandlung der Adapter mit Caching-Funktion zusätzlich erstellt. Der normal fungierende Service ruft den zugehörigen API-Service auf, wohingegen der Caching-Adapter zuerst den Preis bzw. Artikel aus dem Cache lädt und zurückgibt, sofern dieser noch als aktuell markiert ist. Sollte der Eintrag veraltet sein, wird der eigentliche Adapter angesprochen, um die zum jetzigen Zeitpunkt validen Daten aus dem externen System zu erfragen. Im Beispiel 6.5 ist die, für das Aktualisieren des Preises zuständige, Klasse abgebildet.

```
1  class CachedPriceAdapter {
2
3      variable PriceRepository priceRepository
4      variable PriceAdapter priceAdapter
5
6      function fetchPrice(PriceId priceId) returns Price {
7          return priceRepository.getAndUpdateIfInvalid(priceId, fallback = {
8              priceAdapter.fetchPrice(priceId)
9          })
10     }
11 }
```

Codebeispiel 6.5: Preisadapter mit Caching-Funktion

- Zeile 3-4: Der Cache-Preisadapter hat eine Abhängigkeit zum normalen Preisadapter und zu einem Repository zum Abrufen des zwischengespeicherten Preises
- Zeile 7: Abfragen des Preises aus dem Cache-Repository. Sollte der Preis invalide sein wird Zeile 8 ausgeführt.
- Zeile 8: Weiterleitung der Anfrage an den zuständigen Adapter. Das Ergebnis wird mit einem aktuellen Zeitstempel im Cache abgelegt.

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.

7 Fazit und Empfehlungen

Die Begründungen für einen anderen Aggregationsschnitt bilden sind nach den Grundprinzipen des Domain-Driven Designs aus erhöhter Skalierbarkeit und Performance. Letzteres wurde anhand der Performance-Analyse widerlegt, da sowohl für den Proof-of-Concept als auch für die Live-Umgebung eine MongoDB eingesetzt wird. Die Förderung der Skalierbarkeit wird durch weniger Datenbankoperationen erhöhte Parallelität erreicht. Auch dieser Aspekt findet kaum eine Anwendung auf mögliche Aggregationsschnitte. Hinzukommt, dass die Softwarekomplexität durch andere Architekturen weiter ansteigt. Letztendlich besteht kein Anreiz für ein Neudesign der Anwendung.

Sofern zukünftig eine Notwendigkeit zur parallelen Bearbeitung der Artikel im Warenkorb besteht, kann positiv für einen Abtrennung der Items argumentiert werden. Jedoch besteht aktuell kein solcher Anwendungsfall.

Dieses Resultat kommt aus dem hohen logischen Zusammenhalt des Warenkorbs zustande. Jedes mögliche Aggregate hat Abhängigkeiten zu anderen Aggregates, wodurch die transaktionalen Grenzen sich über den ganzen Warenkorb spannen. Andere Applikationen können weiterhin von kleineren Aggregationsschnitten profitierten, dies ist allerdings hier nicht anwendbar.

Als Empfehlungen kann dargeboten werden, dass durch eine genaue Analyse der Anwendungsfälle und Integritätsgrenzen der idealen Aggregationsschnitt sich herauskristallisiert. Invarianten zwischen Objekte bestimmen maßgeblich mögliche Designs. Weiterhin kann durchaus verwendete Technologien einen Einfluss auf die Architektur der Software haben, allerdings sollte dies mit dem Bewusstsein geschehen, dass sich Techniken zeitnahe ändern können und weiterhin ein Risiko bei deren Einbindung in den Entscheidungsprozess darstellen. Tatsächliche Anforderungen an die Applikation und in der Praxis existierende Kompromisse stehen meist über den theoretischen Prinzipien des Softwaredesigns. Diese bieten zwar Richtlinien für eine langlebige Anwendung, allerdings sind sie nicht immer die beste Lösung für ein konkretes Problem und der zugrundeliegenden Antrieb für ihre Einhaltung sollte hinterfragt werden. Nichtsdestotrotz ist in den meisten Fällen ein kleinerer Aggregationsschnitt zu bevorzugen, da viele Applikationen keine starken Invarianten besitzen. Lediglich eine Checkout-Software hat viele Businessbedingungen, welche aus rechtlichen Gründen streng überprüft werden müssen.

8 Anhang

8.1 Ergebnisse des Lasttests

=:

Name	Anzahl Durchläufe	min. AZ	max. AZ	durchschn. AZ	Median der AZ	Durchläufe pro Sekunde	Testdauer in Millisekunden
Variante A-M	100	16	53	28,66	29	92,08	1086,00
Variante A-M	100	17	51	24,70	25	88,42	1131,00
Variante A-M	100	16	53	28,19	25	91,07	1098,00
Variante A-M	1000	16	83	43,56	42	197,36	5067,00
Variante A-M	1000	15	92	41,56	40	207,13	4828,00
Variante A-M	1000	16	86	41,53	41	205,00	4878,05
Variante A-M	10000	15	78	37,67	36	259,24	38574,00
Variante A-M	10000	15	76	36,86	35	265,32	37690,00
Variante A-M	10000	15	78	36,78	35	266,16	37571,00
Variante D-FM	100	21	55	29,10	29	88,18	1134,00
Variante D-FM	100	22	58	31,03	30	86,43	1157,00
Variante D-FM	100	22	60	33,25	31	86,73	1153,00
Variante D-FM	1000	21	75	42,46	41	199,60	5010,02
Variante D-FM	1000	22	83	44,97	44	189,47	5278,00
Variante D-FM	1000	21	80	43,41	42	195,69	5110,00
Variante D-FM	10000	21	73	37,00	35	264,63	37788,00
Variante D-FM	10000	20	97	36,82	35	266,36	37543,01
Variante D-FM	10000	20	72	36,83	35	265,82	37619,99
Variante D-CM	100	19	54	25,22	24	90,33	1107,00
Variante D-CM	100	18	60	28,40	27	90,42	1106,00
Variante D-CM	100	18	63	27,24	28	91,32	1095,00
Variante D-CM	1000	19	72	39,67	38	211,77	4722,00
Variante D-CM	1000	18	69	40,13	39	211,46	4729,00
Variante D-CM	1000	17	79	38,92	38	215,52	4640,00
Variante D-CM	10000	17	93	32,82	31	297,78	33582,00
Variante D-CM	10000	17	89	33,17	31	293,90	34025,00
Variante D-CM	10000	17	91	33,42	31	292,53	34185,00

Tabelle 8.1: Analyseergebnis des Lasttests der verschiedenen Variationen in Kombination mit MongoDB

Name	Anzahl Durchläufe	min. AZ	max. AZ	durchschn. AZ	Median der AZ	Durchläufe pro Sekunde	Testdauer in Millisekunden
Variante A-P	100	37	65	48,08	48	76,75	1303,00
Variante A-P	100	34	77	55,08	55	73,58	1359,00
Variante A-P	100	35	80	55,04	54	70,95	1409,45
Variante A-P	1000	32	149	60,15	57	148,39	6739,00
Variante A-P	1000	33	133	60,02	57	147,21	6793,00
Variante A-P	1000	34	118	59,83	59	148,35	6741,00
Variante A-P	10000	33	93	57,28	57	170,35	58703,02
Variante A-P	10000	33	109	55,62	55	175,18	57083,00
Variante A-P	10000	33	138	55,61	55	174,23	57395,99
Variante D-FP	100	42	93	63,38	63	69,93	1430,00
Variante D-FP	100	42	90	63,53	63	70,13	1426,00
Variante D-FP	100	39	169	69,44	64	69,49	1439,00
Variante D-FP	1000	39	170	62,82	60	143,66	6961,00
Variante D-FP	1000	37	148	60,63	59	148,13	6751,00
Variante D-FP	1000	38	88	61,54	60	143,55	6966,00
Variante D-FP	10000	37	135	56,06	55	174,40	57338,00
Variante D-FP	10000	41	113	58,71	58	166,68	59994,98
Variante D-FP	10000	38	116	56,99	56	172,22	58063,98
Variante D-CP	100	31	72	47,26	44	78,93	1267,00
Variante D-CP	100	31	70	45,93	45	77,82	1285,00
Variante D-CP	100	30	74	46,81	46	80,13	1248,00
Variante D-CP	1000	30	78	48,06	47	180,86	5529,00
Variante D-CP	1000	30	105	50,88	49	170,68	5859,00
Variante D-CP	1000	30	75	50,10	50	173,97	5748,00
Variante D-CP	10000	34	94	48,87	49	199,60	50099,00
Variante D-CP	10000	34	111	49,95	50	195,57	51132,01
Variante D-CP	10000	34	93	48,28	48	200,92	49771,99

Tabelle 8.2: Analyseergebnis des Lasttests der verschiedenen Variationen in Kombination mit Postgres

Name	Anzahl Durchläufe	min. AZ	max. AZ	durchschn. AZ	Median der AZ	Durchläufe pro Sekunde	Testdauer in Minuten
Variante A-M	10000	563	1526	612,04	612	16,30	10,22
Variante A-M	10000	565	725	617,67	617	16,06	10,38
Variante A-M	10000	576	743	619,05	617	16,04	10,39
Variante D-FM	10000	1405	2904	1625,89	1612	6,13	27,19
Variante D-FM	10000	1510	1720	1608,00	1608	6,19	26,91
Variante D-FM	10000	1512	1970	1654,91	1646	6,01	27,72
Variante D-CM	10000	1210	3307	1380,43	1364	7,23	23,06
Variante D-CM	10000	1207	1675	1384,97	1368	7,17	23,26
Variante D-CM	10000	1219	1637	1374,96	1365	7,21	23,11
Variante A-P	10000	3354	4237	3848,23	3836	2,49	66,96
Variante A-P	10000	3402	4305	3769,70	3765	2,59	64,46
Variante A-P	10000	3364	4911	3833,21	3796	2,36	70,67
Variante D-FP	10000	5386	6919	5839,03	5804	1,66	100,49
Variante D-FP	10000	5168	6883	5778,68	5725	1,52	109,65
Variante D-FP	10000	5235	6174	5727,97	5720	1,67	99,97
Variante D-CP	10000	4189	5799	4571,81	4549	2,11	79,14
Variante D-CP	10000	3919	5553	4553,76	4554	2,12	78,61
Variante D-CP	10000	4000	5702	4544,34	4527	2,02	82,69

Tabelle 8.3: Lasttest-Ergebnisse mit Datenbanken von einem externen Cloud-Anbieter

Name	Anzahl Durchläufe	min. AZ	max. AZ	durchschn. AZ	Median der AZ	Durchläufe pro Sekunde	Testdauer in Minuten
Variante A-M	10000	488	605	504,39	503	19,68	8,47
Variante A-M	10000	489	585	502,43	501	19,71	8,45
Variante A-M	10000	490	604	502,69	501	19,73	8,45
Variante D-FM	10000	494	604	511,01	511	19,39	8,59
Variante D-FM	10000	494	600	509,20	508	19,38	8,60
Variante D-FM	10000	492	601	508,31	507	19,50	8,55
Variante D-CM	10000	490	575	503,62	502	19,69	8,46
Variante D-CM	10000	491	627	503,50	502	19,68	8,47
Variante D-CM	10000	491	571	505,90	504	19,61	8,50
Variante A-P	10000	507	589	522,61	522	18,97	8,78
Variante A-P	10000	507	630	519,33	518	19,08	8,73
Variante A-P	10000	507	602	521,18	520	19,03	8,76
Variante D-FP	10000	511	594	526,00	525	18,86	8,84
Variante D-FP	10000	511	600	526,29	525	18,82	8,85
Variante D-FP	10000	512	626	526,81	525	18,80	8,87
Variante D-CP	10000	504	580	517,00	516	19,14	8,71
Variante D-CP	10000	502	611	518,79	518	19,11	8,72
Variante D-CP	10000	505	594	518,93	518	19,09	8,73

Tabelle 8.4: Lasttest-Ergebnisse mit Simulation der API-Aufruf durch künstliche Verzögerung

Literatur

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