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

Bachelorarbeit

Simon Thalmaier
00108692

Erstprüfer	Prof. Dr. Sebastian Apel
Zweitprüfer	Prof. Dr. rer. nat. Franz Regensburger
Betreuer	Stefano Lucka
Ausgabedatum	15.03.2022
Abgabedatum	09.06.2022

Erklärung zur Bachelorarbeit

Ich erkläre hiermit, dass ich diese Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ingolstadt, 09.06.2022 _____

Simon Thalmaier

Abstract

In the e-commerce sector software projects fulfill complex business requirements and therefore demand a stable architecture as their foundation. This thesis examines an online shop checkout domain and how the bounded context can be implemented utilizing hexagonal architecture and domain-driven design. The focus is placed on the design of the aggregates. Various data models are analyzed and evaluated based on their complexity, performance, parallelism and client-friendliness. Generally, big aggregates suffer from lower parallelism, however can be implemented more easily since all required information is loaded at once from the database. Splitting the data model into different aggregates entails the usage of eventual consistency or force transactions to modify more than one aggregate at the same time. Eventual consistency increases the complexity of the checkout software, on the other hand cross-aggregate transactions make the operation of multiple database hosts within one application more difficult. The performance of individual aggregate designs are measured with a load test. On average, applications with one cohesive aggregate process more requests per second than a model using separated aggregates. Conclusively, the checkout software profits from a higher performance and reduced complexity by implementing only one aggregate. If a common use case is the mutation of one resource by distinct processes simultaneously then the affected objects need to be placed in different aggregates.

Zusammenfassung

Softwareprojekte im E-Commerce-Bereich erfüllen komplexe Businessanforderungen und benötigen aufgrund dessen eine stabile Architektur. Dieses Projekt untersucht die Domain eines Onlineshop-Checkouts und wie der Bounded-Context mithilfe einer Hexagonalen Architektur und Domain-Driven Design implementiert werden kann. Besonders liegt der Aggregationsschnitt im Fokus, wobei unterschiedliche Datenmodelle analysiert und anhand von Komplexität, Performance, Parallelität und Client-Freundlichkeit bewertet werden. Große Aggregates leiden generell unter verringerter Parallelität, jedoch bietet ein zusammengehöriges Datenmodell eine vereinfachte Umsetzung von Businessanforderungen, da stets alle Informationen aus der Datenbank geladen werden. Die Aufteilung in mehrere Aggregates erzwingt die Anwendung von Eventueller Konsistenz oder einer Transaktion über mehrere Aggregates. Eventuelle Konsistenz erhöht die Komplexität der Checkout-Software, wohingegen eine aggregate-übergreifende Transaktion die Verwendung von unterschiedlichen Datenbank-Hosts erschwert. Anhand eines Lasttests wird die Performance der Designansätze betrachtet. Im Durchschnitt verarbeitet die Applikation mit einem zusammengehörigen Datenmodell mehr Anfragen pro Sekunde als unter Verwendung getrennter Aggregates. Als Fazit dieser Arbeit wird argumentativ begründet, dass innerhalb einer Checkout-Software die Vorteile eines Designs mit einem einzigen Aggregate dank der erhöhten Performance und reduzierten Komplexität überwiegen. Falls eine zeitgleiche Bearbeitung einer Ressource durch unterschiedliche Prozesse ein gängiger Anwendungsfall ist, müssen die betroffenen Objekte in separate Aggregates verlagert werden.

Danksagung

Mein Dank gilt der Firma MediaMarktSaturn Technology, welche mir ermöglicht hat als Werkstudent über die vergangenen drei Jahre zu arbeiten und das Thema für diese Bachelorarbeit bereitgestellt hat. Besonders bedanke ich mich bei Sebastian Jurjanz für die Unterstützung in meiner Ausbildung und während dieses Projektes. Zusätzlich ist die Bearbeitung des Forschungsthemas in diesem Umfang durch Stefano Lucka dank seiner Betreuung und seinem Feedback ermöglicht worden.

Aufseiten der Technischen Hochschule Ingolstadt bedanke ich mich bei Professor Dr. Sebastian Apel für das konstruktive Feedback und die umfangreiche Beratung.

Inhaltsverzeichnis

Darstellungsverzeichnis

Codebeispiel-Verzeichnis

Akronyme

CQRS	Command-Query-Responsibility-Segregation. Trennung des Datenmodells in Befehle für Schreiboperationen und Abfragen für Leseoperationen zum Erreichen einer Aufteilung der Zuständigkeiten und erhöhter Performance. [1, S. 223ff.]
CRUD	Create Read Update Delete
DDD	Domain-Driven Design
DIP	Dependency-Inversion-Prinzip
HTTP	Hypertext Transfer Protocol
ISP	Interface-Segregation-Prinzip
KPI	Key-Performance-Indicator
LSP	Liskovsches Substitutionsprinzip
OCP	Open-Closed-Prinzip
REST	Representational State Transfer. Die Transition von Zuständen der Clients wird durch Abrufen einer Ressource des Servers erreicht. [2]
SRP	Single-Responsibility-Prinzip

Glossar

Anemic Domain Model	Ein Anti-Pattern in welchem die Domain-Objekte keine bzw. kaum Businesslogik implementieren [3]
Boilerplate-Code	Ein Codeabschnitt, welcher viele Zeilen im Quelltext einnimmt bzw. wiederholt in diesem vorkommt, obwohl hierdurch nur wenige bis gar keine Funktionen bereitgestellt werden [4, 5]
Collection	MongoDB persistiert Datensätze in Collections, welche gleichbedeutend sind mit Tabellen einer relationalen Datenbank [6]
Connection-Pool	Eine Gruppe von Verbindungen zu Datenbanken oder APIs zur Performance-Optimierung und Isolierung, indem zuvor erzeugte Verbindungen wiederverwendet werden [7]
Dependency Injection	Eine Anwendung des Inversion-of-Control-Prinzips, welches Implementierungen zu passenden Abstraktionen erst zur Laufzeit des Programmes lädt [8]
Immutable	Die Unveränderlichkeit von Werten bzw. Variablen
Information-Expert-Prinzip	Die Verantwortung einer Funktionalität soll bei der Komponente liegen, welche die notwendigen Informationen zur erfolgreichen Abwicklung besitzt [9, S. 218]
Invariante	Businessbedingung, welche jederzeit erfüllt sein muss [10, S. 353]
Kohäsion	Grad der logischen inneren Zusammengehörigkeit einer Komponente. Komponenten, welche nur eng beinaheliegende Aufgaben erfüllen, haben einen hohen Grad an Kohäsion [11, S. 95]
Lazy Loading	Unnötiges Zuvorladen wird minimiert, indem das Laden der Daten aus dem Datenspeicher erst bei ihrem konkreten Aufruf stattfindet. [12, S. 200]
Lost Update	Phänomen, welches bei zeitgleichen Operationen auf den gleichen Datensätzen innerhalb einer Datenbank auftreten kann. Die angepassten Datensätze einer Transaktion gehen verloren, da sie direkt von einer zweiten Transaktion überschrieben werden. Die zweite Transaktion wurde jedoch noch auf dem alten Datenstand durchgeführt [13]
Product Owner	Eine Scrum-Rolle, welche die Verantwortung über die Arbeitsergebnisse des Teams besitzt und hierbei ihre Produktivität maximiert [14]
Race Condition	Zwei gleichzeitg bzw. nahezu gleichzeitig stattfindende Prozesse bedingen sich gegenseitig und führen zu nicht vorgesehenem Verhalten [15]

Scrum	Ein agiles Vorgehensmodell, welches hohen Fokus auf kontinuierliche Verbesserung in einem geregelten Zyklus legt [16]
Serialisierung	Konvertierung von Datenobjekten in ein sequenzielles Format
Sprint	Ein wiederkehrender festgelegter Zeitraum in Scrum, indem ein vorher definierter Umfang an Arbeitspaketen abgearbeitet wird [17]
Stakeholder	Ein Zusammenschluss von Personen außerhalb des Teams mit Interesse und/oder Einfluss auf das Projekt [18]
technische Schulden	Bewusst akzeptierte Vernachlässigung von qualitätsschadenden Eigenschaften einer Software [19]
User Story	Ein möglicher Ablauf des kompletten Businessprozesses aus Sicht des Kunden

1 Einleitung

Heutzutage werden Applikationen für den langfristigen Gebrauch in einer produktiver Umgebung entwickelt. Im Durchschnitt kann hierbei der erwartete Lebenszyklus dieser Anwendungen auf circa 10 Jahre festgelegt werden [20], weshalb sich über die vergangenen Jahrzehnte ein starker Fokus auf ein wartbares und flexibles Softwaredesign gebildet hat. Darüber hinaus müssen sie umfangreiche Anwendungsprofile fehlerfrei und performant bewältigen, wodurch ebenfalls ein stabiler Architekturansatz von Nöten ist [21]. Software Engineers stehen daher mittlerweile eine große Bandbreite an Entwurfsmustern und Anti-Pattern zur Verfügung, wie beispielsweise die Hexagonale Architektur, Event-Storming, *Anemic Domain Model* und Microservices. Letzteres gewann aufgrund ihrer Skalierbarkeit und losen Kopplung der Komponenten in den vergangenen Jahren zunehmend an Bedeutung [22, 23]. Gleichzeitig erfuhr das im Jahre 2011 erschienene Buch *Domain-driven design: Tackling complexity in the heart of software* von Evans an Beliebtheit, da viele der hierin bearbeiteten Themenschwerpunkte auf Microservices adaptiert werden können [10, S. 130ff.][25]. Domain-Driven Design bietet zum Lösen gängiger Problemstellungen in der Softwareentwicklung ein Vorgehensmodell für die Realisierung eines businessorientierten Datenmodells, das bei der Umsetzung von Businessanforderungen unterstützt. Die Checkout-Software, welche im Rahmen dieser Bachelorarbeit als Proof-of-Concept entwickelt wird, profitiert von den Vorteilen einer solchen Architektur.

1.1 Problemstellung

Eine elementare Funktion eines Onlineshops ist der Warenkorb. In diesem können Waren zum späteren Erwerb abgelegt oder zusätzliche Dienstleistungen hinzugefügt werden. Im Verlauf des Kaufprozesses ist es zudem möglich, Kundendaten zu hinterlegen, sowie die gewünschte Versandart und Zahlungsarten auszuwählen. Nach erfolgreicher Überprüfung der Validierungsrichtlinien findet die Kaufabwicklung statt – der sogenannte 'Checkout'. Um die zuvor genannten Anwendungsfälle umzusetzen, wird eine Softwarekomponente benötigt. In dieser Bachelorarbeit wird diese Anwendung vereinfacht implementiert und als 'Checkout-Software' bezeichnet. Sie erfährt stetige Änderungen, beinhaltet im Vergleich zu anderen Softwareprojekten umfangreiche Businesslogik und ihre Antwortzeiten haben durch die Einbindung in das Frontend auch direkten Einfluss auf das Kundenerlebnis. Dadurch liegt stets ein Fokus auf die Erfüllung von Qualitätsanforderungen, wie Stabilität, Testbarkeit und Wartbarkeit. Der Checkout-Prozess muss für alle Länder, in denen die Applikation eingesetzt wird, und ihre individuellen gesetzlichen Anforderungen fiskalisch korrekt ausgeführt werden. Jederzeit können neue Businessanforderungen entstehen, wodurch weitere länderspezifische Richtlinien in den Zuständigkeitsbereich der Anwendung fallen und beispielsweise eine Anpassung des Datenmodells erfordern. Zudem benötigt das System zur Abwicklung ihrer Arbeitsprozesse Daten aus verschiedensten Unternehmensbereichen wie Preise, Lieferkosten, Produkt- und Bestandsinformationen. Die Kommunikation mit externen Komponenten erhöht die Komplexität der Anwendung und erfordert die Berücksichtigung systemübergreifender Anforderungen. Eine große Rolle spielt hierbei die Performance. Vor allem bei hoher Auslastung, beispielsweise während Kampagnen, muss das Gesamtsystem weiterhin zuverlässig alle Anfragen in akzeptabler Zeit abarbeiten können. Dementsprechend stellt die Implementierung einer solchen Checkout-Applikation für Software Engineers eine große Herausforderung dar. Sollte die verwendete Architektur im langjährigen Entwicklungsprozess an Übersichtlichkeit verlieren, leidet zugleich auch die Wartbarkeit des Sourcecodes. Als Folge können weitere Qualitätsmerkmale negativ betroffen sein und der Aufwand zur Umsetzung von neuen Businessanforderungen steigt [26, S. 3f.]. Daraus ergibt sich für den Checkout der grundsätzliche Bedarf zur Einhaltung bestimmter Industriestandards hinsichtlich der Softwarearchitektur und des Datenmodells. Somit besteht ein Teilaspekt der Problemstellung in der Auswahl einer geeigneten Architektur zur Realisierung der Software.

Auf Basis der kommenden Kapitel wird die Verwendung einer Hexagonalen Architektur inklusive Domain-Driven Design für den Proof-of-Concept argumentativ begründet. Die Projektdurchführung orientiert sich hierbei am empfohlenen Entwicklungsprozess von Domain-Driven Design. Im Fokus der Bachelorarbeit stehen die möglichen Einteilungen des Datenmodells in Aggregates, in dieser Arbeit als 'Aggregationsschnitt' bezeichnet, welche durch die Untersuchung der Anwendungsfälle erschlossen werden. Das Forschungsthema leitet sich aus der Frage ab, welche funktionalen und nicht-funktionalen Implikationen die unterschiedlichen Aggregationsschnitte auf die Applikationen besitzen. Hierzu werden sie in Form eines Proof-of-Concepts implementiert, analysiert und anhand von Performance-Tests bewertet.

1.2 Das Unternehmen MediaMarktSaturn

Dieses Projekt wurde in Zusammenarbeit mit dem Unternehmen *MediaMarktSaturn Retail Group GmbH*, kurz *MediaMarktSaturn*, erarbeitet. Mit über 1.000 Märkten in 13 Ländern sowie den Onlineshops ist MediaMarktSaturn Europas größter Anbieter von Unterhaltungselektronik sowie zugehöriger Dienstleistungen und Services. Dabei soll eine umfangreiche Produktauswahl in Kombination mit passenden Services und Kundennähe ein einzigartiges Einkaufserlaubnis schaffen - über alle Verkaufskanäle hinweg. Die Zugehörigkeiten der Märkte ist hierbei in die Marken *Media Markt* und *Saturn* unterteilt. [27]

Wegen des massiv steigenden Onlineanteils gewann der Onlineshop in den letzten Jahren für Media Markt und Saturn zunehmend an Bedeutung. Der Ausbruch der COVID-19-Pandemie und die damit verbundene europaweite Schließung der Märkte hat die Verlagerung der Umsatzeinnahmen vom Markt- zum Onlinegeschäft nochmals verschärft. Folglich wurden die Unternehmensziele auf die Entwicklung komplexer Software zur Unterstützung des Onlineshops neu ausgelegt. Die Umsetzung obliegt der zentralen IT-Gesellschaft MediaMarktSaturn Technology, die über 700 interne und 1000 externe Engineers in einer skalierten Produktorganisation mit dem Ziel der steten Optimierung der Systemlandschaft beschäftigt [28].

Die Durchführung des Projektes bzw. Proof-of-Concepts erfolgte in Kooperation mit dem Bereich *Checkout & Payment*. Die sechs Teammitglieder sind dafür zuständig einen unternehmensweiten, universellen Checkout für alle Länder bereitzustellen – sowohl für den Onlineshop als auch im Markt und per Handyapp. Durch den Einsatz von *Scrum* wird eine konstante Verbesserung der Applikation und des Arbeitsprozesses erzielt. In iterativen *Sprints* wird die Checkout-Software auf Basis von neuen Anwendungsfällen kontinuierlich erweitert. Dieses Projekt soll dem Team als Revision dienen und Schlussfolgerungen über mögliche architektonische Designansätze liefern.

1.3 Motivation

Durch den fortlaufenden Anstieg der Komplexität von Softwareprojekten [29] haben sich gängige Designprinzipien und Architekturstile für den Entwicklungsprozess etabliert, sodass auch weiterhin die Businessanforderungen in einem zukunftssicheren Ansatz erfüllt und die Prozessabläufe jederzeit angepasst und erweitert werden können. Deshalb ist zur Gewährleistung der Langlebigkeit einer solchen Software eine flexible Grundstruktur entscheidend. Folglich ist eine sorgfältige Projektplanung und stetige Revision der Produktivanwendung 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 überprüft die Architektur auf Verbesserungsmöglichkeiten und eventuelle Schwachstellen. Zudem existieren aufgrund des aktuellen Aggregationsschnitts gewisse Nachteile hinsichtlich der Performance und gleichzeitigen Bearbeitungen von Ressourcen. In diesem Projekt wird analysiert, ob die Performance durch eine andere Aufteilung des Datenmodells nachhaltig gesteigert werden kann. Dies dient ebenfalls als Referenz für zukünftige Softwareprojekte, die mit ähnlichen Problemstellungen konfrontiert sind.

1.4 Ziele

Aus der im vorgehenden Kapitel definierten Motivation lassen sich folgenden Projektziele ableiten. Anhand der Analyse und Durchführung des Proof-of-Concepts wird das bestehende Softwaredesign überprüft und herausgefordert. Dabei können konkrete Verbesserungsvorschläge an die produktive Applikation ein mögliches Fazit der Arbeit sein. Sollten sich durch ein anderes Design des Datenmodells erhebliche Vorteile ergeben, kann dies in einem Umbau der Software resultieren. Die Erkenntnisse der Arbeit sind ein wichtiges Ergebnis für das Team und Unternehmen, denn Projekte können auf ihrer Basis eine Architektur und Datenmodell nachhaltig implementieren.

2 Grundlagen

Für das Verständnis des Bachelorthemas werden Kernkompetenzen der Softwareentwicklung vorausgesetzt. Dies betrifft vornehmlich Softwaredesign und Architekturstile. Um nachzuvollziehen, 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 etablierte Designprinzipien über die Jahre festgelegt. Unter anderem die sogenannten 'SOLID'-Prinzipien, die dazu beitragen Architekturansätze miteinander zu vergleichen und zu bewerten.

2.1 SOLID-Prinzipien

Die SOLID-Prinzipen sollen sicherstellen, dass Software auch mit zunehmendem Funktionsumfang weiterhin testbar, anpassbar und fehlerfrei bleibt [30, 31]. Das weitverbreitete Akronym 'SOLID' steht hierbei für die fünf Designprinzipien:

Single-Responsibility-Prinzip (SRP): 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. [32, 31]

Open-Closed-Prinzip (OCP): Um sicherzustellen, dass eine Änderung in einer Komponente keine Auswirkungen auf eine andere hat, werden diese als 'geschlossen' gegenüber Veränderungen aber weiterhin 'offen' für Erweiterungen definiert. Der erste Teil des Prinzips kann durch eine Schnittstelle, ein sogenanntes Interface, realisiert werden. Es gilt als geschlossen, da die Implementierungen keine Signaturänderungen der im Interface definierten Methoden vernehmen können. Ansonsten müsste der darauf basierende 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 betrachtet. [31, 33]

Liskovsches Substitutionsprinzip (LSP): 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 reduziert und der Client kann sichergehen, dass die Funktionalität auch weiterhin den erwarteten Effekt hat. Da sich das LSP mit der Komposition von Klassen beschäftigt, ist es für die nachfolgende Architekturanalyse vernachlässigbar. [31, 34]

Interface-Segregation-Prinzip (ISP): 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 Komponenten gewährleistet. [31][35, S. 135ff.]

Dependency-Inversion-Prinzip (DIP): Module sollten so unabhängig wie möglich voneinander genutzt werden können. Dadurch wird eine erhöhte Testbarkeit und Wiederverwendbarkeit ermöglicht. Das zweiteilige DIP ist von zentraler Bedeutung für eine stabile und flexible Software. Hierbei sollen konzeptionell höhere Komponenten nicht direkt auf darunterliegende Ebenen angewiesen sein, sondern die Kommunikation zwischen ihnen über Interfaces geschehen. Dies erlaubt die Abstraktion von Funktionsweisen und löst die direkten Abhängigkeiten zwischen Modulen auf. Weiterhin wird festgelegt, dass Interfaces nicht an ihre Implementierungen gekoppelt werden sollten, sondern auf deren Abstraktionen

beruhen [36, 31]. Dadurch sind die Abhängigkeiten invertiert, was beispielhaft die Anwendung von *Dependency Injection* ermöglicht [8].

2.2 Architekturmuster

Eine Softwarearchitektur beschreibt die grundlegende Struktur der Module, ihre Relationen zueinander und die Art der Kommunikation zwischen den Modulen. Die Wahl der verwendeten Architektur beeinflusst somit die komplette Applikation und ihre Qualitätsmerkmale. Das zu bevorzugende Design einer Anwendung ist gekoppelt an die Anwendungsfälle und ihre Anforderungen.

In diesem Projekt soll ein Backend-Service erstellt werden, welcher mit den vorgelagerten Systemen über *HTTP* und *REST* kommuniziert, wodurch die Auswahl der Architekturen beschränkt wird. Ansätze wie Peer-to-Peer, welche eine Kommunikation zwischen zwei gleichberechtigten Knoten bereitstellen, sind somit in diesem Anwendungsgebiet nur bedingt vertreten. Etablierte Architekturen für Backend-Software, welche die Businessprozesse als Kern der Applikation halten, werden hingegen genauer untersucht. Die Schichtenarchitektur und Hexagonale Architektur werden als Grundlage für das Projekt herangezogen. Im folgenden Abschnitt werden beide Stile untersucht und anhand ihrer Tauglichkeit für eine Checkout-Software bewertet. Dabei wird hinterfragt, in wie fern Entwickler bei der Umsetzung der SOLID-Prinzipien unterstützt werden.

2.2.1 Schichtenarchitektur

In einer Schichtenarchitektur werden die Softwarekomponenten in einzelne Schichten eingeteilt. Die Anzahl der Schichten kann je nach Anwendungsfall variieren, liegt jedoch meist zwischen drei und vier Ebenen. Eine verbreitete Variante beinhaltet die Präsentations-, Business- und Datenzugriffsschicht. Dadurch wird eine Trennung der Verantwortlichkeiten erzwungen [37, S. 185]. Der Kontrollfluss der Applikation fließt hierbei stets von einer höheren Schicht in eine tiefer gelegene oder innerhalb einer Ebene zwischen einzelnen Komponenten. Ohne eine konkrete Umkehrung der Abhängigkeiten ist der Abhängigkeitsgraph gleichgerichtet zum Kontrollflussgraph. [12, S. 17ff.] Der beschriebene Aufbau einer solchen Architektur ist in Abbildung 2.1 beispielhaft dargestellt.

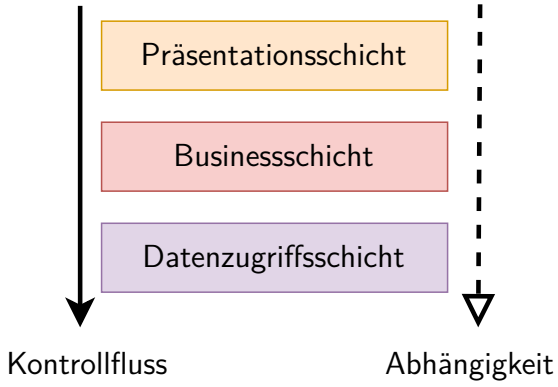


Abbildung 2.1: Beispielhafte Darstellung einer Drei-Schichtenarchitektur

Wesentliche Ziele einer Schichtenarchitektur sind die Entkopplung der einzelnen Schichten voneinander und das Erreichen von geringen Abhängigkeiten zwischen den Komponenten [12, S. 17]. Dadurch sollen Qualitätseigenschaften wie Testbarkeit, Erweiterbarkeit und Flexibilität erhöht werden. Dank des simplen Aufbaus gewann dieser Architekturstil an großer Beliebtheit. Weitere bewertende Aspekte einer solchen Softwarestruktur ergeben sich aus der Analyse der SOLID-Prinzipien:

Single-Responsibility-Prinzip: Durch die Schichteneinteilung wird die natürliche Einhaltung des SRPs unterstützt, da eine Komponente zum Beispiel keine Businesslogik und zugleich Funktionen der Datenzugriffsschicht implementieren kann. Nichtsdestotrotz ist eine vertikale Trennung innerhalb einer Schicht nicht gegeben, daher können weiterhin Klassen mehrere, konzeptionell verschiedene Aufgaben entgegen des SRPs erfüllen.

Open-Closed-Prinzip & Interface-Segregation-Prinzip: Um die einzelnen Schichten zu entkoppeln, kann die Kommunikation zwischen den Ebenen durch Schnittstellen geschehen. Dadurch wird eine grundlegende Befolgung des ISPs erreicht. Das Open-Closed-Prinzip soll hierbei helfen, dass Änderungen an den Schnittstellen und ihren Implementierungen die Funktionsweise nicht beeinflussen, auf denen tieferliegende Schichten basieren. Die logische Zuteilung dieser Interfaces ist entscheidend, um eine korrekte Anwendung des Dependency-Inversion-Prinzips zu gewährleisten.

Dependency-Inversion-Prinzip: Meist wird bei webbasierten CRUD-Applikationen eine Schichtenarchitektur verwendet. CRUD steht im Softwarekontext für '**C**reate **R**ead **U**pdate **D**elete' und meint Anwendungen, die Daten mit geringer bis keiner Geschäftslogik erzeugen, bearbeiten und löschen [38, S. 381]. Im Kern einer solchen Software liegen die Daten selbst. Dabei werden Module und die umliegende Architektur angepasst, um die Datenverarbeitung zu vereinfachen. Die Abhängigkeiten in einer Schichtenarchitektur richten sich daher oft von der Businessschicht zur Datenzugriffsschicht [39]. Bei einer Applikation, die als Hauptbestandteil Businesslogik enthält, sollte hingegen die Abhängigkeiten 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 im Grundaufbau wiederzuerkennen.

2.2.2 Hexagonale Architektur

Durch architektonische Vorgaben können Entwickler zu besserem 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. Der Aufbau wird in Abbildung 2.2 veranschaulicht.

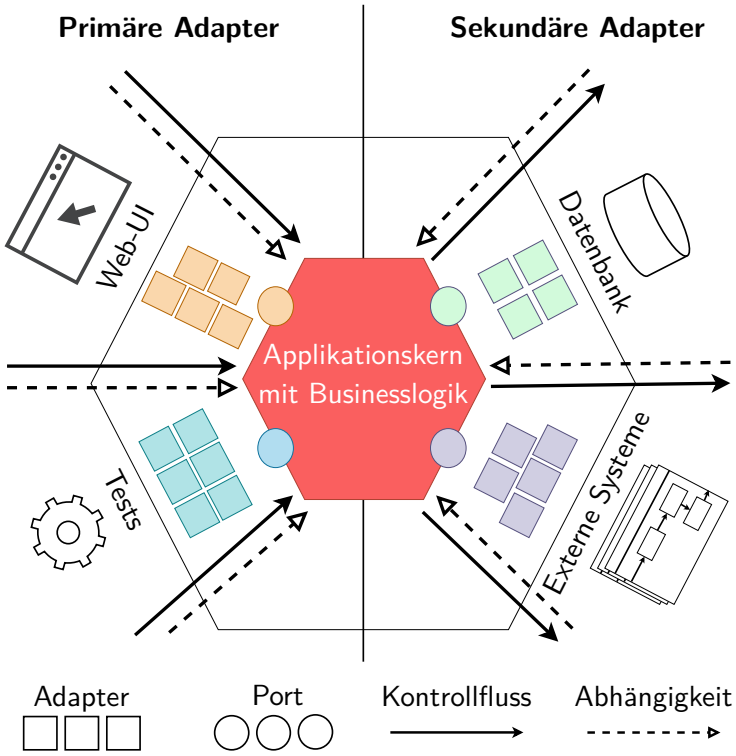


Abbildung 2.2: Grundstruktur einer Hexagonalen Architektur [angelehnt an 40]

Hierbei existieren drei Bereiche in denen die Komponenten angesiedelt werden können: [41, 42]

Ports: 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 anhand des Open-Closed-Prinzips. Realisiert werden Ports meist durch Interfaces, welche hierarchisch dem Zentrum zugeteilt und deren Design durch diesen maßgeblich bestimmt werden. Somit erfolgt die Einhaltung des *Dependency-Inversion-Prinzips*, wodurch die Applikationslogik von externen Systemen und deren konkreten Implementierungen abgekoppelt wird. Dies verringert die Abhängigkeiten zwischen Komponenten und erhöht zugleich die Testbarkeit der Anwendung. [43]

Adapter: Die Komponenten zwischen externen Systemen und der Geschäftslogik heißen Adapter. Ein *primärer Adapter* wird durch das externe System angestoßen, welcher daraufhin den Steuerfluss durch einen wohldefinierten Port in den Applikationskern trägt. Zu diesen externen Systemen zählen unter anderem Benutzerinterfaces, Kommandokonsolen sowie Testfälle. Andererseits bilden alle Komponenten, bei denen der Steuerfluss vom Applikationskern zu den externen Systemen gerichtet ist, die Gruppe der *sekundären Adapter*. So entsteht der Impuls im Vergleich zu den primären Adaptern nicht außerhalb der Applikation, sondern innerhalb. Die von den sekundären Adaptern angesprochenen Systeme können beispielsweise Datenbanken, Message-Broker und weitere Nachbarapplikation sein. [40]

Applikationskern: Letztendlich werden alle übrigen Module im Applikationskern erschlossen. Diese beinhalten Businesslogik und sind mithilfe der von ihnen zur Verfügung gestellten Ports von konkreten Implementierungen entkoppelt.

Zum Verdeutlichen der Funktionsweise einer hexagonalen Applikation wird ein simpler Anwendungsfall durchgespielt. Konkret sollen von Clients übertragene Daten in einer Datenbank gespeichert werden. Ein Webclient spricht eine Schnittstelle des Systems mit den Nutzdaten an, wodurch dieser den Steuerfluss der Applikation initiiert. Die Schnittstelle 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 angestoßen. Innerhalb des Applikationszentrums werden alle business-relevanten Aufgaben erfüllt. Darunter fallen das logische Überprüfen der Werte anhand von Businessrichtlinien, Erstellen neuer Daten und die Steuerung des Entscheidungsflusses. In diesem Anwendungsfall sollen die Nutzdaten in einer Datenbank abgespeichert werden. Dementsprechend wird aus dem Anwendungskern über einen weiteren Port ein sekundärer Adapter aufgerufen, welcher die dauerhafte Speicherung in der Datenbank übernimmt.

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

Single-Responsibility-Prinzip: Durch die Struktur wird eine strengere konzeptionelle Trennung der Verantwortlichkeiten ermöglicht. Dies wirkt sich positiv auf die Einhaltung des Single-Responsibility-Prinzips aus.

Open-Closed-Prinzip & Interface-Segregation-Prinzip: Als Folge der Nutzung von Ports zwischen dem Applikationskern und den außenstehenden Komponenten ist die Anwendung der beiden Prinzipien erleichtert und teilweise automatisch gegeben. Die Applikation profitiert von erhöhter Stabilität und Kohäsion.

Dependency-Inversion-Prinzip: In einer Hexagonalen Architektur ist das Dependency-Inversion-Prinzip fest durch die vorgeschriebene Komposition verankert. Dadurch wird das Austauschen von Komponenten ermöglicht, ohne dabei den Businesskern verändern zu müssen. Dies entkoppelt nicht nur den wichtigsten Bestandteil der Software, sondern fördert schlussfolgernd auch die Testbarkeit. Durch eine native Invertierung der Abhängigkeiten gewinnt somit die Applikation viele positive Qualitätsmerkmale. [44, 45]

So ergibt sich eine natürliche Einhaltung der SOLID-Prinzipien, wobei der Applikationskern in den Vordergrund gerückt wird. Anzumerken ist, dass erfahrene Entwickler ebenfalls mit einer Schichtenarchitektur ein gleiches Maß an Softwarequalität erzielen können, sofern die Designprinzipien diszipliniert eingehalten werden, da bei genauer Betrachtung eine Hexagonale Architektur äquivalent mit einer dreiteiligen Schichtenarchitektur mit erzwungenem Dependency-Inversion-Prinzip ist [46] [10, S. 125ff.].

2.3 Domain-Driven Design

In der Entwicklungsphase von komplexer Software besteht stets die Gefahr, dass sie zu einem sogenannten 'Big Ball of Mud' verschmelzt, weil die steigende Anzahl von Anforderungen und Codeänderungen die Übersichtlichkeit des Sourcecodes beeinträchtigt. Die bestehende Architektur wird unübersichtlich, 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. [47] Die oben analysierten Architekturstile können bei strikter Umsetzung diese Risiken minimieren, jedoch bestimmen sie nur begrenzt, wie das zugrundeliegende Datenmodell und die damit verbundenen Komponenten gestaltet werden sollen. In dem Buch *Domain-driven design: Tackling complexity in the heart of software* entwickelte Eric Evans im Jahr 2003 zu diesem Zweck Domain-Driven Design, kurz DDD. Der Buchtitel beschreibt bereits den Hauptgedanken hinter Domain-Driven Design. Liegen die Businessanforderungen im Herzen der Software, sollte dementsprechend auch ihre Implementierung zentral verankert sein. Der Applikationskern stellt somit den 'lebenden' Teil der Anwendung dar. Die verbleibenden Komponenten dienen zur Unterstützung der Businesslogik, indem sie benötigte Dienste dem Kern bereitstellen. Die Businessanforderungen werden somit in DDD strukturell aus dem Quelltext hervorgehoben. Das Datenmodell spiegelt zudem die Sprache der Geschäftsprozesse wider, wodurch die Realisierung der Applikationslogik erleichtert wird. Vor allem Anwendungen mit komplexen Entscheidungssträngen und vielen jederzeit gültigen Konditionen können dadurch übersichtlich implementiert werden. Zu diesem Zweck definiert Domain-Driven Design einige Vorgehensweisen, Richtlinien und Entwurfsmuster, welche in diesem Kapitel erläutert werden. [24, 10]

2.3.1 Unterteilung der Problemebene in Domains und Subdomains

Der Problemraum eines Projektes spannt in Domain-Driven Design eine *Domain* auf [10, S. 56]. Dieser Bereich umfasst logisch zusammengehörige Verantwortlichkeiten und Businessprozesse. Anfangs sollte die Domain anhand einer ausführlichen Umfeldanalyse definiert werden, damit alle Aspekte des Problemraums und seine Abhängigkeiten beleuchtet werden. Innerhalb einer Domain liegen die dazugehörigen *Subdomains*. Eine Subdomain repräsentiert einen kleineren, spezifischeren Teil der Domain, wodurch der Problemraum in mehrere Bereiche unterteilt wird. Sie helfen im nachfolgenden Schritt bei der Formulierung des Lösungsraumes. Zur Bestimmung der Subdomains werden die Verantwortlichkeiten stets aus Businesssicht betrachtet und technische Aspekte vernachlässigt. Der Domainumfang ist dabei entscheidend. Sollte dieser zu groß geschnitten sein, sind die Subdomains ebenfalls zu weitreichend. Das gefährdet die Kohäsion der Lösungsebene und somit der Software. Über den Verlauf der Entwicklungsphase könnten aufgrund dessen architektonische Konflikte auftreten. Enthält eine Subdomain mehrere logisch unabhängige Aufgaben, kann sie in kleinere Subdomains weiter unterteilt werden. Für einen Domain-Driven Ansatz ist es entscheidend die Definitionsphase gewissenhaft durchzuführen, damit eine stabile Grundlage für die Umsetzung des Projekts geschaffen werden kann.

2.3.2 Bounded-Contexts und ihre Ubiquitous Language

Als Ausgangspunkt für die Bestimmung der Lösungsebene dienen die sogenannten Bounded-Contexts [10, S. 57], welche eine oder mehrere Subdomains umfassen und ihre zugehörigen Verantwortlichkeiten bündeln. Wie es in der Praxis häufig der Fall ist, können Subdomains und Bounded-Contexts durchaus identisch sein [10, S. 57]. In jedem Bounded-Context sollte maximal ein Team tätig sein, um Kommunikationsprobleme zu vermeiden und eine klare Zuteilung der Kompetenzen zu gewährleisten [48]. Jeder Bounded-Context besitzt zudem eine zugehörige *Ubiquitous Language* [10, S. 62]. Sie wird als wichtiger Bestandteil während der Projektplanung festgelegt und definiert Begriffe, welche durch die *Stakeholder* und das Business verwendet werden. Dadurch können Missverständnisse in der Kommunikation zwischen dem Business und den Entwicklern vorgebeugt und eventuelle Inkonsistenzen aufgedeckt werden [24, S. 336f.]. Der größte Vorteil ergibt sich allerdings, sobald auch das Datenmodell diese Sprache wiedergibt. Entities können als Nomen dargestellt, Funktionen können als Verben definiert und Aktionen können als Events abgebildet werden, wodurch die Businessprozesse auch im Quelltext wiederzufinden sind. Folglich wird die Verständlichkeit und Wartbarkeit der Software [24, S. 24ff.] gesteigert. Zudem werden Entwickler bei der Umsetzung von Test- und Anwendungsfällen unterstützt, da ihre textuellen Definitionen auf das Datenmodell übertragbar sind. Zu beachten ist, dass die *Ubiquitous Language* nur innerhalb eines Bounded-Context Gültigkeit hat [10, S. 62]. Beispielhaft kann der Begriff 'Kunde' in einem Onlineshop einen zivilen Endkunden, jedoch im Wareneingang eine Lieferfirma beschreiben. Daher ist bei der Kommunikation zwischen Teams in unterschiedlichen Subdomains zu berücksichtigen, dass Begriffe eventuell verschiedene Bedeutungen besitzen.

Die Domains, Subdomains, Bounded-Contexts und ihre Kommunikation zueinander wird durch eine Context-Map dargestellt. Diese ist ein wichtiges Artefakt der Definitionsphase und kann als Werkzeug zur Bestimmung von Verantwortlichkeiten und der Einteilung neuer Anforderungen genutzt werden. Sollte eine eindeutige Zuteilung von Funktionalitäten nicht möglich sein, spricht dies für die Entstehung eines neuen Bounded-Contexts und eventuell einer Subdomain. Wie eine Software Anpassungen erlebt, entwickelt sich gleichermaßen die Context-Map stetig weiter. [10, S. 87ff] Zur Veranschaulichung wurde in Abbildung 2.3 das Personalwesen eines Unternehmens als Domain ausgewählt und in Subdomains bzw. Bounded-Contexts aufgeteilt. Abhängig von der Unternehmensgröße und -strategie können die Bounded-Contexts auch umfassender oder feingranularer ausfallen.

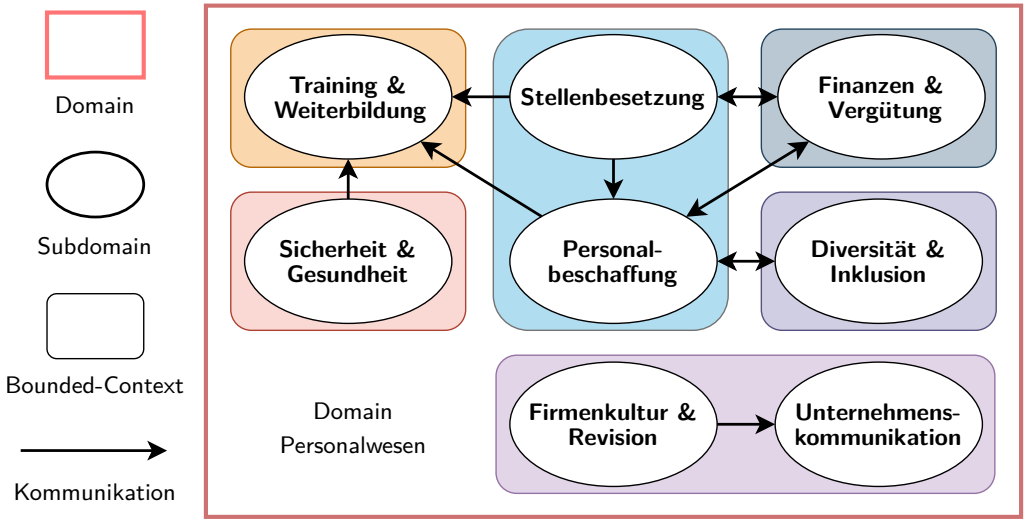


Abbildung 2.3: Beispiel einer Context-Map anhand des Personalwesens einer Firma

2.3.3 Kombination von Domain-Driven Design und Hexagonaler Architektur

Innerhalb eines Bounded-Contexts wird die grundlegende Architektur durch das zugehörige Team bestimmt. Diese kann sich je nach Sachverhalt des jeweiligen Anwendungsgebietes stark zwischen den Bounded-Contexts unterscheiden. Beliebte Modellierungs- und Designstile in Verbindung mit DDD sind unter anderem Microservices, *Command-Query-Responsibility-Segregation (CQRS)*, Event-Driven Design, Schichtenarchitektur und Hexagonale Architektur [10, S. 113ff.]. In den vorhergehenden Unterkapiteln wurden bereits die Vorzüge und Nachteile der zwei zuletzt genannten Architekturen erläutert. Auf Basis der Analyse wird generell für komplexere Software eine Hexagonale Architektur bevorzugt. Zudem steht im Zentrum von Domain-Driven Design und Hexagonaler Architektur das Domain-Modell, wodurch die Software an Kohäsion und Stabilität gewinnt. Die Kombination beider Ansätze ermöglicht es, bei häufigen technischen Neuerungen und komplexen Businessanforderungen weiterhin eine anpassbare, testbare und übersichtliche Software zu implementieren. 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, Modifikation von Attributen oder ihre dauerhafte Speicherung. Für diesen Zweck existieren in Domain-Driven Design mehrere Arten von Komponenten, welche anhand ihrer Verantwortlichkeiten unterschieden werden. Die korrekte Zuordnung der Klassen zu ihren Rollen ist entscheidend für einen skalierbaren Aufbau. Daher wird in den folgenden Unterkapiteln ein zentraler Überblick der einzelnen Bestandteile aufgeführt.

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 Objects 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 [10, S. 227]. Aus diesem Grund gelten Value Objects als *immutable*, da sie selbst keinen Werteverlauf besitzen [24, S. 99]. Eine Neuzuweisung der Attribute ist deshalb nicht möglich. Stattdessen wird die Referenz auf eine andere, angepasste Instanz der Klasse umgesetzt [10, S. 226]. Dies gilt als ein erstrebenswertes Designmuster, da unveränderbare Objekte eine erhöhte Wiederverwendbarkeit ermöglichen und unerwünschte Seiteneffekte vermieden werden [10, S. 228f.]. Folglich sind sie aufgrund des fehlenden Lebenszyklus lediglich eine Momentaufnahme des Applikationszustandes.

Beispiel: In den meisten E-Commerce Bounded-Contexts sind alleinig die konkreten Werte eines *Preises*, wie Bruttobetrag, Nettobetrag und Mehrwertsteuer relevant, weshalb dieser meist als Value Object angesehen wird. Sollten Preise die gleichen Wertebelegungen besitzen, gelten sie dementsprechend als identisch. Bei einer Aktualisierung eines Preises, kann das vorherige Objekt gelöscht und durch einen Preis mit den neuen Werten ersetzt werden. Ist es notwendig, den Werteverlauf des Preises über eine Zeitspanne zu verfolgen, wird oftmals eine ID innerhalb der Datenstruktur hinterlegt. Die Identität ist dadurch nur noch von der ID abhängig, nicht mehr von den Werten. Die Definition eines Value Objects trifft auf die Klasse nicht mehr zu und ein Design als Entity ist zu bevorzugen.

2.3.5 Entity

Im Gegenzug zu einem Value Object wird eine Entity nicht durch den Zusammenschluss ihrer Werte identifiziert, sondern enthält ein vordefiniertes Set an immutable Attributen, welche ihre Eindeutigkeit bestimmen [24, S. 94]. Auch nach dem Aktualisieren ihrer Informationen bleibt die ursprüngliche Identität bestehen. Demzufolge gelten die Attribute einer Entity als veränderlich und besitzen ihren eigenen Lebenszyklus, auch wenn dieser nicht explizit abgespeichert werden muss [10, S. 172]. In einer Entity werden Businessanforderungen, die sich auf enthaltenen Daten beziehen, direkt implementiert und ihre *Invarianten* sichergestellt [10, S. 208f.]. Dadurch wird eine hohe Kohäsion erzeugt und entsprechend des *Information-Expert-Prinzips* korrekt verankert.

Beispiel: Ein *Kunde* innerhalb eines Domainmodells ist meist eine Entity. 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, ist ihre Identität weiterhin äquivalent zur vorhergehenden. Invarianten, wie die korrekte Formatierung der hinterlegten E-Mail, können beispielsweise direkt bei der Aktualisierung überprüft werden.

2.3.6 Aggregate

Innerhalb des Bounded-Contexts ist ein Aggregate der Verbund aus Entities und Value Objects, welcher von außen als eine Einheit wahrgenommen wird. Hierbei findet die Gruppierung anhand ihrer logischen Zusammengehörigkeit und Verantwortungen statt. Externe Komponenten dürfen beim Aufruf eines Aggregates nur auf das sogenannte Aggregate Root zugreifen und enthaltene Objekte nicht direkt referenzieren. Das Aggregate Root ist demzufolge eine Schnittstelle zwischen dem Aggregate und der Außenwelt. [24, S. 126f.]

Beispiel: Ein mögliches Aggregate im Bereich des Personalmanagements ist ein *Mitarbeiter*, welches Value Objects, wie *Gehalt* und *Abteilung*, beinhaltet. Die Klasse *Mitarbeiter* ist auch zugleich ihr eigenes Aggregate Root. Bei Gehaltsanpassungen wird eine Funktion der Mitarbeiter-Klasse aufgerufen, welche das neue Gehalt durch Austausch des Value Objects einträgt. In diesem Schritt können Invarianten überprüft werden, sodass beispielsweise ein neues Gehalt nicht negativ oder niedriger als das vorgehende ausfallen darf. Abhängig vom jeweiligen Bounded-Context ist der Werteverlauf des Gehaltes eventuell relevant, weshalb die Klasse stattdessen als eine Entity realisiert werden sollte.

Um die Effektivität von Aggregates zu gewährleisten, wurden in Domain-Driven Design einige Einschränkungen und Richtlinien beschlossen. Businessanforderungen bzw. Invarianten von enthaltenen Objekten müssen stets vor und nach einer Transaktion erfüllt sein. Dadurch sind die Grenzen der Aggregates durch den minimalen Umfang der transaktionalen Konsistenz ihrer Komponenten gesetzt [10, S. 354]. Als Folge dessen, wird immer das komplette Aggregate aus der Datenbank geladen und zurückgeschrieben. Große Aggregates leiden aus diesem Grund an reduzierter Skalierbarkeit und Performance, da die Datenmenge und notwendigen Operationen auf Seiten der Datenbank Last verursacht [10, S. 355]. Weiterhin sollte pro Transaktion jeweils nur ein Aggregate bearbeitet werden [10, S. 354]. Dies schränkt umfangreiche Aggregates durch fehlende Parallelität weiter ein. Unter Beachtung der letzten Regel wäre es nicht möglich das Gehalt und die Abteilung der Mitarbeiter-Klasse durch zwei unterschiedliche Personalmitarbeiter zeitgleich anzupassen. Eine der beiden Transaktionen würde sonst auf einen veralteten Stand operieren und müsste zurückgerollt werden. Sollte ein Anwendungsfall die Anpassung zweier Aggregates benötigt, kann das Konzept der Eventuellen Konsistenz angewandt werden. Dabei 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 damit eine mögliche Alternative für die Zusammenführung der beiden Aggregates. [10, S. 364]

2.3.7 Applicationservice

Aufgaben, welche kein Domainwissen erfordern, werden in den Applicationservices realisiert. Ihre Aufgabe ist die Bereitstellung von notwendigen Dienstleistungen zur Abwicklung der Businesslogik [49]. Dazu gehört das Management von Transaktionen, simple Ablaufsteuerung und Aufrufe anderer Services oder Aggregate Roots. Somit dürfen sie keine Businessanforderungen enthalten oder Invarianten überprüfen, da dies in den Zuständigkeitsbereich der Domainservices fällt [10, S. 267]. Die Namensgebungen der Klassen und ihrer Funktionen stammen meist aus Begriffen der Ubiquitous Language [24, S. 105]. Um Nebeneffekte ausschließen zu können und Parallelität zu ermöglichen, müssen die Applicationservices zustandslos designt werden [24, S. 105].

2.3.8 Domainservice

Soweit anwendbar, werden Businessanforderungen direkt in den zuständigen Entities oder Value Objects umgesetzt. 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. Sollte die auszuführende Logik Abhängigkeiten zu anderen Services besitzen oder die Kohäsion der Entity bzw. des Value Object verringert werden, ist dies ein weiterer Grund für die Nutzung eines Domainservices. Analog zu den Applicationservices werden sie zustandslos implementiert und stammen aus der Ubiquitous Language. Sie unterscheiden sich lediglich darin, dass es für Domainservices erlaubt ist, Businesslogik umzusetzen und Invarianten zu beinhalten. [10, S. 268]

2.3.9 Factory

Die wiederholte Erstellung von komplexen Objekten kann unnötigen Platz im Quelltext einnehmen und die Übersichtlichkeit einschränken, vor allem wenn zusätzliche Services zu diesem Zweck benötigt werden. Der Effekt wird verstärkt, wenn das Codefragment an verschiedenen Stellen auftritt. Zur Auslagerung der Objekterzeugung sind sogenannte Factories vorgesehen. Sie nehmen alle nötigen Daten entgegen und geben das gefragte Objekt zurück. Dadurch wird auch die Kohäsion der Applikation gestärkt. [24, S. 137f.]

2.3.10 Repository

Eine Grundfunktion von Applikationen ist das Speichern und Laden von Daten. Repositories ermöglichen und orchestrieren hierbei den Datenbankzugriff [24, S. 151]. In Domain-Driven Design benötigt jedes Aggregate ihr eigenes Repository, da sie unabhängig voneinander geladen werden müssen [10, S. 401]. Durch diese Zuordnung der Zuständigkeiten werden die konzeptionellen Abhängigkeiten der Domain von den Datenbanken getrennt. Die Kommunikation mit einem Repository sollte über ein fest definiertes Interface geschehen, damit bei Änderungen der darunterliegenden Datenbanktechnologie der Domainkern unbetroffen bleibt [24, S. 152].

Die erarbeiteten Grundgedanken von Domain-Driven Design, Hexagonaler Architektur und der SOLID-Prinzipien bilden den Rahmen für die Durchführung des Projekts. Im Folgenden wird diese Basis in der Planungsphase erweitert und die Checkout-Domain erschlossen.

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 kann das Domain-Modell und die Ubiquitous Language vollständig definiert werden. Besonders ist der Aggregationsschnitt von den Anwendungsfällen abhängig [10, S. 358]. Aus diesen Gründen wird im folgenden Kapitel der Bounded-Context umfassend untersucht. Hierbei ist die Funktionalität eines Warenkorbs und des zugehörigen Datenmodells in dem Term 'Basket' zusammengefasst, um eine einheitliche Benamung innerhalb der Ubiquitous Language einzuhalten.

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 Baskets bis hin zum Kaufabschluss, verwalten können. Zu Dokumentationszwecken ist es empfehlenswert die Anwendungsfälle in Diagrammen abzubilden. Zusätzlich werden dadurch eventuelle Unklarheiten geklärt, Bedingungen an die Datenstrukturen oder den Programmfluss aufgedeckt und eine natürliche Benutzung der Ubiquitous Language etabliert. Auf Basis dieser Anwendungsfälle ist es außerdem 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. Bei der Reduzierung der Prozesse ist zu beachten, dass Bedingungen zwischen Datenstrukturen weiterhin unverändert bleiben, damit die möglichen Aggregationsschnitte nicht von den realisierbaren Varianten der Produktivanwendung abweichen. Dieser Abschnitt dient somit als Grundlage für das Design der Software und wird in späteren Kapiteln referenziert. Die folgenden Prozesse wurden in Zusammenarbeit mit den Teams des Onlineshops, dem Checkout-Team und zuständigen Stakeholdern erarbeitet. Als Darstellungsmethode wurde sich auf Aktivitätsdiagramme geeinigt, damit die Interaktionen zwischen den Systemen ebenfalls aufgezeigt werden können. Die Clients der Checkout-Software, wie der Onlineshop oder die Handy-App, sind unter dem Begriff 'Touchpoint' zusammengefasst. Falls die Graphen keine Relevanz für das leserliche Verständnis der Arbeit besitzen, sind sie dem Anhang hinzugefügt worden.

3.1.1 Erstellung eines neuen, leeren Baskets

Die Erzeugung eines Baskets kann aus verschiedenen Gründen geschehen, beispielsweise wenn ein nicht eingeloggter Kunde zum ersten Mal den Onlineshop aufruft oder ein Mitarbeiter an einer physikalischen Kasse im Markt einen Kauf abschließt. Der Erstellungsprozess besteht hauptsächlich aus dem Empfangen der Kundendaten, die Identifikation des zugehörigen Marktes mithilfe der Outlet-ID und dem permanenten Speichern des neuen Baskets. Es sind keine Invarianten zu prüfen, außer dem korrekten Format der übertragenen Nutzdaten. Dem Touchpoint wird das komplette Basket-Objekt zurückgegeben inklusive einer Basket-ID zur Referenz für spätere Zugriffe. Dieser Prozess ist in Anhang B verdeutlicht.

3.1.2 Abruf eines Baskets anhand der Basket-ID

Anhand einer ID kann ein Basket jederzeit durch den Touchpoint abgerufen werden. Die wichtigste Bedingung für diesen Anwendungsfall besagt, dass angefragte Ressourcen stets mit aktuellen Daten befüllt sein müssen. 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 in einem Cache zwischengespeichert. Dadurch können Aufrufe externer Systeme eingespart werden. Die genaue Speicherdauer wurde durch das verantwortliche Team festgelegt. Dies bedeutet jedoch, dass das Alter der Informationen nachvollziehbar sein muss, weshalb der Zeitstempel im Datenmodell hinterlegt wird. Auf Authentifizierung und Autorisierung wurde in dem Proof-of-Concept und den Diagrammen verzichtet, da es sich um eine rein technische Funktion handelt und keine Relevanz für die Projektumsetzung besitzt. Es verbleiben Aufgaben, wie die Suche des Baskets innerhalb der Datenbank, die De- und Serialisierung der Objekte und das Zurückgeben von Fehlermeldungen, falls beispielsweise Ressourcen nicht gefunden werden konnten. Abbildung 3.1 zeigt das zugehörige Aktivitätsdiagramm.

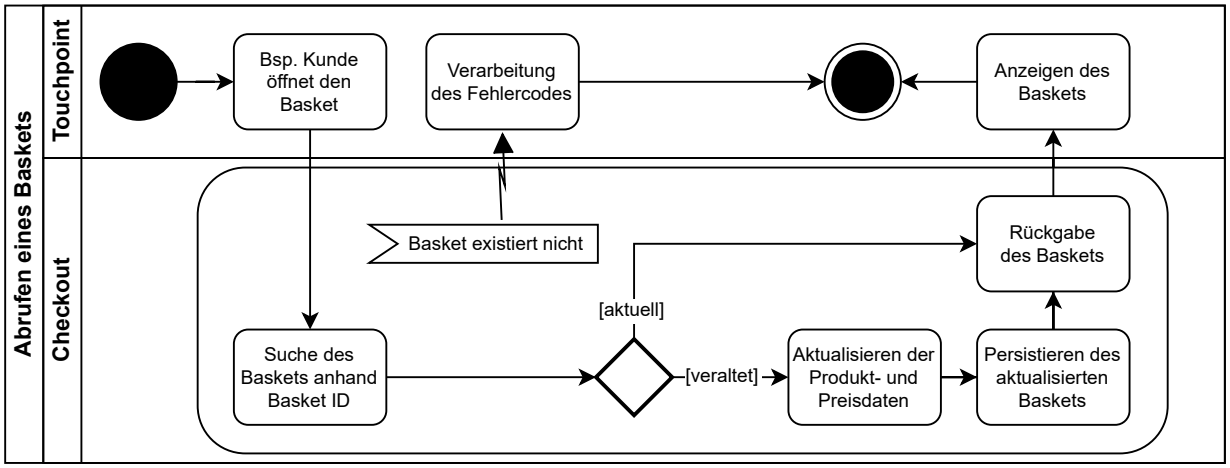


Abbildung 3.1: Aktivitätsdiagramm für den Abruf eines Baskets

3.1.3 Stornierung eines offenen Baskets

Der Basket kann sich in verschiedenen Zuständen befinden. Es wird unterschieden zwischen 'Open', 'Frozen', 'Finalized' und 'Canceled'. Sollte beispielsweise ein Kunde beim Bezahlen an der Kasse im Markt nicht ausreichend Geld bei sich haben, muss der Basket als 'Canceled' vermerkt werden. Für diesen Prozess ist es notwendig, vorher zu prüfen, ob der Zustand sich in 'Open' befindet, da ein eingefrorener, abgeschlossener oder bereits stornierter Basket zur Wahrung des Zustandsverlaufes nicht storniert werden kann. Dies stellt eine Invariante dar, welche in der Software sichergestellt werden muss. Im Diagramm in Anhang C ist der Vorgang zusammenfassend abgebildet. Die tatsächliche Löschung eines Baskets ist aus rechtlichen Gründen strengstens untersagt.

3.1.4 Hinzufügen eines Produktes anhand einer Produkt-ID

Der Basket fungiert ebenfalls als ein Speicher einer Liste von Artikeln, sowie deren Quantität, Produktbeschreibung und ausgewählten Services bzw. Garantien. Der aufwändigste und deswegen in Abbildung 3.2 dargestellte Prozess ist hierbei das Hinzufügen eines neuen Produktes zum Basket. Dabei sendet der Client lediglich die zugehörige Produkt-ID, weswegen externe Systeme 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 wird eine weitere API-Abfrage benötigt, welche zu einer Produkt-ID und Outlet-ID den zugehörigen, aktuellen Preis zurückschickt. Diese zwei Schnittstellen müssen ebenfalls in anderen Anwendungsfällen aufgerufen werden, sofern die zwischengespeicherten Preis- bzw. Artikelinformationen im Cache nicht mehr gültig sind. Zusätzlich folgt eine Validierung des Datenmodells auf verschiedene Parameter. Unter anderem darf die Gesamtanzahl der enthaltenen Artikel keinen festen Wert überschreiten. Der aktualisierte Basket wird beim erfolgreichen Abschluss der Operation dem Touchpoint zurückgegeben.

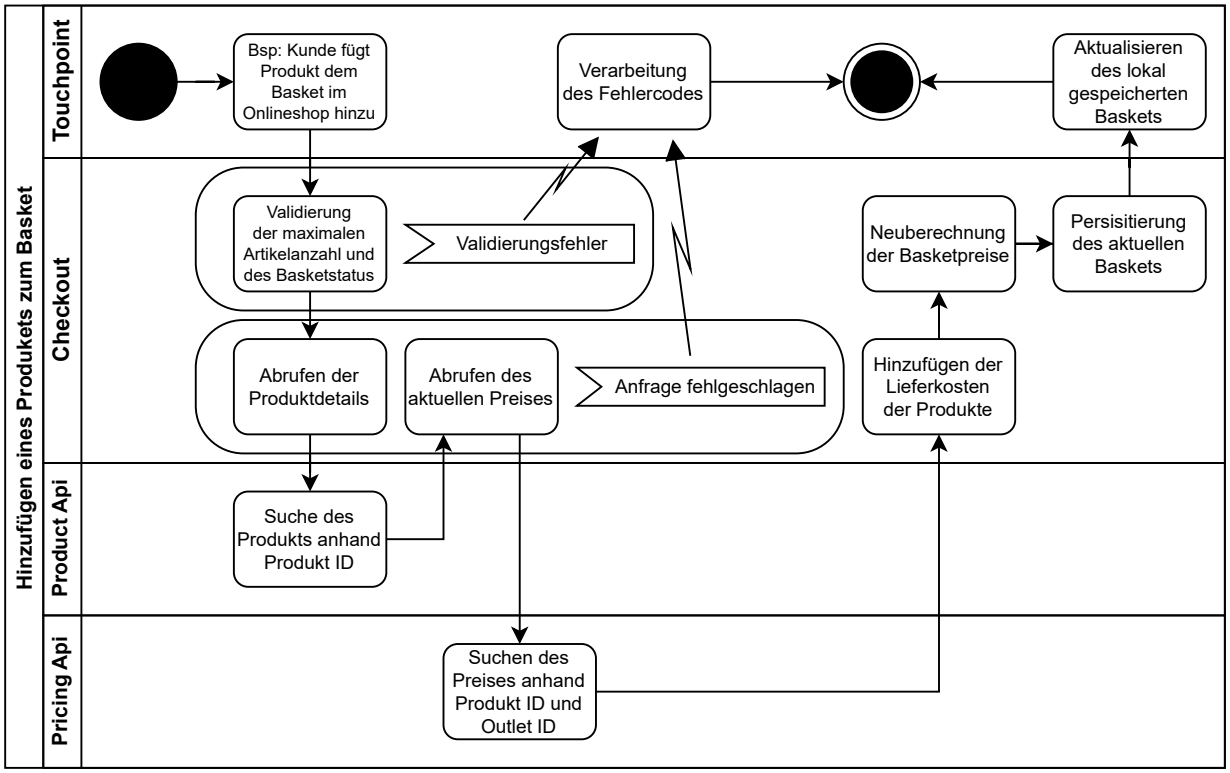


Abbildung 3.2: Aktivitätsdiagramm für das Hinzufügen eines Produktes

3.1.5 Hinzufügen einer Zahlungsmethode

Eine weitere essentielle Funktion ist das Hinzufügen von Zahlungsmethoden, damit der Kauf erfolgreich initiiert werden kann. Da es sich nur um das ungeprüfte Anhängen der Bezahlinformationen handelt, müssen keine strengen Validierungen vorgenommen werden, weil diese Aufgabe durch ein externes System in einem späteren Schritt des Checkouts erledigt wird. Dennoch sind logische Überprüfungen durchzuführen, wie zum Beispiel, dass der Basket nicht leer oder bereits bezahlt ist. Ebenfalls findet eine Neuberechnung aller Bezahlinformationen statt. Analog zu den vorgehenden Fällen wurde ein Aktivitätsdiagramm in Anhang E designt.

3.1.6 Aktualisieren der Checkout Daten des Baskets

Ein Basket 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 Informationen, wie Kundendaten, Zahlungsmethode und Zustellungsart, von den Touchpoints an die API gesendet. Da die Daten innerhalb eines einzelnen Verfahrens durch das vorgelagerte System gesammelt werden, bietet es sich an, die korrespondierende Schnittstelle so zu designen, dass alle betroffenen Attribute gleichzeitig angepasst werden können. Eine Überprüfung der Daten erfolgt in diesem Schritt nicht, mit der Ausnahme, dass die gewählte Zustellungsart, in dem Bounded-Context 'Fulfillment' genannt, für den ausgewählten Basket verfügbar sein muss. Zudem ist es notwendig eine Neuberechnung der Zahlungsmethoden durchzuführen, falls eine neue Zahlart hinterlegt worden ist. Diese Bedingungen und der dazugehörige Prozess sind in Anhang D veranschaulicht worden.

3.1.7 Initiierung des Bezahlprozesses und Einfrieren des Baskets

Nachdem die Bearbeitung des Baskets abgeschlossen ist, kann der Bezahlprozess gestartet werden. Hierbei muss das Datenmodell 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 Bezahlprozess wird von einer externen Software abgewickelt und durch eine Referenz im Basket verlinkt. Eine vereinfachte Darstellung des Prozesses bietet Abbildung 3.3.

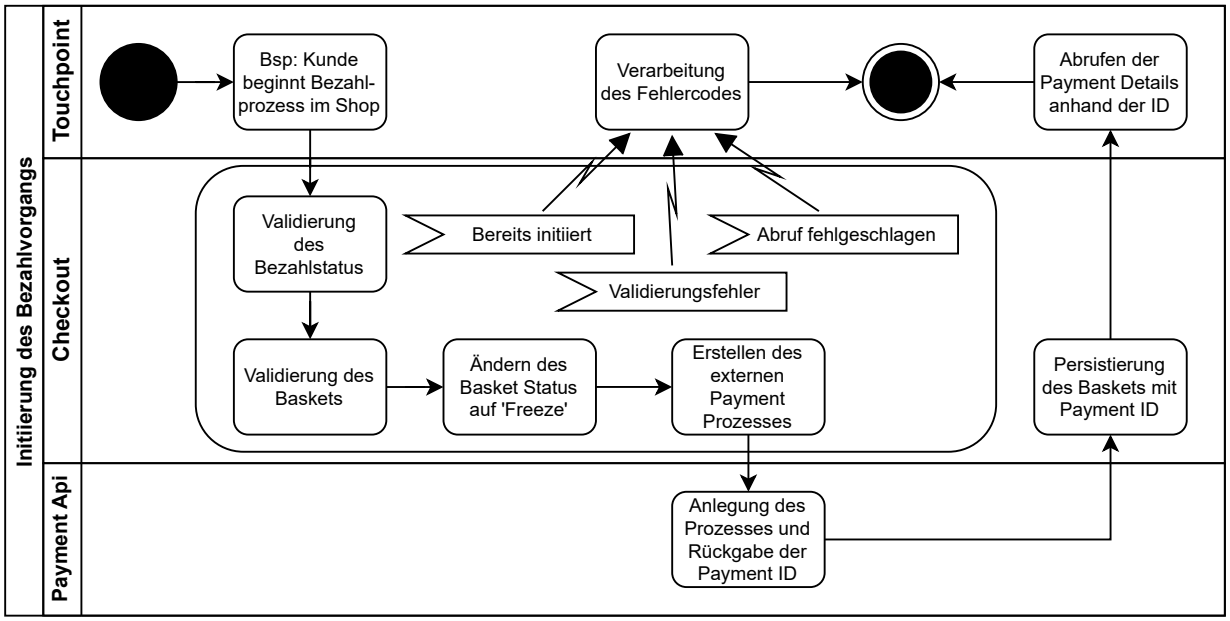


Abbildung 3.3: Aktivitätsdiagramm für das Initiieren des Bezahlvorgangs

3.1.8 Ausführung des Bezahlprozesses und Finalisierung des Baskets

Als letzter, höchst relevanter Anwendungsfall gilt der Abschluss des Bezahlvorgangs, visualisiert in Abbildung 3.4. Die Checkout-Software dient hierbei als Proxy zwischen den Touchpoints und der 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.

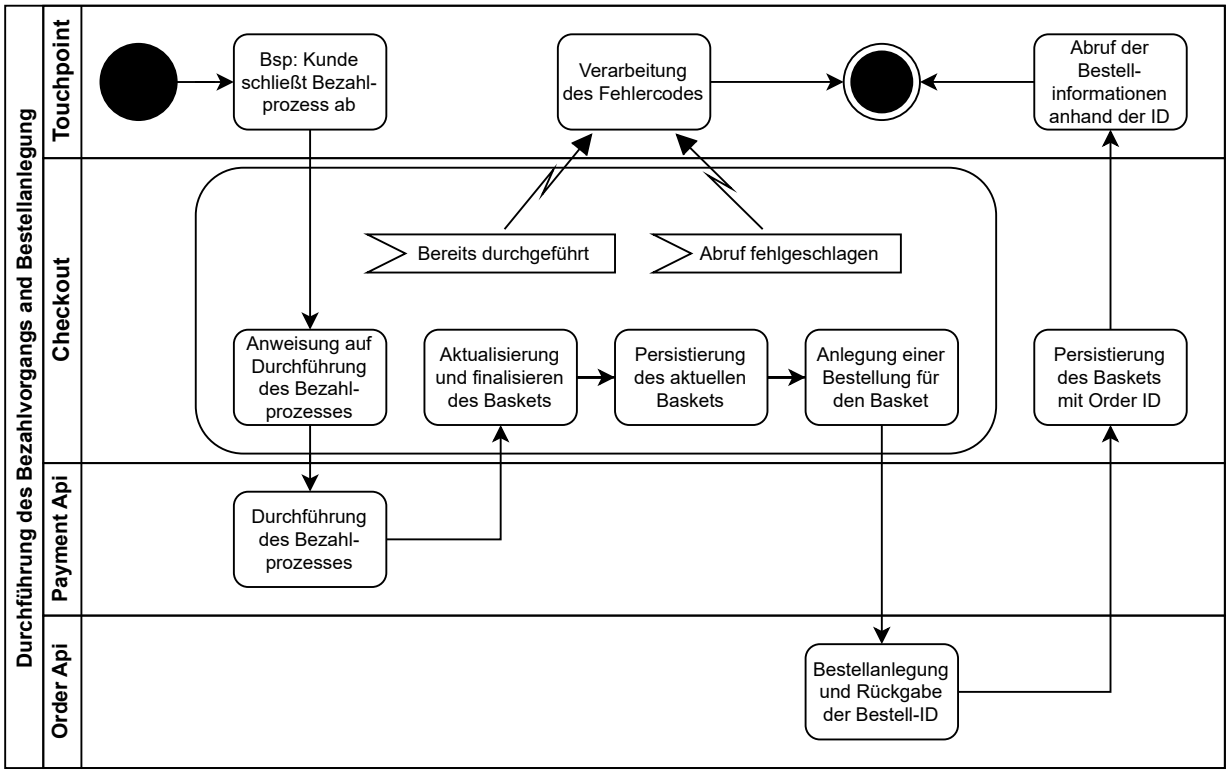


Abbildung 3.4: Aktivitätsdiagramm für das Ausführen des Bezahlvorgangs

3.1.9 Aus Anwendungsfällen resultierende API-Schnittstellen

Anhand der Anwendungsfälle ist es möglich eine klare Schnittstellendefinition für die Checkout-Applikation zu erstellen. Hierbei beinhaltet diese alle benötigten Operationen zum erfolgreichen Bewältigen der Anforderungen aus Sicht der Touchpoints. Die Kommunikation zwischen den Systemen geschieht über eine REST-API und somit auf Basis des HTTP-Protokolls. In Anhang F sind alle Endpunkte enthalten, mitsamt ihrer HTTP-Methoden, Parametern und gesendeten bzw. empfangenen Datenstrukturen.

3.2 Projektumfeld und technologische Vorschläge

Die komplette Systemumgebung von MediaMarktSaturn ist eine komplexe Architektur 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 das 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, welche die benötigten Informationen, mögliche Fehlerfälle und die zurückgelieferten Daten festlegt. Der Proof-of-Concept orientiert sich an diesen Vereinbarungen, erleichtert allerdings die Kommunikationsbedingungen, um unnötigen *Boilerplate-Code* zu unterdrücken. Die Abbildung 3.5 stellt ein Context-Diagramm der Umgebung dar.

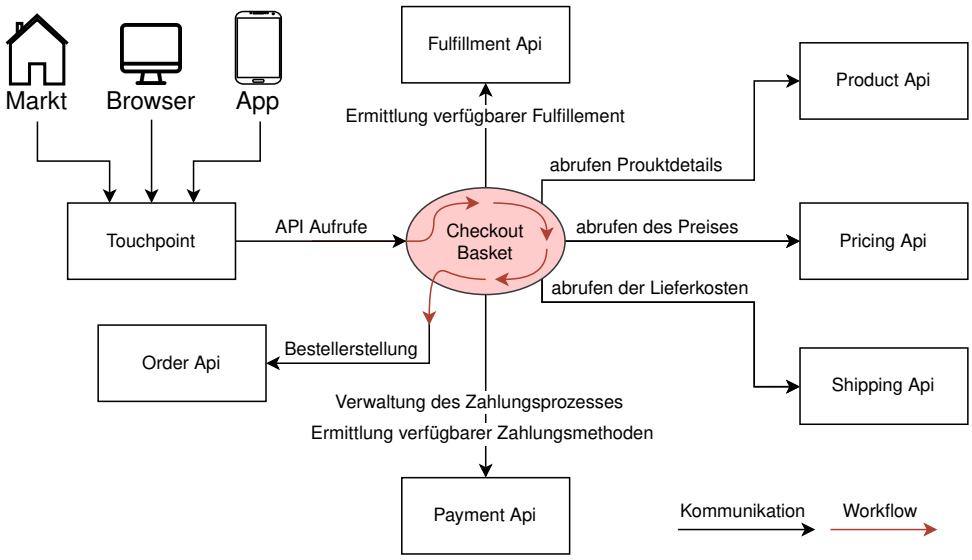


Abbildung 3.5: Context Diagramm der produktiven Checkout-Umgebung

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 bei 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 diesem Grund wird ebenfalls der Proof-of-Concept in Kotlin umgesetzt. Zudem ist die Technologie für eine systemübergreifende 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. In einem späteren Kapitel wird die Verwendung einer MongoDB für das Projekt begründet.

4 Festlegung des Datenmodells durch Domain-Driven Design

Durch die Schaffung eines grundlegenden Verständnisses für Designprinzipien, Hexagonaler Architektur und Domain-Driven Design kann auf zusätzlicher Basis der vorherigen Analysen ein Proof-of-Concept der Checkout-Software entwickelt werden. Hierzu wird weiterhin das typische Vorgehen eines Domain-Driven Designs 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 3.5 hierbei unsere Domaingrenzen. Eine Domain und die dazugehörigen Subdomains spannen den Problemraum über alle definierten Anwendungsfälle und Businessanforderungen auf [10, S. 56]. Die Größe der Domain ist entscheidend für die Bestimmung der Subdomains und des Bounded-Contexts. Wird der Checkout als eine Domain angesehen, ergibt sich insgesamt nur ein Bounded-Context, da grundlegend pro Bounded-Context nur maximal ein Team zuständig sein sollte [48]. Der Checkout müsste somit weitere Unterteilungen erfahren oder alternativ die Domain vergrößert werden. Folglich wird als nächstmögliche Eingrenzung der Checkout und alle abhängigen Systeme angesehen. Zu beachten ist hierbei, sich nicht auf die konkreten Systeme zu fixieren, da sie eher der Lösungsebene zuweisbar sind, sondern logisch naheliegende Aufgaben in einer 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 bestehende Architektur vorgegeben, wodurch das Context-Diagramm 3.5 zugleich als Context-Map fungieren kann.

4.2 Festlegen einer Ubiquitous Language

In der Kommunikation zwischen dem Business und Entwicklern kann es oft zu Missverständnissen 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 Anwendungsfälle verwendet 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, ist die Ubiquitous Language ein wichtiger Meilenstein im Domain-Driven Design. [24, S. 335ff.]

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

Ubiquitous Language des Domain-Modells:

- **Basket:** Repräsentiert die Funktionalität eines Warenkorbs mit allen Artikeln, 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. Einem nicht-eingeloggten Kunden wird der zugehörige Basket im Onlineshop 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, welcher durch eine einzigartige Outlet-Nummer referenziert wird.
- **BasketItemID:** Eine, innerhalb eines Baskets, eindeutige Referenz auf einen enthaltenen Artikel. Wird aus technischen Gründen benötigt, um Einträge zu bearbeiten oder entfernen.
- **Net Amount (Nettobetrag):** Ein Nettobetrag mit Währung.
- **VAT (Steuersatz):** Der Steuersatz eines zugehörigen Nettobetrags.
- **Gross Amount (Bruttobetrag):** Der Bruttobetrag eines Preises errechnet aus dem Steuersatz und Nettobetrag. Die Währung gleicht der 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 (Zahlungsvorgang):** Beinhaltet alle relevanten Informationen für das Verwalten eines Zahlungsvorgangs, wie Beträge und getätigte Zahlungen.
- **Payment:** Eine einzelne Zahlung des Kunden inklusive Betrag und Zahlungsmethode, wie Barzahlung oder PayPal.
- **Order:** Bestellung eines Baskets nach Abschluss des Zahlungsvorgangs. Wird durch nachfolgende Systeme angelegt und verwaltet.

Ubiquitous Language der Businessprozesse:

- **Touchpoint:** Eine Komponente, welche mit der Checkout-Software interagiert, wie Kassensysteme im Markt, die Onlineshop-Seite oder Handy-App.
- **Basket Cancellation:** Stornieren des zugehörigen Baskets mithilfe eines Zustandswechsels auf 'Canceled'. Nach der Cancellation dürfen keine weiteren Änderungen an dem Basket durchgeführt werden. Der Zustand muss zuvor 'Open' sein.
- **Basket Creation:** Explizite oder Implizite Erstellung eines neuen Baskets. Geschieht automatisch sofern noch kein Basket für den Customer existiert oder nach einer Basket Finalization.
- **Basket Calculation:** Die Kalkulation von Bruttobeträgen aller Artikel sowie der Summe von beinhalteten Preisen des Baskets. Beträge aus unterschiedlichen Mehrwertsteuersätzen 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 innerhalb eines Baskets oder in mehreren Bestellungen das gleiche Produkt mehrfach kauft.
- **Basket Validation:** Durchführung einer Validierung des Baskets auf Inkonsistenzen oder fehlenden, jedoch notwendigen, Werten.
- **Payment Initialization:** Start des Zahlungsvorgangs, nachdem das Datenmodell auf invalide Zustände überprüft worden ist. Nur möglich bei einem offenen Basket, welcher Produkte und Payments enthält. Resultiert in den Zustand 'Frozen', wodurch keine weiteren Inhaltsänderungen an dem Basket vorgenommen werden können.
- **Payment Execution:** Durchführung des Zahlungsvorgangs eines gefrorenen Baskets. Anschließend findet die Basket Finalization statt.
- **Basket Finalization:** Erfolgt automatisch nach erfolgreichem Zahlungsvorgang und setzt den Basket in den Zustand 'Finalized'. Danach folgt die Reservierung der Produkte und das Anlegen einer neuen Bestellung.

Im Verlaufe der Definitionsphase der Ubiquitous Language wurden die Prozesse näher beleuchtet, Benamungen von Datenobjekten aufgedeckt und Businessanforderungen vorgegeben. Ein gutes Modell spiegelt die Sprache des Bounded-Contexts wider, weshalb auf Basis dieses Unterkapitels die Klassen designt werden.

4.3 Definition der Value Objects

Aufgrund der positiven Eigenschaften von Value Objects sollte anfangs jede Datenstruktur des Domain-Modells als ein solches implementiert und erst nach gründlicher Überlegung, falls die Notwendigkeit besteht, zu einer Entity umgeschrieben werden [10, S. 219f.]. Der Basket ist hierbei der Ausgangspunkt des Modells. Es wird auf ein schlankes Design im Vergleich zur Produktivanwendung geachtet, ohne dabei mögliche Aggregationsschnitte zu beeinflussen. Die Ubiquitous Language unterstützt bei der richtigen Klassen-Benamung.

Basket:

- **BasketId:** Eindeutige Identifikation des Baskets zur Referenzierung durch die Touchpoints.
- **OutletId:** Eine Referenz zugehörig zu einem Markt oder Onlineshop, durch welchen der Basket angelegt wurde. Unerlässlich für die Bestimmung von unter anderem Lagerbeständen, Lieferzeiten, Fulfillment-Optionen und Versandkosten.
- **BasketStatus:** Repräsentiert den aktuellen Zustand des Baskets. 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 Produkte und ihre zugehörigen Informationen.
- **BasketCalculationResult:** Beinhaltet die berechneten Werte des Basket, wie Nettobetrag, Bruttobetrag und Mehrwertsteuer. Die Speicherung dieser Werte wäre technisch nicht notwendig, spart aber Rechenzeit, da nicht bei jeder Abfrage des Basket dieser Wert neu berechnet werden muss.
- **PaymentProcess:** Enthält alle Informationen zur erfolgreichen Abwicklung des Zahlungsprozesses.
- **Order:** Speichert eine Referenz auf die Bestellung eines Baskets. Wird erst nach Zahlungsabschluss befüllt.

Durch diese Datenstruktur ist es möglich, alle geforderten Anwendungsfälle korrekt abzuarbeiten. Die untergegliederten Klassen sind ebenfalls mit der gleichen Vorgehensweise in Anhang 8.4 definiert worden, sofern sie nicht durch einen einfachen Text oder Aufzählungen realisierbar sind. Um eine klare Gesamtübersicht zu bieten, wurde ein Klassendiagramm der Datenstruktur dem Anhang G hinzugefügt. Die Klassen *Customer* und *PaymentProcess* wurden aus Platzgründen in separate Klassendiagramme in Anhang H und I verlagert. Anzumerken ist, dass bei fehlender Multiplizität eine Eins-zu-Eins Beziehung vorliegt. Speziell, ist der Kunde in diesem Kontext genau einem Basket zugewiesen, da alleinig die Daten abgespeichert werden, nicht aber seine Kundennummer, wodurch keine Zuweisung zwischen einem Kunden und mehreren Warenkörben existiert.

4.4 Bestimmung der Entities anhand ihrer Identität und Lebenszyklen

Auf Basis der vorangehenden 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 Datengruppierungen je nach Bounded-Context unterschiedliche Eigenschaften besitzen. Als Hilfestellung für diese Entscheidung können grundlegende Richtlinien aus Tabelle 4.1 verwendet werden.

	Value Object	**Entity**
Identität	Summe aller Attribute des Objekts. Objekte mit gleichen Werten besitzen gleiche Identität. [24, S. 99]	Bestimmt anhand eines Identifikators, zum Beispiel einer Datenbank-ID. Objekte gelten als ungleich, außer ihre Identifikatoren sind identisch. [24, S. 92]
Lebenszyklus	Stellt nur eine Momentaufnahme des Applikationszustands dar, da sie bei Änderungen ersetzt werden. [10, S. 226]	Werden zu einem bestimmten Zeitpunkt erstellt, bearbeitet, gespeichert oder gelöscht. Besitzen somit einen impliziten Verlauf ihrer Wertänderungen. [24, S. 91]
Veränderbarkeit	Durch einen fehlenden Lebenszyklus gelten Value Objects als immutable. [24, S. 99]	Aufgrund ihrer Eigenschaften sind Entities veränderbar. [24, S. 91]
Abhängigkeit	Nur als Unterobjekt von Entities persistierbar, da sie kein Aggregate Root sein können.	Damit ein eigener Lebenszyklus ermöglicht wird, können sie unabhängig von anderen Objekten existieren.
Zugriffsmethode	Auf Daten und Funktionen wird mithilfe einer Entität zugegriffen.	Können als Aggregate Root, oder durch dieses, direkten Zugriff erfahren. [24, S. 129]

Tabelle 4.1: 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 dessen 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: Werden innerhalb eines Bounded-Contexts die Kundendaten verarbeitet, stellen diese meinst eine Entity dar. In der Checkout-Domain finden keine Operationen auf den Informationen statt. Die vorangestellten Systeme senden bei Änderungen die aktualisierten Kundendaten an die Checkout-Software. Folglich besitzen sie keinen eigenen Lebenszyklus und können als *Value Object* designt werden.

SessionCustomer: Die Identifikation dieses Objekts geschieht über die Session-ID. Dadurch ist ein SessionCustomer in der Gruppe der *Entities* aufzuhängen.

Basket-Item: Auf den ersten Blick ist ein Basket-Item als Entity zu designen. Es besitzt eine eigene ID und wird durch das Aktualisieren der Preise und Produktdaten bearbeitet. Sie haben somit einen Lebenszyklus. Jedoch lassen sich auch Argumente finden, warum ein Item 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 im Datenmodell auch mehrere zugehörige Basket-Items. Bei der Reduzierung der Stückzahl eines Artikels beispielsweise von vier auf eins, werden alle Items gesucht, welche das gleiche Produkt repräsentieren, und davon drei gelöscht. Dies würde bedeuten, dass ein Basket-Item zusätzlich anhand seines Produktes identifiziert wird. Sollte die ProductID die Identität des Baskets ausmachen, dann wären alle Basket-Items mit dem gleichen Produkt auch identisch. Dies stimmt allerdings nur bedingt, da sie sich theoretisch durch unterschiedliche Preise und Serviceangebote (in der Produktivumgebung) differenzieren können. Als Folgerung kann geschlossen werden, dass ein Basket-Item lediglich eine Momentaufnahme darstellt, wodurch das Design als Value Object berechtigt wäre. Letztendlich kann das Basket-Item in diesem Bounded-Context als Entity oder Value Object definiert werden. Für den Proof-of-Concept wurde das Basket-Item als Entity festgelegt. Die Begründung hierfür ist die schiere Anzahl von Datenanpassungen und Operationen auf einem Basket-Item, welche als *Entity* anhand ihrer Veränderbarkeit natürlicher bewältigt werden können.

Product und Price: Die vorgehende Analyse des Basket-Items kann auch auf das Product und den Price angewendet werden. Beide besitzen eine ID und werden stetig aktualisiert. Dennoch sind Prices bzw. Products mit ungleichen Werten aber gleicher ID in dem Checkout-Kontext unterschiedliche Objekte. Beide Klassen sind als *Value Object* umgesetzt worden, da der Zusammenschluss aller ihrer Attribute als Identifikationsmerkmal verwendet wird.

Calculation-Result: Als Datenstruktur, welche bei jeder Neuberechnung aktualisiert wird, könnte die Eigenschaft eines Lebenszyklus erfüllt sein. Allerdings ist die Klasse einzig ein Zwischenspeicher der Ergebnisse zur Performance-Verbesserung. Ohne den Kontext eines darüberlegenden, zugehörigen Objektes besitzen diese Daten keine Aussagekraft. Die gleichen Berechnungsergebnisse unterschiedlicher Baskets sind im Sinne der Identität äquivalent. Dadurch überwiegen die Argumente eines *Value Objects*.

Payment-Process: Der Payment-Process besitzt zur Ablaufsteuerung einen eigenen Status, weshalb ein Lebenszyklus entsteht. 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 Payment-Process alle darunterliegenden Payments. Zusammenfassend sprechen die Eigenschaften für ein Design als *Entity*.

Payment: Ein Payment hat eine eindeutige ID, welche für den Ablauf des Bezahlprozesses und alle folgenden rechtlichen Prozesse eine hohe Relevanz hat. Dadurch ist weder der konkrete Betrag, noch die Bezahlmethode bei der Identifikation wichtig. Ähnlich zum Payment-Process ist auch hier ein Lebenszyklus in Form eines Statusfeldes vorhanden. Eine Umsetzung als *Entity* ist zu empfehlen.

5 Entwerfen möglicher Aggregate-Designs

Ein Domainmodell kann auf unterschiedliche Weise realisiert werden. Dies gilt ebenfalls für den Schnitt der Aggregates. Verschiedene Gruppierungen genießen zugleich andere Vor- und Nachteile. In diesem Kapitel werden mehrere dieser Aufteilungen 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 Aggregates 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 umspannt. Die erste Variante des Proof-of-Concepts wurde mit diesem Design in Gedanken entwickelt und stellt das Grundgerüst für alle kommenden Ansätze dar. Dementsprechend 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 umfassenderen Aggregates muss immer die gesamte Datenstruktur geladen werden, da auf darunterliegende Objekte nur durch das Aggregate Root zugegriffen werden darf [24, S. 128]. Die Auswirkungen dieser Tatsache kann mithilfe von *Lazy Loading* oder durch Einsatz einer dokumentenorientierten Datenbank eingeschränkt werden. Ein kleinerer Aggregationsschnitt ermöglicht das unabhängige Laden und Bearbeiten der Aggregates. Generell gilt, dass bei Verwendung von größeren Aggregates ein verringerter Datendurchsatz zu erwarten ist als bei kleineren [10, S. 355f.]. Diese Richtlinie wird anhand des vorliegenden Bounded-Contexts analysiert, denn je nach Anwendungsgebiet und Implementierung kann diese Aussage sogar gegensätzlich ausfallen.

Um eine Basis für die Argumentation zu formen, wird ein beispielhafter untergliederter Aggregationsschnitt gebildet. In Betrachtung der Anwendungsfälle beziehen sich Aktionen meist auf einzelne Value Objects, Basket-Items oder den Payment-Process. Dementsprechend wäre das explizite Abfragen dieser Daten aus einer Datenquelle effektiver. Wird die *Variante A* unterteilt in ein Basket-, Payment und Basket-Item-Aggregate (*'Variante B'*) können sie unabhängig voneinander gehandhabt werden. Eine kurze Übersicht über den neuen Aggregationsschnitt bietet Abbildung 5.1.

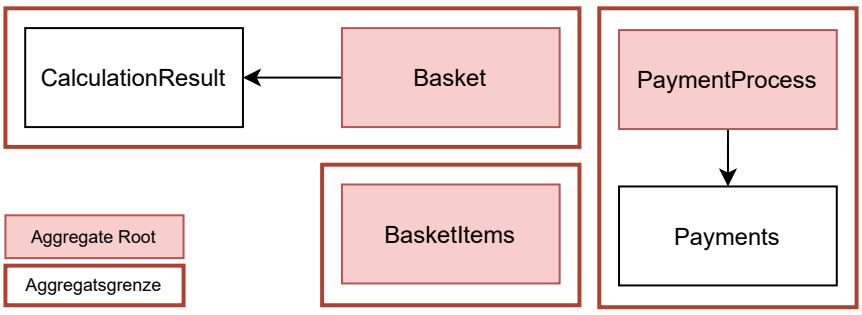


Abbildung 5.1: Aggregationsschnitt der Variante B

Ohne Beachtung, welche Auswirkungen das neue Design auf andere Faktoren hat, ist eine erhöhte parallele Bearbeitbarkeit zu erwarten. Die vermeintlich gewonnene Leistungsverbesserung wird allerdings aufgrund folgender Umstände minimiert.

Invarianten über Aggregationsgrenzen hinweg

Innerhalb eines Checkout-Kontextes existieren viele klassenübergreifende Businessanforderungen, weshalb die Kopplung des Datenmodells oftmals eine unabhängige Datenanpassung eines Aggregates in *Variante B* verhindert. Durch eine genauere Untersuchung der Anwendungsfälle kann eine Abwägung getroffen werden, wie viele Prozesse tatsächlich isoliert voneinander durchführbar sind. Das Stornieren des Baskets und Setzen der Checkout-Daten ist beispielsweise positiv von diesem Aggregationsschnitt betroffen, denn die benötigten Daten verwaltet alleinig der Basket. Die meisten API-Anfragen beziehen sich allerdings auf die Basket-Items und die Verwaltung des Bezahlvorgangs, welche Wechselwirkungen mit dem Basket oder seinem zugehörigen Gesamtpreis besitzen. Wird ein neuer Artikel hinzugefügt, hat dies nicht nur das Anlegen eines Items, sondern auch die Neukalkulation des Baskets zur Folge. Sofern eine Trennung zwischen Basket und Basket-Item vorliegt, müssen bei der Ermittlung des neuen Gesamtpreises somit dennoch alle Items aus der Datenbank ausgelesen werden. Wird bei einem API-Aufruf das vollständige Datenmodell als Antwort erwartet, muss auch hier ein aggregationsübergreifender Ladevorgang stattfinden. Letztendlich ist nur eine Bruchzahl der Anwendungsfälle mithilfe von Variante B unabhängig abschließbar, sodass pro Operation mehrere Aggregates bearbeitet werden müssten. Hierdurch steigt die Anzahl der pro Transaktion benötigten Datenbankoperationen und die damit verbundene Bearbeitungszeit auch bei einem kleineren Aggregationsschnitt an. In dem Buch '*Implementing domain-driven design*' wird zusätzlich auf eine Richtlinie hingewiesen, dass bei korrektem Aggregationsdesign eine Transaktion maximal ein Aggregate bearbeiten sollte [10, S. 354]. Das führt zu einem Konflikt mit den vorliegenden Businessanforderungen. Beispielsweise wäre es nicht möglich bei der Initiierung des Zahlungsvorgangs zeitgleich den Basket einzufrieren. Ein möglicher Lösungsansatz ist hierbei die Verwendung von Eventueller Konsistenz [10, S. 364]. Konkret findet die Bearbeitung der Aggregates zeitversetzt voneinander statt, weshalb eine Zeitspanne existiert in welcher der Datensatz einen invaliden Stand besitzt. 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, abgekapselt von verwendeten Technologien sein. Technologien entwickeln sich weiter, werden durch neuere 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. Deshalb wird im folgenden Abschnitt das Aggregationsdesign aus Sicht der Datenbank bewertet.

Laut Definition erhält jedes Aggregate bzw. Aggregate Root eine eigene Tabelle in der darunterliegenden, relationalen Datenbank. Deshalb existiert in *Variante B* mindestens eine Tabelle für jeweils den Basket, Basket-Items und Payment-Process. Hingegen kann bei *Variante A* der Payment-Process 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 Basket-Item. Dementsprechend müssen beim Abruf eines kompletten Baskets in *Variante A* im Vergleich zu *Variante B* weniger Ladevorgänge durchgeführt werden. Das kleinere 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 vorherige Aussage nur noch bedingt. Hierbei benötigt

Variante B pro Aggregate eine eigene *Collection*. *Variante A* kann das komplette Datenmodell hingegen in einem einzigen Eintrag persistieren. An sich ist der einzelne Datensatz umfangreicher, allerdings 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 der 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, halten sich die Performance-Unterschiede in Grenzen, wohingegen bei einem umfangreichen Aggregationsschnitt mit einer dokumentenorientierten Datenbank das nicht gewährleistet werden kann. Anhand eines konkreten Lasttests wird in einem späteren Kapitel dieser Effekt genauer untersucht und bewertet.

5.1.2 Parallele Bearbeitbarkeit von Aggregates

Ausgehend von *Variante A* resultiert eine Bearbeitung des Baskets in der Speicherung des kompletten Datenmodells in der Datenbank. Bei Schreibprozessen können Anomalien entstehen, 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 denselben Startzustand, jedoch beim Zurückschreiben ihrer Ergebnisse stößt der letztere von beiden auf einen nun neueren Stand. Hierbei muss die 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 gleichen Basket hinzufügen. So legt hierbei der erste Kunde ein neues Handy hinein, wohingegen zeitgleich ein anderer Nutzer einen Fernseher kaufen will. Beide Transaktionen starten mit einem leeren Basket 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 Basket initiiert worden ist, wird der Datensatz ebenfalls mit nur einem Artikel statt zwei persistiert. Letztendlich enthält der Basket nur einen Fernseher anstatt 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. Solche Lösungsansätze sind die optimistischen bzw. pessimistischen Sperrverfahren [10, S. 385f.].

Innerhalb eines großen Aggregates kann es bei zeitgleichen Aktionen vermehrt zu Schreibanomalien kommen, wodurch sie sich für parallele Bearbeitung nicht eignen [50, S. 2]. In *Variante B* ist es möglich Basket-Items 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 anwendbar. Deswegen muss vor dem Entwicklungsprozess bei der Aufnahme des Funktionsumfanges auf solche Anforderungen geachtet werden. Zum jetzigen Zeitpunkt ist eine parallele Bearbeitung eines Baskets nicht vorgesehen.

5.1.3 Bewertung des Baskets als ein großes Aggregate

- **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:**
  - Bei jeder Anfrage erfolgt 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. Eine Verkleinerung des Aggregationsschnittes kann hierbei zu Performance-Verbesserungen führen.
- **Parallelität:**
  - Eine zeitgleiche Bearbeitung des Baskets oder eines enthaltenden Objektes 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

Als Ausgangspunkt für die Bildung eines neuen Aggregationsschnittes wird die *Variante A* untersucht. Damit ein neues Aggregate aus der großen Basket-Klasse herausgeschnitten werden kann, wird ein Root Aggregate und somit eine Entity benötigt. Hierbei ist der *Payment-Process* nur schwach an den Basket gebunden und mögliche Anwendungsfälle beziehen sich meist alleinig auf entweder den Basket und seine Items oder den Zahlvorgang. Daher wird in der *Variante C* das Datenmodell in Basket und Payment-Process, entlang der Abbildung 5.2, aufgeteilt.

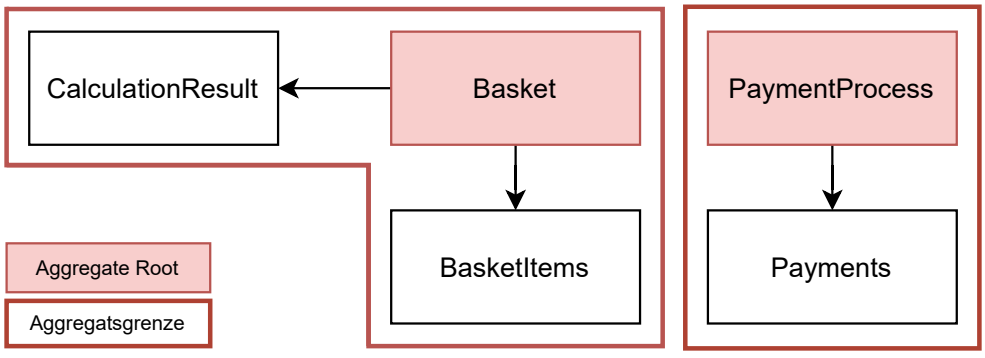


Abbildung 5.2: Aggregationsschnitt der Variante C

Da die Aggregates unabhängig voneinander agieren, müssen sie auch getrennt geladen und abgespeichert werden können. Folglich wird eine Repository-Klasse angelegt, welches diese Operationen für das neue Aggregate absolviert. Die Referenz auf den Payment-Process wird aus dem Basket entfernt und alle Funktionen, welche Aufrufe auf dieses Objekt benötigt haben, müssen entweder in die Payment-Process-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 vorliegenden Anwendungsfalles lautet wie folgt:

Bevor ein Bezahlvorgang gestartet werden darf, muss eine Evaluierung des Datenmodells stattfinden, wie beispielsweise die Überprüfung der hinzugefügten Zahlungsarten, Kundendaten und errechneten Geldbeträge. Nach erfolgreicher Validierung wird der Zustand 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 Baskets 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 aufgespalten werden. Zwischen den beiden Aktionen existiert deswegen eine Zeitspanne, in welcher der Basket zwar eingefroren, allerdings der Bezahlvorgang noch nicht gestartet wurde. Das Datenmodell hat dadurch 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 [50, S. 364f.]. 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 Baskets und dem Starten des Payment-Processes kann, aufgrund von parallel ablaufenden Anfragen, das Datenmodell weiterhin bearbeitet werden. Diese *Race Conditions* können 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 die gleiche Ressource. Der rot markierte Bereich stellt die Zeitspanne dar, in welcher andere Prozesse das Datenmodell 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.

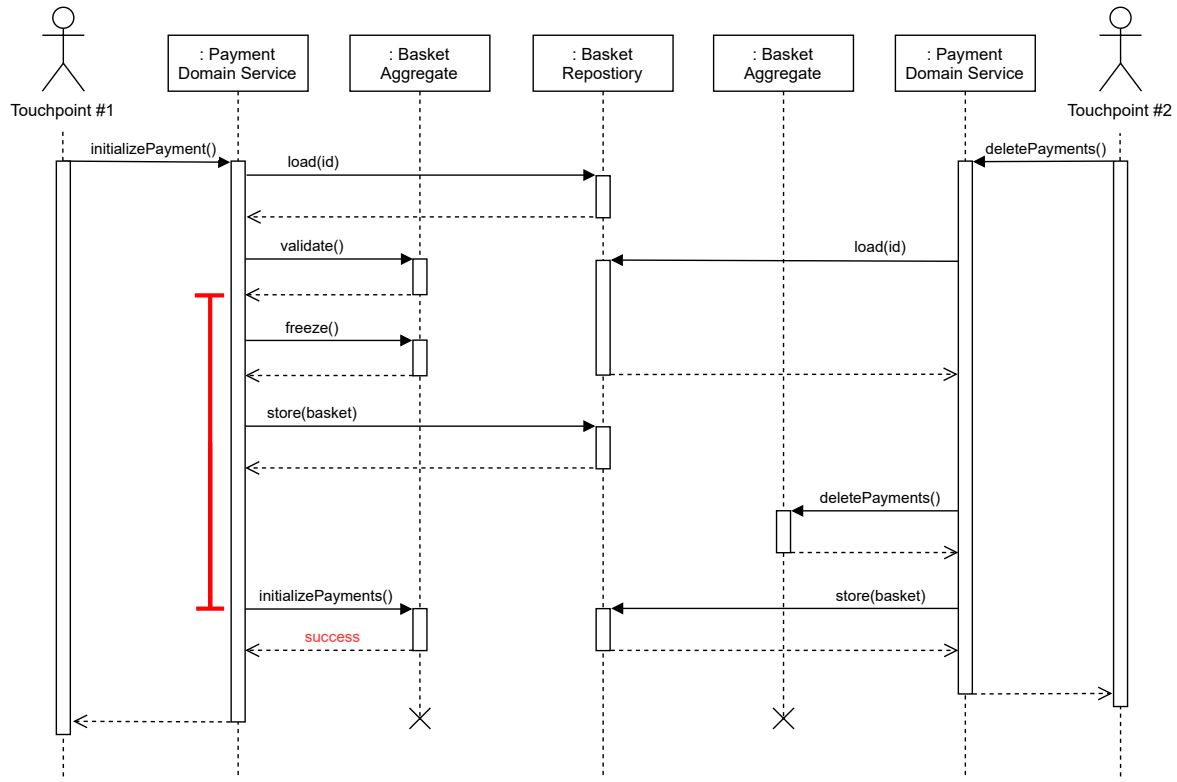


Abbildung 5.3: Sequenzdiagramm einer Race Condition bei der Initiierung des Bezahlvorgangs in Variante C

Ein valider Zustand des Baskets besitzt in der Checkout-Domain, vor allem aus rechtlichen Gründen, extreme Wichtigkeit. Mithilfe der Architektur von Variante C 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 diverse 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 im Zusammenschluss konsistent sein müssen [24, S. 128f.]. 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 [10, S. 364]. Die Aggregationsgrenzen sind folglich mit der transaktionalen Konsistenz äquivalent. Dadurch entspricht die Speicherung zweier Aggregates innerhalb einer einzelnen Transaktion eine Überschreitung dieser Grenzen und lässt vermuten, dass der Aggregationsschnitt neu eingeteilt werden muss [10, S. 358f.]. Die Richtlinie stellt dementsprechend lediglich ein Indiz für korrektes bzw. inkorrektes Design dar. Bedenklich ist hierbei, wenn die zugehörigen Aggregates auf unterschiedlichen Datenbankhosts liegen, wie bei einer Microservice-Architektur üblich. Eine atomare Transaktion über mehrere Server ist nur bedingt möglich bzw. Eventuelle Konsistenz

muss angewandt werden [51]. Aktuell besteht kein Anlass zur Annahme, dass zukünftig mehrere Datenbankhosts benötigt werden, weshalb die Überlegung entstehen kann, das Einfrieren des Baskets und die Initiierung des Zahlungsvorgangs innerhalb einer Transaktion auszuführen. Dazu ist erforderlich, dass die darunterliegende Datenbank eine atomare 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 [52]. Demzufolge ist eine Implementierung dieses Lösungsansatzes durch ein Sperrverfahren technisch anwendbar. Mithilfe eines pessimistischen Ansatzes werden beispielsweise andere Zugriffe auf den Basket bzw. Payment-Process erst ausgeführt, wenn die Initiierung vollständig abgeschlossen ist. Das Auftreten von Race Conditions wird dadurch verhindert. [53] Im Vergleich zum Einsatz von Eventueller Konsistenz wird diese Vorgehensweise bevorzugt.

5.2.3 Bewertung des Aggregationsschnittes

Weiterhin besteht die Frage, ob *Variante C* ein valides Aggregationsdesign bildet, da in vielen Funktionen eine Überschreitung der transaktionalen Grenzen stattfindet. Der Bezahlvorgang ist eine, aus rechtlicher Sicht, kritische Operation. Sobald dieser gestartet wird, muss das ganze Datenmodell stets konsistent sein. Wenn es 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 genauere 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 aggregatsübergreifenden Transaktionen bleibt die resultierende Komplexität überschaubar.
  - Sofern Eventuelle Konsistenz eingesetzt wird, treten die negativen Eigenschaften einer asynchronen Verarbeitung auf und die Wechselwirkungen zwischen Anwendungsfällen müssen stets berücksichtigt werden.
- **Performance:**
  - Bei Verwendung einer dokumentenorientierten Datenbank werden die Bearbeitungszeiten von Anfragen anhand der theoretischen Überlegungen nur minimal beeinflusst.
  - Mit einer relationalen Datenbank als Datenspeicher muss eine kleinere Anzahl an Operationen bewältigt werden, vor allem weil das Payment-Aggregate relativ gesehen selten bearbeitet wird. Daher sollte insgesamt ein Performance-Gewinn entstehen.
- **Parallelität:**
  - Der Zahlungsvorgang darf nie gleichzeitig mit dem Bearbeiten des Baskets geschehen, weshalb dieser Aspekt unverändert ist.
- **Client-Freundlichkeit:**
  - Die Aufteilung des Datenmodells 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 Gruppierung von Klassen hängt stark von den Invarianten ab, welche sie zusammenbinden. Anhand einer Untersuchung der Anwendungsfälle 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 eine 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 dem Basket hinzu und öffnet seine Ansichtsseite erst vor Abschluss des Kaufes. Dadurch ist es möglich, den Zeitpunkt der Gesamtpreiskalkulation bis zu einem expliziten Abruf zu verzögern, um Berechnungszeiten 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 *Calculation-Result* getrennt vom Basket verwaltet werden. Dieses Design kommt allerdings mit einigen Fragen, welche zuerst beantwortet werden müssen.

Ist die Trennung des Calculation-Results vom Basket überhaupt möglich aus Sicht des Business?

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

Wann muss sowohl der Basket als auch das Calculation-Result bearbeitet werden?

Als Folge der gewonnenen Erkenntnisse kann es problematisch sein, zwei Aggregates gleichzeitig anzupassen. Das Calculation-Result wird nur bei der Anzeige des Baskets 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 Ansichten des Onlineshops dargestellt. Die roten Pfeile zeigen auf wichtige Stellen des Warenkorbs für diesen Abschnitt.

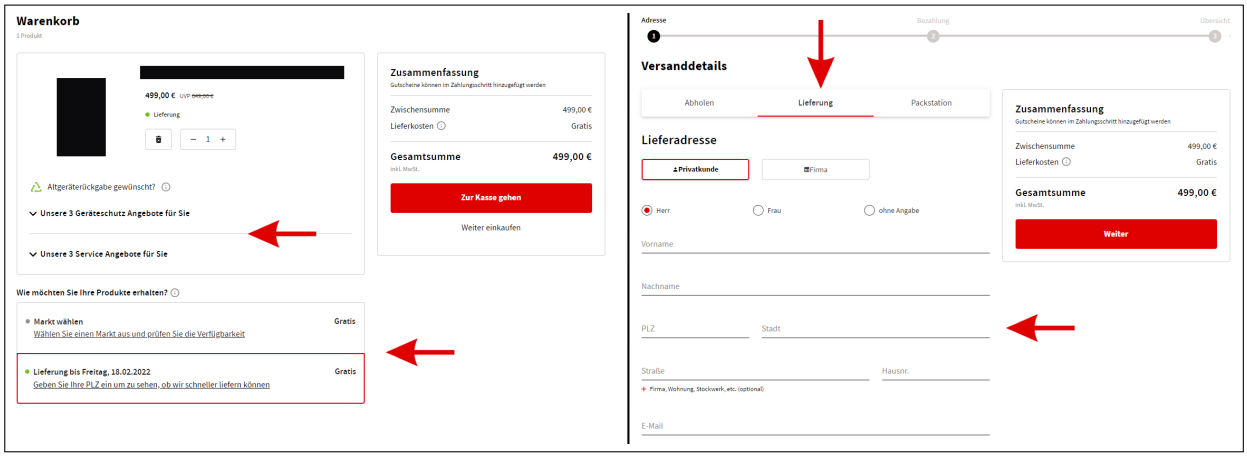


Abbildung 5.4: Aktueller Checkout-Prozess des Onlineshops von MediaMarkt.de

Eine Neukalkulation findet statt, sofern die Anzahl der Produkte oder die Lieferkosten modifiziert werden. In der linken Ansicht kann Ersteres durch eine Quantitätsänderung sowie durch Hinzufügen von Produktservices geschehen. Ebenfalls ermöglicht die nachfolgende Seite das Anpassen der Fulfillment-Methode und Lieferadresse, wodurch sich die finalen Lieferkosten eventuell ändern. Schlussfolgernd existieren einige Anwendungsfälle in denen Transaktionen über beide Aggregates hinweg notwendig sind.

Gibt es Invarianten zwischen dem Basket und dem Calculation-Result?

Es wird bereits angenommen, dass eine Verzögerung bei der Berechnung des Baskets aus Businesssicht möglich ist. Aus rechtlichen Gründen ist es zudem notwendig, eine Änderung des Gesamtpreises nach Initiierung des Zahlungsvorgangs zu verhindern. Das Problem wird in der Software gelöst, indem der Basket-Status zeitgleich mit dem Zahlungsbeginn auf 'Frozen' gesetzt wird. Weitere Datenmodifikationen werden in diesem Zustand abgewiesen. Zusätzliche Invarianten existieren zum jetzigen Zeitpunkt nicht, jedoch müssen eventuelle, zukünftige Anwendungsfälle berücksichtigt werden, ansonsten kann die Flexibilität der Anwendung gefährdet sein.

Zur Veranschaulichung dieses Aspektes werden die Auswirkungen einer neuen Regelung untersucht. Exemplarisch kann angenommen werden, dass der Gesamtpreis eines Baskets nicht über 20.000€ liegen darf. In diesem Fall würde eine Manipulation der Basket-Items auch eine Neukalkulation benötigen, wodurch die Abtrennung des Calculation-Results den Vorteil der verzögerten Berechnung verliert. Allerdings kann eine mildere Form dieser Richtlinie keinen negativen Effekt besitzen. Zum Beispiel kann die Prüfung erst beim Start des Bezahlvorgangs ausgeführt werden, da zu diesem Zeitpunkt beide Aggregates immer synchron sind und keine Änderungen mehr erfahren können. Weitere erdenkbare Businessanforderungen sollten berücksichtigt werden, bevor ein Neudesign der Applikation durchgeführt wird.

Vorläufige Analyse der Bewertungskriterien

Diese Vorgehensweise der Analyse kann analog auf verschiedene Anwendungsfälle durchgeführt werden, um weitere Teile des Baskets zu finden, welche separat agieren können. Grundsätzlich ist anhand der folgenden Evaluation eine Abspaltung der Berechnungsergebnisse vom Basket durchaus plausibel.

- **Komplexität:** Die Implementierung kann ohne umfassendere Codeanpassungen realisiert werden.
- **Performance:** Zwar wird die Anzahl der Kalkulationen minimiert, jedoch steigen die Datenbankoperationen an, weil zuvor beide Objekte innerhalb einer Tabelle persistiert werden konnten. Die Performance wird beiderseits positiv sowie negativ beeinflusst.
- **Parallelität:** Preisanpassungen ergeben sich als eine Wechselwirkung von Aktionen, 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 ermöglichen die Rückgabe beider Aggregates und neutralisieren dieses Argument, jedoch beeinflussen sie wiederum wegen zusätzlichen Datenbankoperationen und Berechnungen die Performance.

5.3.2 Herausschneiden der Checkout-Data aus dem Basket-Aggregate

In dem Aktivitätsdiagramm D wurde das Hinzufügen der Kundendaten, Zahlungsmethode und des Fulfillments beschrieben. Innerhalb eines einzelnen API-Aufrufs soll eine Anpassung dieses Datenumfangs möglich sein. Das resultierende Webshop-Design in Bild 5.4 ist ein Hinweis darauf, dass die Attribute eventuell vom Basket-Aggregate abgetrennt werden können. Sie sind in der Ubiquitous Language als *'Checkout-Data'* betitelt und beinhalten Kundendaten, Fulfillment, Rechnungs- und Lieferadresse. Ähnlich zum vorgehenden Unterkapitel ist eine Untersuchung der Implikationen eines solchen Aggregationsschnittes notwendig. 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. 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 [24, S. 129]. Deshalb wird die ursprüngliche eins-zu-eins Relation wiederum in der Datenbank als zwei separate Tabellen designt und zieht somit Performance-Einbußen bei Abfragen beider Datensätze mit sich. Jedoch werden die Checkout-Data in nahezu allen User Stories nur einmalig bearbeitet, sodass dieser Effekt minimal ausfällt.

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 Datenbankeintrag bis zum Abschluss der Bearbeitung gesperrt werden. Aufgrund von möglichen Race Conditions kann ansonsten beispielsweise die Initiierung des Bezahlvorgangs auf invalide Checkout-Data stattfinden. Weiterhin können sich bei der Auswahl einer anderen Fulfillment-Methode die Lieferkosten ändern und folglich müssen die aktualisierten Preise persistiert werden. Hierbei stößt die Applikation auf eine Transaktion über zwei Aggregates und zugleich auf die vorher untersuchte Problemstellung.

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 *Calculation-Result*, die *Checkout-Data* und den *Payment-Process* aus dem Basket in ihre eigenen Aggregates heraus trennt. Abbildung 5.5 zeigt das resultierende Datenmodell '*Variante D*'. Eine Abspaltung der Basket-Items erfordert zu viele zusätzliche Datenbankoperationen und steigert die Softwarekomplexität weiter, wodurch diese nach wie vor unter dem Basket aufgehängt sind. Falls in zukünftigen Szenarien die parallele Bearbeitung des Datenmodells unerlässlich wird, kann eine Trennung der beiden Klassen voneinander in Betracht gezogen werden.

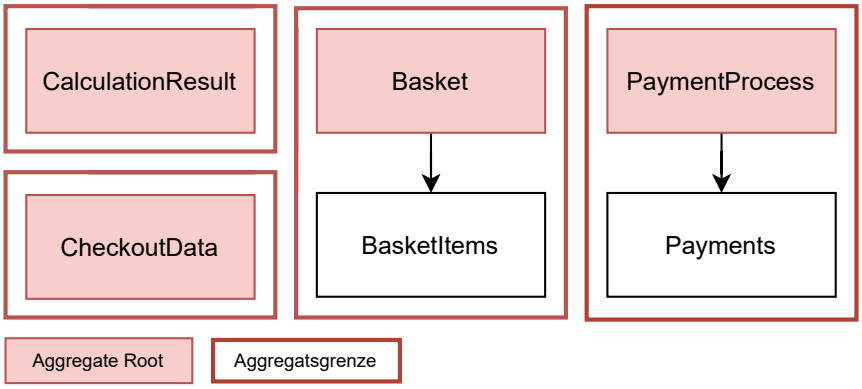


Abbildung 5.5: Aggregationsschnitt der Variante D

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 Basket-Items ist die Neuberechnung des Payment-Processes und Calculation-Results notwendig. Zur Abwicklung einer solchen Abhängigkeit sind mehrere Implementierungsmöglichkeiten denkbar.

1. Sollten die Kalkulationen erst bei Abruf stattfinden, werden Zusatzinformation benötigt, um zu indizieren, dass der aktuelle Berechnungswert veraltet ist. Zu diesem Zweck wird ein Wahrheitswert (auch 'Flag' genannt) in dem Basket und Checkout-Data hinterlegt. Bei Zustandsänderungen, welche den Gesamtwert des Baskets beeinflussen, wird dieser auf 'wahr' gesetzt. Sobald ein Calculation-Result geladen wird, findet auch eine Neuberechnung statt, sofern der Wert 'wahr' ist, welche zugleich auch den Payment-Process aktualisiert. Dieses Vorgehen spart Berechnungszeit auf Kosten von zusätzlichen Datenbankoperationen ein. Die Komplexität und Fehleranfälligkeit der Software steigen hierbei an.

2. Das Calculation-Result kann alternativ 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 ressourcenintensiv ist, sind Performance-Einbußen zu erwarten.

3. Unverändert zu *Variante A* kann die Kalkulation auch sofort bei Datenänderungen geschehen. Allerdings werden die Aggregationsgrenzen überschritten und innerhalb einer Transaktion mehrere Aggregates bearbeitet, wobei ein Zwischenweg der beiden vorgehenden Implementierungen erreicht wird.

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 werden in Services ausgelagert. Die Erzeugung der Services ist eine grundlegende Aufgabe innerhalb der Software. Verschiedene Implementierungen besitzen auch unterschiedliche Vor- bzw. Nachteile. Dieses Unterkapitel geht auf einige Realisierungsmöglichkeiten genauer ein.

Der Proof-of-Concept verwendet das Prinzip der *Dependency Injection*. Hierbei initialisiert ein Framework erforderliche Objekte von Klassen durch ihren Konstruktor, indem die obligatorischen Parameter ebenfalls injiziert werden. Dadurch entsteht ein rekursives Muster bis schließlich alle Objekte erzeugt worden sind. Das erlaubt eine lose Kopplung der Module und eine Möglichkeit wird geschaffen, die konkreten Implementierungen anhand von Konfigurationsdateien auszutauschen. Eine Problematik dieser Vorgehensweise ist die Bildung einer 'Circular Dependency' (dt. Zirkelbezug). Entsprechend Abbildung 5.6 tritt dies auf, sofern zwei Komponenten voneinander abhängig sind. Das Framework versucht eine der beiden Klassen zu initialisieren, wobei im Konstruktor ein Objekt der verbleibenden Klasse benötigt wird. Da eine Erzeugung dieses Objekt jedoch wiederum ein Objekt der ersten Klasse erwartet, entsteht ein ewiger Kreislauf von Konstruktoraufrufen. Eine Applikation mit einer Circular Dependency ist folglich nicht mehr ausführbar und die Eliminierung von Abhängigkeiten ist erforderlich. Je mehr Abhängigkeiten eine Klasse besitzt, desto höher ist die Wahrscheinlichkeit einer Circular Dependency. Bei der konkreten Implementierung des Proof-of-Concepts stellte die Vermeidung einer solchen Situation eine Herausforderung dar. [54, S. 93ff.]

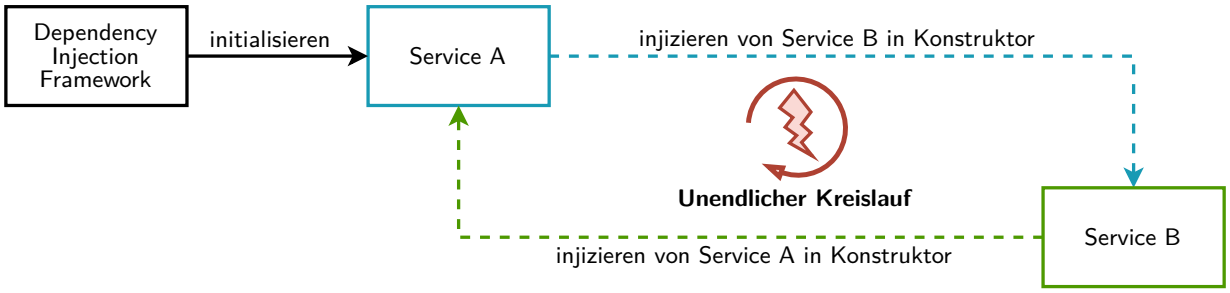


Abbildung 5.6: Darstellung einer Circular Dependency

Steuerung des Programmablaufs innerhalb von Domainservices

Sofern mehrere Komponenten für die Durchführung eines Anwendungsfalles notwendig sind, fällt das Wissen über deren Aufrufreihenfolge in den Kontext der Domain. Deshalb sind Klassen mit mehreren Serviceaufrufen generell den Domainservices zuzuordnen. Im ersten Lösungsansatz wird der Programmablauf außerhalb des Aggregates durch einen Domainservice bestimmt. Das Codebeispiel 5.2 implementiert einen Service zur Validierung einer Invariante.

```
class DomainService {

    variable SomeService someService

    function doStuff(Aggregate aggregate) {
        if (someService.isActionValid()) {
            aggregate.doStuff()
        }
    }
}
```

Codebeispiel 5.2: Bestimmung des Steuerflusses durch einen Domainservice

Das Domainwissen wird durch die zuvor stattfindende Überprüfung der Validität einer Aktion umgesetzt. Die Eintrittswahrscheinlichkeit einer Circular Dependency ist gesenkt, da in den meisten Fällen 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 benötigte Validierung geschehen. Die Konsistenz des Aggregatzustandes ist gefährdet und damit verbundenen Qualitätsmerkmale werden geschwächt. Zur Verhinderung einer solchen Situation ist eine Verlagerung des Funktionsaufrufs in das Aggregate denkbar.

Übergabe der Referenz an das Aggregate als Parameter

Ein Aggregate sollte stets alle wesentlichen Invarianten direkt überprüfen, sodass die Ausführung von invaliden Aktionen unterbunden wird. Um dies zu gewährleisten, muss der Service innerhalb des Aggregates aufgerufen werden, weswegen dieser eine Referenz auf das gefragte Objekt benötigt. Dementsprechend wird in Abbildung 5.3 der Service als Parameter an das Aggregate übergeben. 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. Der Proof-of-Concept verwendet häufig diesen Ansatz, um Abhängigkeiten zu realisieren.

```
1  class DomainService {
2
3      variable SomeService someService
4
5      function doStuff(Aggregate aggregate) {
6          aggregate.doStuff(someService)
7      }
8  }
9
10 class Aggregate {
11     // Kann nicht umgegangen werden, da Validierung direkt im Aggregate geschieht
12     function doStuff(SomeService someService) {
13         if (someService.isActionValid()) {
14             ...
15         }
16     }
17 }
```

Codebeispiel 5.3: Übergabe der Referenz an das Aggregate als Parameter

Injektion von Services in ein Aggregate durch das Repository

Weiterhin können auch zur Minimierung der Funktionsparameter benötigte Services in die Aggregates injiziert werden. Folglich halten die Aggregates selbst eine Referenz auf die jeweiligen Klassen und rufen diese bei Notwendigkeit auf. Die Injektion muss bei Objekterzeugung geschehen, weshalb die Verantwortung innerhalb des Repositories liegt. Im kurzen Beispielcode 5.4 wird der Gedanke verdeutlicht. 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 weitere Abhängigkeiten erzeugt werden. Zudem stehen in *Variante D* die Aggregates in enger Bindung zueinander und die Repositories importieren sich gegenseitig. Deshalb kann es bei Wechselwirkungen zwischen den Aggregates vorkommen, dass eine Circular Dependency auftritt. Im Buch '*Implementing domain-driven design*' auf Seite 387 warnt Vernon vor dieser Variante und präferiert die Übergabe von Abhängigkeiten per Parameter an das Aggregate.

```
class Aggregate {

    variable SomeService domainService

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

class AggregateRepository {

    variable SomeService domainService

    function load(ID id) returns Aggregate {
        variable aggregate = searchInDatabase(id)
        aggregate.inject(domainService)
        return aggregate
    }
}
```

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

5.4.3 Performance-Analyse der Aggregationsschnitte unter Einsatz von Lasttests

Damit eine genauere 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-Data, zweimaliges Abrufen des Baskets, Hinzufügen eines Payments und das Initiieren inklusive Durchführen des Bezahlvorgangs. Nach dem Vorbefüllen der Datenbank mit 10.000 Datensätzen wird anschließend dieser Prozessablauf in zehn parallelen Threads 'X'-mal durch die verschiedenen Aggregationsschnitte abgearbeitet. Zum Ausschließen von abweichenden Ergebnissen wird der Vorgang dreimal wiederholt. Die Auswertungen sind in Kapitel 8.6 zu finden.

Bemerkungen zum Performance-Test

Die Performance-Tests dienen als genereller Vergleich und weichen je nach Anwendungsfall, Hardware, Netzwerk und Softwareimplementierung ab. Anhand der Analyse sollen vorgehende Aussagen verdeutlicht und Argumente für das Fazit gebildet werden. Datenbankoperationen wie Suchanfragen können durch eine erhöhte Anzahl von Daten in der Datenbank negativ beeinflusst werden. Zu Beginn jedes Tests enthält die Datenbank 10.000 Datensätze, hingegen umfasst beispielsweise die produktiven Umgebungen der Checkout-Software Stand 11.04.2022 circa 8 Millionen Datensätze. Dies trägt zu einer möglichen Abweichung zwischen den Testergebnissen und den Applikationen im produktiven Betrieb bei. Hinzukommt, dass weitere Optimierungen vorgenommen werden können, wie die angepasste Einstellung des *Connection-Pools*, das Einsparen von ganzen Datenbankoperationen durch Caching oder schnellere Kommunikation der Systeme untereinander.

Zur 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 erkannt wird, wann eine Neukalkulation notwendig ist. Folglich steht beispielsweise 'Variante D-PF' für den Aggregationsschnitt D mit einer PostgreSQL-Datenbank und der Implementierung von Flags.

Erster Testdurchlauf und 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 J entnommen werden. Um die Konsequenz bei Verwendung einer relationalen Datenbank zu beobachten, wurde ebenfalls der Versuch mit einer PostgreSQL-Datenbank wiederholt. Die Auswertungen sind in Anhang K hinterlegt. Zum Präsentieren einer schnellen Übersicht werden die durchschnittlichen 'Abläufe pro Sekunde' über einen Gesamtumfang von 10.000 Abläufen in der Abbildung 5.7 visualisiert.

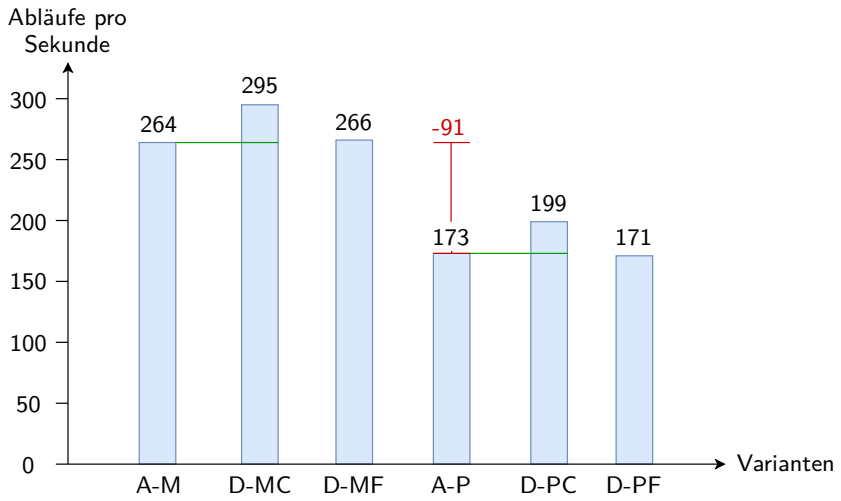


Abbildung 5.7: Performance-Ergebnisse aus dem ersten Testdurchlauf

Mithilfe von Flags werden Neukalkulationen im Austausch für zusätzliche Datenbankoperationen eingespart, jedoch sind die Berechnungen 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 komplizierteren Programmablauf einhergeht, da die Aktualisierung von Daten asynchron stattfindet. Bei der Implementierung von neuen Funktionen müssen stets die Wechselwirkungen zwischen den Wahrheitswerten und Businessprozessen beachtet werden, wodurch die Fehleranfälligkeit der Software steigt. Flags sollten nur in Betracht gezogen werden, sofern 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 Aggregates in kleinere unter Verwendung einer dokumentenorientierten Datenbank in geringeren Performance-Gewinnen resultieren kann, als beim Einsatz einer relationalen 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 Aggregate 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 gebildet und die Gesamtzahl der hierfür benötigten Datenbankoperationen aufgelistet.

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 Performance-Unterschied beläuft sich auf 4 Millisekunden zwischen A-M und D-MC, sofern eine MongoDB als Datenbankmanagementsystem genutzt wird, wohingegen eine Einsparung von 6 Millisekunden bei einer PostgreSQL möglich ist. Dieser Effekt kann deutlicher in kommenden Testaufbauten beobachtet werden, worauf die vorliegende These gestützt wird. Aus den Messwerten 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, nicht wie zu vermuten die eigentliche Datenbankoperation. Deshalb sind die Variationen A insgesamt langsamer, da eine größere Datenmenge 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 anhand von Testmessungen 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.

Migration der Datenbank in eine Cloud-Umgebung

Damit eine ähnliche 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 der Datenspeicher wiederholt und tabellarisch den Anhand L hinzugefügt. Die Abbildung 5.9 bietet die dazugehörige grafische Auswertung.

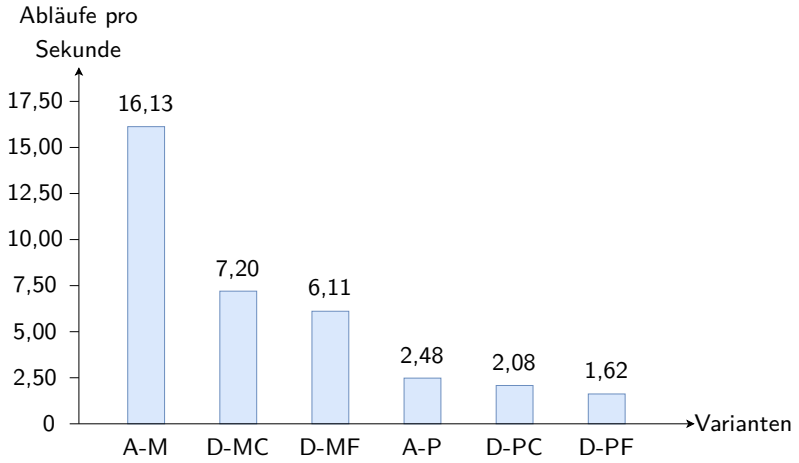


Abbildung 5.9: Performance-Ergebnisse bei Auslagerung der Datenbanksysteme

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 dem Datenspeicher aktuell die einzige Wartezeit der Applikation ist, wodurch der Einfluss auf die Performance höher ausfällt. Zudem können 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 Aufrufzeiten externen APIs

Die Tatsache, dass Aufrufe externer Systeme in zusätzlichen Wartezeiten resultieren, 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 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 Anhang M und visuell im Balkendiagramm 5.10.

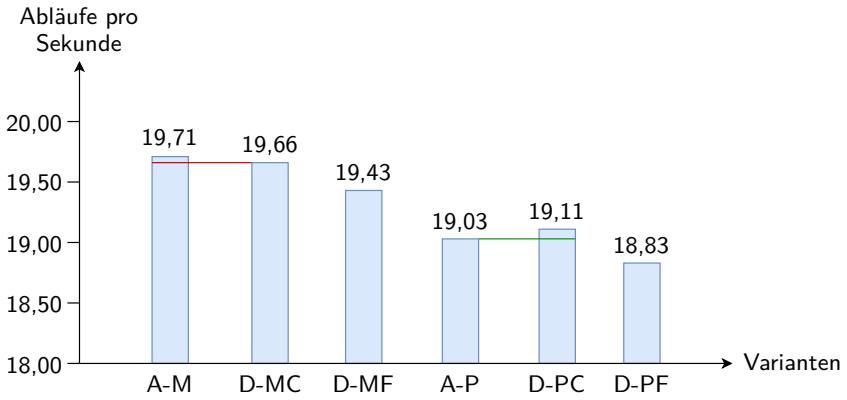


Abbildung 5.10: Performance-Ergebnisse unter Beachtung der Wartezeiten von API-Aufrufen

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

Fazit aus den Performance-Tests

Jegliche Arten von Kommunikationen mit externen Systemen gehen mit Performance-Verlusten einher. Da die eigentliche Datenverarbeitung in dem Proof-of-Concept minimal gehalten wurde, ist dies besonders bemerkbar. Die Performance-Auswirkungen müssen anhand des konkreten Anwendungsfalles bewertet werden. Bei Systemen, in welchen zusätzliche 100 Millisekunden keine relevanten Konsequenzen auf die Businessprozesse haben, kann der Einfluss des Datenmodells auf die Verarbeitungsdauer generell vernachlässigt werden. In produktiven Umgebungen fallen die erwarteten Performance-Einbußen zudem geringer aus als hier dargestellt. Letztendlich besteht jedoch die Tendenz, dass ein kleinerer Aggregationsschnitt im Kontext einer Checkout-Software in Kombination mit einer dokumentenorientierten Datenbank negative Auswirkungen auf den maximalen Datendurchsatz besitzt.

5.4.4 Bewertung des verkleinerten Aggregationsschnitts

Die finale Evaluation ist ein Resultat aus den vorhergehenden theoretischen Überlegungen, tatsächlichen Implementierungen und ein Vergleich mit anderen Varianten des Aggregationsschnitts.

- **Komplexität:** Viele Prozesse benötigen für die Einhaltung von Invarianten mehr als ein Aggregat, wodurch zusätzliche Funktionsaufrufe und Ladevorgänge 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 Aggregates dort abgerufen werden können. 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 anhand der Analyse 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 Schreibprozessen angestrebt wird. *Insgesamt gewinnt die Applikation an Komplexität, welche je nach Realisierungsmethode stärker oder schwächer ausfallen kann.*
- **Performance:** *Basierend auf die vorgehende Analyse ist ein Verlust von Performance zu erwarten, da in der Live-Umgebung eine MongoDB verwendet wird.*
- **Parallelität:** Generell können zwei unterschiedliche Aggregates unabhängig voneinander bearbeitet werden. Als konkretes Beispiel ist die Anpassung 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 Basket-Items mitaufgenommen werden, gewinnt dieser Aspekt an Bedeutung. *Die gewonnene Parallelität ist folglich nahezu vernachlässigbar.*
- **Client-Freundlichkeit:** Touchpoints erhalten als Antwort auf API-Anfragen in dieser Variante nur einen Teilaspekt des Baskets, denn andererseits müssten wieder alle Aggregates 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 Aufrufe weiterhin ihre Arbeitsprozesse abwickeln können. *Letztendlich sind Clients bestenfalls nur neutral von der neuen Architektur betroffen.*

Zusammenfassend existiert nur sehr beschränkt eine Grundlage 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 gewonnenen Erkenntnisse. Ein Verweis auf den Sourcecode ist in Kapitel 8.1 hinterlegt.

Zu Beginn wurde das Projekt mit benötigten Frameworks und Abhängigkeiten aufgesetzt, um Boilerplate-Code weitestgehend zu vermeiden. Die Software wurde mithilfe von Kotlin entwickelt. Die Komponenten sind analog zu einer Hexagonalen Architektur aufgeteilt in primäre Adapter, Applikationskern und sekundäre Adapter. 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 Ansatzes mit einem einzelnen, großen Basket-Aggregate tiefer ein.

6.1 Design der primären Adapter

Primäre Adapter sind die grundlegenden Kommunikationsschnittstellen zwischen Clients und der Software. Sie initiieren den Datenfluss anhand eines externen Signals. Im Proof-of-Concept fallen in diesen Bereich hauptsächlich die sogenannten Controller, welche für den Empfang von REST-API-Anfragen, die Deserialisierung übergebener Daten, sowie die Serialisierung des Antwortinhaltes zuständig sind. Zu Beginn jedes Anwendungsfalles wird ein Controller durch den Touchpoint angesprochen. Der jeweils zuständige Adapter wird aus dem Zusammenschluss der aufgerufenen URL und HTTP-Methode bestimmt. Ein Controller beinhaltet lediglich Logik für den Empfang von Daten und der Formulierung zugehöriger Antworten. Alle externen Informationen müssen vor der Weitergabe an den Applikationskern in ein Objekt des Domainmodells umgewandelt werden. Ist dies nicht der Fall, erhält der zentrale Teil der Software eine Abhängigkeit nach außen und das Dependency-Inversion-Prinzip wird verletzt. Ein konkreter Controller ist im Codebeispiel 6.1 abgebildet. Zur Implementierung dieser Funktionalitäten wurde das Framework 'Ktor' eingesetzt.

```
1 put("/basket/{id}/customer") {
2     variable basketID = parseParameterFromUrl("id")
3     variable customer = request.parseBody<Customer>()
4
5     variable basket = basketApiPort.setCustomer(basketID, customer)
6
7     request.respond(HttpStatusCode.OK, basket)
8 }
```

Codebeispiel 6.1: Beispiel eines Controllers zum Aktualisieren 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 Pfadparameter
- Zeile 3: Deserialisierung der übertragenen Daten zu einem Customer-Objekt
- Zeile 5: Weitergabe der Parameter an den zuständigen Applicationservice mithilfe eines Ports
- Zeile 7: 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' 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 anderer Klassen und Test-Fehlschläge können eindeutig einem bestimmten Abschnitt der Software zugeschrieben werden. [55, 56]

6.2 Realisierung des Applikationskerns

Der Applikationskern stellt das Herz der Anwendung dar. Das Ziel einer Hexagonalen Architektur ist es, das Zentrum komplett von äußeren Modulen zu entkoppeln [41]. In Domain-Driven Design liegen die Applicationservices, Domainservices und das Datenmodell im Inneren des Applikationskerns [10, S. 125ff.].

6.2.1 Applicationservices

Ein simples Beispiel für einen implementierten Applicationservice bietet der Basket-Item-Applicationservice, welcher bei Änderungen an den Items angesprochen wird. Das Codebeispiel 6.2 behandelt die Methode zum Entfernung eines Basket-Items. Hierbei wird keine Businesslogik im Service verankert, lediglich das Transaktionsmanagement und die Ablaufsteuerung der Funktionsaufrufe von Aggregates bzw. Domainservices.

```
1 function removeBasketItem(BasketID basketID, BasketItemID basketItemID) {
2     transaction {
3         variable basket = basketRepository.findByID(basketID)
4         basket.removeBasketItem(basketItemID)
5         basketStorageService.store(basket)
6     }
7 }
```

Codebeispiel 6.2: Funktion zum Entfernen von Basket-Items in einem Applicationservice

- Zeile 2: Starten einer Transaktion über die Zeilen 3 bis 5.
- Zeile 3: Laden eines Baskets anhand seiner ID durch ein Repository.
- Zeile 4: Aufruf einer Funktion des Aggregate Roots zum Entfernen des übergebenen Items. Innerhalb dieser Funktion werden zusätzlich Aufgaben erledigt, wie das Neuberechnen des Gesamtpreises. Sollte die Kalkulation zu komplex ausfallen, kann ein Berechnungsservice als Parameter übergeben werden, sodass der Basket weiter für seine eigene Konsistenz verantwortlich ist.
- Zeile 5: Speichern des Baskets mit den abgeänderten Daten.

6.2.2 Basket-Aggregate

Damit der Basket für seine eigene Konsistenz zuständig sein kann, müssen jegliche Änderungen durch eine Methode im Aggregate selbst geschehen. Anwendungsfälle, die es erfordern tiefer gelegene Objekte anzupassen, werden durch eine Kette von Funktionsaufrufen umgesetzt. Zur Trennung der Datenhaltung von den Funktionalitäten implementiert der Basket ein Interface. Dadurch kann das darunterliegende Datenmodell ausgetauscht oder in Tests simuliert werden. Der Codeauszug 6.3 veranschaulicht das Abändern der Fulfillment Methode innerhalb des Baskets.

```
1  function setFulfillment(Fulfillment fulfillment, FulfillmentPort fulfillmentPort) {
2      validateIfModificationIsAllowed()
3
4      variable availableFulfillment = fulfillmentPort.getAvailableFulfillment(outletID)
5
6      throwIf(availableFulfillment.doesNotContain(fulfillment)) {
7          IllegalModificationError("$fulfillment is not avaiable")
8      }
9
10     this.fulfillment = fulfillment
11 }
```

Codebeispiel 6.3: Setzen der Fulfillment Methode im Basket Aggregate

- Zeile 2: Überprüfung der Invariante, ob der Basket anhand seines Status aktuell Änderungen zulässt.
- Zeile 4: Laden der verfügbaren Fulfillment Methoden für diesen Basket durch einen sekundären Adapter. Die Kommunikation mit dem Adapter erfolgt über einen Port.
- Zeile 6-8: Falls der neue Wert nicht unter den verfügbaren Fulfillments ist, wird der Aufruf zurückgewiesen und eine entsprechende Fehlermeldung an den Client durch den Controller geliefert.
- Zeile 10: Überschreiben des alten Wertes. Dieser Punkt wird nicht erreicht, wenn zuvor eine Businessanforderung gescheitert ist.

6.2.3 Domainservices

Aufgaben, welche nicht direkt einem Objekt zugewiesen werden können oder mehrere Aggregates betreffen sind in Domainservices zu implementieren [10, S. 267]. Beispielsweise wurde im Proof-of-Concept aus Gründen der Übersichtlichkeit und Kohäsion die Abwicklung des Bezahlverfahrens aus dem Basket herausgetrennt und in einem Domainservice implementiert. In Code-Ausschnitt 6.4 ist die Ausführung des Bezahlvorgangs abgebildet.

```
1  function executePaymentProcessAndFinalizeBasket(BasketID basketID) {
2      variable basket = basketRepository.findByID(basketID)
3
4      throwIf(basket.isNotFrozen() or basket.paymentIsNotInitialized()) {
5          IllegalModificationError("cannot cancel payment process")
6      }
7
8      variable externalPaymentRef = basket.getExternalPaymentRef()
9      paymentPort.executePayment(externalPaymentRef)
10     basket.executePayments() and basket.finalize()
11     basketStorageService.store(basket)
12     createOrderAfterFinalization(basket)
13 }
```

Codebeispiel 6.4: Ausführung des Bezahlvorgangs in einem Domainservice

- Zeile 2: Laden des Baskets aus dem Repository.
- Zeile 4-6: Weist die Durchführung zurück, sofern der Basket sich nicht in dem erwarteten Zustand befindet. Dies kann auftreten, wenn die REST-API aufgerufen worden ist, ohne dass ein Zahlungsprozess zuvor gestartet wurde.
- Zeile 8-10: Durchführung des Bezahlvorgangs. Die erforderliche Aufrufreihenfolge stellt einen Teil des Domainwissens dar und begründet die Zuteilung der Klasse in die Gruppe der Domainservices.
- Zeile 11: Speichern des angepassten Baskets.
- Zeile 12: Erstellen eines Bestellvorgangs durch einen separaten Domainservice.

6.3 Anbinden externer Systeme und Datenbanken durch sekundäre Adapter

Das in der Planungsphase erstellte Context-Diagramm 3.5 zeigt verschiedenste Systeme mit denen die Anwendung zum Erfüllen ihrer Aufgaben kommunizieren muss. Für diesen Zweck wurden Services, welche das Aufrufen externen API-Schnittstellen simulieren, erstellt. Damit Brücken zwischen der Domain und den Services gewährleistet sind, implementieren die sekundären Adapter ein vom Applikationskern definiertes Interface. Analog zu den primären Adaptern, existieren keine direkten Abhängigkeiten des Anwendungskerns zu dem Teil der Software.

Grundsätzlich wird pro Aggregate ein Repository implementiert. Sie verwalten den Zugriff auf die Datenbank und alle Funktionalitäten, welche in dieses Aufgabengebiet fallen, wie elementare Speicher- und Suchfunktionen. Zusätzlich existieren spezielle Komponenten für das Erfragen der aktuellen Preis- bzw. Artikelinformationen. Aufgrund von Performance-Verbesserungen wurden zusätzliche Abwandlungen der Adapter mit Caching-Funktion erstellt. Der normal fungierende Adapter ruft den zugehörigen API-Service auf, wohingegen der Caching-Adapter den Preis bzw. Artikel aus dem Cache lädt. Falls der Eintrag veraltet ist, wird der eigentliche Adapter angesprochen, um die zum jetzigen Zeitpunkt validen Daten zu erfragen und im Cache abzulegen. Das Beispiel 6.5 stellt die Komponente für das Aktualisieren der Preisinformationen dar.

```
1  class CachedPriceAdapter {
2
3      variable PriceCachingRepository priceCachingRepository
4      variable PriceAdapter priceAdapter
5
6      function fetchPrice(PriceID priceID) returns Price {
7          return priceCachingRepository.getAndUpdateIfInvalid(priceID, fallback = {
8              priceAdapter.fetchPrice(priceID)
9          })
10     }
11 }
```

Codebeispiel 6.5: Preisadapter mit Caching-Funktion

- Zeile 3-4: Der *CachedPriceAdapter* hat eine Abhängigkeit zum normalen Preisadapter und zu einem Repository zum Abrufen des zwischengespeicherten Preises
- Zeile 7: Abfragen des Preises aus dem Caching-Repository. Sollte der Preis invalide sein, weil er beispielsweise veraltet ist, wird Zeile 8 ausgeführt.
- Zeile 8: Weiterleitung der Anfrage an den zuständigen Adapter, welcher das externe System aufruft. Das Ergebnis wird mit einem aktuellen Zeitstempel im Cache abgelegt.

7 Fazit und Empfehlungen

Zu Beginn der Projektdurchführung wurden verschiedene Architekturstile untersucht und bewertet. Der Aufbau einer Hexagonalen Architektur unterstützt bei der Entkopplung des Applikationskerns und ermöglicht das instinktive Einhalten der SOLID-Prinzipien. Im Proof-of-Concept hat eine Einteilung in Adapter, Ports und Businesslogik den Entwicklungsprozess erleichtert. Die Hexagonale Architektur bildet ein stabiles und erweiterbares Fundament für die Checkout-Applikation. Zusätzlich fördert diese Softwarestruktur den Einsatz eines Domain-Driven Design, was einen effektiven Aggregationsschnitt ermöglicht, wodurch die Skalierbarkeit und Performance der Anwendung erhöht wird.

Damit ein Umbau der aktuellen Produktivsoftware zu empfehlen ist, sollten die analysierten Domain-Modelle diese Qualitätsmerkmale positiv beeinflussen. Die Performance-Tests haben ergeben, dass der Einsatz eines kleineren Aggregationsschnitts keine Optimierung der Bearbeitungszeit zur Folge hat. Dies lässt sich auf die vielzähligen Datenbankoperationen bei einem aufgeteilten Aggregationsschnitt zurückführen, welche aufgrund der starken Kopplung einzelner Entitäten untereinander entstehen. In vielen Softwareprojekten existieren wenige Invarianten, die sich über das ganze Domain-Modell spannen [10, S. 355ff.]. Oftmals ist deshalb eine Trennung der Klassen voneinander problemlos möglich, sodass ein effektiveres Design erreicht werden kann. Der Basket ist allerdings ein enger Verbund aus Businessrichtlinien. Die transaktionale Konsistenz muss wegen fiskalischen Anforderungen stets eingehalten werden. Ein allumfassender Aggregationsschnitt ist aus Businesssicht somit ebenfalls sinnvoll. Zudem ist die parallele Bearbeitung eines Baskets zum jetzigen Zeitpunkt nicht notwendig, weshalb aus diesem Aspekt keine negativen Einflüsse durch den umfangreichen Aggregationsschnitt entstehen. Sofern die zeitgleiche Modifikation von Basket-Items zukünftig einen gängigen Anwendungsfall darstellt, müssen die Artikel als eigenständige Aggregates designt werden, wodurch ein verringerter Datendurchsatz und erhöhte Softwarekomplexität zu erwarten ist. Das Abspalten der Preiskalkulation ist ebenfalls denkbar, um Berechnungszeiten einzusparen. Die Komplexität des Sourcecodes steigt in diesem Fall leicht an. Sollten die Touchpoints bei der Mehrheit der API-Anfragen jedoch auch zugleich die neu kalkulierten Preise erwarten, verfallen die gewonnenen Performance-Verbesserungen. Weitere Designvariationen sind, anhand der in den jeweiligen Unterkapiteln besprochenen Auswirkungen, zur Verbesserung der Qualitätsmerkmale ungeeignet.

Schlussendlich fallen die Argumente für eine Umgestaltung des Aggregationsschnittes in Zusammenhang mit den aktuellen Businessanforderungen zu schwach aus, sodass ein Re-Design der Applikation nicht empfehlenswert ist.

Dennoch ist in den meisten Fällen ein kleinerer Aggregationsschnitt zu bevorzugen, da viele Applikationen kaum Invarianten besitzen. Allgemein kann sich in Softwareprojekten ein idealer Aggregationsschnitt mithilfe einer detaillierten Analyse der Anwendungsfälle und Integritätsgrenzen herauskristallisieren. Die ermittelten Invarianten zwischen Datenstrukturen bestimmen maßgeblich umsetzbare Designansätze. Ferner können verwendete Technologien durchaus Einfluss auf die Architektur der Software haben, allerdings sollte das mit dem Bewusstsein geschehen, dass sich diese zeitnah ändern können und deshalb bei deren Einbindung in den Entscheidungsprozess ein Risiko entsteht. Anforderungen an die Applikation und entstandene *technische Schulden* besitzen oftmals eine höhere Priorität im Vergleich zu den theoretischen Prinzipien des Softwaredesigns, da letztendlich eine funktionierende Software im Vordergrund steht. Eine zukunftssichere Applikation gelingt somit unter Einhaltung der gängigen Richtlinien eines Domain-Driven Designs in Kombination mit etablierten Designprinzipien und intuitiven Lösungsansätzen von erfahrenen Entwickler:innen.

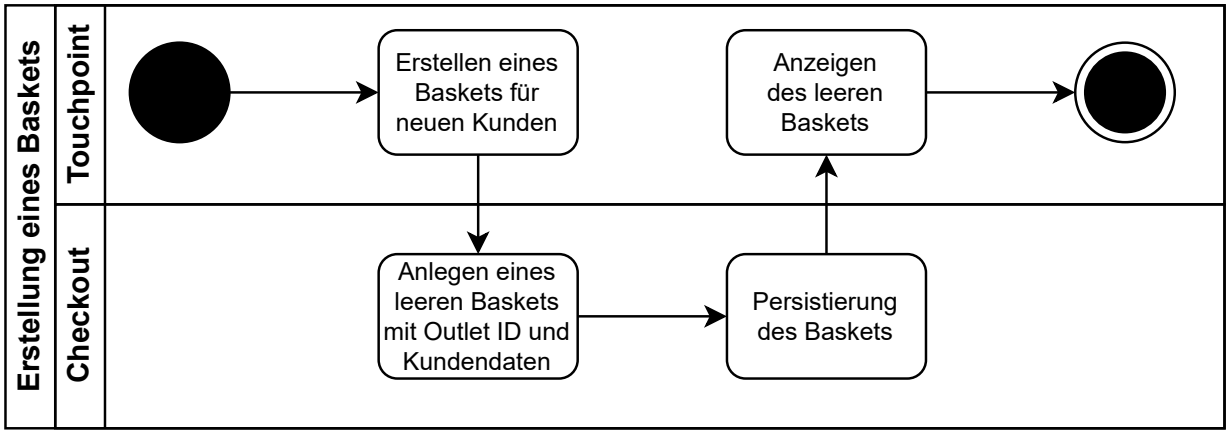
8 Anhang

8.1 Sourcecode des Proof-of-Concepts

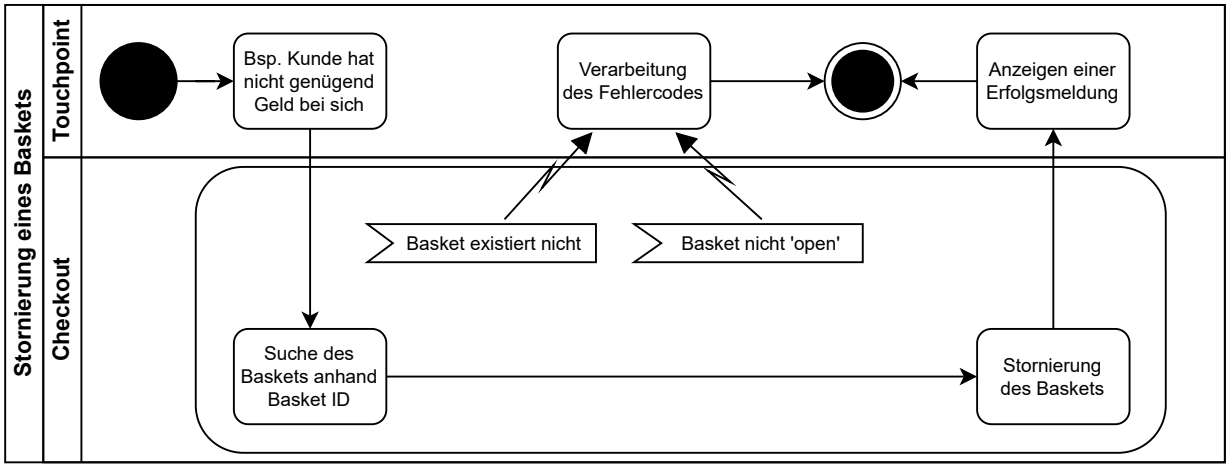
Die Binärdateien des Projekts wurden zur Versionierung in ein Git-Repository hinterlegt. Diese umfassen die Definition der Lasttests, API-Beschreibung, und den Quelltext, inklusive der analysierten Aggregationsschnitte. Das Repository kann unter dem Link '`https://github.com/Thalmaier/bachelor-thesis-checkout-poc`' aufgerufen werden. Alternativ ist in Anhang A ein QR-Code abgebildet.

Anhang A: QR-Code des GitHub Repositories
URL: `https://github.com/Thalmaier/bachelor-thesis-checkout-poc`

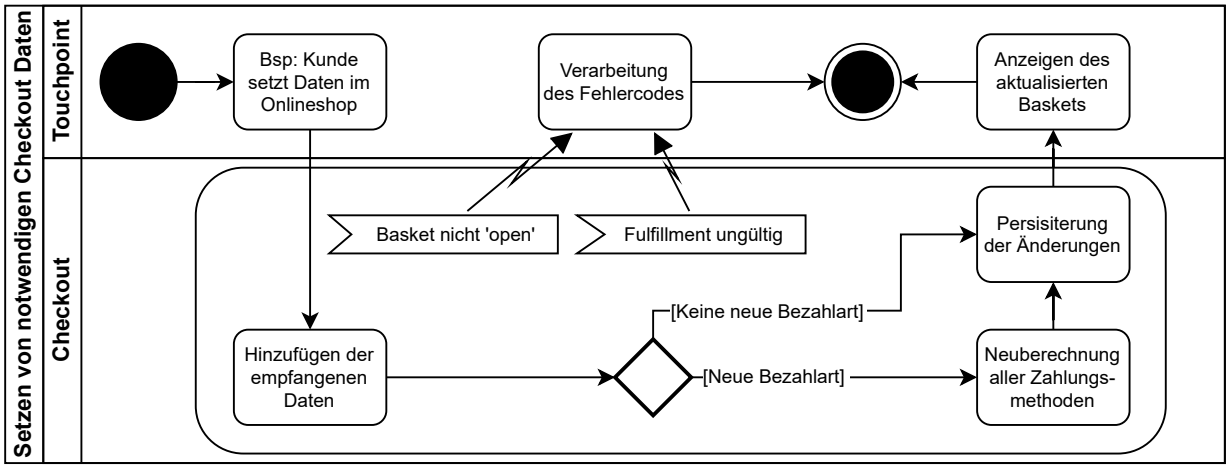
8.2 Aktivitätsdiagramme der Anwendungsfälle



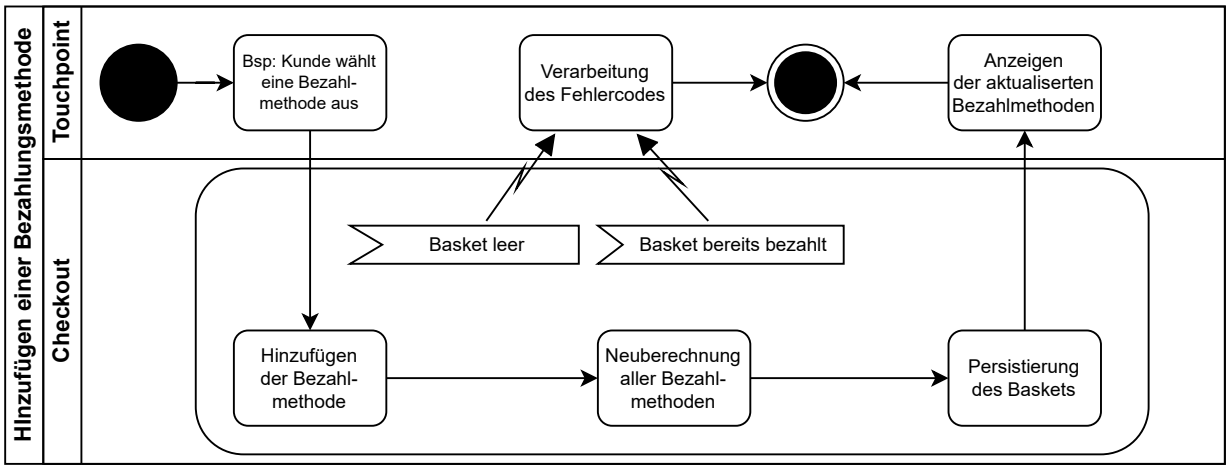
Anhang B: Aktivitätsdiagramm für die Erstellung eines Baskets



Anhang C: Aktivitätsdiagramm für die Stornierung eines Baskets



Anhang D: Aktivitätsdiagramm für das Setzen der Checkout Daten



Anhang E: Aktivitätsdiagramm für das Hinzufügen einer Bezahlmethode

8.3 API-Endpunkte

Die OpenAPI Definition der API ist in Anhang A hinterlegt. Zudem bietet Anhang F eine grafische Übersicht der Endpunkte für Variante A des Aggregate-Designs.

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		

Anhang F: REST-API der Checkout-Software für diesen Proof-of-Concept

8.4 Vollständiges Datenmodell des Proof-of-Concepts

Basket:

- **BasketId:** Eindeutige Identifikation des Baskets zur Referenzierung durch die Touchpoints.
- **OutletId:** Eine Referenz zugehörig zu einem Markt oder Onlineshop, durch welchen der Basket angelegt wurde. Unerlässlich für die Bestimmung von unter anderem Lagerbeständen, Lieferzeiten, Fulfillment-Optionen und Versandkosten.
- **BasketStatus:** Repräsentiert den aktuellen Zustand des Baskets. 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 Produkte und ihre zugehörigen Informationen.
- **BasketCalculationResult:** Beinhaltet die berechneten Werte des Basket, wie Nettobetrag, Bruttobetrag und Mehrwertsteuer. Die Speicherung dieser Werte wäre technisch nicht notwendig, spart aber Rechenzeit, da nicht bei jeder Abfrage des Basket dieser Wert neu berechnet werden muss.
- **PaymentProcess:** Enthält alle Informationen zur erfolgreichen Abwicklung des Zahlungsprozesses.
- **Order:** Speichert eine Referenz auf die Bestellung eines Baskets. Wird erst nach Zahlungsabschluss befüllt.

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.
- **E-Mail:** E-Mail 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, die errechneten Nettokosten, Lieferkosten und den Gesamtpreis.

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

Vat:
- **Sign:** Identifizierung des Steuertyps, abhängig vom 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:** Ein zusammengebautes Set von VatAmounts aller Preise der BasketItems. Benötigt, da VATs mit unterschiedlichen Prozentbeträgen rechtlich nicht kombiniert werden dürfen.

VatAmount:
- **Sign:** Identifizierung des Steuertyps, abhängig vom 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:** Fasst alle VATs zusammen, welche das gleiche Sign besitzen.

PaymentProcess:

- **BasketId:** ID des zugehörigen Baskets.
- **ExternalPaymentRef:** Referenz auf den Bezahlvorgang im externen System. Bis zur Initiierung des Payments leer.
- **AmountToPay:** Betrag der insgesamt bezahlt werden muss. Entspricht dem GrandTotal des Baskets.
- **AmountPayed:** Rechnet alle Payments zusammen und bestimmt im 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 des Bezahlvorgangs anhand von AmountToPay. Kann die Werte 'TO_PAY', 'PARTIALLY_PAID' und 'PAID' annehmen.
- **Payment:** Liste aller Payments zugehörig zu diesem Prozess.

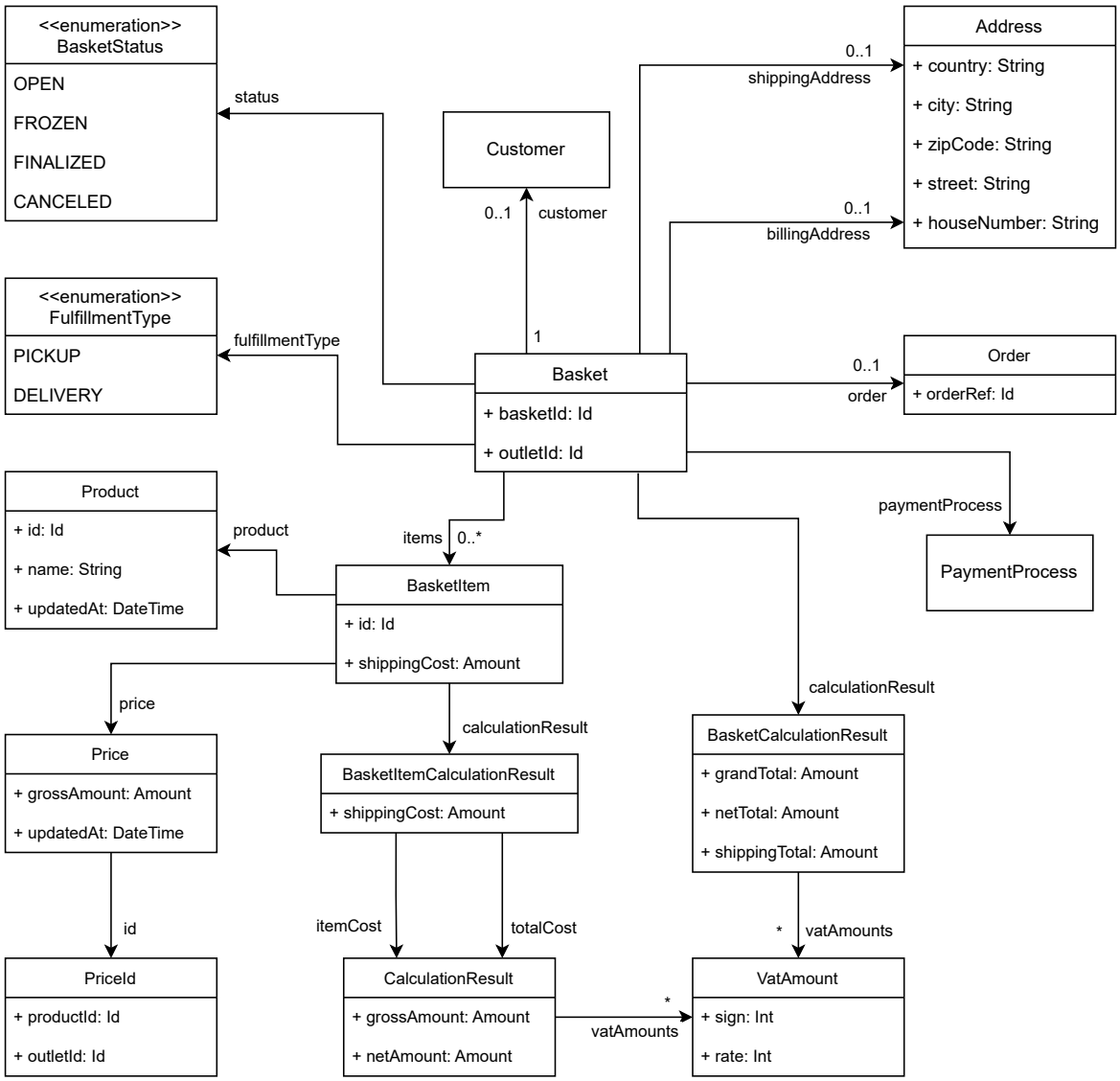
Payment:

- **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 initial 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.

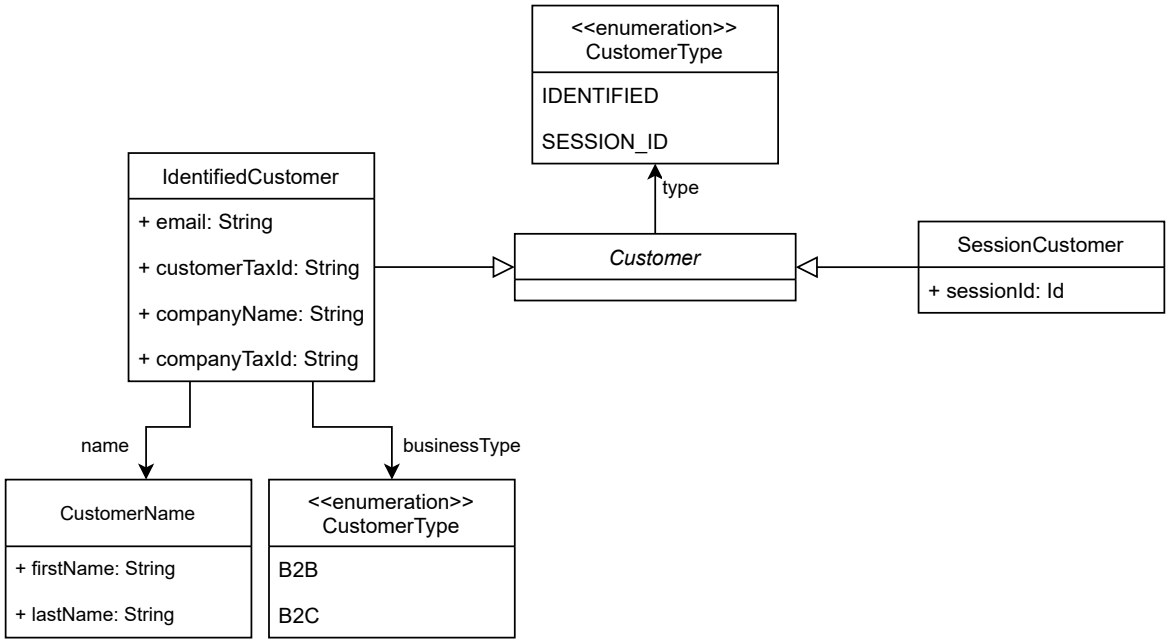
Order:

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

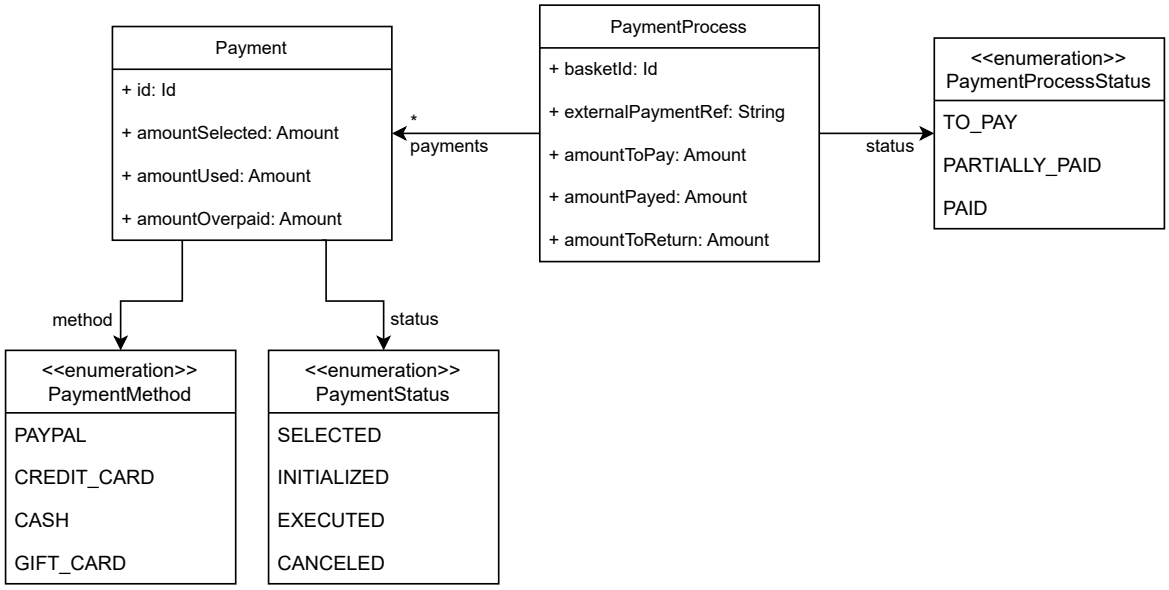
8.5 Klassendiagramme des Datenmodells



Anhang G: Klassendiagramm eines Baskets



Anhang H: Klassendiagramm des Customer Value Objects



Anhang I: Klassendiagramm des Payment Process

8.6 Ergebnisse des Lasttests

Der Lasttest wurde mithilfe der Software 'JMeter' durchgeführt. Die Vorlage der jeweiligen Testfälle sind im Repository des Proof-of-Concepts unter Anhang A zu finden. Ein Durchlauf bezieht sich auf eine typische *User Story*, welche folgende Aspekte beinhaltet: 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.

Folgende Abkürzungen wurde zur Übersichtlichkeit genutzt:

- **A**: 'Variante A' des Aggregationsschnittes
- **D**: 'Variante D' des Aggregationsschnittes
- **M**: Verwendung des Datenbankmanagementsystems MongoDB
- **P**: Verwendung des Datenbankmanagementsystems PostgreSQL
- **F**: Kurz für 'Flag'. Die Kalkulation des Gesamtpreises geschieht erst bei expliziter Abfrage
- **C**: Die Kalkulation des Gesamtpreises findet umgehend bei Anpassungen von relevanten Werten statt.
- **AZ**: Kurz für 'Ablaufzeit'. Angabe in Millisekunden. Die Zeit für einen einzelnen Ablauf des Anwendungsfalles.

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

Anhang J: 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

Anhang K: 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

Anhang L: 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

Anhang M: Lasttest-Ergebnisse mit Simulation der API-Aufrufe durch künstliche Verzögerung

Literatur

[1] Dominic Betts u. a. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*. 1st. 2013. ISBN: 1621140164.

[2] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm (besucht am 23. 04. 2022).

[3] Martin Fowler. *AnemicDomainModel*. 2003. URL: https://www.martinfowler.com/bliki/AnemicDomainModel.html (besucht am 24. 04. 2022).

[4] Ralf Lämmel und Simon Peyton Jones. "Scrap your boilerplate". In: *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation - TLDI '03*. Hrsg. von Peter Lee und Zhong Shao. New York, New York, USA: ACM Press, 2003, S. 26. ISBN: 1581135262. DOI: 10.1145/604174.604179.

[5] Meet Zaveri. *What is boilerplate and why do we use it? Necessity of coding style guide*. 2018. URL: https://medium.com/free-code-camp/whats-boilerplate-and-why-do-we-use-it-let-s-check-out-the-coding-style-guide-ac2b6c814ee7 (besucht am 24. 04. 2022).

[6] MongoDB. *Databases and Collections*. URL: https://www.mongodb.com/docs/manual/core/databases-and-collections/ (besucht am 23. 04. 2022).

[7] Shaikh Sohel und Vinod Pachghare. "A Comparative Study of Database Connection Pooling Strategy". In: *International Research Journal of Engineering and Technology* 04.05 (2017). (Besucht am 24. 04. 2022).

[8] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. URL: https://martinfowler.com/articles/injection.html (besucht am 10. 04. 2022).

[9] Craig Larman. *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development*. 3. ed., 12. print. Upper Saddle River, NJ: Prentice Hall, 2009. ISBN: 978-0131489066.

[10] Vaughn Vernon. *Implementing domain-driven design*. Fourth printing. Upper Saddle River, NJ: Addison-Wesley, 2015. ISBN: 9780321834577.

[11] Edward Yourdon und Larry L. Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*. Englewood Cliffs, NJ: Prentice-Hall, 1979. ISBN: 978-0-13-854471-3.

[12] Martin Fowler und David Rice. *Patterns of enterprise application architecture*. 17. print. The Addison-Wesley signature series. Boston, Mass.: Addison-Wesley, 2011. ISBN: 978-0321127426.

[13] *Lost-Update*. URL: https://wikis.gm.fh-koeln.de/Datenbanken/Lost-Update (besucht am 24. 04. 2022).

[14] *What is a Product Owner?* URL: scrum.org/resources/what-is-a-product-owner (besucht am 24. 04. 2022).

[15] IBM, Hrsg. *Secure programmer: Prevent race conditions: Resource contention can be used against you*. 2004. URL: https://web.archive.org/web/20090201132237/http://www-128.ibm.com/developerworks/linux/library/l-sprace.html (besucht am 24. 04. 2022).

[16] *What is Scrum?* URL: https://www.scrum.org/resources/what-is-scrum (besucht am 07. 06. 2022).

[17] *What is a Sprint in Scrum?* URL: https://www.scrum.org/resources/what-is-a-sprint-in-scrum (besucht am 24. 04. 2022).

[18] *Scrum Glossary*. URL: https://www.scrum.org/Resources/Scrum-Glossary (besucht am 24. 04. 2022).

[19] Martin Fowler. *TechnicalDebt*. 2019. URL: https://martinfowler.com/bliki/TechnicalDebt.html (besucht am 29. 04. 2022).

[20] T. Tamai und Y. Torimitsu. "Software lifetime and its evolution process over generations". In: *Proceedings Conference on Software Maintenance 1992*. IEEE Comput. Soc. Press, 1992, S. 63–69. ISBN: 0-8186-2980-0. DOI: 10.1109/ICSM.1992.242557.

[21] J. Bosch und P. Bengtsson. "Assessing optimal software architecture maintainability". In: *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. IEEE Comput. Soc, 2001, S. 168–175. ISBN: 0-7695-1028-0. DOI: 10.1109/CSMR.2001.914981.

[22] Mike Loukides und Steve Swoyer. *Microservices Adoption in 2020: Everyone's talking about microservices. Who's actually doing it?* 2020. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/ (besucht am 10. 04. 2022).

[23] Adalberto R. Sampaio u. a. "Supporting Microservice Evolution". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, S. 539–543. ISBN: 978-1-5386-0992-7. DOI: 10.1109/ICSME.2017.63.

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

[25] Roland Hienz Steinegger u. a. *Designing Microservice-Based Applications by Using a Domain-Driven Design Approach*. 2017. URL: https://cm.tm.kit.edu/download/domain_driven_microservice-architecture_17-03-15.pdf (besucht am 10. 04. 2022).

[26] Thomas M. Pigoski. *Software Maintenance*. 2001. URL: https://sce.uhcl.edu/helm/SWEBOK_IEEE/data/swebok_chapter_06.pdf (besucht am 10. 04. 2022).

[27] *Unternehmensinformationen zu MediaMarktSaturn*. URL: https://www.mediamarktsaturn.com/unternehmen (besucht am 10. 04. 2022).

[28] *MediaMarktSaturn Technology Informationen (Firmeninternes Dokument)*. URL: https://mediasaturn.sharepoint.com/sites/tech-betriebsrat-msitis (besucht am 10. 04. 2022).

[29] David P. Darcy, Sherae L. Daniel und Katherine J. Stewart. "Exploring Complexity in Open Source Software: Evolutionary Patterns, Antecedents, and Outcomes". In: *2010 43rd Hawaii International Conference on System Sciences*. IEEE, 2010, S. 1–11. ISBN: 978-1-4244-5509-6. DOI: 10.1109/HICSS.2010.198.

[30] Robert C. Martin. *Design Principles and Design Patterns*. 2000. URL: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf (besucht am 10. 04. 2022).

[31] Robert C. Martin. *Clean architecture: A craftsman's guide to software structure and design*. Robert C. Martin series. Boston: Prentice Hall, 2018. ISBN: 978-0134494166. URL: http://proquest.tech.safaribooksonline.de/9780134494272.

[32] Robert C. Martin. *SRP: The Single Responsibility Principle*. URL: https://web.archive.org/web/20150202200348/http://www.objectmentor.com/resources/articles/srp.pdf (besucht am 10. 04. 2022).

[33] Bertrand Meyer. *Object-oriented software construction*. 2. ed., 15. print. Upper Saddle River, NJ: Prentice Hall PTR, 2009. ISBN: 978-0-13-629155-8.

[34] Barbara H. Liskov und Jeannette M. Wing. "A behavioral notion of subtyping". In: *ACM Transactions on Programming Languages and Systems* 16.6 (1994), S. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383.

[35] Robert C. Martin. *Agile software development: Principles, patterns, and practices*. Alan Apt series. Upper Saddle River, NJ: Pearson Education, 2003. ISBN: 978-0135974445.

[36] Robert C. Martin. *The Dependency Inversion Principle*. 1996. URL: https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf (besucht am 10. 04. 2022).

[37] Frank Buschmann, Kevlin Henney und Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A pattern language for distributed computing*. Reprinted August 2011. Bd. / Frank Buschmann ... ; Vol. 4. Wiley series in software design patterns. Chichester: Wiley, 2011. ISBN: 978-0470059029.

[38] James Martin. *Managing the data base environment*. 1st. Prentice Hall PTR, 1983. ISBN: 0135505828. (Besucht am 10. 04. 2022).

[39] Sylvia Fronczak. *Layered Architecture: Still a Solid Approach*. URL: https://blog.ndepend.com/layered-architecture-solid-approach/ (besucht am 10. 04. 2022).

[40] hgraca. *DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together*. 2017. URL: https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/ (besucht am 10. 04. 2022).

[41] Alistair Cockburn. *Hexagonal architecture*. 2005. URL: https://alistair.cockburn.us/hexagonal-architecture/ (besucht am 10. 04. 2022).

[42] Jesse Griffin. "Hexagonal-Driven Development". In: *Domain-Driven Laravel*. Hrsg. von Jesse Griffin. Berkeley, CA: Apress, 2021, S. 521–544. ISBN: 978-1-4842-6022-7. DOI: 10.1007/978-1-4842-6023-4{\textunderscore}17.

[43] philipbrown. *What is Hexagonal Architecture?* 2014. URL: https://www.culttt.com/2014/12/31/hexagonal-architecture/ (besucht am 10. 04. 2022).

[44] Erwan Alliaume und Sébastien Roccaserra. *Hexagonal Architecture: three principles and an implementation example*. 2018. URL: https://blog.octo.com/hexagonal-architecture-three-principles-and-an-implementation-example/ (besucht am 10. 04. 2022).

[45] Pablo Martinez. *Hexagonal Architecture, there are always two sides to every story*. 2021. URL: https://medium.com/ssense-tech/hexagonal-architecture-there-are-always-two-sides-to-every-story-bc0780ed7d9c (besucht am 10. 04. 2022).

[46] Mark Seemann. *Layers, Onions, Ports, Adapters: it's all the same*. 2013. URL: https://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/ (besucht am 10. 04. 2022).

[47] Brian Foote und Joseph Yoder. *Big Ball of Mud*. 1999. URL: http://www.laputan.org/mud/ (besucht am 10. 04. 2022).

[48] Alberto Brandolini. *About Team Topologies and Context Mapping*. 2021. URL: https://blog.avanscoperta.it/2021/04/22/about-team-topologies-and-context-mapping/ (besucht am 10. 04. 2022).

[49] Lev Gorodinski. *Services in Domain-Driven Design (DDD)*. 2012. URL: http://gorodinski.com/blog/2012/04/14/services-in-domain-driven-design-ddd/ (besucht am 19. 04. 2022).

[50] Vaughn Vernon. *Effective Aggregate Design Part I: Modeling a Single Aggregate*. 2011. URL: https://www.dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf (besucht am 20. 04. 2022).

[51] Martin Fowler und James Lewis. *Microservices: a definition of this new architectural term*. 2014. URL: https://martinfowler.com/articles/microservices.html (besucht am 22. 04. 2022).

[52] *Atomicity and Transactions*. URL: https://www.mongodb.com/docs/manual/core/write-operations-atomicity/ (besucht am 10. 04. 2022).

[53] Goetz Graefe. *Revisiting optimistic and pessimistic concurrency control*. 2016. URL: https://www.labs.hpe.com/techreports/2016/HPE-2016-47.pdf (besucht am 22. 04. 2022).

[54] Girish Suryanarayana, Ganesh Samarthyam und Tushar Sharma. *Refactoring for software design smells: Managing technical debt*. Software development and engineering. Amsterdam: Morgan Kaufmann an imprint of Elsevier, 2015. ISBN: 978-0-12-801397-7.

[55] Ekaterina Razina und David Janzen. "Effects of Dependency Injection on Maintainability". In: *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*. SEA '07. Cambridge, Massachusetts: ACTA Press, 2007, 7–12. ISBN: 9780889867062.

[56] Ricardo Lindooren. *Testability of Dependency injection: An attempt to find out how the testability of source*. 2007. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.1999&rep=rep1&type=pdf (besucht am 23. 04. 2022).