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

Bachelor-Arbeit

Simon Thalmaier

Erstprüfer	-
Zweitprüfer	-
Betreuer	Sebastian Apel
Ausgabedatum	-
Abgabedatum	-

Angaben zum Autor oder Vergleichbares.

Dokumenteninformation, Veröffentlichung, Rahmen, bibliographisches Angaben

Abstract

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

Zusammenfassung

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

Danksagung

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

Inhaltsverzeichnis

1 Einleitung

Der Schwerpunkt dieses Projekts und somit zugleich dieser Bachelorarbeit ist die Entwicklung eines Proof-of-Concept (POC) einer Checkout-Software mithilfe von Domain-Driven Design und Hexagonaler Architektur. In den folgenden Kapiteln wird diese Problemstellung detaillierter beschrieben, das vorliegende Projektumfeld aufgezeigt, sowie die dahinterliegende Motivation und Ziele erläutert. Hierdurch soll ein grundlegendes Verständnis der Hintergründe dieser Arbeit geschaffen werden.

1.1 Problemstellung

In einem Onlineshop ist ein elementarer Bestandteil der sogenannte Warenkorb. In diesem können unter anderem Waren abgelegt werden, um sie eventuell zu einem späteren Zeitpunkt zu erwerben. Die Kernfunktionen eines Warenkorbs umfasst demnach das Hinzufügen bzw. Löschen von Artikeln und das Abändern ihrer Stückzahl. Im verlaufe des Kaufprozess sollte es weiterhin möglich sein eine Versandart einzustellen, Kundendaten zu hinterlegen und eine Zahlungsart auszuwählen. Nach erfolgreicher Überprüfung von Sicherheitsrichtlinien findet die Kaufabwicklung statt, der sogenannte 'Checkout'. Um die hier beschriebenen Anwendungsfälle zu verwirklichen, wird eine dafür designierte Software benötigt. In diesem Projekt wird diese realisiert und als eine 'Checkout-Software' bezeichnet.

Eine solche Anwendung stellt das Rückgrat des Onlineshops dar. Sie erfährt stetige Änderungen, besitzt eine Vielzahl von Businessanforderungen und ihre Einbindung in das Frontend beeinflusst weitergehend auch das Kundenerlebnis. Dadurch liegt ein hoher Fokus auf Qualitätsmerkmale, wie Stabilität, Wartbarkeit und Testbarkeit. Der Checkout-Prozess, welcher durch die Software abgewickelt wird, muss für alle relevanten Länder und ihre individuellen gesetzlichen Voraussetzungen fiskalisch korrekt ausgeführt werden. Jederzeit können Länder und dadurch eventuell neue Richtlinien hinzugefügt werden, wodurch umso mehr ein Fokus auf Flexibilität des Systems gelegt wird. Weiterhin können hinzukommende Businessprozesse eine Anpassung des Modells erfordern, dadurch wird ein anpassbares Datenkonstrukt benötigt, welches zugleich performant und wartbar bleibt. Das Datenmodell bestimmt weitestgehend wie die internen Softwarekomponenten und die externen Systeme mit der Anwendung interagieren. Daher hat eine Umfeldanalyse und klare Definition von Anwendungsfällen hohen Einfluss auf die resultierende Qualität des Entwicklungsprozesses. Die verwendete Architektur und das Softwaredesign muss somit Entwicklern dabei unterstützen diese Kriterien zu erfüllen.

Durch welche Entwurfsmuster und Vorgehensweise eine solche Software realisiert werden kann stellt die initiale Problemstellung des Projektes dar. Hierbei wurde aufgrund der im nachfolgenden Kapitel stattfindenden Analyse eine Hexagonale Architektur und ein Domain-Driven Ansatz ausgewählt. Weitergehend liegt der Hauptfokus hierbei darauf, wie ein konkreten Aggregationsschnitt des Datenmodells, basierend auf die vorliegende Systemumgebung und erforderlichen Anwendungsfällen des Proof-of-Concepts, die vorher genannten Bedingungen erfüllt.

1.2 Projektumfeld

- MediaMarktSaturn
- Currently Checkout-Software exist
- Vor bzw Nachgelagerte Systeme (CAR (Product Data?), Pricing API, Webshop BOS, XPay, Campaign Promotion CRM, Mirakl, Customer Order Service (COS)) [(aroma (FOM?), WCS, DMC, SAP)]

1.2.1 Das Unternehmen MediaMarktSaturn

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

Als größte Elektronik-Fachmarktkette Europas bietet MediaMarktSaturn Kunden in über 1023 Märkten eine Einkaufmöglichkeit einer Vielzahl von Waren. Die Marktzugehörigkeit ist hierbei unterteilt in den Marken *Media Markt* und *Saturn*.

Über die Jahre gewann der Onlineshop für Media Markt und Saturn an zunehmender Bedeutung, da die prozentuale Verteilung des jährlichen Gewinns in den Märkten zurückgegangen und im Onlineshop gestiegen ist. Dadurch wurden die Unternehmensziele dementsprechend auf die Entwicklung von komplexer Software zur Unterstützung des Onlineshops neu ausgelegt. Der MediaMarktSaturn Retail Group unterteilte Firma *MediaMarktSaturn Technology* ist hierbei verantwortlich für alle Entwicklungstätigkeiten und die X Mitarbeiter bündelt die technischen Kompetenzen des Unternehmens am Standort Ingolstadt.

Die Entwicklung des POCs wurde im Scrum-Team *Chechkout & Payment* durchgeführt. Die sieben zugehörigen Teammitglieder sind zuständig einen unternehmensweiten universellen Checkout für alle Länder bereitzustellen, sowohl für den Onlineshops als auch im Markt oder per Handyapp.

1.2.2 Benachbarte Systeme der Checkout-Software

1.3 Motivation

Durch den stetigen Anstieg an Komplexität von Softwareprojekten haben sich gängige Designprinzipien und Architekturstile für den Entwicklungsprozess von Software etabliert, um auch weiterhin die vielen Businessanforderungen in einem zukunftssicheren Ansatz zu realisieren. Eine Checkout-Software beinhaltet multiple Prozessabläufe, welche jederzeit angepasst und erweitert werden können. Dadurch ist eine flexible Grundstruktur entscheidend, um die Langlebigkeit der Software zu gewährleisten. Da der Checkout ein wichtiger Bestandteil eines jeden Onlineshops ist, besitzt die Software für MediaMarktSaturn eine zentraler Bedeutung. Folglich ist eine sorgfältige Projektplanung und stetige Revision der bestehenden Software relevant, um auch weiterhin einen reibungslosen Ablauf der Geschäftsprozesse zu ermöglichen. Zur Erreichung dieses Ziels verwendet die zum aktuellen Zeitpunkt bestehende Anwendung Domain-Driven Design und eine Hexagonale Architektur. Der erste Abschnitt dieser Arbeit beschäftigt sich mit der Entscheidung, ob auch weiterhin bzw. warum ein solcher Aufbau verfolgt werden sollte, um die aktuelle Lösung nach Verbesserungsmöglichkeiten zu überprüfen.

Zudem existieren aufgrund des zugrundeliegenden Aggregationsschnitts Performance-Einbusen. In diesem Projekt wird analysiert, ob die Performance durch einen anderen Aufteilung des Datenmodells

und einem vertretbaren Aufwand gesteigert werden kann. Dies dient ebenfalls als nützliche Untersuchung der bestehenden Anwendung und kann als Reverenz für zukünftige Softwareprojekte verwendet werden, da viele Projekte mit ähnlichen Problemstellungen konfrontiert sind.

1.4 Ziele

Aus den vorhergehenden Motivationen lassen sich folgenden Projektziele ableiten. Grundlegend stellt diese Arbeit eine Referenz für neue Softwareprojekte und Mitarbeiter dar. Dies kann zu einem erhöhten Grad an Softwarequalität im Unternehmen beitragen. Zugleich wird, durch die Analyse und Durchführung des Proof-of-Concepts, das bestehende Softwaredesign überprüft und herausgefordert. Dadurch kann ein mögliches Fazit der Arbeit sein, dass die aktuelle Architektur die erwünschten Merkmale nicht erfüllen oder womöglich sich keine Verbesserungsvorschläge ergeben. Letzteres stellt dennoch eine wichtige Erkenntnis für das Team und Unternehmen dar, da zukünftige Projekte mithilfe der verwendeten Vorgehensmodelle ähnliche Ergebnisse erzielen können. Sollten sich durch einen anderen Aggregationsschnitt Vorteile bilden, kann eine Folge dem Umbau der Software entsprechen.

2 Grundlagen

Zur erfolgreichen Durchführung dieses Projekts werden Kernkompetenzen der Softwareentwicklung vorausgesetzt. Diese beschäftigen sich weitestgehend mit Softwaredesign und Architekturstilen. Um zu verstehen, wie eine Architektur die Programmierer bei der Entwicklung einer Software unterstützt, muss zunächst festgelegt werden, welche Eigenschaften Quellcode erfüllen sollen, damit dieser positive Qualitätsmerkmale widerspiegelt. Hierzu wurden gängige Designprinzipien über die Jahre festgelegt, unter anderem die SOLID-Prinzipien, welche im nächsten Abschnitt erläutert werden.

2.1 SOLID-Prinzipien

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

- **Single-Responsibility-Prinzip:** Jede Softwarekomponente darf laut SRP nur eine zugehörige Aufgabe erfüllen. Eine Änderung in den Anforderungen muss somit eine Anpassung in einer einzelnen Komponente darstellen. Dies erhöht stark die Kohäsion der Software und lindert die Wahrscheinlichkeit von gewünschten Nebeneffekten bei Codeanpassungen.
- **Open-Closed-Prinzip:** Zur Sicherstellung, dass eine Änderung in einer Komponente keine Auswirkung auf eine andere besitzt, definiert das OCP Komponente als geschlossen gegenüber Veränderungen aber offen für Erweiterungen. Der erste Teil des Prinzips kann durch ein Interface realisiert werden, welches geschlossen ist, da die existierenden Methoden keine Anpassung erfahren dürfen, da sonst darauf basierender Code angepasst werden müsste. Dennoch können Modifikationen stattfinden, durch das Vererben der Klasse oder die Einbindung von neuen Interfaces.
- **Liskovsches Substitutionsprinzip:** Eine wünschenswerte Eigenschaft der Vererbung ist, dass eine Unterklasse S einer Oberklasse T die Korrektheit einer Anwendung nicht beeinflusst, wenn ein Objekt vom Typ T durch ein Objekt vom Typ S ersetzt wird. Dadurch wird die Fehleranfälligkeit bei einer Substitution im Code erheblich gesenkt und der Client kann sichergehen, dass die Funktionalität auch weiterhin den erwarteten Effekt birgt. Da das LSP sich mit der Komposition von Klassen beschäftigt, ist es für die nachfolgende Analyse vernachlässigbar.
- **Interface-Segregation-Prinzip:** Der Schnitt von Interfaces sollte so spezifisch und klein wie möglich gehalten werden, damit Clients Abhängigkeiten zu mehrere Interfaces besitzen, statt zu einem großen. Dadurch wird die Wiederverwendbarkeit und Austauschbarkeit der Komponente gewährleistet.

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

Architekturen und Kompositionen können anhand dieser Prinzipien bewertet werden, zu welchem Grad sie in dem Modell verankert sind. Diese Vorgehensweise wurde ebenfalls verwendet, um die nachfolgenden Architekturstile miteinander zu vergleichen.

2.2 Architekturmuster

Eine Softwarearchitektur beschreibt die grundlegende Struktur der Module, Relationen zueinander und den Kommunikationsstil. Die Wahl der verwendeten Architektur beeinflusst somit die komplette Applikation und ihre Qualitätsmerkmale. Die zu bevorzugende Struktur ist stark abhängig von den Anwendungsfällen und existierenden Anforderungen.

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

2.2.1 Schichtenarchitektur

Durch die Einteilung der Softwarekomponenten in einzelne Schichten wird eine fundamentale Trennung der Verantwortlichkeiten und ihre Domänen erzwungen. Die Anzahl der Schichten kann je nach Anwendungsfall variieren, liegt jedoch meist zwischen drei und vier Ebenen. Die meistverbreitete Variante beinhaltet die Präsentations-, Business- und Datenzugriffsschicht. Der Kontrollfluss der Anwendung fließt hierbei stets von einer höheren Schicht in eine tiefere gelegene oder innerhalb einer Ebene zwischen einzelnen Komponenten. Ohne eine konkrete Umkehrung der Abhängigkeiten ist der Abhängigkeitsgraph gleichgerichtet zum Kontrollflussgraph. Hierbei dient Abbildung 2.1 als eine beispielhafte Darstellung einer solchen Architektur.

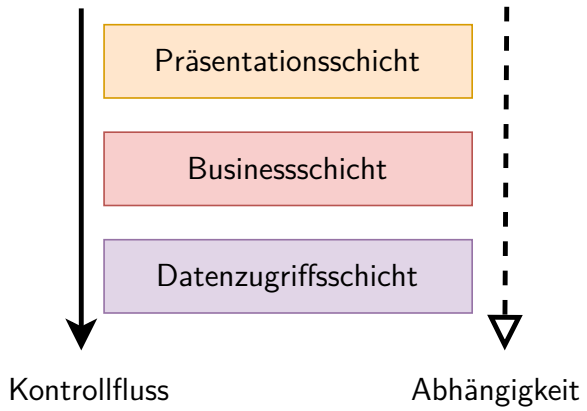

Abbildung 2.1: Beispielhafte Darstellung einer Drei-Schichtenarchitektur

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

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

2.2.2 Hexagonale Architektur

Durch weitere architektonische Einschränkungen können Entwickler zu besseren Softwaredesign gezwungen werden, ohne dabei die Implementierungsmöglichkeiten einzuengen. Dieser Denkansatz wird in der von Alistair Cockburn geprägten Hexagonalen Architektur angewandt, indem eine klare Struktur der Softwarekomposition vorgegeben wird. Hierbei existieren drei Bereiche in denen die Komponenten

angesiedelt sein können, namentlich die primären Adapter, der Applikationskern und die sekundären Adapter.

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

Unter den Adaptern fallen jegliche Komponenten, welche als Schnittstellen zwischen externen Systemen und der Geschäftslogik dienen. Dabei sind die primären Adapter jeglicher Code, welche durch externe Systeme angestoßen wird und hierbei den Steuerfluss in den Applikationskern trägt. Diese externen Systemen, wie Benutzerinterfaces, Test-Engines und Kommandokonsolen, können beispielsweise einen Methodenaufruf initiieren, welcher durch die primären Adapter verarbeitet werden. Andererseits bilden alle Komponenten, welche den Steuerfluss von dem Applikationskern zu externen Systemen tragen, die sekundären Adapter. Hierbei entsteht der Impuls im Vergleich zu den primären Adaptern nicht außerhalb der Applikation sondern innerhalb. Die, von den sekundären Adaptern angesprochenen Systemen, können beispielsweise Datenbanken, Massage-Broker und jegliche im Prozess tieferliegende Software sein.

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

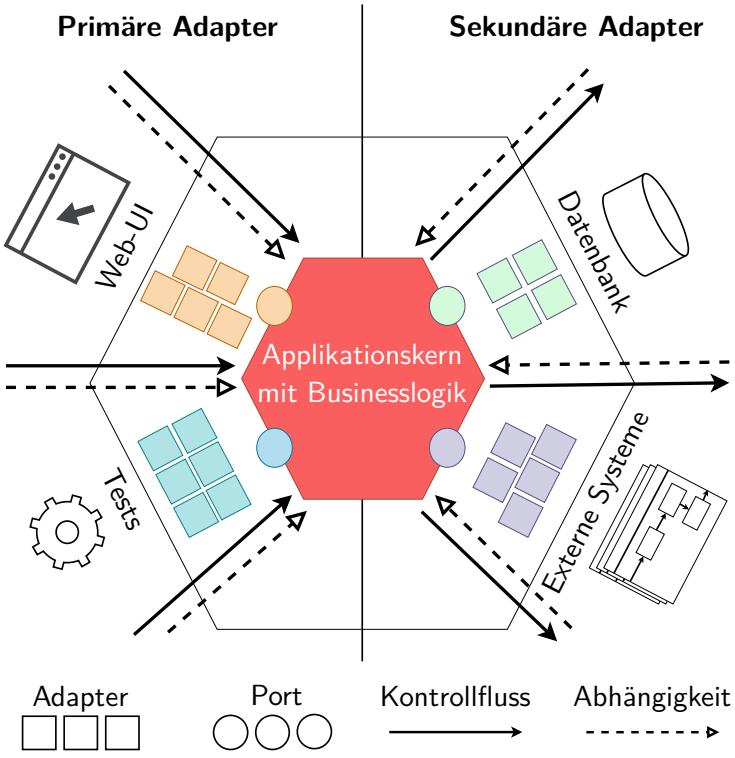

Abbildung 2.2: Grundstruktur einer Hexagonalen Architektur

Das Speichern von Daten ist ein simpler Anwendungsfall, welcher im folgenden beispielhaft in einer hexagonalen Applikation dargestellt wird. Die Daten, welche von einer Website an die Anwendung geschickt werden, initiieren den Steuerfluss in einem sogenannten Controller. Dieser ist den primären Adaptern zugeteilt und erledigt Aufgaben, wie Authentifizierung, Datenumwandlung und erste Fehlerbehandlungen. Über einen entsprechenden Port wird der Applikationskern mit den übergebenen Daten angesprochen. Innerhalb werden alle business-relevanten Aufgaben erfüllt. Darunter fallen das logische Überprüfen der Daten anhand von Businessrichtlinien, Erstellen neuer Daten und Steuerung des Entscheidungsflusses. In diesem Anwendungsfall sollen die Daten in einer Datenbank abgespeichert werden. Dementsprechend wird aus dem Anwendungskern über einen Port ein sekundäre Adapter aufgerufen, welcher für das Speicher von diesen Daten in einer Datenbank zuständig ist.

Durch diesen Aufbau wird eine, im Vergleich zur Schichtenarchitektur, strengere konzeptionelle Trennung der Verantwortlichkeiten erzwungen. Dies wirkt sich positiv auf die Einhaltung des *Single-Responsibility-Prinzips* aus. Zusätzlich wird die Anwendung des *Open-Closed-Prinzips* und *Interface-Segregation-Prinzips* erleichtert, durch die Einführung von Ports zwischen den Applikationskern und den, für das business-irrelevanten, Komponenten. Für erweiterbare Software ist das *Dependency-Inversion-Prinzip* von großer Bedeutung, da viele der vorherigen genannten Qualitätsmerkmalen beeinflusst

2.3 Domain-Driven Design

Bei der Entwicklung von Software, welche mehr als triviale Anwendungsfälle einer CRUD-Anwendung erfüllen soll, besteht stets die Gefahr bei steigender Anzahl von Anforderungen und Änderungen zu einem sogenannten 'Big Ball of Mud' zu degradieren. Die bestehende Architektur wird undurchschaubar, die Entstehungschancen für Bugs steigen und Businessanforderungen finden sich überall verteilt in der Anwendung. Somit kann die Wartbarkeit der Software nicht mehr gewährleistet werden und ihre Langlebigkeit ist stark eingeschränkt. Die oben analysierten Architekturstile können bei strikter Einhaltung dieses Risiko einschränken, jedoch bestimmen sie nur begrenzt wie das zugrundeliegende Datenmodell und die damit verbundenen Komponente designt werden sollen. In dem Buch *Domain-driven design: Tackling complexity in the heart of software* hat Eric Evans im Jahre 2003 zu diesem Zweck Domain-Driven Design entwickelt. Grundlegend wird durch diese Herangehensweise die Businessprozesse in den Vordergrund gerückt, der Problemraum in Domains eingeteilt und Richtlinien für das Design von dem Domainmodell festgelegt. Domain-Driven Design, kurz DDD, ist nicht gebunden an die darunterliegende Architektur oder verwendeten Technologien und folglich an verschiedenste Einsatzgebiete anpassbar.

Bevor die Bestandteile von DDD bestimmt werden können, sollte zu Beginn eine ausführliche Umfeldanalyse der Problemebene durchgeführt werden, um festzulegen welche Verantwortungen in den zu bestimmenden Bereich fallen. Somit wird unser Problemraum als eine *Domain* aufgespannt. Der Domainschnitt ist hierbei entscheidend, da basierend auf der Domain die dazugehörigen *Subdomains* und ihre *Bounded Contexts* bestimmt werden. Eine Subdomain bündelt die Verantwortlichkeiten und zugehörigen Anwendungsfälle in eine spezifischeren Bereich der Domain. Zur Bestimmung der Subdomains wird der Problemraum stets aus Businesssicht betrachtet und technische Aspekte werden vernachlässigt. Sollte die Domain zu groß geschnitten sein, sind dementsprechend die Subdomains ebenfalls zu umfangreich. Dadurch ist die Kohäsion der Software gefährdet und führt über den Lauf der Entwicklungsphase zu architektonischen Konflikten. Sollte eine Subdomain multiple Verantwortlichkeiten tragen, kann diese Subdomain weiter in kleinere Subdomains eingeteilt werden. Für einen

Domain-Driven Ansatz ist es entscheidend die Definitionsphase gewissenhaft durchzuführen, damit eine stabile Grundlage für die Entwicklung geboten werden kann.

Als Ausgangspunkt für die Bestimmung der Lösungsebene, werden die Subdomains in sogenannte Bounded-Contexts eingeteilt. Ein Bounded-Context kann eine oder mehrere Subdomains umfassen und bündelt ihre zugehörigen Aufgabengebiete. Wie es in der Praxis häufig der Fall ist, können Subdomains und Bounded-Context durchaus identisch sein. In jedem Bereich sollte nur ein Team agieren, um Konflikte zu vermeiden. Andernfalls kann dies ein Indiz sein, dass die Subdomains zu groß geschnitten worden sind. Jeder Bounded-Context besitzt zudem eine zugehörige Ubiquitous Language. Die Festlegung der *Ubiquitous Language* stellt einen wichtigen Schritt in deinem Domain-Driven Ansatz dar. Diese definiert die Bedeutung von Begriffen, welche durch die Stakeholder und das Business verwendet werden, eindeutig. Dadurch können Missverständnisse in der Kommunikation zwischen dem Business und den Entwicklern vorgebeugt und eventuelle Inkonsistenzen aufgedeckt werden. Der größte Vorteile ergibt sich allerdings, sobald auch das Datenmodell diese Sprache widerspiegelt. Entities können Nomen darstellen, Funktionen können Verben realisieren und Aktionen können als Event verwirklicht werden. Somit sind Businessprozesse auch im Quelltext wiederzufinden. Folglich steigert dies die Verständlichkeit und Wartbarkeit der Software. Zudem lassen sich Testfälle und Anwendungsfälle leichter definieren und umsetzten. Wichtig ist, dass diese Sprache nur innerhalb eines Bounded-Context gültig ist. Beispielhaft kann der Begriff 'Kunde' in einem Onlineshop einen zivilen Endkunden, jedoch im Wareneingang eine Lieferfirma beschreiben. Daher ist bei der Kommunikation zwischen Teams unterschiedlicher Subdomains zu beachten, dass Begriffe eventuell unterschiedliche Bedeutung besitzen.

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

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

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,

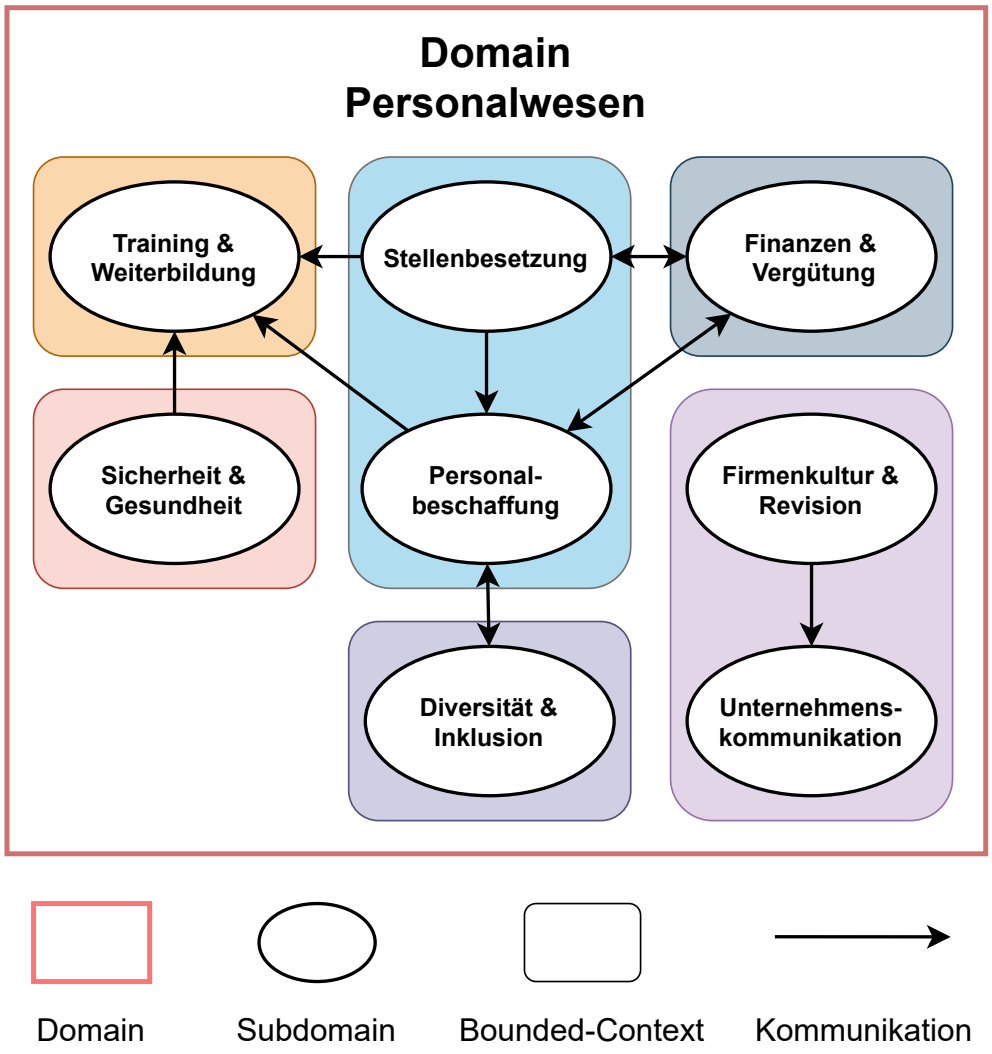

Abbildung 2.3: Beispiel einer Context-Map anhand des Personalwesens

wie zum Beispiel die Erstellung von Objekten, die Modifikation ihrer Attribute oder die dauerhafte Speicherung. Für diesen Zweck existieren in Domain-Driven Design mehrere Arten von Komponente, welche anhand ihrer Verantwortlichkeiten zugeordnet werden. Die korrekte Zuordnung der Klassen und ihrer Rollen in DDD ist entscheidend für ein skalierbares Modell, daher wird in den folgenden Unterkapiteln ein zentraler Überblick über die einzelnen Bestandteile ausgeführt.

2.3.1 Value Object

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

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

Aus diesen Grund gelten Value Objects als immutable, da sie selbst keinen Werteverlauf besitzen. Dies gilt als positives Designmuster, da unveränderbare Objekte eine erhöhte Wiederverwendbarkeit genießen und unerwünschte Seiteneffekte unterdrücken. Weiterhin ist ein Resultat aus dieser Richtlinie, dass sie selbst keinen Lebenszyklus besitzen, jegliche eine Momentaufnahme des Applikationszustandes darstellen.

2.3.2 Entity

Im Gegenzug zu einem Value Object wird eine Entity nicht durch ihre Werte identifiziert, sondern behalten bei Wertanpassung die gleiche Identität bei. Folglich gelten hier zwei Entities mit einer identischer Wertbelegung als ungleich und repräsentieren unterschiedliche Objekte.

Eine Klasse *Kunde* stellt einen guten Vertreter dieser Kategorie dar. In vielen Bounded-Contexts wird ein Kunde durch eine eindeutige Id ausgewiesen. Somit sind zwei Kunden mit gleichen Namen dennoch nicht die gleichen Personen. Sollte eine Person einen Namenswechsel erfahren, muss das zugehörige Objekt ebenfalls aktualisiert werden.

Daher folgt auch der Unterschied zu den Value Objects, dass eine Entity durchaus einen Lebenszyklus aufweisen. Sie werden erstellt, erhalten Anpassungen und irgendwann vom System wieder gelöscht.

2.3.3 Aggregate

Innerhalb des Bounded-Context ist ein Aggregate ein Verbund aus Entities und Value Objects, welches von außen als eine einzige Einheit wahrgenommen wird. Externe Komponente dürfen ein Aggregat nur durch das sogenannte Aggregate Root referenzieren und nicht direkt auf enthaltene Objekte zugreifen. Es stellt die Schnittstelle zwischen dem Aggregate und der Außenwelt dar.

Im Bereich der Personalverwaltung ist ein mögliches Aggregat das des *Mitarbeiter*s. Das Aggregat Root ist die Klasse *Mitarbeiter* selbst. Diese beinhaltet Value Objects, wie *Gehalt* und *Abteilung*. Zu

beachten ist, dass abhängig vom jeweiligen Bounded-Context zum Beispiel der Werteverlauf des Gehalt dieses Mitarbeiters vielleicht relevant ist und dementsprechend als eine Entity realisiert werden kann.

Um ein effektives Design der Aggregates zu gewährleisten, wurden einige Einschränkungen an Aggregates beschlossen. Unter anderem müssen Businessanforderungen bzw. Invarianten der enthaltenen Objekte vor und nach einer Transaktion erfüllt sein. Dadurch sind die Aggregates-Grenzen durch den minimalen Umfang der transaktionalen Konsistenz ihrer Komponente gesetzt. Zur Folge dessen, wird für jedes Select oder Update eines Datensatzes immer das komplette Aggregat aus der Datenbank geladen. Große Aggregates leiden aus diesem Grund an eingeschränkter Skalierbarkeit und Performance. Eine weitere Bedingung ist, das pro Transaktion jeweils nur ein Aggregat bearbeitet werden darf. Dies schränkt umfangreichere Aggregate durch fehlende Parallelität weiter ein. Bei Anwendung der letzteren Regel an die Mitarbeiter-Klasse, wäre es nicht möglich das Gehalt und die Abteilung durch zwei unterschiedliche Personalmitarbeiter anzupassen, da eine Transaktion der beiden Änderungen auf einen veralteten Stand operiert und zur Vermeidung eines Lost Updates zurückgerollt werden muss.

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

2.3.4 Applicationservice

Sofern Funktionalitäten innerhalb der Domain nicht eindeutig einer oder mehreren Entities bzw. Wertobjekt zugewiesen werden können, werden konzeptionell zusammenhängende Aufgaben in einem Domainservice gebündelt. Um Seiteneffekte durch Zustandsänderungen zu vermeiden halten Service allgemein keinen eigenen Zustand.

2.3.5 Domainservice

Ähnlich zu den Domainservices sind Applicationservices zur zustandslosen Bereitstellung von Funktionen zuständig. Hierbei ist das Unterscheidungsmerkmal, dass sie kein Kenntnisse über Wissen der Domain besitzen dürfen.

2.3.6 Factory

Die wiederholten Erstellung von komplexen Objekten kann in eine Factory ausgelagert werden, um eine erhöhte Wiederverwendbarkeit zu erreichen.

2.3.7 Repository

Der Datenzugriff mittels einer Datenbank wird durch ein Repository ermöglicht. Dadurch werden die konzeptionelle Abhängigkeiten zur Datenbank von der Domain getrennt. Daher sollte generell die Kommunikation zu Repositories über ein fest definiertes Interface geschehen.

3 Planungs- und Analysephase

Einleitend werden Struktur, Motivation und die abgeleiteten Forschungsfragen diskutiert.

3.1 Umfeldanalyse

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

3.2 Anwendungsfälle

- Basket erstellen und abrufen
- Basket finializing/cancel
- Produkte hinzufügen, löschen und Quantity ändern
- Lieferdaten abändern bzw hinzufügen ?
- Caching Funktion für Product und Price

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

3.3 Aktuelles Design der Produktivanwendung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die

Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

4 Erstellung eines Proof of Concepts

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

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

4.1 Festlegung und Analyse der Domain

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

4.2 Abgrenzung des Bounded Contexts

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

4.3 Festlegen einer Ubiquitous Language

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

In Zusammenarbeit mit den Lead-Developer und Product Owner des Teams wird im folgenden Abschnitt die Ubiquitous Language definiert, um ein solches Verständnis über den Checkout Bounded-Context zu gewährleisten. Hierbei wurde sich auf die, für dieses Projekt, relevanten Terme beschränkt. Eingeklammerte Wörter beschreiben Synonyme zu dem vorangestellten Ausdruck.

Domain-Modell:

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

Prozesse:

- **Basket Validation:** Durchführung einer Validierung des Baskets auf Inkonsistenzen oder fehlenden, notwendigen Werten.
- **Basket Finalization:** Überprüfung der Korrektheit des Baskets durch die Basket Validation, Sperrung des Baskets vor weiteren Abänderungen, Einleitung des Zahlungsprozesses und Reservierung der Produkte im Basket.
- **Basket Cancellation:** Stornieren des zugehörigen Baskets. Kann nur auf einen nicht geschlossenen oder stornierten Basket ausgeführt werden. Nach Cancellation dürfen keine weiteren Änderungen an den Basket durchgeführt werden.

- **Basket Creation:** Explizite oder Implizite Erstellung eines neuen Baskets. Findet automatisch statt sofern noch kein Basket für den Customer existiert. Dies ist ebenfalls der Fall nach einer Basket Finalization.
- **Basket Calculation:** Das Berechnen der Geldbeträge des Baskets. Beinhaltet die Auswertung aller Products mitsamt ihren Net Amounts und VATs. Der resultierende Betrag aus unterschiedlichen VATs muss weiterhin aus rechtlichen Gründen einzeln verwiesen werden können.
- **High Volume Ordering:** Die Bestellung von Artikeln in hoher Stückzahl. Aufgrund von Businessanforderungen soll es nur begrenzt möglich sein, dass ein Kunde in einem Basket oder wiederholt das gleiche Produkt mehrfach kauft.

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

4.4 Definition der Value Objects

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

Anfangs wird das Daten-Modell mithilfe dem erlangten Domainwissen grundlegend aufgebaut. Die Ubiquitous Language unterstützt bei der richtigen Benamung der Klassen. Der Basket stellt den Ausgangspunkt der Modellierung dar. Folgende Attribute werden benötigt:

Basket:

- **OutletId:** Zur Referenz in welchem Markt oder Onlineshop der Warenkorb angelegt wurde. Unerlässlich für die Bestimmung von unter anderem Lagerbeständen, Lieferzeiten, Fulfillment-Optionen und Versandkosten. Wir benötigen nur die Id des Outlets und nicht alle Daten, da sie für unsere Kommunikationspartner nicht relevant sind.
- **Customer:** Speichert Kundendaten.
- **BasketItems:** Liste an allen Artikeln aktuell im Warenkorb.
- **CalculationResult:** Beinhaltet die berechneten Werte des Warenkorbs, wie Nettobetrag, Bruttobetrag und VAT. Die Speicherung dieser Werte wäre technisch nicht notwendig, spart aber an Rechenzeit ein, da nicht bei jeder Abfrage des Basket dieser Wert neu berechnet werden muss.
- **Status:** Repräsentiert, ob der Warenkorb als offen, canceled oder finalized gilt. Wichtig für Businessprozesse und Validierung von Änderungen am Basket.

Anschließend werden die, im Basket enthaltenen, Klassen ebenfalls mit der gleichen Vorgehensweise definiert, sofern sie mehr als nur einfache Strings oder Enumerations darstellen:

Customer:

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

BasketItem:

- **Id:** Eindeutige Nummer des Items in diesem Basket.
- **Product:** Beinhaltet alle Produktinformationen, welche durch die Software benötigt werden.
- **Price:** Aktueller Preis des zugehörigen Products. Kann sich zeitlich ändern, muss daher durch eine Businessfunktion aktualisiert werden, jedoch beeinflusst dies nicht die Modellierung des Datenobjekts.

Product:

- **Id:** Eindeutige Referenz für dieses Product im externen System.
- **Name:** Textuelle Produktbezeichnung des Products.
- **Vat:** Mehrwertsteuerinformationen des Products.

Vat:

- **Sign:**
- **Rate:** Prozentualer Wert der Mehrwertsteuer.

Price:

- **GrossAmount:** Bruttobetrag mit Währung.
- **NetAmount:** Nettobetrag mit Währung.
- **VatAmount:** Mehrwertsteuerklasse des Preises. Benötigt für die Berechnung des Preises.

VatAmount:

- **Sign:**
- **Rate:** Prozentualer Wert der Mehrwertsteuer. item **Amount:** Berechneter Betrag der Mehrwertsteuer zugehörig zu einem Bruttobetrag.

CalculationResult:

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

4.5 Bestimmung der Entities anhand ihrer Identität

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

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

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

- **Basket:** Als zentrales Datenobjekt besitzt ein Basket zur eindeutigen Identifikation eine Id, dies spricht stark für eine Entity. Zusätzlich bestimmen nicht die enthaltenen Attribute wie Products oder der zugehörige Kunde die Identität des Baskets. Aufgrund der geforderten Anwendungsfälle entsteht zugleich ein Lebenszyklus für die Instanzen eines Baskets.
- **Customer:** In Bounded-Contexts, welche mit den Kundendaten operieren, kann diese Klasse durchaus eine Entity darstellen. In unserer Domain finden keine Operationen auf diesen Informationen statt und die vorangestellten Systeme senden bei Änderungen der Kundendaten diese zu. Folglich besitzen sie keinen eigenen Lebenszyklus, werden ebenfalls nicht separat gespeichert und kann weiterhin als Value Object designt werden.
- **BasketItem:**
- **Product:**
- **Price:**
- **CalculationResult:** Als Datenstruktur, welche bei jeder Neuberechnung aktualisiert wird, könnte die Eigenschaft eines Lebenszyklus erfüllt sein. Jedoch stellt die Klasse einzig ein Zwischenspeicher der Ergebnisse dar und ohne den Kontext eines darüberlegenden, zugehörigen Objekt besitzen diese Daten keine Aussagekraft. Weiterhin sind die gleichen Ergebnisse im Sinne der Identität äquivalent.

4.6 Design der Aggregates anhand der Anwendungsfälle

- Eine Einheit
- Behält Integrität
- Um abzuspeichern in Datenbank
- Eine Transaktion darf nur ein Aggregat bearbeiten
- Changes within can be inconsistent for a while until operation is done.

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

4.7 Domainservice

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

4.8 Auslagerung von Funktionen in Applicationservices

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

4.9 Erstellen der Primären und Sekundären Adapter

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige

Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

5 Fazit und Empfehlungen

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

Akronyme

DDD Domain-Driven Design. 9

DIP Dependency-Inversion-Prinzip. 4–7

ISP Interface-Segregation-Prinzip. 4, 7

LSP Liskovsches Substitutionsprinzip. 4

OCP Open-Closed-Prinzip. 4, 6, 7

POC Proof-of-Concept. 1, 2

SRP Single-Responsibility-Prinzip. 4, 6, 7

Glossar

Enumeration Eine Auflistung von konstanten, unveränderlichen Werten. 18

immutable Die Unveränderlichkeit von Werten bzw. Variablen innerhalb einer Klasse. 11

Product Owner Eine Scrum-Rolle, welche die wirtschaftliche Ziele und Prioritäten der Aufgabenpakete bestimmt. 17

Scrum Eine agile Entwicklungsmethode, welches hohen Fokus auf kontinuierliche Verbesserung in einem geregelten Zyklus legt. 2

Literatur

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