



Technische Hochschule
Ingolstadt

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 | 1 |
| 1.1 | Problemstellung | 1 |
| 1.2 | Projektumfeld | 2 |
| 1.2.1 | Das Unternehmen MediaMarktSaturn | 2 |
| 1.2.2 | Benachbarte Systeme der Checkout-Software | 2 |
| 1.3 | Motivation | 2 |
| 1.4 | Ziele | 3 |
| 2 | Grundlagen | 4 |
| 2.1 | SOLID-Prinzipien | 4 |
| 2.2 | Architekturmuster | 5 |
| 2.2.1 | Schichtenarchitektur | 5 |
| 2.2.2 | Hexagonale Architektur | 6 |
| 2.3 | Domain-Driven Design | 8 |
| 3 | Planungs- und Analysephase | 12 |
| 3.1 | Umfeldanalyse | 12 |
| 3.2 | Anwendungsfälle | 12 |
| 3.3 | Aktuelles Design der Produktivianwendung | 12 |
| 4 | Erstellung eines Proof of Concepts | 14 |
| 4.1 | Festlegung und Analyse der Domain | 14 |
| 4.2 | Abgrenzung des Bounded Contexts | 14 |
| 4.3 | Festlegen einer Ubiquitous Language | 15 |
| 4.4 | Design der Wertobjekte | 16 |
| 4.5 | Definition der Entitäten | 16 |
| 4.6 | Aggregatbestimmung anhand der Anwendungsfälle | 16 |
| 4.7 | Wahl der Aggregate Roots | 17 |
| 4.8 | Domänenservice | 17 |
| 4.9 | Applikationsservice | 17 |
| 4.10 | Primäre und Sekundäre Adapter | 17 |
| 5 | Fazit und Empfehlungen | 18 |
| | Akronyme | i |
| | Glossar | i |
| | Literatur | ii |

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

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 Einkaufsmöglichkeit 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 *Checkout & 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 darüberlegenden Module und ihre Relationen zueinander. Die Wahl der verwendeten Architektur beeinflusst somit die komplette Applikation und ihre Qualitätsmerkmale. Die zu bevorzugende Struktur ist stark abhängig von den Anwendungsfall 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 ist geeignet für die Verarbeitung von einer Vielzahl an Daten. Jedoch ist das Abbilden von Entscheidungsstränge und Businessrichtlinien nur umständlich zu verwirklichen. Etablierte Architekturen für Backend-Software, welche die Businessprozesse als Kern der Applikation halten, müssen hingegen genauer untersucht werden. Die Schichtenarchitektur und die 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 bewertet anhand ihrer Tauglichkeit für eine Checkout-Software. Diese Bewertung beinhaltet ebenfalls die native 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.

Das Ziel einer Schichtenarchitektur ist die Entkopplung der einzelnen Schichten voneinander und das Erreichen von geringen Abhängigkeiten zwischen den Komponenten. Dadurch sollen Qualitätseigenschaften

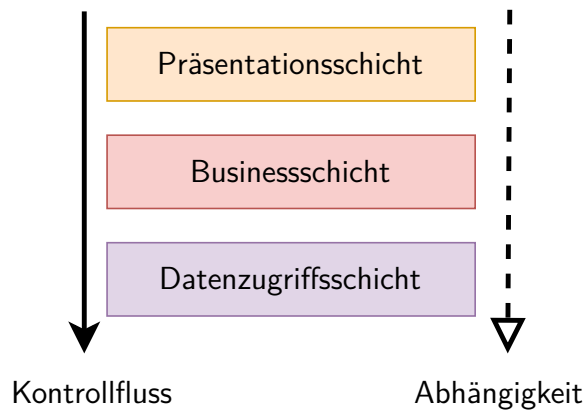


Abbildung 2.1: Beispielhafte Darstellung einer Drei-Schichtenarchitektur

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 entkoppeln, 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 'Create Update Delete', 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, Message-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.

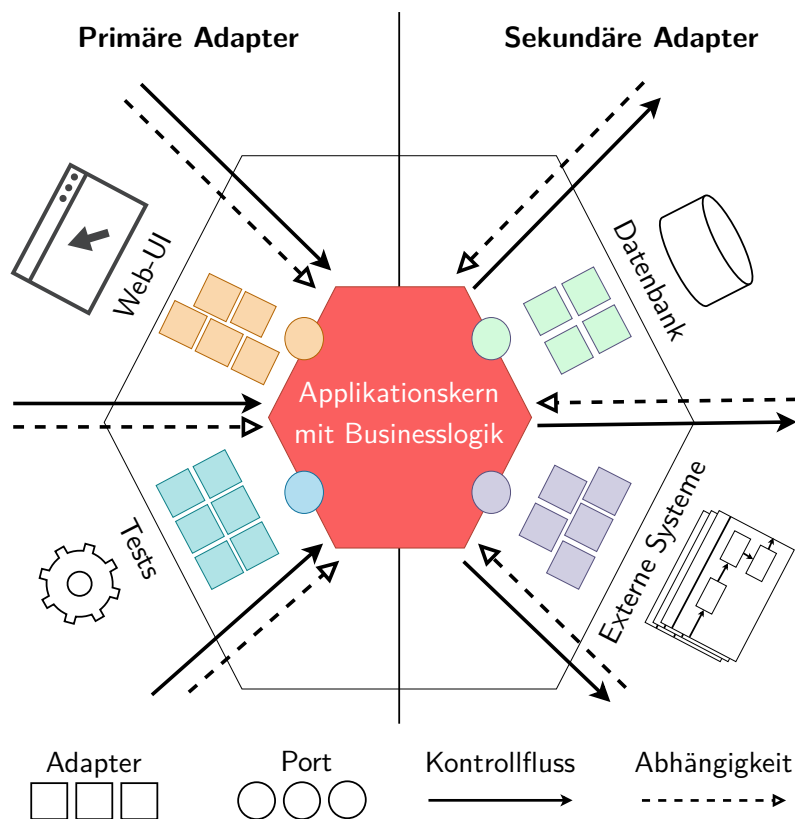


Abbildung 2.2: Grundstruktur einer Hexagonalen Architektur

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 Speichern 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 einen 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. Entitäten 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 umsetzen. 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.

Die Kenntnisse über den Bounded-Context wird als Domain-Modell dargestellt und stellt das Zentrum von DDD dar. Es umfasst sowohl die Datenhaltung als auch das zugehörige Verhalten, wie zum Beispiel die Erstellung von Objekten, die Modifikation ihrer Attribute oder ihre dauerhafte Speicherung. Für diesen Zweck existieren in Domain-Driven Design mehrere Objektgruppierungen, welche anhand ihrer Verantwortlichkeiten zugeordnet werden. Die Gruppen bestehen unter anderem aus:

- **Wertobjekte:** Zusammengehörige Daten, welche durch ihre konkreten Werte identifizierbar sind, werden durch Wertobjekte modelliert. Somit gelten Objekte mit der gleichen Wertbelegung als identisch. Aus Gründen der Wiederverwendbarkeit und zur Unterdrückung von unerwünschten Seiteneffekten gilt es als positives Designmuster Werteobjekte immutable zu gestalten.
- **Entitäten:** Im Gegensatz zu Wertobjekten sind Entitäten nicht durch ihre Werte identifizierbar sondern behalten in ihren Lebenszyklus die gleiche Identität bei. Somit sind zwei Entitäten mit gleichen Werten nicht identisch. Aufgrund dieser Eigenschaften werden Entitäten meist benutzt, um Daten in einer Datenbank zu persistieren und nach diesen zu suchen.
- **Aggregate:** Um Datenänderungen durchzuführen wird ein Zusammenschluss aus Entitäten und Wertobjekten benötigt, um die **transaktionale Integrität** der Daten zu bewahren. Von außen darf ein Aggregat nur durch das sogenannte Aggregate Root referenzieren. Somit gilt das Aggregat Root als Schnittstelle zwischen den externen Komponenten und den inneren Daten

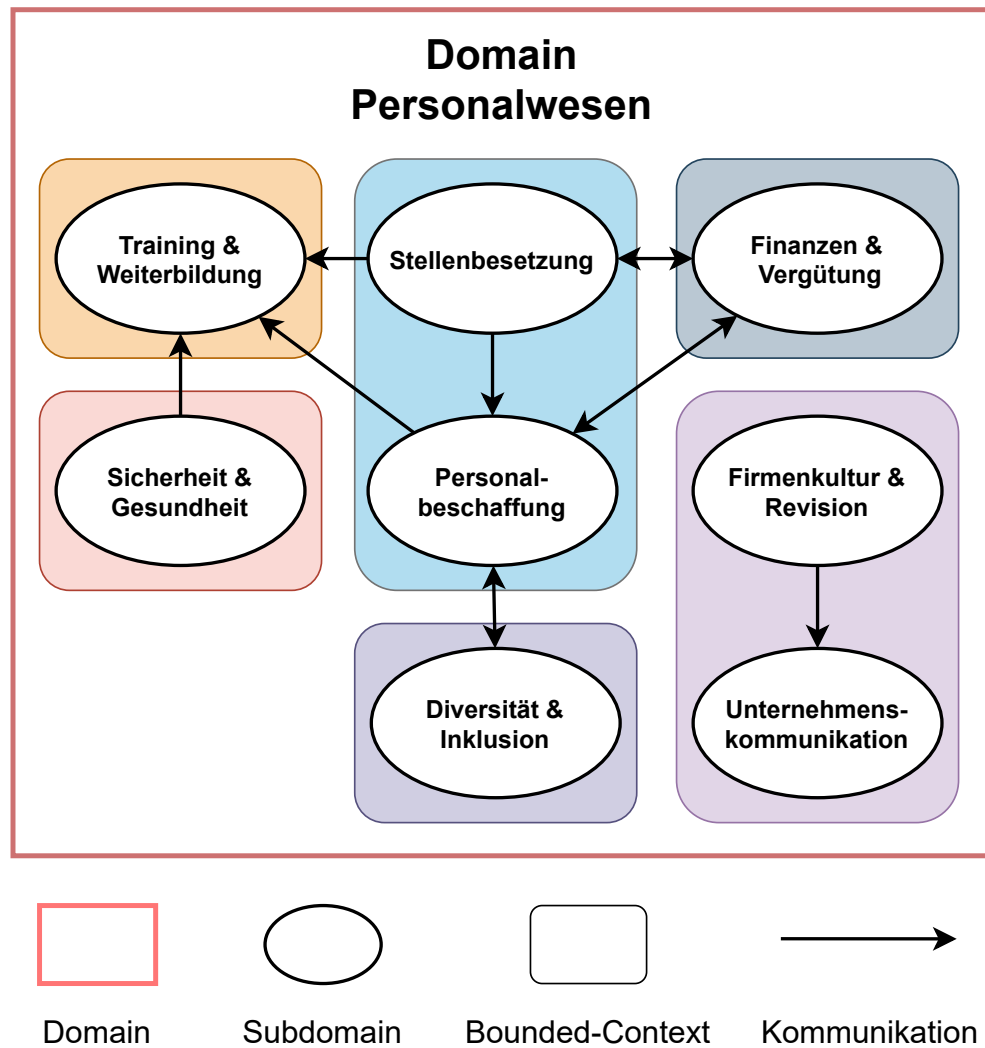


Abbildung 2.3: Beispiel einer Context-Map anhand des Personalwesens

bzw. Funktionen. Aufgrund dessen muss das Wurzelobjekt eine Entität darstellen, um eindeutig in der Datenbank referenziert zu werden. Innerhalb eines Aggregates gilt, dass Anforderungen bzw. Invarianten an die enthaltenen Objekte vor und nach einer Transaktion erfüllt sein müssen.

- **Domänenservice:** Sofern Funktionalitäten innerhalb der Domäne nicht eindeutig einer oder mehreren Entität bzw. Wertobjekt zugewiesen werden können, werden konzeptionell zusammenhängende Aufgaben in einem Domänenservice gebündelt. Um Seiteneffekte durch Zustandsänderungen zu vermeiden halten Service allgemein keinen eigenen Zustand.
- **Applikationsservice:** Ähnlich zu den Domänenservice sind Applikationsservice zur zustandslosen Bereitstellung von Funktionen zuständig. Hierbei ist das Unterscheidungsmerkmal, dass sie kein Domänenwissen besitzen dürfen.
- **Fabriken:** Die wiederholten Erstellung von komplexen Objekten kann in eine Fabrikklasse ausgelagert werden, um eine erhöhte Wiederverwendbarkeit zu erreichen.
- **Repositories:** Der Datenzugriff mittels einer Datenbank wird durch ein Repository ermöglicht. Dadurch werden die konzeptionelle Abhängigkeiten zur Datenbank von der Domäne getrennt. Daher sollte generell die Kommunikation zu Repositories über ein fest definiertes Interface geschehen.

Zusätzlich zum Design des Domain-Modells muss innerhalb eines Bounded-Contexts die grundlegende Architektur durch das zugehörige Team bestimmt werden. 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.

Mithilfe des, in diesem Kapitel erarbeiteten, Wissen wurden die Grundgedanken hinter den verwendeten Prinzipien verdeutlicht und bildet somit ein solides Fundament für die Durchführung dieses Projekts. Im folgendem wird die Planungsphase des Proof-of-Concepts erläutert auf Basis einer Domain-Driven Hexagonalen Architektur.

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 finalizing/cancel
- Produkte hinzufügen, löschen und Quantity ändern
- Lieferdaten abändern bzw hinzufügen

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 Produktivianwendung

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 Missverständnissen 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 umsetzen 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.
- **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, hier Euro.
- **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:**
- **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 aufgedeckt 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 Design der Wertobjekte

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.

4.5 Definition der Entitäten

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.

4.6 Aggregatbestimmung anhand der Anwendungsfälle

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 Wahl der Aggregate Roots

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 Domänenservice

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 Applikationsservice

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.

4.10 Primäre und Sekundäre 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. 8

DIP Dependency-Inversion-Prinzip. 4–8

ISP Interface-Segregation-Prinzip. 4, 8

LSP Liskovsches Substitutionsprinzip. 4

OCP Open-Closed-Prinzip. 4, 6, 8

POC Proof-of-Concept. 1, 2

SRP Single-Responsibility-Prinzip. 4, 6, 8

Glossar

Product Owner Eine Scrum-Rolle, welche die wirtschaftliche Ziele und Prioritäten der Aufgabenpakete bestimmt. 15

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.