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.

Contents

1 Einleitung **1**
1.1 Problemstellung . . . 1
1.2 Projektumfeld . . . 1
1.2.1 Das Unternehmen MediaMarktSaturn . . . 1
1.2.2 Benachbarte Systeme der Checkout-Software . . . 2
1.3 Motivation . . . 2
1.4 Ziele . . . 2

2 Grundlagen **3**
2.1 Architekturmuster . . . 3
2.1.1 Schichtenarchitektur . . . 3
2.1.2 Hexagonale Architektur . . . 5
2.2 Domain-Driven Design . . . 7

3 Planung und Analyse **10**
3.1 Umfeldanalyse . . . 10
3.2 Anwendungsfälle . . . 10
3.3 Aktuelles Design der Produktivanwendung . . . 10

4 Erstellung eines Proof of Concepts **11**
4.1 Analysieren der Domäne . . . 11
4.2 Festlegen einer Ubiquitous Language . . . 11
4.3 Abgrenzung des Bounded Contexts . . . 11
4.4 Design der Wertobjekte . . . 12
4.5 Entitäten . . . 12
4.6 Aggregatbestimmung anhand der Anwendungsfälle . . . 12
4.7 Wahl der Aggregate Roots . . . 12
4.8 Domänenservice . . . 13
4.9 Applikationsservice . . . 13
4.10 Primäre und Sekundäre Adapter . . . 13

5 Fazit und Empfehlungen **14**

1 Einleitung

Anfangs wird die vorliegende Problemstellung erläutert, sowie das Projektumfeld und die dahinterliegende Motivation und ihre Ziele. Hierdurch soll ein grundlegendes Verständnis der Hintergründe dieser Arbeit geschaffen werden.

1.1 Problemstellung

- Alot of Business Rules -¿ future proof architecture
- Modelling the Domain defines how the software and the external systems interact with the software

Ein elementarer Bestandteil eines Onlineshops ist der sogenannte Warenkorb. In diesem können unter anderem Waren, welche eventuell zu einem späteren Zeitpunkt gekauft werden wollen, abgelegt werden. Die Kernfunktionen eines Warenkorbs umfasst das Hinzufügen bzw. Löschen von Waren und das Abändern ihrer Stückzahl. Weiterhin soll es möglich sein eine Versandart einzustellen, Kundendaten zu hinterlegen und eine Zahlungsart auszuwählen. Nach erfolgreicher Überprüfung von Sicherheitskriterien soll abschließend die Kaufabwicklung, der sogenannte 'Checkout', möglich sein. Um die hier beschriebenen Anwendungsfälle zu verwirklichen wird eine dafür designierte Software benötigt. In diesem Projekt wird diese als 'Checkout-Software' bezeichnet.

Zusätzliche Anforderungen an der Software können die Modellierung der Daten beeinflussen, daher muss vor der eigentlichen Implementierung eine sorgfältige Use-Case-Analyse durchgeführt werden. Diese wird in einem späteren Kapitel erläutert.

Ein weiterer Teil der Problemdomäne sind die bereits existierenden Systeme, welche vor- bzw. nach dem Checkout-Prozess liegen. Um eine nahtlose Einbindung der Software zu gewährleisten muss eine Kommunikation mit den zuständigen Entwicklerteams, sowie eine Umfeldanalyse stattfinden. Die erarbeiteten Ergebnisse werden im folgenden Kapitel dargestellt.

1.2 Projektumfeld

- MediaMarktSaturn
- Currently Checkout-Software exist
- Vor bzw Nachgelagerte Systeme

1.2.1 Das Unternehmen MediaMarktSaturn

Diese Bachelorarbeit wird in Zusammenarbeit mit dem Unternehmen der *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 in den Onlineshops gestiegen ist. Dadurch wurden die Unternehmensziele dementsprechend auf die Entwicklung von Software zur Unterstützung des Onlineshops neu ausgelegt. Die, der MediaMarktSaturn Retail Group unterteilte, Firma *MediaMarktSaturn Technology* ist hierbei verantwortlich für alle Entwicklungstätigkeiten. Dieses Projekt wurde im Team *Chechkout & Payment* erarbeitet, welches zuständig ist für die Checkout-Software.

1.2.2 Benachbarte Systeme der Checkout-Software

1.3 Motivation

- MediaMarktSaturn. Warum braucht MMS eine Checkout-Solution bzw dieses Projekt?
- Performance
- Interaction with the system (?????????)
- Reverenz für zukünftige Projekte

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

Zudem existieren aufgrund der zugrundeliegenden Architektur Performance-Einbusen. In diesem Projekt wird analysiert, ob die Performance durch einen anderen Aggregationsschnitt 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 weitere Projekte mit ähnlichen Problemstellungen konfrontiert sind.

1.4 Ziele

- Eventueller Umbau
- Überprüfen der aktuellen Architektur
- Bewertung für zukünftigere Softwareprojekte

2 Grundlagen

Zur erfolgreichen Durchführung dieses Projekts werden Kernkompetenzen der Softwareentwicklung vorausgesetzt. Diese beschäftigen sich weitestgehend mit Softwaredesign und Architekturstilen. Gängige Designprinzipien sollen sicherstellen, dass die Software positive Qualitätsmerkmale widerspiegelt und von diesen profitieren kann. Dadurch wird die Zukunftssicherheit sichergestellt, sodass auch bei Expansion der Software diese testbar, anpassbar und fehlerfrei agiert. Das weitverbreitete Akronym SOLID steht hierbei für eine Ansammlung an fünf solcher Designprinzipien, namentlich Single-Responsibility-Prinzip (SRP), Open-Closed-Prinzip (OCP), Liskovsches Substitutionsprinzip (LSP), Interface-Segregation-Prinzip (ISP) und das Dependency-Inversion-Prinzip (DIP). Architekturen und Kompositionen können anhand dieser bewertet und verglichen werden. Diese Vorgehensweise wurde verwendet, um die nachfolgenden Architekturstile zu vergleichen und bewerten.

2.1 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.

2.1.1 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.

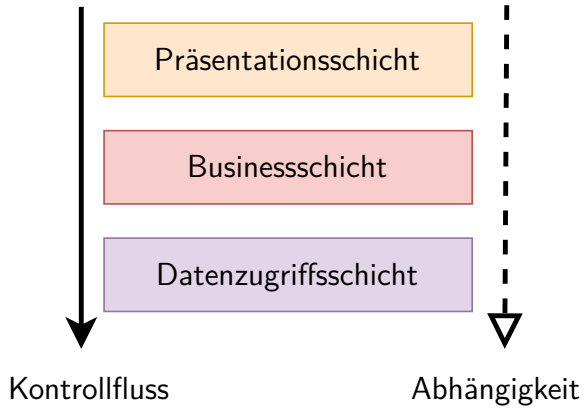

Figure 2.1: Beispielhafte Darstellung einer Drei-Schichtenarchitektur

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 DIP beachtet werden, da entgegen der intuitiven Denkweise einer Schichtenarchitektur gearbeitet werden muss. Folglich bietet dieser Architekturansatz durch seine Simplizität beiderseits Vorteile und Nachteile.

2.1.2 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-Prinzips*, 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.

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-Prinzip* 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

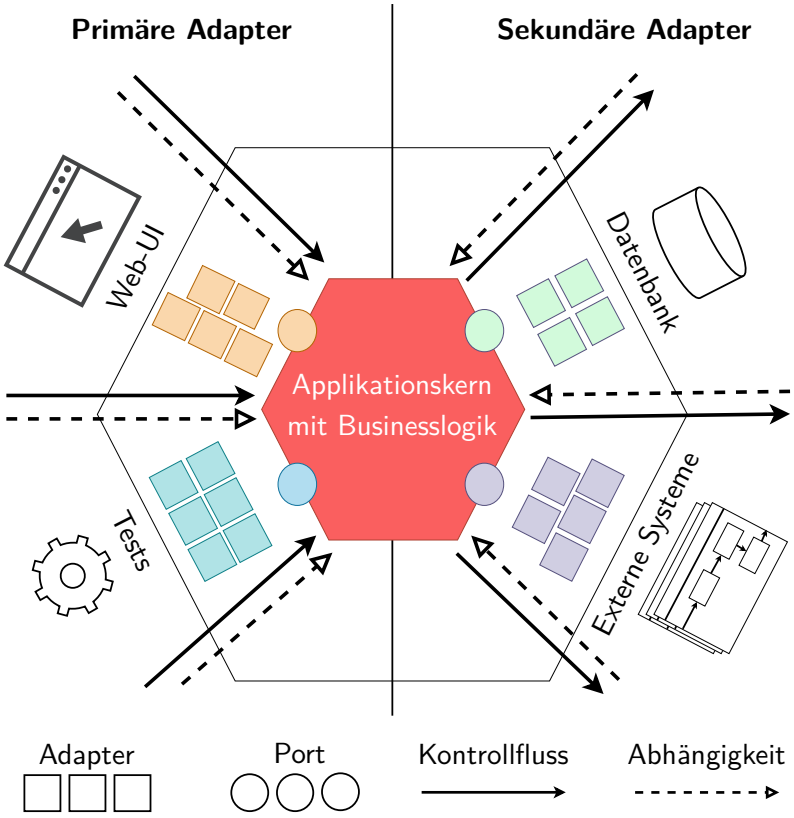

Figure 2.2: Grundstruktur einer Hexagonalen Architektur

2.2 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 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. 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 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.

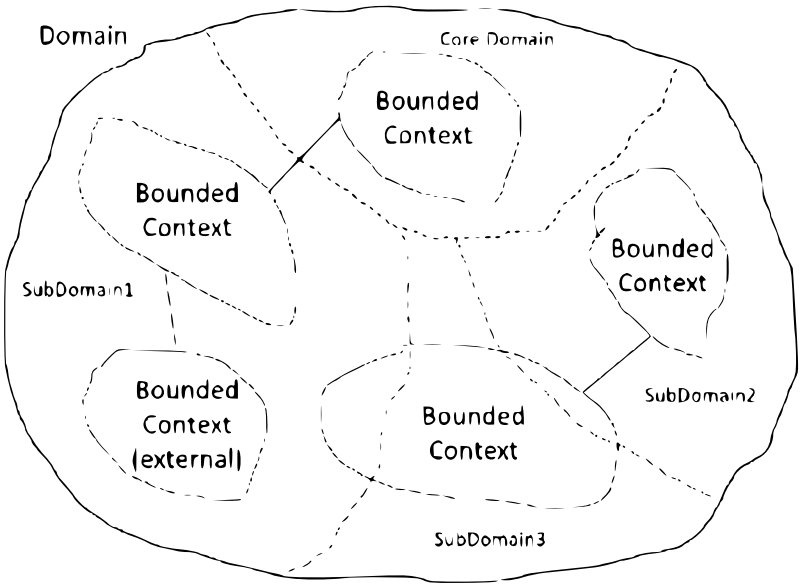

Figure 2.3: Beispiel einer Context-Map

Im Zentrum von DDD steht das Domain-Modell, welches die Kenntnisse über den Bounded-Context darstellt. Es umfasst sowohl die Datenhaltung als auch das zugehörige Verhalten, wie zum Beispiel die Erstellung von Objekten 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 Gegenzug 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 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.

3 Planung und Analyse

Einleitend werden Struktur, Motivation und die abgeleiteten Forschungsfragen diskutiert.

3.1 Umfeldanalyse

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.

3.2 Anwendungsfälle

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.

3.3 Aktuelles Design der Produktivanwendung

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.

4 Erstellung eines Proof of Concepts

Einleitend werden Struktur, Motivation und die abgeleiteten Forschungsfragen diskutiert.

4.1 Analysieren der Domäne

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.

4.2 Festlegen einer Ubiquitous Language

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.

4.3 Abgrenzung des Bounded Contexts

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.

4.4 Design der Wertobjekte

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.

4.5 Entitäten

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.

4.6 Aggregatbestimmung anhand der Anwendungsfälle

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.

4.7 Wahl der Aggregate Roots

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.

4.8 Domänenservice

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.

4.9 Applikationsservice

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.

4.10 Primäre und Sekundäre Adapter

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.

5 Fazit und Empfehlungen

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.

Bibliography

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