

HSLU SWDA

Eldar Omerovic

January 2025

Contents

1 Architektur Einführung	5
1.1 Einige Grundbegriffe	5
1.2 Was ist Software Architektur	5
1.2.1 Definition von Software Architektur	5
1.2.2 Aspekte und Herausforderungen der Software Architektur	5
1.2.3 Verschiedene Arten von Applikationen	6
1.2.4 Erkenntnisse der klassischen Client/Server-DB Anwendung	6
1.2.5 Fliessender Wechsel zwischen Design und Architektur	6
1.3 Systeme und Komponenten	7
1.3.1 Softwaresysteme und Subsysteme	7
1.3.2 Komponenten und (Sub-)systeme	7
1.4 Monolithische Architektur	7
1.4.1 Verteilte Applikationen / Architektur	8
1.4.2 Muster für verteilte Applikationen	8
1.5 Modularisierung	9
1.5.1 Modularisierung im Softwaredesign	9
1.5.2 Kriterien zur Modularisierung	9
1.5.3 Modularisierung - Einfluss auf die Architektur	9
1.5.4 Zentrale Eigenschaften eines Moduls	9
1.5.5 Kriterien für den Entwurf eines Modules	10
1.6 Prinzipien für Modulentwurf	10
1.6.1 REP - Reuse-Release-Equivalence Prinzip	10
1.6.2 CCP - Common-Closure Prinzip	10
1.6.3 CRP - Common-Reuse Prinzip	11
1.6.4 REP/CCP/CRP	11
1.6.5 Prinzipien für die Modulkopplung	11
1.7 Beispiel: SLF4J	12
2 Anforderungen	13
2.1 Requirements Engineering	13
2.2 Aufgaben und Ziele	13
2.3 Stakeholder	13
2.4 Anforderung	14
2.4.1 Funktionale vs. Nicht-funktionale Anforderungen	14
2.4.2 Qualitätskriterien für Anforderungen	14
2.4.3 Unterschiedliche Methoden / Techniken	14
2.5 User Stories	15
2.5.1 Akzeptanzkriterien	15
2.6 Event Storming	16
2.7 Story Mapping	16
3 Vorgehen im Projekt	17
3.1 Product-Backlog	17
3.2 SWDA: User Story Content	17
3.3 SWDA: Akzeptanzkriterien	17
3.4 Technische Randbedingungen	18
3.5 Aufwandschätzung	18
3.6 Definition of Ready (DoR)	18
3.7 Sprint Planning	18
3.8 Sprint Review	19
3.9 Gitlab	19
3.9.1 Naming in Gitlab	19
3.9.2 Meilensteine	19

4 Architekturen und -muster	20
4.1 Architekturmuster	20
4.2 Schichtenbildung	20
4.2.1 Layers	20
4.2.2 Schichten vs. Features - Umsetzung in Java	21
4.2.3 Potential: Verteilte Schichten auf Tiers	21
4.2.4 Der Klassiker: Logische 3-Schicht-Architektur	21
4.2.5 1,2,3 oder n-Schichten - Allgemeine Punkte	22
4.3 Service-Oriented-Architecture	22
4.3.1 SOA - Konsequenzen	23
4.4 Message- oder Event-Driven-Architecture	23
4.4.1 Fachlicher nutzen	23
4.4.2 Messages (Events) als Architekturmittel	23
4.4.3 Enterprise Service Bus (ESB)	24
4.5 Zwischenbilanz / Zusammengefasst	24
5 Microservices	25
5.1 Aufteilung einer Applikation in kleine Microservices	25
5.2 Jeder Service hat eigenen Prozess / Plattform	26
5.3 Leichtgewichtige Kommunikation	26
5.4 Voneinander unabhängig Deploy- und Releasebar	26
5.5 Deployment automatisiert (DevOps)	26
5.6 Schnittstellen und Kommunikation	26
5.6.1 Gateway-Pattern mit Microservices	27
5.6.2 Inter-Kommunikation zwischen Microservices	27
5.7 Anforderungen an die Resilienz	27
5.7.1 Was passiert wenn ein Microservice ausfällt?	27
5.7.2 Microservice Pattern: Circuit-Breaker	28
5.8 Beispiele für Technologien und Frameworks	28
5.9 Deployment von Microservices	28
5.10 Umgang mit Transaktionen	29
5.11 Logging, Metrics und Tracing	29
5.11.1 Logging	29
5.11.2 Metriken	30
5.11.3 Tracing	30
5.12 Service Discovery	30
6 Modelle	31
6.1 Anforderungsdiagramme & -modelle	31
6.2 Kontext Diagramm	31
6.3 Anwendungsfall Diagramm	32
6.4 Geschäftsprozess Modell	32
6.4.1 BPMN - Business Process Management Notation	33
6.5 Domain Modell	33
6.5.1 Domain Model mit Kontexten	34
6.6 Feature-Liste	34
6.7 C4 Modell	35
6.8 Microservice Architektur Diagramm [SWDA]	36
6.9 API Specification	36
7 Methodik	37
7.1 Prozessanalyse mit Event Storming	37
7.1.1 Schritt für Schritt	37
7.2 Wie finde ich "meine" Microservices?	39
7.2.1 Bounded Context	39

8 API-Design	40
8.1 Wann wird eine Schnittstelle zum API	40
8.2 Motivation für gute APIs	40
8.3 API - Herausforderungen	40
8.3.1 Hauptziele einer guten API	41
8.4 API Qualitätsanforderungen	41
8.5 API Patterns	41
8.6 Mehr als nur Nutzer und Implementation	41
8.7 SPI - Die API für den Anbieter	41
8.8 Class-based API	42
8.9 REST - REpresentational State Transfer	42
8.9.1 Zustandslosigkeit	42
8.9.2 REST - Standardmethoden	42
8.9.3 Endpoint Naming, Statuscodes, Request-parameters	43
8.9.4 Beurteilen von REST Schnittstellen	43
8.9.5 Versionierung	43
9 Testen von Applikationen	44
9.1 Testarten	44
9.2 Ziele für gute Tests	44
9.3 Tests in Schichtenarchitekturen	44
9.4 Testen von DB-Applikationen	44
9.5 Testen von REST-(Micro-)Services	45
10 Summary of Braindumps	46
10.1 Person 1	46
10.2 Person 2	46
10.3 Person 3	46
10.4 Person 4	46
10.5 Person 5	47
10.6 Person 6	47
10.7 Person 7	47
10.8 Person 8	47
10.9 Person 9	47
10.10 Person 10 (aus 1 Semester vorher)	48
11 Vorbereitung eigene MEP	49
11.1 Bounded Context im Domain Modell	49
11.2 Requirements Engineering	49
11.3 Diagramme und Modelle	51
11.4 Technische & Allgemeine Fragen	52
11.5 Unsere Diagramme aus dem Projekt - Analyse	54
11.5.1 BPMN	55
11.5.2 Microservice-Architektur-Diagramm	56
11.5.3 Domain-Modell	56

1 Architektur Einführung

1.1 Einige Grundbegriffe

Datenkapselung (hohe Kohäsion)

Datenkapselung bedeutet, dass die Daten einer Klasse nur über definierte Methoden zugänglich sind. Dadurch wird der Zugriff auf die Daten kontrolliert, und die Implementierung kann verändert werden, ohne dass andere Teile des Programms beeinflusst werden.

Information Hiding (lose Kopplung)

Information Hiding bedeutet, dass eine Klasse oder ein Modul Details seiner Implementierung verbirgt. Dadurch bleibt die Außenwelt von Änderungen unberührt, solange das Interface gleich bleibt. Dies reduziert die Abhängigkeiten zwischen Modulen (lose Kopplung).

Modularisierung (hohe Kohäsion)

Modularisierung ist das Aufteilen eines Programms in kleinere, unabhängige Module. Jedes Modul sollte eine klar definierte Aufgabe (hohe Kohäsion) haben. Dadurch wird der Code besser wartbar und wiederverwendbar.

Schnittstellen (lose Kopplung)

Eine Schnittstelle (Interface) definiert, wie verschiedene Komponenten miteinander interagieren können, ohne Details der Implementierung preiszugeben. Dies fördert lose Kopplung und erleichtert die Zusammenarbeit zwischen Modulen.

1.2 Was ist Software Architektur

Eine Architektur ist eine Abstraktion, es wird etwas zusammenfassend und vereinfachend dargestellt.
Vergleiche mit Modellierung und Design - fliessender Übergang!

Die Architektur beschreibt ein System durch:

- dessen Struktur und Aufbau: Sub- und Teilsystemen, Schichtung, Verteilung
- der darin enthaltenen Softwareteile:
 - Komponenten, gegliedert nach Aufgaben und Zuständigkeiten.
 - Teilweise aber auch gegliedert nach Technologie.
- deren Beziehungen (im weitesten Sinne) untereinander:
 - Abhängigkeiten, Schnittstellen, Daten- und Kontrollflüsse (Kommunikation), Transaktionen, Deployment etc.

1.2.1 Definition von Software Architektur

Softwarearchitektur definiert sich durch die Kernelemente eines Systems, welche als Basis für alle weiteren Teile nur schwer und aufwändig verändert werden können. - Martin Fowler

Die Architektur repräsentiert die signifikanten Designentscheidungen die ein System festhalten, wobei die Signifikanz an den Kosten von Änderungen bemessen wird. - Gray Booch

1.2.2 Aspekte und Herausforderungen der Software Architektur

Grundlegende Architektur

- Schichten, Client/Server, n-tiers, Services etc.
- Art der Applikation und des Deployment (äußere Struktur)
- Architekturmuster und -prinzipien (innere Struktur)

Kommunikation und Verarbeitung

- Verteilbarkeit, Parallelität, Performance, Robustheit, Resilienz
- Kommunikationsmuster, z.B. Synchron / Asynchron
- Interaktion der Systeme, Transaktionalität

Eingesetzt Technologien

- Userinterface (Fat-, Rich- oder Thin-Client)
- Persistenz der Daten (Frameworks, O/R-Mapping, NoSQL etc.)
- Referenzarchitektur

1.2.3 Verschiedene Arten von Applikationen

Wir haben eine grosse Bandbreite von Applikationen! Beispiele:

Einzelbenutzerapplikation

- Eher klein, eventuell sogar "nur" eine Mobile-App
- Lokale Persistenz, eher wenig komplexe Funktionalität

Mehrbenutzerapplikationen

- z.B. Unternehmensapplikation (Enterprise Software).
- Zentrale Services und Daten, beliebig hohe Komplexität.
- Typisch in mehrere (Teil-)System aufgebrochen.

Internetanwendungen

- Typisch mit Web-GUI, beliebig viele Benutzer
- Verteile Datenspeicherung, Skaliert, beliebige Komplexität

All diese unterschiedlichen Arten von Applikationen haben sehr unterschiedliche Anforderungen an die Architektur!

1.2.4 Erkenntnisse der klassischen Client/Server-DB Anwendung

- Es führte oft zum berühmt-berüchtigten FAT-Client (Geschäftslogik im Client) und einer überforderten Datenbank.
 - Vorteil: Zentrale Datenhaltung, hohe Konsistenz
 - Nachteile: Transaktionen, Locking, Ressourcen, Verteilung, ...
- Im Kleinen z.B. für lokale (Endbenutzer-)Apps auf Handys funktioniert das gerade sehr gut!

Die Kunst ist die richtige Architektur am richtigen Ort zu wählen.

1.2.5 Fliessender Wechsel zwischen Design und Architektur

Wir kennen und nutzen in der SW-Architektur eine grosse Palette von "altbekannten" Mustern, Prinzipien und Techniken:

- Modularisierung durch Komponenten, Schnittstellen, Packages
- Gruppierung dieser Teile zu (Sub-)System
- Nutzung von verschiedenen Mustern zur Strukturierung (z.B. MVC - Model View Control)

Die grosse Herausforderung: Alle diese Prinzipien auf den unterschiedlichen Abstraktionsebenen angemessen zu befolgen und zu beurteilen.

Schlussendlich haben wir "nur" einen Haufen von Klassen und Schnittstellen - jede weitere Strukturierung ergibt sich weitgehend nur durch Organisation und Einhaltung von Konventionen!

1.3 Systeme und Komponenten

1.3.1 Softwaresysteme und Subsysteme

Softwaresysteme werden wenn möglich immer in einzelne Subsysteme oder Teilsysteme zerlegt.
Die Subsysteme haben eine gewisse Unabhängigkeit und interagieren dennoch über wohldefinierte Schnittstellen.

- Kopplung: über möglichst lose Schnittstellen
- Kohäsion: mit Datenkapselung und Information Hiding

Vorteile dieser hierarchischen Strukturierung:

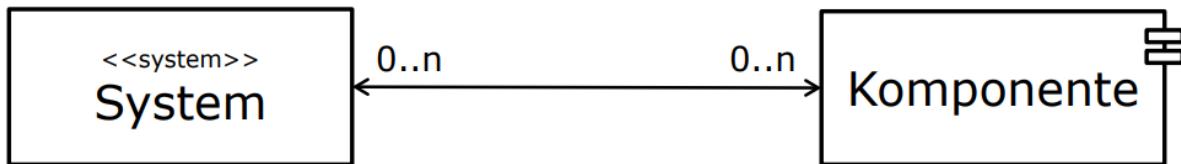
- Präzisere Schätz- und Planbarkeit
- Unabhängige Entwicklung möglich
- Einfachere Testbarkeit
- Unabhängiges Deployment
- Potential für Wiederverwendung höher

Ziele: Einfacher! Verständlicher! Schneller!

1.3.2 Komponenten und (Sub-)systeme

Eine Komponente ist eine softwaretechnische Einheit. Durch den Einsatz von Komponenten (und Klassen und Interfaces) kann ein System realisiert werden. Ein System selbst muss nicht zwingend eine Komponente enthalten (es ist aber hoffentlich sehr häufig der Fall).

Eine Komponente kann nur Realisation von mehreren Systemen verwendet werden (Wiederverwendung).



”Komponente” und ”System” sind somit orthogonal (unabhängig voneinander).

1.4 Monolithische Architektur

Monolithische Applikationen sind in jüngster Zeit zu Unrecht stark in Verruf gekommen. Dies, da viel zu wenig differenziert wird.

Es ist sehr wichtig zu unterscheiden, ob es sich um

- **Monolithisches Design** (ohne Modularisierung), oder
- um **monolithisches Deployment** handelt.

Das sind zwei ebenfalls orthogonale Aspekte, die sich auf sehr unterschiedliche Aspekte auswirken!

- **Monolithisches Design:** Bezieht sich auf die Struktur des Codes, bei der keine Modularisierung vorliegt. Codebase hat eine schlechte Struktur. Dies kann die **Wartbarkeit und Erweiterbarkeit** erschweren.
- **Monolithisches Deployment:** Bedeutet, dass die gesamte Anwendung als eine Einheit bereitgestellt wird, unabhängig davon, ob sie intern modular aufgebaut ist. Dies beeinflusst vor allem **Betrieb und Skalierung**.
 - Nachteil: Bei einem Deploy muss immer die ganze Applikation deployed werden. Heisst die ganze Applikation ist während dieser Zeit unverfügbar.
 - Nachteil: Konfigurationsmanagement aufwändig (alle Komponente wieder in eine komplette Anwendung bringen).
 - Gutes Beispiel: Mobile-App
 - Schlechtes Beispiel: Größere, umfangreiche Webapplikation welche streng modularisiert ist. Hier könnte die Skalierbarkeit Thema werden, ein monolithisches Deployment ist aber unter Umständen auch hier okay.

1.4.1 Verteilte Applikationen / Architektur

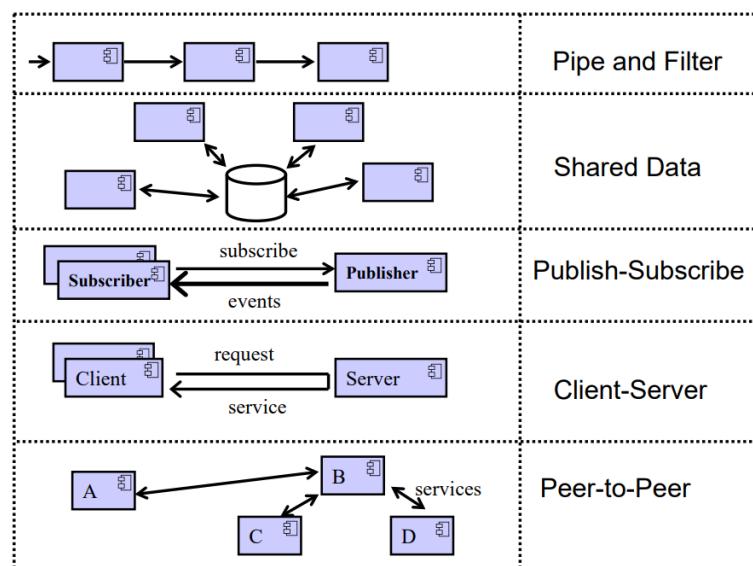
Bei verteilten Applikationen werden einzelne Teile (Teilsysteme, Komponenten, Module, Schichten, etc.) auf mehrere, verschiedene Rechner (hosts, tiers) verteilt.

Offensichtliche Konsequenzen: Die Teile laufen in einzelnen, unabhängigen Prozessen und somit in echter Parallelität.

Sich daraus ergebende Anforderungen sind:

- Die Teile müssen miteinander kommunizieren: Auswahl der geeigneten Kommunikationstechnologie.
- Die einzelnen Teile müssen sich finden und kennen.
- Deployment wird aufwändiger: Mehr und (insgesamt) häufiger. Abhängigkeiten müssen beachtet und koordiniert werden.
- Es muss mit (Teil-)Ausfällen umgegangen werden können.

1.4.2 Muster für verteilte Applikationen



1.5 Modularisierung

1.5.1 Modularisierung im Softwaredesign

- Die kleinste Einheit in der objektorientierten SW-Entwicklung stellt im Prinzip die Klasse dar.
- Eine grosse Einheit stellt eine vollständige Applikation oder auch ein ganzes Softwaresystem dar.
- Die Modularisierung identifiziert zwischen diesen beiden Extremen sinnvolle funktionale Module, die möglichst abgeschlossene und austauschbare Einheiten bilden.
 - Verfügen über wohldefinierte Schnittstellen zur Interaktion
 - Diese einzelnen Module sind einfacher verständlich, können parallel entwickelt werden und sind wiederverwendbar.

1.5.2 Kriterien zur Modularisierung

Die zwei wichtigsten Kriterien zur Modularisierung sind aus dem OO-Design bekannt:

- Kopplung (Coupling):
 - Beschreibt das Ausmass der Kommunikation zwischen Modulen.
 - Mass für die Abhängigkeiten zwischen den einzelnen Modulen.
 - Ziel: Minimierung der Kopplung!
- Kohäsion (Cohesion):
 - Ausmass der Kommunikation innerhalb eines Moduls.
 - Mass für den internen Zusammenhalt eines Moduls.
 - Ziel: Maximierung der Kohäsion!

Die Kopplung und Kohäsion werden auf allen Abstraktionsebenen immer wieder neu beurteilt. Die Anforderungen werden aber zunehmend strenger.

1.5.3 Modularisierung - Einfluss auf die Architektur

Die Modularisierung führt meistens zu einer vielfältig ausgeprägten, übergeordneten Struktur und Organisation aller Module in einem System.

- **Gruppierung:** Eine Menge von Modulen mit gemeinsamen Eigenschaften wird als Gruppe gehandhabt. z.B. Module für Datenexport in verschiedenen Formaten.
- **Hierarchie (Rekursiv):** Ein Modul fasst mehrere (Sub-)Module zu einem einzigen zusammen. z.B. Persistenzmodul als Datenspeicher mehrerer Entitäten.
- **Geschichtet:** Modul(-gruppen) können eine logische Kette bilden, die vertikal meist als Schichten betrachtet werden.

1.5.4 Zentrale Eigenschaften eines Moduls

Ein Modul soll nur über seine Schnittstelle verwendet werden können. Die Schnittstelle soll möglichst einfach und schmal sein. Es soll eine möglichst in sich geschlossene Aufgabe erfüllen und einen starken inneren Zusammenhalt aufweisen (Starke Kohäsion / Single Responsibility Principle (SRP) und Separation of Concerns (SoC)).

Daraus ergeben sich:

- **Information Hiding:** Die Implementation eines Moduls bleibt vor deren Nutzern verborgen, und kann somit auch jederzeit verändert werden.
- **Lose Kopplung:** Die nur über die Schnittstelle mögliche Nutzung des Modules fördert die möglichst lose Kopplung zwischen den Modulen.

1.5.5 Kriterien für den Entwurf eines Modules

Es gibt vier fundamentale Kriterien für den Entwurf von Modulen:

- Verständlichkeit: Die Module sind einzeln (und ohne Kontext) leicht verständlich.
- Stetigkeit / Kontinuität: Die Module haben eine hohe Beständigkeit, müssen also nicht bei jeder Wiederverwendung wieder angepasst werden.
- Zerlegbarkeit / Dekomposition: Die einzelnen Module sind maximal unabhängig voneinander und auf ein Minimum reduziert (SRP).
- Kombinierbarkeit: Die entstandenen Module sind sinnvoll neu kombinierbar.

1.6 Prinzipien für Modulentwurf

Prinzipien und Regeln für die Modulkohäsion:

- REP: Reuse-Release-Equivalence Prinzip
- CCP: Common-Closure Prinzip
- CRP: Common-Reuse Prinzip

Prinzipien und Regeln für die Modulkopplung:

- ADP: Acyclic Dependencies Prinzip
- SDP: Stable Dependencies Prinzip
- SAP: Stable Abstractions Prinzip*

* wird nicht ganz alles behandelt in diesem Modul.

1.6.1 REP - Reuse-Release-Equivalence Prinzip

Die Granularität der Wiederverwendung ist die Granularität des Release.

Heisst: Man fügt sinnvollerweise das zusammen, was auch sinnvollerweise gemeinsam in einer Version veröffentlicht wird. Ist eigentlich kein Prinzip sondern eine best-practice, die sich aus der Erfahrung gegeben hat.

Buildtools wie z.B. Maven fördern dieses Prinzip: Ein Maven-Modul kann einer Komponente entsprechen, daraus wird z.B. ein versioniertes JAR als die Deploy-, bzw. Releaseeinheit.

REP ist ein inkludierendes Prinzip (macht Dinge grösser).

1.6.2 CCP - Common-Closure Prinzip

Klassen, die aus denselben Gründen und zur selben Zeit modifiziert werden, fasst man in denselben Komponenten zusammen.

Im Gegenzug: Klassen, die man aus unterschiedlichen Gründen zu unterschiedlichen Zeitpunkten modifiziert, separiert man möglichst in verschiedene Komponenten.

Das ist nichts anderes als SRP (Single Responsibility Prinzip) und OCP (Open Closed Prinzip) auf Architekturebene angewendet. Das sind zwei der fünf zentralen S.O.L.I.D-Prinzipien!

CCP ist ein inkludierendes Prinzip.

1.6.3 CRP - Common-Reuse Prinzip

Zwinge Nutzende einer Komponente nicht in eine Abhängigkeit von Elementen, die er nicht benötigt.

Heisst: Ein Modul sollte als Ganzes sinnvoll nutzbar sein, und nicht nur Teile oder Bruchstücke davon. Sind Einzelteile losgelöst sinnvoll nutzbar, sollte man sie auch in eigenständig wiederverwendbare Module ausgliedern.

Achtung: Besonders bei Libraries und Frameworks (Komponentenbibliotheken) sollte man mit den transitiven (indirekten, durch andere Komponenten geladene) Abhängigkeiten sehr sorgsam umgehen.

CRP ist eine exkludierendes Prinzip (macht Dinge kleiner).

1.6.4 REP/CCP/CRP

Diese drei Prinzipien arbeite somit gegeineinander:

- REP fasst zusammen, so dass die Wiederverwendung einfacher wird.
- CCP fasst zusammen, was gemeinsam einfacher Wartbar ist.
- CRP teilt feiner auf, damit man weniger (unnötige) Abhängigkeiten hat.

Erkenntnis

Beim Entwurf von Modulen sollten wir diese gegensätzlichen Interessen berücksichtigen und ausgewogen ausgleichen.

- Wiederverwendung vs. Wartung vs. Abhängigkeiten.
- Module als sinnvolle Grundbausteine einer Architektur.

Das sollte natürlich auch in Einklang mit den individuellen Anforderungen einer konkreten Anwendung geschehen, die sich im Verlaufe noch verändern werden. Längerfristig kann sich z.B. der Fokus von der Wartbarkeit durchaus in die Wiederverwendung verlagern.

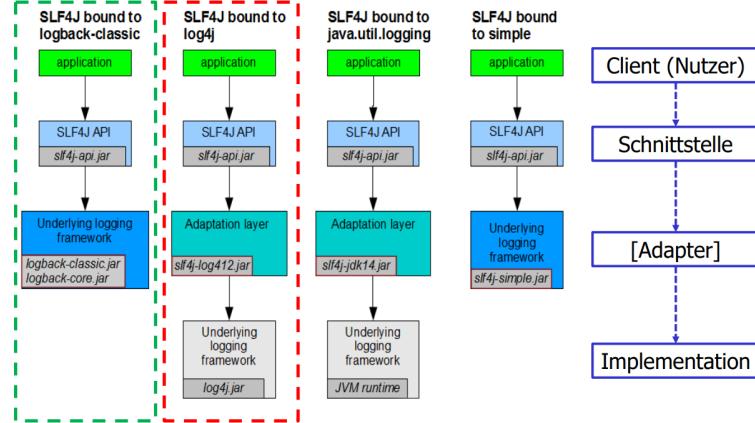
Das alles zu berücksichtigen, sollte die Kunst und die Fähigkeit eines guten Softwarearchitekten sein.

1.6.5 Prinzipien für die Modulkopplung

- ADP - Acyclic Dependencies Prinzip: Zwischen Komponenten sind keine Zyklen erlaubt.
- SDP - Stable Dependencies Prinzip: Die Abhängigkeiten zwischen Komponenten sollten in Richtung der stabileren Komponenten verlaufen.
- SAP - Stable Abstractions Prinzip: Stabile Komponenten sollten im gleichen Masse auch abstrakt sein.

1.7 Beispiel: SLF4J

Die Architektur und das Design nutzt Komponenten, Schnittstellen, Schichtenbildung und gezieltes Deployment für ein konkretes Einsatzszenario:



Anforderungen die gestellt sind:

- Client/Server - Struktur: Der Client will über eine einheitliche unabhängige API loggen.
- Das tatsächliche verwendete Logging-Framework soll dabei vollständig abstrakt bleiben.
- Das konkret eingesetzte Logging-Framework soll möglichst einfach austauschbar sein.
- Der Austausch soll rein über das Deployment eines entsprechenden Adapters (Logger-Binding) erfolgen.

2 Anforderungen

2.1 Requirements Engineering

Requirements engl. für Anforderungen.

”Anforderungen sind die Fähigkeiten und Eigenschaften des gewünschten Produkts.”

Engineering engl. für Ingenieurwissenschaften

Ingenierwesen ist die Anwendung wissenschaftlicher Prinzipien, um Maschinen, Strukturen und andere Gegenstände zu entwerfen und zu bauen.

Requirements Engineering ist das systematische Vorgehen beim Spezifizieren und Verwalten von Anforderungen an ein System, ein Produkt oder eine Software.

Es hilft uns folgende Fragen zu beantworten:

- Was sind die Anforderungen - also die Fähigkeiten und Eigenschaften - des gewünschten Produkts?
- Wie lassen sich diese Anforderungen spezifizieren - also erheben, analysieren, dokumentieren und validieren?
- Wie sollten Anforderungen verwaltet - also freigegeben, evtl. verändert oder rückverfolgt - werden?

2.2 Aufgaben und Ziele

Aufgaben:

- Die Ermittlung von Anforderungen inklusive Detaillierung und Verfeinerung.
- Die Dokumentation als adäquate Beschreibung von Anforderungen.
- Die Prüfung und Abstimmung von Anforderungen mit dem Ziel, die Qualität sicherzustellen.
- Die Verwaltung - auch als Requirements Management bezeichnet - von Anforderungen.

Ziele:

- Risikominimierung
- Gemeinsames Verständnis der Stakeholder

2.3 Stakeholder

Als Stakeholder wird eine Person oder Gruppe bezeichnet, die ein berechtigtes Interesse am Verlauf oder am Ergebnis hat.

Beispiele für Stakeholder:

- Management
- Tester
- Gesetzgeber
- Sicherheitsbeauftragte
- Kunden
- Entwickler

Stakeholder sind die Informationslieferanten für Ziele, Anforderungen und Randbedingungen an ein zu entwickelndes System oder Produkt.

2.4 Anforderung

- **Nutzungsanforderungen:** Anforderungen an die effiziente Erbringung eines Ergebnisses mit einem interaktiven System.
- **Gesetzliche Anforderungen:** Anforderungen von Gesetzgebern und Behörden z.B. in Form von Richtlinien, Gesetzen, Verordnungen, Normen, usw.
- **Fachliche Anforderungen:** Anforderung an die Vollständigkeit und Korrektheit eines Arbeitsergebnisses.
- **Organisatorische Anforderungen:** Anforderung an das Verhalten von Personen oder Organisationseinheiten bei der Erbringung von Arbeitsergebnissen.
- **Marktanforderungen:** Anforderungen die für die Kaufentscheidung entscheidend / relevant sind.

2.4.1 Funktionale vs. Nicht-funktionale Anforderungen

Funktionale Anforderungen legen Fest, **was** das Produkt zu tun hat.

Nicht funktionale Anforderungen:

- Performanz und Effizienz
- Zuverlässigkeit (Reliability)
- Verfügbarkeit (Availability)
- Sicherheit (Security)
- Wartbarkeit (Maintainability)
- Portierbarkeit (Portability)

2.4.2 Qualitätskriterien für Anforderungen

- Verständlichkeit, Klarheit
- Eindeutigkeit
- Vollständigkeit
- Konsistenz
- Korrektheit
- Gültigkeit (aktuell)
- Verfolgbarkeit / Traceability
- Testbarkeit / Prüfbarkeit
- Machbarkeit / Umsetzbarkeit
- Bewertet (Prio, Aufwand, Risiko)

2.4.3 Unterschiedliche Methoden / Techniken

Wir können z.B. folgende Methoden / Techniken unterscheiden, um die Anforderungen zu erheben, festzuhalten und zu analysieren.

- Kontext-Diagramm
- Domain-Modell
- Use Case Modell und Beschreibungen
- Geschäftsprozess-Modell
- Feature-Listen
- User Stories

2.5 User Stories

Eine User Story ist eine textliche Anforderung an das System.

Als "Stakeholder" möchte ich "Ziel / Wunsch" um "Nutzen"

- User Stories beschreiben Anforderungen an ein System aus Anwender-Sicht. Jede Story muss für den Anwender einen klaren Nutzen erbringen und für den Anwender verständlich sein (keine technischen Lösungsbeschreibungen).
- User Stories müssen im Rahmen des Projektes so umformuliert und in mehrere Stories zerlegt werden, dass die im Rahmen eines Sprints umgesetzt werden können.

Beispiel:

Als **HR-Mitarbeiter** möchte ich *nach Mitarbeitern suchen*, die heute Geburtstag haben, damit **ich ihnen ein Geschenk bringen kann**.

Die Fett geschriebenen Teile sind gut. Der kursiv geschriebene Teil, schlägt bereits eine Lösung vor. Dies sollte eine User Story nicht. Was also wünscht sich der HR-Mitarbeiter hier? Besser ist:

Als **HR-Mitarbeiter** möchte ich **die Namen aller Mitarbeiter wissen, die heute Geburtstag haben**, damit **ich ihnen ein Geschenk bringen kann**.

Dann kann eine Lösung gefunden werden und es wird keine vorgegeben.

2.5.1 Akzeptanzkriterien

Was sind Akzeptanzkriterien?

- Akzeptanzkriterien definieren, was getan werden muss, um eine User Story abzuschliessen.
- Sie legen die Grenzen der Story fest und werden verwendet, um zu bestätigen, dass die Story wie vorgesehen funktioniert.
- Helfen ein gemeinsames Verständnis zu bilden.

Was sollten / müssen Akzeptanzkriterien erfüllen?

- Jede User Story sollte mindestens ein Akzeptanzkriterium haben
- Akzeptanzkriterien werden vor der Implementierung geschrieben
- Jedes Akzeptanzkriterium ist unabhängig testbar
- Akzeptanzkriterien müssen ein klares Pass / Fail Ergebnis haben.
- Akzeptanzkriterien konzentrieren sich auf das Endergebnis - Das Was, nicht das Wie.
- Schliessen Sie sowohl funktionale als auch nicht-funktionale Kriterien ein.

2.6 Event Storming

Event Storming ist eine Methode, die dazu dient, komplexe Geschäftsprozesse und Domänen schnell zu verstehen, indem man sich auf Ereignisse konzentriert, die in einem System auftreten. Es wird oft im Kontext von Domain-Driven-Design (DDD) verwendet, kann aber auch unabhängig davon eingesetzt werden.

- **Ziel:** Geschäftsprozesse, Anforderungen und Domänenlogik in einer kollaborativen und visuell verständlichen Weise darzustellen.
- Technik:
 1. **Ereignisse (Events):** Zentrale Elemente sind Ereignisse, die in der Domäne passieren (z.B. "Kunde bestellt ein Produkt").
 2. **Visualisierung:** Diese Ereignisse werden als Post-Its auf einer grossen Leinwand oder Wand angeordnet.
 3. **Facilitation:** Ein Moderator leitet die Sitzung, während alle Stakeholder (Entwickler, Fachbereich, Product Owner, usw.) ihr Wissen einbringen.
 4. **Zusätzliche Elemente:** Kommandos, Aggregate, externe Systeme, Policies usw. um den Prozess zu vervollständigen.

Event Storming fördert die Zusammenarbeit zwischen technischen und nicht-technischen Stakeholdern. Außerdem werden Wissenslücken oder Problemstellen schnell identifiziert. Auch wird das allgemeine Verständnis der Domäne bei den Stakeholdern verbessert.

2.7 Story Mapping

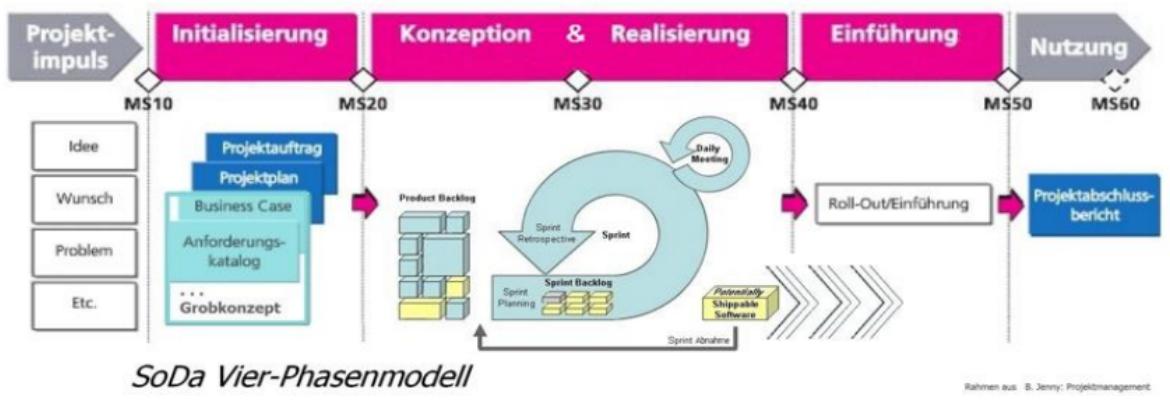
Story Mapping ist eine Technik zur Visualisierung von Anforderungen und Nutzerbedürfnissen in Form einer Hierarchie von User Stories, die oft in einem agilen Kontext verwendet wird.

- **Ziel:** Ein strukturiertes Backlog zu erstellen, das die Wertschöpfung für den Nutzer priorisiert und gleichzeitig die gesamte User Journey abdeckt.
- Technik:
 1. **User Journey:** Die Hauptaktivitäten eines Nutzers werden horizontal angeordnet, z.B. "Produkt finden", "Produkt kaufen".
 2. **User Stories:** Unter jedem Hauptschritt werden detaillierte Aufgaben oder Features als Stories notiert.
 3. **Priorisierung:** Stories werden vertikal angeordnet, um ihre Priorität zu zeigen (wichtigste oben).
 4. **Releases:** Geschichten können nach Iteration oder Releases gruppiert werden.

Story Mapping verbessert die Übersicht über das gesamte Produkt oder Projekt. Es stellt sicher, dass alle Nutzerbedürfnisse abgedeckt werden. Es stellt eine klare Verbindung zwischen Anforderung und Nutzerwert her. Das Priorisieren hilft beim Planen von MVPs (Minimal Viable Products) und Releases.

3 Vorgehen im Projekt

Vorgehen im Projekt ist SoDa. Dazu gibt es folgendes SoDa Vier-Phasenmodell:



d.h. es wird angelehnt an Scrum entwickelt:

- Sprint mit Sprint-Planung und Sprint-Review.
- Sprints werden mittels Stories inkl. Akzeptanzkriterien beschrieben.
- Stories werden im GitLab als ScrumBoard verwaltet.
- Projekt teams aus 4 Personen

3.1 Product-Backlog

- Eine geordnete Liste von User Stories, primär aus Anwender-Sicht formuliert. Siehe dazu vorheriges Kapitel "User Stories". Tendenziell umfassen User Stories einen vertikalen Schnitt durch ein System d.h. Code für eine einzelne Funktionalität vom Benutzerinterface bis zur Persistierung.
- Stories, welche für die Umsetzung in einem Sprint geplant werden, müssen im Rahmen eines Sprints umsetzbar sein. Falls eine Story zu gross ist, muss sie entsprechend in mehrere Stories aufgeteilt werden.

3.2 SWDA: User Story Content

- User Story
- Akzeptanzkriterien
- Technische Randbedingungen
- Aufwandschätzung

3.3 SWDA: Akzeptanzkriterien

In einer Liste Bedingungen aufführen, die erfüllt sein müssen, damit die Story akzeptiert wird.

- gesetzliche Anforderungen
- nicht-funktionale Anforderungen
- Fehlerbedingungen

Sie sollten Lösungsneutral (nicht-technisch) sein.

Beispiel:

User Story: Als Student will ich mich online zu einer Vorlesung anmelden, damit ich dies ortsunabhängig tun kann.

Akzeptanzkriterien:

- Wenn sich ein Student einer anderen Fakultät anmeldet, muss eine Fehlermeldung erscheinen.
- Wenn bereits alle Plätze in einer Vorlesung mit begrenzter Teilnehmerzahl vergeben sind, kann sich der Studierende nicht mehr anmelden.
- Der Studierende kann sich bei einer Vorlesung mit begrenzter Teilnehmerzahl anmelden, wenn noch nicht alle Plätze vergeben sind.

3.4 Technische Randbedingungen

Technische Randbedingungen müssen für die Umsetzung der Story eingehalten werden, diese müssen mit dem PO abgesprochen und schriftlich festgehalten werden.

z.B. DB ist noch nicht vorhanden, also wird in den technischen Randbedingungen festgehalten, dass die Daten über ein definiertes Interface und einem entsprechenden Mock "persistiert" werden.

3.5 Aufwandschätzung

Zu jeder Story soll das Team eine Aufwandschätzung abgeben.

Im Team können Aufwände mit dem Planning Pojer ermittelt werden.

3.6 Definition of Ready (DoR)

Example Definition of Ready

These considerations are often summarized as the "INVEST criteria", and they provide us with a useful Definition of Ready which can be applied to Product Backlog Items. By actively participating in Product Backlog refinement, a good Development Team will collaborate with the Product Owner in making sure that a standard such as this is observed.

I (Independent). The PBI should be self-contained and it should be possible to bring it into progress without a dependency upon another PBI or an external resource.

N (Negotiable). A good PBI should leave room for discussion regarding its optimal implementation.

V (Valuable). The value a PBI delivers to stakeholders should be clear.

E (Estimable). A PBI must have a size relative to other PBIs.

S (Small). PBIs should be small enough to estimate with reasonable accuracy and to plan into a time-box such as a Sprint.

T (Testable). Each PBI should have clear acceptance criteria which allow its satisfaction to be tested.

3.7 Sprint Planning

- PO verantwortlich für das Produkt, er priorisiert die User Stories im Backlog.
- Das Team ist verantwortlich für das Erstellen der User Stories, Schätzen der User Stories und die detaillierte Task-Planung.
- Das Team muss sich zu den Zielen der Sprints "commiten". Was im Sprint geplant wird sollte auch so umgesetzt werden.

3.8 Sprint Review

- Installierte & funktionierende Software (eingecheckt, getestet, installiert auf der Umgebung).
- Produkt/Software entspricht den in den Stories definierten Anforderungen.
- Software entwickelt sich von Sprint zu Sprint inkrementell.
- Dokumentation und Diagramme aktualisiert.

3.9 Gitlab

3.9.1 Naming in Gitlab

SoDa / Scrum	Bezeichnung in GitLab
User Story	➔ Issue
Epic	➔ Epic
Aufwandschätzung	➔ Weight oder /estimate /spend
Product Backlog	➔ Issue List
Sprint	➔ Milestones & Milestone-Filtering
Board-Spalten (Todo, Doing etc.)	➔ Labels (Group-Labels)
Task	➔ Task List (im Beschreibungsfeld / Editor)

3.9.2 Meilensteine

Es gibt einen Meilenstein pro Sprint. Insgesamt über alle 4 Sprints also 4 Meilensteine. Für jeden Meilenstein entsteht dann ein Sprint Board.

4 Architekturen und -muster

4.1 Architekturmuster

- Beschreiben als Konzept den Grundaufbau eines ganzen Systems.
- Es gibt Architekturmuster für verschiedene zentrale Aspekte bzw. Schwierigkeiten in der Softwarearchitektur:
 - um (große, komplexe) Systeme zu strukturieren
 - für (stark) verteilte Systeme
 - für hinter(re)aktive Systeme
 - für hochverfügbare Systeme
- Architekturmuster sind nicht so stark vereinheitlicht und 'standardisiert' wie beispielsweise die Entwurfsmuster (GoF-Patterns).

Einige Beispiele (Architekturmuster) für Enterprise Applikationen:

- Domain Logic Patterns: Domain Model, Service Layer, etc.
- Data Source Architectural Patterns: Data Mapper, etc.
- Object-Relational Behavioural Patterns: Unit of Work, etc.
- Object-Relational Metadata Mapping Patterns: Repository, etc.
- Web Presentation Patterns: Model View Controller, Page Controller, Front Controller, etc.
- Distribution Patterns: Remote Facade, Data Transfer Object, etc.
- Base Patterns: Gateway, Mapper, Registry, Value Object, Special Case, Plugin, Service Stub, etc.

4.2 Schichtenbildung

- Gliederung eines Systems in aufeinander aufbauende, funktional getrennte Schichten.
 - Kommunikation über wohldefinierte Schnittstellen
 - Abhängigkeit nur in Richtung der tieferliegenden Schicht
 - Kein Überspringen, und auf keinen Fall Zyklen
- Zur Strukturierung sowohl innerhalb eines Systems als auch Systemübergreifend.
- Bei möglicher physischer Verteilung der einzelnen Schichten spricht man von "Tiers".

4.2.1 Layers

- Kann grundsätzlich nach verschiedenen Kriterien erfolgen: Logisch/Funktional, Technik, Abstraktionsebene, etc.
- Kann auch hierarchisch verfeinert werden, z.B:
 - Funktionale Schichtung
 - Nach Abstraktionslevel
 - Nach Technologie / Sprache / Framework
- Bei einer Implementation mit Java manifestieren sich verschiedene Schichten häufig in der Packagestruktur:
 - ch.domain.system.client.*
 - ch.domain.system.server.*
 - Achtung: Das kann ein sehr schlechter Ansatz sein! Es empfiehlt sich freingranularer zu arbeiten, speziell auch wegen der Modularisierung.

4.2.2 Schichten vs. Features - Umsetzung in Java

- Ein alternativer Ansatz der Schichten als Packages (Deployment-Perspektive) wird als "package by feature" propagiert: Alles, was aus fachlicher Sicht zusammen gehört (Model- oder funktionale Perspektive) wird in ein Package gefügt.
- Problematik: Beim Einsatz der mit Java 9 eingeführten Modularisierung wurde eine neue Restriktion eingeführt: Ein Java-Package darf nicht mehr in mehrere JAR-Dateien aufgebrochen/zerteilt werden!
 - Macht sehr Sinn, aber
 - Verunmöglicht "package by feature", bzw. wir müssen das gleichzeitig mit einer Feinaufteilung auf Layers verknüpfen!
 - Konsequenz: Feingranulare Packages/JARs → wieder positiv!

4.2.3 Potential: Verteilte Schichten auf Tiers

- Die Schichtenbildung ist eine fundamentale Grundlage, um verteilte Anwendungen bzw. Architekturen zu realisieren.
- Naheliegenderweise eignen sich die Schichtengrenzen sehr gut zur Auftrennung, um Teile auf verschiedene Systeme zu deployen.
- Zur Abstraktion der Aufteilung bzw. der Kommunikation verwendet man idealerweise wiederum (austauschbare) Schichten!

4.2.4 Der Klassiker: Logische 3-Schicht-Architektur

Aufteilung in drei fundamentale Schichten:

- Präsentation (Presentation-Layer, [G]UI-Layer): Visualisierung, User Interface, UI-Logik
- Geschäftslogik (Business[logic] Layer, Domain Layer): Implementation der Geschäftsprozesse und -modelle
- Datenhaltung (Data Layer, Persistence Layer): Persistente Datenspeicherung, Datenlogik

Diese Aufteilung lohnt sich praktisch immer, unabhängig von der physischen Verteilung! SoC, SRP, Modularisierung!

Es ist auch möglich den Client (GUI) noch weiter zu trennen, z.B. durch Model View Control (MVC).

Auch ist es möglich, die Geschäftslogik (Business-Logic) weiter zu trennen (Services / Domain Models, etc.).

Und letztlich gilt das auch für die Datenhaltungsschicht. Hier trennt man die Datenlogik unabhängig vom verwendeten (R)DBMS. Dazu kann man auch Persistenz-Frameworks einsetzen (wie z.B. JPA - Java Persistence API).

Bei einer solchen Weiter-verfeinerung auf den Schichten spricht man allgemein von einer **n-Schicht-Architektur**.

4.2.5 1,2,3 oder n-Schichten - Allgemeine Punkte

Positive Effekte:

- Bessere Strukturierung, einzelne Schichten kleiner und einfacher (somit besseres und schnelleres Verständnis).
- Grössere Chance auf Wiederverwendung (einzelner Layers)
- Höhere Flexibilität, z.B. Austausch einzelner Schichten
- Bessere Skalierbarkeit (primär vertikal)
- Einfachere und präzisere Planung/Schätzbarkeit
- Parallele und getrennte Entwicklung möglich
- Austauschbare/Mehrere Clients möglich
- Unterschiedliche Schnittstellen möglich

Negative Effekte:

- Komplexität des ganzen Systems wird grösser.
- Mehr Schnittstellen, mehr Aufwand, mehr Planung!

4.3 Service-Oriented-Architecture

Services kapseln sauber abgegrenzte Sub-Domänen in eigenständige, verteilte Dienste (Services), die dann von übergeordneten Applikationen zur Realisierung eines Businessprozesses genutzt werden können.

- Ein Service verfügt über eine wohldefinierte Schnittstelle.
- Die Services sind in einem Verzeichnisdienst eingebunden und werden dynamisch gesucht und gebunden (binding).
- Die Kommunikation kann über (fast) beliebige Protokolle erfolgen (RPC, RMI, CORBA, HTTP, SOAP, REST, etc.)

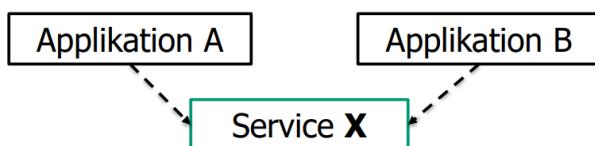
Mit SOA gehen wir in der Abstraktion und Granularität also noch mal eine Ebene höher. Eine zu grosse Applikation kann in einzelne Services aufgebrochen, oder Teile davon in Services herausgebrochen werden.

Beispiel: Ein (zu) grosses System, das die komplette Warenbewirtschaftung mit Bestellungen, Kunden, Artikel und Lager verwaltet.

- Einfache Erkenntnis: Kunden und Artikel sind zwei sehr lose (Sub-)Domänen, die von fast allen Prozessen benötigt werden.
- Einfache Lösung: Man löst diese "Stammdaten" in eigene Services heraus.

Eine Service Oriented Architecture führt implizit auch zu einer fachlichen Entkopplung. Wenn beispielsweise zwei Applikationen (Domänen) dieselben Daten benötigen, können nicht beide der Master sein. Dies würde zu einer inakzeptablen, zyklischen Abhängigkeit führen (A hängt von B ab, B hängt von A ab).

Lösung: Die gemeinsamen Teile als Subdomäne in einen neuen Service auslösen:



4.3.1 SOA - Konsequenzen

Mit SOA machen wir definitiv den Schritt zu (auch horizontal) verteilten Applikationen! Das bringt einige Konsequenzen mit sich:

- Verteilte Datenhaltung
- Laufzeitabhängigkeiten
- Globale/verteilte Transaktionen
- Unterschiedliche Plattformen / Sprachen
- Logging und Tracing
- Versionierte Schnittstellen

4.4 Message- oder Event-Driven-Architecture

Auf der tiefsten Programmierebene ist Ereignissesteuerung (Events!) schon lange etabliert und verstanden. Beispiel: Ein Mausklick auf einen Button löst einen Event aus, auf welchen sich beliebige Klassen/Komponenten registrieren, und somit darauf reagieren können.

Hauptvorteil und Motivation: Lose Kopplung!

- Empfänger und Sender eines Events müssen sich nicht kennen.
- Einzig über die Semantik des Events (Ereignis) muss Einigkeit herrschen.

Bei der Message- oder Event-Driven Architecture setzen wir dieses Prinzip auf einer höheren Abstraktionsebene (Architektur) ein!

4.4.1 Fachlicher nutzen

Bei SOA manifestieren sich die Abhängigkeiten zwischen den Services meistens in einer synchronen Kommunikation.

Es gibt aber viele fachliche Situationen, in denen keine unmittelbare, synchrone fachliche Abhängigkeit besteht:

- Es wurde ein neuer Artikel erfasst / geändert / gelöscht.
- Es ist soeben die 1-Millionste Bestellung eingegangen.

Bei diesen Ereignissen sind Aktionen denkbar, die zwar erfolgen sollen, die aber auch asynchron zu einem späteren Zeitpunkt erledigt werden können.

4.4.2 Messages (Events) als Architekturmittel

Wo bisher persistente, klassische Datenbanken für den Informationsaustausch verwendet wurden, können Message-Systeme diese mindestens* unterstützen.

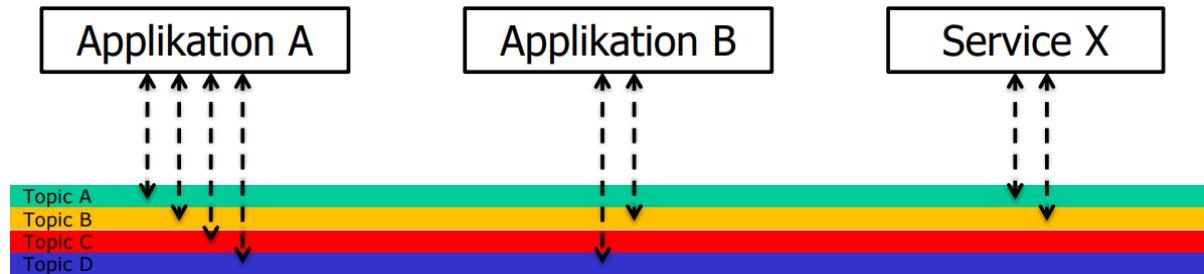
- Systeme deutlich und stärker entkoppeln und vereinfachen.
- Unterstützen mehrheitlich auch Persistenz.

Beispiele von Produkten: RabbitMQ, Kafka, MQSeries, etc.

4.4.3 Enterprise Service Bus (ESB)

Man kombiniert die Idee der Services mit dem Messaging. Man stellt einer Menge von Services einen sogenannten Message Bus (nichts anderes als eine Sammlung differenzierter Topics oder Queues) zur Verfügung.

- Services registrieren sich für 1..n Topics
- Services publizieren Messages (Events) für 0..n Topics.
- Es ergibt sich eine sehr lose Kopplung
- Eine Reaktion kann oft auch asynchron erfolgen.



Die Entkopplung wird auf vielfältige Weise und markant verbessert:

- Meistens asynchrone Kommunikation - schneller
- Fire & Forget - einfacher
- Bei einem Ausfall wird (persistent) gequeued - robuster.
- Neue Applikationen können sich frei registrieren - flexibler.
- Das System kann elegant skalieren - leistungsfähiger.

Die Herausforderungen dieser Architektur:

- Welche Topics werden erstellt?
- Welche Messages / Events existieren?
- Welche Granularität haben die Messages bzw. Events?
- Verfügbarkeit des Message-Bus zentral!

4.5 Zwischenbilanz / Zusammengefasst

Heute haben wir einen sehr umfangreichen und tollen Werkzeugkasten um für fast jede Problemstellung eine wirklich passende, gute Architektur zusammenstellen zu können.

Was ganz sicher ist: Keines der vorhandenen (und auch zukünftigen) Werkzeuge kann jemals für sich alleine die perfekte Lösung für jedes Problem sein.

Die Modularisierung ist dabei fundamental wichtig, aber letztlich auch "nur" ein Rad im gesamten Getriebe.

Bei jüngeren Hype-Themen wie "Microservices" oder "Stream-Processing" oder "Reactive-Design", etc. fallen aber sehr viele (und immer wieder) auf wundersame Versprechen herein.

5 Microservices

Die Grundidee: **Man schneidet die Services "einfach" noch kleiner**

Man unterteilt eine Domain in kleinere, voneinander möglichst unabhängige "Teildomänen" → "bounded context". Microservices nähern sich in der Granularität somit den Modulen, sorgen dort aber für harte Modularität.

Durch getrenntes Deployment wird eine Architekturverletzung durch eine unerlaubte Kopplung unmöglich, bzw. stark erschwert, bzw. wiederum deutlich sichtbar.

Gemäss Martin Fowler:

"In short, the microservice architectural style is an approach to developing a **single application** as a **suite of small services**, each **running its own process** and **communicating** with lightweight mechanisms, often an **HTTP resource API**. These services are **built around business capabilities** and **independently deployable** by **fully automated deployment machinery**."

Etwas strukturierter formuliert:

1. Eine Applikation wird aufgeteilt in mehrere, kleine Services.
2. Jeder Service läuft in eigenem Prozess / auf eigener Plattform.
3. Leichtgewichtige Kommunikation (meist RESTfull/http/JSON).
4. Voneinander unabhängig deploybar (somit auch Release).
5. Automatisiertes Deployment (DevOps, PaaS, IaaS).

5.1 Aufteilung einer Applikation in kleine Microservices

Man teilt eine Applikation (primär vertikal) in mehrere, möglichst eigenständige Teile auf. Diese Teile sollten auf eigene Daten zurückgreifen können.

Das erfordert typisch ein Aufbrechen des Domänenmodells in verschiedene "bounded context"*. Dies ist grundsätzlich gut für das Design, aber nicht einfach. Die Datenhaltung wird somit auch getrennt.

* Bounded Context - Definition

Ein Bounded Context bezeichnet einen klar definierten, abgeschlossenen Bereich einer Domäne, in dem bestimmte Begriffe, Modelle und Regeln gelten. Innerhalb eines Bounded Context wird ein konsistentes Domänenmodell verwendet, das speziell für diesen Kontext entwickelt wurde. Das bedeutet, dass Begriffe, Daten und Prozesse in diesem Kontext eine eindeutige Bedeutung haben und unabhängig von anderen Kontexten behandelt werden.

In der Welt der Microservices repräsentiert ein Bounded Context in der Regel einen einzelnen Microservice, der seine eigene Logik, Daten und Regeln kapselt.

Die einzelnen Teile sollten möglichst nicht (direkt) miteinander kommunizieren müssen, sondern werden direkt über das GUI orchestriert.

5.2 Jeder Service hat eigenen Prozess / Plattform

Die einzelnen Services laufen als eigenständige Prozesse.

- Können unterschiedliche Plattformen (OS, Programmiersprache, etc.) haben, typisch in einem virtualisierten Container.
- Plattformen müssen also nicht für die ganze Applikation identisch sein und können auch schrittweise geändert werden.

Die Services laufen in echter Parallelität als verteilte Applikation

- Somit ganzes Potential und alle Herausforderungen von verteilten Systemen: Kommunikation über Netzwerk, Latenz, Skalierung, Ausfall, etc.
- Nutzt einen Client mehrere Services asynchron (oder nebenläufig synchron), erhöht sich natürlich die Performance.

Insgesamt erhöht sich aber die Komplexität des Systems.

5.3 Leichtgewichtige Kommunikation

Derzeit sind auf JSON basierende REST-Schnittstellen aufgrund ihrer Einfachheit sehr populär (nach XML). Man darf kritisch fragen, ob JSON/REST-basierende Schnittstellen wirklich so "leichtgewichtig" sind, wenn man sie mit schnellen, effizienten binären Protokollen (z.B. RMI, [g]RPC, etc.) vergleicht.

Ohne Zweifel ist eine auf http(s) basierende Kommunikation relativ leicht zu implementieren und auch automatisiert testbar. Authentifizierung und Verschlüsselung ist damit ebenfalls bereits abgedeckt. Gute Akzeptanz im Operating, bestehende, bekannte Protokolle.

5.4 Voneinander unabhängig Deploy- und Releasebar

Microservices sind eigenständige Projekte und Releaseeinheiten, und werden erst durch die gemeinsame Orchestrierung zu einer Applikation. Kleinere Einheiten können einfacher und flexibler entwickelt werden, Änderungen können mit kleinerem Risiko durchgeführt und Erweiterungen leicht ergänzt werden (OCP - Open/Closed Principle).

Unabhängig Deploybar heißtt, dass somit auch nur einzelne Microservices einer Applikation (re-)deployed werden können. Die Applikation muss somit damit umgehen können, dass einzelne Teile/Services ausfallen und nicht verfügbar sind.

Ausfälle werden häufiger! Hat jeder von (nur) fünf Services eine Verfügbarkeit von 95%, dann resultiert daraus eine Gesamtverfügbarkeit von $0.95^5 = 77\%$ → Resilienz wird wichtig!

5.5 Deployment automatisiert (DevOps)

Bei klassischen, monolithischen Deployments von "schwergewichtigen" Applikationen (z.B. in Jakarta EE Containern) war ein (halb-)manuelles Deployment und Start-/Stop-Sequenzen häufig.

Ersetzt man diese Applikation durch 10 Microservices ist das manuelle Deployment unhaltbar: Alle Abläufe würden sich um den Faktor 10 (pessimistisch betrachtet) verlängern. Je feiner ein System aufgeteilt ist, je mehr Microservices.

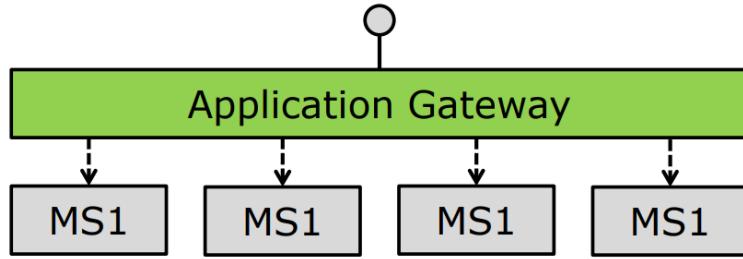
Weitgehende Automatisierung (mit schnellen Start-/Stop-Sequenzen) ist für Microservices ein absolutes Muss-Feature!

5.6 Schnittstellen und Kommunikation

Zentrale Herausforderung sind und bleiben die Abhängigkeiten, welche bei Microservices primär durch Kommunikation entstehen.

Microservices forcieren (rein technisch) die Modularisierung, verlagern die Herausforderungen aber ins Deployment (Operating).

5.6.1 Gateway-Pattern mit Microservices



Damit ein Client nicht jeden einzelnen Microservice kennen muss, nutzt man das Gateway-Pattern. Das Gateway entspricht dem Konzept des Fassaden-Patterns (GoF) auf Ebene der Architektur. Er vereinfacht die Implementation des Clients deutlich. Zusätzliches Potential:

- Übernimmt auch Authentifizierung und Verschlüsselung
- Kann einfache Mappings / Transformationen von Daten vornehmen (wenn nötig).

Der Gateway wird explizit als eigener Service implementiert, es gibt aber auch generische Gateways.

5.6.2 Inter-Kommunikation zwischen Microservices

Eine direkte Kommunikation zwischen Microservices ist zwar möglich, sollte aber wenn immer möglich vermieden werden. Direkte Kommunikation bedeutet starke Kopplung, sowohl im Design als auch zur Laufzeit!

Deshalb entlehnen wir uns eine tolle Technik aus der Message-Oriented Architektur: Wir nutzen Queues und/oder Topics!

- Wenn immer möglich asynchron!
- Events/Messages und Commands.
- Synchrone Kommunikation auch hier möglichst vermeiden.

Neue Services können sich extrem elegant und lose gekoppelt registrieren, und so neue Funktionalität dynamisch ergänzen.

5.7 Anforderungen an die Resilienz

Resilienz im technischen Kontext:

Die Fähigkeit von technischen Systemen, bei Störungen bzw. Teilausfällen nicht vollständig zu versagen, sondern wesentliche Systemdienstleistungen aufrechtzuerhalten.

Bezogen auf Microservices: Eine Applikation sollte damit umgehen können, und somit auch (eingeschränkt) weiter funktionieren, wenn einzelne Microservices temporär nicht verfügbar sind.

Netflix hat eine spezielle Methode, die Resilienz zu fördern: github.com/Netflix/chaosmonkey

5.7.1 Was passiert wenn ein Microservice ausfällt?

- Wenn wir viele Microservices haben, müssen wir damit umgehen können, dass einzelne Services ausfallen und temporär nicht verfügbar sind (z.B. auch wegen Redeployment).
- Szenario 1: Ein Service ist komplett weg / oder wird nicht gefunden. Verbindung kann nicht aufgebaut werden, Fehlersituation wird vom Aufrufer relativ schnell erkannt → Abbruch, klare Situation.
- Szenario 2: Ein Service arbeitet extrem langsam, Antwortzeiten erhöhen sich zunehmen. Verbindung kann aufgebaut werden, Aufrufer ist wesentlich länger blockiert (bei synchron), läuft aber nicht in ein Timeout.
- Das Problem schaukelt sich selber hoch und der Ressourcenverbrauch steigt umso schneller an.

5.7.2 Microservice Pattern: Circuit-Breaker

Situation: Zwei Services kommunizieren synchron miteinander. Beispiel: API-Gateway zu Port-Microservice.

Man legt einen transparenten circuit-breaker-Proxy dazwischen:

- Dieser prüft permanent, wie lange die Requests dauern.
- Wird ein Zeitlimit überschritten, unterbricht der Proxy die Verbindung und bricht alle neuen Requests sofort ab (fail-fast).
- Nach einem konfigurierbaren Timeout öffnet sich der Proxy wieder und prüft erneut.
- Der Vorgang kann sich beliebig wiederholen.

Vorteil: Bei einer Störung/Ausfall werden weniger Ressourcen verschwendet, Aufrufer brechen schneller ab.

5.8 Beispiele für Technologien und Frameworks

Ich fasse hier nur die im Projekt verwendeten zusammen (und einige mehr), aber nicht alle.

- Sprint Framework: Starke Vereinfachung für die Erstellung von Microservices. Schwäche: Basiert weitgehend noch immer auf Laufzeit-Konfiguration
- Micronaut.io [VERWENDET]: Ist spezialisiert auf Microservice und Serverless Applikationen. Hat Compile-Time-Annotations (Code wird während Kompilation erzeugt, kleinere Runtime, schnellere Startzeiten). Starke und flexible Integration von Infrastrukturdiensten.
- GraalVM: Alternative Runtime-VM. Hat die Fähigkeit Applikationen nativ (kompiliert, plattform-spezifisch) zu erzeugen, und somit deutlich kleiner und schneller zu machen. Startzeiten von Sekunden auf Mikrosekunden!
- Quarkus: Kubernetes Stack für auf OpenJDK oder GraalVM basierende Applikationen. Kompiliert die Applikation und somit native Ausführung. Optimiert für Kubernetes, ebenfalls "small footprint".

5.9 Deployment von Microservices

Für klassische, eher "schwergewichtige" Services/Server, stehen viele Möglichkeiten offen:

- Eigene, dedizierte, spezialisierte HW
- Voller VM-Stack mit eigenem Betriebssystem
- Individuell gepflegte, spezifische Docker-Container

Wenn ein System aber aus vielen einzelnen Microservices besteht, ist das nicht mehr realistisch.

- Ressourcenverbrauch viel zu hoch, wenn pro Service ein Prozess läuft.
- Arbeitsaufwand für Erstellung, Wartung und Betrieb viel zu hoch, muss zwingend automatisiert werden.

Technologien wie Docker-Images und -Container, oder gar server-less sind absolut zwingend → DevOps.

5.10 Umgang mit Transaktionen

Es gibt eine überraschend klare Erkenntnis: **Microservices und Transaktionen vertragen sich nicht!**

Beste Lösung: Keine globalen Transaktionen!

- Transaktionen verursachen eine enge, semantische Kopplung.
- Das wollen wir bei Microservices vermeiden!

Mögliche Lösungsansätze:

- Transaktionen aufbrechen und mit dem SAGA-Pattern* arbeiten.
- Getrennte Services zusammenfassen
- Fachliche Umgestaltung: Transaktionen minimalsieren (was muss wirklich unbedingt in einer Transaktion?).

* Das SAGA-Pattern:

Das SAGA-Pattern ist ein Ansatz zur Verwaltung von verteilten Transaktionen in Microservices-Systemen, ohne auf klassische, globale Transaktionsmechanismen wie ACID zurückzugreifen. Eine SAGA besteht aus einer Reihe von lokalen Transaktionen, die jeweils in einem einzelnen Service ausgeführt werden. Wenn eine Transaktion fehlschlägt, werden Kompensationssaktionen (eine Art "Rückgängig machen") ausgeführt, um das System in einen konsistenten Zustand zu bringen. Es gibt zwei Varianten: Choreografie, bei der Services durch Events miteinander kommunizieren, und Orchestrierung, bei der ein zentraler Koordinator den Ablauf steuert. Dadurch wird das System skalierbarer, fehlertoleranter und besser an Microservices-Architekturen angepasst.

5.11 Logging, Metrics und Tracing

Durch die Realisierung von Applikationen als Microservices, welche feingranularer auf- und verteilt werden, stellen sich neue Herauforderungen:

- Durchgängiges, konsistentes, einheitliches Logging von fachlichen und systemtechnischen Ereignissen.
- Durchgängige Verfolgbarkeit von Geschäftsprozessen über mehrere Schritte in verteilten Services und Systemen.
- Überwachung der Performance und der Ressourcen zur Feststellung von Engpässen in einzelnen Services oder bei Teilausfällen.

5.11.1 Logging

Das OWASP (Open Web Application Security Project) stellt regelmäßig eine Rangliste der wichtigsten Sicherheitsrisiken von Webapplikationen zusammen. In den top 10 war auch "Security Logging and Monitoring Failures".

Kernaussage: Angriffe werden mangels Überwachung und Logging gar nicht (oder erst zu spät) bemerkt.

Hauptfehler: Essentielle Ereignisse werden entweder gar nicht, unzureichend, unverständlich oder nur lokal geloggt, bzw. ungenügend überwacht und ausgewertet.

Es lohnt sich in jedem Fall, ein einfaches, minimales Logging-Konzept festzulegen (Konvention)

- Was wird geloggt: Welche Ereignisse mit welchem Level und mit welcher Granularität.
- Welches Log-Format wird verwendet (möglichst einheitlich).

Eine Umsetzung als (eigene) Basis-Library ist nur beschränkt sinnvoll, weil das die Logging-Performance massiv drücken kann.

Die Einhaltung von reinen Konventionen verlangt umgekehrt eine sehr hohe Eigendisziplin aller Entwickler. Aktive Reviews, intensive Tests, laufende Nachbesserung!

Ein bekanntes Logging-Framework ist Apache Log4J.

5.11.2 Metriken

Laufzeitmetriken sind explizit in die Software eingebaute (programmierte) Messpunkte, welche einen Rückschluss auf die Leistung eines (Teil-)Systems erlauben.

Es gibt verschiedene Arten von Messpunkten. Einige Beispiele:

- Einfacher Zähler (counter): Anzahl Ereignisse kumulieren.
- Messwert (gauge): Absoluter Wert (z.B. Elemente in Queue).
- Meter (meter): Messwert/Zeiteinheit (z.B. Ereignisse pro Minute).
- Histogramme: Statistische Verteilung von Messwerten (z.B. Mittelwert, Min, Max, etc.)

Erste Herausforderung: Auswahl eines geeigneten (sprich aussagekräftigen) Messpunktes für eine bestimmte Grösse.

Es gibt Metrics als Teil des DropWizard-Frameworks.

5.11.3 Tracing

Problemstellung: Wenn ein Service eine Anzahl von Folgeaktivitäten aufruft, und diese asynchron und durch Queues verzögert ausgeführt werden, ist eine Fehler-Verfolgung des gesamten Request schwierig.

Lösungsansatz: Man verpasst jedem Client-Request einen Fingerabdruck (technisch: Correlation-ID) und reicht diesen konsequent an alle Unter- bzw. Folgeaufrufe weiter.

Beispiel für Collreation IDs: UUID (Universally Unique IDentifier). Eine UUID besteht aus einer 16-Byte-Zahl, die hexadezimal notiert und in fünf Gruppen unterteilt wird. Konkretes Beispiel: "4e6abea6-d18a-49c1-a7b0-4f57702e6602".

5.12 Service Discovery

Wenn wir viele unterschiedliche Dienste haben, die alle einzeln deployed werden (unterschiedliche IPs oder Ports), stellt sich die Frage, wie sich die Dienste gegenseitig finden.

Lösung: Spezielle Service Discovery Dienste (schlank!).

Sehr vereinfacht erklärt funktionieren Service-Discoverives wie lokale DNS Server. Sie ordnen logische Namen einem Service zu (IP und Port). Diese Service-Discoveries können auch eine Art Load-Balancing übernehmen, indem Sie die registrierten Services selbstständig monitoren und Anfragen zählen.

6 Modelle

6.1 Anforderungsdiagramme & -modelle

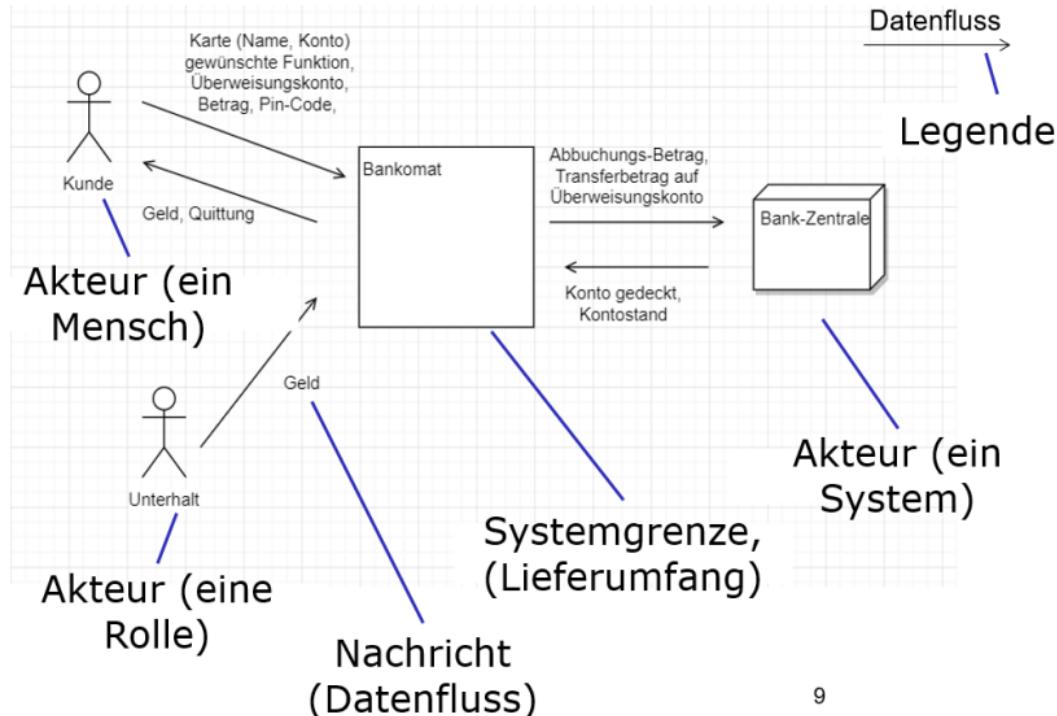
Wir können folgende Methoden/Techniken unterscheiden, um die Anforderungen zu erheben, festzuhalten und zu analysieren.

- Domain-Modell
- Kontext-Diagramm
- Use Case Modell und Beschreibungen
- Geschäftsprozess-Modell
- Feature-Listen
- User Stories

6.2 Kontext Diagramm

Im Kontextdiagramm sehen wir:

- Das System mit seinen Grenzen (hier der Bankomat)
- Alle Akteure
- Alle Nachrichten (Datenfluss) zwischen den Akteuren und dem System

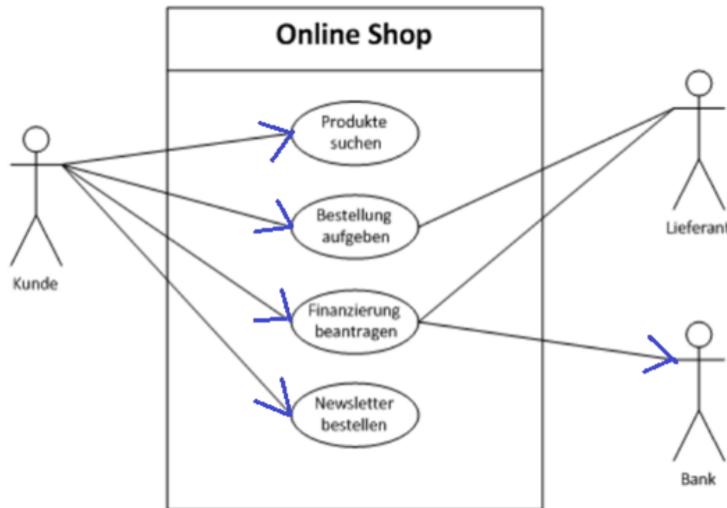


9

6.3 Anwendungsfall Diagramm

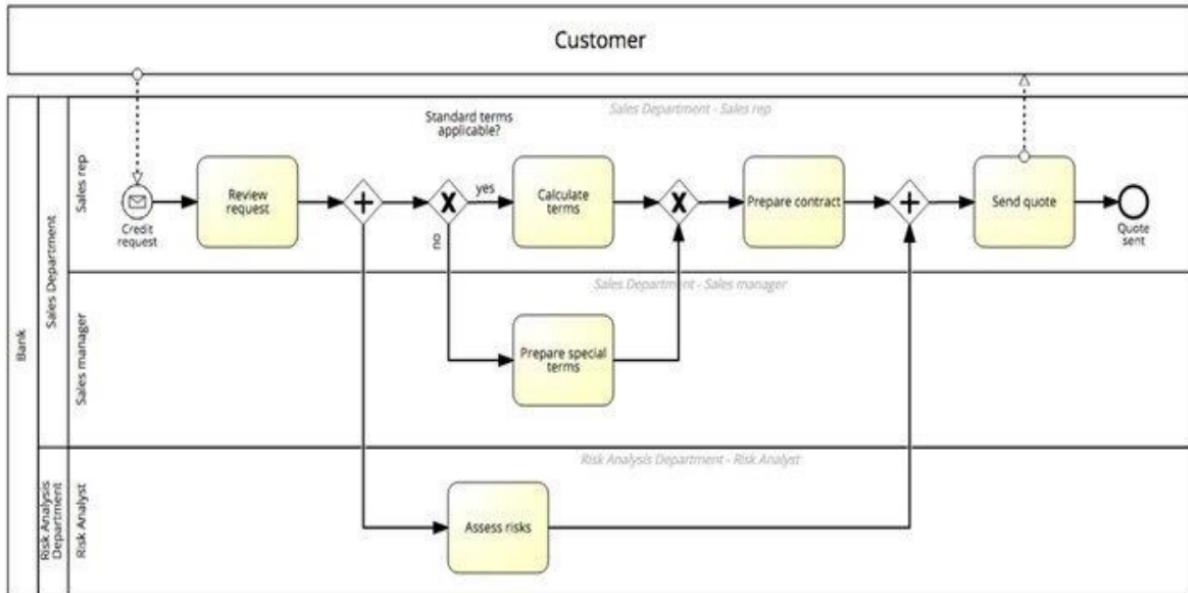
Ein Anwendungsfall bündelt alle möglichen Szenarien, die eintreten können, wenn ein Akteur versucht, mit Hilfe des betrachteten Systems ein bestimmtes fachliches Ziel (engl. business goal) zu erreichen. Er beschreibt, was inhaltlich beim Versuch der Zielerreichung passieren kann und abstrahiert von konkreten technischen Lösungen.

Im Use Case Diagramm werden sämtliche Use Cases / Anwendungsfälle und die Verbindungen zu den Akteuren aufgezeigt. Es empfiehlt sich mittels Pfeilen einzulegen, wer die Use Cases anstösst.

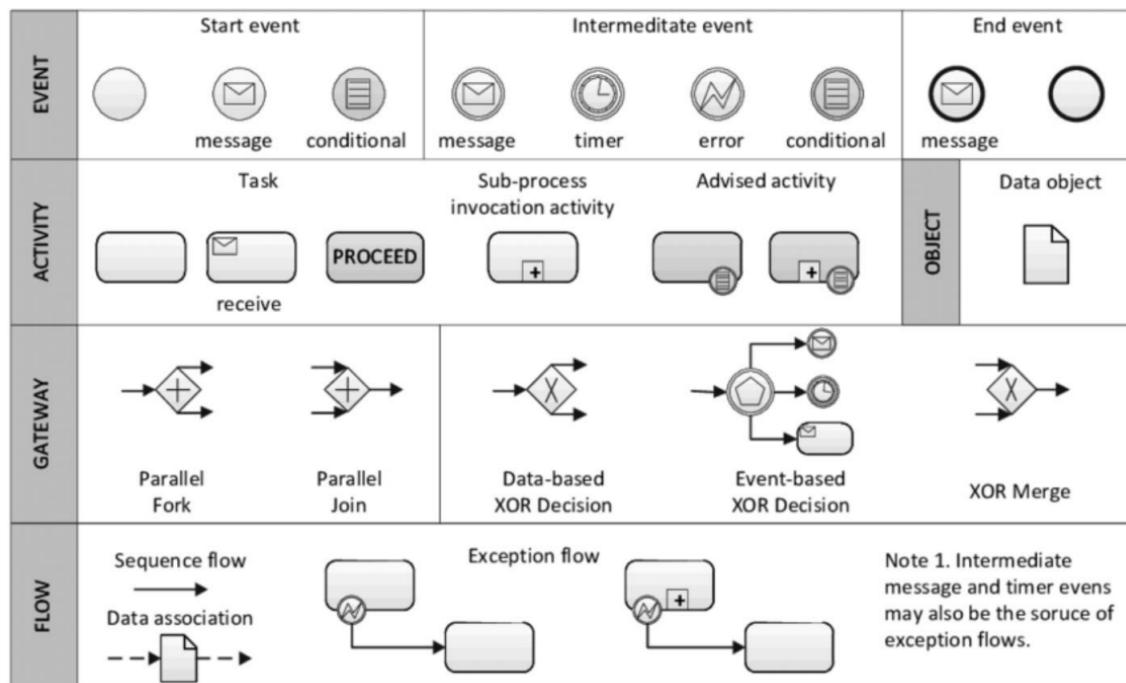


6.4 Geschäftsprozess Modell

Die Struktur spiegelt im Wesentlichen die zeitlich-sachlogische Abfolge der betrachteten Funktionen wider. Auf Grund ihrer allgemeinen Modellcharakteristik dienen Geschäftsprozessmodelle der Dokumentation, Analyse und Gestaltung von Geschäftsprozessen, als Grundlage für die automatisierte Bearbeitung oder Unterstützung von Prozessen sowie zur Unterstützung der Kommunikation über Geschäftsprozesse.



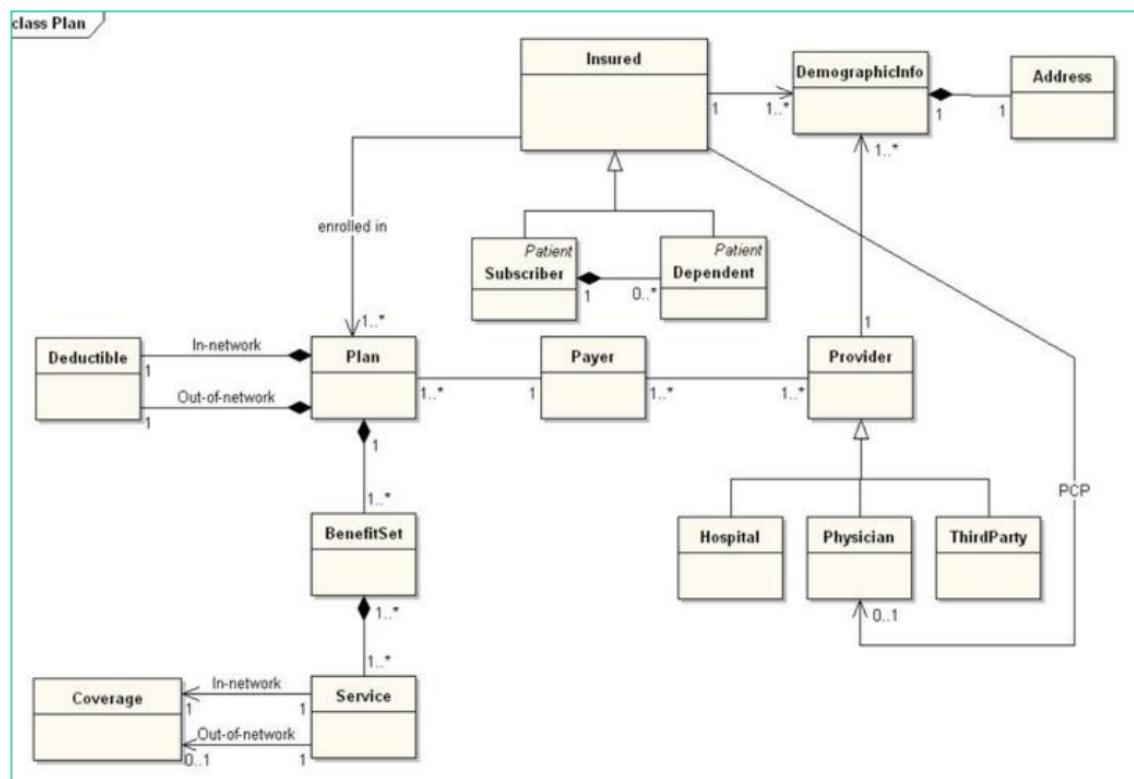
6.4.1 BPMN - Business Process Management Notation



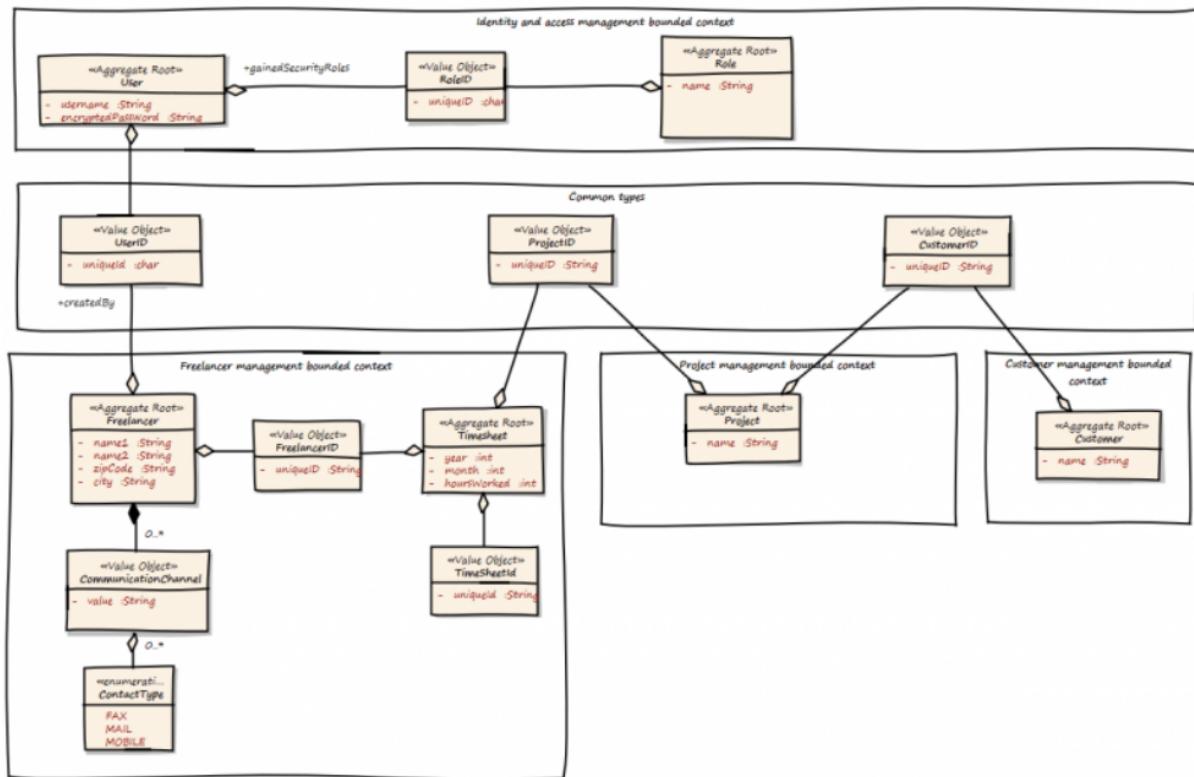
6.5 Domain Modell

Anwendungsdomäne (Fachdomäne, Problemdomäne, Domain): ein abgrenzbares Problemfeld oder ein bestimmter Einsatzbereich für ein Computersystem oder ein Software.

Ein Domain-Modell ist ein konzeptionelles Modell der Anwendungsdomäne und beinhaltet sowohl Verhalten als auch Daten. Es wird typischerweise als Klassendiagramm erstellt.



6.5.1 Domain Model mit Kontexten

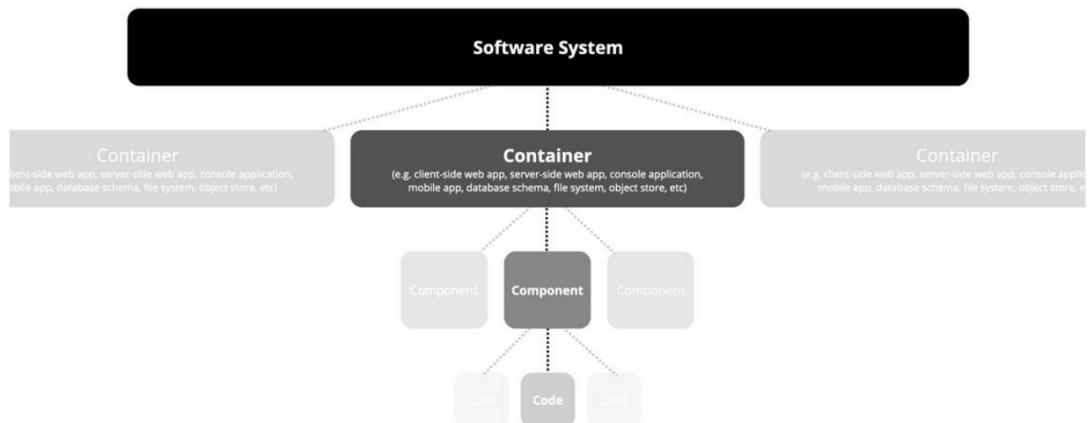
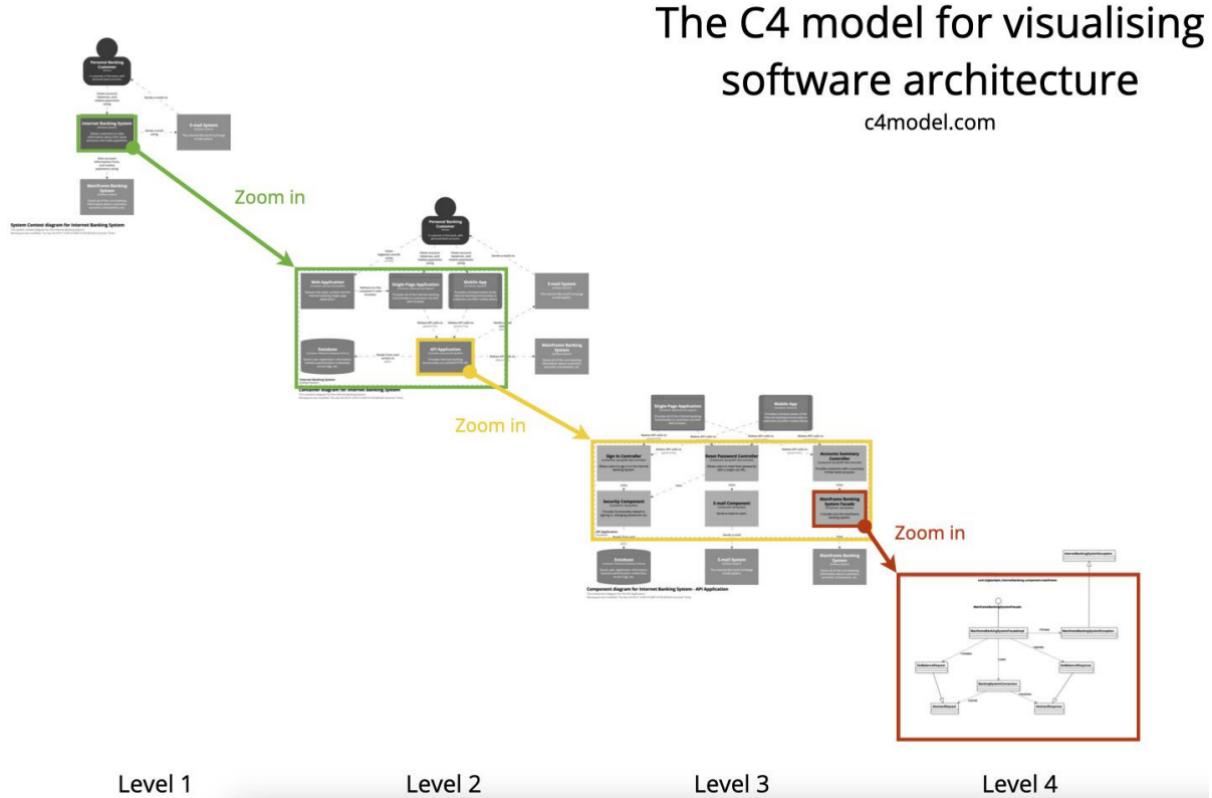


6.6 Feature-Liste

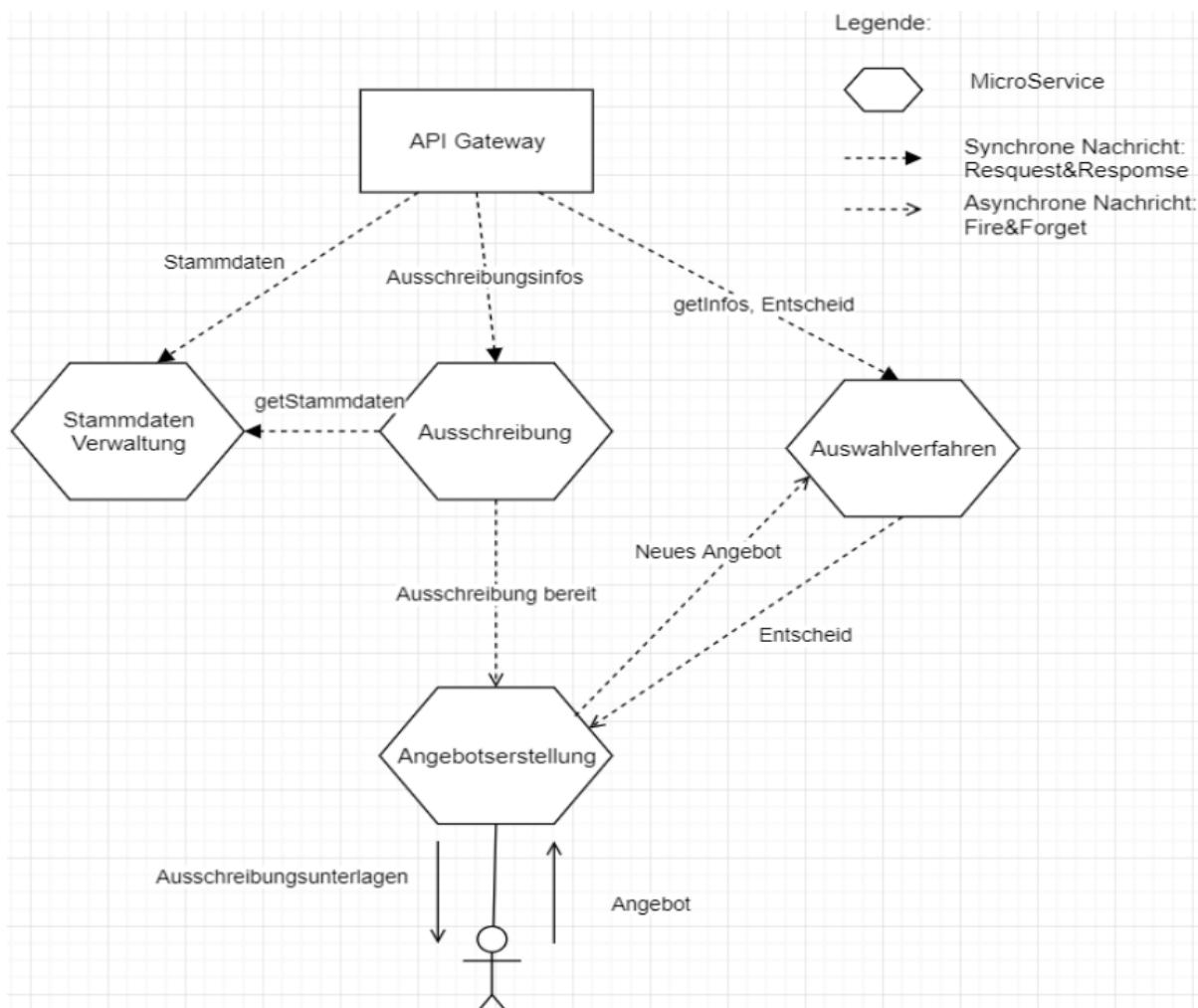
Feature Use Right	RECOMMENDED		
	Professional	Basic	Essential
View Announcements	X	X	X
Manage Shared Views	X	X	X
Use Relationships between records	X	X*	X*
Create Personal Views	X	X	X*
Advanced Find Search	X	X	X*
Search	X	X	X*
Use a Queue Item	X	X*	X*
Export Data to Microsoft Excel	X	X	X
Perform Mail Merge	X	X	X
Start Dialog	X	X*	X*
Run as an On-Demand Process	X	X*	X*

6.7 C4 Modell

Das C4 Model (Context, Containers, Components, and Code) ist für das Visualisieren von Software Architekturen.



6.8 Microservice Architektur Diagramm [SWDA]



Benefits:

- Welche Microservices gibt es?
- Wie sieht die Kommunikation dieser aus? (Synchro vs Asynchron)?
- Welche Daten werden ausgetauscht?

Falls ein verwendetes Symbol für einen Block oder eine Beziehung oder eine Interaktion nicht absolut klar ist, MUSS eine entsprechende Legende direkt bei der Skizze hinterlegt werden (siehe oben rechts im Bild).

6.9 API Specification

Die OpenAPI Specification setzt einen Standard für HTTP APIs. Dies hilft anderen Entwicklern die API besser zu verstehen und dadurch folgt: schnellere Client Entwicklung, einfaches testen, etc.

7 Methodik

7.1 Prozessanalyse mit Event Storming

$$\text{BrainStorming} + \text{DomainEvents} = \text{EventStorming}$$

Event Storming ist eine Workshop-basierte Methode, um schnell herauszufinden, was in der Domäne eines Softwareprogramms passiert.

Sie wurde im Rahmen des domain-driven-Designs (DDD) erfunden.

Die Grundidee ist, Softwareentwickler und Domänenexperten zusammenzubringen und voneinander zu lernen.

Um diesen Lernprozess zu erleichtern, soll Event Storming Spass machen.

7.1.1 Schritt für Schritt

1. Identifiziere Domain Events (orange Post-its):

- Beginne mit den Domain Events - das sind wichtige Ereignisse, die in der Domäne stattfinden.
- Schreibe jedes Event auf ein eigenes orangefarbenes Post-it und platziere es auf einer zeitlichen Achse, die den Ablauf der Ereignisse darstellt.
- Beispiel: "Bestellung wurde aufgegeben", "Zahlung erfolgt".

2. Füge Commands (blaue Post-its) hinzu:

- Überlege, welche Commands (Befehle) die Domain Events auslösen.
- Commands repräsentieren Aktionen, die von einem Akteur ausgelöst werden.
- Schreibe jeden Command auf ein blaues Post-it und verbinde es mit dem entsprechenden Domain Event
- Beispiel: "Bestellung aufgeben" → führt zu "Bestellung wurde aufgegeben"

3. Definiere Akteure (gelbe Post-its):

- Identifiziere, wer die Commands auslöst, also die Akteure.
- Schreibe jeden Akteur auf ein gelbes Post-it und platziere es in der Nähe der zugehörigen Commands.
- Beispiel: Ein Kunde löst das Command "Bestellung aufgeben" aus.

4. Markiere Aggregates (gelbe Post-its):

- Bestimme die Aggregates, die als zentrale Domänenobjekte agieren und die Commands und Events kapseln.
- Aggregates repräsentieren wichtige Teile der Domäne, die Zustandsänderungen durch Commands und Events verarbeiten.
- Schreibe den Namen des Aggregates auf ein separates Post-it (gelb) und verknüpfe es mit den zugehörigen Commands und Events
- Beispiel: Das "Bestellungs-Aggregat" verarbeitet den Command "Bestellung aufgeben".

Domain event

An event that occurs in the business process. Written in past tense.

User

A person who executes a command through a view.

Business process

Processes a command according to business rules and logic. Creates one or more domain events.

Command

A command executed by a user through a view on an aggregate that results in the creation of a domain event.

Aggregate

Cluster of domain objects that can be treated as a single unit.

External system

A third-party service provider such as a payment gateway or shipping company.

View

A view that users interact with to carry out a task in the system.

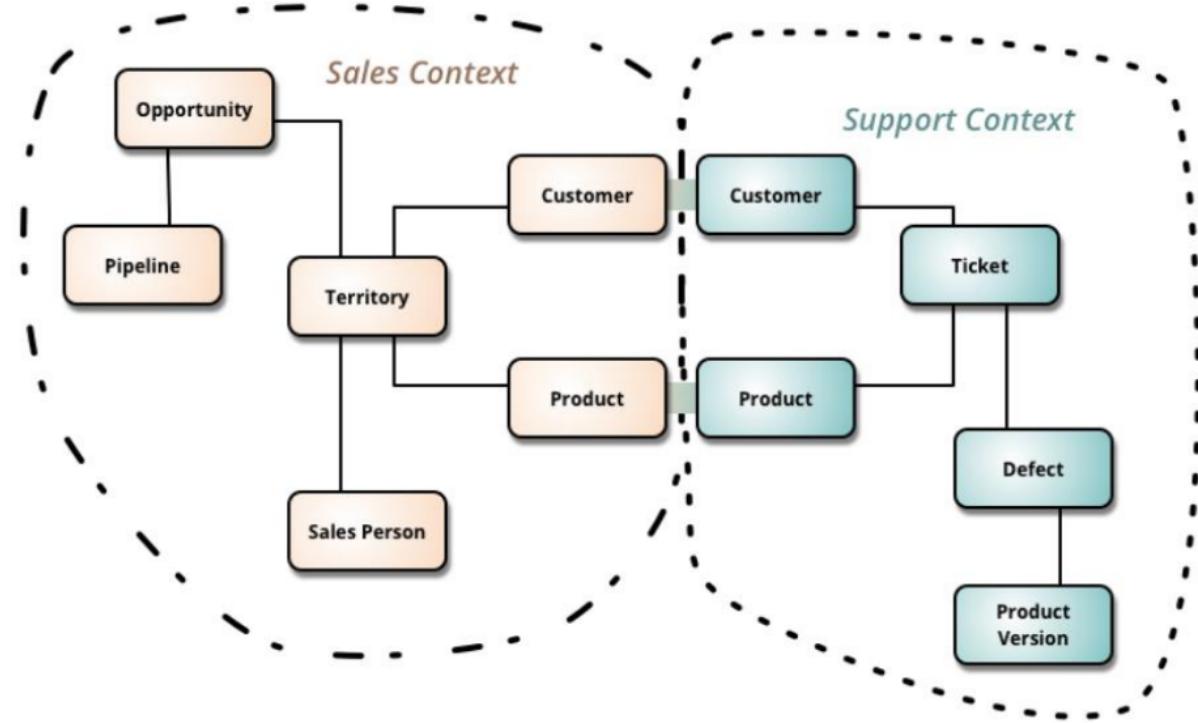
Das Event Storming Erebnis sieht dann wie folgt aus:



7.2 Wie finde ich "meine" Microservices?

7.2.1 Bounded Context

In Anlehnung an die Methode "Domain Driven Design" ist dies ein abgegrenzter Bereich des Domain-Models, der Gültigkeit für ein bestimmtes fachliches Modell hat. Die meisten Domänen werden aus mehreren Bounded-Contexts bestehen.



Hier haben zum Beispiel Customer und Produkt jeweils eine andere Bedeutung für den Sales- und den Support Context.

8 API-Design

Ein API ist eine Schnittstelle die explizit darauf ausgelegt ist, eine Softwareeinheit (Modul, Library, System) nutzbar zu machen.

- Soll eine möglichst einfache Nutzung ermöglichen.
- Soll den Aufwand für die Nutzer minimieren.
- Soll Unabhängigkeit von einer konkreten Implementation bieten.
- Hat ein implizites Mass an Öffentlichkeit.

Die API wird typisch von Nutzern verwendet und von Anbietern implementiert. Beispiele: JPA (Java Persistence API) als Schnittstelle zur Persistierung.

8.1 Wann wird eine Schnittstelle zum API

Eine (rein technisch betrachtet) identische Schnittstelle kann:

- Im (Detail-)OO-Design in einer Implementation existieren.
- Die Schnittstelle zu einer Komponente darstellen.
- Die Schnittstelle zu einem Framework darstellen.
- Die Schnittstelle zu einem Teilsystem darstellen.

Die Definition welchen Scope und somit welche Wichtigkeit eine Schnittstelle hat, erfolgt auf Architektur-Ebene.

8.2 Motivation für gute APIs

Gute APIs haben einen grossen Wert:

- Fördern die Modularität und Entkopplung.
- Machen komplizierte Dinge möglichst einfach nutzbar.
- Machen die Implementation austauschbar.
- Gute, einfache APIs machen ein Produkt attraktiv.

Schlechte APIs verursachen Ärger und Kosten:

- Komplizierte, missverständliche Nutzung, Fehlersuche, Support.
- Verführen dazu, eine bereits existierende, gute Lösung nicht wiederzuverwenden, sondern wieder selber zu machen.
- Verführen zu komplizierten Umwegen und unnötigen Fassaden.

8.3 API - Herausforderungen

Öffentlich verfügbare APIs leben (fast) ewig!

- Somit leben alle im API-Design gemachten Fehler auch ewig.
- APIs sind nur mit grossem Aufwand / Konsequenzen änderbar.
- Meist hat man nur eine Chance es richtig zu machen.

Es ist viel einfacher ein schlechtes API zu erkennen, als ein wirklich gutes API selber zu entwerfen.

Grosse Kunst: API so entwerfen, dass sie entwicklungsfähig ist. Geheheimnis ist das KISS- und YAGNI-Prinzip mit gesundem Mass.

- KISS - Keep it Simple / Stupid
- YAGNI - You ain't gonna need it

8.3.1 Hauptziele einer guten API

- Maximale Entkopplung (lose Kopplung) der Implementation.
- Maximales Information Hiding (keine Implementationsdetails).
- Maximale Kohäsion.

8.4 API Qualitätsanforderungen

- Konsistenz: Namensgebung und -regeln konsequent einhalten.
- Namensgebung: Einfache, eingängige, treffende Namen wählen.
- Verhalten: Klare Erwartungen erfüllen, ohne Nebeneffekte.
- Erweiterbarkeit: Offen für Weiterentwicklung (der API).
- Dokumentation: Einfache, hilfreiche, kompakte Dokumentation.
- Perspektive: Anbieter vs Nutzer. Einfache Nutzung.
- KISS-Prinzip: Schnittstelle möglichst einfach halten.
- Sicherheit: Die Nutzung ist sicher zu gestalten.

8.5 API Patterns

API Patterns werden verwendet, um in APIs die Namensgebung und das Verhalten möglichst vorhersehbar und durchgängig zu gestalten.

Beispiele:

- Repetition: Verwende immer den gleichen Namen für die gleichen Dinge.
- Periodisch: Verwende immer die gleiche Bezeichnung für gleiches Verhalten.
- Symmetrie: Biete wenn möglich symmetrische Methoden an, z.B. für das Öffnen und Schliessen, oder das Belegen und Freigeben.

8.6 Mehr als nur Nutzer und Implementation

Häufig gibt es mehrere Implementationen für ein und dieselbe API. Beispiel: JDBC wird von verschiedenen DB-Herstellern implementiert, es gibt unterschiedliche Treiber für diverse DBs.

8.7 SPI - Die API für den Anbieter

SPI (Service Provider Interface) ist eine spezielle Teilmenge oder Ergänzung zu einer API. Sie ist für die Anbieter von Implementationen einer API gedacht. Erlaubt z.B. sich in einer Factory der API zu registrieren. SPI muss nicht zwingend Class-based sein, sondern kann auch eine Datei-Schnittstelle sein (Property-Datei, JSON, YAML, etc.).

Beispiel: Stell dir vor, du hast ein Programm, dass verschiedene Zahlungsanbieter wie PayPal oder Stripe nutzen kann.

1. SPI. Es gibt eine Schnittstelle "PaymentService" mit der Methode "processPayment(amount)".
2. Implementierungen:
 - PayPalService implementiert "PaymentService".
 - StripeService implementiert "PaymentService".
3. Das Programm ruft nur die Methode der SPI auf ("processPayment"), ohne zu wissen, ob es sich um PayPal oder Stripe handelt.

Die SPI erlaubt also Flexibilität und Austauschbarkeit der Implementierungen.

8.8 Class-based API

Class-based APIs werden über 1..n Interfaces und Klassen realisiert.

Klassen die als formale Parameter oder Return genutzt werden, sind wie auch mitgelieferte Hilfsklassen (z.B. Factories) automatisch Bestandteil der Schnittstelle.

In geschichteten Architekturen entsteht häufig ein API-Layer.

Beispiel: Klasse ist StudentService. Das Interface von StudentService mit dessen Methoden (getById(), findByLastName(), getAll(), etc.) ist eine Class-based API.

8.9 REST - REpresentational State Transfer

REST ist ein ARchitekturstil und definiert einen Satz von Prinzipien welche eine konkrete, REST-konforme Architektur einhalten soll.

Die fünf REST-Prinzipien sind:

1. Statuslose Kommunikation (Zustandslosigkeit)
2. Verwendung von (http[s]-)Standardmethoden.
3. Ressourcen mit eindeutiger Identifikation (URI).
4. Unterschiedliche Repräsentationen von Ressourcen.
5. Verknüpfen / Hypermedia (HATEOAS).

Eine REST-konforme Architektur basiert zwangsläufig auf einer verteilten Client/Server Architektur, meist mit http[s]-Protokollen.

8.9.1 Zustandslosigkeit

Die Kommunikation muss bei REST zustandslos sein. Ein client-spezifischer, auf dem Server gehaltener Zustand über die Grenze eines Requests hinaus ist nicht erlaubt (vgl. Session).

Ein Zustand muss entweder vom Client gehalten, oder (viel besser) vom Server in einer veränderten Ressource abgebildet werden.

Der Status der Applikation ergibt sich ausschliesslich aus der Ressourcenrepräsentation (URI) und dem Ressourcenstatus. Es dürfen keine Sitzungsinformationen existieren.

Vorteile:

- Setzen von Bookmarks ist problemlos möglich.
- Bessere horizontale Skalierbarkeit der Infrastruktur.

8.9.2 REST - Standardmethoden

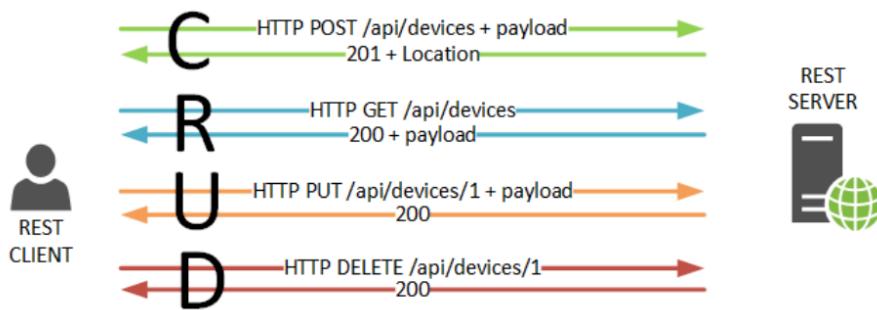
Eine REST-konforme Applikation verwendet die http-Methoden GET, PUT, PATCH, POST, DELETE, etc. in verbindlicher Weise:

- GET: Fordert die angegebene Ressource vom Server an.
- PUT: Falls die angegebene Ressource existiert, wird sie aktualisiert, sonst neu angelegt.
- DELETE: Löscht die angegebene Ressource.
- POST: Fügt (erstellt) eine neue Ressource hinzu.

Die Methoden GET, PUT, PATCH und DELETE sollen beliebig oft aufgerufen werden können, ohne Seiteneffekte hervorzurufen (Implementation als **idempotente** Methoden).

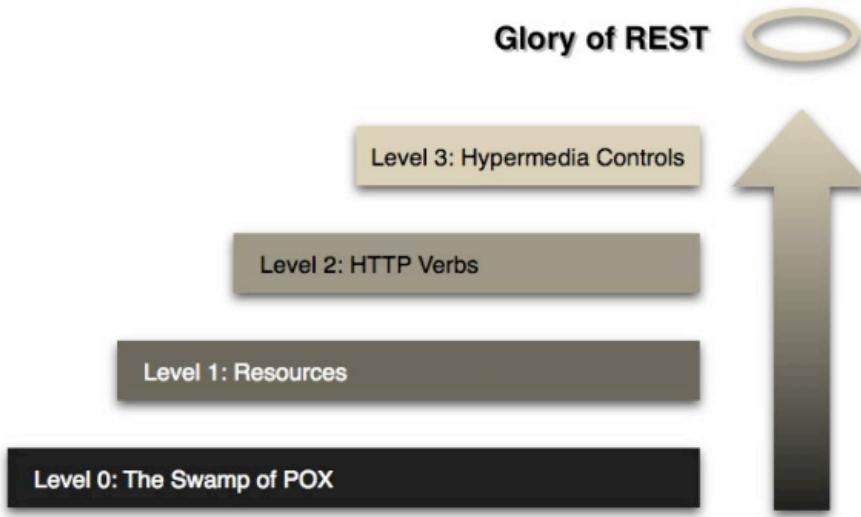
Die Methode POST darf nicht mehrfach aufgerufen werden, da sie Seiteneffekte hat (somit eine **nicht idempotente** Methode).

8.9.3 Endpoint Naming, Statuscodes, Request-parameters



8.9.4 Beurteilen von REST Schnittstellen

Zur Beurteilung kann das Richardson Maturity Model (RMM) herangezogen werden:



8.9.5 Versionierung

HTTP Schnittstellen kann man Versionieren. Einfaches Beispiel:

- <http://fbs.hslu.ch/api/v1/orders>
- <http://fbs.hslu.ch/api/v2/orders>

Wenn möglich ersetzt man die alten Versionen jeweils durch eine Wrapper-Implementation (Translator) auf die neueste Version. So vermeidet man Coderedundanzen und erhöht den Wartungsaufwand nicht. Trotzdem hat man einen zentralen Zugriffspunkt.

9 Testen von Applikationen

9.1 Testarten

Unit- und Komponententest (Entwicklertest)

Schnelle, einfache Tests von einzelnen Klassen und/oder Komponenten, während der Entwicklung, ohne Abhängigkeiten, automatisiert, überall lauffähig und beliebig wiederholbar.

Integrations- und (Teil-)Systemtest

- Testen eines gesamten (Teil-)Systems innerhalb seiner (Teil-)Systemumgebung, auf Basis der Spezifikation, auf seine fachliche Korrektheit.
- Aufgrund der Komplexität und des Aufwandes werden solche Tests noch viel zu häufig manuell durchgeführt → speziell z.B. bei GUI-Tests.

9.2 Ziele für gute Tests

- Testausführung so weit wie möglich automatisieren!
- Erstellung von Testscripts/-code lohnt sich.
- Fröhre Erstellung der Teststories auf Basis der Use Cases.
- Möglichst kurze und einfache Testfälle.
- Use Cases / Funktionen möglichst einzeln testen.
- Testen der normalen Abläufe als Basis (Beispiele:)
 - Korrektes (erlaubtes) Login
 - Suchen (und finden) von vorhandenen Daten.
 - Erfassen (und korrektes speichern) von Daten.
- Unbedingt auch absichtliche provozierte Fehler, Ausnahmen und nicht erlaubte Vorgänge testen, Beispiele:
 - Falsches oder nicht zulässiges Login
 - Ungenügende Rechte eines Benutzers, um bestimmte Daten zu sichten, zu verändern oder zu löschen.
 - Korrektur Abbruch einer Aktion (ohne Veränderung der Daten).

Testfälle gerade so komplex wie nötig, und so einfach wie möglich halten!

9.3 Tests in Schichtenarchitekturen

Die Schichten sollten möglichst einzeln und entkoppelt voneinander getestet werden. Einsatz von Test Doubles um dies zu vermeiden. So wird der untere Layer so "simuliert", dass er eine Antwort retourniert, die die echte Implementation in diesem Szenario zurückgeben sollte/könnte.

9.4 Testen von DB-Applikationen

In den meisten Fällen kann der Persistence-Layer auch einfach mit einem Double ersetzt werden. Wenn man aber den Datenbankzugriff testen möchte, empfehlen sich Integrationstests mit einer Testdatenbank. Die Production-Data darf durch tests nicht verändert werden.

Mögliche Lösungen:

- Initialisierung von Testdaten
- Verwendung mehrerer Schemas
- Lokale Datenbank zum testen
- In-Memory-Datenbank
- Produktive Datenbank als Docker Container (Identische Datenbank, immer sauber Initialisiert).

9.5 Testen von REST-(Micro-)Services

Da REST-Services (Implementations-)sprachunabhängig sind, gibt es eine grosse Zahl von Testframeworks.

Für simple Fälle reichen JUnit und Java-Bordmittel schon aus. Die meisten Microservice-Frameworks (SpringBoot, Micronaut.io) bieten jeweils eigene, direkt integrierte Ergänzungen für (Unit-)Testing an.

Swagger oder Postman können da auch helfen.

10 Summary of Braindumps

10.1 Person 1

- Requirements Engineering: Was ist das, für was? Was für Diagramme sind hilfreich? Was ist das Ziel? Was sind Stakeholder?
- BPMN / Kontextdiagramm (weil von Student erwähnt): Was ist das?
- Domain Model: Was ist das? Was zeigt es?
- Micro Services: Wieso würde man Micro Services nutzen? Vorteile, etc.?

Bemerkung von P1: *Es sind Thoeriefragen gekommen, bis auf Requirement Engineering war alles Projektspezifisch. Code wurde nicht abgefragt, nur auf hohem level wie die Services miteinander interagieren.*

10.2 Person 2

- Requirement Engineering: Was ist das? Für was ist es gut? Was sind Stakeholder? Was ist Event-Storming? Für was kann man es einsetzen? Was für Diagramme oder Techniken hatten wir im Modul genutzt?
- Domain Model: Was ist es? Was sind bounded context? Wie bestimmt man bounded context (am Beispiel vom domain model)? Wie erkennt man ob bounded context schön definiert sind?
- Micro Services: Was ist ein Event (in unserem Projekt)? Wie funktioniert die Kommunikation zwischen den Services? Wie kann man synchrone Validierungen besser (resp. "schöner") lösen (konkret Projekt)?

Bemerkung von P2: *Auch keine Codefragen, nur Thoerie und Konzepte im Projekt*

10.3 Person 3

- Requirement Engineering: Was sind Stakeholder? Wer sind Stakeholder im SWDA Projekt?
- Was ist INVEST (nicht Buchstabe für Buchstabe sondern für was es ungefähr steht)? Eine Anforderung aus dem Projekt nach INVEST beurteilen.
- Wie haben wir Requirements modelliert? Was ist BPMN?
- Domain model: Was ist es und einige projektspezifische Fragen dazu.
- Projektspezifische Fragen zum Microservice Diagramm.
- Was sind DTOs? Wozu verwendet man diese?

Bemerkung von P3: *Von jemand anderem gehört: Unterschied Modul & Microservice (nennen sie konkrete Unterschiede)*

Bemerkung 2 von P3: *Die Dozenten entschieden wann man in die Doku schauen darf, man darf dies nicht immer. Also wenn man etwas in der Doku stehen hat kann es sein das man für diese Frage nicht nachschauen darf.*

10.4 Person 4

- Requirement Engineering: Was sind die Ziele und der Nutzen? Was sind Stakeholder? Funktionale vs. nicht funktionale Anforderungen?
- Model: Welche Modelle haben wir im Projekt gehabt? Was sind die wichtigsten gewesen und diese erklären. Für was ist das kontextmodel? Was ist das Domain model? ...
- Wie hätte man die Schnittstellen definieren sollen.
- Eigenschaften von Microservices und einen genauer erklären.

Bemerkung von P4: *Kein Code*

10.5 Person 5

- Unterschied funktionale vs nicht-funktionale Anforderungen, wie kann man Anforderungen bewerten?
- Was für Diagramme hatten wir gemacht im Projekt, welches ist wieso/wozu hilfreich?
- Was ist REST, welche Methoden sind idempotent?
- Rest war projektspezifisch auf Domain Model und Microservices
- Eine Klasse im Code erklären und Fragen dazu (da von Student die Klasse erwähnt wurde).

10.6 Person 6

- Event Storming, Was ist es und für was?
- Anforderungen? Was für Arten? Funktional vs. nicht-funktional? Beispiele? Unterschiede?
- Wo im Projekt sind funktionale Anforderungen?
- Domain Model: Was ist es? Für wen? Für was? Was würde man anderst machen (projekt spezifisch)?
- Code Beispiel zu Entitäten. Wo erkennt man die Kardinalität Warum ist diese so?
- Status im Diagramm, warum so gezeichnet?
- Bounded Contexts? Für was? Wo im Diagramm ersichtlich?
- Micro Service Diagramm, asynchron vs synchron. Wo verwendet und warum? Wie könnte man das besser machen (→weniger asynchrone Calls)? Wie kann man die Reihenfolge von asynchronen Calls überwachen und sicherstellen? → Correlation Id über alle Folge-Requests mitsenden (eine random generierte v4 UUID).
- REST, was ist es? Auszeichnungen? Wie im Projekt umgesetzt?
- testing, was war das Konzept im Projekt, wie wärs gut?

10.7 Person 7

- Was ist Requirements Engineering
- Was ist eine User Story
- Was sind Akzeptanzkriterien
- Was für Modelle/Techniken gibt es um Requirements zu finden
- BPMN erklären
- Sehr viel Fragen zum erstellten Domain Model

10.8 Person 8

Bemerkung von P8: *Sehr ähnlich wie bei P7 - alle Modelle durchgegangen und gefragt was man besser machen könnte.*

10.9 Person 9

Bemerkung von P9: *Domain-Elemente wurden wie DB-Entities angeschaut und verglichen. Bei P9 im Domain Model, bei P6 im Code. Dies von den zwei Entitäten: "Order" und "Article" bei beiden Personen.*

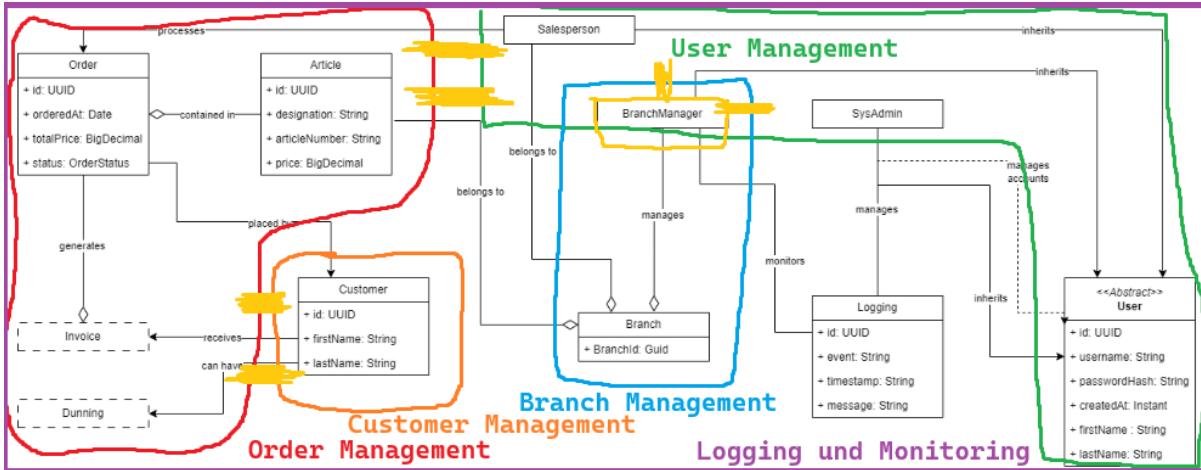
10.10 Person 10 (aus 1 Semester vorher)

- Requirements Engineering (laut P10 war dies bei ALLEN Teams Thema): Was ist es? Aufgaben? Ziele? Wie kann man sie bewerten? (INVEST & SMART)?
- User Stories: Was sind User Stories? Bestandteil von User Stories
- REST: Was zeichnet eine REST api aus? Kann man REST nur im Zusammenhang mit HTTP brauchen? (→ nein ist Protokoll-unabhängig! Ist REST stateless? Wie kann man trotzdem states brauchen? Client und Serverseitige Sessions beschreiben, wie würde man im SWDA Projekt einen Warenkorb implementieren.
- HTTP: Was sind HTTP Methoden? Welche HTTP Methoden gibt es? Kann man auch eine custom HTTP Methode definieren und implementieren?
- Synchrone vs asynchrone Kommunikation: Kann man synchrone Endpoints auch asynchron aufrufen? Bei einem synchronen Aufruf kommt man direkt eine Antwort über, welche Möglichkeiten gibt es bei asynchroner Kommunikation eine Antwort zu erhalten?
- Das BPMN aus dem Projekt erklären und zeigen, wann/wo eine Bestellung stornierbar ist.
- Bounded context: Was sind die Bounded Context im Projekt? Wo in der Dokumentation sind Bounded Context ersichtlich?

Bemerkung von P10: *Keine Fragen zum Code*

11 Vorbereitung eigene MEP

11.1 Bounded Context im Domain Modell



Bounded Context:

- Order Management
- Customer Management
- User Management
- Branch Management
- Logging und Monitoring (nur "Logging")

Beziehungen zwischen den Bounded Contexts:

- Order Management und Customer Management: Über Order und Customer
- User management und Order Management: Über Salesperson
- Logging und Monitoring mit allen anderen Kontexten
- Branch Management und User Management: BranchManager

11.2 Requirements Engineering

Was ist das?

Requirements Engineering ist das systematische Vorgehen beim Spezifizieren und Verwalten von Anforderungen an ein System, ein Produkt oder ein Software.

Für was brauchen wir es?

Hilft uns folgende Fragen zu beantworten:

- Was sind die Anforderungen* des gewünschten Produktes?
- Wie lassen sich die Anforderungen spezifizieren (also erheben, analysieren, dokumentieren und validieren)?
- Wie sollten Anforderungen verwaltet - also freigegeben, evtl. verändert oder rückverfolgt - werden?

* die Fähigkeiten und Eigenschaften

Welche Diagramme sind hilfreich?

Kontext-Diagramm, Domain-Modell, Use Case Modell (mit Beschreibungen), Geschäftsprozess-Modell (BPMN), Feature-Listen, User Stories

Was ist das Ziel?

- Risikominimierung
- Gemeinsames Verständnis der Stakeholder

Was sind Stakeholder?

Personen oder Gruppen die ein berechtiges Interesse am Verlauf oder Ergebnis haben. z.B. Management, Tester, Gesetzgeber, Sicherheitsbeauftragte, Kunden, Entwickler

Wer sind die Stakeholder im SWDA Projekt?

PO (Name von Dozent der PO war wird nicht erwähnt), Entwickler, (die anderen Dozenten im Modul egal in welcher Rolle, da sie ein berechtigtes Interesse daran hatten).

Was ist Event Storming

Eine Methode, die dazu dient, komplexe Geschäftsprozesse und Domänen schnell zu verstehen, in dem man sich auf die Ereignisse konzentriert. Daraus schliesst man dann auf die Commands, die Akteure, und die Aggregates. Das gibt einem ein gutes Verständnis der Domäne.

Für was kann man Event Storming einsetzen?

Um ein gutes allgemeines Verständnis der Domäne bei Stakeholdern zu verbessern. Auch um Problemstellen und Wissenslücken schnell zu identifizieren. Es fördert generell die Zusammenarbeit zwischen technischen und nicht-technischen Stakeholdern.

Was für Diagramme / Techniken haben wir im Modul genutzt?

Microservice-Diagramm, Domain-Modell, GEschäftsprozess-Modell (BPMN).

Funktionale vs nicht-funktionale Anforderungen?

Funktionale Anforderungen legen Fest, **WAS** das Produkt zu tun hat.

Nicht-Funktionale Anforderungen legen Fest, **WIE** es das zu tun hat.

Beispiele für nicht-funktionale: Performanz, Zuverlässigkeit, Verfügbarkeit, Sicherheit, Wartbarkeit, Portierbarkeit.

Wo im Projekt sind funktionale Anforderungen?

Wo "definiert"? → im Gitlab in den Issues.

Wo "umgesetzt"? → überall im Code wo eine Funktionalität implementiert wird.

Was für Arten von Anforderungen gibt es?

Nutzungsanforderungen (effiziente Erbringung eines Ergebnisses), **Gesetzliche Anforderungen** (von Behörden, z.B. in Form von Richtlinien, Gesetzen, etc.), **Fachliche Anforderungen** (Vollständigkeit und Korrektheit des Ergebnisses), **Organisatorische Anforderungen** (Verhalten von Personen oder Teams bei der Erbringung des Ergebnisses), **Marktanforderungen** (für die Kaufentscheidung relevant).

Was ist eine User Story?

Eine User Story ist eine textliche Anforderung an das System, geschrieben aus Anwendersicht.

Als "Stakeholder" möchte ich "Wunsch", damit "Nutzen"

Was sind Akzeptanzkriterien?

Akzeptanzkriterien definieren, was getan werden muss, um eine User Story abzuschliessen. Sie legen die Grenzen der Story fest. Sie müssen ein klares Pass / Fail Ergebnis haben. Jede User Story sollte mindestens ein Akzeptanzkriterium haben.

Wie kann man Anforderungen bewerten? INVEST & SMART?

INVEST: Independent, Negotiable, Valuable, Estimable, Small, Testable - Kriterien für gute User Stories.

SMART: Specific, Measurable, Achievable, Relevant, Time-bound - Kriterien für klare und erreichbare Ziele.

11.3 Diagramme und Modelle

Was ist BPMN?

BPMN: Business Process Management Notation. Eine Notation für Geschäftsprozess-Modelle. Ein Geschäftsprozessmodell spiegelt im Wesentlichen die zeitlich-sachlogische Abfolge der betrachtenden Funktion wider. Dienen der Dokumentation, Analyse und Gestaltung von Geschäftsprozessen.

Was ist ein Kontextdiagramm?

Im Kontextdiagramm sehen wir die System und seine Grenzen, alle Akteure, und alle Nachrichten (Datenflüsse) zwischen den Akteuren und dem System.

Was ist ein Domain Model?

Ein Domain-Modell ist ein konzeptionelles Modell der Anwendungsdomäne und beinhaltet sowohl Verhalten als auch Daten. Typischerweise als Klassendiagramm erstellt. Kann mit Kontexten ergänzt werden.

Was sind bounded context?

Ein klar definierter Bereich innerhalb einer Domäne, in dem eine spezifische Sprache und Logik gilt. Kerkonzept von DDD (Domain-Driven-Design).

Für was sind bounded context?

Zur Abgrenzung und Strukturierung von Domänen. Sie helfen, Komplexität zu reduzieren, Verantwortlichkeiten klar zu trennen und Konflikte zwischen Teams zu vermeiden.

Wie bestimmt man bounded context?

Am Beispiel vom Domain Model? → Siehe oben 11.1. Generell aber sind die bounded context OFT gleich Microservices. Da jeder Microservice in einen eigenen Context steht, stimmt dies OFT überein mit den bounded contexts.

Projektspezifische Fragen Domain Model & Microservice Diagramm

Hier muss man einfach sicherstellen, dass man die eigenen Diagramme versteht und vlt. mithilfe ChatGPT sich fragen stellen lässt dazu. Auch kritische, offene Fragen wie: Was könnte man besser machen?, was fehlt?, etc.

Welche Modelle hatten wir in SWDA?

Kontext-Diagramme, Geschäftsprozess-Modell, Domain-Modell, Feature-Listen, C4-Modell (Context, Containers, Components, Code), Microservice Architektur Diagramm.

Was sind die wichtigsten Modelle, diese Erklären?

Kontext-Diagramm: Im Kontextdiagramm sehen wir die System und seine Grenzen, alle Akteure, und alle Nachrichten (Datenflüsse) zwischen den Akteuren und dem System.

Geschäftsprozess-Diagramm: Ein Geschäftsprozessmodell spiegelt im Wesentlichen die zeitlich-sachlogische Abfolge der betrachtenden Funktion wider. Dienen der Dokumentation, Analyse und Gestaltung von Geschäftsprozessen.

Domain-Modell: Ein Domain-Modell ist ein konzeptionelles Modell der Anwendungsdomäne und beinhaltet sowohl Verhalten als auch Daten. Typischerweise als Klassendiagramm erstellt. Kann mit Kontexten ergänzt werden.

Domain Model - für wen und für was?

Für was wird oben beantwortet, für wen: Für Entwickler, Architekten, Fachexperten (allg: technische Stakeholder) um besseres Verständnis der Domäne zu schaffen.

Microservice Diagramm, synchron vs. asynchron. Wo verwendet und warum?

Synchron, wo man eine Antwort will, die man weiter verarbeiten möchte oder sofort dem Benutzer anzeigen möchte.

Asynchron, wo man das "fire-and-forget" einhalten / umsetzen kann (oder will).

Microservice Diagramm, synchron vs. asynchron. Was könnte man besser machen?

Siehe "Wie kann man synchrone Validierung besser lösen (konkret im Projekt)?"

11.4 Technische & Allgemeine Fragen

Was versteht man unter Domain?

Die Domäne beschreibt den fachlichen Kontext oder das Problemfeld, das durch die Softwarelösung abgedeckt wird. Sie umfasst alle Begriffe, Prozesse, Regeln und Zusammenhänge, die für das Geschäft oder die Organisation relevant sind.

Wieso würde man Micro Services nutzen? Vorteile?

Die Services laufen in echter Parallelität als verteilte Applikation. Somit kann ganzes Potential einer verteilten Anwendung genutzt werden. Skalierung wird einfacher, nur die Services die eine hohe Auslastung haben müssen skaliert werden. Unabhängige Entwicklung an einzelnen Services, Technologievielfalt (jeder Service kann mit einer anderen Technologie entwickelt werden). Bei Ausfällen (wenn richtig gemacht) fällt nur ein Teil-system aus.

Was ist ein Event (und wo im eigenen Projekt?)

Event im Sinne von RabbitMQ-Kommunikation: Überall dort, wo Nachrichten auf den Message-Bus gelegt werden, wie in der Klassen CommandSender.

Event im Sinne von asynchroner Kommunikation (Event vs Command): In allen Bereichen, in denen ein Service eine Nachricht sendet und keine Reaktion erwartet. z.B. Order Ready for Shipment im Order Service

Wie funktioniert die Kommunikation zwischen den Services?

Kommunikation erfolgt über RabbitMQ, einem Message-Broker, der die Kommunikations mittels Commands (synchron) und Events (asynchron) ermöglicht.

Commands: Werden an eine Queue geschickt (basierend auf Routing-Key). Der empfangende Service und kann über eine reply-to eine Antwort zurücksenden. Das simulierende RPC-Pattern ermöglicht Synchronität

Events: Ein Service publiziert das Event an RabbitMQ. Andere Service, die sich für dieses Event interessieren, abonnieren die entsprechende Queue. Das Prinzip ist fire-and-forget, und der Publisher erwartet keine Antwort.

Wie kann man synchrone Validierung besser lösen (konkret im Projekt)?

Auch asynchron machen (bei uns im Projekt). z.B.

1. Order Create Event (sync an order service, status = created)
2. Customer Service bekommt created event und validiert customer, sendet CustomerVerified event für die Order (status = customer verified)
3. Article Service prüft ob die Artikel auch existieren und sendet ArticleVerified event an den Order Service.

Falls etwas failed: status auf cancelled.

Was sind DTOs, Wozu verwendet man diese?

Ein DTO (Data Transfer Object) ist ein Datenobject, das verwendet wird, um Daten sicher und effizient zwischen zwei Modulen, Schichten oder Systemen zu übertragen. Es dient dazu, die Datenstruktur zu vereinfachen und die Entkopplung zwischen den Schichten oder Services zu fördern. - "ein Distribution Pattern".

Was ist der Unterschied zwischen einem Modul und Microservice? Konkrete Unterschiede

Ein Modul kann von einer einzelnen Klasse bis hin zu einem kompletten (kleinen) Microservice alles sein. Ein Modul sollte als Ganzes sinnvoll nutzbar sein, nicht nur Teile oder Bruchstücke davon, sonst muss es weiter aufgeteilt werden. Ein Modul soll möglichst eine in sich geschlossene Aufgabe erfüllen und einen starken inneren Zusammenhalt aufweisen.

Wie sollte man Schnittstellen definieren?

Eindeutig, Lose gekoppelt (minimale Abhängigkeiten), Einfach, Robust, Versionierbar, Standardkonform (z.B. REST), Sicher.

YAGNI (You aint gonna need it) und KISS (Keep it Simple / Stupid).

Eigenschaften von Microservices - einen MS im Projekt genauer erklären

Sind unabhängig, eigenständig, kommunizieren über RabbitMQ (z.B. InventoryService).

Wo erkennt man im Code die Kardinalität der Entitäten? Warum ist dies so im Code wie es ist?

Man erkennt dies sicher in den Models für die Datenbank. Wenn man in zwei Models hineinschaut sieht man dort, ob nur ein Objekt des anderen Typen referenziert wird, oder eine Liste davon. Oder auch abgebildet durch JPA-Annotations (@OneToMany, @ManyToOne)

Wie kann man die Reihenfolge von asynchronen Calls überwachen und sicherstellen?

Die Correlation ID verknüpft alle Nachrichten eines bestimmten Workflows oder Aufrufs. Die Sequencenumber innerhalb derselben Correlation ID gibt die genaue Reihenfolge vor.

Was ist REST? Wie/Wo im Projekt umgesetzt?

REpresentational State Transfer. Es ist ein Architekturstil den wir im API Gateway eingesetzt haben. Statuslose Kommunikation, Verwendung von http[s]-Standardmethoden, Ressourcen mit eindeutiger Identifikation (URI), Unterschiedliche Repräsentationen von Ressourcen, Verknüpfen / Hypermedia (HATEOAS).

Kann man REST nur im Zusammenhang mit HTTP brauchen?

Nein REST ist ein Architekturstil und Protokoll-unabhängig. HTTP ist aber das am häufigsten genutzte Protokoll für REST-APIs. z.B Hätte man es in VSK einsetzen können (Protokoll: gRPC).

Wie kann man trotzdem states brauchen? [REST]

Man kann States nutzen, indem der Zustand ausserhalb der REST-Services verwaltet wird.

Client-seitig: Client speichert Zustand zwischen den Requests und sendet diesen bei jedem neuen Request mit (Beispiel: Session-IDs oder Token im Header, z.B. Authorize Header)

Server-Seitig: Der Zustand wird auf dem Server in einer Datenbank oder einem Cache gespeichert, während der Client eine Referenz (z.B. Session-ID, Token) erhält. Beispiel: Server nutzt Session-Id vom Client um Zustand auf Datenbank abzurufen.

Client vs. Serverseitige Sessions beschreiben. Unterschiede?

Client: z.B. Cookies, Local Storage [Vorteil: Skalierbarer, Nachteil: Sicherheitsrisiko]

Server: z.B. Datenbank, Cache [Vorteil: Sicherer, Nachteile: Belastung Server]

Unterschied: clientseitig liegt Zustand bei Client, Server ist stateless. Serverseitig liegt Zustand auf Server, erfordert aber Session-Management.

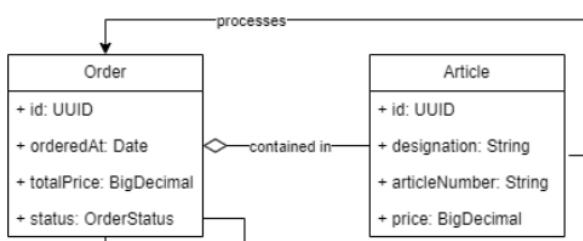
Wie würde man im Projekt einen Warenkorb implementieren?

Der Warenkorb wäre ein eigener Microservice, der Warenkorbdaten (Artikel, Mengen, Benutzer) verwaltet. Die Daten kann man dann auch wieder in eine DB speichern. Sonst wäre der Aufbau ziemlich gleich wie er auch in den anderen Microservices ist. Herausforderungen wären die Synchronisation mit dem ArticleService (Verfügbarkeit und Preise), Konsistenz bei parallelen Zugriffen und regelmässige bereinigen inaktiver Warenkörbe. Beim Checkout muss man mit dem OrderService zusammenarbeiten, wie bei einer normalen Order.

Was war das Testing Konzept im Projekt, wie wärs gut?

Das Testing Konzept war das testen per cURL requests, automatisiert in Shell-Scripts. Es gab ein Repo, auf welchem man die Pipeline manuell anstoßen konnte, was die Tests ausführte. Problem dabei: Wird auf prod getestet, also sind verändernde Requests nicht testbar. (eigentlich nur GET möglich).

DB Entities im Domain Modell angeschaut. Spezifisch Order und Article



Was ist hier nicht gut? → Es fehlen im Modell einige Felder, die im Code vorhanden sind (wie z.B: SellerId, CustomerId, bei Order). Generell fehlen auch einige Entitäten.

Was sind HTTP Methoden?

HTTP Methoden definieren die Aktionen, die auf Ressourcen in einer REST-API ausgeführt werden können. z.B:

Welche HTTP Methoden gibt es?

GET, POST, PUT (Create or Update), PATCH (Partial Update), DELETE, HEAD (get only headers), OPTIONS (zeigt verfügbare Methoden für die Ressource).

Kann man auch eine custom HTTP Methode definieren und implementieren?

Ja, meistens sollte man aber die typischen verwenden.

Eine custom Methode könnte in einer spezialisierten Anwendung wie einem IoT System genutzt werden z.B: SYNC (Synchronisiert den Zustand eines IoT-Geräts mit dem Server). POST oder PATCH würden nicht klar ausdrücken, dass es sich um eine Synchronisierung handelt.

Trotzdem oft besser vorhandene Methoden zu verwenden wie z.B: POST mit spezifischen Endpunkten (/device-sync) und erklärenden Headern. Hier müsste man dann auf das REST naming verzichten.

Welche Methoden sind idempotent, welche nicht? Was bedeutet dies?

Idempotent: GET, PUT, DELETE, PATCH (laut FOlien, aber kommt eig. auf Implementation an: "PATCH - increase count of xyz" wäre nicht idempotent)!

Nicht idempotent: POST

Idempotenz bedeutet, dass eine Methode, wenn sie mehrmals ausgeführt wird, immer das gleiche Ergebnis liefert, ohne den Zustand erneut zu verändern.

Kann man synchrone Endpoints auch asynchron aufrufen?

Ja, synchrone Endpoints können asynchron aufgerufen werden, Unterschied liegt in der Verarbeitung auf der Client-Seite. Endpoint bleibt synchron und verarbeitet den Request wie gewohnt. Der Client kann den Aufruf asynchron ausführen, indem er den Request in einem separaten Thread oder über ein Event-Loop ausführt.

Bei einem synchronen Aufruf kommt man direkt eine Antwort über, welche Möglichkeiten gibt es bei asynchroner Kommunikation eine Antwort zu erhalten?

Callback: Sender registriert eine Funktion oder Methode, die aufgerufen wird, sobald die Antwort eintrifft.

Polling: Der Sender fragt periodisch den Status des Ergebnis ab, bis Antwort verfügbar ist.

Message Queue: Eine Antwort wird über eine separate Queue (z.B. RabbitMQ) an den Sender zurückgesendet. Oft wird hierfür die Correlation ID verwendet, um die Antwort der ursprünglichen Anfrage zuzuordnen.

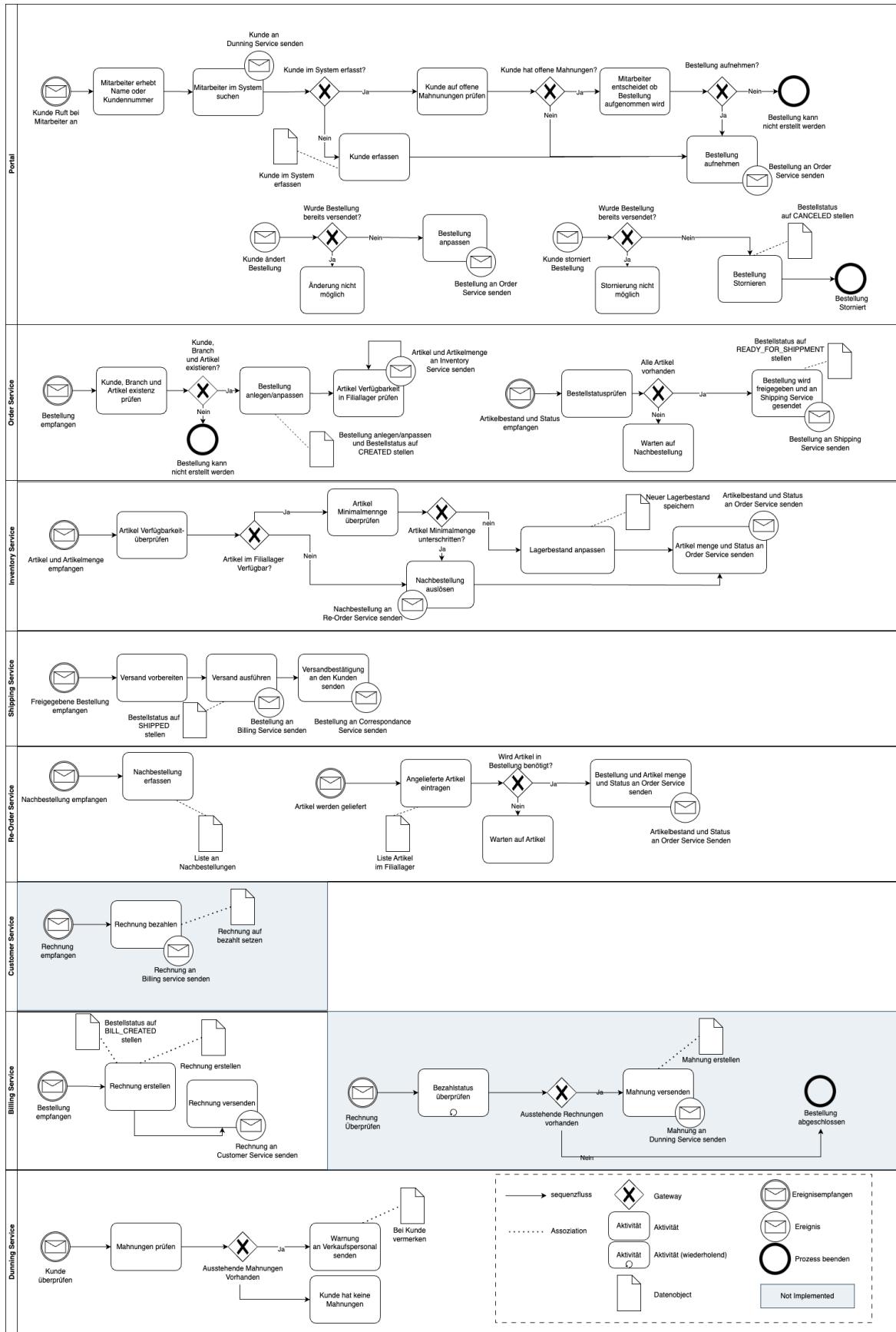
Event-basierte Antwort: Der Empfänger sendet ein Event, das die Antwort enthält, und der Sender reagiert darauf.

11.5 Unsere Diagramme aus dem Projekt - Analyse

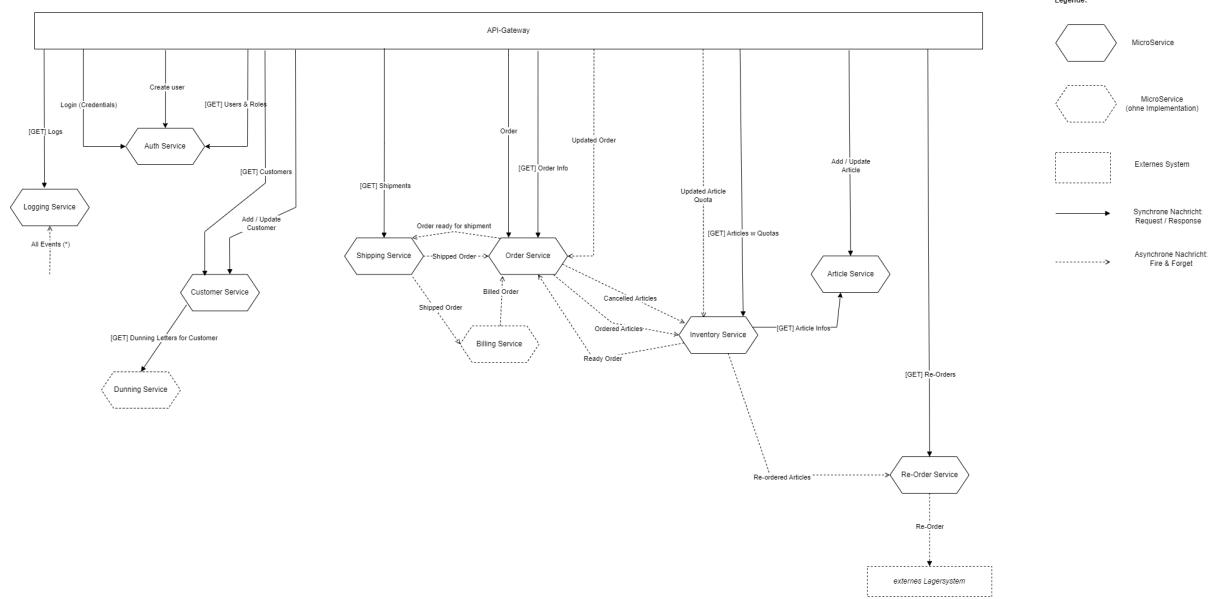
Bis wo ist eine Bestellung stornierbar? Im BPMN zeigen!

Eine Bestellung kann immer storniert werden, ausser sie schon versendet. Dies sieht man unten rechts in der ersten Lane.

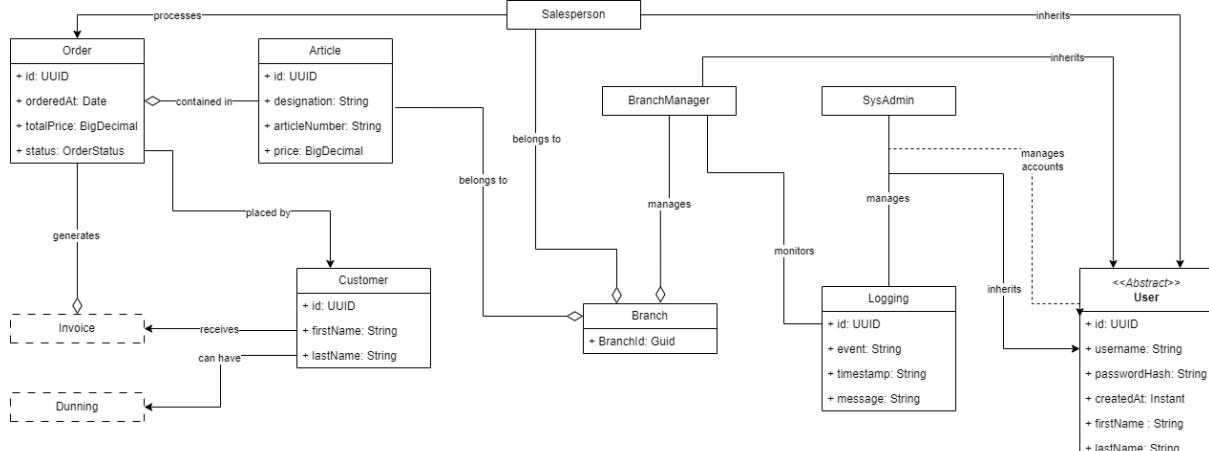
11.5.1 BPMN



11.5.2 Microservice-Architektur-Diagramm



11.5.3 Domain-Modell



Domain-Modell ist fehlerhaft, bzw. es fehlt sicher einiges.