

# Projektarbeit

Thema:

## **Gemeinsamkeiten und Unterschiede von Service-orientierter Architektur (SOA) und Microservices**

**Stefan Kruk**

geboren am 14.08.1992

Matr.-Nr.: 7084972

An der Fachhochschule Dortmund im Fachbereich Informatik erstellte  
Projektarbeit  
im Studiengang Softwaretechnik (Dual) - Modul Entwicklung verteilter  
Anwendungen

**Betreuer:** Prof. Dr. Johannes Ecke-Schüth

**Fachbereich Informatik**

Dortmund, 10. September 2016

# TODOS

1. Dokument auf das Wort SOLL überprüfen und ggf. durch wird/werden austauschen
2. Dokument auf das zu oft vorkommen vom Wort "man"überprüfen
3. Reicht die Zitierform in 1.2 Ausgangssituation ?
4. richtige Überschriften Wahl (vorallem in 2 Problemanalyse)
5. Anführungsstiche durch glqq und grqq ersetzen.
6. Bilder richtig anordnen.

**Eidesstattliche Erklärung**

Ich versichere an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, den 10. September 2016

Stefan Kruk

**Erklärung**

Mir ist bekannt, dass nach § 156 StGB bzw. § 163 StGB eine falsche Versicherung an Eides Statt bzw. eine fahrlässige falsche Versicherung an Eides Statt mit Freiheitsstrafe bis zu drei Jahren bzw. bis zu einem Jahr oder mit Geldstrafe bestraft werden kann.

Dortmund, den 10. September 2016

Stefan Kruk

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>iii</b>
<b>Überblick</b>	<b>iv</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Ausgangssituation . . . . .	2
1.3 Vorgehen und Kapitel . . . . .	3
<b>2 Problemanalyse</b>	<b>4</b>
2.1 Herausforderungen . . . . .	4
2.2 Beispiel . . . . .	5
2.3 Die zu untersuchende Fragestellung . . . . .	6
<b>3 Allgemeine Grundlagen</b>	<b>8</b>
3.1 Architektur . . . . .	8
3.1.1 Das Gesetz von Conway . . . . .	9
3.1.2 Domain-Driven Design und Bounded Context . . . . .	10
<b>4 Service-orientierte Architektur (SOA)</b>	<b>12</b>
4.1 Grundlagen . . . . .	13
4.1.1 Unternehmens Komponenten . . . . .	13
4.1.2 Wertschöpfungskette . . . . .	14
4.2 Architektur . . . . .	14
4.2.1 Orchestration . . . . .	15
4.2.2 Choreographie . . . . .	16
4.3 Enterprise Service Bus - ESB . . . . .	17
<b>5 Microservices</b>	<b>20</b>
5.1 Überblick . . . . .	20
5.2 Größe von Microservices . . . . .	21
5.3 Orchestration vs Choreographie . . . . .	22
5.3.1 Herausforderung . . . . .	23
5.4 PUSH- VS PULL-Architektur . . . . .	23

<b>6</b>	<b>Ergebnisse</b>	<b>26</b>
6.1	Vergleich . . . . .	26
6.1.1	Grundlagen . . . . .	26
6.1.2	Architektur . . . . .	27
6.1.3	Kommunikation . . . . .	29
6.1.4	Beteiligte Personen . . . . .	29
6.2	Fazit . . . . .	30
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>31</b>
	<b>Literaturverzeichnis</b>	<b>32</b>
<b>A</b>	<b>Diagramme und Tabelle</b>	<b>33</b>

# Abbildungsverzeichnis

3.1	Bounded Context . . . . .	10
4.1	Unternehmens Komponenten . . . . .	14
4.2	Orchestration . . . . .	16
4.3	Choreographie . . . . .	17
4.4	ESB . . . . .	18
A.1	Darstellung einer typischen Integrations-Pyramide . . . . .	33

# Überblick

## Kurzfassung

In dieser Arbeit sollen die Vor- und Nachteile von Service-orientierte Architektur und Microservices erörtert und anschließend miteinander verglichen werden. Dabei soll explizit auf die Unterschiede und Gemeinsamkeiten der beiden Architekturmodelle eingegangen werden. Für eine bessere Verständlichkeit wird zusätzlich ein bestimmter Prozesskontext mit beiden Modellen implementiert.

## Abstract

In this Projekt i will describe the Pro and Cons of service-oriented architecture and Microservices. After that i will compare both architectural models and describe the differences and similarities between these two. For a better understanding i will implement an application with both architectural models.

# Kapitel 1

## Einleitung

### 1.1 Motivation

Das Internet entwickelt sich rasend schnell und damit auch Online Unternehmen, welche dieses Medium nutzen. Dadurch entsteht ein enormer Wettbewerbsdruck unter den Unternehmen, denn Nutzer können sich innerhalb kürzester Zeit entscheiden ein Produkt zu nutzen oder nicht. Um den Anforderungen gerecht zu werden und eine möglichst große Preisspanne zu haben, nutzen Unternehmen zunehmend sogenannte Cloud-Angebote, bei dem der Kunde monatlich einen Betrag für die Nutzung des Dienstes zahlt. Hat ein Unternehmen eine zulange Time-to-Market (TTM) Zeit, können sich Nutzer schnell für andere, modernere Dienste entscheiden.

»Die Zeitspanne des Time-to-Market kann ein sehr bedeutsamer Faktor für den Erfolg des Unternehmens darstellen und ist daher nicht zu vernachlässigen. Kurze Entwicklungszeiträume bei der Time-to-Market garantieren dem Unternehmen nämlich einen Vorteil gegenüber der Konkurrenz. Hier geht es darum, dem Kunden so schnell wie möglich ein neues und innovatives Produkt anbieten zu können.

Um den Erfolg durch eine kurze Time-to-Market zu unterstützen und zu fördern, investieren Unternehmen in ihre Entwicklungsabteilungen. Diese können sich so nur auf ihre jeweiligen Bereiche konzentrieren. Durch eine leistungsfähige Entwicklungsabteilung kann so, in Kombination mit einem gut organisierten Projektplan und eindeutigen



Zuständigkeiten, die Zeitspanne Time-to-Market so klein wie möglich gehalten werden. Gerade in Branchen, in denen Innovation eine übergeordnete Rolle spielt und der Produkt-Lebens-Zyklus generell eher kurz ausfällt, spielt eine kurze Time-to-Market eine außerordentlich wichtige Rolle.«[3]

Zusätzlich entstehen Kosten für die Planung und Entwicklung des Produktes. Bis zur Veröffentlichung des Produktes hat das Unternehmen Geld und Zeit investiert. Je länger die Zeitspanne des TTM ist, umso erfolgreicher muss das Produkt sein, bzw. der Preis groß genug sein, damit möglichst zeitnah der Break-even-Point, also die Schwelle, ab der die Produktionskosten wieder eingenommen worden sind und ein Gewinn entsteht, erreicht wird.

Große Unternehmen haben dieses Problem schon früh erkannt und haben sich von Monolithischen Applikationen, bei der alle Funktionen in einer großen Applikation stecken, zu dynamischeren Architekturen weiterentwickelt. Diese Unternehmen haben angefangen Funktionen in einzelne Services zu packen und diese über Schnittstellen, wie zum Beispiel REST, anzubieten.

## 1.2 Ausgangssituation

**Bei der Ausgangssituation gehen wir von einem fiktiven Szenario aus, welche ich an das aus [8, S. 15] anlehne und im wesentlichen übernehme.**

»Die neugegründete *OnlineCommerceShop GmbH* möchte einen E-Commerce-Shop, als Hauptgeschäft betreiben. Es ist eine Web-Anwendung, die sehr viele unterschiedliche Funktionalitäten anbietet. Unter anderem zählen dazu die Benutzerregistrierung und -verwaltung, sowie die Produktsuche, Überblick über die Bestellungen und der Bestellprozess.« (vgl. [8, S. 15])

Das Unternehmen ist, auch wenn es ein reines IT-Unternehmen ist, Gewinn orientiert. A.1 zeigt den internen Aufbau eines typischen Unternehmens.

Die Geschäftsführung der GmbH hat bereits als Software-Entwickler in anderen Unternehmen Erfahrung mit dem Umgang und den Aufbau von E-Commerce-Shops

gesammelt. Zu den Erfahrungen zählen unter anderem das programmieren, testen, deployen und weiterentwickeln der Anwendung. Während dieser Prozesse sind die Programmierer zur Erkenntnis gelangt, dass bei steigender Größe der Anwendung, die Aufwände für Wartung und Weiterentwicklung stark ansteigen. Dies war auch der Grund warum das Unternehmen irgendwann Insolvenz anmelden musste. Die Kosten für Wartung und einer zeitnahen Reaktion auf die Veränderungen der Nutzerbedürfnisse konnten nicht mehr getragen werden und Anbieter wie Amazon, welche deutlich schneller, als die OnlineCommerceShop GmbH, auf diese reagieren konnten, haben sie schließlich vom Markt gedrängt.

Aus diesem Grund möchte die Geschäftsführung der neugegründeten OnlineCommerceShop GmbH eine Software-Architektur wählen, welche schneller anpassbar und einfacher zu warten ist. Für sie kommen daher das Microservice-Modell und Service-orientierte Architektur-Modell infrage.

### **1.3 Vorgehen und Kapitel**

Zunächst werden die Probleme bei der Verwendung von Monolithischen Architekturmodellen, unter dem Aspekt der Softwareentwicklung und Wartung analysiert. Darauf aufbauend soll die Grundlegende Problematik herausgearbeitet werden. Anschließend werden die Grundlagen zur Verwendung von Service orientierten Architekturen und deren Vorteile, aufbauend auf die zuvor erläuterte Problematik, erklärt. In den Grundlagen werden außerdem Werkzeuge erklärt, mit deren Hilfe solche Architekturen realisiert werden können und die Wartung vereinfachen.

Darauf folgend wird das Architekturmodell "Microservice" unter der Verwendung von den erläuterten Werkzeugen erklärt und der Begriff Continuous Delivery eingeführt. Anschließend wird, unter Verwendung des angeeigneten Wissens aus dem Kapitel "Microservice", auf das Architekturmodell "Service-orientierte Architektur (SOA)" eingegangen.

Abschließend werden die gewonnenen Kenntnisse zusammengetragen und ausgewertet. Dabei sollen beide Architekturmodelle verglichen und bewertet werden. Zuletzt wird das Thema noch einmal zusammengefasst und ein Ausblick auf kommende Projekte bzw. die Bachelorarbeit gegeben.

# Kapitel 2

## Problemanalyse

Jedes Internet-Unternehmen fängt klein an und wächst mit der Zeit. In dieser Zeit muss das Unternehmen viele Probleme bewältigen. Es muss die Produkte, welches das Unternehmen verkauft, weiterentwickeln und auf die Bedürfnisse der Nutzer reagieren. Dabei ist die Wahl des Architekturmodells entscheidend.

### 2.1 Herausforderungen

Oft passiert es, dass ein Unternehmen große Vorstellungen von dem Unternehmens-Ziel hat. Dies führt häufig dazu, dass Software entwickelt wird, welches weit über den aktuellen Anforderungen hinaus gehen. Man möchte damit verhindern, Software an einem späteren Zeitpunkt neu zu entwickeln oder austauschen zu müssen. Jedoch kann dies zu großen Problemen führen sobald das Unternehmen wächst und den am Anfang genannten Vorstellungen näher kommt. In dieser Phase entstehen meistens Probleme, welche vorher nicht berücksichtigt worden sind, weil niemand sie kannte. Dadurch muss die vorhandene Software, welche eigentlich für dieses Szenario ausgelegt war, geändert werden.

Wie es bereits in **1.1 Motivation** erläutert wurde, ist die Zeitspanne es Time-to-Market für ein Unternehmen von äußerster Bedeutung. Damit diese Zeitspanne möglichst gering ist, bestehen besondere Anforderungen an Software. Vor allem, wenn es um die Einführung neuer Funktionen geht entstehen meistens lange TTM-Zeiten.

## 2.2 Hypothetisches Beispiel in Anlehnung an ein reales Problem

Ein Modernes und aktives Internet-Unternehmen ist eBay und darf man der Seite [5] glauben, so startet die Seite 1995 als Monolithische Perl Applikation. Selbst wenn diese Informationen nicht auf eBayzutreffen sollten, so dient es uns doch als gutes Beispiel einer realen Problemstellung.

eBay wurde 1995 von Pierre Omidyar unter dem Namen *ActionWeb* gegründet und wurde 1997 in eBay umbenannt. eBay wurde wie oben schon angesprochen als Monolithische Perl Anwendung implementiert (siehe [2]). Mit der Steigerung der Reichweite und der täglichen Benutzung, hatte man sich dann aber dazu entschlossen auf C++ als Code-Basis umzusteigen und die Seite mit CGI zu implementieren. Mittlerweile war eBay ein großes und sich rasant entwickelndes Unternehmen. Das bedeutet aber auch, dass eBay ständig auf das Verhalten der Nutzer reagieren und sich anpassen muss. Man versuchte also eine Monolithische Applikation mit der Fähigkeit auszustatten, auf Änderungen schnell zu reagieren, implementieren und deployen. Aber ein Monolith zu deployen, bedeutet, entweder die gesamte Infrastruktur für Wartungszwecke offline zu nehmen und die Applikation neu zu deployen, oder Server im Parallel betrieb laufen zu lassen und jeden neuen Traffic auf die neue Version zu routen. Die letzte Methode bietet jedoch einige Schwierigkeiten, denn es könnten Änderungen eingebaut worden sein, welche im Konflikt mit der alten Version stehen. Dann muss in jedem Fall die erste Variante gewählt werden und die gesamte Infrastruktur offline genommen werden.

Beide Varianten der Änderungen sind jedoch zeitaufwendig und schwierig, denn nicht nur das deployen könnte Probleme bereiten, sondern auch die darauf folgende Ausführung des Programms. Ändert man Code in Monolithischen Applikationen kann das auch Auswirkungen auf bestehende Teile des Codes haben, welche vorher, ohne Probleme, funktioniert haben. Werden Tests vernachlässigt oder wird nicht ausreichend getestet, kann es leicht passieren, dass sich ungewollt Fehler einschleichen, wodurch dann eine Version in betrieb genommen wird, welche Fehler enthält. Darauf folgend müssten diese wieder behoben werden und die Anwendung erneut deployed werden.

Im Falle einer Monolithischen Architektur bedeutet das viele Änderungen und neue Features. Oft passiert es daher, dass Code Stücke zurückbleiben, welche nicht mehr benötigt werden. Irgendwann ist die Applikation daher so groß, dass sie nicht mehr Wartbar ist und neue Features nur noch schwer zu implementieren sind. Entstehen Fehler in solch einer Anwendung ist es um so schwerer diese zu finden und zu beheben. Schließlich hat sich eBay entschlossen ihre Anwendung in Java neu zu implementieren. Dieses mal jedoch mit dem Hintergrund einer leicht erweiterbaren und wartbaren Architektur.

## **2.3 Die zu untersuchende Fragestellung**

Das Problem besteht also darin, eine Anwendung "flexibel und einfach erweiterbar zu gestalten, damit ein Unternehmen schnell auf Änderungen und die veränderten Bedürfnisse der Nutzer reagieren kann. Damit solch eine Software ebenfalls gut Wartbar ist, sollten Redundanzen möglichst vermieden werden. Hier steigen wir in Modularität ein, denn damit eine Software möglichst einfach Wartbar ist, sollte jeder Code mit einem bestimmten Kontext in ein eigenes Modul gepackt werden. So können zum Beispiel alle Codestücke einer bestimmten Berechnung in ein Modul gepackt werden, wodurch diese dann nur an einer zentralen Stelle geändert werden muss. Wenn die Module jedoch auch von verschiedenen Applikationen genutzt werden, müssen diese in Libraries ausgelagert werden. Auch hier hat man wieder nur eine zentrale Stelle, an der Änderungen vorgenommen werden müssen, um die Berechnung anzupassen oder zu ändern. Jedoch besteht hier das Problem, dass wenn Änderungen durchgeführt werden, diese noch lange nicht in jeder laufenden Applikation zu finden sind. Dazu müssen diese nämlich erst einmal neu gebaut und deployed werden, damit diese Produktiv werden.

Um dieses Vorgehen zu vereinfachen hat man sich dazu entschieden, diese Libraries in eigene Applikationen (Services) zu verpacken und diese über eine Schnittstelle anzubieten. Das sorgt dafür, dass jede Software, welche dieses Modul benötigt, sie nicht mehr eigenständig implementieren muss, sondern den dafür vorgesehenen Service aufrufen kann, um die nötigen Informationen zu erhalten. Diese Services werden auch als Microservices bezeichnet, da sie nur einem bestimmten Zweck

dienen, dafür ihre Aufgabe aber besonders gut erledigen. Hierbei kann man unter anderem von einer Service-orientierte Architektur (SOA) sprechen. Es sollte jedoch darauf geachtet werden, dass weder zu viel, noch zu wenig in ein Service gepackt wird. Welche Größe genau richtig ist, wird im Kapitel **3 Allgemeine Grundlagen** weiter erläutert.

# **Kapitel 3**

## **Allgemeine Grundlagen zur Verwendung von Service-orientierten Systemen**

Software zu entwickeln ist nicht immer einfach. Umso größer diese ist, umso mehr Probleme können auftreten. Es bedarf einer genauen Planung und Verständnis von Infrastruktur um eine Software mit allen Anforderungen zufriedenstellend zu implementieren.

Vor allem wenn es um die Weiterentwicklung und Wartung von Software geht, können große Probleme auftreten. Wurde die Architektur nicht gut gewählt oder schlecht umgesetzt, kann es das weitere Vorgehen stark beeinträchtigen, bis hin zum unmöglich machen. Es wurden daher Software-Architekturen entwickelt, welche flexibel und einfacher zu ändern sind. Außerdem kann in diesen Architekturen neue Funktionen deutlich schneller hinzugefügt werden.

### **3.1 Architektur**

Microservice-Architekturen und Service-orientierte Architektur(SOA) Architekturen können, wenn sie richtig angewendet werden, sehr flexibel und schnell änderbar sein. Beide Architekturen zielen, wie der Name schon sagt, auf eigenständige Services ab, welche durch verschiedene Kommunikationskanäle miteinander kommunizieren und dadurch die gewünschten Geschäftsprozesse abbilden.

»Ein Programm soll nur eine Aufgabe erledigen, und das soll es gut machen«[7, S. 2]

Anstatt eine einzige große Anwendung einzusetzen, setzt man auf viele kleine, verteilte, autarke Anwendungen, welche jeweils Schnittstellen nach außen hin anbieten damit der Service genutzt werden kann. Diese Schnittstellen können unter anderem durch REST-HTTP angeboten werden. Durch die verteilten Anwendungen funktioniert das System auch dann noch, wenn einzelne Services nicht verfügbar sind, jedoch bringt es ebenfalls die typischen Probleme von Verteilten Anwendungen mit sich, welche in ?? ?? noch genauer erläutert werden.

### **Services kann man in drei Kategorien einteilen:**

**Producer** Ein Service der etwas Produziert oder auf eine Anfrage reagiert. Das reicht von Daten aus einer Datenbank extrahieren bis hin zu komplexen Berechnungen.

**Consumer** Ein Service oder eine Anwendung, welche einen oder mehrere Produzierende Services verwendet und entweder weiterverarbeitet oder ausgibt. Im Falle der Weiterverarbeitung ist ein Consumer ebenfalls ein Producer sein.

**Self-Contained System (SCS)** "»Microservice mit UI« oder »Self-Contained System« wie es Stefan Tilkov nennt, sind in sich abgeschlossene Systeme. [...] Sie enthalten eine UI und sollten möglichst nicht mit anderen SCS kommunizieren." [8, vgl S. 55]. SCS sind jedoch nur im Microservice und nicht im SOA Umfeld zu finden. Warum wird in ?? ?? weiter erläutert.

### **3.1.1 Das Gesetz von Conway**

Wie in [8, S. 39 ff.] beschrieben, stammt das Gesetz von dem amerikanischen Informatiker Melvin Conway und besagt:

*Organisationen, die Systeme designen, können nur solche Designs entwerfen, welche die Kommunikationsstruktur dieser Organisationen abbilden.*

"Conway möchte damit ausdrücken, dass die internen Kommunikationswege wichtig bei der Planung der Architektur ist. Jedes Team innerhalb einer Organisation

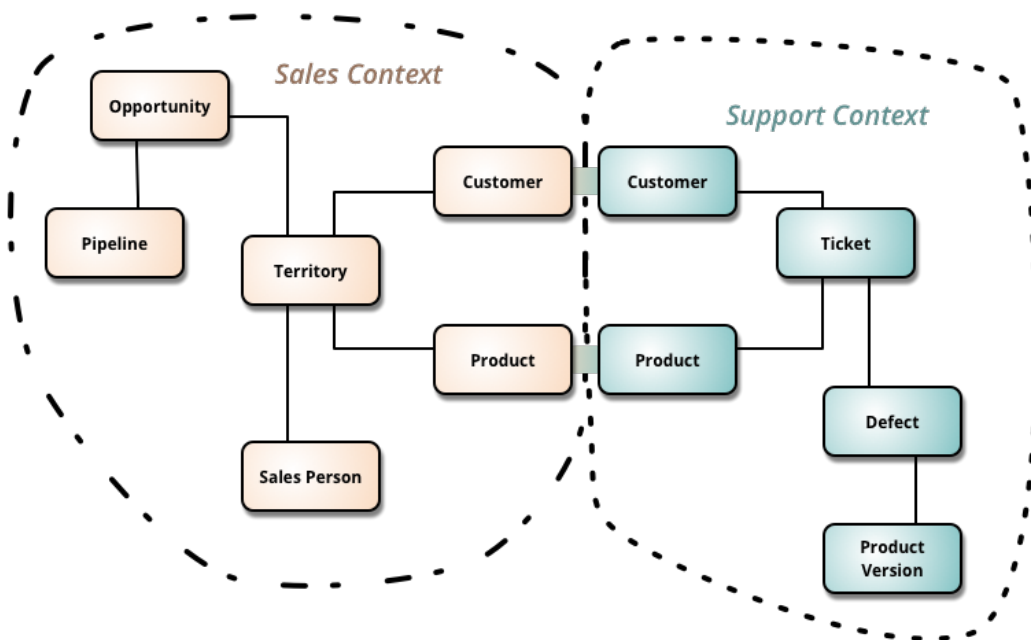


trägt bei der Entwicklung der Architektur bei. Wird eine Schnittstelle zwischen zwei Teams benötigt, so müssen diese Teams auch kommunizieren können. Dabei müssen Kommunikationswege nicht immer offiziell sein. Oft gibt es informelle Kommunikationsstrukturen, die ebenfalls in diesem Kontext betrachtet werden können." [8, vg. S. 39]

### 3.1.2 Domain-Driven Design und Bounded Context

Arbeitet man mit Service-Orientierten Architekturen, versucht man Services, welche zu einem bestimmten Kontext gehören, möglichst nahe beieinander zu halten. Man spricht hierbei von *Bounded Context*.

»Bounded Context ist ein zentrales Muster in Domain-Driven Design.[...] DDD arbeitet mit großen Modellen, indem es diese in kleine verschiedene zusammengehörige Kontexte unterteilt und auf ihre Wechselwirkung unterteilt.« [4]



Quelle: <http://martinfowler.com/bliki/BoundedContext.html>

**Abbildung 3.1:** Bounded Context

In dieser Grafik wird noch einmal der Begriff Bounded Context genauer verdeutlicht. Es existieren zwei eigenständige Prozesse. Auf der linken Seite der Sales Kon-

text und auf der rechten Seite der Support Kontext. Jeder Kontext besitzt verschiedene Services, welche benötigt werden um den Prozess durchführen zu können. Lediglich zwischen den *Customer* und *Product* Services besteht eine Verbindung der beiden Prozesse.

# Kapitel 4

## Service-orientierte Architektur (SOA)

Der Begriff “Service-orientierte Architektur (kurz SOA)“ ist nicht eindeutig definiert. Je nachdem welche Person in einem Unternehmen man fragt, erhält man unter Umständen eine komplett andere Definition. »Die Meisten Definitionen stimmen jedoch zum größten Teil überein und stehen nicht im Konflikt mit einander.«[6, vgl. Seite 6]

Für einen Kaufmann ist SOA etwas anderes als für einen Analysten, damit SOA jedoch verstanden werden kann, werden zunächst einmal einige Definitionen nach [6] genannt:

1. »To the chief information officer (CIO), SOA is a journey that promises to reduce the lifetime cost of the application portfolio [...].«[6, vgl. Seite 6]
2. »To the business executive, SOA is a set of services that can be exposed to their customers, partners, and other parts of the organization. Business capabilities, function, and business logic can be combined and recombined to serve the needs of the business now and tomorrow. Applications serve the business because they are composed of services that can be quickly modified or redeployed in new business contexts, allowing the business to quickly respond to changing customer needs, business opportunities, and market conditions.«[6, vgl. Seite 6]
3. »To the business analyst, SOA is a way of unlocking value, because business

processes are no longer locked in application silos. Applications no longer operate as inhibitors to changing business needs.«[6, vgl. Seite 6]

4. »To the chief architect or enterprise architect, SOA is a means to create dynamic, highly configurable and collaborative applications built for change. SOA reduces IT complexity and rigidity. SOA becomes the solution to stop the gradual entropy that makes applications brittle and difficult to change. SOA reduces lead times and costs because reduced complexity makes modifying and testing applications easier when they are structured using services.«[6, vgl. Seite ]

Jeder der genannten Rollen hat eine eigene klare Definition von dem was Service-orientierte Architektur ist. Jeder der Definitionen ist jedoch nur ein Teil dessen für was SOA verwendet werden kann.

## **4.1 Grundlagen**

Bevor jedoch damit angefangen werden kann SOA in ein Unternehmen einzuführen, muss zunächst einmal geklärt werden, welches Ziel hinter Service-orientierte Architekturen stehen. Zunächst einmal sei erwähnt, das SOA nicht das Ziel hat die Entwicklung zu vereinfachen oder voran zu bringen. SOA soll die Unternehmensweiten Geschäftsprozesse optimieren und eine Wertschöpfungskette erzeugen.

### **4.1.1 Unternehmens Komponenten**

Dazu müssen zunächst einmal alle Komponenten eines Unternehmens identifiziert werden. Die nachstehende Abbildung (4.1) zeigt ein Beispiel dieser Identifizierung:

	Business Administration	New Business Development	Relationship Management	Servicing & Sales	Product Fulfillment	Financial Control and Accounting
Directing	Business Planning	Sector Planning	Account Planning	Sales Planning	Fulfillment Planning	Portfolio Planning
Controlling	Business Unit Tracking	Sector Management	Relationship Management	Sales Management	Fulfillment Monitoring	Compliance
	Staff Appraisals	Product Management	Credit Assessment			Reconciliation
Executing	Account Administration	Product Directory	Credit Administration	Sales	Product Fulfillment	Customer Accounts
	Product Administration	Marketing Campaigns		Customer Service	Document Management	
	Purchasing			Collections		General Ledger
	Branch/Store Operations					

Quelle: [http://www.jot.fm/issues/issue\\_2008\\_05/column5/](http://www.jot.fm/issues/issue_2008_05/column5/)

**Abbildung 4.1:** Unternehmens Komponenten

Aus diesen Komponenten müssen nun die Geschäftsprozesse identifiziert werden. Daraus ergeben sich anschließend die Komponenten, welche unter einander kommunizieren müssen. Die in der Abbildung Rot dargestellten Komponenten sind schließlich das Resultat aus der Analyse.

### 4.1.2 Wertschöpfungskette

»Die Wertschöpfungskette stellt die zusammenhängenden Unternehmensaktivitäten des betrieblichen Gütererstellungsprozesses grafisch dar.«[1]

SOA soll dabei helfen die Wertschöpfungskette sowohl darzustellen, als auch sie zu verwalten und damit zu optimieren. In Abbildung 4.1 sind die zentralen Komponente eines Geschäftsprozesses rot markiert. Mit Hilfe von SOA werden aus diesen Komponenten eine Wertschöpfungskette gebildet.

## 4.2 Architektur

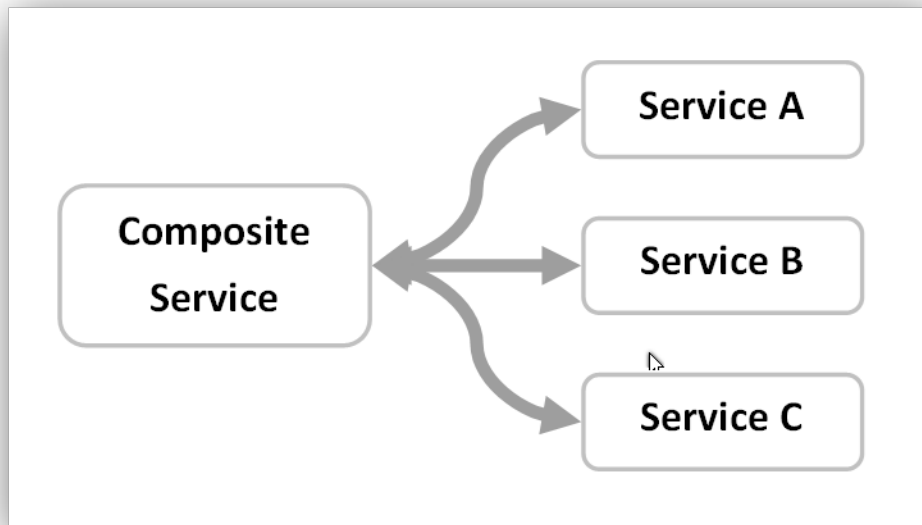
Aus der oben dargestellten Auflistung können verschiedene Definitionen für Service-orientierte Architekturen entnommen werden. Die Architektur kommt der 3. Definition schon ziemlich nahe. Genauso wie Microservices sind auch die Services innerhalb einer Service-orientierten Architektur eigenständig und sollen nur eine Aufgabe erledigen. Anders als bei einem Microservice-System jedoch, bei dem die

einzelnen Services untereinander kommunizieren dürfen, ist in einem SOA-System die Kommunikation zwischen einzelnen Services untersagt.

Die Abbildung der Geschäftsprozesse erfolgt in einem sogenannten Enterprise-Service-Bus (Kurz ESB), welcher die einzelnen Services Orchestriert (siehe Abbildung 4.3).

#### **4.2.1 Orchestration**

Bei der Orchestration handelt es sich um eine Komposition von Services. Ein Geschäftsprozess wird zwar mit Hilfe von mehreren Services abgebildet, jedoch ist nur ein Service dafür zuständig den Geschäftsprozess durchzuführen.



**Abbildung 4.2:** Orchestration

Wie die Abbildung zeigt besteht keine Verbindung zwischen:

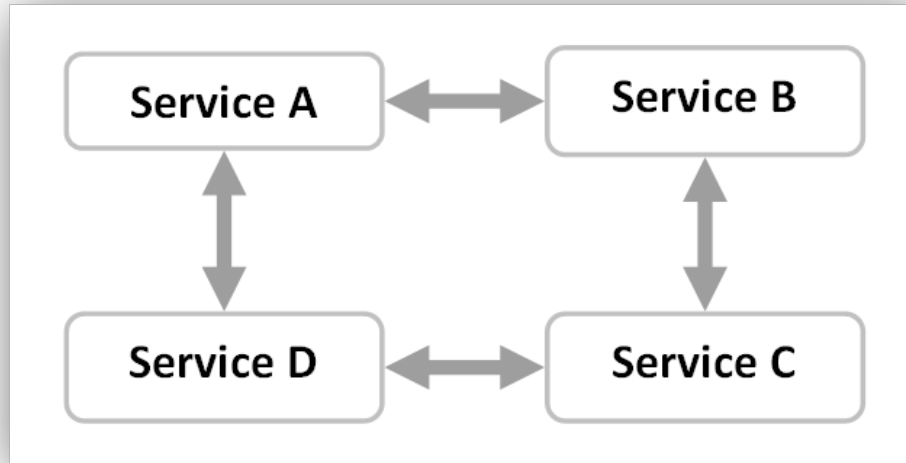
- A & B
- A & C
- B & C

Nur der „Composite Service“ nutzt die anderen Services, um den Geschäftsprozess abzubilden. Diese Art der Kommunikation nennt man Orchestration.

### 4.2.2 Choreographie

Anders als bei der Orchestration können Services bei der Choreographie beliebig untereinander kommunizieren. Das ist vor allem dann sinnvoll, wenn verschiedene Service andere Service über Änderungen oder andere Aktionen informieren müssen.

Wie bereits oben erwähnt ist der Enterprise Service Bus die zentrale Einheit bei der Service-orientierten Architektur (SOA) über die alle Kommunikation läuft und welcher die Geschäftsprozesse abbildet. Daher ist eine Choreographie unter den Services in diesem Architekturmodell nicht möglich.



**Abbildung 4.3:** Choreographie

### 4.3 Enterprise Service Bus - ESB

Der *Enterprise Service Bus* ist die zentrale Einheit in einem SOA-System. Am einfachsten lässt sich das an einem Beispiel erklären.

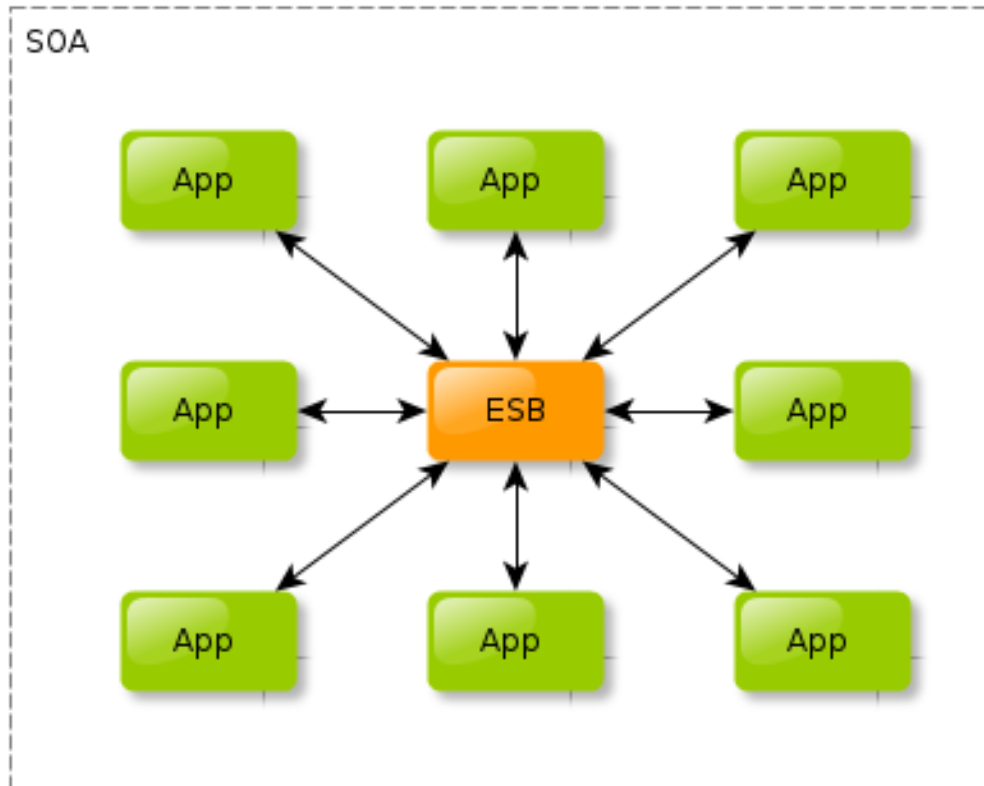
Man Stelle sich eine Banking-Anwendung vor, an der man sich anmeldet. Es werden daraufhin folgende Informationen angezeigt:

1. Name
2. Kontostand
3. EC- und Kreditkarten
4. Liste der Aktienfonds

Jede Information stammt aus einem anderen Teil des Systems und werden von verschiedenen Anwendungen über Schnittstellen bereitgestellt. Die Schnittstellen können sich dabei von Anwendung zu Anwendung unterscheiden. So kann zum Beispiel eine Anwendung die Informationen über HTTP bereitstellen, während eine andere sie über SOAP bereitstellt. So können die Informationen zum Beispiel aus einem CRM-System (Kundenbeziehungsmanagement-System) stammen oder durch PHP oder Ruby erzeugt werden.



Anders als man jedoch vermuten würde, lässt man die Oberfläche, bzw. das System, nicht direkt mit den einzelnen Komponenten reden. Die gesamte Kommunikation läuft dabei über den *Enterprise Service Bus* ab. Nachstehende Abbildung soll dies genauer erläutern:



**Abbildung 4.4:** Enterprise Service Bus

Quelle: <https://zato.io/docs/intro/esb-soa-de.html>

Benötigt eine Anwendung (in der Abbildung App genannt) Informationen, konsultiert diese zunächst den ESB. Der ESB kennt die anderen Anwendungen und stellt daraufhin die angeforderten Informationen bereit.

# Kapitel 5

## Microservices

### 5.1 Überblick

»Modularisierung ist nichts Neues. Schon lange werden große Systeme in kleine Module unterteilt, um Software einfacher zu erstellen, zu verstehen und weiterzuentwickeln. Das Neue: Microservices nutzen als Module einzelne Programme, die als eigene Prozesse laufen. Der Ansatz basiert auf der UNIX-Philosophie. Sie lässt sich auf drei Aspekte reduzieren:«[8, S. 2]

- Ein Programm soll nur eine Aufgabe erledigen, und das soll es gut machen.
- Programme sollen zusammenarbeiten können.
- Nutze eine universelle Schnittstelle. In UNIX sind das Textströme.

Diese Art der Aufteilung wurde schon lange von großen Unternehmen wie Amazon oder Google genutzt und wurde zu nächst Service-orientierte Architektur(SOA) genannt. Jedoch unterscheiden sich Microservices und SOA voneinander. Daher wird SOA im Nächsten Kapitel genauer erläutert und im Kapitel 6 die Ergebnisse zusammen gefasst und beide Technologien mit einander verglichen.

Der Begriff Microservices ist nicht eindeutig definiert. Als erste Näherung dienen, nach Eberhard Wolff [8, S. 2], folgende Kriterien:

- Microservices sind ein Modularisierungskonzept. Sie dienen dazu, ein großes Software-System aufzuteilen - und beeinflussen die Organisation und die Software-Entwicklungsprozesse.
- Microservices können unabhängig von Änderungen an anderen Microservices in Produktion gebracht werden.
- Microservices können in unterschiedlichen Technologien implementiert sein. Es gibt keine Einschränkung auf eine bestimmte Programmiersprache oder Plattform.
- Microservices haben einen eigenen Datenhaushalt: eine eigene Datenbank - oder ein vollständig getrenntes Schema in einer gemeinsamen Datenbank.
- Microservices können eigene Unterstützungsdienste mitbringen, beispielsweise eine Suchmaschine oder eine spezielle Datenbank. Natürlich gibt es eine gemeinsame Basis für alle Microservices - beispielsweise die Ausführung virtueller Maschinen.
- Microservices sind eigenständige Prozesse - oder virtuelle Maschinen, um auch die Unterstützungsdienste mitzubringen.
- Dementsprechend müssen Microservices über das Netzwerk kommunizieren. Dazu nutzen Microservices Protokolle, die lose Kopplung unterstützen. Das kann beispielsweise REST sein - oder Messaging-Lösungen.

Obwohl der Begriff nicht eindeutig definiert ist, liegt jedoch bei jedem Service-orientierten System, ein Verteiltes System zu Grunde und auch die damit verbundenen Probleme und Herausforderungen.

## 5.2 Größe von Microservices

»Der Name »Microservices« verrät schon, dass es um die Servicegröße geht - offensichtlich sollen die Services klein sein.«[8, S. 31]  
Es gibt verschiedene Möglichkeiten die Größe von Programmen zu ermitteln. Eine Variante ist zum Beispiel das Zählen von Lines of Code

(LOC), jedoch hat diese Methode auch Nachteile. Denn die Anzahl der Codezeilen hängen stark von der verwendeten Programmiersprache ab. Einige Programmiersprachen benötigen mehr Zeilen Code, um eine bestimmte Tätigkeit abzubilden, als andere.

Die Größe von Services sollte jedoch nicht von zentraler Bedeutung sein, denn eine untere Grenze gibt es für Services nicht. "Wohl aber eine obere Grenze: Wenn der Microservice so groß ist, dass er von einem Team nicht mehr weiterentwickelt werden kann, ist sie zu groß. Ein Team sollte dabei eine Größe haben, wie sie für agile Prozesse besonders gut funktioniert. Das sind typischerweise drei bis neun Personen.«[8, S. 34]

Bei der Größe eines Services ist jedoch darauf zu achten, dass ein Service nicht zu viele oder zu wenige Funktionen besitzt. Wie bereits beschrieben, sind Microservices modulare, lose gekoppelte Services. Wird ein Service zu klein angesetzt, können daraus Abhängigkeiten zu anderen Services entstehen und damit das Gesetz der losen Kopplung verletzt werden. Besitzt hingegen ein Service zu viele Funktionen, wird es meistens nicht mehr als Microservice angesehen, da es nicht eine, sondern mehrere Aufgaben übernimmt und diese wahrscheinlich nicht mehr gut erledigen kann.

## 5.3 Orchestration vs Choreographie

Möchte man ein Microservice System aufbauen, stellt sich die Frage, wie einzelne Services strukturiert werden und wie diese untereinander kommunizieren sollen. Ein bestimmter Vorgang startet in der Regel bei einem Service. Nun muss man entscheiden, ob weitere Services hinzugezogen, beziehungsweise informiert werden müssen. Je nach Anwendungsfall muss man sich zwischen Service Orchestration und Choreographie entscheiden. Dabei ist es fast unmöglich ein ganzes Microservice-System aus nur einem der beiden Varianten zu bauen.

### 5.3.1 Herausforderung

Wie bereits in [5.1 Überblick](#) erwähnt, liegt ein Verteiltes System zu Grunde und damit auch die Grundlegenden Probleme der Kommunikation (siehe [8, S. 25]). Der Ausfall eines Services kann im schlechtesten Fall dazu führen, dass alle anderen Microservices nicht mehr funktionieren. Um das zu verhindern muss klar definiert werden, was Microservices in dieser Situation tun sollen. Zusätzlich muss, sofern Datenbank Operationen eine wichtige Rolle spielen, das Problem der einheitlichen Transaktion gelöst werden. Wenn zum Beispiel eine Operation Daten über verschiedene Microservices in Datenbanken schreibt, muss bei nicht Erreichbarkeit oder Fehlers eines Microservices eine einheitlicher Rollback durchgeführt werden, um keine inkonsistente Dateien im System zu haben. Eine weitere Herausforderung besteht in dem Grundkonzept von Microservices. Da nicht definiert ist, welche Programmiersprache für Microservices verwendet wird, kann ein Service zum Beispiel in Java, ein anderes in Scala oder Python geschrieben werden. Es muss daher dafür gesorgt werden, dass die einzelnen Services untereinander interoperabel sind. Um das zu gewährleisten, müssen die Schnittstellen möglichst einheitlich und auf dem gleichen Protokoll aufbauend programmiert werden. Hier bieten REST-Schnittstellen eine gute Lösung. Diese können auf dem HTTP-Protokoll aufgebaut werden. Zusätzlich bietet das HTTP-Protokoll die Möglichkeit, ein einheitliches Medium, wie zum Beispiel XML oder JSON, als Informationsträger zu nutzen. Dabei kann theoretisch jeder Microservice mit jedem anderen Microservice kommunizieren, sofern die Schnittstellen einheitlich definiert sind.

## 5.4 PUSH- VS PULL-Architektur

Grundlegend können Microservices mit Hilfe zwei verschiedener Kommunikations-Architekturen kommunizieren, PUSH- und PULL-Architektur. Dabei ist jedoch nicht ausgeschlossen, dass sobald eine Architektur gewählt worden ist, die andere nicht mehr genutzt werden kann. Genauso wie bei der Entscheidung über die Kommunikationsstruktur (siehe [4.2.1 Orchestration](#)), kann es von Vorteil sein, beide Architekturen zu nutzen.

## **PULL-Architektur**

Eine PULL-Architektur basiert auf einem einfachen Request-Replay-Schema. Dementsprechend ist das Web PULL-basiert. Der Browser macht eine Anfrage an einen Server, dieser wiederum verarbeitet die Anfrage und liefert eine Antwort (Replay) zurück. Dies hat den Vorteil, dass nicht lange auf eine Antwort gewartet werden muss und die teilhabenden Kommunikationspartner gegenseitig kennen, jedoch bringt es auch den Nachteil, dass dadurch weitgehend eine synchrone Kommunikation stattfindet und eine Antwort häufig nicht gleichzeitig an mehrere Empfänger senden kann.

## **PUSH-Architektur**

Eine PUSH-Architektur wird eingesetzt, wenn man verschiedene Kommunikationspartner über bestimmte Ereignisse informieren möchte. Hier stehen meist nicht die Kommunikationspartner, sondern die Informationen im Vordergrund. Dafür wird meistens ein eigenständiger Service (Broadcaster) eingesetzt, der die Verteilung dieser Informationen übernimmt. Dabei kann ein Service als Informationsprovider dienen, zum Beispiel ein Nachrichten-Feed (Von einer Nachrichtenseite). Alle anderen Services abonnieren den Broadcaster und erhalten dadurch alle Nachrichten, die der Informationsprovider sendet. Es gibt jedoch auch den Fall, dass die Kommunikation sternförmig um den Broadcaster angeordnet sind. Dadurch ist jeder Service der diesen abonniert, sowohl Provider, als auch Consumer. Anders als bei PULL-basierten Systemen kann hier nicht unbedingt sichergestellt werden, dass alle Nachrichten von allen Konsumenten gleichzeitig gelesen und ggf. verarbeitet werden. Jedoch können so Informationen innerhalb eines Microservice-Systems relativ zuverlässig verteilt werden. Der Vorteil von PUSH-Architekturen ist, dass eine asynchrone Informationsverbreitung aufgebaut werden kann. Zudem können Serviceausfälle, solange es nicht der Broadcaster oder wichtige Microservices sind, überbrückt werden, indem der Broadcaster die Nachrichten für eine bestimmte Zeit vorhält und so der Microservice, welcher nicht erreichbar war, die Nachrichten trotzdem noch erhält.

Oft ist es nicht notwendig eine Antwort zu erhalten. Zum Beispiel muss eine Registrierung in unserem fiktiven Unternehmen, der OnlineCommerceShop GmbH mög-

lich sein. Dabei sendet der Microservice der für die Registrierung zuständig ist eine einfache Event-Nachricht, wie Benutzer XY hat sich Registriert. Im Hintergrund kann dann zum Beispiel ein anderer Microservice diese Nachricht erhalten und zusätzliche Aktionen durchführen, wie erstellen des Warenkorbs.



# Kapitel 6

## Ergebnisse

Nachdem erläutert wurde, was unter **Microservice** und **Service-orientierte Architektur (SOA)** zu verstehen ist, werden nur beide Architektur Modelle mit einander Verglichen und kritisch betrachtet. Anschließend wird aus dem Vergleich ein Fazit gezogen und dieses erläutert.

Bevor die Architekturmodelle mit einander verglichen werden können, muss erwähnt werden das ein direkter Vergleich zwischen Funktionen und den Vor- und Nachteilen der jeweiligen Modelle nicht möglich ist. Außerdem ist weder der Begriff Microservice noch Service-orientierte Architektur eindeutig definiert. Literaturen wie [6] und [8] bieten einen guten Ansatz, um diese beiden Begriffe grob einzuordnen.

### 6.1 Vergleich

Wie bereits erwähnt können die beiden Architekturmodelle nicht direkt miteinander verglichen werden, daher wird zunächst einmal der Grundlagen beider Modelle Verglichen.

#### 6.1.1 Grundlagen

Der Grundgedanke beider Architekturen ist sehr unterschiedlich. Zwar stammen beide Modellarten aus dem Bereich der „Service orientierten Architekturen“ und auch *SOA* ist eine Abkürzung dieses Begriffes, jedoch verfolgen beide Modelle ver-

schiedene Ansätze.

SOA hat seine Grundlagen nicht in der IT, sondern in Geschäftswelt. Dies wird dadurch bestätigt, dass die Zielgruppe der Befragten in Kapitel 4 **Service-orientierte Architektur (SOA)** aus dem Kaufmännischen und analytischen Bereich stammen. So wird zum Beispiel der «business executive» und der «business analyst» gefragt, wie SOA zu verstehen ist.

Microservices hingegen stammen aus der Notwendigkeit große Software möglichst effizient zu erstellen und die Wartung zu vereinfachen. Wie auch schon in Kapitel 5.1 **Überblick** erwähnt, ist Modularisierung nichts neues und wird aus den gerade genannten Notwendigkeiten eingesetzt. Die Microservice-Architektur ist nichts anderes als die Modularisierung einer Software, bei der diese in kleine Software Pakete, sogenannte Services, geteilt werden.

Während bei SOA das Geschäft und die Geschäftsprozesse im Vordergrund steht, will man mit Microservices die Entwicklung von Software unterstützen. Das Microservice-Modell wurde entwickelt, damit große Software möglichst einfach und schnell mit vielen Personen entwickelt werden kann. Bei SOA geht es um den möglichst effizienten Einsatz von Software und nicht deren Entwicklung. Oftmals existiert schon Unternehmenssoftware. Durch SOA soll diese möglichst effizient untereinander kommunizieren.

Grundsätzlich wurden schon viel früher Architektur-Modelle entwickelt, um die in den jeweiligen Kapiteln 4 **Service-orientierte Architektur (SOA)** und 5 **Microservices** genannten Probleme und Anforderungen umzusetzen. Ein Begriff für diese Modelle existierte jedoch noch nicht. Die jeweiligen Begriffe wurden erst später verwendet, um die jeweiligen Modelle spezifizieren und erklären zu können.

## **6.1.2 Architektur**

Ein großer Unterschied der beiden Modelle liegt in der Architektur. Da beide Modelle aus dem Gebiet der verteilten Anwendungen stammen, spielt die Kommuni-

kation hinsichtlich der Architektur eine wichtige Rolle. Die Kommunikation der beiden Modelle soll daher zu einem späteren Zeitpunkt im Kapitel **6.1.3 Kommunikation** verglichen werden und wird in diesem Kapitel außer acht gelassen.

Wie bereits erwähnt liegen beide Modelle der Architektur der verteilten Anwendungen zur Grunde, jedoch unterscheiden sie sich hinsichtlich der Strukturierung. Grundsätzlich benutzen beide Modelle eigenständige Dienste, welche sich jedoch hinsichtlich ihrer Größe unterscheiden. Dies kommt dadurch zur Stande, dass beide Modelle einen unterschiedlichen Ansatz besitzen.

Wie bereits weiter oben erläutert, wird mit SOA versucht verschiedene Geschäftsprozesse miteinander zu verknüpfen und eine leichtere Kommunikation unter diesen zu ermöglichen. Dadurch sind die meisten Dienste nicht Bestandteil einer großen Software, sondern nur Bestandteil eines Unternehmensbereiches. Oft wird daher versucht alle EDV-Komponenten miteinander zu verknüpfen ohne eine Abhängigkeit zu erschaffen. Dabei sind die einzelnen Dienste oft eigenständige Anwendungen wie ERP oder COBALD Systeme. Um dies zu ermöglichen wird bei SOA ein sogenannter „Enterprise-Service Bus (ESB)“ eingesetzt, welcher Frontend Dienste wie eine Website mit Datenbanken und Server-Anwendungen verbindet. Der ESB bildet dabei die Zentrale Einheit einer SOA-Architektur und dient außerdem als Dienst-Register, der jegliche Dienste in einem Unternehmen kennt und weiß wie sie anzusprechen sind.

Das Microservice-Modell arbeitet zwar auch mit Diensten, jedoch sind diese meist nicht so groß wie im SOA-Modell. Während bei dem SOA-Modell auch ganze Anwendungen, wie ERP-Systeme als Services verwendet werden, wird mit dem Microservice-Modelle versucht eine ganze Anwendung in eigenständige Dienste aufzuteilen. Dabei soll ein Dienst nur eine Aufgabe erledigen, diese aber jedoch besonders gut. Zudem gibt es in der Regel keine zentrale Anwendung, welche als Schnittstelle zwischen Datenbanken und Frontend dient.

### 6.1.3 Kommunikation

Nachdem die Architektur Unterschiede erläutert wurden, werden nun die Unterschiede in der Kommunikation erläutert. Grundsätzlich liegt jedoch bei beiden Architektur-Modellen, wie auch schon in 6.1.2 Architektur erwähnt, das Modell der verteilten Anwendungen zur Grunde und auch die damit verbundenen Probleme.

Die Modelle unterscheiden sich jedoch hinsichtlich der Anordnung von Diensten und damit der Kommunikationsfluss innerhalb der Modelle. Grundsätzlich kann man zwischen zwei Kommunikationsflüssen unterscheiden:

1. Orchestration
2. Choreographie

Nicht beide Kommunikationsflüsse sind in beiden Service-orientierten Architekturen möglich. In SOA ist der ESB die zentrale Einheit über welches die gesamte Kommunikation läuft. Dadurch ist nur eine Orchestration der Dienste möglich und die direkte Kommunikation zwischen den Diensten ist nicht möglich. Bei Microservices hingegen können beide Kommunikationsflüsse realisiert werden, wodurch die Kommunikation viel dynamischer gestaltet werden kann, da jeder Dienst mit jedem anderen Dienst kommunizieren kann. Oftmals sprechen Frontend Anwendungen wie Websites direkt mit mehreren Services um bestimmte Informationen zu erhalten.

### 6.1.4 Beteiligte Personen

Neben den Technischen Unterschiede, gibt es auch Unterschiede hinsichtlich der Personengruppen die beteiligt sind. Dabei geht es nicht um die Nutzer einer Software, sondern um Personen die aktiv im Prozess der jeweiligen Modelle beteiligt sind.

Bei dem SOA-Modell wurden schon einige Personengruppen identifiziert, als die Unterschiede der Grundlagen erläutert wurden. Unter anderem gehören darunter Business Executiver und Business Analysten. Weitere Personengruppen sind zum

Beispiel Personen aus dem Kaufmännischen Bereich. Natürlich spielen auch Personen aus der IT-Abteilung eine Rolle, jedoch sind hier meistens nur Administratoren und Architekten mit gemeint, da SOA keinen direkten Bezug zur Entwicklung besitzt und ohne eine Entwicklungsabteilung verwendet werden kann.

Anders als beim SOA-Modell, steht beim Microservice-Modell die Entwicklung im Vordergrund. Dadurch ändern sich die Personengruppen stark. Während beim SOA-Modell hauptsächlich Personen aus dem Business Bereich beteiligt waren, ist beim Microservice-Modell kaum einer aus diesem Personenkreis beteiligt. Stattdessen sind Personen aus der Entwicklung und der IT beteiligt.

Die Unterschiede in den beteiligten Personen zeigen neben den unterschiedlichen Personengruppen auch in welchem Umfeld sich beide Architekturmodelle bewegen bzw. aus welchem sie entstanden sind.

## **6.2 Fazit**

## **Kapitel 7**

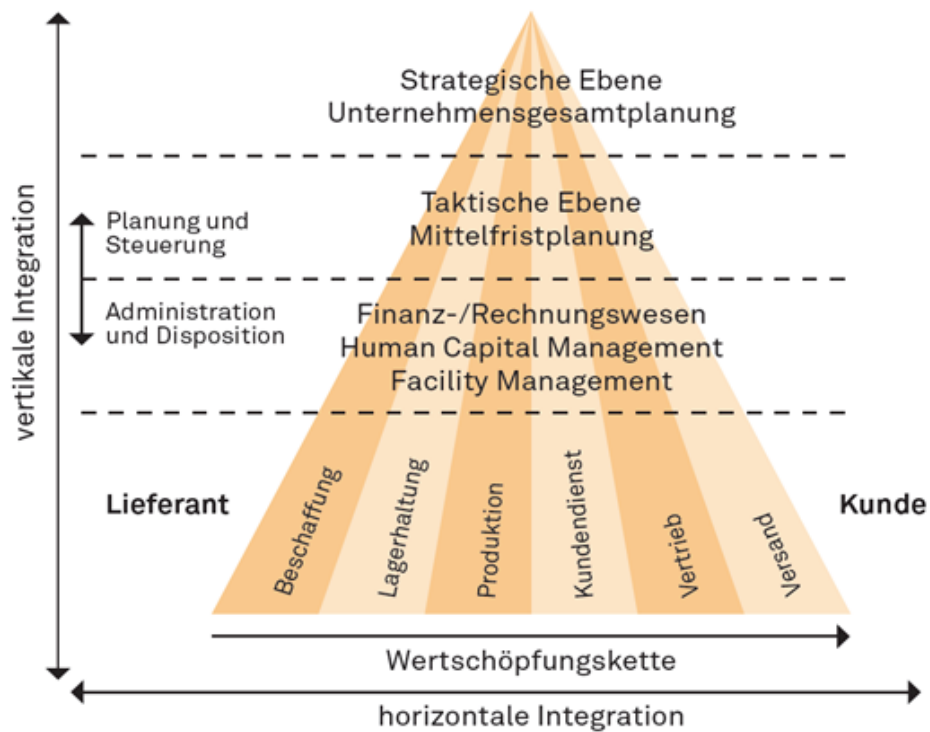
### **Zusammenfassung und Ausblick**

# Literaturverzeichnis

- [1] *Definition: Wertschöpfungskette*. <http://wirtschaftslexikon.gabler.de/Definition/wertschoepfungskette.ht>  
– visited: 06.08.2016
- [2] : *Ebay*
- [3] *Time-to-Market (TTM)*. <http://www.gruenderszene.de/lexikon/begriffe/time-to-market-ttm>. – Stand 30.04.2016
- [4] FOWLER, Martin: *BoundedContext*. <http://martinfowler.com/bliki/BoundedContext.html>.  
– Stand 09.05.2016
- [5] HOFF, Todd: *Deep Lessons from Google and eBay on Building Ecosystems of Microservices - High Scalability* -. <http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html>. – Stand 25.02.2016
- [6] KERRIE HOLLEY, Dr. Ali A.: *100 SOA Questions Asked and Answered*. <http://www.professores.uff.br/screspo/100-SOA-Questions.pdf>
- [7] WOLFF, Eberhard: *Continuouos Delivery - Der Pragmatische Einstieg*. Heidelberg : dpunkt.verlag, 2015. – ISBN 978-3-86490-208-6
- [8] WOLFF, Eberhard: *Microservices - Grundlagen flexibler Softwareentwicklung*. Heidelberg : dpunkt.verlag, 2016. – ISBN 987-3-86490-313-7

# Anhang A

## Diagramme und Tabelle



Quelle: <http://www.referenzportal.ch/fuehrung/vom-erp-zum-integrierten-informationssystem/>

**Abbildung A.1:** Darstellung einer typischen Integrations-Pyramide