

Dezentrale Systeme - Grundlagen und Prinzipien

Skriptum zur Vorlesung

Dipl.-Ing. Paul Panhofer BSc.^{1*}

1 ZID, TU Wien, Taubstummengasse 11, 1040, Wien, Austria

Abstract: Die Verbesserung der LEISTUNGSFÄHIGKEIT von **Rechnersystemen** und die Fortschritte in der NETZWERKTECHNOLOGIE haben in den letzten Jahren zu einer verstärkten **Dezentralisierung** von Diensten und Daten in Anwendungssystemen geführt. Zusammen mit dieser Entwicklung haben die Anstrengungen bei der STANDARDISIERUNG von **Kommunikationsdiensten** dazu beigetragen, daß Systeme verschiedener Hersteller miteinander verbunden werden können. Die Verbindungen werden hier zur *Datenübertragung* und zum *elektronischen Nachrichtenaustausch* benutzt.

Durch die Verbindung von leistungsstarken ARBEITSPLATZRECHNERN mittels NETZWERKEN konnte eine **Client/Server-Architektur** geschaffen werden. Manchesmal ist die Aufteilung des Anwendungssystems in Clients und Server für den Benutzer zu erkennen. Gegenwärtig versucht man verschiedene bestehende verteilte Anwendungssysteme zu einem einzigen verteilten Anwendungssystem zusammenzufassen, um den Informati-onsaustausch in großen Unternehmen und Organisationen effizienter zu gestalten.

Als Ergebnis erhält man hochintegrierte Anwendungssysteme, die mit dem Begriff **Enterprise computing** bezeichnet werden

MSC: paul.panhofer@gmail.com

Keywords: Dezentrale Systeme • verteilte Systeme • Prinzipien verteilter Systeme • Koppelung • Kohäsion • Open-Closed Prinzip • Client Server • SOA • Services • SOAP • WSDL • REST • JSON • Service Oriented Architecture • Microservices

Contents		
1. Einführung: SOA	6	1.3.1. Architekturnuster 8
1.1. Service	6	1.4. Monolithische Architektur 8
1.1.1. Modularisierung	6	1.4.1. FTGO Webanwendung 8
1.1.2. Softwareservice	6	1.4.2. FTGO Architektur 8
1.2. Verteilte Systeme	7	1.4.3. Monolithische Architektur: Vorteile 9
1.2.1. Systembegriff	7	1.4.4. Monolithische Architektur: Nachteile 9
1.2.2. Verteiltes System	7	1.5. Servicearchitektur 10
1.2.3. Merkmale verteilter Systeme	7	1.5.1. Softwareservice 10
1.3. Evolution verteilter Systeme	8	1.5.2. SOA Architektur 10
		2. Programmierung: Strukturierung 14
		2.1. Strukturierung von Code 14
		2.1.1. Unterprogramme 14
		2.2. Objektorientierung 15

*E-mail: paul.panhofer@tuwien.ac.at

2.2.1. Objektorientierung	15	5.4.1. Liskovsche Substitutionsprinzip	41
2.3. Schichtenmodell	15	5.4.2. Fallbeispiel: Substitutionsprinzip	41
2.3.1. Prinzipien des Schichtenmodells	15	5.5. Interface Segregation Prinzip	43
2.3.2. Fallbeispiel: Schichtenmodell	16	5.5.1. Interface Segregation Prinzip	43
2.4. Komponenten	17	5.5.2. Fallbeispiel: Interface Segregation	43
2.4.1. Fallbeispiel: Restaurantverwaltung	17		
2.5. Service	18	6. Verteilte Systeme: Eigenschaften	47
3. Programmierung: Metriken	20	6.1. Verteilte Systeme	47
3.1. Softwaremetriken	20	6.1.1. Systembegriff	47
3.1.1. Metriken	20	6.1.2. Verteilte Systeme	47
3.1.2. Qualitätsmetriken	20	6.1.3. Merkmale verteilter Systeme	48
3.2. Koppelung	21	6.1.4. Dezentrale Systeme	48
3.2.1. Koppelung	21	6.2. Eigenschaften verteilter Systeme	49
3.2.2. Interaktionskoppelung	21	6.2.1. Eigenschaft: Transparenz	49
3.2.3. Auflösen von Interaktionskoppelung	22	6.2.2. Eigenschaft: Skalierbarkeit	49
3.2.4. Fallbeispiel: Interaktionskoppelung	22	6.2.3. Bigdata	50
3.2.5. Vererbungskoppelung	23	6.2.4. Skalierbarkeit vs: Performance	50
3.2.6. Fallbeispiel: Vererbungskoppelung	24	6.2.5. Dimensionen der Skalierbarkeit	51
3.2.7. Objektkomposition	24	6.2.6. Eigenschaft: Offenheit	51
3.3. Kohäsion	25	6.2.7. Portabilität	52
3.3.1. Kohäsion	25	6.2.8. Interoperabilität	52
3.3.2. Fallbeispiel: Servicekohäsion	25		
4. Programmierung: Muster	26	7. Verteilte Systeme: Konzepte	53
4.1. Entwurfsmuster	26	7.1. Load Balancing	53
4.1.1. Grundlagen	26	7.1.1. Grundlagen	53
4.1.2. Arten von Pattern	26	7.1.2. Probleme des Load Balancing	53
4.1.3. Einsatz von Entwurfsmustern	27		
4.2. Erzeugungsmuster	27	7.2. Routing	54
4.2.1. Erzeugungsmuster - Singleton	27	7.2.1. Routing	54
4.3. Strukturmuster	28	7.2.2. Z Scaling und Routing	55
4.3.1. Strukturmuster - Adapter	28	7.3. Verschachtelte Transaktionen	55
4.3.2. Strukturmuster - Dekorator	29	7.3.1. Verschachtelte Transaktionen	55
4.3.3. Strukturmuster - Kompositum	29	7.3.2. 2PC- Two Phase Commit	56
4.4. Verhaltensmuster	34		
4.4.1. Strukturmuster - Command	34	8. Verteilte Systeme: Replikation	57
5. Programmierung: SOLID	37	8.1. Replikation	57
5.1. SOLID Prinzipien	37	8.1.1. Grundlagen	57
5.1.1. SOLID Prinzipien	37	8.1.2. Replikation von Daten	57
5.1.2. Kosten schlechten Codes	38		
5.2. Single Responsibility Prinzip	38	9. Kommunikationsprotokoll: Http	59
5.2.1. Single Responsibilty Prinzip	38	9.1. Kommunikationsprotokolle	59
5.2.2. Fallbeispiel: S. Responsibility Prinzip	38	9.1.1. Grundlagen: Syntax	59
5.3. Open Closed Prinzip	39	9.1.2. Grundlagen: Semantik	60
5.3.1. Open Closed Prinzip	39	9.1.3. Grundlage: Synchronisation	60
5.4. Liskovsche Substitutinsprinzip	41	9.2. Netzwerkprotokolle	60
		9.2.1. Grundlagen	60
		9.2.2. Internetprotokolle	60
		9.2.3. Schichten der Protokollstapels	61
		9.3. Http Protokoll	61
		9.3.1. Kommunikationsmuster	61
		9.3.2. Http Nachrichten	61
		9.4. Http Request	62
		9.4.1. Anatomie eines Http Requests	62
		9.4.2. Request Methoden	62

9.5. Http Response	63	12.1.5. Kommunikationsmuster	81
9.5.1. Anatomie eines Http Response	63	12.2. Kommunikationsendpunkte	81
9.5.2. Status Codes	63	12.2.1. Destinationen	81
10. Kommunikationsprotokoll: Rest	65	12.3. MOM Standards	82
10.1. REST Grundprinzipien	65	12.3.1. Protokoll: STOMP	82
10.1.1. Rest Grundprinzipien	65	12.3.2. Protokoll: MQTT	82
10.1.2. Eindeutige Identification von Ressourcen	65	12.3.3. Protokoll: AMQP	83
10.1.3. Repräsentationen von Ressourcen	66	12.3.4. JMS	83
10.1.4. Hypermedia - Hateos	66	12.4. Fallbeispiel: Routenplaner	83
10.1.5. HTTP Methoden	67	12.4.1. Usecasebeschreibung	83
10.2. Ressourcen	68	13. Kommunikationsprotokoll: Websocket	85
10.2.1. Arten von Ressourcen	68	13.1. Websocket Protokoll	85
10.2.2. Primärressourcen	68	13.1.1. Http vs Websockets	85
10.2.3. Subressourcen	68	13.1.2. Kommunikationsablauf	86
10.2.4. Listenressourcen	68	13.1.3. Websocket Protokoll	86
10.3. Http Methoden	69	13.1.4. Zusammenfassung	86
10.3.1. Httpmethode - Get	69	14. Service - Programmierung der Modelenschicht	89
10.3.2. Httpmethode - Put	69	14.1. Struktur eines Services	89
10.3.3. Httpmethode - Post	69	14.1.1. Schichten eines Services	89
10.3.4. Httpmethode - Delete	70	14.2. Programmierung des Models	90
10.3.5. Httpmethode - Patch	70	14.2.1. Prinzipien der objektorientierten Programmierung	91
10.3.6. Httpmethode - Options	70	14.2.2. Prinzip der Objektrelationale Abbildung	91
10.4. Web Api Entwicklung - Fallbeispiel	70	14.3. Persistierung einfacher Objekte	92
Ordermanager	70	14.3.1. Grundlagen der Persistierung	92
10.4.1. Ressourcen einer Anwendung	70	14.3.2. POJO und Annotationen	92
10.4.2. Kategorien von Ressourcen	70	14.3.3. Entität	93
10.4.3. Repräsentationen von Ressourcen	71	14.3.4. Entitätsidentität - Vergleich von Entitäten	94
10.4.4. Analyse einer Repräsentation	73	14.3.5. DRY Prinzip	96
11. Kommunikationsprotokoll: Soap	75	14.3.6. SQL Datentypen	99
11.1. SOAP Webservice Grundlagen	75	14.3.7. Java Datentypen	99
11.1.1. Kommunikation zwischen Servicen	75	14.4. Vererbung von Entitäten	100
11.1.2. Kommunikationsprotokoll	76	14.4.1. Vererbung und Entitäten - Grundlagen	100
11.1.3. Kommunikation zwischen Servicen	76	14.4.2. Single Table Inheritance	100
11.2. WSDL	76	14.4.3. Joined Table Inheritance	101
11.2.1. Aufbau eines WSDL Dokuments	76	14.5. Validieren von Daten	103
11.2.2. WSDL Element - <wsdl:types>	77	14.5.1. Constraints	103
11.2.3. WSDL Element - <wsdl:message>	77	14.5.2. Anwendungsconstraints	105
11.2.4. WSDL Element - <wsdl:portType>	78	14.6. Entitätkomposition	107
11.2.5. WSDL Nachrichtentypen	78	14.6.1. Objektkomposition	107
12. Kommunikationsprotokoll: MOM	79	14.6.2. Objektkomposition - 1:1 Relation	107
12.1. Message Oriented Middleware	79	14.6.3. Objektkomposition - 1:n Relation	108
12.1.1. Grundlagen	79	14.6.4. Objektkomposition - n:m Relation auflösen mit einer Zwischentabelle	109
12.1.2. Messagebroker	80	14.6.5. Bidirektionale Relationen abbilden	110
12.1.3. Nachrichtenaustausch	80	14.7. Auditing von Daten	111
12.1.4. Eigenschaften von MOM Systemen	81		

14.7.1. Auditing Grundlagen	111	16.2.3. Anatomie einer Serverantwort	141
14.7.2. Auditing in Spring	111	16.2.4. Statuscode	143
14.7.3. Historisierungsdiagramm	112	16.2.5. Http Methoden	143
14.7.4. Revisions und Entities	113	16.3. Serviceendpoint	144
15. Service - Domainschicht		16.3.1. Serviceendpoint Aufruf	144
Programmierung	115	16.3.2. Anlegen von Ressourcen	145
15.1. Struktur eines Services	115	16.3.3. Ressourcen verwalten	146
15.1.1. Schichten eines Services	115	16.3.4. Übergabeparameter validieren	147
15.2. Programmierung der Domäne	116	16.4. Json Repräsentation	148
15.2.1. Die JPARepository Schnittstelle	116	16.4.1. JSON Grundlagen	148
15.2.2. Parametrisierung der		16.4.2. JSON - Objekte und Arrays	149
JPARepository Schnittstelle	119	16.4.3. JSON - Strings und Werte	150
15.2.3. Methoden der JPARepository		16.4.4. JSON - Number	151
Schnittstelle	119	16.4.5. Jackson	152
15.2.4. Schnittstellenbeispiel	121	16.4.6. Vererbung	153
15.2.5. JpaRepository erweitern	124	16.4.7. Bidirektionale Relationen mappen	153
15.2.6. JpaRepository - Übergabe von		16.4.8. Annotationen für Datumsformate	155
Queries	124	16.4.9.	155
15.3. JPA Query Language	125	Index	156
15.3.1. JPA SQL Grundlagen	125		
15.3.2. Pfadausdrücke	125		
15.3.3. Select Klausel	126		
15.3.4. From Klausel	126		
15.3.5. Schnittstellenbeispiel	126		
15.4. Fetchgraphen	127		
15.4.1. Objektgraphen	127		
15.4.2. Laden von Objektgraphen	128		
15.4.3. Entitygraphen	129		
15.5. Kaskadierende Operationen	130		
15.5.1. Datenkonsistenz	130		
15.5.2. Kaskadierende Operationen	130		
15.6. Transaktionsverwaltung	131		
15.6.1. Transaktionen - Grundlagen	131		
15.6.2. Problemkontext	131		
15.6.3. ACID Prinzip	131		
15.6.4. Isolationslevel	132		
15.6.5. Optimistic Locking	133		
15.7. Eventhandling	134		
15.7.1. Lebenszyklus einer Entität	134		
15.7.2. Callback Methoden	135		
15.8. Caching	136		
15.8.1. Grundlagen	136		
15.8.2. Arten von Caches	137		
15.8.3. Verwendung von Caches	137		
16. Service - Programmierung der REST			
Serviceschicht	139		
16.1. Struktur eines Services	139		
16.1.1. Schichten eines Services	139		
16.2. Http Protokoll	140		
16.2.1. HTTP Protokoll	140		
16.2.2. Anatomie einer Klientanfrage	141		

Einführung: SOA

Version 2018.09.01

1. Einführung: SOA

01

Service Oriented
Architecture

01. Service	6
02. Verteilte Systeme	7
03. Evolution verteilter Systeme	8
04. Monolithische Architektur	8
05. Service Architektur	10

1.1. Service

Im Mittelpunkt des Fachs SYTD steht die Entwicklung komplexer **Softwareanwendungen**.

Zur Programmierung komplexer **Softwareanwendungen** wird in erster Linie auf die **Modularisierung** von Software gesetzt.

1.1.1 Modularisierung



Modularisierung ▾

Modularisierung IST EIN Strukturierungsverfahren FÜR SOFTWARE.

SOFTWAREANWENDUNGEN WERDEN DABEI IN **Module** UNTERTEILT UM DIE ANWENDUNGEN EINFÄCHER VERSTÄNDLICH UND WARTBAR ZU MACHEN.

Ein **Modul** ist ein Teil der **Anwendung** der unabhängig von den anderen Teilen der Anwendung entwickelt werden kann.

► Erklärung: Aspekte der Modularisierung ▾

- **Köhäsion:** EIN **Modul** ERFÜLLT EINEN SPEZIFISCHEN AUFGABENBEREICH DER ANWENDUNG.
- **Koppelung:** **Module** TAUSCHEN UNTEREINANDER Nachrichten AUS. MODULE SIND VONEINANDER UNABHÄNGIG.

1.1.2 Softwareservice

In der **Softwareentwicklung** spricht man im Kontext eines **Moduls** auch von einem **Softwareservice**.

► Erklärung: Softwareservice ▾

- **Softwareservice** KÖNNEN UNABHÄNGIG VONEINANDER entwickelt UND AUSGEFÜHRT WERDEN.
- EIN **Service** IST FÜR SEINEN EIGENEN **Datenhaushalt** VERANTWORTLICH. JEDES SERVICE VERWALTEN DAMIT SEINE EIGENEN **Ressourcen**.
- DIE **Service** EINER SOFTWAREANWENDUNG KÖNNEN IN UNTERSCHIEDLICHEN **Technologien** IMPLEMENTIERT SEIN. ES GIBT KEINE EINSCHRÄNKUNG AUF EINE BESTIMMTE Programmiersprache ODER Plattform.

Verteilte Systeme



1.2. Verteilte Systeme

1.2.1 Systembegriff



System ▾

Das griechische Wort **Systema** bedeutet ein aus mehreren **Teilen** zusammengesetztes, gegliedertes **Ganzes**.

Ein System ist dabei stets **mehr** als die **Summe** seiner Teile.

► Erklärung: System ▾

- **Systeme** bestehen aus **Elementen**, die untereinander in Beziehung stehen.



1.2.2 Verteiltes System



Verteiltes System ▾

In der **Softwareentwicklung** sprechen wir von einem **verteilten System** wenn eine Softwareanwendung aus mehreren **Servicen** besteht, die untereinander kommunizieren.

Moderne **Softwareanwendungen** werden in der Regel als **Verteilte Systeme** programmiert.

► Erklärung: Verteiltes System ▾

- Ein **verteiltes System** wird in der Regel, verteilt auf mehrere Rechner in einem Netzwerk, ausgeführt.
- Aus der Sicht des **Systementwicklers** macht es dabei keinen Unterschied ob **Service** im **Speicher** eines einzelnen Rechners oder **verteilt** auf mehrere Rechner ausgeführt werden.

- Die wohl bekanntesten **verteilten Systeme** sind **Webanwendungen** wie YouTube, Twitter oder Facebook.
- **Webanwendungen** unterscheiden dabei in der Regel 2 Arten von Elementen: **Clientservice** und ein **Serverservice**.

Die **Clientservice** entsprechen dabei dem im Internetbrowser ausgeführten Programm, während das **Serverservice** die Logik der Webanwendung implementiert.

1.2.3 Merkmale verteilter Systeme

Der **Entwurf** und die Konzeption **verteilter Systeme** erwuchs aus der Notwendigkeit, **Ressourcen** verteilt im **Netzwerk**, nutzbar zu machen.

► Auflistung: Merkmale verteilter Systeme ▾

- **Parallele Verarbeitung:** PARALLELE VERARBEITUNG VON BENUTZER- UND SERVICEANFRAGEN.
- **Shared Access:** GEMEINSAME NUTZUNG VON BETRIEBSMITTELN - z.B.: DATEN BZW. DATEIEN.
- **Modularisierung:** DIE MODULARISIERUNG DER ANWENDUNG IN EINZELNE DIENSTEN ERLAUBT ES DIENSTE IN ANDEREN ANWENDUNGEN WIEDERZUVERWENDEN.
- **Replikation:** SPEICHERUNG VON DATEN AUF MEHRERE KNOTEN IM NETZWERK ZUR ERHÖHUNG DER AUSFALLSICHERHEIT.

1.3. Evolution verteilter Systeme ▾

Aus der Notwendigkeit zunehmend komplexere **Geschäftsprozesse** abilden zu müssen, wurden für den Entwurf verteilter Systeme unterschiedliche Architekturen entwickelt.



1.3.1 Architekturmuster



Architekturmuster ▾

Architekturmuster BESCHREIBEN DIE GRUNDLEGENDE ORGANISATION UND INTERAKTION ZWISCHEN DEN **Modulen** EINER SOFTWAREANWENDUNG.

Für **Verteilte Systeme** wählt man je nach geforderter **Komplexität** eines der folgenden 3 **Architekturmuster**.

► Auflistung: Architekturmuster ▾

- **Monolithische Architektur:** DAS **Monolithische Entwurfsmuster** WIRD EINGESETZT FÜR DIE ENTWICKLUNG EINFACHER WEBANWENDUNGEN.

DIE **Monolithische Architektur** BESCHREIBT DAS ZUSAMMENSPIEL 2ER TYPEN VON MODULEN: EINEM **CLIENTSERVICE** UND EINEM **SERVERSERVICE**.

- **Serverservice:** IM **Serverservice** WIRD DIE **Geschäftslogik** DER WEBANWENDUNG IMPLEMENTIERT. EINE BELIEBIGE ZAHL VON **Clientservicen** KOMMUNIZIEREN MIT DEM SERVERSERVICE.
- **Clientservice:** **Clientservice** WERDEN VOM BENUTZER VERWENDET UM DIE DIENSTE DES SERVERSERVICES IN ANSPRUCH NEHMEN ZU KÖNNEN.

- **SOA - Service Oriented Architecture:** DIE **SOA Architktur** WIRD VERWENDET UM KOMPLEXE GESCHÄFTSPROZESSE IN WEBANWENDUNGEN ABZUBILDEN.

Die Implementierung eines komplexen Geschäftsprozesses lässt sich in der Regel nicht auf 2 Module bzw. Service herunterbrechen. In diesem Fall brauchen wir eine Vielzahl von Servicen die miteinander kommunizieren. SOA als Entwurfsmuster erlaubt die **Komposition** unterschiedlicher Service.

1.4. Monolithische Architektur ▾

Wir wollen die **Architektur** einer **Webanwendung** zur Verwaltung einer Restaurantkette analysieren.



1.4.1 FTGO Webanwendung

Für die Analyse möchten wir die **Food to Go Inc.** Webanwendung betrachten.

► Auflistung: Food to Go Usecases ▾

- **Benutzer** PLATZIEREN Bestellungen ÜBER DIE HOMEPAGE BZW. EINE SMARTPHONE ANWENDUNG IN UNTERSCHIEDLICHEN LOKALEN.
- **Lokalbesitzer** VERWALTEN IHRE **Speisekarte** ÜBER EINEN EIGENEN HOMEPAGE. HIER HABEN SIE AUCH DIE MÖGLICHKEIT DIE BESTELLUNGEN DER BENUTZER EINZUSEHEN.
- DIE ANWENDUNG ERLAUBT ES EINER REIHE VON **Kurierdiensten** DIE BESTELLUNGEN AUSZULIEFERN.
- DIE ANWENDUNG VERWENDET **Webservice** WIE **PayPal** UM DIE ZAHLUNGEN DER KUNDEN ABZUWICKELN BZW. **TWILIO** UM NACHRICHTEN ZU VERSCHICKEN.

1.4.2 FTGO Architektur

Die **FTGO Webanwendung** ist eine gewöhnliche Webanwendung wie wir sie aus dem Internet kennen.

Die Architektur einer Webanwendung wird als **Monolithische Architektur** bezeichnet.

► Architektur: Webanwendung ▾

- DIE **Businesslogik** DER ANWENDUNG IST AUFGETEILT AUF MEHRERE **Komponenten**¹.
- DIE **Kommunikation** DER ANWENDUNG MIT DER AUSSENWELT ERFOLGT ÜBER EINE REIHE VORDEFINIERTER **Schnittstellen**.
- IN DER REGEL WERDEN ANWENDUNGEN DIESER ART ALS EINZELNES **Artefakt** AUSGELIEFERT.

¹ aka Module

Monolithische – Architektur

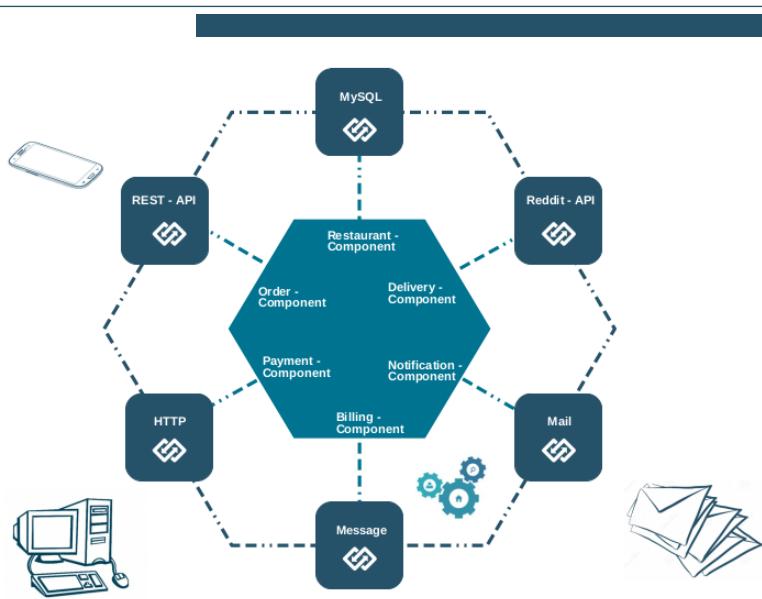


Abbildung 1. Monolithische Architektur

1.4.3 Monolithische Architektur: Vorteile

Einfache **Webanwendungen** werden in der Regel als Monolithen implementiert.

► Vorteile: Monolithische Architektur ▾

- DIE ENTWICKLUNG VON **Webanwendungen** IST EINFACH.
Die Entwicklung von Webanwendungen folgt einer Zahl einfacher Regeln. Solange keine komplexen Geschäftsprozesse abgebildet werden müssen, kann die Implementierung von Webanwendungen einfach gehalten werden.
- IDEs UND ENTWICKLUNGWERKZEUGE SIND OPTIMIERT FÜR DIE ENTWICKLUNG **Monolithischer Architekturen**.
- **Webanwendungen** KÖNNEN EINFACH GETESTET WERDEN.



1.4.4 Monolithische Architektur: Nachteile

Die **Monolithische Architektur** hat jedoch erhebliche Einschränkungen.

► Nachteile: Monolithische Architektur ▾

- **Evolution von Webanwendungen:** ERFOLGREICHE WEBANWENDUNGEN BIETEN DEM BENUTZER MIT DER ZEIT ZUSÄTZLICHE **Geschäftsfelder** AN.

Die **Komplexität** ursprünglich einfacher **Geschäftsprozesse** nimmt mit der Zeit immer mehr zu. Bereits die Dokumentation dieser Geschäftsprozesse stellt einen nicht unerheblichen Aufwand dar.

- **Kodekomplexität:** DESTO GRÖSSER DIE ZAHL AN **Geschäftsprozesse** DIE FÜR EINE ANWENDUNG IMPLEMENTIERT WERDEN MÜSSEN, UMSO STÄRKER STEIGT DIE **Kodekomplexität** DER ANWENDUNG.

Zusätzliche Programmierer müssen angestellt werden um die neuen Geschäftsfelder zu implementieren. Keiner der Programmierer kennt zu diesem Zeitpunkt die Codebasis der Anwendung zur Gänze. Änderungen an der Codebasis sind aufwändig und führen oft zu Fehlern.

- **Verlässlichkeit:** DIE MODULE DER **Geschäftslogik** WERDEN ALLE IM SELBEN **Betriebssystemprozess** AUSGEFÜHRT.

Tritt bei der **Verarbeitung** einer Benutzeranfrage ein Fehler auf kann das zum Absturz der gesamten Anwendung führen. Damit ist es zunehmend herausfordernder die **Zuverlässigkeit** der Anwendung für ihre Anwender zu gewährleisten.

- **Agile Entwicklung:** DIE HOHE **Kodecomplexität** DER ANWENDUNG MACHT EINE SINNVOLE **agile Entwicklung** DER ANWENDUNG SCHWER.

Da die Codebasis als einzelnes **Artefakt** vorliegt wird es mit der Zeit zunehmend herausfordern der der Webanwendung neue Geschäftsfelder hinzufügen.



1.5. Servicearchitektur ▾

1.5.1 Softwareservice



Service ▾

EIN **Service** IST EINE **Softwarekomponente**, DIE IN EINEM EIGENEN **Betriebssystemprozess** AUSGEFÜHRT WIRD.

► Erklärung: Softwareservice ▾

- BEI EINEM **Service** SPRECHEN WIR VON EINEM **Selfcontained System**. EIN **Selfcontained System** ENTHÄLT ALLE ABHÄNGIGKEITEN DIE ES BENÖTIGT UM AUTONOM AUSGEFÜHRT WERDEN ZU KÖNNEN.
- EIN **Service** HAT EINE KLAR DEFINIERTE **Grenze**. DIE SCHNITTSTELLEN DES SERVICES DEFINIEREN DIE GRENZE DES SERVICES.
- DER ZUGRIFF AUF DAS MICROSERVICE IST NUR ÜBER DIE **Schnittstellen** DES MICROSERVICES MÖGLICH.

1.5.2 SOA Architektur



SOA Architekturen ▾

DAS **SOA Entwurfsmuster** BESCHREIBT EIN SYSTEM VON SERVICEN DIE UNTEREINANDER NACHRICHTEN AUSTAUSCHEN.

Die **SOA Architektur** wird verwendet um komplexe Geschäftsprozesse zu implementieren.

► Vorteile: Microservicearchitektur ▾

- **Agile Softwareentwicklung:** DIE ENTWICKLUNG VON SERVIESYSTEME SIND UNTERSTÜTZT DEN EINSATZ ITERATIVER PROJEKTMANAGEMENT MODELLE.

Für die Implementierung komplexer Geschäftsprozesse werden in der Regel große Entwicklerteams benötigt. Durch die Aufteilung der Anwendung in **Micorservices** können die Teams fachlich und technisch unabhängig voneinander arbeiten. Damit wird es möglich große Projekte ohne hohen **Koordinierungsauwand** zu stemmen.

Microservice – Architektur

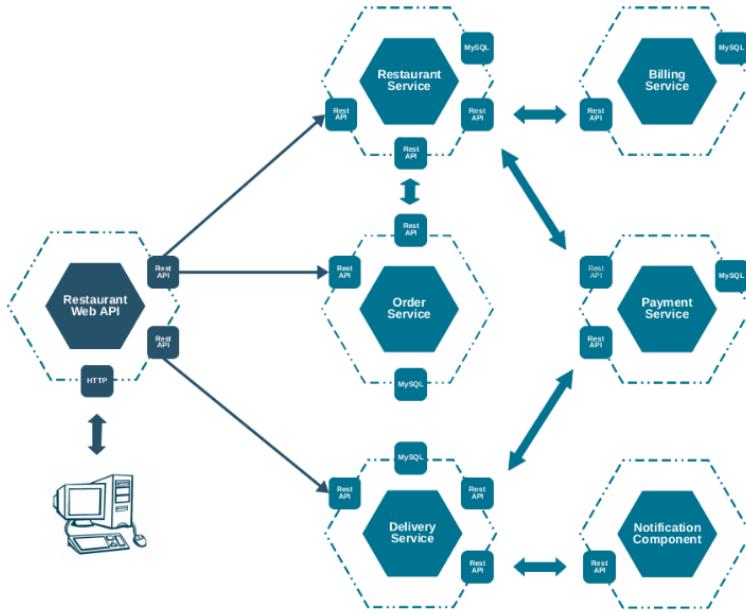


Abbildung 2. Microservice Architektur

- **Hohe Wartbarkeit:** DURCH DEN EINSATZ DER **SOA Architektur** BLEIBEN **Softwareanwendungen** ÜBER JAHRE HINWEG WARTBAR.

Mit seinen Schnittstellen besitzt jedes Service eine klar definierte Grenze. Die hohe **Wartbarkeit** resultiert aus der **Ersetzbarkeit** von Services. Wenn ein einzelnes Service nicht mehr wartbar ist kann es einfach durch einen neuen Service ersetzt werden. Lediglich die Schnittstellen von Servicen müssen unverändert bleiben.

- **Offenes System:** Service KÖNNEN IN BELIEBIGEN **Programmiersprachen** IMPLEMENTIERT WERDEN.

In einer einzelnen **SOA Anwendung** können mit Java programmierte Service neben C# Servicen eingesetzt werden. Diese Vorgehensweise erlaubt es die für eine bestimmte Aufgabe beste Technologie für die Implementierung eines Services zu wählen. Gleichzeitig ist es leichter ein veraltetes System Stück für Stück auf eine neue Technologie umzustellen.

- **Isoliertheit:** IM GRUNDE LASSEN SICH DIE **Eigenschaften** VON SERVICESYSTEMEN AUF DIE STARKE **Entkopplung** DIESER SYSTEME ZURÜCKFÜHREN

Softwareservice werden isoliert voneinander betrieben. Sie sind bezüglich Ausfällen isoliert was die Robustheit dieser Systeme ausmacht. Jedes Service kann isoliert von anderen skaliert werden. Technologien können isoliert für ein Service bestimmt werden was zu Technologiefreiheit führt.

- **Robustheit:** **Servicesysteme** SIND **robust**. WENN IN EINEM MICROSERVICE EIN SPEICHERLEAK AUFTRITT, STÜRZT NUR DIESES SERVICE AB. DIE ANDEREN SERVICE DES SOA SYSTEMS STELLEN DEM BENUTZER IHREN DIENST WEITER ZUR VERFÜGUNG.

- **Unabhängige Skalierbarkeit:** JEDES **Microservice** KANN UNABHÄNGIG VON DEN ANDEREN MICROSERVICEN **skaliert** WERDEN. DAMIT KANN DIE SKALIERBARKEIT EINER SOFTWAREANWENDUNG ZUSÄTZLICH VERBESSERT WERDEN.



Prinzip: Programmierparadigmen

Version 2018.09.01

2. Programmierung: Strukturierung

01

Strukturierung von Code

01. Strukturierung von Code	14
02. Objektorientierung	15
03. Schichtenmodell	15
04. Komponenten	17
05. Service	18

2.1. Strukturierung von Code ▾

Historisch gesehen hat alles mit einem bunten Gemisch aus **Anweisungen** und **Daten** innerhalb eines **Betriebssystemprozesses**² begonnen. Der Prozess spannte die Laufzeitumgebung für den Code auf. Programme waren zu dieser Zeit kurz und einfach.

Die kleinste Einheit eines Programms war die **Anweisung**.

2.1.1 Unterprogramme

Die zunehmende **Codekomplexität** von Softwareanwendungen verlangte nach neuen Wegen **Code** zu strukturieren.

► Erklärung: Unterprogramme ▾

- **Unterprogramme**³ ENTSTANDEN ALS PROGRAMME UMFANGREICHER WURDEN.
- SIE WAREN EIN ERSTER SCHRITT ZUR **Kapselung** VON CODE.
- DIE ZAHL DER ANWEISUNGEN PRO ANWENDUNG KONNTEN ANSTEIGEN, OHNE DASS DIE **Wartbarkeit**⁴ DER ANWENDUNG GESUNKEN WÄRE.
- ALS NÄCHSTES WURDEN **Container für Daten**⁵ ENTWICKELT.

► Codebeispiel: Unterprogramme ▾

```

1 struct Point3D {
2     double x,y,z;
3 };
4
5 main(){
6     settextstyle(BOLD_FONT,HORIZ_DIR,2);
7     outtextxy(220,10,"PIE CHART");
8
9     x = getmaxx()/2;
10    y = getmaxy()/2;
11
12    return 0;
13 }
```



² Unter einem Betriebssystemprozess verstehen wir ein sich in Ausführung befindendes Programm

³ Funktionen, Prozeduren

⁴ Codelesbarkeit, Anpassbarkeit

⁵ Die Sprache C spiegelt diesen Entwicklungsstand wider: sie bietet **Unterprogramme** (Prozeduren und Funktionen) sowie **Strukturen**

2.2. Objektorientierung ▾

2.2.1 Objektorientierung

Der nächste Schritt in der Evolution der **Anwendungsprogrammierung** war das **objektorientierte Programmierparadigma**. **Objektorientierung** faßt Strukturen und Unterprogramme zu **Klassen**⁶ zusammen. Dadurch wurde Software nochmal etwas **grobgranularer**, so dass sich mehrere Anweisungen innerhalb eines Prozesses verwalten ließen.

Die kleinste Einheit eines objektorientierten Programms ist die **Klasse**.

► Erklärung: Klasse ▾

- DIE KLASSE IST DIE **kleinste Einheit** BEI DER IMPLEMENTIERUNG EINES **objektorientierten Systems**.
- EINE **Klasse** STELLT **Funktionalität**⁷ ZUR VERFÜGUNG, DIE DEN **Zustand**⁸ VON INSTANZEN DER KLASSE VERÄNDERT UND VERARBEITET.
- VARIABLEN UND METHODEN BILDEN EINE **Einheit**.



► Codebeispiel: Klassen ▾

```

1  @RequiredArgsConstructor
2  @NoArgsConstructor
3  @Data
4  public class AProject implements Serializable {
5
6      @NotNull
7      @NotBlank
8      @Size(max=50)
9      private String title;
10
11     @NotNull
12     @NotBlank
13     @Size(max=4000)
14     private String description;
15
16 }
```

⁶ Die hauptsächliche **Strukturierung** von Software befindet sich heute auf dem Niveau der **1990er**, als die **Objektorientierung** mit C++, DELPHI und dann JAVA ihren Siegeszug angetreten hat.

⁷ Methoden

⁸ Variablen

2.3. Schichtenmodell ▾

Das **Schichtenmodell** ist ein häufig angewandtes **Strukturierungsprinzip** für die Architektur von Softwaresystemen. Dabei werden einzelne logisch zusammengehörende **Aspekte** des Softwaresystems konzeptionell einer **Schicht** zugeordnet.



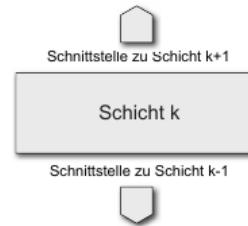
2.3.1 Prinzipien des Schichtenmodells

► Prinzipien: Schichtenmodell ▾

- **Teile und Herrsche:** EIN komplexes Problem WIRD IN **unabhängige Teilprobleme** ZERLEGT, DAS JEDES FÜR SICH, EINFACHER HANDHABBAR IST, ALS DAS GESAMTPROBLEM.
- Oft ist es erst durch die Fromulierung von Teilproblemen möglich, ein KOMPLEXE PROBLEME zu lösen.
- **Unabhängigkeit:** DIE EINZELNEN SCHICHTEN DER ANWENDUNG **kommunizieren** MITEINANDER, INDEM DIE **Schnittstellenspezifikation**⁹ DES DIREKTN VORGÄNGERS BZW. NACHFOLGERS GENUTZT WIRD.

DURCH DIE **Entkoppelung** DER **Spezifikation** DER SCHICHT VON IHRER **Implementierung** WERDEN ABHÄNGIGKEITEN ZWISCHEN DEN SCHICHTEN VERMIEDEN.

- **Abschirmung:** EINE SCHICHT KOMMUNIZIERT AUSCHLIESSLICH MIT SEINEN BENACHBARTEN SCHICHTEN. DAMIT WIRD EINE **Kapselung** DER EINZELNEN SCHICHTEN ERREICHT, WODURCH DIE ZU BEWÄLTIGENDE **Komplexität** sinkt.



- **Standardisierung:** DIE GLIEDERUNG DES GESAMTPROBLEMS IN **einzelne Schichten** ERLEICHTERT DIE ENTWICKLUNG VON **Standards** FÜR DIE EINZELNEN SCHICHTEN.

⁹ Schnittstelle, Interface

2.3.2 Fallbeispiel: Schichtenmodell

► Codebeispiel: Modelschicht ▾

```

1 //-----
2 // Model Schicht
3 //-----
4
5 //Die Modelschicht setzt sich in der Regel aus
6 //Data Transferobjects (DTO) zusammen.
7
8 package at.ac.htl.tcm.model
9
10 public enum EProjectState{
11     SUBMISSION_PHASE, IMPLEMENTATION_PHASE,
12     TERMINATION_PHASE
13 }
14
15 @RequiredArgsConstructor
16 @NoArgsConstructor
17 @Data
18 @Entity
19 @Table(name="PROJECTS")
20 public class AProject implements Serializable{
21
22     @GeneratedValue
23     @Id
24     private Long id;
25
26     @NotNull
27     @NotBlank
28     @Size(max=50)
29     @Column(name="TITLE", length=50,
30             nullable=false, unique=true)
31     private String title;
32
33     @NotNull
34     @NotBlank
35     @Size(max=4000)
36     @Column(name="DESCRIPTION")
37     private String description;
38
39     @NotNull
40     @Past
41     @Temporal(TemporalType.DATE)
42     @Column(name="BEGIN_AT", nullable=false)
43     private Date beginAt;
44
45     @Temporal(TemporalType.DATE)
46     @Column(name="END_AT")
47     private Date endAt;
48
49     @NotNull
50     @Enumerated(EnumType.STRING)
51     @Column(name="PROJECT_STATE", nullable=false)
52     private EProjectState projectState;
53
54 }
```

► Codebeispiel: Domainschicht ▾

```

1 //-----
2 // Domainschicht
3 //-----
4
5 //Die Domainschicht ist verantwortlich fuer die
6 //datenschreibende und datenlesende Logik der
7 //Anwendung
8
9 package at.ac.htl.tcm.service
10
11
12 public class IProjectRepository extends
13     JpaRepository<AProject, Long>{
14
15     Set<AProject>
16         findAProjectsByTitleContains(String
17             titleToken);
18
19 }
```

► Codebeispiel: Serviceschicht ▾

```

1 //-----
2 // Service Schicht
3 //-----
4
5 //Die Serviceschicht ist verantwortlich fuer die
6 //Kommunikation der Service untereinander
7
8 package at.ac.htl.tcm.data
9
10 @RequestMapping(value="projects")
11 @RestController
12 public class PersonResource implements
13     Serializable{
14
15     private static Logger log =
16         LoggerFactory.createLogger(PersonResource.class);
17
18     @Autowired
19     private IProjectRepository projectRepository;
20
21
22     @GetMapping(produces =
23         MediaType.APPLICATION_JSON_UTF8_VALUE)
24     @ResponseStatus(HttpStatus.OK)
25     public Set<AProject> getProjectByTitle(
26         @RequestParam("title") String titleToken){
27
28         Set<AProject> projects = projectRepository.
29             findAProjectsByTitleContains(title);
30
31     }
32 }
```



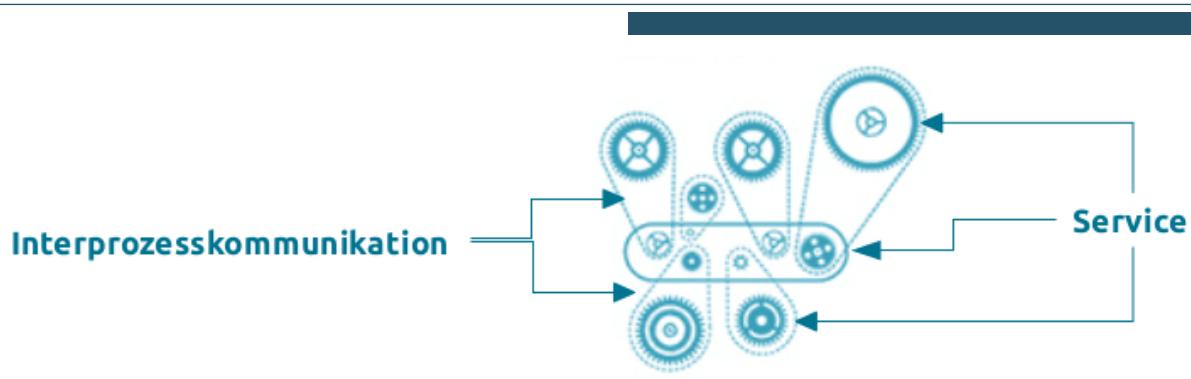


Abbildung 3. SOA - Zusammenspiel von Services

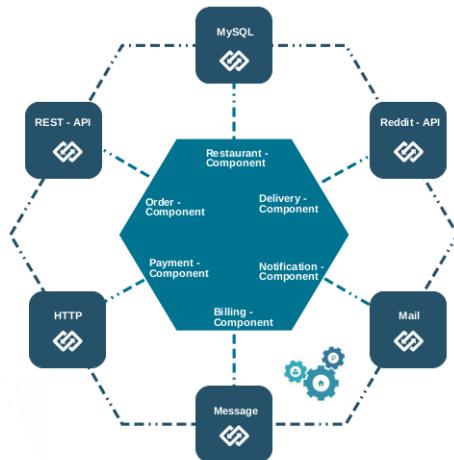
2.4. Komponenten

Bei der Entwicklung von Softwareanwendungen besteht die erste Aufgabe der Softwareentwickler darin, die voneinander unabhängigen Teile der **Anforderungsbeschreibung** voneinander zu isolieren. Wir nennen diese Teile **Komponenten** bzw. Module in der Softwareentwicklung.

Komponenten werden in **Schichten** unterteilt. Jede Schicht wiederum besteht aus **Klassen**.

► Erklärung: Komponente ▼

- **Komponenten** DEFINIEREN SICH ALS VON EINANDER **unabhängige Teile** DER ANFORDERUNGSBESCHREIBUNG EINES SYSTEMS.
- FÜR DIE KOMMUNIKATION STELLEN KOMPONENTEN **Schnittstellen** ZUR VERFÜGUNG.



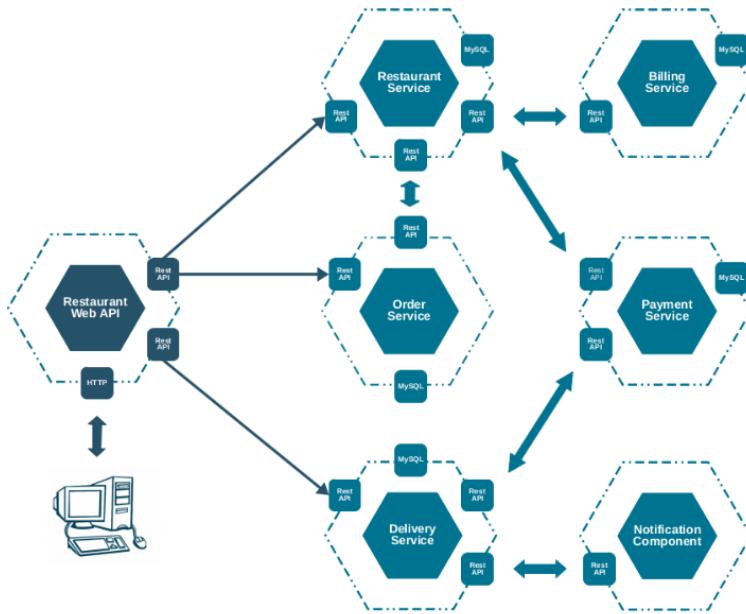
2.4.1 Fallbeispiel: Restaurantverwaltung

► Beispiel: Restaurantverwaltungssoftware ▼

- ES SOLL EINE **Restaurantverwaltungssoftware** ENTWICKELT WERDEN.
- ALS ERSTES **isolieren** WIR DIE EINZELNEN **Komponenten** VONEINANDER.
- **Komponenten** DER **Restaurantverwaltungssoftware**:
 - **Restaurantkomponente:** Lokalbesitzer BENUTZEN DIE **Funktionalität** DER RESTAURANTKOMPONENTE UM DIE SPEISEKARTE FÜR IHRE LOKALE ZU VERARBEITEN.
 - **Orderkomponente:** Benutzer PLATZIEREN **Bestellungen** ÜBER EINE HOMEPAGE BZW. SMARTPHONEANWENDUNG BESTELLUNGEN IN BESTIMMTEN LOKALEN. DIE ORDERKOMPONENTE STELLT DAZU DIE FUNKTIONALITÄT ZUR VERFÜGUNG.
 - **Deliverykomponente:** DIE **Anwendung** ERLAUBT ES EINER REIHE VON KURIERDIENSTEN **Bestellungen** AUSZULIEFERN. DIE DELIVERYKOMPONENTE HILFT BEI DER VERWALTUNG DER BESTELLUNGEN.
 - **Notificationkomponente:** DIE **Anwendung** VERSCHICKT **Benachrichtigungen** AN DIE LOKALE UND KUNDEN. DIE FUNKTIONALITÄT DAFÜR WIRD VON DER NOTIFICATIONKOMPONENTE UMGESETZT.
 - **Billingkomponente:** DIE **Billingkomponente** WIRD EINGESetzt UM DIE ABRECHNUNG DER BESTELLUNG DER KUNDEN DRUCHFÜHREN ZU KÖNNEN.
- DIE EINZELNEN **Komponenten** KÖNNEN NUN UNABHÄNGIG VONEINANDER ENTWICKELT WERDEN.



Microservice – Architektur



2.5. Service



Service ▾

EIN **Service** IST EINE Softwarekomponente DIE IN EINEM EIGENEN **Betriebssystemprozess** AUSGEFÜHRT WIRD.

In einer **SOA** Anwendung bzw. in einer **Microsystemanwendung** ist das **Service** die kleinste **Strukturierungseinheit** der Anwendung.

► Analyse: Service ▾

- KOMPLEXE Softwareanwendungen VERTEILEN IHRE **Geschäftslogik** AUF MEHRERE **Service**.
- EIN **Service** DEFINIERT UNABHÄNGIG VON SEINER IMPLEMENTIERUNG EINE **Schnittstelle**. DER **Zugriff** AUF DAS SERVICE ERFOLGT EXKLUSIV ÜBER DIESE **Schnittstelle**.
- DIE **Servicekommunikation** ERFOLGT ÜBER EIN TECHNIKONOGIE **unabhängiges Protokoll**.
- DIE **Service** EINER Softwareanwendung KÖNNEN IN **unterschiedlichen** TECHNOLOGIEN IMPLEMENTIERT WERDEN DA DIE **Servicekommunikation** ÜBER EIN TECHNOLOGIEUNABHÄNGIGES PROTOKOLL ERFOLGT.

2.5.0 Zusammenfassung

Qualität und Kosten der Erstellung von Softwareanwendungen hängen entscheidend von der **Codekomplexität** ab. **Fehleranzahl** und **Robustheit** eines Codes stehen in engem Zusammenhang zur **Softwarekomplexität**.

Zur Senkung der **Codekomplexität** wurden unterschiedliche Methoden zur **Strukturierung** von Code entwickelt.

► Analyse: **Codestrukturierung** ▾

- Softwareanwendungen BESTEHEN AUS **Services**. EIN SERVICE IST EINE **Softwarekomponente** IN EINEM EIGENEN **Betriebssystemprozess**.
- **Komponenten** BESTEHEN AUS **Schichten**. SCHICHTEN BESTEHEN AUS **Klassen**.
- **Klassen** WERDEN DURCH **Methoden** STRUKTURIERT.



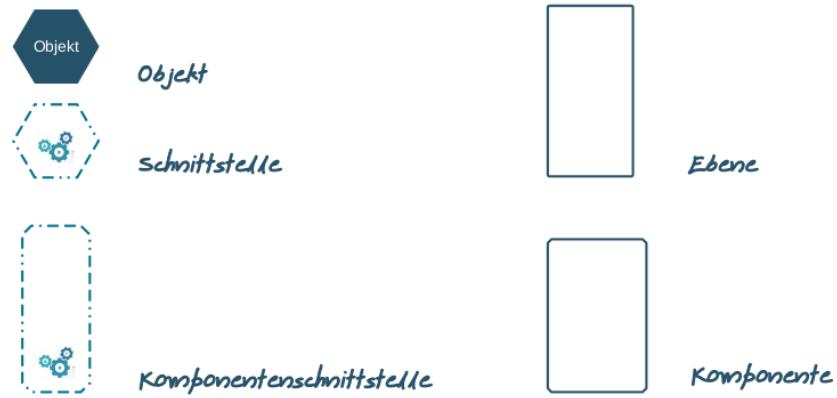
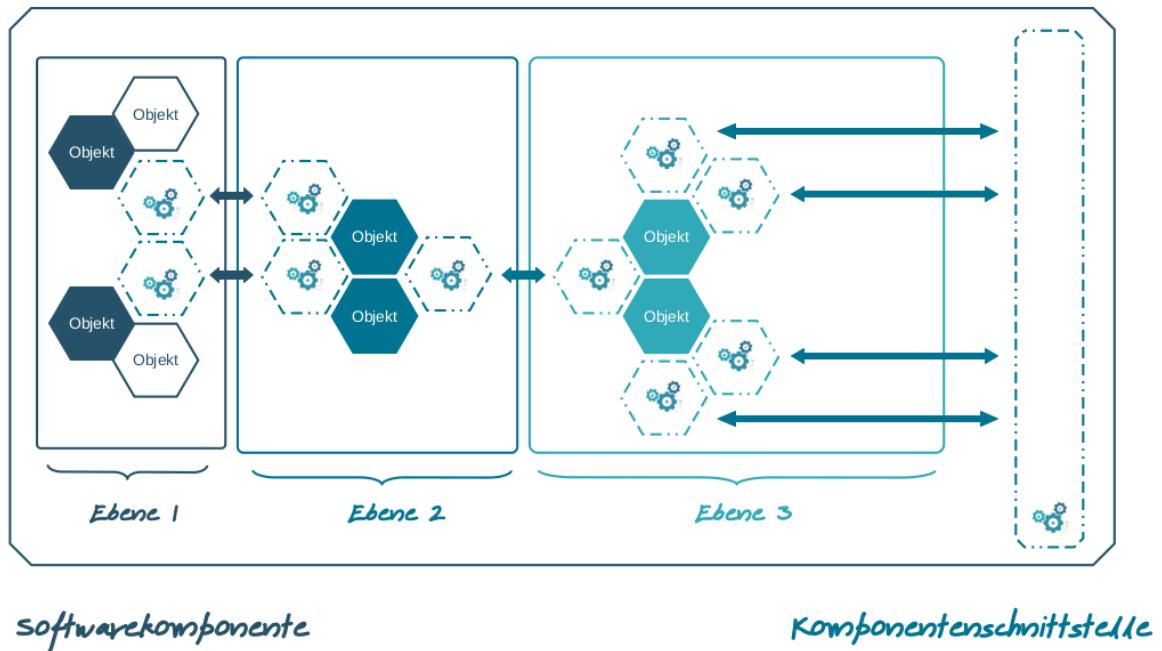


Abbildung 4. Strukturierung einer Komponente

3. Programmierung: Metriken

02

Softwaremetriken

01. Softwaremetriken	20
02. Koppelung	21
03. Kohäsion	25

3.1. Softwaremetriken

3.1.1 Metriken



Softwaremetrik ▾

EINE **Softwaremetrik**, ODER KURZ **Metrik**, IST EINE FUNKTION, DIE EINE EIGENSCHAFT VON SOFTWARE IN EINEN ZAHLENWERT, AUCH **Maßzahl** GENANNT, ABBILDET.

Eine **Softwaremetrik** versucht Programmcode bzw Software im Allgemeinen mit der Hilfe einer **Maßzahl** messbar bzw. vergleichbar zu machen.

► Erklärung: **Softwaremetriken** ▾

- MIT **Softwaremetriken** WIRD PROGRAMMCODE VERGLEICHBAR.
- DABEI KÖNNEN UNTERSCHIEDLICHE **Aspekte von Software** IM VORDERGRUND DER MESSUNG STEHEN: UMFANG, AUFWAND, KOMPLEXITÄT BZW. QUALITÄT.
- DURCH DIE MATHEMATISCHE **Abbildung** EINER SPEZIFISCHEN **Eigenschaft** DER SOFTWARE AUF EINEN ZAHLENWERT WIRD EIN EINFACHER VERGLEICH ZWISCHEN VERSCHIEDENEN TEILEN DER SOFTWARE ERMÖGLICHT.
- DIE **Zeilenmetrik** BESCHREIBT BEISPIELSGEWEISE DEN **Umfang** EINES PROGRAMMS MIT HILFE DER PROGRAMMZEILEN¹⁰ DIE FÜR DIE ERSTELLUNG DES PROGRAMMS NOTWENDIG WAREN.
- WIR WOLLEN UNS HIER JEDOCH AUF METRIKEN BESSCHRÄNKEN DIE DIE **Qualität** DES PROGRAMMCODES MESSEN.



3.1.2 Qualitätsmetriken

Wir unterscheiden 2 **Metriken** zur Beschreibung der **Qualität** von objektorientiertem Code.

► Auflistung: **Softwaremetriken** ▾

- **Koppelung:** MASS DER **Abhängigkeiten** ZWISCHEN **Softwareelementen**.
- **Kohäsion:** MASS DES INNEREN **Zusammenhalt** EINES **Softwareelements**.



¹⁰ Lines of Code

softwaremetriken



3.2. Koppelung

3.2.1 Koppelung



Koppelung ▾

Koppelung IST EIN MASS FÜR DIE Abhängigkeit UNTER Softwareelementen - KLASSEN BZW. KOMPONENTEN. DIESER ABHÄNGIGKEIT ENTSTEHT DURCH DIE NUTZUNG DER FUNKTIONALITÄT DES JEWELLS ANDEREN ELEMENTS.

Beim Entwurf eines **Systems** ist eine **geringe Koppelung** anzustreben.

► Auflistung: Arten der Koppelung ▾

- **Interaktionskoppelung:** **Interaktionskoppelung** BESCHREIBT DAS MASS AN **Funktionalität**¹¹, DAS OBJEKTE EINER KLASSE VON OBJEKten ANDERER KLASSEN IN ANSPRUCH NEHMEN.
- **Vererbungskoppelung:** **Vererbungskoppelung** BE-SCHREIBT DAS AUSMASS DER ABHÄNGIGKEIT ZWISCHEN erbender UND vererbender KLASSE.



3.2.2 Interaktionskoppelung

Interaktionskoppelung beschreibt das Mass an **Funktionalität**, das Objekte einer Klasse von Objekten anderer Klassen in Anspruch nehmen.

Interaktionskoppelung tritt auf wenn Objekte einer Klasse, Methoden von Objekten anderer Klassen aufrufen.

► Codebeispiel: Interaktionskoppelung ▾

```

1 //-----
2 // Interaktionskoppelung
3 //-----
4 public class ISBNGenerator{
5
6     @Inject
7     private UUIDGenerator UUID_GEN;
8
9     @Inject
10    private AtomicInteger INDEX;
11
12    public String generateISBNToken(){
13        StringBuilder tokenBuilder = new
14            StringBuilder();
15
16        tokenBuilder.append(UUID_GEN.nextToken());
17        tokenBuilder.append(INDEX.incrementAndGet());
18
19        return tokenBuilder.toString();
20    }
21
22    public String generateUDDIToken(){
23        StringBuilder tokenBuilder = new
24            StringBuilder();
25
26        tokenBuilder.append(UUID_GEN.nextUUID());
27        tokenBuilder.append(INDEX.incrementAndGet());
28
29        return tokenBuilder.toString();
30    }

```

¹¹ Methodenaufruf

► Codebeispiel: Interaktionskoppelung ▾

```

1 //-----
2 // Book
3 //-----
4 @AllArgsConstructor
5 @RequiredArgsConstructor
6 @Data
7 public class Book implements Serializable{
8
9     private String isbnToken;
10
11    private String uddiToken;
12
13    @NotNull
14    private String title;
15
16    private String description;
17
18    private Author author;
19
20    private Date releasedAt;
21
22    public Book(){
23        isbnToken = ISBNGenerator.generateISBNToken();
24        uddiToken = ISBNGenerator.generateUDDIToken();
25    }
26
27 }
```

3.2.4 Fallbeispiel: Interaktionskoppelung

► Codebeispiel: Entkoppelter Code ▾

```

1 //-----
2 // Entkoppelter Code
3 //-----
4 @AllArgsConstructor
5 @RequiredArgsConstructor
6 @Data
7 public class Book implements Serializable{
8
9     private String isbnToken;
10
11    private String uddiToken;
12
13    @NotNull
14    private String title;
15
16    ...
17
18    public Book(){
19        isbnToken = ISBNGenerator.generateISBNToken();
20        uddiToken = ISBNGenerator.generateUDDIToken();
21    }
22
23 }
24
25 //-----
26 // ISBNGenerator
27 //-----
28 public class ISBNGenerator implements
29     ITokenGenerator{
30
31     @Inject
32     private UUIDGenerator UUID_GEN;
33
34     @Inject
35     private AtomicInteger INDEX;
36
37     public String generateISBNToken(){
38         StringBuilder tokenBuilder = new
39             StringBuilder();
40
41         tokenBuilder.append(UUID_GEN.nextToken());
42         tokenBuilder.append(INDEX.incrementAndGet());
43
44         return tokenBuilder.toString();
45     }
46
47     public String generateUDDIToken(){
48         StringBuilder tokenBuilder = new
49             StringBuilder();
50
51         tokenBuilder.append(UUID_GEN.nextUUID());
52         tokenBuilder.append(INDEX.incrementAndGet());
53
54     }
55 }
```

3.2.3 Auflösen von Interaktionskoppelung

Durch die **Trennung** von **Definition** und **Implementierung** kann die Implementierung einer Klasse verändert werden ohne dass andere Klassen davon betroffen werden.

► Analyse: Interaktionskoppelung ▾

- **OBJEKTE DER KLASSE BOOK UND ISBNNUMBER SIND UNTEREINANDER GEKOPPELT.**
- **BOOK OBJEKTE VERWENDEN ISBNNUMBER OBJEKTE ZUR STRING DARSTELLUNG.**
- **GIBT ES EINE ÄNDERUNG IM CODE DER ISBNNUMBER KLASSE MUSS MIT HOHER WAHRSCHEINLICHKEIT AUCH DER CODE DER BOOK KLASSE VERÄNDERT WERDEN.**
- **Koppelung zwischen Objekten wird durch die Definition von Schnittstellen vermieden.**
- **Mit einer Schnittstelle wird die **Definition** einer Klasse von ihrer **Implementierung** getrennt.**

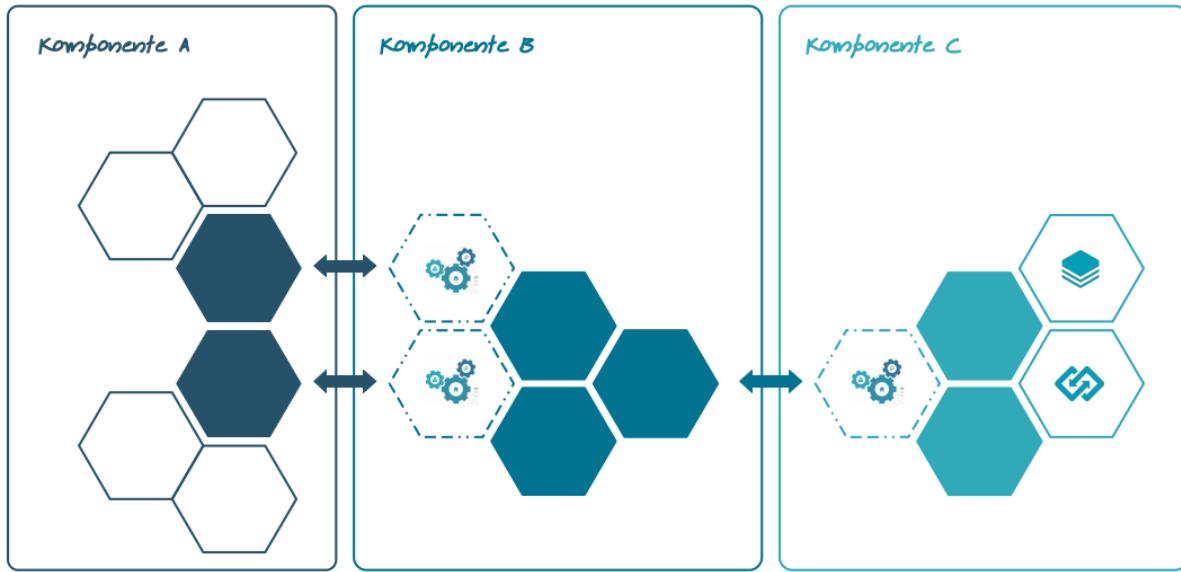


Abbildung 5. Kommunikation: Interface

▶ Codebeispiel: Entkoppelter Code ▾

```

1 //-----
2 // Interaktionskoppelung - Schnittstelle
3 //-----
4 public interface ITokenGenerator{
5
6     String generateISBNToken();
7
8     String generateUDDIToken();
9
10 }
11
12 public class BookStore{
13
14     public static final void main(String args[]){
15         Book book = new Book();
16     }
17
18 }
```



3.2.5 Vererbungskoppelung



Vererbungskoppelung ▾

Vererbungskoppelung BESCHREIBT DAS Ausmaß DER ABHÄNGIGKEIT ZWISCHEN erbender UND vererbender KLASSE.

Vererbungskoppelung tritt auf wenn ein Objekt von einem anderen Objekt erbt.

▶ Erklärung: Vererbungskoppelung ▾

- **Vererbung** IST EINES DER FUNDAMENTALEN Prinzipien DER Objektorientierten PROGRAMMIERUNG.
- **Vererbung** ERLAUBT DAS Verhalten DER BASISKLASSE AUF KINDKLASSEN ZU ÜBERTRAGEN.
- DER EINSATZ VON **Vererbung** KANN JEDOCH ZU KOMPLEXEN **Vererbungsstrukturen** FÜHREN.
WIRD ES NOTWENDIG, DIE VON DER **Basisklasse** GEERBTEN METHODEN, IN KINDKLASSEN ZUR GÄNZE ZU ÜBERSCHREIBEN VERLIERT VERERBUNG SEINEN SINN. IN DIESEM FALL SPRICHT MAN VON **Vererbungskoppelung**.
- **Vererbungskoppelung** KANN MIT HILFE VON **Objekt-komposition** AUFGELÖST WERDEN.



3.2.6 Fallbeispiel: Vererbungskoppelung

► Codebeispiel: Vererbungskoppelung ▾

```

1 //-----
2 // Vererbungskoppelung
3 //-----
4 public class Duck{
5
6     public String quack(){
7         return "quack";
8     }
9
10    public String fly(){
11        return "flying high in the sky";
12    }
13
14 }
15
16 public class RedheadDuck extends Duck{
17
18     public String quack(){
19         return "loudly: " + super.quack();
20     }
21
22 }
23
24 public class EntlingDuck extends RedheadDuck{
25
26     public String quack(){
27         return "proudly and " + super.quack();
28     }
29
30 }
31
32 public class RubberDuck extends Duck{
33
34     public String quack(){
35         return "squeeze";
36     }
37
38     public String fly(){
39         return "can't fly";
40     }
41 }
42 }
```

Zur Auflösung der Vererbungskoppelung setzen wir auf **Objektkomposition** statt Vererbung.

► Erklärung: Vorteile der Objektkomposition ▾

- DER VORTEIL DER **OBJEKTKOMPOSITION** GEGENÜBER DER **OBJEKTVERERBUNG** LIEGT IN DER **CODEFLEXIBILITÄT**.
- MIT **OBJEKTKOMPOSITION** KANN DAS VERHALTEN VON **OBJEKten** ZUR LAUFZEIT VERÄNDERT WERDEN.



► Codebeispiel: Objektkomposition ▾

```

1 //-----
2 // Objektkomposition vs. Vererbungskoppelung
3 //-----
4 public interface IQuackable{
5     String quack();
6 }
7
8 public interface IFlyable{
9     String fly();
10 }
11
12 public class DefaultQuackBehaviour implements
13     IQuackable{
14
15     public String quack(){
16         return "quack";
17     }
18 }
19
20 //-----
21 // Objektkomposition
22 //-----
23
24 public class LoudQuackBehaviour implements
25     IQuackable{
26
27     public String quack(){
28         return "loudly: quack";
29     }
30 }
31
32 public class ProudQuackBehaviour implements
33     IQuackable{
```

3.2.7 Objektkomposition



Objektkomposition ▾

Objektkomposition basiert auf der Technik, **Objekte bestehender Klassen** in andere Klassen **einzubetten**. (z.B. durch Aggregation oder Referenz).

```

33     public String quack(){
34         return "proudly and loudly: quack";
35     }
36 }
37
38 public SqueezeQuackBehaviour implements IQuackable{
39     public String quack(){
40         return "squeeze";
41     }
42 }
43
44 }
45
46 public class DefaultFlyingBehaviour implements
47     IFlyable{
48
49     public String fly(){
50         return "flying high in the sky";
51     }
52 }
53
54 public class NoFlyBehaviour implements IFlyable{
55
56     public String fly(){
57         return "can't fly";
58     }
59 }
60
61 }
62
63 @NoArgsConstructor
64 @AllArgsConstructor
65 @Data
66 public class Duck implements Serializable{
67     private IQuackable quackBehaviour;
68
69     private IFlyable flyBehaviour;
70 }
71
72 public class DuckSimulator{
73     public static final void main(String args[]){
74         Duck redheadDuck = new Duck(
75             new LoudQuackBehaviour(),
76             new DefaultFlyingBehaviour()
77         );
78
79         Duck EntlingDuck = new Duck(
80             new ProudQuackBehaviour(),
81             new DefaultFlyingBehaviour()
82         );
83
84         Duck rubberDuck = new Duck(
85             new SqueezeQuackBehaviour(),
86             new NoFlyBehaviour()
87         );
88     }
89 }

```

3.3. Kohäsion

3.3.1 Kohäsion



Kohäsion ▾

Kohäsion IST EIN MASS FÜR DEN inneren Zusammenhalt EINES Softwareelements - z.B.: KLASSE BZW. SOFTWAREKOMPONENTE.

Beim Entwurf eines Systems ist eine **hohe Kohäsion** anzustreben.

► Erklärung: Kohäsion ▾

- WIRD DURCH EIN ELEMENT **zuviel Funktionalität** umgesetzt, IST DAS ELEMENT ZU GENERELL - DIE KOHÄSION NIMMT AB.
- DAS SELBE GILT FÜR EIN ELEMENT DAS ZUWENIG FUNKTIONALITÄT IMPLEMENTIERT.
- HOHE KOHÄSION BEGÜNSTIGT GERINGE **Koppelung**.

► Auflistung: Arten der Kohäsion ▾

- **Servicekohäsion:** Methoden EINER KLASSE SOLLTEN NICHT ZU WENIG, ABER AUCH NICHT ZU VIEL **Funktionalität** IMPLEMENTIEREN.
- **Klassenkohäsion:** KLASSEN SOLLTEN KEINE UNGENUTZEN ATTRIBUTE BZW. METHODEN DEFINIEREN.

3.3.2 Fallbeispiel: Servicekohäsion

► Codebeispiel: Servicekohäsion ▾

```

1  //-----
2 // Servicekohäsion - schwache Kohsion
3 //-----
4 @NoArgsConstructor
5 @AllArgsConstructor
6 @Data
7 class Vector implements Serializable{
8
9     private int x, y, z;
10
11    public float add(Vector v){
12        this.x += v.x;
13        this.y += v.y;
14        this.z += v.z;
15
16        return Math.SQRT(x * x + y * y + z * z);
17    }
18
19 }

```

4. Programmierung: Muster

03

Entwurfsmuster

01. Entwurfsmuster	26
02. Erzeugungsmuster	27
03. Strukturmuster	28
04. Verhaltensmuster	34

4.1. Entwurfsmuster

4.1.1 Grundlagen



Entwurfsmuster ▾

EIN **Entwurfsmuster** BESCHREIBT EIN BESTIMMTES, IMMER WIEDERKEHRENDES **Entwurfsproblem** IN DER SOFTWAREENTWICKLUNG, SOWIE EIN SCHEMA ZU SEINER **Lösung**.

► Erklärung: Entwurfsmuster ▾

- **Entwurfsmuster** HELFEN, EINE ADEQUATE **Lösung** FÜR EIN WIEDERKEHRENDES **Entwurfsproblem** ZU FINDEN.
- **Entwurfsmuster** STÜTZEN SICH DABEI AUF DIE ERFAHRUNGEN DER **Softwarecommunity**.
- **Entwurfsmuster** ETABLIEREN GLEICHZEITIG EINE EIGENE **Terminologie** FÜR DIE SOFTWAREENTWICKLUNG.

4.1.2 Arten von Pattern

Je nach **Einsatzgebiet** werden unterschiedliche Kategorien von **Entwurfsmustern** definiert.

► Auflistung: Arten von Entwurfsmustern ▾



Idiom ▾

Idiome SIND **Entwurfsmuster** DIE TEIL DER EI- GENTLICHEN PROGRAMMIERSPRACHE SIND.

ANNOTATIONEN SIND ZUM BEISPIEL EIN IDIOM DAS NUR VON BESTIMMTEN PROGRAMMIERSPRACHEN UN- TERSTÜTZT WIRD.



Entwurfsmuster ▾

Entwurfsmuster BESCHREIBEN DAS ZUSAMMEN- SPIEL VON **Klassen**.



Architekturmuster ▾

Architekturmuster BESCHREIBEN DAS ZUSAMMEN- SPIEL VON **Komponenten**.

Entwurfsmuster



4.1.3 Einsatz von Entwurfsmustern

► Analyse: Motivation ▾

- **Entwurfsmuster** VERBESSERN DIE **Codestruktur** UND DAMIT DIE **Codequalität** INNERHALB VON KOMPONENTEN.
- MUSTER REPRÄSENTIEREN WESENTLICHE **Konzepte** DER SOFTWAREENTWICKLUNG UND BRINGEN SIE IN EINE VERSTÄNDLICHE FORM. MUSTER HELFEN IN DIESEM SINNE ENTWÜRFE ZU **verstehen** UND SIE ZU **dokumentieren**.
- MUSTER VERBESSERN DIE **Kommunikation** IM TEAM. **Entwurfsmuster** BILDEN EINE NÜTZLICHE TERMINOLOGIE FÜR DIE **Kommunikation** UNTER ENTWICKLERN.
- MUSTER **dokumentieren** UND FÖRDERN DEN STAND DER TECHNIK. DER EINSATZ VON **Entwurfsmuster** HILFT DAS SUCHEN VON LÖSUNGEN FÜR PROBLEME, DER LÖSUNG BEKANNT IST, ZU VERMEIDEN.

► Erklärung: Zielsetzung ▾

- DER EINSATZ VON **Entwurfsmustern** FÜHRT ZU ANWENDUNGSCODE MIT STARKER **Kohäsion** UND SCHWACHER **Koppelung**.
- DER EINSATZ VON **Entwurfsmustern** ETABLIERT DIE **SOLID** PRINZIPIEN FÜR EINE GEGEBENE SOFTWAREANWENDUNG.
- **Entwurfsmuster** WERDEN JE NACH IHREM EINSATZGEBIET IN UNTERSCHIEDLICHE KATEGORIEN EINGETEILT.
- FÜR **Entwurfsmuster** HABEN WIR DAMIT EINE DIFFERENZIERUNG IN ERZEUGUNGS-, STRUKTUR- UND VERHALTENSMUSTER.



4.2. Erzeugungsmuster

Erzeugungsmuster unterstützen das **Erzeugen** von komplexen Objekten bzw. helfen bei der Kapselung des **Erzeugungsprozesses** von Objekten.



4.2.1 Erzeugungsmuster - Singleton



Singleton ▾

DAS **Singleton** ENTWURFSMUSTER HILFT SICHERZUSTELLEN, DASS FÜR EINE BESTIMMTE KLASSE NUR EINE KONKRETE **Instanz** ERZEUGT WERDEN KANN.

DER **Zugriff** AUF DAS **SINGELTON** MUSS GLOBAL MÖGLICH SEIN.

► Erklärung: Motivation und Kontext ▾

- INNERHALB EINER SOFTWAREANWENDUNG SOLL ES Z.B.: NUR EINE INSTANZ EINES DRUCKERSPOOLERS GEBEN KÖNNEN.
- ES SOLL ALSO NUR EINE **Instanz** EINER BESTIMMTEN KLASSE IN DER ANWENDUNG EXISTIEREN.
- DIE KLASSE SOLL DURCH **Vererbung** ERWEITERBAR SEIN.

► Erklärung: Eigenschaften eines Singletons ▾

- SINGLETONS ERLAUBEN EINEN KONTROLLIERTEN **Zugriff** AUF SICH SELBST.
- SINGELTON UNTERSTÜTZEN **Vererbung**.

► Codebeispiel: Singleton ▾

```

1 //-----
2 // Entwurfsmuster: Singleton
3 //-----
4 public class ProjectContext{
5     private static final ProjectContext instance =
6         new ProjectContext();
7
8     private ProjectContext(){}
9
10    public static final ProjectContext
11        getInstance(){
12        return instance;
13    }
14}

```

► Codebeispiel: Entwurfsmuster: Singleton ▾

```

1 //-----
2 // Entwurfsmuster: Singleton
3 //-----
4 public class DefaultProjectContext extends
5     ProjectContext{
6
7     private static final DefaultProjectContext
8         instance = new
9         DefaultProjectContext();
10
11    private DefaultProjectContext(){}
12    ...
13
14    public static final DefaultProjectContext
15        getInstance(){
16        return instance;
17    }
18
19    public class DefaultProjectContext extends
20        ProjectContext{
21
22        private static final DefaultProjectContext
23            instance = new
24            DefaultProjectContext();
25
26        private DefaultProjectContext(){}
27        ...
28
29        public static final DefaultProjectContext
30            getInstance(){
31            return instance;
32        }
33}

```

4.3. Strukturmuster ▾

Strukturmuster beschreiben die **Struktur** komplexer Objekte.



4.3.1 Strukturmuster - Adapter ▾



Adapter ▾

Mit dem Einsatz eines **Adapters** kann ein **Interface** in ein anderes **umgewandelt** werden.

► Erklärung: Motivation und Kontext ▾

- WIR MÖCHTEN IN EIN BESTEHENDES **Softwaresystem**, DIE **Klassen** EINER EXTERNEN KLASSENBIOTHEK INTEGRIEREN. DIE SCHNITTSTELLE BEIDER SYSTEME SIND JEDOCH NICHT KOMPATIBEL.
- DABEI IST ZU BEACHTEN DASS DIE **Schnittstellen** DES BESTEHENDEN SOFTWARESYSTEMS NICHT VERÄNDERT WERDEN DÜRFEN.

► Erklärung: Eigenschaften eines Adapters ▾

- DER **Adapter** FUNGIERT ALS **Vermittler**, DER ANFRAGEN VOM CLIENT ERHÄLT UND DIESE IN ANFRAGEN UMWANDELNT, DIE DIE NEUEN KLASSEN VERSTEHEN.
- EIN **Adapter** ERMÖGLICHT ES KLASSEN MIT INKOMPATIBLEN SCHNITTSTELLEN IN EINE SOFTWAREANWENDUNG ZU INTEGRIEREN.

► Codebeispiel: Entwurfsmuster: Adapter ▾

```

1 //-----
2 // Entwurfsmuster: Adapter
3 //-----
4 public interface IQuackable{
5
6     void quack();
7
8 }
9
10 public class RedheadDuck implements IQuackable{
11
12     public void quack(){
13         System.out.println("quack");
14     }
15
16 }

```



► Codebeispiel: Entwurfsmuster: Adapter ▾

```

1 //-----
2 // Entwurfsmuster: Adapter
3 //-----
4 public class MullardDuck implements IQuackable{
5
6     public void quack(){
7         System.out.println("quack quack");
8     }
9
10 }
11
12 public class RubberDuck implements IQuackable{
13
14     public void quack(){
15         System.out.println("squeeze");
16     }
17
18 }
19
20 //-----
21 // Interface: Hissable
22 //-----
23 public interface IHissable{
24
25     void hiss();
26 }
27
28 public class Goose implements IHissable{
29
30     public void hiss(){
31         System.out.println("hiss");
32     }
33
34 }
35 }
```

► Codebeispiel: Entwurfsmuster: Adapter ▾

```

1 //-----
2 // Entwurfsmuster: Adapter
3 //-----
4 @AllArgsConstructor
5 public class HissableAdapter implements IQuackable{
6
7     private IHissable hissable;
8
9     public void quack(){
10         hissable.hiss();
11     }
12
13 }
14
15 public class DuckSimulator{
16
17     public void simulate(List<IQuackable>
18                         quackable){
19         quackables.stream().forEach(IQuackable::quack);
20     }
21
22 }
23
24 public class SimulatorLaunch{
25
26     public static final main(String args[]){
27         List<IQuackable> quacks = Arrays.asList(
28             new ReadHeadDuck(),
29             new MullardDuck(),
30             new RubberDuck(),
31             new HissableAdapter(new Goose()));
32
33         DuckSimulator sim = new DuckSimulator();
34         sim.simulate(quacks);
35     }
36 }
```

4.3.2 Strukturmuster - Dekorator



Dekorator ▾

DAS DEKORATORMUSTER HILFT **Objekte** DYNAMISCH UM **Verhalten** ZU ERWEITERN.

► Erklärung: Motivation und Kontext ▾

- OFT IST ES NOTWENDIG **Objekte** DYNAMISCH UM **Verhalten** ZU ERWEITERN.
- **Vererbung** KANN IN DIESEM ZUSAMMENHANG NUR BE-DINGT EINGESETZT WERDEN.

► Erklärung: Eigenschaften von Dekoratoren ▾

- **Dekorierer** HABEN DEN GLEICHEN **Supertyp**, WIE DIE **OBJEKTE**, DIE SIE DEKORIEREN.

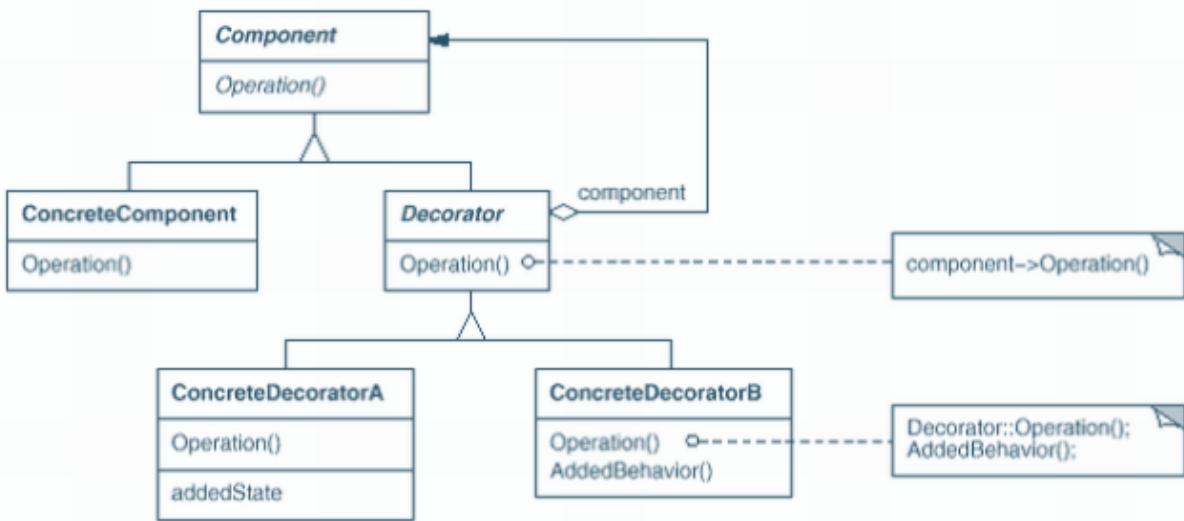


Abbildung 6. Klassendiagramm - Dekorator

- DAMIT KANN DER DEKORIERER **stellvertretend** FÜR DAS 26 }
ZU DEKORIERENDE OBJEKT VERWENDET WERDEN. 27
28 }
- DER DEKORIERER FÜGT ZUR LAUFZEIT SEIN **Verhalten**
DEM ZU DEKORIERENDEN OBJEKT HINZU.

► Codebeispiel: Entwurfsmuster: Dekorator ▾

```

1 //-----
2 // Entwurfsmuster: Dekorator
3 //-----
4 public interface IIngredient{
5     String getDescription();
6     Integer getPrize();
7 }
8
9 @NoArgsConstructor
10 @RequiredArgsConstructor
11 public class TomatoDecorator implements
12     IIngredient{
13     @NonNull
14     private IIngredient ingredient;
15
16     private String description = "Tomatosauce";
17
18     private Integer prize = 1;
19
20     public String getDescription(){
21         return String.format("%s %s",
22             ingredient.getDescription(),
23             description);
24     }
25
26     public Integer getPrize(){
27         return ingredient.getPrize() + prize;
28     }
  
```

▶ Codebeispiel: Entwurfsmuster: Dekorator ▼

```

1 //-----
2 // Dekoratoren
3 //-----
4 @NoArgsConstructor
5 @RequiredArgsConstructor
6 public class SausageDecorator implements
7     IIngredient{
8
9     @NotNull
10    private IIngredient ingredient;
11
12    private String description = "Sausage";
13
14    private Integer prize = 2;
15
16    public String getDescription(){
17        return String.format("%s %s",
18            ingredient.getDescription(),
19            description);
20    }
21
22    public Integer getPrize(){
23        return ingredient.getPrize() + prize;
24    }
25
26    @NoArgsConstructor
27    @RequiredArgsConstructor
28    public class OnionDecorator implements IIngredient{
29
30        @NotNull
31        private IIngredient ingredient;
32
33        private String description = "Onion";
34
35        private Integer prize = 1;
36
37        public String getDescription(){
38            return String.format("%s %s",
39                ingredient.getDescription(),
40                description);
41
42        public Integer getPrize(){
43            return ingredient.getPrize() + prize;
44    }
45
46    @NoArgsConstructor
47    @AllArgsConstructor
48    @Data
49    public class Pizza implements IIngredient{
50
51        private String description = "Pizza";
52
53        private Integer prize = 3;
54
55    }
56 }
```

▶ Codebeispiel: Entwurfsmuster: Dekorator ▼

```

1 //-----
2 // Dekoratoren
3 //-----
4 @NoArgsConstructor
5 @RequiredArgsConstructor
6 public class TunaDecorator implements IIngredient{
7
8    @NotNull
9    private IIngredient ingredient;
10
11    private String description = "Tuna";
12
13    private Integer prize = 2;
14
15    public String getDescription(){
16        return String.format("%s %s",
17            ingredient.getDescription(),
18            description);
19
20    public Integer getPrize(){
21        return ingredient.getPrize() + prize;
22    }
23 }
```



► Codebeispiel: Entwurfsmuster: Dekorator ▾

```

1 //-----
2 // Ausfuerung
3 //-----
4 public class DishSimulator{
5
6     public static void main(String[] args){
7         IIIngredient tonno = new OnionDecorator(
8             new TunaDecorator(
9                 new ChesseDecorator(
10                new TomatoDecorator(
11                    new OliveDecorator(
12                        new Pizza()))));
13
14        IIIngredient margarita = new TomatoDecorator(
15            new CheeseDecorator(
16                new Pizza())));
17
18        IIIngredient salami = new SausageDecorator(
19            new CheeseDecorator(
20                new OnionDecorator(
21                    new TomatoDecorator(
22                        new Pizza()))));
23
24        IIIngredient calcone = new HamDecorator(
25            new EggDecorator(
26                new OnionDecorator(
27                    new TomatorDecorator(
28                        new Pizza()))));
29    }
30
31 }
```

4.3.3 Strukturmuster - Kompositum



Kompositum ▾

DAS **Composite Muster** HILFT OBJEKTE ZU EINER Baumstruktur ZUSAMMENZUSETZEN.

DAS MUSTER ERLAUBT ES DEM CLIENT, **individuelle Objekte** GENAUZO ZU BEHANDELN WIE EINE MENGE VON OBJEKten.

► Erklärung: Motivation und Kontext ▾

- ES WIRD EINE **Struktur** BENÖTIGT DIE IHRE ELEMENTE IN FORM EINER BAUMSTRUKTUR AUFNEHMEN KANN.
- ES IST NOTWENDIG, INDIVIDUELLE OBJEKTE UND GRUPPEN VON OBJEKten AUF GLEICHE WEISE ZU BEHANDELN.
- EIN TYPISCHES BEISPIEL FÜR EIN **Kompositum** SIND z.B.: HIERARCHISCHE **Dateisysteme**, INSbesondere IHRE REPRÄSENTATION INNERHALB VON **Dateibrowsern**.
- INNERHALB DES DATEIBROWSERS WERDEN DATEIEN UND ORDNER AUF DIESELBE WEISE BEHANDELT.
- ES GIBT 2 ARTEN VON **Elementen**: DATEIEN UND ORDNER. **Ordner** SIND ELEMENTE, DIE ANDERE ELEMENTE ENTHALTEN, DATEIEN SIND INDIVIDUELLE **Objekte**.

► Erklärung: Eigenschaften der Komponenten ▾

- **Komponente:** DIE **Komponente** DEFINIERT ALS **Basisklasse** DAS GEMEINSAME VERHALTEN ALLER TEILNEHMER.
- **Blatt:** EIN **Blatt** REPRÄSENTIERT EIN INDIVIDUELLES **Objekt**, ES BESITZT KEINE KINDOBJEKTE.
- **Kompositum:** DAS **Kompositum** ENTHÄLT KOMPONENTEN, ALSO WEITERE KOMPOSITA BZW. BLÄTTER, ALS **Kindobjekte**.

► Codebeispiel: Entwurfsmuster: Kompositum ▾

```

1 //-----
2 // Entwurfsmuster: Kompositum
3 //-----
4 public interface IGraphic{
5
6     public void print();
7
8 }
9
10 public class CompositeGraphic implements IGraphic{
11
12     private List<IGraphic> childList = new
13         ArrayList<>();
```

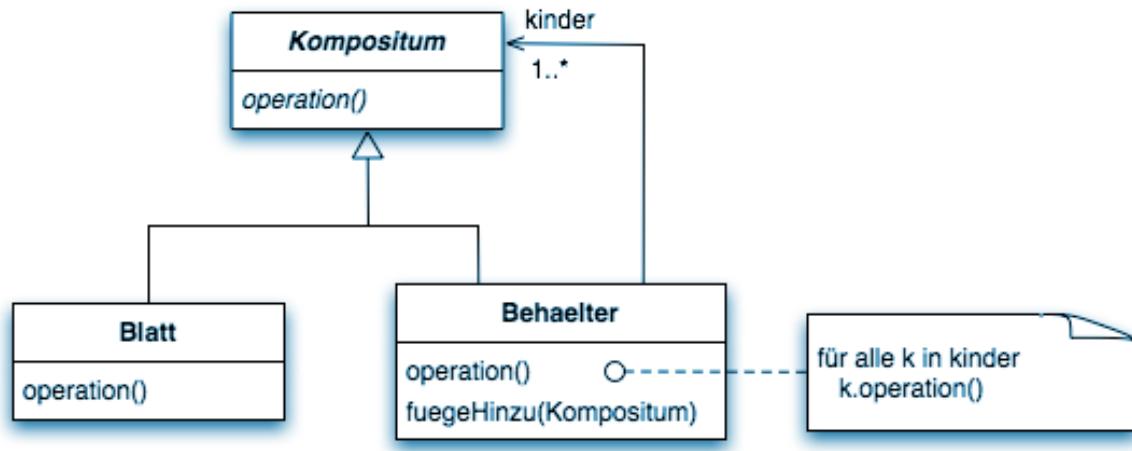


Abbildung 7. Entwurfsmuster - Komposit

```

13
14     public void print(){
15         childList.forEach(
16             (IGraphic graphic) -> {
17                 graphic.print();
18             }
19         );
20     }
21
22     public void add(IGraphic graphic){
23         childList.add(graphic);
24     }
25
26     public void remove(IGraphic graphic){
27         childList.remove(graphic);
28     }
29
30 }
31
32 public class Triangle implements IGraphic{
33
34     public void print(){
35         System.out.println("triangle");
36     }
37
38 }
39
40 public class Circle implements IGraphic{
41
42     public void print(){
43         System.out.println("circle");
44     }
45
46 }
47
48 public class Ellipse implements IGraphic{
49
50     public void print(){
51         System.out.println("ellipse");
52     }
53
54 }
55
56
57 public class Quader implements IGraphic{
58
59     public void print(){
60         System.out.println("quader");
61     }
62
63 }
64
65 public class Octaeder implements IGraphic{
66
67     public void print(){
68         System.out.println("octaeder");
69     }
70
71 }
  
```

► Codebeispiel: Entwurfsmuster: Kompositum ▾

```

1 //-----
2 // Entwurfsmuster: Kompositum
3 //-----
4 public class Program{
5
6     public static void main(String[] args){
7         CompositeGraphic graphic = new
8             CompositeGraphic();
9
10        graphic.add(new Triangle());
11        graphic.add(new Triangle());
12        graphic.add(new Triangle());
13        graphic.add(new Triangle());
14
15        CompositeGraphic graphic2 = new
16            CompositeGraphic();
17
18        graphic2.add(new Quader());
19        graphic2.add(new Quader());
20        graphic2.add(new Quader());
21        graphic2.add(new Quader());
22
23        graphic.add(graphic2);
24
25        CompositeGraphic graphic3 = new
26            CompositeGraphic();
27
28        graphic3.add(new Circle());
29        graphic3.add(new Circle());
30        graphic3.add(new Circle());
31        graphic3.add(new Ellipse());
32
33        graphic.add(graphic3);
34
35        CompositeGraphic graphic4 = new
36            CompositeGraphic();
37
38        graphic4.add(new Octaeder());
39        graphic4.add(new Octaeder());
40        graphic4.add(new Octaeder());
41        graphic4.add(new Ellipse());
42
43        graphic.add(graphic4);
44
45        CompositeGraphic graphic5 = new
46            CompositeGraphic();
47
48        graphic5.add(new Triangle());
49        graphic5.add(new Quader());
50        graphic5.add(new Circle());
51        graphic5.add(new Ellipse());
52    }
53 }
```

4.4. Verhaltensmuster ▾

Verhaltensmuster beschreiben die Zuständigkeiten und Interaktionen zwischen Objekten.



4.4.1 Strukturmuster - Command

Command ▾

DAS **Command Muster** KAPSELT **Verhalten** ALS EIN **OBJEKT**.

DAMIT WIRD ES MÖGLICH OPERATIONEN IN WARTESCHLANGEN ZU STELLEN, LOGBUCHEINTRÄGE ZU FÜHREN BZW. OPERATIONEN RÜCKGÄNGIG ZU MACHEN.

► Erklärung: Eigenschaften der Komponenten ▾

- **ICommand:** ICOMMAND DEKLARIERT DIE SCHNITTSTELLE ALLER BEFEHLE. KONKRETE BEFEHLE WERDEN DADURCH MIT DER **EXECUTE** METHODE AUSGEFÜHRT.
- **PlayCommand, usw. :** EIN KONKRETER BEFEHL IMPLEMENTIERT DIE ICOMMAND SCHNITTSTELLE. DER KONKRETE BEFEHL DEFINIERT EINE BINDUNG ZWISCHEN EINER AKTION UND EINEM EMPFÄNGER.

► Codebeispiel: Command ▾

```

1 //-----
2 // Entwurfsmuster: Command
3 //-----
4 public interface ICommand{
5
6     void execute();
7
8     void undo();
9 }
10
11
12 public class PlayCommand implements ICommand{
13
14     public void execute(){
15         ...
16         System.out.println("play ...");
17     }
18
19     public void undo(){
20         ...
21         System.out.println("stop ...");
22     }
23
24
25 }
```

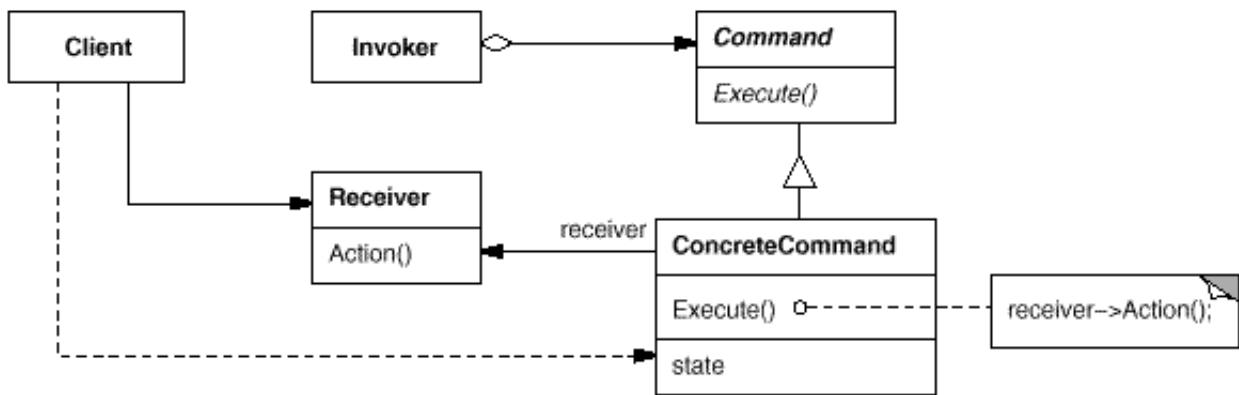


Abbildung 8. Entwurfsmuster - Command

► Codebeispiel: Command ▾

```
1 //-----  
2 // Entwurfsmuster: Command  
3 //-----  
4 public class PlayCommand implements ICommand{  
5  
6     public void execute(){  
7         ...  
8         System.out.println("play ...");  
9     }  
10  
11    public void undo(){  
12        ...  
13        System.out.println("stop ...");  
14    }  
15  
16 }  
17  
18 public class StopCommand implements ICommand{  
19  
20    public void execute(){  
21        ...  
22        System.out.println("stop ...");  
23    }  
24  
25    public void undo(){  
26        ...  
27        System.out.println("play ...");  
28    }  
29  
30 }  
31  
32 public class ForwardCommand implements ICommand{  
33  
34    public void execute(){  
35        ...  
36        System.out.println("forward ...");  
37    }  
38  
39 }
```

▶ Codebeispiel: Command ▾

```

1 //-----
2 // Entwurfsmuster: RemoteControl
3 //-----
4 public class RemoteControl{
5
6     private Stack< ICommand> history = new Stack<>();
7
8     private Stack< ICommand> undoHistory = new
9         Stack<>();
10
11    public void play(){
12        ICommand c = new PlayCommand();
13        c.execute();
14
15        history.push(c);
16    }
17
18    public void stop(){
19        ICommand c = new StopCommand();
20        c.execute();
21
22        history.push(c);
23    }
24
25    public void forward(){
26        ICommand c = new ForwardCommand();
27        c.execute();
28
29        history.push(c);
30    }
31
32    public void rewind(){
33        ICommand c = new RewindCommand();
34        c.execute();
35
36        history.push(c);
37    }

```

▶ Codebeispiel: Command ▾

```

1 //-----
2 // Entwurfsmuster: Command
3 //-----
4 public void do(){
5     ICommand command = undoHistory.pull();
6
7     if(command != null){
8         command.execute();
9         history.push(command);
10    }
11 }
12
13 public void undo(){
14     ICommand command = history.pull();
15
16     if(command != null){
17         command.undo();
18         undoHistory.push(command);
19    }
20 }
21
22 }
23
24
25 public class Launch{
26
27     public static void main(String args[]){
28         RemoteControl remote = new RemoteControl();
29
30         remote.play();
31         remote.stop();
32         remote.forward();
33     }
34 }

```



5. Programmierung: SOLID

04

SOLID Kriterien

01. SOLID Prinzipien	37
02. Single Responsibility Prinzip	38
03. Open Closed Prinzip	39
04. L. Substitutions Prinzip	41
05. Interface Segregation Prinzip	43

5.1. SOLID Prinzipien ▾

5.1.1 SOLID Prinzipien

Um die Programmierung hochwertigen Codes zu erleichtern, wurden **Prinzipien** für die Softwareentwicklung formuliert. Prinzipien objektorienterten Designs sind Prinzipien, die zu gutem objektorientierten Design führen.

Die **SOLID Prinzipien** sind eine Sammlung von objektorientierten **Programmierprinzipien**.

► Analyse: Objektorientiertes Design ▾

- GUTES OBJEKTOIENTIERTES DESIGN FÜHRT ZU GUT **lesbarem Code**.
- DAMIT WIRD ES FÜR MEHRERE **Entwickler** LEICHTER GLEICHZEITIG AN DER **Codebasis** ZU ARBEITEN.
- OBJEKTOIENTIERTES DESIGN FÜHRT ZU SCHWACHER **Koppelung** SOWIE STARKER **Kohäsion** DER SOFTWAREELEMENTE DER ANWENDUNG.

Die **SOLID Prinzipien** gemeinsam angewandt führen zu schwacher **Koppelung** und starker **Kohäsion** der Softwareelemente des Softwaresystems.

► Auflistung: SOLID Prinzipien ▾

Single Responsibility Prinzip ▾

DAS SINGLE RESPONSIBILITY PRINZIP FORDERT, DASS JEDOCH KLASSE DER ANWENDUNG NUR EINEN **einzelnen Aspekt** DER **Anwendungsspezifikation** IMPLEMENTIEREN DARF.

Open Closed Prinzip ▾

EIN SOFTWARESYSTEM MUSS STETS UM NEUE FUNKTIONALITÄT **erweitert** WERDEN KÖNNEN. WIRD EIN SYSTEM ERWEITERT DARF BESTEHENDER CODE JEDOCH **nicht verändert** WERDEN.

L. Substitutionsprinzip ▾

DAS LISKOVSCHE SUBSTITUTIONSPRINZIP ODER **Ersatzbarkeitsprinzip** FORDERT, DASS EINE INSTANZ DER ABGELEITETEN KLASSE SICH SO ZU VERHALTEN HAT WIE EIN OBJEKT DER BASISKLASSE.

SOLID Prinzipien



Interface Segregation Prinzip ▾

DAS INTERFACE SEGREGATION PRINZIP FORDERT ZU GROSSE **Interfaces** AUFZUTEILEN. DIE AUFTEILUNG DER SCHNITTSTELLE ERFOLGT GEMÄSS DER ANFORDERUNG DES CLIENTS AN DAS INTERFACES .



Dependency Injection ▾

DAS DEPENDENCY INVERSION PRINZIP DIENT ZUR Entkoppelung VON SOFTWAREELEMENTEN.



5.1.2 Kosten schlechten Codes

Die Programmierung einer Anwendung ist nur ein kleiner Teil der **Softwareentwicklung**. Etwa 70% der Tätigkeit der Softwareentwicklung fallen in den Bereich der **Softwarewartung**.

► Analyse: Kosten schlechten Codes ▾

- SCHLECHTE WARTBARKEIT DER ANWENDUNG.
- HOHER WARTUNGSARBEIT.
- HOHE KOSTEN IM RAHMEN DER WEITERENTWICKLUNG DER ANWENDUNG.
- AUFWENDNIGE FEHLERSUCHE.
- ERSCHWERTE CODEDOKUMENTATION.
- ERSCHWERTE CODELESBARKEIT.

5.2. Single Responsibility Prinzip ▾

5.2.1 Single Responsibility Prinzip

Gute **Objektorientierte Programmierung** ist daran zu erkennen, dass der Code auf viele, von ihrem Codeumfang her, **kleine Klassen** aufgeteilt ist.

► Analyse: Single Responsibility Prinzip ▾

- WIRD VERSUCHT IN EINER KLASSE **mehrere Aspekte** DER ANWENDUNG ABZUBILDEN, FÜHRT DAS UNWEIGERLICH ZU **kompliziertem, schlecht wartbarem Code**.
- DIE WAHRSCHEINLICHKEIT, DASS DIESER KLASSE ZU EINEM SPÄTEREM ZEITPUNKT **geändert** WERDEN MUSS, STEIGT ZUSAMMEN MIT DEM RISIKO, SICH BEI SOLCHEN ÄNDERUNGEN **Fehler** EINZUHANDELN.
- DAS **Single Responsibility Prinzip** FÜHRT IN DER REGEL ZU KLASSEN MIT **hoher Kohäsion**.

5.2.2 Fallbeispiel: S. Responsibility Prinzip

► Codebeispiel: S. Responsibility Prinzip ▾

```

1 //-----
2 // Single Responsibility Prinzip
3 //-----
4 public interface IDataOperation{
5
6     void execute() throws RichDataException;
7
8     String info();
9 }
```

► Codebeispiel: S. Responsibility Prinzip ▾

```

1  @NoArgsConstructor
2  @RequiredArgsConstructor
3  @Data
4
5  //-----
6  // Single Responsibility Prinzip
7  //-----
8  public abstract ADataOperation<T> implements
9      IDataOperation{
10
11     @NonNull
12     private Entity<T> entity;
13
14     public abstract void execute() throws
15         RichDataException;
16
17 }
18
19 @NoArgsConstructor
20 @Data
21 public InsertOperation<T> extends
22     ADataOperation<T>{
23
24     @Inject
25     private EntityManager em;
26
27     public void execute() throws RichDataException{
28         em.insert(this.getEntity());
29     }
30
31 }
32
33 @NoArgsConstructor
34 @Data
35 public UpdateOperation<T> extends
36     ADataOperation<T>{
37
38     @Inject
39     private EntityManager em;
40
41     public void execute() throws RichDataException{
42         em.merge(this.getEntity());
43     }
44
45 }
46
47 @NoArgsConstructor
48 @Data
49 public DeleteOperation<T> extends
50     ADataOperation<T>{
51
52     @Inject
53     private EntityManager em;
54 }
```

5.3. Open Closed Prinzip ▾

5.3.1 Open Closed Prinzip



Open Closed Prinzip ▾

EIN SOFTWARESYSTEM MUSS STETS UM NEUE FUNKTIONALITÄT **erweitert** WERDEN KÖNNEN. WIRD EINE ANWENDUNG ERWEITERT, DARF BESTEHENDER CODE JEDOCH **nicht verändert** WERDEN.

Durch den Einsatz von **objektorientierter Entwurfsmuster** wird sichergestellt das die Anwendung das **Open Closed Prinzip** nicht verletzt.

► Erklärung: Open Closed Prinzip ▾

- DAS **Open-Closed-Prinzip** IST EINES DER PRIMÄRE PRINZIPIEN DES **objektorientierten Entwurfs**.
- **Objektorientierte Entwurfsmuster** FOLGEN DEM **Open Closed Prinzip**.

► Codebeispiel: Open Closed Prinzip ▾

```

1  //-----
2  // Verletzung der Open Closed Prinzips
3  //-----
4  @NoArgsConstructor
5  @AllArgsConstructor
6  @Data
7  public class Apple implements Serializable{
8
9      private EColor color;
10
11 }
12
13 @NoArgsConstructor
14 @Data
15 public class AppleHandler{
16
17     public List<Apple>
18         filterGreenApples(List<Apple> apples){
19
20         List<Apple> filteredApples = new
21             ArrayList<>();
22
23         for(Apple a: apples){
24             if(a.getColor().equals(EColor.GREEN)){
25                 filteredApples.add(a);
26             }
27         }
28
29         return filteredApples;
30     }
31 }
```

► Analyse: Codebeispiele ▾

- SOLANGE NUR GRÜNE ÄPFEL AUSSORTIERT WERDEN SOLLEN, FUNKTIONIERT DER CODE EINWANDTFREI.
- SOLLEN NUN ABER ZUSÄTZLICH ALLE GRÜNEN APFEL FILTERED WERDEN, DIE NICHT MEHR ALS 200G WIEGEN MUSS DER BESTEHENDE CODE VERÄNDERT WERDEN.
- DAS BEDEUTET ABER DASS CODE DER BEREITS GETESTET UND AUSGELIEFERT WORDEN IST, VERÄNDERT WERDEN MUSS.
- ES LIEGT DAMIT EINE VERLETZUNG DES **Open Closed Prinzips** VOR.
- WIR WOLLEN NUN EINE LÖSUNG ENTWICKELN DIE **offen ist für neue Anforderungen, bestehender Code aber nicht verändert** WERDEN MUSS UM NEUE FUNKTIONALITÄT IM SYSTEM ZU INTEGRIEREN.

► Codebeispiel: Open Closed Prinzip ▾

```

1 //-----
2 // Strategieimplementierung
3 //-----
4 @AllArgsConstructor
5 @Data
6 public WeightFilter implements Predicate<Apple>{
7     private int weight;
8
9     public boolean test(Apple a){
10         if(a.getWeight() <= weight)
11             return true;
12         else
13             return false;
14     }
15 }
16
17 @AllArgsConstructor
18 @Data
19 public ColorFilter implements Predicate<Apple>{
20     private EColor color;
21
22     public boolean test(Apple a){
23         if(a.getColor().equals(color))
24             return true;
25         else
26             return false;
27     }
28 }
29
30 @AllArgsConstructor
31 @Data
32 public TypeFilter implements Predicate<Apple>{
33     private EAppleType type;
34
35     public boolean test(Apple a){
36         if(a.getType().equals(type))
37             return true;
38         else
39             return false;
40     }
41 }
42
43 @AllArgsConstructor
44 @Data
45 public TypeFilter implements Predicate<Apple>{
46     private EAppleType type;
47
48     public boolean test(Apple a){
49         if(a.getType().equals(type))
50             return true;
51         else
52             return false;
53     }
54 }
55
56 @AllArgsConstructor
57 @Data
58 public TypeFilter implements Predicate<Apple>{
59     private EAppleType type;
60
61     public boolean test(Apple a){
62         if(a.getType().equals(type))
63             return true;
64         else
65             return false;
66     }
67 }
```

► Codebeispiel: Open Closed Prinzip ▾

```

1 //-----
2 // Einsatz der Strategie Musters
3 //-----
4 @AllArgsConstructor
5 @NoArgsConstructor
6 @Data
7 public class Apple implements Serializable{
8
9     private String description;
10
11    private String label;
12
13    private EAppleType appleType;
14
15    private EColor color;
16
17    private int weight;
18
19 }
20
21 public enum EAppleType{
22     GOLDEN_LADY, ROSE
23 }
```

► Codebeispiel: Open Closed Prinzip ▾

```

1 //-----
2 // Strategieimplementierung
3 //-----
4 @AllArgsConstructor
5 @Data
6 public LabelFilter implements Predicate<Apple>{
7
8     private String label;
9
10    public boolean test(Apple a){
11        if(a.getLabel().equals(label))
12            return true;
13        else
14            return false;
15    }
16
17 }
18
19 @Data
20 public class AppleHandler implements Serializable{
21
22    public List<Apple> filterGreenApples(
23        List<Apple> apples,
24        Predicate<Apple> filter){
25
26        List<Apple> filteredApples = new
27            ArrayList<>();
28
29        for(Apple a: apples){
30            if(filter.test(a)){
31                filteredApples.add(a);
32            }
33        }
34
35        return filteredApples;
36    }
37
38 }
39
40 public class AppleHandlerTest{
41
42     @Test
43     public void testAppleHandler{
44         List<Apple> apples = Arrays.asList(
45             new Apple(...),
46             new Apple(...));
47
48         AppleHandler handler = new AppleHandler();
49         List<Apple> filteredApples = null;
50
51         filteredApples = handler.filter(
52             apples,
53             new ColorFilter(EColor.Green));
54     }
55 }
```

5.4. Liskovsche Substitutionsprinzip ▾

5.4.1 Liskovsche Substitutionsprinzip



L. Substitutionsprinzip ▾

DAS **Liskovsche Substitutionsprinzip** FORDERT, DASS EINE INSTANZ EINER abgeleiteten KLASSE SICH IM KONTEXT DES **polymorphen Aufruf** VERHÄLT WIE EIN OBJEKT DER **Basisklasse**.

► Erklärung: Substitutionsprinzip ▾

- EIN WICHTIGES ELEMENT OBJEKTOORIENTIERTER Programmierung IST DIE **Vererbung**: EINE KLASSE¹² WIRD VON EINER ANDEREN KLASSE¹³ ABGELEITET UND ERBT DABEI DEREN **Methoden** UND **Datenelemente**.
- IN DER **Kindklasse** KÖNNEN NEUE **Datenelemente** UND **Methoden** HINZUGEFÜGT ODER ERSETZT WERDEN.
- DIES FÜHRT JEDOCH ZUR FRAGE, WAS **Vererbung** ÜBER DIE **Beziehung** DER BASISKLASSE ZUR UNTERKLASSE AUSSAGT. DIESER FRAGE WIRD NORMALERWEISE BEANTWORTET MIT: **Vererbung** BESCHREIBT EINE **ist-ein** Beziehung.



5.4.2 Fallbeispiel: Substitutionsprinzip

► Beispiel: Substitutionsprinzip ▾

- EINE TYPISCHE HIERARCHIE VON KLASSEN IN EINEM GRAFIKPROGRAMM KÖNNTE Z. B. AUS EINER **Basisklasse** **GRAFISCHESELEMENT** UND DAVON ABGELEITETEN **Unterklassen** WIE **RECHTECK**, **ELLIPSE** ODER **TEXT** BESTEHEN.
- BEISPIELSWEISE WIRD MAN DIE ABLEITUNG DER KLASSE **ELLIPSE** VON DER KLASSE **GRAFISCHESELEMENT** BEGRÜNDEN MIT: EINE **ELLIPSE** IST EIN GRAFISCHES ELEMENT.
- DIE KLASSE **GRAFISCHESELEMENT** KANN DANN BEISPIELSWEISE EINE ALLGEMEINE METHODE **ZEICHNE** DEFINIEREN, DIE VON **ELLIPSE** ERSETZT WIRD DURCH EINE METHODE, DIE SPEZIELL EINE **ELLIPSE** **ZEICHNET**.



¹² Unterklasse

¹³ ihrer Basisklasse

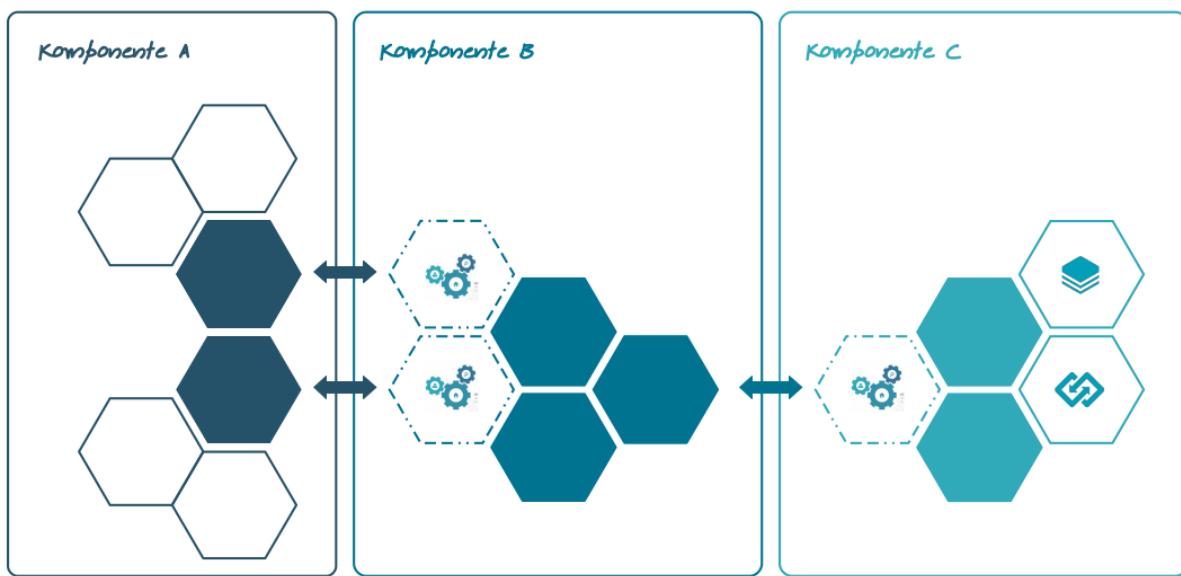


Abbildung 9. Kommunikation: Interface

► Analyse: Grafikbibliothek ▾

- DAS PROBLEM HIERBEI IST JEDOCH, DASS DAS **ist-ein-Kriterium** MANCHMAL IN DIE **Irre** FÜHRT.

Wird für das Grafikprogramm beispielsweise die Klasse **Kreis** definiert, so würde man bei naiver Anwendung des „**ist-ein-Kriteriums**“ diese Klasse von **Ellipse**¹⁴ ableiten.

- DIESE ABLEITUNG KANN JEDOCH IM KONTEXT DES GRAFIKPROGRAMMS FALSCH SEIN

Grafikprogramme erlauben es üblicherweise, die grafischen Elemente zu verändern. Beispielsweise lässt sich bei Ellipsen die Länge der beiden Halbachsen unabhängig voneinander ändern.

- FÜR EINEN KREIS GILT DIES JEDOCH NICHT, DENN NACH EINER SOLCHEN ÄNDERUNG WÄRE ER KEIN KREIS MEHR.
- HAT ALSO DIE KLASSE **ELLIPSE** DIE METHODEN **SKALIEREX** UND **SKALIEREY**, SO WÜRDE DIE KLASSE **KREIS** DIESER METHODEN ERBEN, OBWOHL IHRE ANWENDUNG FÜR EINEN KREIS NICHT ERLAUBT IST.

Das **Liskovsche Substitutionsprinzip** deckt hier das Problem auf.

Im vorliegenden Fall würde festgestellt, dass die Aussage „die Achsen können unabhängig voneinander skaliert werden“ zwar für die Klasse Ellipse, jedoch nicht für die Klasse Kreis gilt. Wäre jedoch Kreis eine Unterklasse von Ellipse, so müsste nach dem Liskovschen Substitutionsprinzip diese Aussage auch für die Klasse Kreis gelten.

□

¹⁴ denn ein Kreis ist eine Ellipse, nämlich eine Ellipse mit gleich langen Halbachsen

5.5. Interface Segregation Prinzip ▾

5.5.1 Interface Segregation Prinzip

☒ Interface Segregation Prinzip ▾

DAS INTERFACE SEGREGATION PRINZIP FORDERT ZU GROSSE **Interfaces** AUFZUTEILEN. DIE AUFTEILUNG DER SCHNITTSTELLE ERFOLGT GEMÄSS DER ANFORDERUNG DES CLIENTS AN DAS INTERFACES .

Das **Interface Segregation Prinzip** fordert komplexe **Schnittstellen** in mehrere einfache Schnittstellen **aufzuteilen**.

▶ Erklärung: Interface Segregation Prinzip ▾

- EINE **Softwareanwendung** BESTEHT AUS **Komponenten**.
- KOMPONENTEN **kommunizieren** ÜBER **Schnittstellen** MITEINANDER.
- FÜR DIE KOMMUNIKATION SOLLTE EINE KOMPONENTE NICHT EIN EINZELNES KOMPLEXES INTERFACE SONDERN MEHRERE VONEINANDER **unabhängige Interfaces** BEARBTETSTELLEN.
- DIE AUFTEILUNG SOLL GEMÄSS DER ANFORDERUNGEN DER **Client Komponenten** DURCHGEFÜHRT WERDEN DIE MIT DER KOMPONENTE KOMMUNIZIEREN.

```

17     Person create(Person p);
18
19     Person delete(Person p);
20
21 }
22
23
24 //-----
25 // Interface Segregation
26 //-----
27
28 public interface IPersonRepository{
29
30     Page<Person> findByAddress(Pageable pageable,
31                               Address address);
32
33     Page<Person> findByName(Pageable pageable,
34                             String name);
35
36     List<Person> findByAddress(Address address);
37
38     List<Person> findByName(String name);
39 }
40
41 public interface IPersonService{
42
43     Person create(Person p);
44
45     Person update(Person p);
46
47     Person delete(Person p);
48
49 }
```

5.5.2 Fallbeispiel: Interface Segregation

▶ Codebeispiel: Interface Segregation Prinzip ▾

```

1 //-----
2 // Überbeladenes Interface
3 //-----
4 public interface IPersonService{
5
6     Page<Person> findByAddress(Pageable pageable,
7                               Address address);
8
9     Page<Person> findByName(Pageable pageable,
10                            String name);
11
12    List<Person> findByAddress(Address address);
13
14    List<Person> findByName(String name);
15
16    Person getByID(Long personID);
17
18    Person update(Person p);
```


Verteilte Systeme: Eigenschaften

Version 2018.09.01

6. Verteilte Systeme: Eigenschaften

01

Verteilte Systeme

01. Verteilte Systeme	47
02. Eigenschaften verteilter Systeme	49
03. Load Balancer	??

6.1. Verteilte Systeme

6.1.1 Systembegriff



System ▾

DAS GRIECHISCHE WORT **Systema** BEDEUTET EIN AUS MEHREREN **Teilen** ZUSAMMENGESETZTES, GEGLIEDERTES **Ganzes**.

► Erklärung: **System** ▾

- **Systeme** BESTEHEN DEMNACH AUS MEHREREN **Elementen**, DIE UNTEREINANDER **Nachrichten** AUSTAUSCHEN.
- EIN **SYSTEM** IST STETS MEHR ALS DIE SUMME SEINER **TEILE**.



6.1.2 Verteilte Systeme



Verteiltes System ▾

IN DER **Softwareentwicklung** SPRECHEN WIR VON EINEM **verteilten System** WENN EINE SOFTWAREANWENDUNG AUS MEHREREN **Betriebssystemprozessen** Besteht, die miteinander kommunizieren.

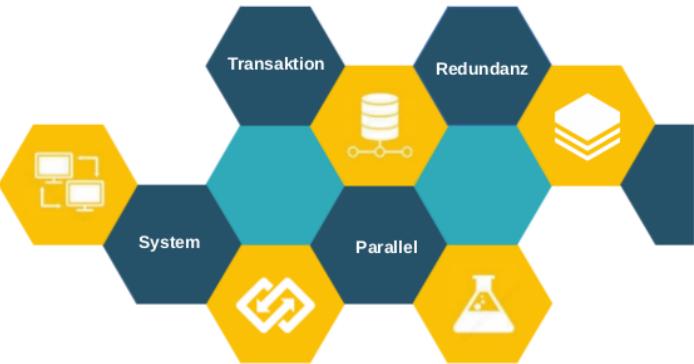
Ein System ist dabei stets **mehr** als die **Summe** seiner Teile.

► Erklärung: **Verteiltes System** ▾

- EIN **verteiltes System** WIRD IN DER REGEL, VERTEILT AUF MEHRERE RECHNER IN EINEM NETZWERK, AUSGEFÜHRT. DIE EINZELNEN TEILE DES SYSTEMS WERDEN DABEI IN EINEM EIGENEN **Betriebssystemprozess** AUSGEFÜHRT.
- DIE **Geschäftslogik** DER ANWENDUNG IST DABEI AUF MEHRERE **Komponenten** AUFGTEILT, DIE IN EINEM EIGENEN BETRIEBSSYSTEMPROZESS AUSGEFÜHRT WERDEN. EINE SOLCHE KOMPONENTE WIRD AUCH ALS **Service** BEZEICHNET.
- AUS DER SICHT DES **Systementwicklers** MACHT ES DABEI KEINEN UNTERSCHIED OB EIN SERVICE IM **Speicher** EINES EINZELNEN RECHNERS ODER **verteilt** AUF MEHRERE RECHNER AUSGEFÜHRT WIRD.



Verteilte Systeme



6.1.3 Merkmale verteilter Systeme

Der **Entwurf verteilter System** erwuchs aus der Notwendigkeit, Ressourcen verteilt im **Netzwerk**, nutzbar zu machen.

► Auflistung: Merkmale verteilter Systeme ▾

- parallele Verarbeitung** ▾
PARALLELE Verarbeitung VON BENUTZER- UND Serviceanfragen.
- Shard Access** ▾
GEMEINSAME Nutzung VON Betriebsmitteln - z.B.: DATENREPOSITORIES, DATEIEN USW.
- Modularisierung** ▾
DIE MODULARISIERUNG DER ANWENDUNG IN EINZELNE Diensten ERLAUBT ES DIENSTE IN ANDEREN ANWENDUNGEN wiederzuverwenden.
- Replikation** ▾
Speicherung VON DATEN AUF MEHREREN KNOTEN IM NETZWERK ZUR ERHÖHUNG DER Ausfallsicherheit.
- verteiltes Transaktionsmodell** ▾
SICHERSTELLUNG DER Konsistenz DER Daten ÜBER ALLE DIENSTE DER ANWENDUNG HINWEG.

► Auflistung: Forderungen ▾

- **Ausfallsicherheit:** KOMPLEXE Anwendung MÜSSEN IN DER LAGE SEIN EBENFALLS BEIM Ausfall VON TEILEN DER HARDWARE ZU FUNKTIONIEREN.
Durch die **Verteilung** von **Daten** und **Diensten** auf mehrere Netzwerknoten wird eine höhere **Robustheit** und Verfügbarkeit der Anwendung erreicht.
- **Kommunikationsverbund:** Übertragung VON DATEN IN EINEM NETZWERK.
- **Lastenverbund:** AUFTEILUNG VON Useranfragen AUF MEHRERE Instanzen DER ANWENDUNG.

6.1.4 Dezentrale Systeme

- Dezentrale Systeme** ▾
EIN STARK ENTKOPPELTES verteiltes System WIRD ALS dezentrales System BEZEICHNET.

► Erklärung: Dezentrale Systeme ▾

- **Dezentrale Systeme** STELLEN EINE WEITERENTWICKLUNG DER **Verteilten Systeme** DAR.
- **Dezentrale Systeme** ZEICHEN SICH DADURCH AUS, DASS SIE WESENTLICHER STÄRKER **entkoppelt** SIND, ALS GEWÖHNLICHE VERTEILTE SYSTEME.
- ENTWICKELT WURDEN DEZENTRALE SYSTEM UM KOMPLEXE **Geschäftsprozesse** ABZUBILDEN.

6.2. Eigenschaften verteilter Systeme

Um die Architekturkonzepte **verteilter Systeme** zu verstehen, wollen wir zuerst die Eigenschaften verteilter Systeme besprechen.



6.2.1 Eigenschaft: Transparenz



Transparenz ▾

EIN **verteiltes System**, DAS SICH BENUTZERN SO PRÄSENTIERT, ALS HANDLE ES SICH UM EIN **einzelnes Service**, WIRD ALS **transparent** BEZEICHNET.

Ein wichtiges Ziel **verteilter Systeme** ist es, die Tatsache zu verbergen, dass seine PROZESSE und RESOURCEN **physisch** auf mehrere Computer **verteilt** sind.

► Auflistung: Arten der Transparenz ▾

- **Zugriffstransparenz:** EIN ZUGRIFFSTRANSPARENTE SYSTEM IST IN DER LAGE, **Unterschiede** IN DER **Datenspeicherung** UND DER ART DES ZUGRIFFS AUF RESOURCEN ZU **verbergen**.

Ein **verteiltes System** kann **Dienste** beinhalten, die auf unterschiedlichen **Betriebssystemen** ausgeführt werden. Jedes Betriebssystem folgt seinen eigenen **Namenskonventionen** bei der Speicherung und Verarbeitung von Daten. Ein Verteiltes System muß in der Lage sein diese Unterschiede vor Benutzern des Systems zu verbergen.

- **Replikationstransparenz:** **Replikation**¹⁵ SPIELT EINE ZENTRALE ROLLE IN **verteilten Systemen**. **Replikationstransparente Systeme** VERBERGEN VOR DEM BENUTZER DIE TATSACHE DASS DIE ANWENDUNG FÜR BESTIMMTE RESSOURCEN MEHRERE KOPIEN GLEICHZEITIG VERWALTET.
- **Nebenläufigkeitstransparenz:** EINES DER PRIMÄREN ZIELE **verteilter Systeme** IST DIE GEMEINSAME **Nutzung** VON RESSOURCEN.

¹⁵ Allgemein wird Replikation eingesetzt, um Daten an mehreren Knoten im Netzwerk verfügbar zu machen. Die einfachste Form der Datenreplikation ist die Speicherung, der Kopie einer Datei an mehreren Knoten im Netzwerk

- **Ortstransparenz:** **Ortstransparenz** VERBIRGT VOR DEN BENUTZER EINER ANWENDUNG DIE PHYSISCHE POSITION EINER RESSOURCE IM NETZWERK.

In Verteilten Systemen wird **Ortstransparenz** mit der Hilfe von **Namensdiensten** implementiert. Jeder Ressource im System wird ein **logischer Name**¹⁶ zugeordnet. Der logische Name für sich enthält keine Information zum Ort an dem sich die Ressource im System befindet. Mit der Hilfe des Namensdienstes kann der logischen Namen in eine IP Adresse umgewandelt werden.



6.2.2 Eigenschaft: Skalierbarkeit



Skalierbarkeit ▾

Skalierbarkeit BEZEICHNET DIE FÄHIGKEIT EINES VERTEILTN SYSTEM BEI WACHSENDER **Systemlast** SEINEN DIENST WEITER AUFRECHTHALTEN ZU KÖNNEN.

Ein hochskalierendes, verteiltes System ist in der Lage tausende von Anfragen in kürzester Zeit zu bearbeiten.

Folgende **Problemaspekte** müssen bei einer Skalierbarkeitsprüfung beachtet werden.

► Auflistung: Problemaspekte ▾

- **Zentrale Dienste:** EIN GLOBAL ERREICHBARER **Dienst** KANN VON EINER BELIEBIGEN ANZAHL VON **USERN** IN ANSPRUCH GENOMMEN WERDEN. WIRD DIE **Last** FÜR DAS SERVICE ZU GROSS, MUSS DAS SERVICE **repliziert** WERDEN.
- **Zentrale Daten:** ALS **Zentralen Daten** WIRD EIN DATENBANKSERVER BZW. EINE DATEIRESOURCE BEZEICHNET AUF DIE GLEICHZEITIG VON MEHREREN BENUTZERN ZUGEGRIFFEN WERDEN KANN.
- **Zentrale Algorithmen:** EIN BEKANNTES BEISPIEL FÜR EINEN **zentralen Algorithmus** IST DER **2PC ALGORITHMUS**. MIT DER HILFE DES 2PC ALGORITHMUS KÖNNEN **Transaktionen** ÜBER DIE GRENZEN EINER EINZELNEN ANWENDUNG HINWEG DURCHGEFÜHRT WERDEN.



¹⁶ Der logische Name kann mit der URL einer Resource im Internet verglichen werden.

Eigenschaften - Verteilter Systeme



6.2.3 Bigdata



Bigdata ▾

DER BEGRIFF **Big Data** BESCHREIBT **Datenbestände**, DIE AUFGRUND IHRES UMFANGS, IHRER UNTERSCHIEDLICHKEIT ODER IHRER SCHNELLEBIGKEIT EINE IMMENSE **Last** AN DAS ZU VERARBEITENDE SYSTEM STELLEN.

Ein verteiltes System muß hoch skalierbar sein um im Bigdata Bereich eingesetzt werden zu können.

Die technischen **Herausforderungen** im **Big Data** Umfeld werden abstrahiert als die 3V - Volume, Velocity, Variety.

► Auflistung: Herausforderungen im Big Data ▾



Volume ▾

Mit **Volume** IST DIE SCHIERE MENGE AN DATEN GEMEINT, DIE PRO ZEITEINHEIT VERARBEITET WERDEN MUSS.



Velocity ▾

Velocity BESCHREIBT DIE **Geschwindigkeit**, MIT DER DATEN VERARBEITET WERDEN MÜSSEN.



Variety ▾

Variety BESCHREIBT DIE UNTERSCHIEDLICHEN GRADE DER **Strukturierung** DER DATEN.

6.2.4 Skalierbarkeit vs: Performance

Die Begriffe **Skalierbarkeit** und **Performance** werden häufig synonym benutzt, obwohl es sich hier um 2 **unterschiedliche** Konzepte handelt.



► Erklärung: Performance vs Skalierbarkeit ▾

- **Performance:** DIE **Performance** EINER ANWENDUNG, GIBT DIE **Geschwindigkeit** AN, IN DER EINE **Anfrage** AN DAS SYSTEM, VERARBEITET WERDEN KANN.
- **Skalierbarkeit:** **Skalierbarkeit** MISST DIE FÄHIGKEIT EINER ANWENDUNG, DIE **Geschwindigkeit** EINER ANFRAGE BEI STEIGENDER **Last** AUFRECHTERHALTEN ZU KÖNNEN.

Unter Skalierbarkeit wird die Eigenschaft eines Systems bezeichnet trotz wachsender **Systemlast** die Verfügbarkeit des Systems zu gewährleisten zu können.

- **Verfügbarkeit:** DIE **Verfügbarkeit**¹⁷ EINES **verteilten Systems** IST DIE WAHRSCHEINLICHKEIT, DASS DAS SYSTEM **Clientanfragen** ZU EINEM BESTIMMTEN ZEITPUNKT BZW. INNERHALB EINES VEREINBARTEN ZEITRAHMENS UMSETZEN KANN.

¹⁷ Beachten Sie dass die Begriffe **Performance**, **Skalierbarkeit** und **Verfügbarkeit** unterschiedliche Eigenschaften eines Verteilens Systems beschreiben

6.2.5 Dimensionen der Skalierbarkeit

Es gibt mehrere Techniken um für ein verteiltes System **Skalierbarkeit** zu erreichen.

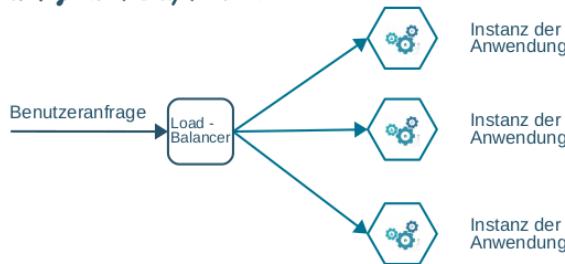


► Auflistung: Dimensionen der Skalierbarkeit ▾

- **X Scaling - Horizontal Duplication:** DIE **X Skalierung** IST DIE AM HÄUFIGSTEN EINGESETZTE SKALIERUNGSTECHNIK.

Bei der X Skalierung werden mehrere **Instanzen** der Anwendung hinter einem **Load Balancer** ausgeführt. Der **Load Balancer** verteilt die Benutzeranfragen auf die Instanzen der Anwendung. Die X Skalierung ist die einfachste Form der Skalierung. Das System wird über die Hardware skaliert.

Horizontal Duplication



- **Z Scaling - Data Partitioning:** BEI DER **Z Skalierung** WERDEN EBENFALLS MEHRERE INSTANZEN DER ANWENDUNG AUSGEFÜHRT.

Im Gegensatz zur X Skalierung ist jede Instanz der Anwendung jedoch nur für einen bestimmten Teil der Daten¹⁸ verantwortlich. Die Benutzeranfragen werden nun über einen **Router** verteilt.

- **Y Scaling - Functional Decomposition:** MIT DER **X** UND **Z Skalierung** WIRD DIE **Verfügbarkeit** UND **Kapazität** EINER ANWENDUNG VERBESSERT. DIESSE FORMEN DER SKALIERUNG HABEN JEDOCH KEINE AUSWIRKUNG AUF DIE **Komplexität** EINER ANWENDUNG.

Mit der Y Skalierung wird eine Anwendung in Services unterteilt.

6.2.6 Eigenschaft: Offenheit



Offenheit ▾

Offenheit BESCHREIBT DIE MÖGLICHKEIT EINES verteilen Systems ERWEITERT WERDEN ZU KÖNNEN.

► Erklärung: Softwarewartbarkeit ▾

- EINE VERTEILTE ANWENDUNG MUSS **offen** SEIN UM **wartbar** ZU SEIN.
- GUTE **Wartbarkeit** BEDEUTET, MIT ÄNDERUNGEN EINFACHER UND FINANZIELL GÜNSTIGER UMGEHEN ZU KÖNNEN, SOWIE ÄNDERUNGEN ZU VERMEIDEN, DIE NICHT NOTWENDIG SIND.
- UNWARTBARE SOFTWARE IST KURZLEBIG. SOFTWARE DIE NICHT NUR SCHWER ERWEITERT WERDEN KANN, WIRD IN DER REGEL SCHNELL ERSETZT.

Um **Offenheit** zu erreichen, muß es möglich sein eine Anwendung in **Module** bzw. Komponenten strukturieren zu können.

■ Offene Verteilte Systeme sind stark **entkoppelte** Systeme.



► Auflistung: Erweiterte Formen der Offenheit ▾



Interoperabilität ▾

Interoperabilität BESCHREIBT DAS AUSMASS, IN WELCHEM ZWEI Systeme bzw. Komponenten unterschiedlicher Hersteller zusammenarbeiten KÖNNEN.



Portabilität ▾

Portabilität BESCHREIBT, IN WELCHEM AUSMASS EINE ANWENDUNG, DIE FÜR EIN BESTIMMTES Betriebssystem ENTWICKELT WURDE, OHNE VERÄNDERUNG AUF EINEM ANDEREN BETRIEBSSYSTEM ausgeführt WERDEN KANN.

¹⁸ Alle Personen mit einem bestimmten Nachnamen

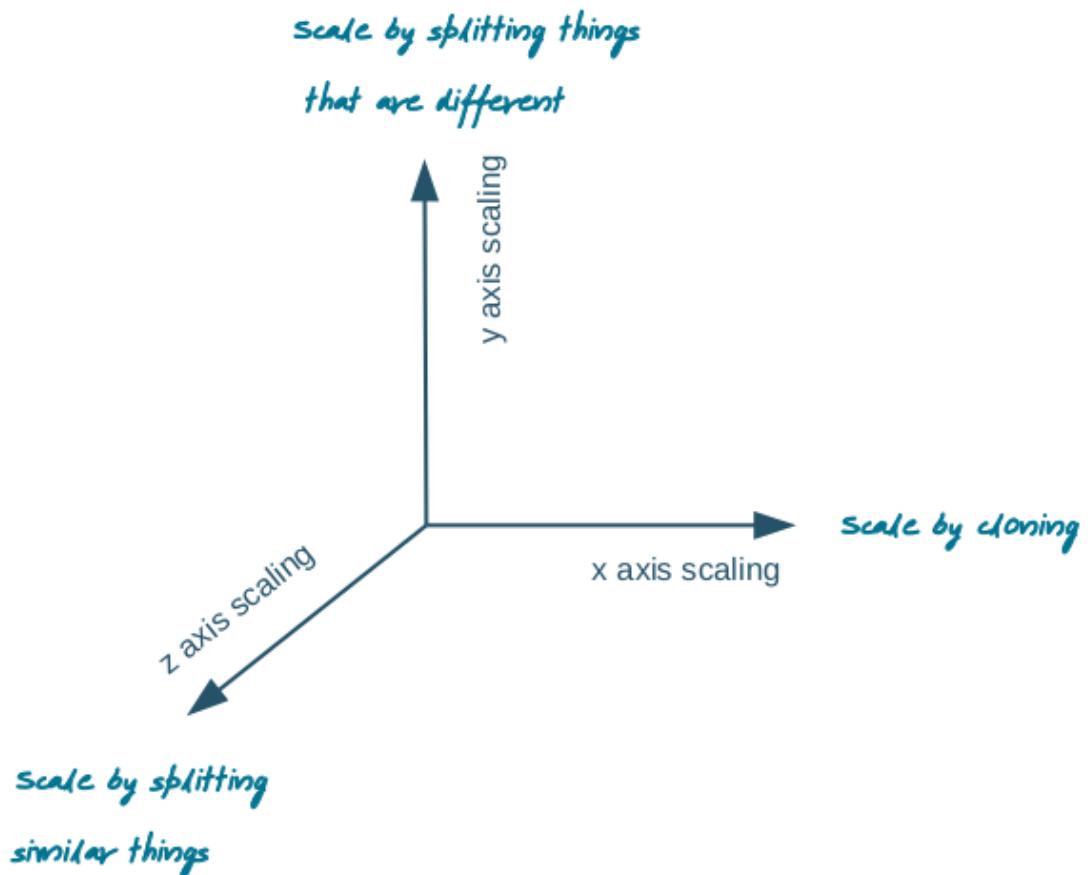


Abbildung 10. Dimensionen der Skalierbarkeit

6.2.7 Portabilität

Eine **Softwareanwendung** benötigt eine **Laufzeitumgebung** - Betriebssystem, virtuelle Maschine - in der sie ausgeführt wird. Portabilität beschreibt die Eigenschaft einer Anwendung auf unterschiedlichen Laufzeitumgebungen ausgeführt werden zu können.



6.2.8 Interoperabilität

Unter **Interoperabilität** versteht man die Fähigkeit eines Systems mit anderen Systemen zusammenzuarbeiten.



► Erklärung: Interoperabilität ▾

- DAMIT **Systeme** UNTERSCHIEDLICHER Hersteller MIT-EINANDER ZUSAMMENARBEITEN KÖNNEN BENÖTIGT ES EINE ZUSÄTZLICHE UNABHÄNGIGE **Kommunikations-schicht**.
- FÜR ANWENDUNGEN IM **Internet** STELLT DAS **HTTP Protokoll** EINE SOLCHE KOMMUNIKATIONSSCHICHT DAR.

7. Verteilte Systeme: Konzepte

02

Konzepte

01. Load Balancing	53
02. Routing	54
03. Verschachtelte Transaktionen	55

7.1. Load Balancing



Lastverteilung ▾

Lastverteilung WIRD IN DER INFORMATIK EINGESetzt WENN UMFANGREICHE **Berechnungen** ODER GROSSE MENGEN VON ANFRAGEN AUF MEHRERE PARALLEL ARBEITENDE SYSTEME **verteilt** WERDEN SOLLLEN.

7.1.1 Grundlagen

Lastverteilung kommt überall dort zum Einsatz, wo Clientsysteme eine derart hohe **Anfragedichte** erzeugen dass ein einzelner Server Rechner damit überlastet werden würde.

Eine einfache **Lastverteilung** findet zum Beispiel auf Rechnern mit mehreren Prozessoren statt.

► Erklärung: Lastverteilung Webserver ▾

- INSbesondere bei **Webservern** ist eine **Lastverteilung** wichtig, da ein einzelner Host nur eine begrenzte Menge an HTTP Anfragen auf einmal beantworten kann.
- Zur Aufteilung der **Last** wird einem Cluster von Servern ein **Load Balancer** vorgeschaltet, der die Anfragen auf die Server aufteilt.
- Problematisch ist bei diesem Verfahren, dass der gesamte Verkehr über den **Load Balancer** fliesst, dieser also früher oder später zum **Engpass** wird, sofern dieser zu klein ausgelegt wurde.

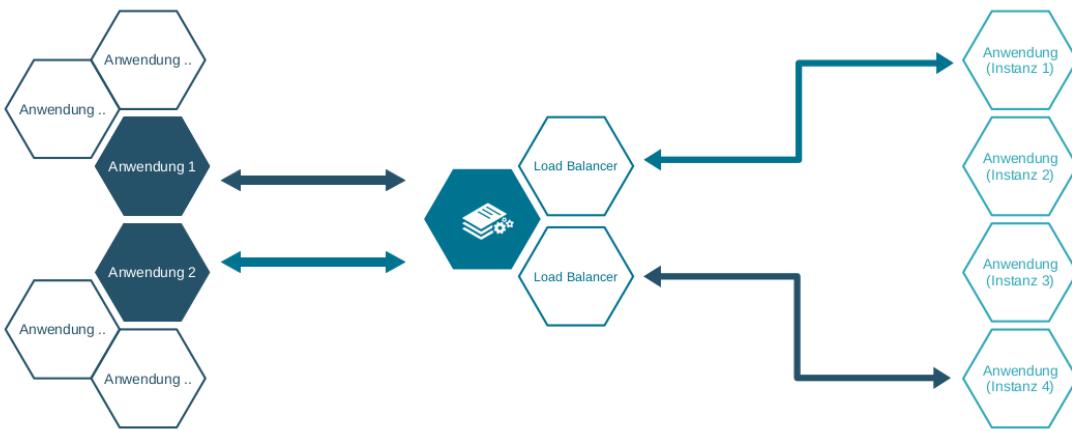
7.1.2 Probleme des Load Balancing

Anwendungen wie Online Shops verwalten Client Anfragen oft über **Sessions**¹⁹.

► Analyse: Probleme des Loadbalancing ▾

- Die Verwendung einer spezifischen **Session** bzw. eines **Contexts** pro User setzt voraus, dass ein Client, für den bereits eine Session eröffnet wurde, immer wieder mit demselben Server kommunizieren sollte.

¹⁹ In einer Session wird z.B.: der Inhalt des Warenkorbs gespeichert



1. Clientkomponenten generieren Http Anfragen an die Serverkomponente

2. Der Load Balancer leitet die Anfragen auf eine der Instanzen der Anwendung um.

Abbildung 11. Load Balancer

- DAMIT MÜSSEN ALLE **Verbindungen** EINES CLIENTS AUF DENSELBNEN SERVER GELEITET WERDEN.
- DAS WEITERLEITEN DER ANFRAGEN AUF IMMER DENSELBNEN **Backendserver** WIRD ALS **Affinität**²⁰ BEZEICHNET.
- ALTERNATIV KANN DAS **Problem** AUCH DURCH DIE **Softwareanwendung** SELBST GELÖST WERDEN Z.B.: DURCH DIE SPEICHERUNG DER SESSEION IN DER DATENBANK. DA-DURCH KANN EINE ANFRAGE VON EINEM BELIEBIGEN SERVER BEANTWORTET WERDEN.

7.2. Routing

7.2.1 Routing



Routing ▾

DER BEGRIFF **Routing** BEZEICHNET DIE **Wegfin-dung** VON INFORMA-TION-S- ODER **Datenströmen** VON EINER QUELLE ZU IHREM ZIEL.

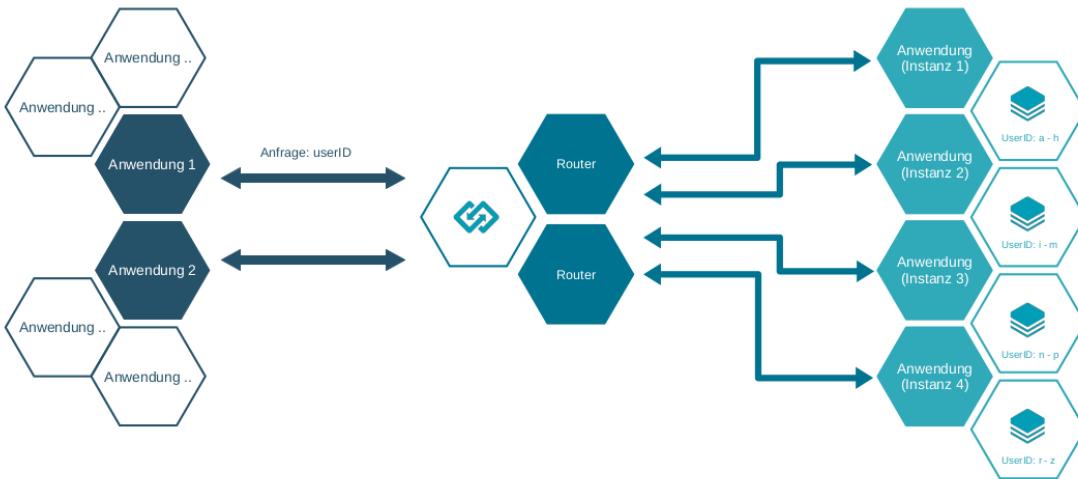


Routing wird auch verwendet um **Z-Skaling** in ver-teilten Systemen zu implementieren.

► Erklärung: Forwarding vs. Routing ▾

- **Routing:** DAS **Routing** BESTIMMT DEN GESAMTEN WEG EINES **Nachrichtenstroms** DURCH DAS NETZWERK.
- **Forwarding:** **Forwarding** BESCHREIBT DEN **Entschei-dungsprozess** EINES EINZELNEN NETZKNOTENS, ÜBER WELCHEN SEINER NACHBARN ER EINE VORLIEGENDE NACHRICHT WEITERLEITEN SOLL.

²⁰ Der dabei vorgeschaltete Load Balancer fügt der HTTP Anfrage dazu zusätzliche Informationen hinzu



1. Clientkomponenten generieren Http Anfragen an die Serverkomponente. Zusätzlich wird im Header der Nachricht ein Token angegeben der für den Router als Entscheidungsgrundlage für das Routing enthält.

2. Der Router analysiert den Token und leitet die Anfrage entsprechend einer der Instanzen der Anwendungen weiter.

Abbildung 12. Routing

7.2.2 Z Scaling und Routing

Routing wird auch verwendet um Z-Skaling in verteilten Systemen zu implementieren.

Erklärung: Z Scaling ▾

- GENAU SO WIE BEI DER X Skalierung WERDEN MEHRERE INSTANZEN DER ANWENDUNG DEPLOYED.
- IM GEGENSATZ ZUR X Skalierung IST JEDOCH DER INSTANZEN DER ANWENDUNG FÜR EINEN BESTIMMTEN BEREICH DER Daten VERANTWORLICH.



7.3. Verschachtelte Transaktionen ▾



Transaktion ▾

ALS Transaktion BEZEICHNET MAN EINE Folge VON Programmschritten DIE ALS LOGISCHE EINHIEFT BETRACHTET WERDEN. NACH FEHLERFREIER AUSFÜHRUNG DER TRANSAKTION WIRD DER DATENBESTAND IN EINEM konsistenten ZUSTAND HINTERLASSEN.

Für Transaktionen wird gefordert, dass entweder vollständig und fehlerfrei oder gar nicht ausgeführt wird.

7.3.1 Verschachtelte Transaktionen

Eine verteile Anwendung ist ein System von Prozessen. In einem verteilten System muß deshalb eine Transaktion über mehrere Prozesse hinweg abgewickelt werden.

Für **verteilte Systeme** muß ein neues Konzept für **Transaktionen** gefunden werden.

Verschachtelte Transaktionen ▾

EINE **verschachtelte Transaktion** BESTEHT AUS EINER MENGE UNABHÄNGIGER TRANSAKTIONEN.

► Erklärung: Verschachtelte Transaktionen ▾

- FÜR JEDEN PROZESS DER TEIL DER **verschachtelten Transaktion** IST, WIRD EINE EIGENE UNABHÄNGIGE TRANSAKTION GESTARTET.
- DIE **verschachtelte Transaktion** VERWALTET DIE SO GESTARTETEN TRANSAKTIONEN.
- JEDE DER **Transaktionen** WIRD BEHANDELT ALS WÄREEN SIE EIN **Programmschritt** INNERHALB EINER HERKÖMMLICHEN TRANSAKTION.
- DIE **verschachtelte Transaktion** KANN NUR DANN DURCHGEFÜHRT WERDEN, WENN JEDE DER EINGEBETTELEN TRANSAKTIONEN DURCHGEFÜHRT WERDEN KONNTE.



7.3.2 2PC- Two Phase Commit

Für das Durchführen **verschachtelter Transaktionen** gibt es ein eigenes **Protokoll**.



2PC ▾

DAS 2PC IST EIN **Protokoll** ZUM DURCHFÜHREN VON **verschachtelten Transaktionen**.

► Erklärung: Ablauf des 2PC ▾

- DAS **2PC Protokoll** BESTEHT AUS 2 SCHRITTEN: **Ergebnis ermitteln** UND **Transaktion abschließen**.
- **1.Schritt: Ergebnis ermitteln** DIE **verschachtelte Transaktion** - DER KOORDINATOR - SCHICKT EINEN **VOTE REQUEST** AN ALLE BEINHALTEDEN TRANSAKTIONEN - DIE TEILNEHMER.
- DIE **Teilnehmer** ANTWORTEN MIT EINEM **VOTE COMMIT** - FALLS DIE TRANSAKTION DURCHGEFÜHRT WERDEN KANN - BZW. MIT EINEM **VOTE ABORT** FALLS DIE TRANSAKTION NICHT DURCHGEFÜHRT WERDEN KANN.

- **2.Schritt: Transaktion abschließen** DER **Koordinator** SENDET EINEN **GLOBAL COMMIT** AN ALLE TEILNEHMER FALLS ALLE TEILNEHMER IM ERSTEN SCHRITT MIT EINEM **vote commit** GEANTWORTET HABEN.
- FÜR DEN FALL DASS MINDESTENS EIN TEILNEHMER IM ERSTEN SCHRITT MIT EINEM **vote abort** GEANTWORTET HAT SENDET DER KOORDINATOR EIN **GLOBAL ABORT**.
- JE NACH ERGANGENER NACHRICHT FÜHREN DIE EINZELNEN **Teilnehmer** DIE **Transaktion** DURCH ODER NICHT.



8. Verteilte Systeme: Replikation

03

Replikation

01. Replikation

57

8.1. Replikation



Replikation ▾

Replikation BEZEICHNET DIE MEHRFACHE Speicherung DERSELBEN DATEN AN MEIST MEHREREN Standorten UND DIE Synchronisation DIESER DATENQUELLE.

8.1.1 Grundlagen

Replikation dient in der Datenverarbeitung dazu, Daten an mehreren Orten verfügbar zu machen.

► Erklärung: Replikation von Daten ▾

- Replikation DIENT EINERSEITS ZUR Datensicherung.
- ANDERERSEITS WIRD REPLIKATION ZUR VERKÜRZUNG DER Antwortzeiten, BESONDERS FÜR LESENDE Datenzugriffe.
- DIE EINFACHSTE FORM DER Datenreplikation IST DIE SPEICHERUNG EINER KOPIE EINER DATEI.



8.1.2 Replikation von Daten

Die Herausforderung bei der Replikation von Daten ist es die Kopien der Daten konsistent zu halten.



Prinzip: Kommunikationsprotokolle

Version 2018.09.01

9. Kommunikationsprotokoll: Http

01

Http Protokoll

01. Kommunikationsprotokolle	59
02. Netzwerkprotokolle	60
03. Http Protokoll	61
04. Http Request	62
05. Http Response	63

9.1. Kommunikationsprotokolle



Kommunikationsprotokoll ▾

EIN KOMMUNIKATIONSPROTOKOLL IST EINE VEREINBARUNG, NACH DER DIE **Datenübertragung** ZWISCHEN ZWEI ODER MEHREREN PARTEIEN ABLÄUFT.

In seiner einfachsten Form kann ein **Protokoll** definiert werden als ein Menge von **Regeln**, die Syntax, Semantik und Synchronisation der Kommunikation bestimmen.

9.1.1 Grundlagen: Syntax

Unter **Syntax** versteht man im allgemeinen ein **Regelsystem** zur Kombination elementarer **Zeichen** zu zusammengesetzten Zeichen in natürlichen oder künstlichen Zeichensystemen.

► Erklärung: Syntax ▾

- UNTER DER **Syntax** EINER **formalen Sparache**²¹ VERSTEHT MAN EIN SYSTEM VON REGELN NACH DENEN WOHLGEFORMTE **Ausdrücke** BZW. PROGRAMMTEXTEAUS EINEM GRUNDLEGENDEN ZEICHENVORRAT GEBILDET WERDEN.
- **Wohlgeformtheit:** BEI DER **Auszeichnungssprache XML** GIBT ES EINE FÜR ALLE DOKUMENTE GÜLTIGE SYNTAX. DIE ÜBEREINSTIMMUNG MIT DER ALLGEMEINEN SYNTAX WIRD ALS **Wohlgeformtheit** BEZEICHNET.
- **Validität:** JE NACH **Anwendungsbereich**²² KANN DIE SYNTAX DURCH ZUSÄTZLICHE SYNTAXREGELN EINGESCHRÄNKKT WERDEN. DIE ÜBEREINSTIMMUNG MIT DEN ZUSÄTZLICH DEFINIERTEN REGELN WIRD **Validität** GENANNT.

► Erklärung: Syntax von Protokollen ▾

- DIE **Syntax** EINES **Kommunikationsprotokolls** BESSCHREIBT WELCHER **Zeichsatz** ZUR DARSTELLUNG DER BEFEHLE DES PROTOKOLLS VERWENDET WIRD.
- ZUSÄTZLICH WIRD DEFINIERT WELCHE WORTE IM KONTEXT DES PROTOLOLLS VALIDE SIND.



²¹ z.B.: Kalkül, Netzwerkprotokoll, XML, C++

²² Für xml Schema dürfen nur ganz bestimmte XML Elemente verwendet werden im Gegensatz zur XML Sprachdefinition.

9.1.2 Grundlagen: Semantik



Semantik ▾

DIE **Semantik** BESCHREIBT DIE BEDEUTUNG DER ZEICHEN EINER FORMALEN Sprache.

Zeichen KÖNNEN HIERBEI BELIEBIGE **Symbole** SEIN, INSbesondere ABER auch SÄTZE, SATZTEILE, WÖRTER ODER WORTTEILE.

Durch die Definition der **Syntax** und **Semantik** können formale Sprachen vollständig beschrieben werden.



9.1.3 Grundlage: Synchronisation

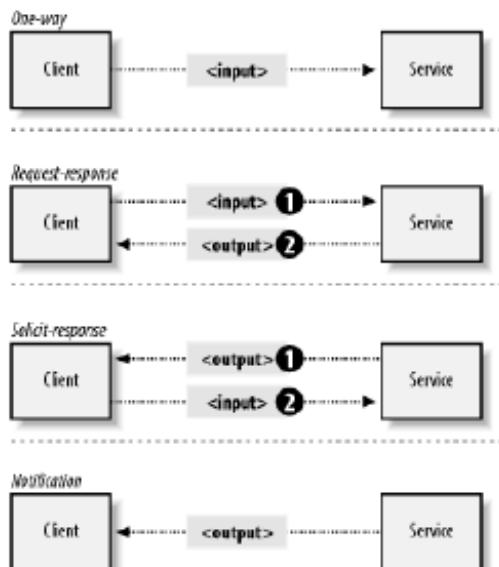


Synchronisation ▾

Synchronisation BEZEICHNET DAS ZEITLICHE ABGLEICHEN VON TECHNISCHEN **Vorgängen**.

► Erklärung: Synchronisation eines Protokolls ▾

- DIE **Synchronisation** EINES PROTOkolls BESCHREIBT IN WELCHER ABFOLGE DIE **Kommunikationsparteien** MITEINANDER KOMMUNIZIEREN.
- JE NACH PROTOkoll WERDEN UNTERSCHIEDLICHE FORMEN DER **Synchronisation** UNTERSTÜTZT.



9.2. Netzwerkprotokolle

Ein **Netzwerkprotokoll** ist ein Kommunikationsprotokoll für den Austausch von **Daten** unter **Prozessen**, die in einem Netzwerk miteinander verbunden sind.

9.2.1 Grundlagen

Ein **Netzwerkprotokoll** besteht aus einem **Satz** von Regeln und Formaten, die das Kommunikationsverhalten der kommunizierenden **Instanzen** bestimmt.

Netzwerkprotokolle legen eine mehrere der folgenden Vorgehensweisen fest.

► Auflistung: Regeln ▾

- DEFINITION DER ZUGRUNDELIEGENDEN **physikalischen Verbindung**.²³
- FESTLEGEN DES **Formats** DER NACHRICHTEN. DEFINITION WIE EINE NACHRICHT BEGINNT ODER ENDET.
- DEFINITION DER **Semantik** EINER NACHRICHT.
- DEFINITION WIE DER UNERWARTETE **Verlust** DER VERBINDUNG FESTGESTELLT WIRD UND WAS DANN ZU GESCHEHEN HAT.



9.2.2 Internetprotokolle

Der Austausch von **Nachrichten** erfordert häufig ein Zusammenspiel verschiedener Protokolle, die unterschiedliche Aufgaben übernehmen.

► Erklärung: Internetprotokolle ▾

- DER AUSTAUSCH VON **Nachrichten** IM **Netzwerk** ERFORDERT HÄUFIG EIN ZUSAMMENSPIEL VERSCHIEDENER **PROTOKOLLE**, DIE UNTERSCHIEDLICHE AUFGABEN ÜBERNEHMEN.
- UM DIE DAMIT VERBUNDENE **Komplexität** BEHERRSCHEN ZU KÖNNEN, WERDEN DIE EINZELNEN PROTOKOLLE IN **Schichten** ORGANISIERT.
- IM RAHMEN DES **Schichtenmodells** GEHÖRT JEDES DER INTERNETPROTOKOLLE EINER BESTIMMTEN SCHICHT AN UND IST FÜR DIE ERLEDIGUNG EINER SPEZIELLEN AUFGABE VERANTWORTLICH.

²³ LAN oder WLAN

- Protokolle höherer Schichten verwenden dabei die Dienste von Protokollen tieferer Schichten.
- Zusammen bilden die so strukturierten Protokolle einen Protokollstapel.

Nr.	Schicht	Beispiele
4	Anwendung	HTTP, FTP, SMTP, POP, Telnet, SOCKS
3	Transport	TCP, UDP
2	Internet	IP (IPv4, IPv6)
1	Netzzugang	Ethernet, Token Bus, Token Ring, FDDI

9.2.3 Schichten der Protokollstapels

► Auflistung: Schichten □

- **Anwendungsschicht:** Die Anwendungsschicht umfasst alle Protokolle, die mit **Anwendungsprogrammen** zusammenarbeiten und die Netzwerkinfrastruktur für den **Austausch** Anwendungsspezifischer Daten nutzen.
- **Transportschicht:** Die **Transportschicht** ermöglicht eine Kommunikation zwischen Netzwerkknoten.

Das wichtigste Protokoll dieser Schicht ist das **TCP Protokoll**, das Verbindungen zwischen zwei Netzwerkteilnehmern zum zuverlässigen Versenden von Datenströmen herstellt.

- **Internetschicht:** Die **Internetschicht** ist für die **Weitervermittlung** von Datenpaketen und das Routing zuständig.

Die Aufgabe dieser Schicht ist es, zu einem empfangenen Paket das nächste Zwischenziel zu ermitteln und das Paket dorthin weiterzuleiten.

- **Netzzugangsschicht:** Die **Netzzugangsschicht** implementiert Protokolle für die **Datenübertragung** von Punkt zu Punkt zu verstehen.

9.3. Http Protokoll ▾



Http Protokoll ▾

Das **Http Protokoll** ist ein Internetprotokoll zur Übertragung von Daten in einem Netzwerk.

Das **Http Protokoll** wird hauptsächlich eingesetzt um **Html Dokumente** aus dem Internet in einen Webbrowser zu laden.

9.3.1 Kommunikationsmuster

Die **Kommunikation** mit dem **Http Protokoll** folgt dem Client Server Muster.

► Erklärung: Kommunikationsmuster ▾

- Der **Http Client** sendet eine Anfrage - **Http Request** - an den **Http Server**.
- Dieser bearbeitet die Anfrage und schickt seine Antwort - **Http Response** - zurück. Nach der Antwort mit dem Server wird die Verbindung beendet.
- Die **Kommunikation** zwischen Client und Server findet auf Basis von **Nachrichten** im Text Format statt. Die Nachrichten werden standardmäßig über TCP auf dem Port 80 verschickt.

9.3.2 Http Nachrichten

Für die Kommunikation im Http Protokoll tauschen Client und Server Nachrichten aus.

► Erklärung: Http Nachrichten ▾

- Die Nachrichten die zwischen Client und Server verschickt werden, werden als **Http Request** und **Http Response** bezeichnet.
- **Http Nachrichten** bestehen immer aus einem **Header** und den eigentlichen Nachrichtendaten.
- Der **Header** enthält dabei **Steuerinformationen** die in Schlüssel Werte Paaren gespeichert werden.

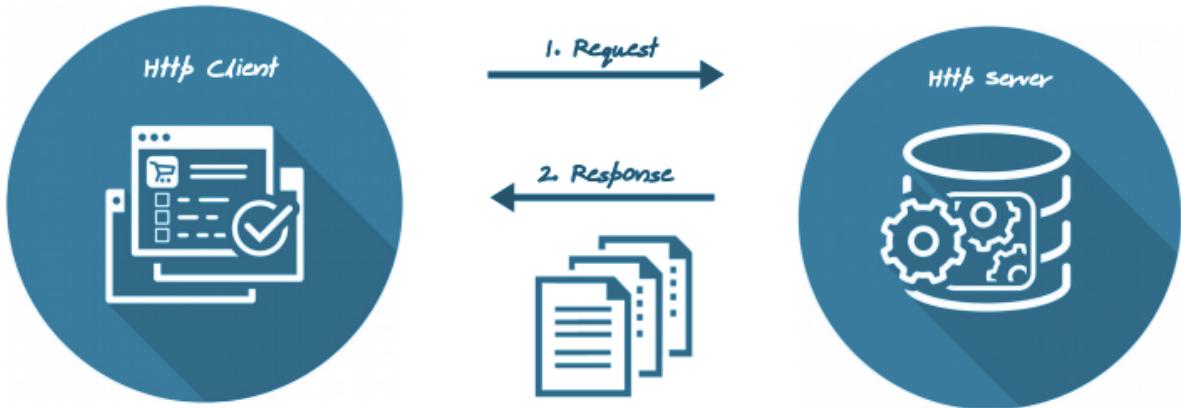
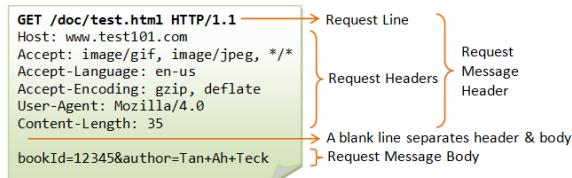


Abbildung 13. Http Protokoll

9.4. Http Request

Ein **Http Request** ist die Anfrage eines Http Clients an einen **Http Server**.



9.4.1 Anatomie eines Http Requests

Jeder **Http Request** besteht aus 2 Teilen: Dem Nachrichtenkopf - **Message Header** - zum Speichern von Metainformation und dem Nachrichtenkörper - **Message Body** - zum Transportieren der eigentlichen Information.

Erklärung: Http Request

- JEDER **Request** SPEZIFIZIERT DURCH DIE ANGABE EINER **Methode**, AUF WELCHE WEISE DER SERVER DIE **Nachricht** VERARBEITEN SOLL.
- DER **Message Header** ENTHÄLT Metainformationen FÜR DIE VERARBEITUNG DES REQUESTS.

9.4.2 Request Methoden

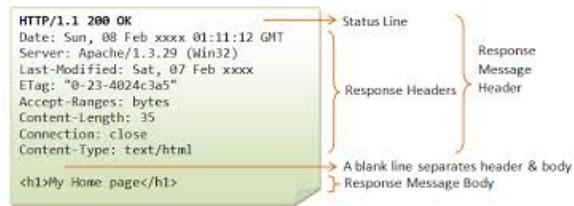
Auflistung: Request Methoden

- Get Methode:** MIT DER **get Methode** WERDEN **Ressourcen** VOM SERVER ANGEFORDERT. DIE ANGEFORDERTE RESSOURCE WIRD IN DER ANFRAGE ÜBER IHRE **url** EINDEUTIG GEKENNZICHNET.
- Post Methode:** MIT HILFE DER **post Methode** WERDEN DATEN AN DEN **Server** ÜBERTRAGEN. AUCH HIER WIRD WIEDER EINE URL ANGEgeben, UM DIE DATEN AN EINE EINDEUTIG IDENTIFIZIERBARE RESSOURCE ZUSCHICKEN.
- Put Methode:** MIT HILFE DER **put Methode** WERDEN NEUE **Ressourcen** AUF DEM SERVER ANGELEGT BZW. MODIFIZIERT. DEN PFAD DER RESSOURCE WIRD MIT HILFE EINER URL SPEZIFIZIERT.
- Delete Methode:** DIE **delete Methode** DIENT ZUM **Löschen** EINER AUF DEM WEBSERVER GESPEICHERTEN RESSOURCEN.
- Head Methode:** DURCH DEN AUFRUF DER **head Methode** WERDEN DIESELBN ERGEBNISSE ERZIELT WIE BEIM AUFRUF DER GET METHODE. ES WERDEN JEDOCH KEINE **Ressourcen** ÜBERTRAGEN SONDERN NUR **Statusinformationen**²⁴ VERSCHICKT.

²⁴ Durch die Head Methode lässt sich somit sehr leicht die Existenz von Links überprüfen oder feststellen, ob Seiten immer noch auf einem Webserver existieren.

9.5. Http Response

Als Antwort auf einen **Http Request** sendet der Sender einen **Http Response**.



9.5.1 Anatomie eines Http Response

Die erste Zeile des **Http Response** wird als **Statuszeile** bezeichnet. Die Statuszeile informiert den Client über den Erfolg der Verarbeitung des Http Requests.

► Erklärung: Statuszeile ▾

- IN DER STATUSZEILE WIRD DER **Statuscode** UND DIE VERWENDETE HTTP VERSION DEKLARIERT.
- **Statuscode:** DER **Statuscode** WIRD VOM SERVER ALS TEIL DER ANTWORT AUF DIE HTTP ANFRAGE GELIEFERT.

Der Server teilt durch den **Statuscode** dem Client mit, ob die Anfrage erfolgreich bearbeitet wurde. Im **Fehlerfall** gibt der Statuscode Auskunft darüber, wo er die gewünschten Informationen erhalten kann. Der Statuscode selbst wird als 3 stellige Ziffer angegeben.

- **Klartextmeldung:** DIE **Klartextmeldung** STELLT EINE VERBALE BESCHREIBUNG DES **Statuscodes** DAR. KLARTEXTMELDUNGEN SIND EINDEUTIG EINEM STATUSCODE ZUGETEILT.



9.5.2 Status Codes

Der **Statuscode** ist eine 3 stellige Nummer. Die erste der 3 Stellen des Zahlencodes wird die **Gruppe** des **Statuscodes** angezeigt. Die restlichen Stellen sind für die Differenzierung innerhalb der **Gruppe** verantwortlich.

► Erklärung: Statuscodegruppen ▾

- **1xx - Status:** DER REQUEST BEFINDET SICH NOCH IN BEARBEITUNG.
- **2xx - Erfolgreiche Verarbeitung:** - DER REQUEST KONNTE VOM SERVER ERFOLGREICH VERARBEITET WERDEN.
- **3xx - Umleitung:** UM EINE ERFOLGREICHE BEARBEITUNG DER ANFRAGE SICHERZUSTELLEN, SIND WEITER SCHritte SEITENS DES CLIENTS ERFORDERLICH.
- **4xx Client Fehler:** - DIR URSCHE FÜR DAS SCHEITERN DES CLIENTS LIEGT IM VERANTWORTUNGSBEREICH DES CLIENTS.
- **5xx Server Fehler:** - DIE URSCHE FÜR DAS SCHEITERN DER ANFRAGE LIEGT IM VERANTWORTUNGSBEREICH DES SERVERS.



Code	Bedeutung	Nachricht
100	Die laufende Anfrage an den Server wurde noch nicht zurückgewiesen.	Continue
101	Wird verwendet, wenn der Server eine Anfrage mit gesetztem „Upgrade“-Header-Feld empfangen hat und mit dem Wechsel zu einem anderen Protokoll einverstanden ist. Anwendung findet dieser Status-Code beispielsweise im Wechsel von HTTP zu WebSocket.	Switching Protocols
200	Die Anfrage wurde erfolgreich bearbeitet und das Ergebnis der Anfrage wird in der Antwort übertragen.	Ok
201	Die Anfrage wurde erfolgreich bearbeitet. Die angeforderte Ressource wurde vor dem Senden der Antwort erstellt. Das „Location“-Header-Feld enthält eventuell die Adresse der erstellten Ressource.	Created
202	Die Anfrage wurde akzeptiert, wird aber zu einem späteren Zeitpunkt ausgeführt. Das Gelingen der Anfrage kann nicht garantiert werden.	Accepted
400	Die Anfrage-Nachricht war fehlerhaft aufgebaut.	Bad Request
401	Die Anfrage kann nicht ohne gültige Authentifizierung durchgeführt werden. Wie die Authentifizierung durchgeführt werden soll, wird im „WWW-Authenticate“-Header-Feld der Antwort übermittelt.	Unauthorized
403	Die Anfrage wurde mangels Berechtigung des Clients nicht durchgeführt, bspw. weil der authentifizierte Benutzer nicht berechtigt ist, oder eine als HTTPS konfigurierte URL nur mit HTTP aufgerufen wurde.	Forbidden
404	Die angeforderte Ressource wurde nicht gefunden. Dieser Statuscode kann ebenfalls verwendet werden, um eine Anfrage ohne näheren Grund abzuweisen. Links, welche auf solche Fehlerseiten verweisen, werden auch als Tote Links bezeichnet.	Not Found
405	Die Anfrage darf nur mit anderen HTTP-Methoden (zum Beispiel GET statt POST) gestellt werden. Gültige Methoden für die betreffende Ressource werden im „Allow“-Header-Feld der Antwort übermittelt.	Method Not Allowed
500	Dies ist ein „Sammel-Statuscode“ für unerwartete Serverfehler.	Internal Server Error
504	Der Server konnte seine Funktion als Gateway oder Proxy nicht erfüllen, weil er innerhalb einer festgelegten Zeitspanne keine Antwort von seinerseits benutzten Servern oder Diensten erhalten hat.	Gateway Timeout
501	Die Funktionalität, um die Anfrage zu bearbeiten, wird von diesem Server nicht bereitgestellt. Ursache ist zum Beispiel eine unbekannte oder nicht unterstützte HTTP-Methode.	Not Implemented
511	Der Client muss sich zuerst authentifizieren um Zugang zum Netzwerk zu erhalten.	Network Authentication Required

Abbildung 14. Http Statuscode

10. Kommunikationsprotokoll: Rest

02

REST

01. Rest Prinzipien	65
02. Ressourcen	68
03. Http Methoden	69
04. Fallbeispiel: Ordermanager	70

10.1. REST Grundprinzipien ▾

Rest für sich ist kein eigenes Kommunikationsprotokoll. Vielmehr ist es eine Sammlung von **Regeln** und Leitsätzen die für die Kommunikation über **http** formuliert werden.

10.1.1 Rest Grundprinzipien

Rest formuliert 5 Grundprinzipien für die Kommunikation:

► Erklärung: **REST Grundprinzipien** ▾

- EINDEUTIGE **Identifikation** VON RESSOURCEN
- UNTERSCHIEDLICHE **Repräsentation** VON RESSOURCEN
- VERWENDUNG VON **Hypermedia**
- VERWENDUNG VON **Http Standardmethoden**



10.1.2 Eindeutige Identification von Ressourcen

Aus der Sicht von Rest ist das Internet eine Sammlung von Ressourcen.²⁵

► Erklärung: **Beispiele für Ressourcen** ▾

- URI EINES YouTube Videos.
- URI EINES Amazon Produktes.

Jede Ressource braucht zur **Identification** einen eindeutigen **Schlüssel**.

Sollen **Ressource** im Web **identifiziert** werden, werden dafür **URIs** verwendet. Der Umkehrschluss gilt jedoch nicht: Eine **Ressource** kann auch unter mehreren URI gefunden werden.



uri ▾

EINE **URI** IST EIN **Identifikator** FÜR **Webressourcen**. IM INTERNET BILDEN URIS EINE GLOBALEN **Namensraum**.



²⁵ Ressourcen sind die semantischen Objekte aus denen das Internet besteht

10.1.3 Repräsentationen von Ressourcen

Arbeiten wir mit einer **REST Anwendung** sehen wir nie die **Ressource** selbst, sondern stets nur eine ihrer **Repräsentationen**

► Erklärung: Beispiele ▾

- IN **REST** WERDEN **Ressourcen** ÜBER EINE **Id** IDENTIFIZIERT.
- IM HTTP REQUEST WIRD ANGEgeben, IN WELCHEM FORMAT DIE **Daten** ÜBERTRAGEN WERDEN SOLLEN.
- DAS GEWÄHLTE **Datenformat** WIRD AUCH ALS DIE REPRÄSENTATION DER RESSOURCE BEZEICHNET.

► Codebeispiel: Repräsentationen ▾

```

1  <!-- ----- -->
2  <!--          XML Repraesentation      -->
3  <!-- ----- -->
4  <?xml version="2.0"?>
5  <person>
6    <surname>Kennedy</surname>
7    <middlename>Fitzgerald</middlename>
8    <givenname>John</givnename>
9  </person>
10 
11 <!-- ----- -->
12 <!--          HTML Representation     -->
13 <!-- ----- -->
14 <!DOCTYPE html>
15 <html lang="en">
16   <head>
17     <meta charset="UTF-8">
18     <title>Personen Daten</title>
19     <script src="js/demo.js"></script>
20   </head>
21   <body>
22     ...
23     <div id="personContainer">
24       <table>
25         <tr>
26           <td>Surname:</td>
27           <td>#{person.surname}</td>
28         </tr>
29         <tr>
30           <td>Givenname:</td>
31           <td>#{person.givenname}</td>
32         </tr>
33       </table>
34     </div>
35   </body>
36 </html>
```

10.1.4 Hypermedia - Hateos



Hypermedia ▾

Hypermedia BEZEICHNET EINE NICHTLINEARE FORM VON MEDIEN, DEREN HAUPTCHARAKTERISTIKUM DAS **Verlinken** VON DATEN MIT **Hyperlinks** IST.

Das bekannteste Beispiel für **Hypermedia** ist **Html**.

► Erklärung: hateos ▾

- **Hateos** IST EIN ACRONYM UND STEHT FÜR **Hypermedia as the engine of application state**.
- MIT **engine** IST IM KONTEXT VON REST DIE **Zustandsverwaltung** VON RESSOURCEN GEMEINT.
- OBJEKTE UND DAMIT **REST Ressourcen** KÖNNEN IN UNTERSCHIEDLICHEN **Zuständen** VORLIEGEN. DIE ZUSTÄNDE VON RESSOURCEN WERDEN MIT DER HILFE VON **Zustandsdiagrammen** VERWALTET.
- BEI REST IST DER **Zustand** EINER **Ressource** VOLLSTÄNDIG IN SEINER **Repräsentation** ENTHALTEN.
- REPRÄSENTAIONEN KÖNNEN MÖGLICHE **Zustandsübergänge** FÜR RESSOURCEN IN FORM VON **Links** AN **Clients** WEITERGEBEN.

► Erklärung: Zustandsdiagramm²⁶ ▾

- EIN **Zustandsdiagramm**²⁷ BESTEHT AUS **Knoten**, DIE DURCH **Kanten** MITEINANDER VERBUNDEN SEIN KÖNNEN.
- DIE **Knoten** REPRÄSENTIEREN DIE MÖGLICHEN **Zustände** DER RESSOURCE. DIE **Kanten** REPRÄSENTIEREN DIE **Zustandsübergänge**.
- EINE **Kante** FÜHRT DEMZUFOLGE VOM **Ausgangszustand** ZUM **Folgezustand**.
- **Repräsentation** VON **Ressourcen** GEBEN DIE MÖGLICHEN **Zustandsübergänge** VON RESSOURCEN IN FORM VON **Links** AN DEN **CLIENT** WEITER.



²⁶ *Statemachine*

²⁷ *deterministischer Automat*

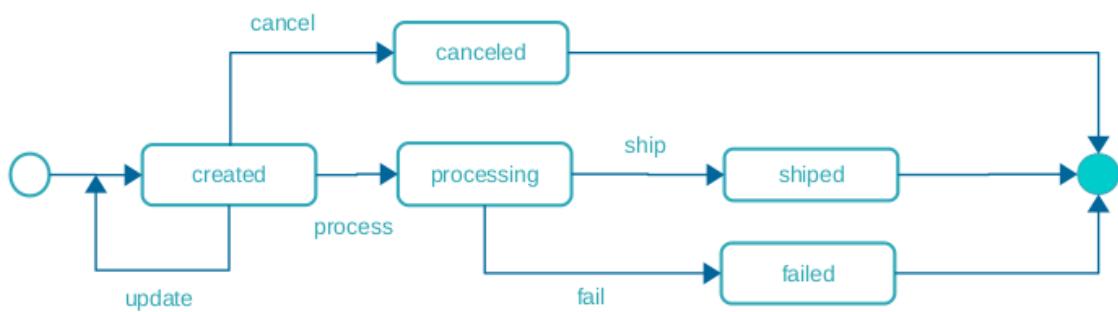


Abbildung 15. State Machine: Produktbestellung

▶ Codebeispiel: Produktbestellung ▾

```

1 //-----
2 //          JSON - Order
3 //-----
4 var order = {
5     "link" : {
6         "href": "http://..../orders/21",
7         "rel": "self"
8     },
9     "status"      : "created",
10    "date"        : "2014-10-03",
11    "total"       : 2090,
12    "updated"     : "2015-01-11",
13    "user": {
14        "link" : {
15            "href": "http://..../users/14",
16            "rel": "self"
17        },
18        "user-name": "Tim"
19    },
20    "actions": [
21        {
22            "href": "http://example.at/orders/34/cancel",
23            "description": "cancel order"
24        },
25        {
26            "href": "http://example.at/orders/34/process",
27            "description": "process order"
28        },
29        {
30            "href": "http://example.at/orders/34/update",
31            "description": "update order"
32        }
33    ],
34    "products" : [
35        {
36            "link" : {
37                "href": "http://..../orders/21/products/47",
38                "rel": "self"
39            },
40            "name"   : "Der Gladiator",
41            "category": "DVD"
42        }
43    ]
44}

```

10.1.5 HTTP Methoden

Rest wird für die Verwaltung von **Ressourcen** eingesetzt. Jede **Ressource** stellt eine **Schnittstelle** zu Verwaltungszwecken zur Verfügung.

Zur Verwaltung von Ressourcen werden die vom http Protokoll definierten Methoden verwendet.

▶ Codebeispiel: Schnittstelle einer Ressource ▾

```

1 //-----
2 //      interface: IHttpRessource
3 //-----
4 public interface IHttpRessource{
5     HttpResponse get();
6
7     HttpResponse put();
8
9     HttpResponse post();
10
11    HttpResponse patch();
12
13    HttpResponse delete();
14
15    HttpResponse options();
16
17
18 public interface IProduct extends IHttpRessource{
19
20        void update();
21
22        void cancel();
23
24        void process();
25
26        void ship();
27
28        void fail();
29
30    }

```

10.2. Ressourcen

Werden die **Ressourcen** für eine Anwendung entworfen, wird damit die Schnittstelle der Anwendung definiert.

10.2.1 Arten von Ressourcen

Rest unterscheidet mehrere **Kategorien** von **Ressourcen**.

► Auflistung: Ressourcen Kategorien ▾

- Primärressourcen
- Subressourcen
- Listenressourcen

10.2.2 Primärressourcen

Wird die fachliche **Domäne**²⁸ einer Anwendung analysiert, kann einfach eine Reihe von **Ressourcen** identifiziert werden.



Primärressource ▾

Mit **Primärressourcen** sind jene Ressourcen gemeint, die sich in der Regel beim klassischen Anwendungsentwurf sehr früh als Kandidaten für **Entitäten** ergeben.

Die **Repräsentation** einer **Primärressource** enthält die gesamten **Daten** der Ressource.

► Codebeispiel: Primärressource ▾

```

1 //-----
2 //          JSON - Project
3 //-----
4 var project = {
5   "href" : "http://example.com/rest/projects/1",
6   "title" : "Finite Methods in engine development
              simulation",
7   "description" : "The finite element method
                  (FEM) is a numerical method for",
8   "creation-date" : "10.09.2022"
9 }
```

10.2.3 Subressourcen



Subressource ▾

Ressourcen, die Teil einer anderen Ressource sind, bezeichnen wir als **Subressourcen**.

► Codebeispiel: Subressource ▾

```

1 //-----
2 //          JSON - Order
3 //-----
4 var project = {
5   "href" : "http://example.com/rest/projects/1",
6   "title" : "Finite Methods in engine development
              simulation",
7   "description" : "The finite element method
                  (FEM) is a numerical method for",
8   "creation-date" : "10.09.2022",
9
10  //subressource von project
11  "project-leader" : {
12    "href" :
13      "http://example.com/rest/staff/23",
14      "name" : "Schuettli"
15  }
```

10.2.4 Listenressourcen

Für die meisten **Primärressourcen** gibt es in der Regel auch eine zugehörige **Listenressource**: Die Menge aller Kunden, alle Buchungen usw. Da diese ebenfalls eigene Ressourcen sind, bedeutet das, dass auch sie eindeutig **identifiziert** werden und möglicherweise mehr als eine Repräsentation haben.

Listenressourcen setzen sich zusammen aus einer Liste von **Subressourcen**.

► Codebeispiel: Listenressourcen ▾

```

1 //JSON Projekt Listenressourcen Repraesentation
2 var projects ={
3   "href" : "http://example.com/rest/projects",
4
5   [
6     {
7       "href" : "http://example.com/rest/projects/1"
8       "title" : "Finite Methods in engine
                   development simulation"
9     },
10    {
11      "href" : "http://example.com/rest/projects/2"
12      "title" : "cloud systems research"
13    },
14    ...
15  ]}
```

²⁸ das Umfeld für das wir die Anwendung entwerfen

► Erklärung: Typen von Listenressourcen ▾

- **Paginierung** VON WEBBERFLÄCHE SIND WIR ES GEWOHNT, **SUCHERGEBNISSE SEITENWEISE** PRÄSENTIERT ZU BEKOMMEN.

Das ist in **ROA Anwendungen** nicht anders. Auch hier möchten wir die Daten nicht im Ganzen an den Anwender schicken. Die Paginierung wird über die URI der Ressource gesteuert.

Uri: Listenressourcen

🔗 .../projects?start=0&count=20

- **Filter:** Um Listenressourcen nach bestimmten **Kriterien** zusammenzustellen werden **Filter** eingesetzt. enthalten.

z.B.: DIE LISTE ALLER KUNDEN AUS DER **Region Nord** ODER ALLE PRODUKTE DEREN BEZEICHNUNG MIT EINEM A BEGINNT

Uri: Listenressourcen

🔗 .../customers?region=Nord

🔗 .../products?name=a*

10.3. Http Methoden

10.3.1 Httpmethode - Get

Get ist die wohl wichtigste und am häufigste verwendete **http Methode**.

► Erklärung: Get Methode ▾

- DIE **get** METHODE GIBT DIE REPRÄSENTATION EINER **Ressource** ZURÜCK. SIE IST DIE WICHTIGSTE **Leseoperation** DES HTTP PROTOKOLLS.

10.3.2 Httpmethode - Put



put Methode ▾

- **Anlegen von Ressourcen**
- **Ändern von Ressourcen**

► Erklärung: Eigenschaften von put ▾

- DIE **Put** METHODE IST DAS GEGENSTÜCK ZUR **Get** METHODE.

10.3.3 Httpmethode - Post



post Methode ▾

DIE **post** METHODE WIRD IN ERSTER LINIE VERWENDET UM **Ressourcen anzulegen**.

- JEDOCH SEHEN WIR AUCH DIE VERWENDUNG VON **POST** ZUM **Verändern** VON **Ressoucen** IN MANCHEN ANWENDUNGEN.

Weil **Post** keine **semantischen Garantien** erfüllen muss, wird es in der Regel zum Anstoß von beliebigen Operationen in einer **Rest Anwendung** eingesetzt.

► Erklärung: post vs put ▾

- **put** UND **post** KÖNNEN BEIDE VERWENDET UM **Ressourcen anzulegen**.

- DER **Unterschied** LIEGT IN DER ART WIE DIE URI AN DIE **http Operation** ÜBERGEBEN WIRD:

- **post:** BEI **post** WIRD DIE RESSOURCE UNTER EINER VOM **Server** GEWÄHLTEN **Uri** ABGELEGT.
- **put:** BEI **put** BESTIMMT DER **Client** DIE **Uri**.

10.3.4 Httpmethode - Delete



delete Methode ▾

DIE **Delete Methode** WIRD VERWENDET UM **Ressourcen** ZU LÖSCHEN. DIE METHODE IST **idempotent**.

10.3.5 Httpmethode - Patch

Die **patch** Methode wird verwendet um einzelne Teile einer **Ressource** zu ändern.

► Erklärung: patch vs put ▾

- **patch** UND **put** WERDEN BEIDE VERWENDET UM DATEN ZU ÄNDERN.
- DER **Unterschied** LIEGT IM AUSMASS DER ÄNDERUNG:
 - ▶ **patch:** MIT DER **patch** METHODE WIRD EIN **Teil** DER Ressource VERÄNDERT.
 - ▶ **put:** MIT DER **put** METHODE WIRD DIE gesamte RESSOURCE ERSETZT.
- MIT DER **patch** METHODE IST DIE INTENTION DES CLIENTS IN SEMANTISCHER HINSICHT WESENTLICH KLARER ALS MIT DER **put** METHODE.

10.3.6 Httpmethode - Options

Die Methode liefert die möglichen **Kommunikationsoptionen** einer **Ressource**.

► Erklärung: Eigenschaften von options ▾

- DIE **options** METHODE GIBT FÜR EINE **Ressource** ALLE MÖGLICHEN **http Methoden** AN DIE DIE **Ressource** UNTERSTÜTZT.
- DIE **options** METHODE IST **idempotent**.



10.4. Web Api Entwicklung - Fallbeispiel Ordermanager

Beim Entwurf einer **Web Api** haben wir die Aufgabe, passende **Ressourcen** zu finden und geeignete **URIs** festzulegen.

10.4.1 Ressourcen einer Anwendung

Prinzipiell kann jedes Objekt, das Ziel eines **Link** sein könnte, einer Ressource sein.

► Erklärung: Ressourcen des Ordermanagers ▾

- **User, Order, Produkt, Review**
- **Liste von Produkten**
- **Liste von Reviews**

10.4.2 Kategorien von Ressourcen

Als nächstes müssen wir die **Ressourcen** in **Kategorien** unterteilen.

► Analyse: Primärressourcen ▾

- **User**
- **Produkt**

► Analyse: Subressourcen ▾

- **Subressourcen** STEHEN IMMER IN ABHÄNGIGKEIT ZU EINER **Primärressource**.
- IM GEGENSATZ ZU **Primärressourcen** UNTERSTÜTZEN **Subressourcen** NUR EINE EINGESCHRÄNKTE AUSWAHL VON METHODEN.
- **Review**
- **Bild**

► Analyse: Listenressourcen ▾

- **Liste von Produkten**
- **Liste von Reviews**

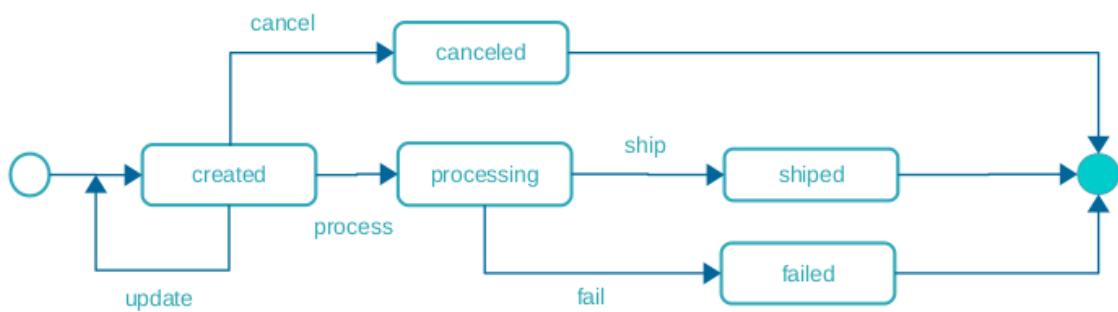


Abbildung 16. Zustände einer Bestellung

10.4.3 Repräsentationen von Ressourcen

Codebeispiel: XML Repräsentationen User ▾

```

1  <!-- ----- -->
2  <!--      XML Repräsentation von User      -->
3  <!-- ----- -->
4  <?xml version="2.0"?>
5  <user>
6      <!-- Jede Repräsentation muss eine Link auf
7          sich selbst enthalten           -->
8      <href rel="self">
9          http://example.at:8082/ordermanager/users/32
10     </href>
11
12     <first-name>Franz</first-name>
13     <middle-name>Josef</middle-name>
14     <last-name>Kurz</last-name>
15
16     <user-name>shorty134</user-name>
17     <joined-at>2011-11-11</joined-at>
18 </user>
  
```

Codebeispiel: Json Repräsentationen User ▾

```

1  //-----
2  //      Json Repräsentation von User
3  //-----
4  var user = {
5      //Link auf sich selbst
6      "link": {
7          "href": "http://.../ordermanager/users/32",
8          "rel" : "self"
9      }
10
11     "first-name" :"Franz",
12     "middle-name" :"Josef",
13     "last-name" :"Kurz",
14     "user-name" :"shorty134"
15     "joined-at" :"2011-11-11"
16 }
  
```

Codebeispiel: Repräsentation Produkt ▾

```

1  //-----
2  //      Json Repräsentation von Product
3  //-----
4  var product = {
5      //Link auf sich selbst
6      "link": {
7          "href": "http://.../ordermanager/products/1",
8          "rel": "self"
9      },
10
11     //Stammdaten
12     "name": "Forbidden Stars",
13     "search-terms": ["toy", "boardgame", "40k",
14         "forbidden stars"],
15     "description": "forbiddenstars is a boardgame
16         ...",
17     "price": 99,
18
19     //Subressource
20     //Listenressource
21     "reviews": {
22         "link": {
23             "href": "http://.../users/32/reviews",
24             "rel": "self"
25         },
26         "entries": [
27             {user: {
28                 "link": {
29                     "href": "http://.../users/32",
30                     "rel": "self"
31                 },
32                 "user-name": "self"
33             },
34             "text": "Das ist ein tolles ..."
35         },
36     }
  
```

► Codebeispiel: Einzelne Bestellung ▾

```

1 //-----
2 //      Json Repraesentatin von Order
3 //-----
4 var order = {
5     "link" : {
6         "href":"http://..../orders/21",
7         "rel":"self"
8     },
9
10    "status"       : "processing",
11    "date"         : "2014-10-03",
12    "total"        : 2090,
13    "updated"      : "2015-01-11",
14
15
16    //subressource
17    "user": {
18        "link" : {
19            "href":"http://..../users/14",
20            "rel":"self"
21        },
22        "user-name": "Tim"
23    },
24
25    //subressource
26    //listressoruce
27    "items": [
28        {
29            "quantity": 1,
30            "product" :{
31                "link" : {
32                    "href":"http://..../products/10",
33                    "rel":"self"
34                },
35                "description": "Laptop X65",
36                "price"      : 799
37            },
38            "quantity": 4,
39            "product":{
40                "link" : {
41                    "href":"http://..../products/13",
42                    "rel":"self"
43                },
44                "description": "MP3 Player",
45                "price": 99
46            }
47        }
48    ]
49}

```

► Codebeispiel: Liste aller Bestellungen ▾

```

1 //-----
2 //      Json Repraesentatin von orders
3 //-----
4 var orders = {
5     "link" : {
6         "href":"http://..../orders?start=1&count=10",
7         "rel":"self"
8     },
9
10    "orders" : [

```

10.4.4 Analyse einer Repräsentation

► Codebeispiel: Order Repräsentation ▾

```

1  var order = {
2      "link" : {
3          "href": "http://..../orders/21",
4          "rel": "self"
5      },
6
7      "status"      : "processing",
8      "date"        : "2014-10-03",
9      "total"       : 2090,
10     "updated"    : "2015-01-11",
11
12
13     //subressource
14     "user": {
15         "link" : {
16             "href": "http://..../users/14",
17             "rel": "self"
18         },
19         "user-name": "Tim"
20     },
21
22     //subressource
23     //listressoruce
24     "items": [
25         {
26             "quantity": 1,
27             "product" : {
28                 "link" : {
29                     "href": "http://..../products/10",
30                     "rel": "self"
31                 },
32                 "description": "Laptop X65",
33                 "price"     : 799
34             },
35             "quantity": 4,
36             "product": {
37                 "link" : {
38                     "href": "http://..../products/13",
39                     "rel": "self"
40                 },
41                 "description": "MP3 Player",
42                 "price": 99
43             }
44         ],
45
46         "actions": [
47             {
48                 "link" : {
49                     "href": "http://..../orders/21/cancel",
50                     "rel": "self"
51                 },
52                 "description": "cancel order"
53             },
54             {
55                 "link" : {
56                     "href": "http://..../orders/21/process",
57                     "rel": "self"
58                 },
59                 "description": "process order"
60             },
61             {
62                 "link" : {
63                     "href": "http://..../orders/21/update",
64                     "rel": "self"
65                 }
66             }
67         ]
68     }
69 }

```

```

60             "rel": "self"
61         },
62         "description": "update order"
63     }]
64
65 }

```

□

► Analyse: Order Repräsentation ▾

- **Hypermedia:** DER Link AUF SICH SELBST IST STETS TEIL DER REPRÄSENTATION DER RESSOURCE.

Nur so kann die Idee von Hypermedia erfolgreich umgesetzt werden.

```

1  var order = {
2      "link" : {
3          "href": "http://..../orders/21",
4          "rel": "self"
5      },
6      ..
7      ..
8  }

```

- Stammdaten der Ressource:

```

1  var order = {
2      "link" : {
3          "href": "http://..../orders/21",
4          "rel": "self"
5      },
6
7      "status"      : "created",
8      "date"        : "2014-10-03",
9      "total"       : 2090,
10     "updated"    : "2015-01-11",
11
12     ..
13 }

```

- Repräsentation von Subressource:

```

1  var order = {
2      "link" : {
3          "href": "http://..../orders/21",
4          "rel": "self"
5      },
6      ..
7      ..
8
9      //subressource
10     "user": {
11         "link" : {
12             "href": "http://..../users/14",
13             "rel": "self"
14         },
15         "user-name": "Tim"
16     },
17
18     //subressource

```

```

19     //listressource
20     "items": [
21         {
22             "quantity": 1,
23             "product" :{
24                 "link" : {
25                     "href": "http://..../products/10",
26                     "rel": "self"
27                 },
28                 "description": "Laptop X65",
29                 "price" : 799
30             },
31             "quantity": 4,
32             "product": {
33                 "link" : {
34                     "href": "http://..../products/13",
35                     "rel": "self"
36                 },
37                 "description": "MP3 Player",
38                 "price": 99
39             }
40         }
41     ]
42 }
```

□

- **Zustand einer Ressource** EINE Bestellung DURCHLÄUFT EINE REIHE VON **Statusübergängen**, DIE IM OBEREN DIAGRAMM DARGESTELLT WERDEN.

Der Status einer Bestellung ist dabei ein Teil der Bestelldaten.

```

1  var order = {
2      "link" : {
3          "href": "http://..../orders/21",
4          "rel": "self"
5      },
6
7      "status" : "created",
8      ..
9      ..
10 }
```

- **Zuständigerung:** IN EINER REST ANWENDUNG IST DIE RESSOURCE SELBST FÜR DIE **Verwaltung** IHRES **Zustandes** VERANTWORTLICH.

Abhängig vom Zustand der Ressource wird eine Zahl von Methoden zur Verfügung gestellt für die Verwaltung des Zustandes.

```

1  var order = {
2      ..
3      "status" : "created",
4      ..
5      ..
6      "actions": [
7          {
8              "href": "http://..../orders/34/cancel",
9              "description": "cancel order"
10         },
11         {
12             "href": "http://..../orders/34/process",
13             "description": "process order"
14         },
15         {
16             "href": "http://..../orders/34/update",
17             "description": "update order"
18         }
19     ]
20 }
```

11. Kommunikationsprotokoll: Soap

03

SOAP Webservice

- | | |
|------------------------|----|
| 01. Soap Kommunikation | 75 |
| 02. WSDL | 76 |

11.1. SOAP Webservice Grundlagen ▾

Soap ist ein auf Xml basierendes **Kommunikationsprotokoll**. Dass Protokoll verwendet http für den Transport²⁹ seiner Nachrichten.

11.1.1 Kommunikation zwischen Servicen

Soap ist ein **Netzwerkprotokoll** das für den **Austausch** von Nachrichten zwischen Softwareservicen verwendet wird. Technisch gesehen erfolgt beim **Aufruf** eines anderen Services ein **RMI** Aufruf.



Remote Method Invocation ▾

RMI IST EINE TECHNIK ZUR REALISIERUNG VON **Interprozesskommunikation**. SIE ERMÖGLICHT DEN AUFRUF VON METHODEN VON OBJEKTS IN FREMDEN **Adressräumen**.

In einer **SOAP Anwendung** basiert die Kommunikation zwischen Servicen auf der Interaktion zwischen 3 Rollen.



► Auflistung: Rollen in einer SOAP Anwendung ▾



Service Provider ▾

Implementiert EIN SERVICE UND STELLT ES IM NETZWERK ZUR VERFÜGUNG. DAS SERVICE BESITZT EINE KLAR DEFINIERTE **Schnittstelle**.



Service Consumer ▾

DER **Service Consumer** NIMMT DEN **Dienst** EINES ANDEREN SERVICES IN ANSPRUCH.



Service Broker ▾

DER **Service Broker** FUNGIERT ALS **Vermittler** ZWISCHEN DEM SERVICE PROVIDER UND DEM SERVICE CONSUMER.

²⁹ Protokollfamilie

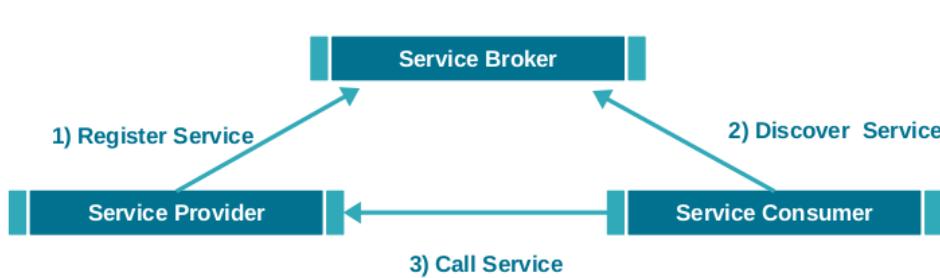


Abbildung 17. Kommunikation zwischen Servicen

11.1.2 Kommunikationsprotokoll

SOAP stützt sich auf 3 unterschiedliche XML Standards.

► Auflistung: [XML Protokolle](#) ▾

- **SOAP:** SOAP WIRD VERWENDET FÜR DEN Austausch VON DATEN ZWISCHEN Servicen.
- **WSDL:** WSDL IST EIN XML Dialekt ZUR BESCHREIBUNG DER SCHNITTSTELLEN DER WEBSERVICE.
- **UDDI:** UDDI IST EIN XML Dialect ZUM Veröffentlichen VON WEBSERVICEN.

11.1.3 Kommunikation zwischen Servicen

Der Kommunikationsablauf zwischen Servicen differenziert 3 unterschiedliche Schritte.

► Auflistung: [Kommunikationsprotokoll](#) ▾

- **1.Schritt:** DER Service Provider WIRD BEIM Service Broker³⁰ REGISTRIERT. DAMIT STEHT DAS SERVICE ANDEREN AUTHORIZIERTEN SERVICEN ZUM AUFRUF ZUR VERFÜGUNG.
- **2.Schritt:** DER Service Consumer STELLT EINE ANFRAGE BEIM Service Broker. KONNTE SICH DER SERVICE CONSUMER ERFOLGREICH AUTHENTIFIZIEREN WIRD DEM SERVICE CONSUMER DIE IP ADRESSE DES SERVICE PROVIDERS ZUGESCHICKT.
- **3.Schritt:** DER SERVICE CONSUMER KANN NUN DIE METHODEN DES SERVICE PROVIDERS AUFRUFEN.

11.2. WSDL

WSDL ist ein XML Dialekt zur Beschreibung von Webservice Schnittstellen.



11.2.1 Aufbau eines WSDL Dokuments

► Erklärung: [Struktur von WSDL](#) ▾

- **Abstrakte Teil:** DER abstrakte Teil BESCHREIBT DIE SCHNITTSTELLE DES SERVICES³¹ UNABHÄNGIG VOM TRANSPORTPROTOKOLL UND DER KONKREten IMPLEMENTIERUNG.

Struktur: abstrakter Teil

- **types:** ANGABE DER Typen DER Parameter UND Rückgabewerte DER METHODEN DER SCHNITTSTELLEN.
- **message:** ANGABE DER Parameterlisten UND RÜCKGABEWERTE DER METHODEN
- **portType:** DEFINITION DER Schnittstelle DES WEBSERVICES
- **Konkrete Teil:** DER KONKRETE TEIL DEFINIERT, WIE DAS WEBSERVICE erreichbar³² IST UND WIE DIE DATEN SERIALISIERT UND CODIERT WERDEN.

³⁰ Der Service Broker dient als Namensdienst für Service

³¹ Operationen, Datentypen

³² URI, Protokoll

► Codebeispiel: WSDL Dokument ▾

```

1  <!-- ----- -->
2  <!--          WSDL Datei          -->
3  <!-- ----- -->
4  <wsdl:definitions name="library">

5    <wsdl:types>
6      ...
7    </wsdl:types>

8    <wsdl:message>
9      ...
10   </wsdl:message>

11   <wsdl:portType>
12     ...
13   </wsdl:portType>

14   <wsdl:binding>
15     ...
16   </wsdl:binding>

17   <wsdl:service>
18     ...
19   </wsdl:service>

20 </wsdl:definitions>
```



11.2.2 WSDL Element - <wsdl:types> □

Das <wsdl:types> Element ist optional. Definieren Sie hier **xml Schemas** die für die Definition der Webbservice Schnittstelle notwendig sind.

► Codebeispiel: Typdefinition ▾

```

1  <!-- ----- -->
2  <!--          Typedefinition         -->
3  <!-- ----- -->
4  <wsdl:definitions>
5    <wsdl:types>
6      <xss:schema targetNamespace="..">
7        <xss:complexType name="book">
8          <xss:sequence>
9            <xss:element name="title"
10              type="xs:string"/>
11            <xss:element name="price"
12              type="xs:float"/>
13          </xss:sequence>
14        </xss:complexType>
15      </xss:schema>
16    </wsdl:types>
17  </wsdl:definitions>
```



11.2.3 WSDL Element - <wsdl:message> □

Verwenden Sie das <wsdl:message> Element um **Nachrichten** zu definieren. Nachrichten werden zwischen Webservicen verschickt.



► Erklärung: <wsdl:message> ▾

- **Request Nachricht:** BEI Client Anfragen ENTSPRICHET EIN WSSDL:PART EINEM Methodenparameter.

Eine Nachricht kann mehrere <wsdl:part> Elemente enthalten.

- **Response Nachricht:** BEI Service Antworten ENTSPRICHET EIN WSSDL:PART DEM Rückgabewert EINER METHODE.

► Codebeispiel: Nachrichtendefinition ▾

```

1  <!-- ----- -->
2  <!--          WSDL Definition       -->
3  <!-- ----- -->
4  <wsdl:definitions>
5    ...
6    <wsdl:types>
7      <xss:schema
8        targetNamespace="http://ex.ample.at">
9          <xss:complexType name="book">
10         <xss:sequence>
11           <xss:element name="title"
12             type="xs:string"/>
13           <xss:element name="price"
14             type="xs:float"/>
15         </xss:sequence>
16       </xss:complexType>
17     </xss:schema>
18   </wsdl:types>
19
20   <wsdl:message name="getBookTitleRequest">
21     <wsdl:part name="isbn" type="xs:string"/>
22   </wsdl:message>
23
24   <wsdl:message name="getBookTitleResponse">
25     <wsdl:part name="title" type="xs:string"/>
26   </wsdl:message>
```



11.2.4 WSDL Element - <wsdl:portType>

Verwenden Sie das **portType** Element um die Schnittstelle des Webservices zu definieren.

► Erklärung: <wsdl:portType>

- **<wsdl:operation>**: MIT DEM <WSSDL:OPERATION> ELEMENT WERDEN **Methoden** FÜR DIE SCHNITTSTELLE DEFINIERT.
- **<wsdl:input>**: DAS <WSSDL:INPUT> ELEMENT BESCHREIBT DIE **Parametern** DER Methode.
- **<wsdl:output>**: DAS <WSSDL:OUTPUT> ELEMENT BE- SCHREIBT DEN **Rückgabewert** DER Methode.

► Codebeispiel: Schnittstellendefinition

```

1  <!-- ----- -->
2  <!--          WSDL Definition          -->
3  <!-- ----- -->
4  <wsdl:definitions>
5    ...
6    <wsdl:types>
7      <xs:schema
8        targetNamespace="http://ex.example.at">
9          <xs:complexType name="book">
10         <xs:sequence>
11           <xs:element name="title"
12             type="xs:string"/>
13           <xs:element name="price"
14             type="xs:float"/>
15         </xs:sequence>
16       </xs:complexType>
17     </xs:schema>
18   </wsdl:types>
19
20   <wsdl:message name="getBookTitleRequest">
21     <wsdl:part name="isbn" type="xs:string"/>
22   </wsdl:message>
23
24
25   <wsdl:protType name="libraryPort">
26     <wsdl:operation name="getBookTitle">
27       <wsdl:input message="getBookTitleRequest"
28         name="titleRequest"/>
29       <wsdl:output message="getBookTitleResponse"
30         name="titleResponse"/>
31     </wsdl:operation>
32   </wsdl:portType>
33   ...
34   ...
35 </wsdl:definition>

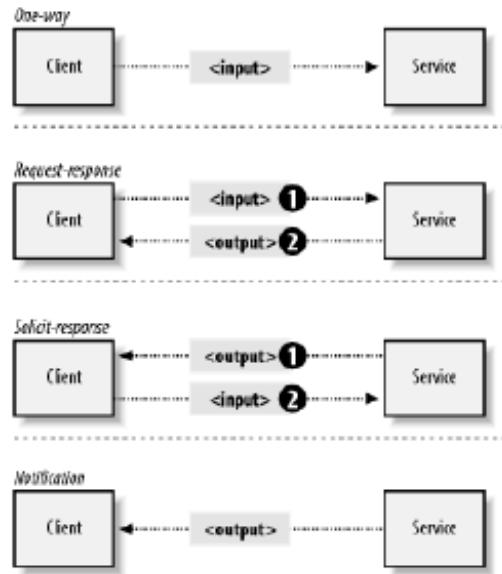
```

11.2.5 WSDL Nachrichtentypen

WSDL unterstützt 4 **Kommunikationstypen**. Die Reihenfolge der **input** und **output** Elemente im **operation** Element legt die Art des Kommunikationstypen fest.

► Auflistung: WSDL Nachrichtentypen

- **Request-Response:** INPUT, OUTPUT
- **Solicit-Response:** OUTPUT, INPUT
- **One-Way:** INPUT
- **Notification:** OUTPUT



□

12. Kommunikationsprotokoll: MOM

04

Message Oriented
Middleware

01. Message Oriented Middleware	79
02. Kommunikationsendpunkte	81
03. MOM Standards	82
04. Fallbeispiel: Routenplaner	83

12.1. Message Oriented Middleware ▾



MOM ▾

Nachrichtenorientierte Middleware - MOM - BEZEICHNET MIDDLEWARE, DIE AUF SYNCHRONER BZW. ASYNCHRONER **Kommunikation** MIT DER HILFE VON NACHRICHTEN BERUHT.

Das **Datenformat** der Nachrichten in einem MOM System ist nicht festgelegt. In der Praxis hat sich jedoch XML als Format etabliert.



12.1.1 Grundlagen

Nachrichtenorientierte Middleware - MOM - hat das Ziel, verlässliche, lose gekoppelte asynchrone **Kommunikation** zwischen den **Servicen** einer Softwareanwendung zu ermöglichen.

► Motivation: Nachrichtenorientierte Systeme ▾

- Nachrichtenorientierte Systeme³³ BASIEREN AUF DEM VERSCHICKEN VON NACHRICHTEN ZWISCHEN DEN SERVICEN EINER SOFTWAREANWENDUNG.
- Nachrichtenorientierte Systeme WURDEN DABEI AUS DER NOTWENDIGKEIT ENTWICKELT FÜR DIE **Service** EINER SOFTWAREANWENDUNG EINE VERLÄSSLICHE, LOSE GEKOPPELTE **Kommunikation** ZUR VERFÜGUNG ZU STELLEN.
- IM VERGLEICH ZUR NACHRICHTENORIENTIERTEN KOMMUNIKATION KÖNNEN ENTFERNTES METHODENAUFRUFE³⁴ FÜR BESTIMMTE SZENARIEN³⁵ KEINE VERLÄSSLICHE KOMMUNIKATION ERMÖGLICHEN.
- NACHRICHTENORIENTIERTE SYSTEM SETZTEN AUF DIE **Entkoppelung** DES SENDER EINER NACHRICHT VON SEINEM EMPFÄNGER. DAMIT KANN DIE **Verlässlichkeit** DER KOMMUNIKATION FÜR EINE SOFTWAREANWENDUNG ERHEBlich GESTEIGERT WERDEN.
- DIE **Entkoppelung** DER SOFTWAREELEMENTE ERREICHT MOM DURCH DEN EINSATZ EINER ZUSÄTZLICHEN ABSTRAKTIONSSCHICHT - DEM **Messagebroker**.



³³ *Message Oriented Systems*

³⁴ *RMI, SOAP, REST, HTTP*

³⁵ *Netzwerkausfall*

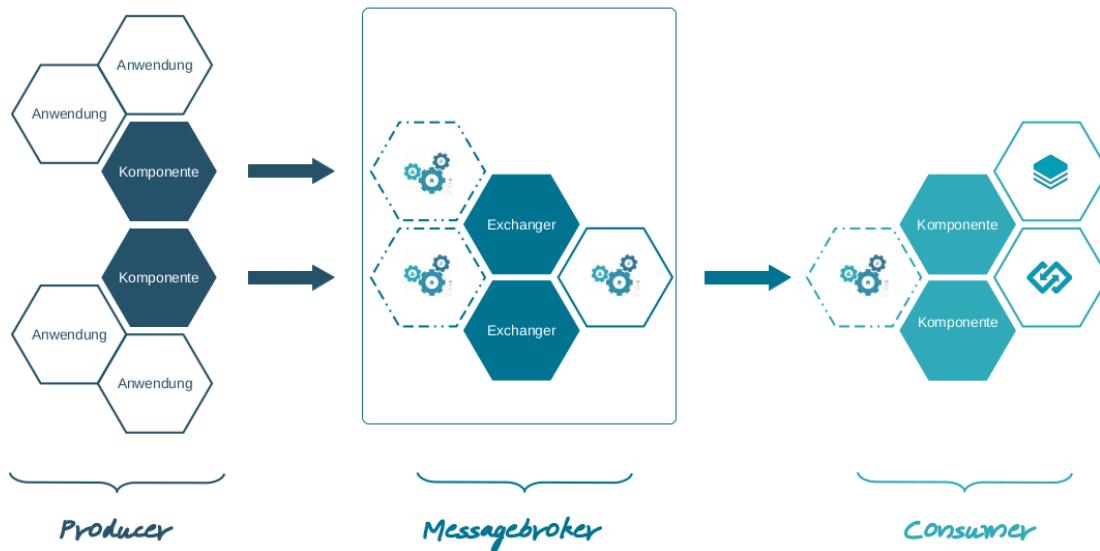


Abbildung 18. Messagebroker

12.1.2 Messagebroker



Messagebroker ▾

EIN **Messagebroker** IST EIN ARCHITEKTURMUSTER, DAS BESCHREIBT, WIE DER Nachrichtenaustausch ZWISCHEN DEN SERVICEN EINES SYSTEMS ZU ERFOLGEN HAT.

► Erklärung: Messagebroker ▾

- EIN **Message Broker** HAT DIE FUNKTION EINES **Servers**, DER NACHRICHTEN VON MEHREREN PRODUCERN EMPFANGEN, DIE KORREKTE **Destination**³⁶ BESTIMMEN UND ZUM NACHRICHTENKONSUMENTEN ROUTEN KANN.
- MIT DEM **Message Broker** ERFOLGT EINE **Entkopplung** DES PRODUCERS VOM CONSUMER. KOMMT ES ZUM AUSFALL EINER DER BEIDEN KOMPONENTEN WIRD DIE NACHRICHT FÜR DIE DAUER DES AUSFALLS ZWISCHENGESPEICHERT UND ANSCHLIESSEND ZUGESTELLT.
- ZUR DAUERHAFTEN PERSISTIERUNG DER NACHRICHTEN KANN DER **Messagebroker** DIE NACHRICHTEN IN EINER DATENBANK SPEICHERN. DIE ÜBERTRAGUNG WIRD SO LANGE WIEDERHOLT, BIS SIE ERFOLGREICH WAR.

12.1.3 Nachrichtenaustausch

Die Verarbeitung von Nachrichten in MOM Systemen erfolgt in der Regel **asynchron**.

► Erklärung: Nachrichtenaustausch ▾

- EIN **Publisher** VERÖFFENTLICHT EINE NACHRICHT, DIE VON DER **Middleware** ZU EINEM ODER MEHREREN **Konsumenten** DER NACHRICHT GESENDET WIRD.
- WEIL DIE **Konsumenten** UND DIE **Publisher** NICHT DIREKT MITEINANDER INTERAGIEREN, KANN DER PUBLISHER NICHT VON DEN KOMPONENTEN BLOCKIERT WERDEN, AUCH WENN DIE VERARBEITUNG DER **Nachricht** LÄNGER DAUERT ODER DIE KONSUMENTEN TEMPORÄR NICHT ERREICHBAR SIND.
- DER **Konsument** BRAUCHT SICH UM DIE **Nachricht** NICHT MEHR ZU KÜMMERN SOBALD ER SIE VERSANDT HAT.
- DER **Messagebroker** IST IN NACHRICHTENORIENTIERTEN SYSTEMEN FÜR DAS VERLÄSSLICHE VERSCHICKEN VON **Nachrichten** VERANTWORTLICH.
- BEI EINEM NACHRICHTENORIENTIERTEM SYSTEM SPRECHEN WIR DESHALB VON EINEM STARK ENTKOPPELTEM SYSTEM.

³⁶ Ziel

12.1.4 Eigenschaften von MOM Systemen

Nachrichtenorientierte Systeme zeichnen sich durch eine Reihe von Eigenschaften gegenüber einfachen RMI Systemen aus.

► Auflistung: Eigenschaften ▾

- **Zuverlässigkeit:** NACHRICHTEN KÖNNEN AUCH BEI AUSFALL DES NETZWERKS ÜBERTRAGEN WERDEN.

Der **Messagebroker** speichert die Nachrichten zwischen und stellt sie zu, wenn das Netzwerk wieder zur Verfügung steht.

- **Schwache Koppelung:** DER SENDER KENNT DEN EMPFÄNGER DER NACHRICHT NICHT.

Der Sender schickt die Nachricht an den **Messagebroker**. Die Empfänger der Nachricht kennen ebenfalls nur den Messagebroker nicht aber den Sender. Damit Sender und Empfänger **entkoppelt**. Es kann sogar mehrere Empfänger geben, ohne dass dies dem Sender bekannt ist.

- **asynchrone Kommunikation:** DAS VERSCHICKEN EINER NACHRICHT BLOCKIERT DEN SENDER NICHT.

In einer **Messaging Architektur** werden Antworten **asynchron** übertragen und bearbeitet. Damit haben hohen **Latenzzeiten** im Netzwerk minimale Auswirkungen auf nachrichtenorientierte Systeme.



12.1.5 Kommunikationsmuster

In nachrichtenorientierten Systemen können wir fünf **Kommunikationsmuster** identifizieren, die sich entlang zweier Dimensionen unterscheiden. Zum einen kann in der **Kardinalität**, zum anderen in der **Synchronizität** unterschieden werden.

► Auflistung: Ausprägungen ▾

- **one to one:** PARTNER KOMMUNIZIEREN 1:1 MITEINANDER.
- **one to many:** PARTNER KOMMUNIZIERT MIT VIELEN ANDEREN.
- **Synchrone Kommunikation:** EINE NACHRICHT WIRD SYNCHRON BEANTWORTET.
- **Asynchrone Kommunikation:** DER SENDENDE PARTNER WARTET NICHT AUF EINE ANTWORT.



12.2. Kommunikationsendpunkte

In **Nachrichtenorientierten Systemen** erfolgt die Kommunikation zwischen Sendern und Empfängern über **Destinationen**³⁷ des Messagebrokers.

Destination
Destinationen SIND DIE Kommunikationsendpunkte DES Messagebrokers.

12.2.1 Destinationen

Messagebroker unterstützen für die Kommunikation 3 Arten von Destinationen.

► Auflistung: Destinationen ▾

- **Queue:** Queues HABEN ZWEI ARTEN VON CLIENTS: **Produzenten** UND **Konsumenten**.

Die Nachricht eines Produzenten kann nur von einem Konsumenten empfangen werden. Nach Bestätigung des Empfangs der Nachricht wird diese automatisch aus der Queue entfernt.

- **Topic:** Topics HABEN EBENFALSS ZWEI ARTEN VON CLIENTS: **Publisher** UND **Subscriber**.

Die Subscriber erhalten die Nachrichten der Publisher, falls sie sich für das passende Topic angemeldet haben. Im Gegensatz zu Queues können mehrere Subscriber die gleiche Nachricht empfangen.

- **Durable Subscriptions:** WENN DIE ABONNIERTEN NACHRICHTEN SO LANGE IN DER QUEUE GESPEICHERT WERDEN SOLLEN, BIS DIE KONSUMENTEN SIE SCHLIESSLICH EMPFANGEN KÖNNEN, SPRICHT MAN VON EINER **durable Subscription**.



³⁷ Vergleichbar mit Netzwerk Ports

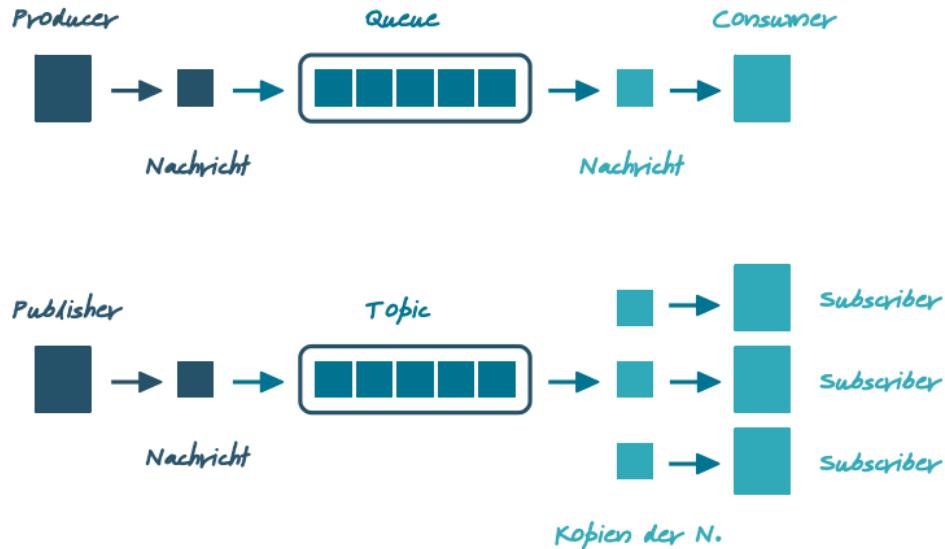


Abbildung 19. Kommunikationsmuster

12.3. MOM Standards

Nachrichtenorientierte Middleware unterstützt in der Regel mehrere unterschiedliche Protokolle.

12.3.1 Protokoll: STOMP

STOMP ist ein Acronym und steht für Simple Text Oriented Messaging Protocol

STOMP eignet sich insbesondere zur **Kommunikation** zwischen Servicen die in unterschiedlichen Programmiersprachen erstellt worden sind.

Erklärung: STOMP

- STOMP BASIERT AUF EINEM TEXTBASIERTEN Format ZUR **Kommunikation** VON SERVICEN.
- MIT ACITIVEMQ BZW. RABBITMQ WIRD DAS PROTOKOLL VON 2 DER FÜHRENDEN MESSAGBROKERN IMPLEMENTIERT.
- DAS STOMP PROTOKOLL IST EINFACH UND KANN LEICHT IMPLEMENTIERT WERDEN.
- ALS **Transportschicht** WIRD TCP/IP VERWENDET.

12.3.2 Protokoll: MQTT

MQTT definiert ein kompaktes **Binärformat**, das für Geräte mit beschränkter Rechenkapazität in einem nicht verlässlichen Netzwerk entworfen wurde.

MQTT eignet sich besonders für **Internet of Things** Anwendungsfälle.

Erklärung: MQTT

- DAS DURCH MQTT UNTERSTÜTZTE **Publish Subscribe** MUSTER EIGNET SICH FÜR DIE 1:N KOMMUNIKATION VERSCHIEDENER GERÄTE. DIE GERÄTE MÜSSEN SICH NICHT KENNEN. ES WERDEN **Topics** FÜR DIE KOMMUNIKATION VERWENDET.
- DIE **Kommunikation** LÄUFT ÜBER EINEN ZENTRALEN MQTT BROKER.
- MQTT LÄUFT AUF **Applikationsebene** UND KANN MIT ETABLIERTEN STANDARDS WIE SSL/TLS GESICHERT WERDEN.
- DIE EFFIZIENZ DES PROTOKOLLS SOWIE DIE ANGEBOTENEN GARANTIEN DER **Nachrichtenübermittlung** MACHEN MQTT AUCH FÜR **mobile Anwendungen** INTERESSANT.

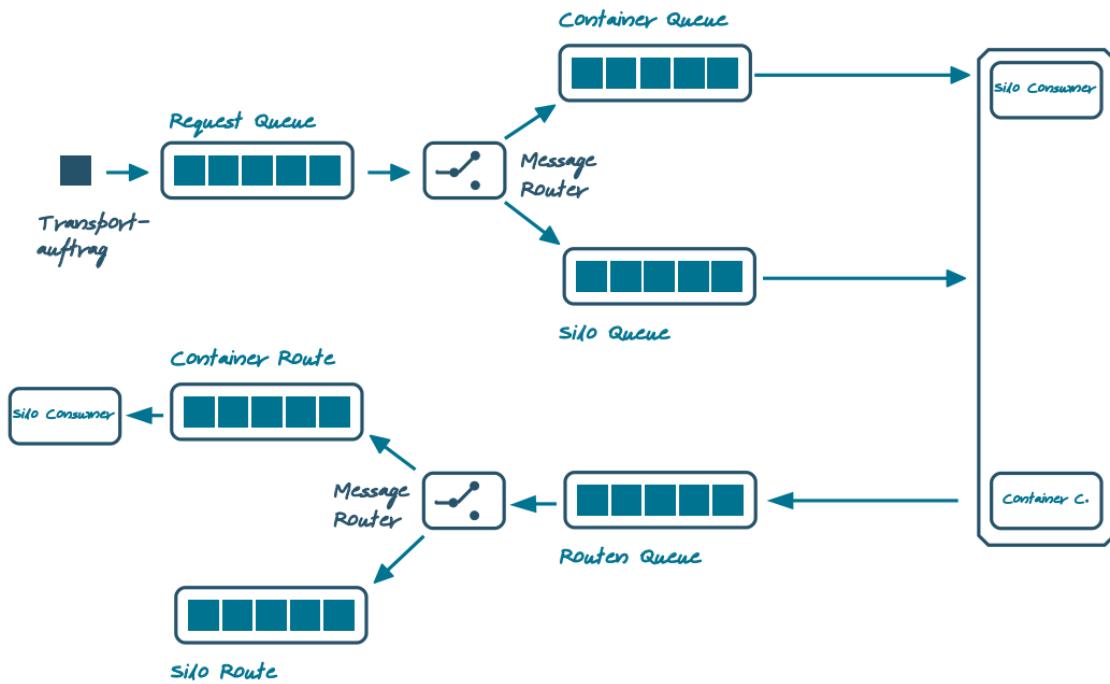


Abbildung 20. Routenplaner

12.3.3 Protokoll: AMQP

AMQP ist ein Acronym und steht für Advanced Message Queue Protocol.

AMQP wird in Szenarien eingesetzt die komplexe Kommunikationsvorgänge abbilden müssen.

► Erklärung: AMQP ▾

- IM AMQP MODELL SENDET EIN **Produzent** SEINE **Nachricht** NICHT DIREKT ZU EINER QUEUE, SONDERN ZU EINEM **Exchange**. DER EXCHANGE ROUTET DIE NACHRICHT DANN NACH BESTIMMTEN REGELN ZUR KORREKten MESSAGEQUEUE WEITER.
- DIE **Messagequeues** MÜSSEN ZUVOR AN DEN EXCHANGE GEBUNDEN WERDEN.



12.3.4 JMS

JMS ist kein Messaging Protokoll, sondern eine standardisierte **Programmierschnittstelle**, mit der **Java** basierte Anwendungen Nachrichten über einen Message-broker erzeugen, senden, empfangen und lesen können.

12.4. Fallbeispiel: Routenplaner ▾

12.4.1 Use Case Beschreibung

Im Mittelpunkt der Anwendung steht ein Dienst zur **Routenplanung**, der Nachrichten wie - Transportiere Container von Lokation A nach B - oder - Transportiere Schüttgut von Lokation C nach D - akzeptiert.

► Use Case: Routenplaner ▾

- DIE EINGEHENDEN **Transportaufträge** WERDEN VOM **Routenplaner** GEspeichert UND BESTÄTIGT.
- DER AUFTRAGGEBER WARTET NICHT BIS DER **Transportauftrag** TATSÄCHLICH DURCHGEFÜHRT WURDE, SONDERN ERHÄLT EINE BESTÄTIGUNG UND BEendet DIE KOMMUNIKATION MIT DEM ROUTENPLANER.
- ES GIBT UNTERSCHIEDLICHE **Arten von Transportaufträgen**: SCHÜTTGUT, CONTAINER, STAPELGUT UND Silogüter.
- JE NACH ART DES **Transportauftrags** WIRD EIN UNTERSCHIEDLICHER **Routenplaner** EINGESETZT.

- DIE BERECHNUNG DER ROUTE BASIERT AUF EINEM KOMPLEXEN **Algorithmus**, DER PRIORITY, ZEIT, ENTFERNUNG UND ANDERE PARAMETER BERÜCKSICHTIGEN KANN, UM EINE MÖGLICHST EFFIZIENTE **Fahrroute** FÜR DEN FAHRER ZU BERECHNEN. KONKRETE DETAILS EINES SOLCHEN ALGORITHMUS SOLLEN IN DIESEM BEISPIEL VERNACHLÄSSIGT WERDEN.
- **Fahrer** MELDEN SICH UNTER ANGABE IHRER ID UND DER **Art** IHRES WAGENS AN. IHM WIRD EINE FAHRT ZUGEWIESEN, DIE DER ART SEINES WAGENS ENTSPRICHT.



13. Kommunikationsprotokoll: Websocket

05

Websocket

01. Websocket

85

13.1. Websocket Protokoll ▾



Websocket Protokoll ▾

DAS **Websocket Protokoll** IST EIN AUF **TCP** BA-SIERENDES NETZWERKPROTOKOLL, DAS ENTWORFEN WURDE UM **bidirektionale** VERBINDUNG ZWISCHEN KOMMUNIKATIONSPARTNERS HERZUSTELLEN.

13.1.1 Http vs Websockets

Im Großen und Ganzen wurde das Internet um das sogenannte **Anforderungs-** und **Antwortmuster** von **http** konstruiert.

► Erklärung: **Http Protokoll** ▾

- URSPRUNGLICH IST DIE **Kommunikation** ÜBER **HTTP** IN ERSTER LINIE **client gesteuert**³⁸.
- MIT **Ajax** WURDE NACHFOLGEND EINE **Technologie** INTEGRiert, DIE ES ERMÖGLICHTE LEDIGLICH JENE TEILE EINER WEBSEITE NEU ZU LADEN, DIE SICH VERÄNDERT HABEN.
- Damit wurde das http Protokoll einigermassen **dynamischer**.
- MIT **Push** WURDE EINE ZUSÄTZLICHE TECHNOLOGIE ETABLIERT, DIE ES DEM **Server** ERLAUBTE **Daten** AN DEN CLIENT ZU SENDEN, SOBALD SIE VERFÜGBAR WURDEN.



Allen **Technologien**, die auf dem **http Protokoll** beruhen, haben jedoch den Nachteil das sie eine erhebliche **Latenzzeit** aufweisen.

► Erklärung: **Websocket** ▾

- DIE **Websocket** SPEZIFIKATION LEGT EINE API FEST, DIE EINE **Verbindung** ZWISCHEN EINEM WEBBROWSER UND EINEM SERVER HERSTELLT.
- ES BESTEHT DAMIT EINE **persistente** VERBINDUNG ZWISCHEN CLIENT UND SERVER. BEIDE PARTEIEN KÖNNEN JEDERZEIT



³⁸ Der Client gibt eine Url in den Webbrower ein. Der Client klickt auf einen Link.



Abbildung 21. Websocket Kommunikation

13.1.2 Kommunikationsablauf

Zu Beginn jeder **Verbindung** führen Server und Client einen sogenannte **Handshake** durch.

Bei der **Kommunikation** bleibt die zugrundeliegende TCP Verbindung bestehen und ermöglicht damit eine **asynchrone Übertragung** in beide Richtungen.

► Erklärung: Kommunikationsablauf ▾

- DER **Websocket Handshake** ÄHNLT DEM **Http Header** UND IST VOLLSTÄNDIG ABWÄRTSKOMPATIBEL ZU DIESEM. DAMIT IST ES MÖGLICH EINE **Websocket Verbindung** ÜBER DEN **Http Port** 80 ZU NUTZEN.
- WÄHREND BEI EINER REINEN **Http Verbindung** JEDO Aktion DES SERVERS EINE VORHERGEHENDE **Anfrage** DES CLIENTS ERFORDET, REICHT ES BEIM WEB SOCKET PROTOKOLL, WENN DER CLIENT DIE **Verbindung ÖFFNET**.
- DER **Server** KANN DANN DIESE OFFENE **Verbindung** AKTIV VERWENDEN UM NEUE **Informationen** AN DEN CLIENT AUSLIEFERN, OHNE AUF EINE NEUE VERBINDUNG DES CLIENTS ZU WARTEN.

► Analyse: Kommunikationsablauf ▾

- DURCH DIE **Nutzung** VON WEBSOCKETS ENTSTEHT EIN VÖLLIG NEUES **Nutzungsmuster** FÜR SERVERSEITIGE ANWENDUNGEN.
- HERKÖMMLICHE **Serverstapel** SIND ENTPRECHEND DES **HTTP Anforderungs-/Antwortzyklus** KONZIPIERT UND FUNKTIONIEREN MEIST NICHT GUT MIT EINER GROSSEN ANZAHL OFFENER WEBSOCKET **Verbindungen**.

13.1.3 Websocket Protokoll

Websockets setzen unter anderem auf **Stomp**³⁹ als Protokoll.



stomp Protokoll ▾

stomp IST EIN **Nachrichtenorientiertes Streamingprotokoll**. DIE ART DER KOMMUNIKATION ENTSPRICHT DEM EINES MESSAGEBROKERS.

► Auflistung: Methoden des stomp Protokolls ▾

- **connect:** AUFBAU EINER **Verbindung** ZWISCHEN CLIENT UND SERVER.
- **subscribe:** DER CLIENT ABONNIERT EIN BESTIMMTES **Topic**.
- **unsubscribe:** DER CLIENT BEENDET DAS ABONNEMENT EINES GEWISSEN **Topics**.
- **send:** DER CLIENT SCHICKT EINE NACHRICHT AN DEN SERVER.



13.1.4 Zusammenfassung

Websockets werden für Anwendungen eingesetzt für die eine schnelle Kommunikation zwischen Client und Server in **bidirektonaler Weise** notwendig ist. Denken Sie in diesem Zusammenhang an MMO und Anwendungen mit ähnlichen Anforderungen.



³⁹ streaming text oriented messaging protocol

Microservice Programmierung

Version 1.0

14. Service - Programmierung der Modelschicht

01

Modelschicht

01 ... Struktur eines Services	89
02 ... Programmierung des Model	90
03 ... Persistierung von Objekten	92
04 ... Polymorphie	100
05 ... Validieren von Daten	103
06 ... Objektkomposition	107
07 ... Auditing	??

14.1. Struktur eines Services

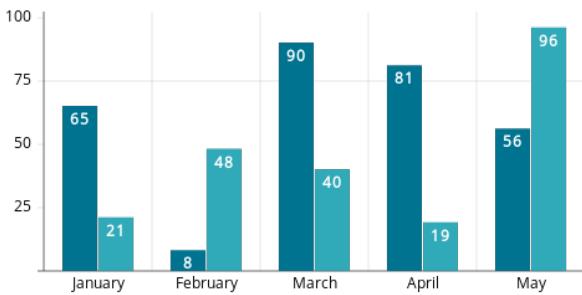


Service ▾

EIN Service IST EINE Softwarekomponente, AUSGEFÜHRT IN EINEM EIGENEN Betriebssystemprozess.

► Erklärung: SOA System ▾

- EINE SOA Anwendung IST EINE Komposition VON Servicen.
- SOA Systeme VERBINDELN SERVICE ZU EINEM Gesamtsystem.
- EIN Service Besteht IN DER Regel AUS 3 Schichten.
- JEDE DER Schichten ERFÜLLT EINE EIGENE AUFGABE.



14.1.1 Schichten eines Services

Ein Service ist softwaretechnisch in Schichten aufgeteilt. In der Regel unterscheiden wir für ein Microservice 3 Schichten.

► Erklärung: Schichten eines Service ▾

- **Modelschicht:** DIE Modelschicht ENTSPRICHT DER Struktur DES Datarepository⁴⁰ DES SERVICES.

Die Schicht beinhaltet jene Klassen⁴¹, die notwendig sind um Daten aus dem Datarepository zu kapseln. Klassen der Modelschicht implementieren kein Verhalten.

⁴⁰ SQL Datenbank, NoSQL Datenbank, Dateien usw.

⁴¹ Objekte deren Aufgabe es ist in erster Linie Daten zu transportieren werden DataTransferObjects genannt

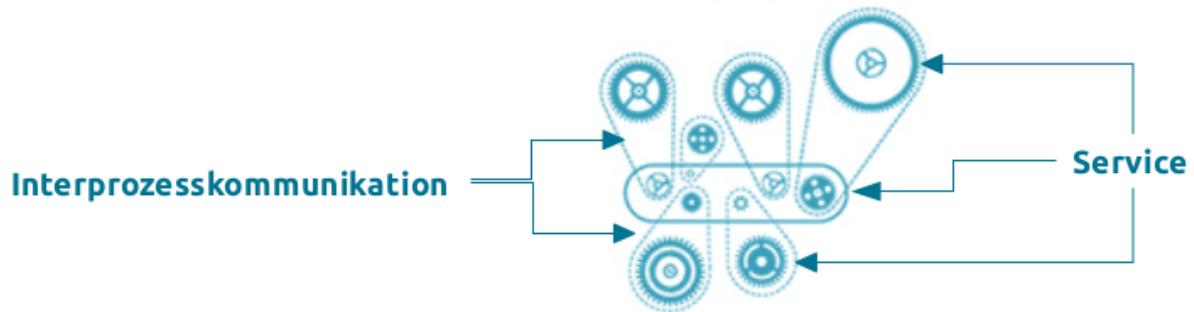


Abbildung 22. SOA - Anwendung

- **Domainschicht:** DIE Domainschicht BEINHALTET DIE Logik DES Services.

Die Schicht kapselt die **datenlesende-** und **datenschreibende** Funktionalität des Services. Die Klassen der Domäinschicht verwalten in der Regel keinen Zustand sondern stellen **Verhalten** zur Verfügung.

- **Serviceschicht:** DIE Serviceschicht IST VERANTWORTLICH FÜR DIE **Kommunikation** DER SERVICE UNTEREINANDER.

Die **Kommunikation** der Service erfolgt über **plattformunabhängige**⁴² Protokolle. Damit ist es möglich Service in unterschiedlichen **Technologien** zu implementieren und sie trotzdem zu einer Anwendung zu **integrieren**.

14.2. Programmierung des Models ▾



Modelschicht ▾

DIE KLASSEN DER **Modelschicht** ENTSPRICHTEN DER Struktur DES Datarepository DES SERVICES.

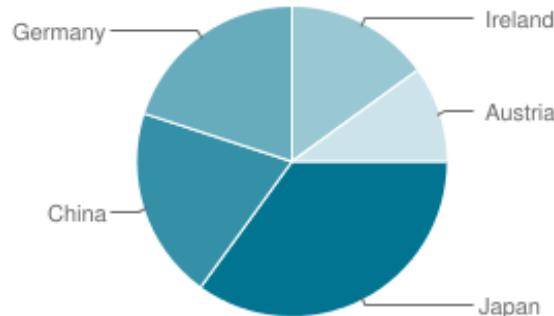
- DIE SCHICKT BEINHALTET JENE Klassen, DIE NOTWENDIG SIND UM **Daten** AUS DEM **Datenrepository** ZU KAPSELN. KLASSEN DER MODELSCHICKT IMPLEMENTIEREN KEIN **Verhalten**.

Für die Programmierung der Klassen der **Modelschicht** arbeiten wir mit **Object Relational Mapping**.



► Erklärung: Programmtechniken ▾

- ZIEL DER **Objektrelationalen Abbildung** IST ES DIE Schnittstellen ZU **relationale Datenbanken** NACH **objektorientierten** PRINZIPIEN ZU PROGRAMMIEREN.
- KLASSEN DER **Modelschicht** SIND DEM **ORM Paradigma** FOLGEND IN ERSTER LINE **POJOs**.
- **POJOs** SIND EINFACHE JAVA KLASSEN DIE KEINE METHODEN BIS AUF **Properties** IMPLEMENTIEREN.
- UM DIE **Struktur** DER DATENBANK AUF DIE **Objekten** DER KLASSE ABBZUBILDEN WERDEN **Annotationen** VERWENDET.



⁴² *Http Protokoll, SOAP usw.*

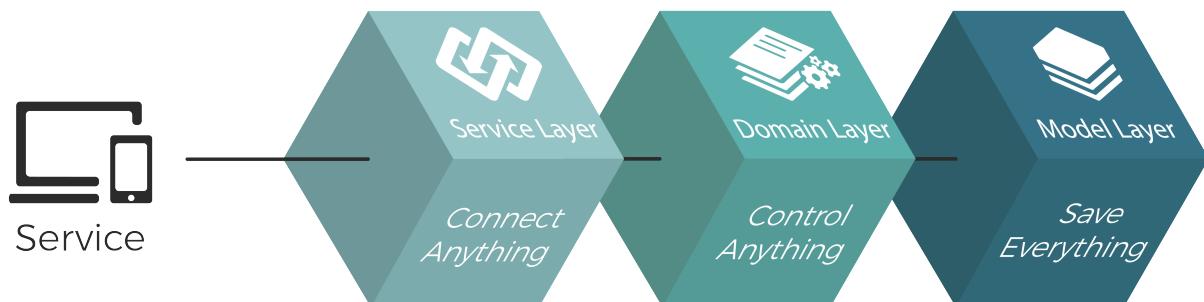


Abbildung 23. Schichten eines Microservices

14.2.1 Prinzipien der objektorientierten Programmierung

Die **objektorientierte Programmierung** ist ein auf dem Konzept der **Objektorientierung** basierendes **Programmierparadigma**.

DIE **Grundidee** BESTEHT DARIN, DIE **Architektur** EINER SOFTWARE AN DEN **Grundstrukturen** DESJE-
NIGEN BEREICHS DER **Wirklichkeit** AUSZURICHTEN,
DER DIE GEGBENE ANWENDUNG BETRIFFT.

DIE REALITÄT WIRD **abgebildet** ALS **Netzwerk**
VON **Objekten**. DAS **Objekt** IST DER GRUNDLEGEN-
DE **Baustein** IN EINER OBJEKTORIENTIERTEN AN-
WENDUNG.

► Erklärung: Prinzipien der Objektorientierung ▾

- **Datenkapselung:** EIN **Objekt** HAT EINEN **Zustand**⁴³ UND EIN **Verhalten**⁴⁴.
- **Objektvererbung:** MITTELS **Vererbung** KÖNNEN SO-
WOHL DER **Typ**, DER DURCH SEINE **Schnittstelle** SPE-
ZIFIZIERT WIRD, ALS AUCH DIE **Funktionalität** AN DIE
ABGELEITETE KLASSE WEITERGEGEBEN WERDEN
- **Objektkomposition:** KLASSEN SIND AUS ANDEREN
KLASSEN **aufgebaut**. DIESES KONZEPT BEZEICHNEN WIR
ALS **Objektkomposition**.

⁴³ Variablen

⁴⁴ Methoden

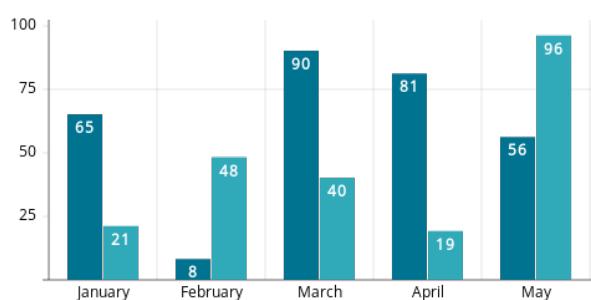
14.2.2 Prinzip der Objektrelationale Abbildung

Die **Objektrelationale Abbildung** ist eine Program-
miertechnik mit der, ein in einer **objektorientierten**
Programmiersprache geschriebenes Programm, Ob-
jekte in einer relationalen Datenbank ablegen kann.

Dem Programmierer erscheint die Datenbank als **objektorientierte Datenbank**, was die Program-
mierung erheblich erleichtert.

► Analyse: Modelvergleich ▾

- **Objektorientierte PROGRAMMERSPRACHEN** KAPSELN
DATEN UND VERHALTEN IN **OBJEKten**
- IN **relationalen** DATENBANKEN GEBEN TABELLEN DIE
STRUKTUR DER DATEN VOR. DATEN SELBST WERDEN ZEI-
LENWEISE IN EINER TABELLE GESPEICHERT.
- DAS **OBJEKTORIENTIERTE** PROGRAMMIERPARADIGMA UND
DAS **RELATIONALE** MODEL SIND 2 GRUNDELIG UNTER-
SCHIEDLICHE **Programmierparadigmen**.
- DIE **Objektrelationale Abbildung** FINDET EINEN
WEG BEIDE KONZEPTE ZU VERSCHMELZEN.



14.3. Persistierung einfacher Objekte

14.3.1 Grundlagen der Persistierung

Das grundlegende Konzept von **ORM** ist es Objekte der Modelschicht auf **Tabellen** abzubilden.

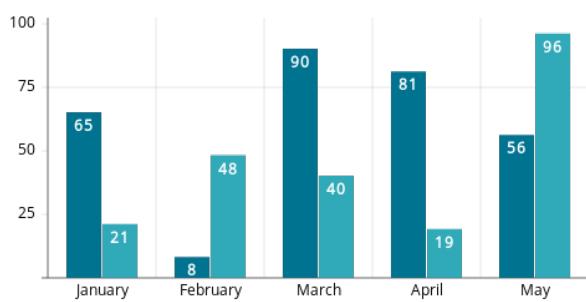
► Erklärung: Prinzipien des ORM ▼

- **Objekte** entsprechen **Tabellenzeile** in der Datenbank.
- **Objektattribute** werden in **Tabellenspalte** gespeichert.
- Die **Identität** eines in der Datenbank gespeicherten Objekts entspricht dem in der zugeordneten Tabellenzeile abgelegten **Primärschlüssels**.
- Speichert ein **Objekt** eine **Referenz** auf ein anderes Objekt, so wird diese Beziehung als Eintrag in einer **Fremdschlüsselspalte** abgebildet.



► Erklärung: Erfordernisse ▼

- Um Objekte **persistieren** zu können müssen wir in der Lage sein die Techniken der **Objektorientierung** - Kapselung, Objektkomposition und Vererbung - auf eine relationale **Datenbank** abzubilden.
- Um das zu erreichen bedient sich **ORM** zweier Techniken: **Annotationen** und **POJOs**



14.3.2 POJO und Annotationen

► Erklärung: POJO ▼

- **POJOs** sind einfache Java Klassen
- **POJO** sind unabhängig von Springinternen Klassen.
- Durch den Einsatz von **POJO** wird die **Komplexität** der Anwendungsentwicklung reduziert.



Annotationen ▼

Annotation bedeutet **Anmerkung**. In der Softwareentwicklung dienen **Annotationen** dazu **Metadaten** in den Quelltext eines Programms einzubinden.



► Erklärung: Annotation ▼

- Bei **Annotationen** handelt es sich nicht um Programmieranweisungen sonder zusätzliche **Artefakte** einer **Programmiersprache**.
- Annotationen erlauben uns in einer **objektorientierten** Programmiersprache **deklarativ** zu programmieren.
- Die **deklarative Programmierung**⁴⁵ ist ein **Programmierparadigma**, bei dem das Problem durch den Programmierer beschrieben, aber nicht durch ihn gelöst wird.
- Der Lösungsweg wird dann automatisch durch das System generiert.
- Wird das Programm kompiliert werden für **Annotationen** entsprechende **Preprozessoren** ausgeführt.
- **Preprozessoren** generieren entsprechend der **Annotation** Code.
- Programmertechnisch werden **Annotationen** in Java mit definiert angefangen mit einem @ gefolgt vom Namen der Annotation z.B.: @Data

⁴⁵ z.B.: SQL

► Codebeispiel: Project.java ▾

```

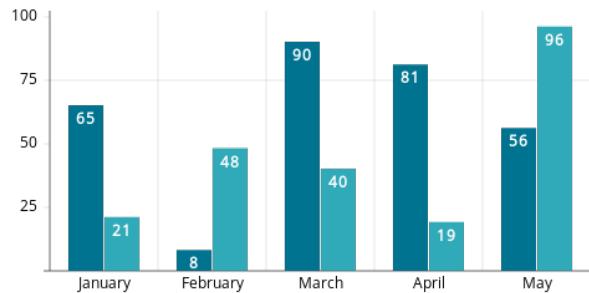
1 //-----
2 // Annotationen
3 //-----
4 //Die @NoArgsConstructor Annotation
5 //stößt die Generierung eines
6 //parameterlosen Konstruktors an
7 @NoArgsConstructor
8 //Mit der @Getter und @Setter Annotation wird
9 //das System angehalten für jede Variable
10 //Properties zu generieren
11 @Getter
12 @Setter
13 public class Project implements Serializable{
14
15     private Long id;
16
17     private String title;
18
19     private String description;
20
21     private Date creationDate;
22
23 }

```

```

5     @Getter
6     @Setter
7     @Entity
8     @Table(name="projects")
9     @Access(AccessType.FIELD)
10    public class Project implements Serializable{
11
12        @GeneratedValue(strategy = GenerationType.AUTO)
13        @Id
14        private Long id;
15
16        @Column(name="title", length=100,
17                 nullable=false)
17        private String title;
18
19        @Column(name="description")
20        private String description;
21
22        @Column(name="appliedResearch")
23        private Integer appliedResearch;
24
25        @Column(name="focusResearch")
26        private Integer focusResearch;
27
28    }

```



14.3.3 Entität



Entität ▾

Entitäten sind Objekte die direkt an die Datenbankschnittstelle zur Bearbeitung übergeben werden können.

- Eine Entität ist damit ein Objekt, das eine Repräsentation in der Datenbank als Tabellezeile einer Tabelle besitzt.

► Codebeispiel: Project Entität ▾

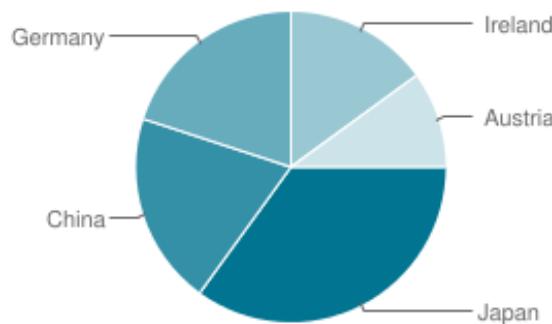
```

1 //-----
2 // Entität
3 //-----
4 @NoArgsConstructor

```

► Erklärung: Project Entität ▾

- @Entity:** Die `@Entity` Annotation transformiert ein gewöhnliches Pojo in eine Entität.
- @Table:** Die `@Table` Annotation stellt die Klasse `Project` in Relation mit der `projects` Tabelle. Zusammen mit der `@Entity` Annotation können `Project` Objekte damit als Zeilen in der `projects` Tabelle gespeichert werden.
- @Id:** Mit der `@Id` Annotation wird ein **Objektattribute**, im Kontext seiner ORM Abbildung als **Datenbank ID** gekennzeichnet.
- @Column:** Die `@Column` Annotation stellt die Spalten einer Tabelle in Relation mit dem **Attribute** einer Entität.



14.3.4 Entitätsidentität - Vergleich von Entitäten

► Erklärung: Objektidentität ▾

- WERDEN Objekte verglichen, WERDEN SIE ALS gleich ANGESEHEN, WENN SIE UNTER DERSELBEN Speicheradresse GESPEICHERT SIND.
- Objekte WERDEN MIT DEM == OPERATOR AUF referentielle Gleichheit GEPRÜFT.

```

1  -----
2  // Vergleich auf Referenzgleichheit
3  -----
4  public class Application{
5
6      public static void main(String args[]){
7          Project p1 = new Project();
8          Project p2 = new Project();
9
10         if(p1 == p2){
11             System.out.println("Objekte sind
12                 gleich.");
13         else{
14             System.out.println("Objekte sind
15                 verschieden.")
16         }
17
18         Project p3 = p1;
19
20         if(p1 == p3){
21             System.out.println("Objekte sind
22                 gleich.");
23         else{
24             System.out.println("Objekte sind
25                 verschieden.")
26     }
27
28     Programmausgabe:
29     Objekte sind verschieden.
30     Objekte sind gleich.

```

- UM ZU VERGLEICHEN OB 2 OBJEKTE DENSELBEN INHALT HABEN, WIRD DIE EQUALS() METHODE VERWENDET.

```

1  -----
2  // equals() - Methode
3  -----
4  @NoArgsConstructor
5  @Getter
6  @Setter
7  @Entity
8  @Table(name="projects")
9  @Access(AccessType.FIELD)
10 public class Project implements Serializable{
11

```

```

12     @GeneratedValue(strategy = GenerationType.AUTO)
13     @Id
14     private Long id;
15
16     @Column(name="title", length=100,
17             nullable=false)
18     private String title;
19
20     @Column(name="description")
21     private String description;
22
23     public boolean equals(Object o){
24         if (this == o) return true;
25
26         if (!(o instanceof Project))
27             return false;
28
29         if (!super.equals(o))
30             return false;
31
32         Project project = (Project) o;
33         boolean flag = true;
34
35         flag &= getId().equals(project.getId());
36         flag &= getTitle().equals(
37             project.getTitle());
38         flag &= getDescription().equals(
39             project.getDescription());
40
41         return flag;
42     }
43
44 }
45
46
47 public class Application{
48
49     public static void main(String args[]){
50         Project p1 = new Project();
51
52         p1.setId(21);
53         p1.setTitle("Finite Methoden");
54         p1.setDescription("Project to verify
55                         mathimaical Methods ...")
56         p1.setCreationDate(new Date());
57
58         Project p2 = new Project();
59
60         p2.setId(31);
61         p2.setTitle("Finite Methoden");
62         p2.setDescription("Project to verify
63                         mathimaical Methods ...")
64         p2.setCreationDate(new Date());
65
66
67         if(p1.equals(p2)){
68             System.out.println("Objektinhalt ist
69                 gleich.");
70         else{
71             System.out.println("Objektinhalt ist
72                 verschieden.")
73     }

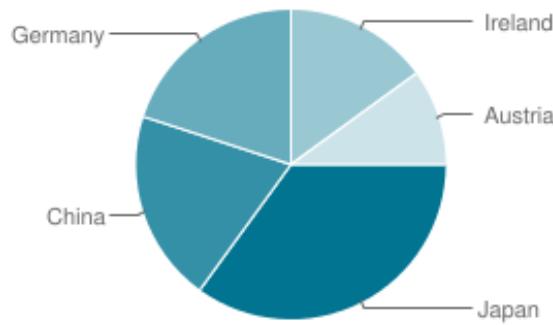
```

```

70     p2.setId(21);
71
72     if(p1.equals(p2)){
73         System.out.println("Objektinhalt ist
74             gleich.");
75     }else{
76         System.out.println("Objektinhalt ist
77             verschieden.")
78     }
79 }
80
81 Programmausgabe:
82
83 Objektinhalt ist verschieden.
84 Objektinhalt ist gleich.

```

- ZWEI Zeilen IN DER DATENBANK WERDEN ALS GLEICH ERACHTET WENN SIE denselben Primärschlüssel HABEN.
- ARBEITEN WIR MIT Entitäten MÜSSEN WIR SOWOHL DIE KRITERIEN FÜR Objektgleichheit PRÜFEN KÖNNEN, ALS AUCH ZEILENGLEICHHEIT FÜR DATENWERTE GEWÄHRLEISTEN.



► Codebeispiel: Entitätsidentität ▾

```

1 //-----
2 // Entitätsidentität
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Entity
8 @Table(name="projects")
9 @Access(AccessType.FIELD)
10 public class Project implements Serializable{
11
12     @GeneratedValue(strategy = GenerationType.AUTO)
13     @Id
14     private Long id;
15
16     @Column(name="title", length=100,
17             nullable=false)
17     private String title;
18
19     @Column(name="description")
20     private String description;
21
22     @Override
23     public boolean equals(Object o) {
24         if (this == o) return true;
25         if (!(o instanceof Project)) return false;
26         if (!super.equals(o)) return false;
27
28         Project project = (Project) o;
29
30         return getId().equals(project.getId());
31     }
32
33     @Override
34     public int hashCode() {
35         int result = super.hashCode();
36         result = 31 * result + getId().hashCode();
37         return result;
38     }
39 }
40

```

► Analyse: Entitätsidentität ▾

- 2 ENTITÄTEN WERDEN ALS Gleich ERACHTET WENN SIE INSTANZEN DERSELBEN Klasse SIND UND IHRE IDs DENSELBEN WERT HABEN.
- WAS PASSIERT JEDOCH WENN DIE Entität NOCH NICHT IN DER DATENBANK GESPEICHERT WORDEN IST UND DAMIT KEINE id HAT.
- FÜR DIESEN FALL MUSS EINE künstliche ID ZUR VERFÜGUNG GESTELLT WERDEN BIS DIE ENTITÄT IN DER DATENBANK GESPEICHERT WORDEN IST.

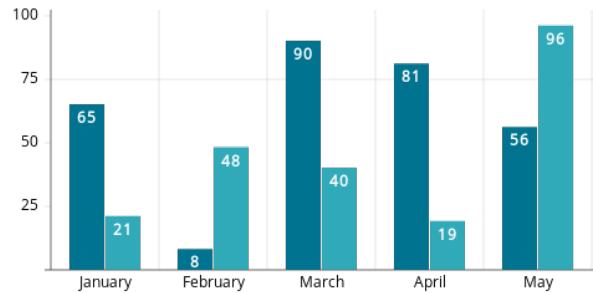
► Codebeispiel: Entitätsidentität ▾

```

1 //-----
2 // Entitätsidentität
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Entity
8 @Table(name="projects")
9 public class Project implements Serializable{
10
11     @GeneratedValue(strategy = GenerationType.AUTO)
12     @Id
13     private Long id;
14
15     @Transient
16     private Long uuid = null;
17
18     @Column(name="title", length = 100, nullable =
19             false)
20     private String title;
21
22     @Column(name="description", length = 4000)
23     private String description;
24
25     public Long getUuid(){
26         return (uuid == null) ? uuid =
27                 UUID.randomUUID().getLeastSignificantBits()
28                         : uuid;
29     }
30
31     public boolean equals(Object o) {
32         if (this == o) return true;
33         if (!(o instanceof Project)) return false;
34         if (!super.equals(o)) return false;
35
36         Project project = (Project) o;
37         if(id == null){
38             return getUuid().equals(((Project)
39                             o).getUuid());
40         }
41
42         return getId().equals(project.getId());
43     }
44
45     public int hashCode() {
46         int result = super.hashCode();
47
48         if(id == null){
49             result = 31 * result +
50                     getUuid().hashCode();
51         }else{
52             result = 31 * result +
53                     getId().hashCode();
54         }
55
56         return result;
57     }
58 }
```

14.3.5 DRY Prinzip

DRY⁴⁶ ist eines der Prinzipien, dass im Zusammenhang mit dem **objektorientierten Entwurf** von Software populär wurde. Das Prinzip fordert, dass **Code-Wiederholungen zu vermeiden** sind, so dass einmal identifizierte Software-Funktionen auch nur einmal im Code zu formulieren sind.



► Codebeispiel: DRY Prinzip ▾

```

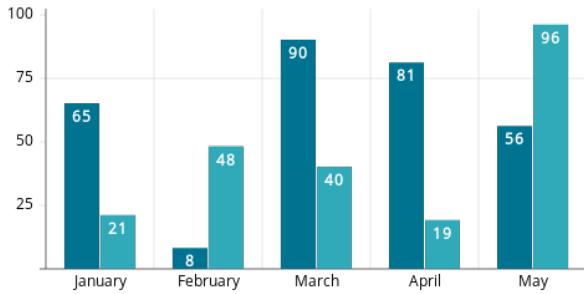
1 //-----
2 // Entitätsidentität - DRY Prinzip
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Entity
8 @Table(name="projects")
9 @Access(AccessType.FIELD)
10 public class Project implements Serializable{
11
12     @GeneratedValue(strategy = GenerationType.AUTO)
13     @Id
14     private Long id;
15
16     @Transient
17     private Long uuid = null;
18
19     @Column(name="title", length = 100, nullable =
20             false)
21     private String title;
22
23     @Column(name="description", length = 4000)
24     private String description;
25
26     @Temporal(TemporalType.DATE)
27     @Column(name = "creationDate")
28     private Date creationDate;
29
30     public Long getUuid(){
31         return (uuid == null) ? uuid =
32                 UUID.randomUUID()
33                         .getLeastSignificantBits() : uuid;
34 }
```

⁴⁶ Don't repeat yourself

```

33     }
34
35     @Override
36     public boolean equals(Object o) {
37         if (this == o) return true;
38         if (!(o instanceof Project)) return false;
39         if (!super.equals(o)) return false;
40
41         Project project = (Project) o;
42
43         if(id == null){
44             return getUuid().equals(((Project)
45                         o).getUuid());
46         }
47
48         return getId().equals(project.getId());
49     }
50
51     @Override
52     public int hashCode() {
53         int result = super.hashCode();
54
55         if(id == null){
56             result = 31 * result +
57                     getUuid().hashCode();
58         }else{
59             result = 31 * result +
60                     getId().hashCode();
61         }
62
63         return result;
64     }
65
66     @NoArgsConstructor
67     @Getter
68     @Setter
69     @Entity
70     @Table(name="subprojects")
71     @Access(AccessType.FIELD)
72     public class Subproject implements Serializable{
73
74         @GeneratedValue(strategy = GenerationType.AUTO)
75         @Id
76         private Long id;
77
78         @Transient
79         private Long uuid = null;
80
81         @Column(name="title", length = 100, nullable =
82                 false)
83         private String subprojectTitle;
84
85         public Long getUuid(){
86             return (uuid == null) ? uuid =
87                     UUID.randomUUID().getLeastSignificantBits()
88                         : uuid;
89         }
89
90         if (this == o) return true;
91         if (!(o instanceof Project)) return false;
92         if (!super.equals(o)) return false;
93
94         Project project = (Project) o;
95
96         if(id == null){
97             return getUuid().equals(((Project)
98                         o).getUuid());
99         }
100
101        return getId().equals(project.getId());
102    }
103
104    @Override
105    public int hashCode() {
106        int result = super.hashCode();
107
108        if(id == null){
109            result = 31 * result +
110                    getUuid().hashCode();
111        }else{
112            result = 31 * result +
113                    getId().hashCode();
114        }
115
116        return result;
116    }

```



► Analyse: DRY Prinzip ▼

- ES IST ANZUMERKEN DASS IN DEN KLASSEN PROJECT.CLASS UND SUBPROJECT.CLASS BESTIMMTE Codeabschnitte WIEDERHOLT AUFTREten.
- DEN PRINZIPIEN DER **objektorientierten Programmierung** FOLGEND, SIND ZU WIEDERHOLENDE CODEABSCHNITTE IN EINER EINZELNEN STRUKTUR ZU **kapseln**.

► Codebeispiel: Entitätsidentität ▾

```

1 //-----
2 // Entitätsidentität - @Mappedsuperclass
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @MappedSuperclass
8 public class abstract AEntity implements
9     Serializable{
10
11     @GeneratedValue(strategy = GenerationType.AUTO)
12     @Id
13     private Long id;
14
15     @Transient
16     private Long uuid = null;
17
18     public Long getUuid(){
19         return (uuid == null) ? uuid =
20             UUID.randomUUID().
21                 getLeastSignificantBits() : uuid;
22     }
23
24     @Override
25     public final boolean equals(Object o) {
26         if (this == o) return true;
27         if (!(o instanceof Project)) return false;
28         if (!super.equals(o)) return false;
29
30         Project project = (Project) o;
31
32         if(id == null){
33             return getUuid().equals(((Project)
34                         o).getUuid());
35         }
36
37         return getId().equals(project.getId());
38     }
39
40     @Override
41     public final int hashCode() {
42         int result = super.hashCode();
43
44         if(id == null){
45             result = 31 * result +
46                 getUuid().hashCode();
47         }else{
48             result = 31 * result +
49                 getId().hashCode();
50         }
51
52         return result;
53     }
54
55     //-----
56     // Project Entitäet
57     //-----
58     @Entity
59     @Table(name="projects")
60     @Access(AccessType.FIELD)
61     public class Project extends AEntity{
62
63         @Column(name="title", length = 100, nullable =
64             false)
65         private String title;
66
67         @Column(name="description", length = 4000)
68         private String description;
69
70         @Temporal(TemporalType.DATE)
71         @Column(name = "creationDate")
72         private Date creationDate;
73
74     //-----
75     // Subproject Entitäet
76     //-----
77     @NoArgsConstructor
78     @Getter
79     @Setter
80     @Entity
81     @Table(name="subprojects")
82     @Access(AccessType.FIELD)
83     public class Subproject extends AEntity{
84
85         @Column(name="title", length = 100, nullable =
86             false)
87         private String subprojectTitle;
88     }

```

► Analyse: AEntity ▾

- **abstract:** WIR MERKEN ALS ERSTES DASS AENTITY abstract IST.

DAS IST NOTWENDIG, DENN WIR WOLLEN NICHT DASS VON DER AENTITY KLASSE Instanzen ANGELEGT WERDEN KÖNNEN.

- **@Entity:** Warum ist AEntity keine @Entity?
WÜRDEN WIR AENTITY MIT DER @ENTITY ANNOTATION VERSEHEN, WÜRDE DAS BEDEUTEN DASS WIR EINE eigene Tabelle HÄTTEN, IN DER WIR DIE ID SPEICHERN UND EINE weitere Tabelle FÜR DIE RESTLICHEN DATEN z.B.: DER PROJEKTDATEN.

Eine derartige Aufteilung der Daten auf mehrere Tabellen würde keinen Sinn machen.

- **@MappedSuperclass:** WIR MÜSSEN DAFÜR SORGEN DASS DIE ID SPALTE FÜR JEDE ENTITÄT IN DIE DATENBANK übernommen WIRD OHNE JEDOCH AUS AENTITY EINE ENTITÄT ZU MACHEN. DAS WIRD MIT DER @MAPPEDSUPERCLASS ANNOTATION ERREICHT.



14.3.6 SQL Datentypen

ORM Systeme legen den **Datentypen** fest, der für die Spalten der Tabellen definiert wird, in denen die **Attribute** der Objekte gespeichert werden.

► Erklärung: SQL Datentypen ▾

- **varchar2:** DAS LENGTH ATTRIBUT DER @COLUMN ANNOTATION BESTIMMT DIE AUSPRÄGUNG DES **varchar2** DATENTYP IN DER DATENBANK.
- **Number:** MIT DEN PRECISION⁴⁷ UND SCALE⁴⁸ ATTRIBUTEN DER @COLUMN ANNOTATION WIRD DER WERTEBEREICH DES **Number** DATENTYP IN DER DATENBANK SPEZIFIZIERT.
- **Date, Time, Timestamp:** MIT HILFE DER @TEMPORAL ANNOTATION WIRD FESTGELEGT, WELCHER DER 3 TYPEN⁴⁹ ZUR SPEICHERUNG VON ZEITANGABEN, DER SPALTE IN DER DATENBANK ZUGEWIESEN WIRD.



► Codebeispiel: SQL Datentypen ▾

```

1  /-----
2  // @Column - SQL Datentypen
3  /-----
4  @NoArgsConstructor
5  @Getter
6  @Setter
7  @Entity
8  @Table(name="projects")
9  @Access(AccessType.FIELD)
10 public class Project extends AEntity{
11
12     @Column(name="title", length = 100)
13     private String title;
14
15     @Column(name="rating", precision = 3,
16             scale = 2)
17     private Float rating;
18
19     @Temporal(TemporalType.DATE)
20     @Column(name="creation_date")
21     private Date createDate;
22
23 }
```

14.3.7 Java Datentypen

In **Entitäten** sollte für die Definition der Datentypen von **Feldern** nur die Komplexe Version der einfachen Datentypen verwendet werden.



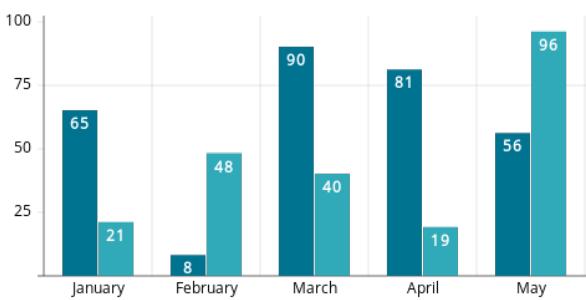
► Erklärung: Java Datentypen ▾

- FÜR JEDEN **einfachen Datentypen** GIBT ES IN JAVA EIN **komplexes Gegenstück**.

- short ⇒ Short
- long ⇒ Long
- int ⇒ Integer
- float ⇒ Float
- double ⇒ Double
- char ⇒ Char
- boolean ⇒ Boolean

- VERWENDEN SIE FÜR DIE **Definition von Attributen** EINER **Entität** EINFACHE DATENTYPEN KANN DAS ZU PROBLEmen FÜHREN.
- EINE DER AUFGABEN VON **ORM** IST ES **Daten** AUS DER DATENBANK **auszulesen** UND ENTSPRECHENDE **Objekte** MIT DEN DATEN ZU INITIALISIEREN. GIBT ES FÜR EINE ZEILE EINEN BESTIMMten SPALTENEINTRAG NICHT SO WIRD DEM ATTRIBUTE DER ENTSPRECHENDEN **Entity** DER WERT **null** ZUGEWIESEN.

Ist der Datentyp des Attributes ein einfacher Datentyp tritt in diesem Fall ein Fehler auf, weil einfachen Datentypen die **Null Referenz** nicht zugewiesen werden kann.



⁴⁷ Anzahl der Stellen insgesamt

⁴⁸ Anzahl der Stellen rechts vom Komma

⁴⁹ Date, Time, Timestamp

14.4. Vererbung von Entitäten

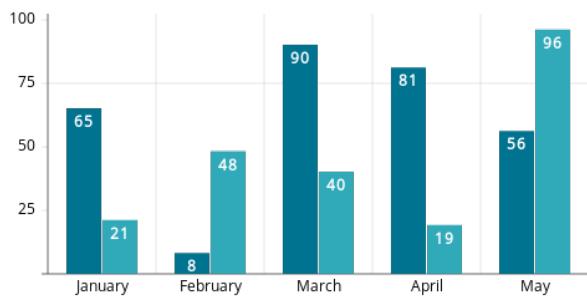
14.4.1 Vererbung und Entitäten - Grundlagen

Vererbung ist eines der grundlegenden Konzepte der objektorientierten Programmierung.

Relationale Datenbanksysteme kennen Vererbung als Konzept nicht. Soll **Vererbung** in der Datenbank abgebildet werden, stehen uns dazu 2 Strategien als Option zur Verfügung.

► Single Table

► Joined Table



14.4.2 Single Table Inheritance



Single Table

Mit der **Single Table** Vererbungsstrategie werden die Attribute der **Basisklasse** als auch alle Felder der **Kindklassen** in einer einzelnen Tabelle gespeichert.

Zusätzlich wird eine einzelne Spalte hinzugefügt um die **Typen** der einzelnen Zeilen unterscheiden zu können.

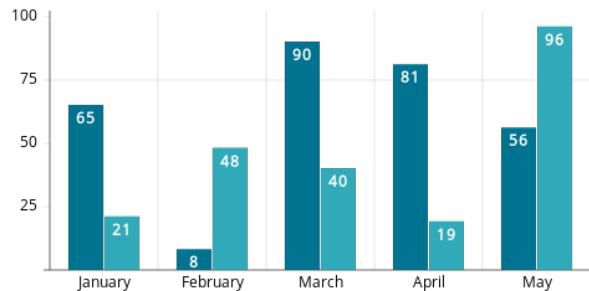
► Codebeispiel: SingleTable Vererbung

```

1 //-----
2 // Singletable Vererbungsstrategie - Basisklasse
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Inheritance(strategy =
8     InheritanceType.SINGLE_TABLE)
9 @DiscriminatorColumn(name = "project_type")
10 @Table(name="projects")
11 @Access(AccessType.FIELD)
12 public abstract class AProject extends AEntity{
13
14     @Column(name="title", length = 100)
15     private String title;
16
17     @Column(name="description", length = 4000)
18     private String description;
19
20     @Temporal(TemporalType.DATE)
21     @Column(name = "creationDate")
22     private Date creationDate;
23
24     @Enumerated(EnumType.STRING)
25     @Column(name = "security_level", length = 10)
26     private EProjectLevel level;
27
28     @Column(name="rating", precision = 3,
29             scale = 2)
29     private Float rating;
30
31 }
32
33 //-----
34 // Kindklasse
35 //-----
36 @NoArgsConstructor
37 @Getter
38 @Setter
39 @DiscriminatorValue("management")
40 @Entity
41 @Access(AccessType.FIELD)
42 public class ManagementProject extends AProject{
43
44     @Column(name="management_note", length = 4000)
45     private String managementNote;
46
47     @Enumerated(EnumType.STRING)
48     @Column(name="indicator")
49     private EManagementIndicator indicator;
50
51 }
52
53 //-----
54 // Kindklasse
55 //-----
56 @NoArgsConstructor
57 @Getter
58 @Setter
59 @DiscriminatorValue("request_funding")
60 @Entity
61 @Access(AccessType.FIELD)
62 public class RequestFundingProject extends
63     AProject{
64
65     @Column(name = "is_ffg_project")
66     private Boolean ffgProject;
67
68     @Column(name = "is_eu_project")
69     private Boolean euProject;
70
71     @Column(name = "is_fwf_project")
72     private Boolean fwfProject;
73
74 }
```

► Erklärung: SingleTable Vererbung ▼

- OBJEKTE DER KLASSEN MANAGEMENTPROJECT, REQUESTFUNDINGPROJECT DER ENTITÄTEN WERDEN IN EINER EINZELNEN TABELLE PROJECTS GESPEICHERT.
- DAMIT ENTHÄLT DIE TABELLE PROJECTS FÜR JEDES Attribut DER KLASSEN APROJECT, MANAGEMENTPROJECT, REQUESTFUNDINGPROJECT EINE EIGENE Spalte.
- UM UNTERSCHIEDEN ZU KÖNNEN OB ES SICH BEI DEM DATENSATZ UM EIN MANAGEMENTPROJECT ODER UM EIN REQUESTFUNDINGPROJECT HANDELT, WIRD IN DER PROJECTS TABELLE EINE EIGENE Spalte PROJECT_TYPE HINZUGEFÜGT.
- DIE PROJECT_TYPE SPALTE WIRD MIT DER @DISCRIMINATORCOLUMN ANNOTATION IN DER Basisklasse APROJECT DEFINIERT.
- MIT DER @DISCRIMINATORVALUE ANNOTATION WIRD FÜR JEDEN Kindklasse FESTGELEGT, MIT WELCHEM WERT DIE KINDKLASSE IN DER @DISCRIMINATORCOLUMN SPALTE IDENTIFIZIERT WIRD.



14.4.3 Joined Table Inheritance



Joined Table ▼

Mit der Joined Table VERERBUNGSSTRATEGIE WERDEN DIE FELDER DER Basisklasse IN EINER TA-BELLE GRUPPIERT, WÄHREND FÜR JEDE KINDKLASSE EINE EIGENE TABELLE ANGELEGT WIRD.

► Codebeispiel: JoinedTable Vererbung ▼

```

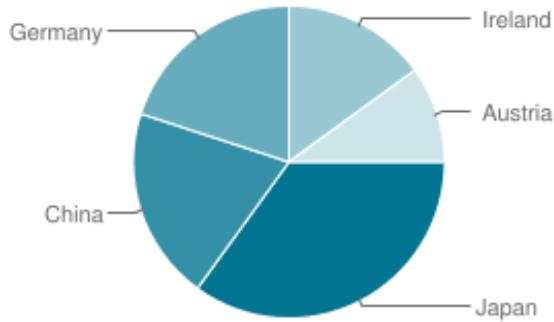
1 //-----
2 // Joined Table - Basisklasse
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Inheritance(strategy = InheritanceType.JOINED)
8 @Entity
9
10 @Table(name="projects")
11 @Access(AccessType.FIELD)
12 public abstract class AProject extends AEntity{
13
14     @Column(name="title", length = 100)
15     private String title;
16
17     @Column(name="description", length = 4000)
18     private String description;
19
20     @Temporal(TemporalType.DATE)
21     @Column(name = "creationDate")
22     private Date creationDate;
23
24     @Enumerated(EnumType.STRING)
25     @Column(name = "security_level", length = 10)
26     private EProjectLevel level;
27
28     @Column(name="rating", precision = 3,
29             scale = 2)
30     private Float rating;
31 }
32
33
34 //EManagementIndicator
35 public enum EManagementIndicator{
36     INTERN, EXTERN;
37 }
38
39 //-----
40 // Kindklasse
41 //-----
42 @NoArgsConstructor
43 @Getter
44 @Setter
45 @Entity
46 @Table(name = "management_projects")
47 @Access(AccessType.FIELD)
48 public class ManagementProject extends AProject{
49
50     @Column(name="management_note", length = 4000)
51     private String managementNote;
52
53     @Enumerated(EnumType.STRING)
54     @Column(name="indicator")
55     private EManagementIndicator indicator;
56 }
57
58 //-----
59 // Kindklasse
60 //-----
61
62 @NoArgsConstructor
63 @Getter
64 @Setter
65 @Entity
66 @Table(name = "request_funding_projects")
67 @Access(AccessType.FIELD)
68 public class RequestFundingProject extends
69     AProject{
70
71     @Column(name = "is_ffg_project")

```

```

71     private Boolean ffgProject;
72
73     @Column(name = "is_eu_project")
74     private Boolean euProject;
75
76     @Column(name = "is_fwf_project")
77     private Boolean fwfProject;
78
79 }

```



► Erklärung: JoinedTable Vererbung ▼

- DIE **Daten** JEDES PROJEKT **Objekts** WERDEN AUF 2 TABELLEN AUFGETEILT. DIE **Attribute** DER BASISKLASSE WERDEN IN DER **PROJECTS** TABELLE GESPEICHERT.
- ABHÄNGIG VOM **Typ** DES **OBJEKTS**⁵⁰ WERDEN DIE RESTLICHEN ATTRIBUTE ENTWEDER IN DER **MANAGEMENT_PROJECTS** ODER DER **REQUEST_FUNDING_PROJECTS** TABELLE GESPEICHERT.



⁵⁰ ManagementProject, ResearchFundingProject

Anwendungsconstraint	Beschreibung	Datentypen
<code>@NotNull</code>	DER IM ATTRIBUT GESPEICHERTE WERT DARF nicht null SEIN.	Object
<code>@Null</code>	DER IM ATTRIBUT GESPEICHERTE MUSS null SEIN.	Object
<code>@Size</code>	MIT DEN MIN UND MAX ATTRIBUTEN DER <code>@SIZE</code> ANNOTATION KANN GENAU SPEZIFIZIERT WERDEN AUS WIEVIEL ZEICHEN DER STRING BESTEHT.	String
<code>@Size</code>	MIT DER <code>@SIZE</code> ANNOTATION KÖNNEN SIE FESTLEGEN WIEVIEL ELEMENTE DIE Collection HABEN KANN.	Collection
<code>@AssertTrue</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT true IST.	Boolean
<code>@AssertFalse</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT false IST.	Boolean
<code>@Future</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT IN DER Zukunft LIEGT.	Date
<code>@Past</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT IN DER Vergangenheit LIEGT.	Date
<code>@Min</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT klein GENUNG IST.	Number
<code>@Max</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT groß GENUG IST	Number
<code>@DecimalMin</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT klein GENUNG IST.	Decimal
<code>@DecimalMax</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT groß GENUNG IST.	Decimal
<code>@Digits</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT IN BE-STIMMten Schranken LIEGT	Decimal
<code>@Pattern</code>	ÜBERPRÜFT OB DER IM ATTRIBUT GESPEICHERTE WERT DER Regular Expression ENTSPRICHT.	String

Abbildung 24. Validierung von Datenwerten - Anwendungsconstraints

14.5. Validieren von Daten

14.5.1 Constraints

Constraint ▾

Mit **Constraints** werden in Datenbanken **Bedingungen** definiert, die zwingend vom Wert einer Variable erfüllt werden müssen, damit der Wert in die Datenbank übernommen werden kann.

Constraints werden mit der Tabelle gespeichert, für die sie definiert wurden.

► Erklärung: **NotNull Constraint** ▾

- Der **notnull Constraint** stellt sicher das für den Spaltenwert der Tabelle **keine null** Werte einge-tragen werden können.
- Der **Constraint** wird definiert indem in der `@COLUMN` ANNOTATION das `NULLABLE` ATTRIBUT auf `FALSE` ge-setzt wird.
- In der Datenbank wird ein entsprechender **Not-Null Constraint** generiert.



► Codebeispiel: NotNull Constraint ▼

```

1 //-----
2 // Datenbankconstraint - NotNull
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Inheritance(strategy = InheritanceType.JOINED)
8 @Entity
9 @Table(name="projects")
10 @Access(AccessType.FIELD)
11 public abstract class AProject extends AEntity{
12
13     @Column(name="title", length = 100, nullable =
14             false)
15     private String title;
16
17     @Column(name="project_code", length="8",
18             unique = true, nullable=false)
19     private String projectCode;
20
21     @Column(name="description", length = 4000)
22     private String description;
23
24     @Temporal(TemporalType.DATE, , nullable =
25             false)
26     @Column(name = "creationDate")
27     private Date creationDate;
28
29     @Enumerated(EnumType.STRING)
30     @Column(name = "security_level", length = 10,
31             nullable = false)
32     private EProjectLevel level;
33
34     @Column(name="rating", precision = 3,
35             scale = 2)
36     private Float rating;
37 }
38
39 public enum EManagementIndicator{
40     INTERN, EXTERN;
41 }
42 @NoArgsConstructor
43 @Getter
44 @Setter
45 @Entity
46 @Table{name = "management_projects"}
47 @Access(AccessType.FIELD)
48 public class ManagementProject extends AProject{
49
50     @Column(name="management_note", length = 4000)
51     private String managementNote;
52
53     @Enumerated(EnumType.STRING)
54     @Column(name="indicator", nullable = false)
55     private EManagementIndicator indicator;
56 }
57

```

► Erklärung: Unique Constraint ▼

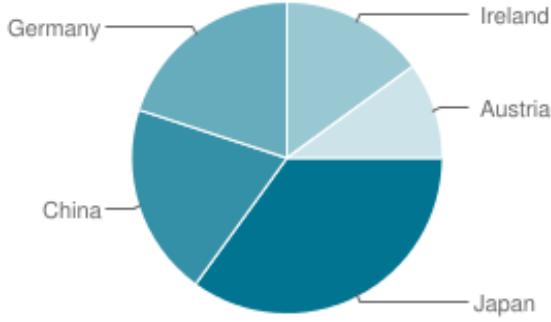
- DER **unique Constraint** STELLT SICHER DASS ALLE Einträge FÜR DIE DEFINIERTE SPALTE UNTERSCHIEDLICH SIND.
- DER **Constraint** WIRD definiert INDEM IN DER @COLUMN ANNOTATION DAS **unique** ATTRIBUT AUF **TRUE** GESETZT WIRD.

► Codebeispiel: Unique Constraint ▼

```

1 //-----
2 // Datenbankconstraint - unique
3 //-----
4 public enum EProjectLevel{
5     PUBLIC, RESTRICTED, PRIVATE;
6 }
7
8 @NoArgsConstructor
9 @Getter
10 @Setter
11 @Inheritance(strategy = InheritanceType.JOINED)
12 @Entity
13 @Table(name="projects")
14 @Access(AccessType.FIELD)
15 public abstract class AProject extends AEntity{
16
17     @Column(name="title", length = 100, nullable =
18             false, unique = true)
19     private String title;
20
21     @Column(name="description", length = 4000)
22     private String description;
23
24     @Temporal(TemporalType.DATE, , nullable =
25             false)
26     @Column(name = "creationDate")
27     private Date creationDate;
28
29     @Enumerated(EnumType.STRING)
30     @Column(name = "security_level", length = 10)
31     private EProjectLevel level;
32
33     @Column(name="rating", precision = 3,
34             scale = 2)
35     private Float rating;
36 }
37
38 public enum EManagementIndicator{
39     INTERN, EXTERN;
40 }
41 @NoArgsConstructor
42 @Getter
43 @Setter
44 @Entity
45 @Table{name = "management_projects"}
46 @Access(AccessType.FIELD)
47 public class ManagementProject extends AProject{
48
49     @Column(name="management_note", length = 4000)
50     private String managementNote;
51
52     @Enumerated(EnumType.STRING)
53     @Column(name="indicator", nullable = false)
54     private EManagementIndicator indicator;
55 }
56

```



```

50      @Column(name="management_note", length = 4000)
51      private String managementNote;
52
53      @Enumerated(EnumType.STRING)
54      @Column(name="indicator", nullable = false)
55      private EManagementIndicator indicator;
56
57  }
58

```

```

11      @Inheritance(strategy = InheritanceType.JOINED)
12      @Entity
13      @Table(name="projects")
14      @Access(AccessType.FIELD)
15      public abstract class AProject extends AEntity{
16
17          @NotNull
18          @Size(min = 3, max = 100)
19          @Column(name="title", length = 100, nullable =
20                  false, unique = true)
21          private String title;
22
23          @Size(max = 4000)
24          @Column(name="description", length = 4000)
25          private String description;
26
27          @NotNull
28          @Past
29          @Temporal(TemporalType.DATE, nullable = false)
30          @Column(name = "creationDate")
31          private Date creationDate;
32
33          @NotNull
34          @Enumerated(EnumType.STRING)
35          @Column(name = "security_level", length = 10,
36                  nullable = true)
37          private EProjectLevel level;
38
39          @DecimalMin(0)
40          @DecimalMax(5)
41          @Digits(integer = 1, fraction = 2)
42          @Column(name="rating", precision = 3,
43                  scale = 2)
44          private Float rating;
45
46      public enum EManagementIndicator{
47          INTERN, EXTERN;
48      }
49
50      @NoArgsConstructor
51      @Getter
52      @Setter
53      @Entity
54      @Table{name = "management_projects"}
55      @Access(AccessType.FIELD)
56      public class ManagementProject extends AProject{
57
58          @Size(max = 4000)
59          @Column(name="management_note", length = 4000)
60          private String managementNote;
61
62          @NotNull
63          @Enumerated(EnumType.STRING)
64          @Column(name="indicator", nullable = false)
65          private EManagementIndicator indicator;
66
67      }
68
69      @NoArgsConstructor
70      @Getter
71      @Setter

```

14.5.2 Anwendungsconstraints

Werden **Datenwerte** in die Datenbank geschrieben muß sichergestellt werden daß die Werte **sinnvoll** im Sinne der Anwendung sind. In der **Datenbankschicht** wird das zu einem bestimmten Teil mit **Constraints** sichergestellt.

Fehler in den Datenwerten sollten bereits gefunden werden **bevor** sie an die **Datenbankschicht** übermittelt werden.

 **Anwendungsconstraints** ▾

Anwendungsconstraints DEFINIEREN Restriktionen FÜR DIE ATTRIBUTE VON OBJEKten. ANWENDUNGSCONSTRAINTS ERLAUBEN DEN Zustand VON OBJEKten ZU BESCHREIBEN.

► **Codebeispiel: Validieren von Datenwerten** ▾

```

1  //-----
2  //  Anwendungsconstraints
3  //-----
4  public enum EProjectLevel{
5      PUBLIC, RESTRICTED, PRIVATE;
6  }
7
8  @NoArgsConstructor
9  @Getter
10 @Setter

```

```
72  @Entity
73  @Table(name = "request_funding_projects")
74  @Access(AccessType.FIELD)
75  public class RequestFundingProject extends
    AProject{
76
77      @NotNull
78      @Column(name = "is_ffg_project", nullable =
        false)
79      private Boolean ffgProject = false;
80
81      @NotNull
82      @Column(name = "is_eu_project", nullable =
        false)
83      private Boolean euProject = false;
84
85      @NotNull
86      @Column(name = "is_fwf_project", nullable =
        false)
87      private Boolean fwfProject = false;
88
89 }
```



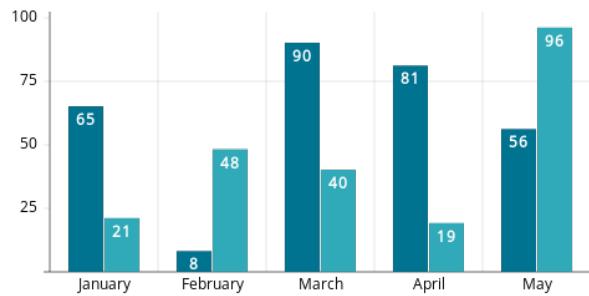
14.6. Entitätskomposition

14.6.1 Objektkomposition

Zur Laufzeit verwalten Objekte **Referenzen** auf andere Objekte. Dieses Konzept bezeichnen wir als **Objektkomposition**. **Objektkomposition** wird in der Datenbank durch **Relationen** umgesetzt.

In einer **Datenbank** unterscheiden wir 3 Arten von **Relationen**:

- **1:1 Relation**
- **1:n Relation**
- **n:m Relation**



14.6.2 Objektkomposition - 1:1 Relation

Jede **Entität** der ersten Entitätsmenge steht mit genau einer Entität der zweiten Entitätsmenge in **Beziehung**.

► Erklärung: 1:1 Relation ▾

- **@OneToOne:** DIE `@ONETOONE` ANNOTATION LEGT EINE **1:1 Relation** ZWISCHEN 2 **Entitäten** FEST.
- **@JoinColumn:** DIE `@JOINCOLUMN` ANNOTATION DEFINIERT EINE **Fremdschlüsselspalte**.



► Codebeispiel: 1:1 Relation ▾

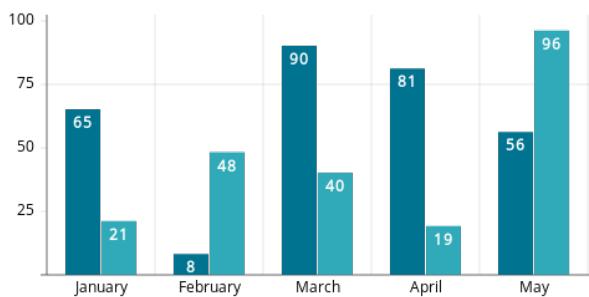
```

1 //-----
2 // @OneToOne - 1..1 Relation
3 //-----
4 public enum EProjectLevel{
5     PUBLIC, RESTRICTED, PRIVATE;
6 }
7
8 @NoArgsConstructor
9 @Getter
10 @Setter
11
12 @Entity
13 @Table(name="project_descriptions")
14 @Access(AccessType.FIELD)
15 public class ProjectDescription extends AEntity{
16
17     @Size(max = 4000)
18     @Column(name="description_de", length = 4000)
19     private String descriptionDE;
20
21     @Size(max = 4000)
22     @Column(name="description_en", length = 4000)
23     private String descriptionEN;
24
25     @Size(max = 4000)
26     @Column(name="description_sp", length = 4000)
27     private String descriptionSP;
28 }
29
30
31 @NoArgsConstructor
32 @Getter
33 @Setter
34 @Inheritance(strategy = InheritanceType.JOINED)
35 @Entity
36 @Table(name="projects")
37 @Access(AccessType.FIELD)
38 public abstract class AProject extends AEntity{
39
40
41     @Column(name="description", length=4000)
42     private String description;
43
44     @NotNull
45     @OneToOne
46     @JoinColumn(name="description_id", nullable =
47         false)
48     private ProjectDescription description;
49
50     ...
51 }
52
53
54
55 public enum EManagementIndicator{
56     INTERN, EXTERN;
57 }
58
59 @NoArgsConstructor
60 @Getter
61 @Setter
62 @Entity
63 @Table{name = "management_projects"}
64 @Access(AccessType.FIELD)
65 public class ManagementProject extends AProject{
66
67     ...
68 }
69

```

► Analyse: 1:1 Relation ▼

- ZWISCHEN DER PROJECTDESCRIPTION UND APROJECT ENTITÄT SOLL EINE **1:1 Relation** ETABLIERT WERDEN.
- DAZU VERWENDEN WIR IN APROJECT DIE @ONETOONE UND @JOINCOLUMN ANNOTATION.
- WIR FORDERN ZUSÄTZLICH DASS JEDESMAL WENN EIN APROJECT OBJEKT **gespeichert** WIRD, INNERHALB DES OBJEKTS DAS DESCRIPTION ATTRIBUT **gesetzt**⁵¹ IST.



► Codebeispiel: n:1 Relation ▼

```

1 //-----
2 // @ManyToOne - n..1 Relation
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Inheritance(strategy = InheritanceType.JOINED)
8 @Entity
9 @Table(name="projects")
10 @Access(AccessType.FIELD)
11 public abstract class AProject extends AEntity{
12
13     ...
14 }
15
16 @NoArgsConstructor
17 @Getter
18 @Setter
19 @Entity
20 @Table(name = "subprojects")
21 @Access(AccessType.FIELD)
22 public class Subproject extends AEntity{
23
24     @NotNull
25     @ManyToOne
26     @JoinColumn(name = "project_id", nullable =
27         false)
28     private AProject project;
29
30 }
```

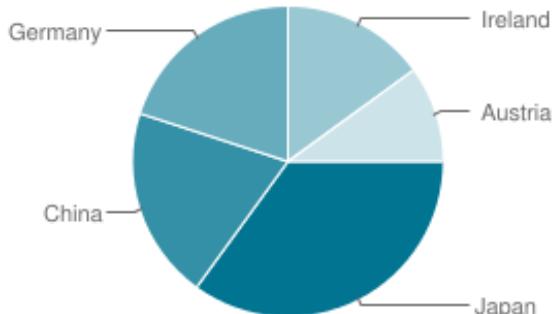
14.6.3 Objektkomposition - 1:n Relation

Jede **Entität** der ersten Entitätsmenge steht mit mindestens einer Entität der zweiten Entitätsmenge in **Beziehung**. Jede Entität der zweiten Entitätsmenge steht mit genau einer Entität der ersten **Entitätsmenge** in Beziehung.



► Erklärung: n:1 Relation ▼

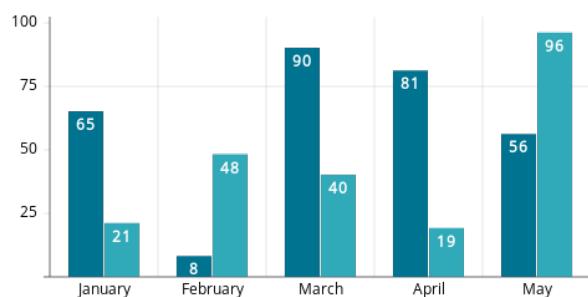
- @OneToMany:** DIE @ONETOMANY ANNOTATION DEKLARIERT EINE **1:n Relation** ZWISCHEN 2 **Entitäten**.
- @ManyToOne:** DIE @MANYTOONE ANNOTATION DEKLARIERT EINE **n:1 Relation** ZWISCHEN 2 **Entitäten**.
- @JoinColumn:** MIT HILFE DER @JOINCOLUMN ANNOTATION KÖNNEN WIR EINE SPALTE ALS **Fremdschlüssel** DEFINIEREN.



⁵¹ Die Verwendung des nullable Attributs ist wie bei @Column Annotation optional

► Erklärung: 1:n Relation ▼

- WIR MÖCHTEN EINE **1:n Relation** ZWISCHEN 2 ENTITÄTEN ETABLIEREN: EIN **Projekt** BESTEHT AUS MEHREREN **Subprojekten**.
- WIR BILDEN DAS AB INDEM JEDES **SUBPROJECT** OBJEKT EINE REFERENZ AUF EIN **PROJECT** OBJEKT SPEICHERT.
- MIT HILFE DER **@MANYTOONE** UND **@JOINCOLUMN** ANNOTATIONEN WIRD DIESER SACHVERHALT IN DER DATENBANK ABGEBILDET.



► Codebeispiel: 1:n Relation ▼

```

1 //-----
2 // @OneToMany - 1..n Relation
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Inheritance(strategy = InheritanceType.JOINED)
8 @Entity
9 @Table(name="projects")
10 @Access(AccessType.FIELD)
11 public abstract class AProject extends AEntity{
12
13     @OneToMany
14     @JoinColumn(name="PROJECT_ID")
15     private List<Subproject> subprojects = new
16     ArrayList<>();
17 }
18
19 @NoArgsConstructor
20 @Getter
21 @Setter
22 @Entity
23 @Table(name = "subprojects")
24 @Access(AccessType.FIELD)
25 public class Subproject extends AEntity{
26
27     ...
28 }
29
30 }
```

□

14.6.4 Objektkomposition - n:m Relation auflösen mit einer Zwischentabelle

Jede **Entität** der ersten Entitätsmenge steht mit mindestens einer Entität der zweiten Entitätsmenge in **Beziehung**, und umgekehrt.

► Erklärung: n:m Relation ▼

- **m:n Relation:** MÖCHTEN WIR IN DER DATENBANK EINE **n:m Relation** ABBILDEN, LÖSEN WIR DIE RELATION MIT EINER **Zwischentabelle** AUF.
- **Zwischenentität:** WIR VERFOLGEN DIESEN ANSATZ EBENFALLS IN DER **Modellschicht**. WIR FÜHREN EINE NEUE **Entität** EIN MIT DER WIR DIE **Zwischentabelle** ABBILDEN.

► Codebeispiel: n:m Relation ▼

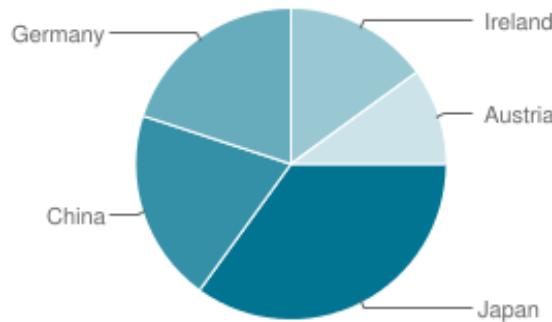
```

1 //-----
2 // n..m Relation - Zusammengesetzter Schluessel
3 //-----
4 @Embeddable
5 @AllArgsConstructor
6 @NoArgsConstructor
7 @Getter
8 public class ProjectPartnerId implements
9     Serializable{
10
11     @NotNull
12     @ManyToOne
13     @JoinColumn(name = "project_id", nullable =
14         false)
15     private AProject project;
16
17     @NotNull
18     @ManyToOne
19     @JoinColumn(name = "partner_id", nullable =
20         false)
21     private Partner partner;
22
23 }
24
25 //-----
26 // n..m Relation - JoinTable Klasse
27 //-----
28 @RequiredArgsConstructor
29 @NoArgsConstructor
30 @Getter
31 @Setter
32 @Entity
33 @Table(name = "project_partners")
34 @Access(AccessType.FIELD)
35 public class ProjectPartner implements
36     Serializable{
37
38     @EmbeddedId
39     private ProjectPartnerID projectPartnerId;
```

```

39 }
40
41 @NoArgsConstructor
42 @Getter
43 @Setter
44 @Inheritance(strategy = InheritanceType.JOINED)
45 @Entity
46 @Table(name="projects")
47 @Access(AccessType.FIELD)
48 public abstract class AProject extends AEntity{
49
50     ...
51
52 }
53
54 @NoArgsConstructor
55 @Getter
56 @Setter
57 @Table(name="partners")
58 @Access(AccessType.FIELD)
59 public class Partner extends AEntity{
60
61     ...
62
63 }

```



14.6.5 Bidirektionale Relationen abbilden

Auf **Objektebene** kann eine einzelne Relation **bidirektional** abgebildet werden. Damit das System eine Relation als bidirektional erkennt wird das `mappedBy` Schlüsselwort verwendet.

► Codebeispiel: 1:n Relation ▾

```

1 //-----
2 // Bidirektionales Mapping
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Inheritance(strategy = InheritanceType.JOINED)
8 @Entity
9 @Table(name="projects")
10 @Access(AccessType.FIELD)
11 public abstract class AProject extends AEntity{
12
13     @OneToMany(mappedBy="project")
14     private List<Subproject> subprojects = new
15         ArrayList<>();
16
17
18
19 //Subproject Class
20 @NoArgsConstructor
21 @Getter
22 @Setter
23 @Entity
24 @Table(name = "subprojects")
25 @Access(AccessType.FIELD)
26 public class Subproject extends AEntity{
27
28     @ManyToOne
29     @JoinColumn(name="PROJECT_ID")
30     private AProject project;
31
32 }

```

► Erklärung: n:m Relation ▾

- WIR MÖCHTEN FÜR UNSER **Projekt** MEHRERE **Partner** SPEICHERN DIE DAS PROJEKT UNTERSTÜTZEN.
- EIN PARTNER UNTERSTÜTZT JEDOCH NICHT EXKLUSIV EIN EINZELNES PROJEKT SONDER UNTERSTÜTZT MEHRERE PROJEKTE
- UM DIESEN SACHVERHALT DARZUSTELLEN ERZEUGEN WIR EINE **neue Entität ProjectPartner** EIN.
- DIE **ProjectPartner** ENTITÄT SPEICHERT JEWELLS EINE **Referenz** ZU **APROJECT** UND **PARTNER**.

14.7. Auditing von Daten

14.7.1 Auditing Grundlagen

Auditing

Auditing beschreibt die automatische Protokollierung von Änderungen des Datenbestands.

► Erklärung: Auditing ▾

- IM RAHMEN DES **Auditing** WIRD FÜR JEDEN TABELLE, DEREN DATEN BEOBSCHAUET WERDEN SOLLEN, EINE **Schattentabelle** ANGELEGT.
- WIRD EIN **NEUER Datensatz** ANGELEGT, WIRD DER DATENSATZ EBENFALLS IN DER **Schattentabelle** EINGETRAGEN, GEMEINSAM MIT DEM DATUM, AN DEM ER ERSTELLT WORDEN IST.
- WIRD EIN **Datensatz** VERÄNDERT, WIRD DER GEÄNDERTE DATENSATZ IN DER **Schattentabelle** EINGETRAGEN, GEMEINSAM MIT EINEM DATUM ANDEM ER VERÄNDERT WORDEN IST.

Das selbe gilt wenn Daten **gelöscht** werden.

14.7.2 Auditing in Spring

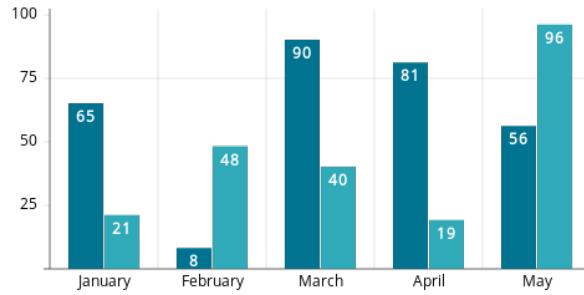
Um Auditing in einer Spring Anwendung zu aktivieren muß das System entsprechend **konfiguriert** werden.



► Erklärung: Konfiguration ▾

- **maven:** Die `HIBERNATE-ENVERS` Dependency MUSS IM CLASSPATH LIEGEN.
- **application.properties:** Die `ENVERS.PROPERTIES` MÜSSEN IN DIE `APPLICATION.PROPERTIES` AUFGENOMMEN WERDEN.
- **@Audited:** Damit die Daten einer **Tabelle** einem **Auditing** unterzogen werden sollen, wird auf KLASSENEBENE bzw. VARIABLEN EINE DIE ANNOTATION `@AUDIT` HINZUGEFÜGT.
- **@NotAudited:** Wird eine **Entity** mit der `@AUDITED` ANNOTATION AUF **Klassenebene** ANNOTIERT, SIND ALLE **Variablen** DER ENTITÄT EINEM AUDITING UNTERZOGEN.

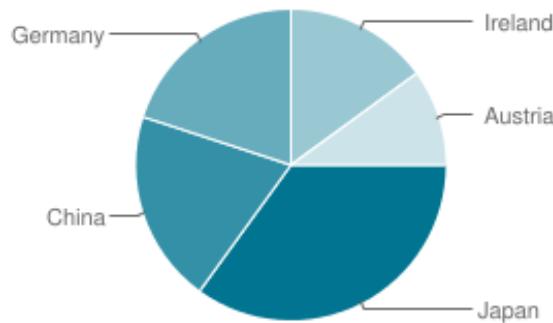
Soll eine bestimmte Variable nicht einem Auditing unterzogen werden, muß sie mit der `@NotAudited` Annotation annotiert werden.



► Codebeispiel: Auditing von Entitäten ▼

```

1  -----
2  // @Audited
3  -----
4  @NoArgsConstructor
5  @Getter
6  @Setter
7  @Audited
8  @Entity
9  @Table(name = "subprojects")
10 @Access(AccessType.FIELD)
11 public class Subproject implements Serializable{
12
13     @NotAudited
14     @GeneratedValue(GenerationStrategy.Auto)
15     @Id
16     private Long id;
17
18     @NotNull
19     @Size(max=30)
20     @Column(name="TITEL", unique=true,
21             nullable=false, length=30)
22     private String titel;
23
24     @Size(max=4000)
25     @Column(name="TITEL", length=4000)
26     private String description;
27
28 }
```



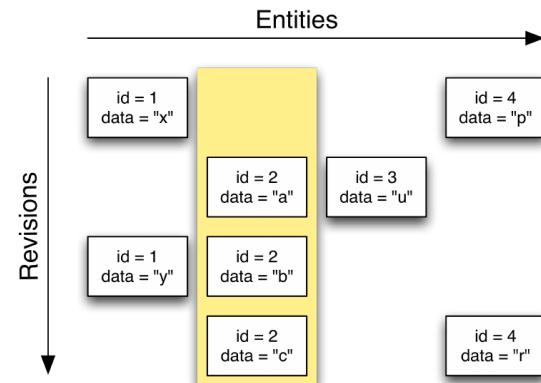
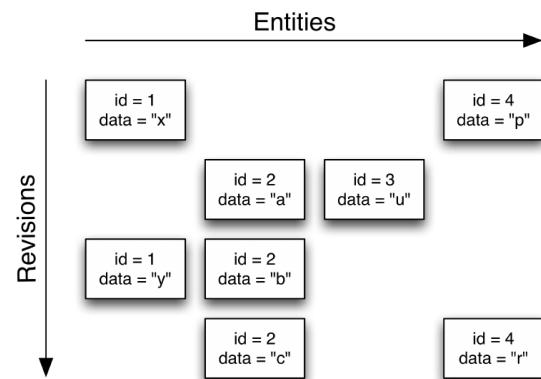
14.7.3 Historisierungsdiagramm

Wird eine **Entität** einem **Auditing** unterzogen existieren in der Datenbank mehrere **Versionen** dieser Entität.

Jede dieser Versionen war zu einem bestimmten Zeitpunkt die **aktuelle Version** der Entität. Mit diesem Konzept kann herausgefunden werden, welche Daten zu einem bestimmten **Zeitpunkt** in einer Entität gespeichert waren. Damit wird eine lückenlose **Historierung** der Daten erreicht.

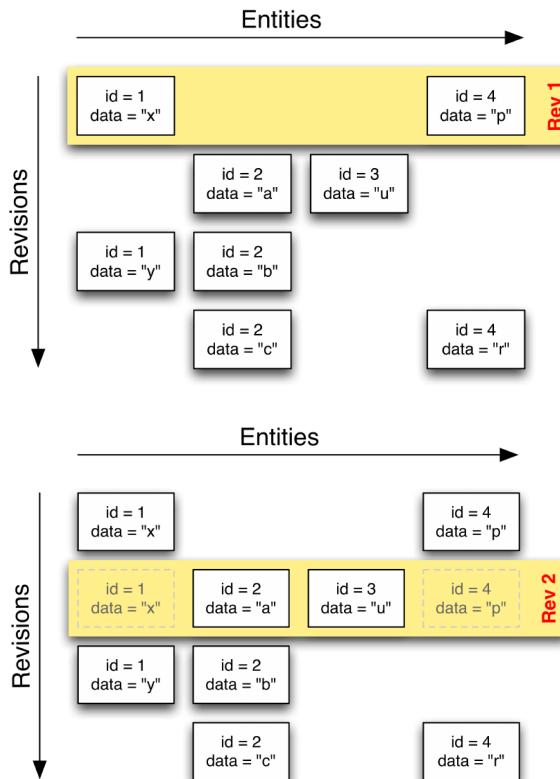
► Erklärung: Historisierungsdiagramm ▼

- DAS **Historisierungsdiagramm** WIRD VERWENDET UM DIE **Datenänderungen**, DER DEM AUDITING UNTERZOGENEN ENTITÄTEN BILDLICH **darzustellen**.
- IN **vertikaler** RICHTUNG WERDEN DIE ÄNDERUNGEN EINER BESTIMMTEN **Entität** DARGESTELLT.
- IN **horizontaler** RICHTUNG WERDEN ALLE DEM AUDITING UNTERZOGENEN ENTITÄTEN MIT DEM ZU DIESEM ZEITPUNKT GESPEICHERTEN DATEN DARGESTELLT.



14.7.4 Revisions und Entities

Eine **Revision** steht für eine bestimmte **Zeitperiode**. Im Historisierungsdiagramm werden **Revisionen** horizontal eingetragen. Ändern sich die **Daten**, der dem Monitoring unterzogenen Entitäten, wird eine neue Revision angelegt. Die Datenänderungen in einer einzelnen Revision werden immer einer einzelnen **Transaktion** der Anwendung zugeordnet.



► Analyse: Revisionen ▾

- **Revision 1:** IN DER ERSTEN REVISION SIND 2 ENTITÄTEN ENTHALTEN.

- Entity 1: id=1, data=x
- Entity 4: id=4, data=p

- **Revision 2:** ES WIRD EINE NEUE REVISION ANGELEGT WEIL 2 NEUE ENTITÄTEN IN DER DATENBANK ERZEUGT WORDEN SIND.

- Entity 1: id=1, data=x
- Entity 2: id=2, data=a
- Entity 3: id=3, data=u
- Entity 4: id=4, data=p



15. Service - Domainschicht Programmierung

02

Domäinschicht

01. Struktur eines Services	115
02. Programmierung der Domäne	116
03. JPA Query Language	125
04. QueryDSL	??
05. Fetchgraphen	127
06. Cascading Operations	130
07. Transaktionsverwaltung	131
08. Eventhandling	134
09. Caching	136

15.1. Struktur eines Services

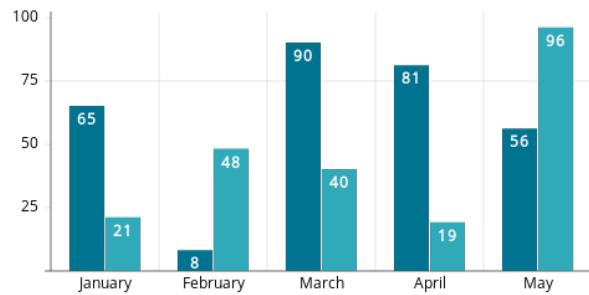


Service ▾

EIN Service IST EINE Softwarekomponente, AUSGEFÜHRT IN EINEM EIGENEN Betriebssystemprozess.

► Erklärung: SOA System ▾

- EINE SOA Anwendung IST EINE Komposition VON Servicen.
- SOA Systeme VERBINDELN SERVICE ZU EINEM Gesamtsystem.
- EIN Service Besteht in der Regel aus 3 Schichten.
- JEDE DER Schichten ERfüLLT EINE EIGENE AUFGABE.



15.1.1 Schichten eines Services

Ein Service ist softwaretechnisch in Schichten aufgeteilt. In der Regel unterscheiden wir für ein Microservice 3 Schichten.

► Erklärung: Schichten eines Service ▾

- **Modelschicht:** DIE Modelschicht ENTSPRICHT DER Struktur DES Datarepository⁵² DES SERVICES.

Die Schicht beinhaltet jene Klassen⁵³, die notwendig sind um Daten aus dem Datarepository zu kapseln. Klassen der Modelschicht implementieren kein Verhalten.

⁵² SQL Datenbank, NoSQL Datenbank, Dateien usw.

⁵³ Objekte deren Aufgabe es ist in erster Linie Daten zu transportieren werden DataTransferObjects genannt

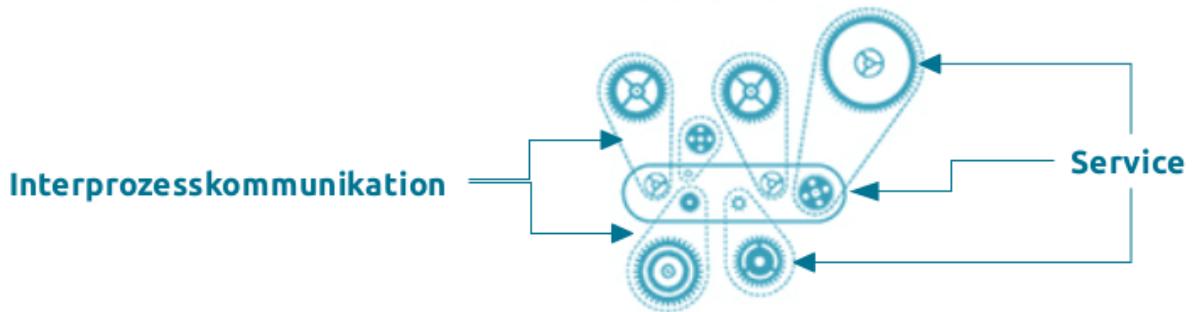


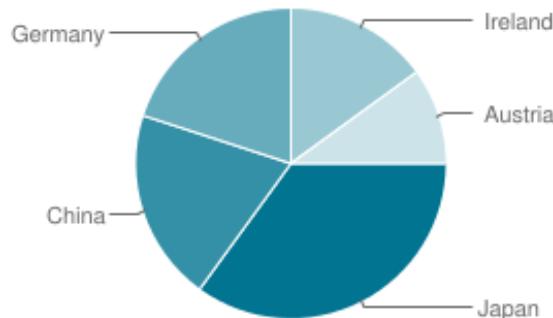
Abbildung 25. SOA - Anwendung

- **Domainschicht:** DIE Domainschicht BEINHALTET DIE Logik DES Services.

Die Schicht kapselt die **datenlesende-** und **datenschreibende** Funktionalität des Services. Die Klassen der Domäinschicht verwalten in der Regel keinen Zustand sondern stellen **Verhalten** zur Verfügung.

- **Serviceschicht:** DIE Serviceschicht IST VERANTWORTLICH FÜR DIE **Kommunikation** DER SERVICE UNTEREINANDER.

Die **Kommunikation** der Service erfolgt über **plattformunabhängige**⁵⁴ Protokolle. Damit ist es möglich Service in unterschiedlichen **Technologien** zu implementieren und sie trotzdem zu einer Anwendung zu **integrieren**.



⁵⁴ *Http Protokoll, SOAP usw.*

15.2. Programmierung der Domäne ▾



Domainschicht ▾

DIE Domäinschicht KAPSELT DIE datenlesende- UND datenschreibende FUNKTIONALITÄT DES SERVICES. DIE KLASSEN DER SCHICHT VERWALTEN IN DER REGEL KEINEN ZUSTAND SONDERN STELLEN **Verhalten** ZUR VERFÜGUNG.

15.2.1 Die JPARepository Schnittstelle

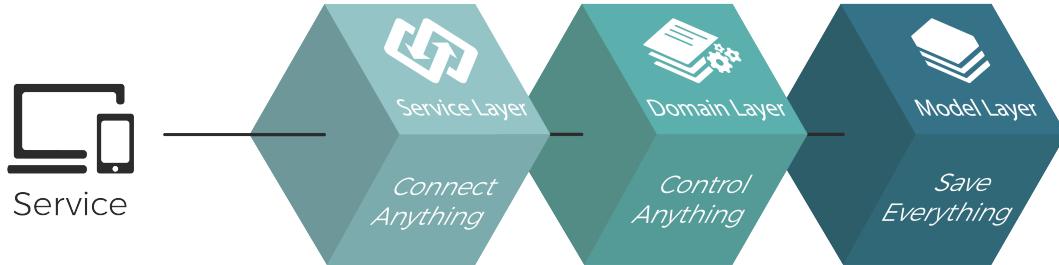
Im **Spring Data Modul** ist die **JPARepository** Schnittstelle die zentrale **Schnittstelle** für die Implementierung des Datenbankzugriffs.



► Codebeispiel: Methoden des JPARepository ▾

```

1 //-----
2 // JPARepository Schnittstelle
3 //-----
4 public interface JpaRepository<T, ID extends
5   Serializable> extends
6   PagingAndSortingRepository<T, ID>,
7   QueryByExampleExecutor<T> {
8
9   /**
10    * Returns all instances of the type
11    */
12   List<T> findAll();
13
14   /**
15    * Returns all entities sorted by the given
16    * options.
17    */
18   List<T> findAll(Sort var1);
19
20 }
```



```

17 /**
18 * Returns all instances of the type with the
19 * given IDs.
20 */
21 List<T> findAll(Iterable<ID> var1);
22
23 /**
24 * Saves a given entity. Use the returned
25 * instance for further operations as the
26 * save operation might have changed the
27 * entity instance completely.
28 */
29 <S extends T> List<S> save(Iterable<S> var1);
30
31 /**
32 * Flushes all pending changes to
33 * the database.
34 */
35 void flush();
36
37 /**
38 * Saves an entity and flushes changes
39 * instantly.
40 */
41 <S extends T> S saveAndFlush(S var1);
42
43 /**
44 * Deletes the given entities in a
45 * batch which means it will create
46 * a single {@link Query}.
47 * Assume that we will clear the
48 * {@link javax.persistence.EntityManager}
49 * after the call.
50 */
51 void deleteInBatch(Iterable<T> var1);
52
53 /**
54 * Deletes all entities in a batch call.
55 */
56 void deleteAllInBatch();
57
58 /**
59 * Returns a reference to the entity with the
60 * given identifier.
61 */
62 T getOne(ID var1);
63
64 /**
65 * Returns all entities matching the given
66
67 * {@link Example}. In case no match could
68 * be found an empty {@link Iterable}
69 * is returned.
70 */
71 <S extends T> List<S> findAll(Example<S> var1);
72
73 /**
74 * Returns all entities matching the given
75 * {@link Example} applying the given
76 * {@link Sort}. In case no match could
77 * be found an empty {@link Iterable}
78 * is returned.
79 */
80 <S extends T> List<S> findAll(Example<S> var1,
     Sort var2);

```



► Codebeispiel: PagingAndSortingRepository ▾

```

1 //-----
2 // PageAndSortRepository Schnittstelle
3 //-----
4 public interface PagingAndSortingRepository<T, ID
5      extends Serializable> extends
6      CrudRepository<T, ID> {
7
8 /**
9 * Returns all entities sorted by the
10 * given options.
11 */
12 Iterable<T> findAll(Sort sort);
13
14 /**
15 * Returns a {@link Page} of entities
16 * meeting the paging restriction
17 * provided in the {@code Pageable} object.
18 */
19 Page<T> findAll(Pageable pageable);
20
21 }

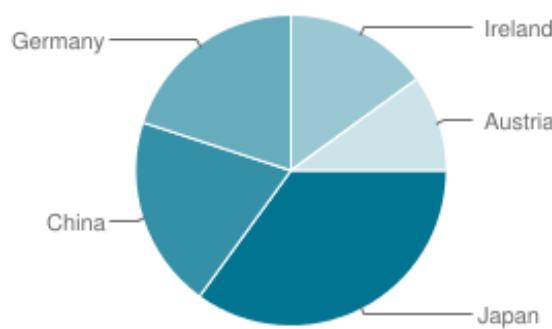
```

► Codebeispiel: CrudRepository ▾

```

1  //-----
2  // CrudRepository Schnittstelle
3  //-----
4  public interface CrudRepository<T, ID extends
5    Serializable> extends Repository<T, ID> {
6
7    /**
8     * Saves a given entity. Use the returned
9     * instance for further operations as the
10    * save operation might have changed the
11    * entity instance completely.
12   */
13  <S extends T> S save(S entity);
14
15  /**
16   * Saves all given entities.
17   */
18  <S extends T> Iterable<S> save(Iterable<S>
19    entities);
20
21  /**
22   * Retrieves an entity by its id.
23   */
24  T findOne(ID id);
25
26  /**
27   * Returns whether an entity with the given
28   * id exists.
29   */
30  boolean exists(ID id);
31
32  /**
33   * Returns all instances of the type.
34   */
35  Iterable<T> findAll();
36
37  /**
38   * Returns all instances of the type
39   * with the given IDs.
40   */
41  Iterable<T> findAll(Iterable<ID> ids);
42
43  /**
44   * Returns the number of entities
45   * available.
46   */
47  long count();
48
49  /**
50   * Deletes the entity with the given id.
51   */
52  void delete(ID id);
53
54  /**
55   * Deletes a given entity.
56   */
57  void delete(T entity);
58
59  /**
60   * Deletes the given entities.
61   */
62  void delete(Iterable<? extends T> entities);
63
64  /**
65   * Deletes all entities managed by the
66   * repository.
67   */
68  void deleteAll();
69 }

```



15.2.2 Parametrisierung der JPARepository Schnittstelle

Um die JpaRepository Schnittstelle in unsere Anwendung zu integrieren muß die Schnittstelle **parametrisiert** werden.

► Erklärung: Parametrisierung ▾

- **JpaRepository<Subproject, Long>:** MIT DER PARAMETRISIERUNG DER JPA-SCHNITTSTELLE STEHT UNS NUN EINE SCHNITTSTELLE ZUR VERARBEITUNG VON SUBPROJECT-ENTITÄTEN ZUR VERFÜGUNG.

Parametrisiert wird die **JPA Schnittstelle** mit der **Klasse** der Entität und dem **Schlüsseltyp** der Entität

- **@Autowired:** DER EINSATZ DER `@AUTOWIRED` ANNOTATION STÖSST **Preprozessoren** AN UM EINE IMPLEMENTIERUNG DER `ISUBPROJECTREPOSITORY` SCHNITTSTELLE ZUR VERFÜGUNG ZU STELLEN.



▶ Codebeispiel: Parametrisierung ▶

```
1 //--  
2 // Parametrisierung der JPARepository s.  
3 //---  
4 @Data  
5 @NoArgsConstructor  
6 @Entity  
7 @Table(name = "subprojects")  
8 @Access(AccessType.FIELD)  
9 public class Subproject implements Serializable{  
10  
11     @GeneratedValue  
12     @Id  
13     private Long id;  
14  
15     @ManyToOne  
16     @JoinColumn(name = "project_id")  
17     private AProject project;  
18  
19     @NotNull  
20     @Size(max = 100)  
21     @Column(name = "title", length = 100)  
22     private String title;  
23  
24 }  
25  
26  
27 public interface ISubprojectRepository extends  
28     JpaRepository<Subproject, Long>{  
29 }
```

15.2.3 Methoden der JPARepository Schnittstelle

Die **JPARepository** Schnittstelle stellt dem Programmierer 24 Methoden für die **Datenverarbeitung** zur Verfügung.

► Erklärung: Erweitern von JpaRepository ▼

- DIE **JPArepository Schnittstelle** KANN JEDERZEIT DURCH DAS HINZUFÜGEN VON METHODEN UM **Funktionalität** ERWEITERT WERDEN.
 - SOLANGE WIR UNS BEI DER **Signatur** HINZUGEFÜGTER METHODEN AN BESTIMMTE **Regeln** HALTEN, STELLT **Spring Data** EINE IMPLEMENTIERUNG DER METHODE FÜR UNS BEREIT.



► Codebeispiel: Signatur von Methoden ▾

- **Spring Data** IST IN DER LAGE EINE **Implementierung** DER METHODE ZU **generieren** SOLANGE WIR UNS AN BE-STIMMTE REGELN FÜR DIE **Signatur** DER METHODE HAL-TEN.
 - **Signatur:** QUERY VERB + SUBJECT + BY + PREDICAT
 - **Beispiel:**



► Erklärung: Query Verben ▾

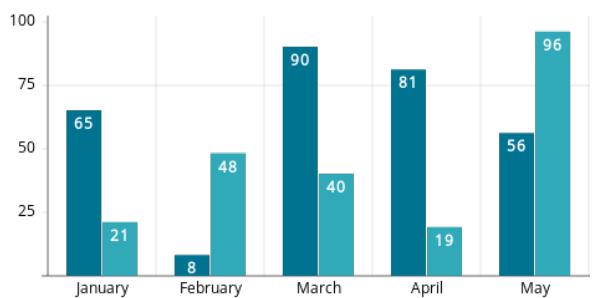
- **Query Verb:** UNS STEHEN 4 VERSCHIEDENE **Query Verben** ZUR VERFÜGUNG: `get`, `read`, `find` und `count`
- **get, read, find:** ALLE 3 **Query Verben** KÖNNEN **gleichbedeutend** VERWENDET WERDEN. MIT `GET`, `READ` AND `FIND` WERDEN ENTITÄTEN AUS DER DATENBANK GELESEN.
- **count:** DURCH DAS **query verb COUNT** WERDEN METHODEN GENERIERT DIE DIE **Anzahl** VON ENTITÄTEN ABZÄHLEN.

► Codebeispiel: Subject ▾

- DER **Subject** TEIL DES NAMENS IST OPTIONAL, DA **Spring Data** DEN TYP DER ENTITÄTEN AUS DEM **generischen Parameter** DER JPARepository SCHNITTSTELLE ERMITTelen KANN.

► Codebeispiel: Prädikat ▾

- MIT DER ANGABE EINES **Prädikats** WIRD DIE ERGEBNISMENGE DER **Query** EINGESCHRÄNKt. Das Prädikat FORMULIERT EINE **Bedingung**, DIE FÜR JEDE ZEILE DER ERGEBNISMENGE ERFÜLLT SEIN MUSS.
- IM **Prädikat** KÖNNEN DIE **Attribute** DER **Entität** BEIEBIG MIT OR ODER AND KOMBINIERT WERDEN.
- DIE **Parameter** DER METHODE MÜSSEN NICHT DENSELBNEN NAMEN HABEN WIE DIE **Attribute** DER ENTITÄT HABEN, DIE **Reihenfolge** IN DER SIE AUFTREten MUSS ABER MIT DER REIHENFLOGE DER PARAMETER IN DER SIGNATUR DER METHODE ÜBEREINSTIMMEN.
- DAS ERGEBNIS DER ABFRAGE KANN DURCH DIE ANGABE DER **OderBy** KLAUSEL SORTIERT WERDEN.



► Erklärung: Vergleichsoperatoren ▾

■ DIE **Parameter** DER METHODE UND DIE **Attribute** DER ENTITÄT KÖNNEN AUF UNTERSCHIEDLICHE ART MITEINANDER **vergleichen** WERDEN.

■ Unterstütze Vergleichsoperatoren:

- ▶ `IsAfter`, `After`, `IsGreaterThan`, `GreaterThan`
- ▶ `IsGreaterThanOrEqualTo`, `GreaterThanOrEqualTo`
- ▶ `IsBefore`, `Before`, `IsLessThan`, `LessThan`
- ▶ `IsBetween`, `Between`
- ▶ `IsNull`, `Null`
- ▶ `IsNotNull`, `NotNull`
- ▶ `IsIn`, `In`
- ▶ `IsNotIn`, `NotIn`
- ▶ `IsStartingWith`, `StartingWith`, `StartsWith`
- ▶ `IsEndingWith`, `EndingWith`, `EndsWith`
- ▶ `IsContaining`, `Containing`, `Contains`
- ▶ `IsLike`, `Like`, `Is`, `Equals`
- ▶ `IsNotLike`, `NotLike`, `IsNot`, `Not`
- ▶ `IsTrue`, `True`



► Codebeispiel: Vergleichsoperatoren ▾

```

1 //-----
2 // Vergleichsoperatoren
3 //-----
4 public interface IProjectRepository extends
5     JpaRepository<Project, Long>{
6
7     List<AProject> findProjectsByCreationDate(
8         Date start);
9
10    //enabling distinct flag for query
11    List<AProject> findDistinctProjectsByTitle(
12        String title);
13
14    //ignoring case for an individual property
15    List<AProjects>
16        findDistinctProjectsByTitleIgnoreCase(
17            String title);
18 }
```



15.2.4 Schnittstellenbeispiel

► Codebeispiel: Schnittstellenimplementierungen ▾

```

1 //-----
2 // Modelklassen - Entitäten
3 //-----
4 @Inheritance(strategy="JOINED")
5 @Entity
6 @Table(name="PROJECTS")
7 @Data
8 public abstract class AProject implements
9     Serializable{
10
11     @OneToMany(mappedBy="project")
12     private List<Subproject> subprojects = new
13         ArrayList<>();
14
15     @GeneratedValue
16     @Id
17     private Long id;
18
19     @NotNull
20     @Column(name="TITLE", length=100,
21             nullable=false)
22     private String titel;
23
24     @NotNull
25     @Column(name="DESCRIPTION", length=4000,
26             nullable=false)
27     private String description;
28
29 }
30
31 //-----
32 // Entität: RequestFundingProject
33 //-----
34 @Entity
35 @Table(name="REQUEST_PROJECTS")
36 @Data
37 public class RequestFundingProject extends
38     AProject{
39
40     @Column(name="IS_SMALL_PROJECT", length=1)
41     private Boolean isSmallProject;
42
43 }
44
45 //-----
46 // Entität: ResearchFundingProject
47 //-----
48 @Entity
49 @Table(name="RESEARCH_PROJECTS")
50 @Data
51 public class ResearchFundingProject extends
52     AProject{
53
54     @Column(name="IS_FFG_SPONSORED", length=1)
55     private Boolean isFFGSponsored;
56
57 }
58
59 //-----
60 // Entitäten - Subproject
61 //-----
62 @Entity
63 @Table(name="SUBPROJECTS")
64 @Data
65 public class Subproject implements Serializable{
66
67     @ManyToOne
68     @JoinColumn(name="PROJECT_ID")
69     private AProject project;
70
71     @ManyToOne
72     @JoinColumn(name="INSTITUT_ID")
73     private Institut institut;
74
75     @GeneratedValue
76     @Id
77     private Long id;
78
79     @Column(name="DESCRIPTION")
80     private String description;
81
82     @Min(0)
83     @Max(100)
84     @Column(name="APPLIED_RESEARCH")
85     private Integer appliedResearch;
86
87     @Min(0)
88     @Max(100)
89     @Column(name="FOCUSED_RESEARCH")
90     private Integer focusedResearch;
91
92 }
93
94 //-----
95 // Entität: Institute
96 //-----
97 @Entity
98 @Table(name="INSTITUTES")
99 @Data
100 public class Institute implements Serializable{
101
102     @GeneratedValue
103     @Id
104     private Long id;
105
106     @Column(name="DESCRIPTION")
107     private String description;
108
109     @Column(name="NAME")
110     private String name;
111
112 }
113
114
115
116

```

```

117 //-----  

118 // Entitaet: Partner  

119 //-----  

120 @Entity  

121 @Table(name="PARTNER")  

122 @Data  

123 public class Partner implements Serializable{  

124  

125     @GeneratedValue  

126     @Id  

127     private Long id;  

128  

129     @NotNull  

130     @Column(name="name", nullable=false)  

131     private String name;  

132  

133     @Column(name="DESCRIPTION")  

134     private String description;  

135  

136 }  

137  

138 @Embeddable  

139 @NoArgsConstructor  

140 @AllArgsConstructor  

141 public class FundingID implements Serializable{  

142  

143     @NotNull  

144     @ManyToOne  

145     @JoinColumn(name="PROJECT_ID", nullable=false)  

146     private AProject project;  

147  

148     @NotNull  

149     @ManyToOne  

150     @JoinColumn(name="PARTNER_ID", nullable=false)  

151     private Partner partner;  

152  

153 }  

154 //-----  

155 // Entitaet: Funding  

156 //-----  

157 @Entity  

158 @Table(name="FUNDINGS")  

159 @Data  

160 public class Funding implements Serializable{  

161  

163     @EmbeddedId  

164     private FundingID fundingID;  

165  

166     @NotNull  

167     @Column(name="AMOUNT", nullable=false)  

168     private Long amount;  

169  

170 }  

171  

172  

173  

174  

175  

176  

177  

178  

179
180 //-----  

181 // Schnittstelle: IProjectRepository  

182 //-----  

183 public interface IProjectRepository extends  

184     JpaRepository<AProject, Long>{  

185  

186     /**  

187      * Findet alle Projekte, die einen bestimmten  

188      * Titel haben.  

189      * Das Ergebnis entspricht einem Full Table  

190      * Scan.  

191      */  

191     List<AProject> getProjectsByTitle(String title);  

192  

193     /**  

194      * Findet alle Projekte, die einen bestimmten  

195      * Titel haben.  

196      * Das Ergebnis ist in Pages unter-  

197      * teilt.  

198      */  

199     Page<AProject> getProjectsByTitle(String title,  

200                                         Pageable page);  

201  

202     /**  

203      * Findet alle Projekte, deren Titel einen  

204      * bestimmten Token enthalten.  

205      * Das Ergebnis ist in Pages unter-  

206      * teilt.  

207      */  

207     Page<AProject> getProjectsByTitleIsLike(String  

208                                                 titleToken, Pageable page);  

209  

210     /**  

211      * Findet alle Projekte, deren Titel einen  

212      * bestimmten Token enthalten.  

213      * Das Ergebnis ist in Pages unter-  

214      * teilt.  

215      */  

215     Page<AProject>  

216         getProjectsByTitleContains(String  

217                                     titleToken, Pageable page);  

218  

219     /**  

220      * Findet alle Projekte, deren Titel einen  

221      * bestimmten Token enthalten.  

222      * Das Ergebnis ist in Pages unter-  

223      * teilt und wird sortiert nach den Werten  

224      * in der name Spalte.  

225      */  

224     Page<AProject>  

226         getProjectByTitleIsLikeOrderByAsc(String  

227                                         titleToken, Pageable page);  

228  

229     /**  

230      * Findet alle Projekte, deren Titel und  

231      * Beschreibungen  

232      * einen bestimmten Token enthalten.  

233      * Das Ergebnis ist in Pages unterteilt.  

234      */  

234     Page<AProject>  

235         getProjectsByTitleContainsAndDescriptionContains(String  

236                                         titleToken, String descriptionToken,  

237                                         Pageable page);

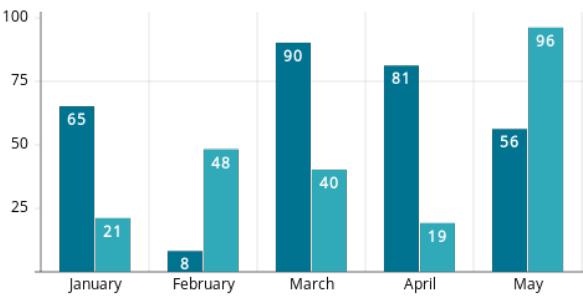
```

```

233                                         287
234     /**
235      * Findet alle Projekte, deren Titel ein
236      * bestimmten Token enthalten.
237      * Die Auswertung erfolgt caseinsensitive.
238      * Das Ergebnis ist in Pages unterteilt.
239     */
240     Page<AProject>                               291
241         getProjectsByTitleContainsIgnoreCase(String 293
242             titleToken, Pageable page);               294
243
244     /**
245      * Findet alle Projekte, deren Titel mit einem
246      * bestimmten Token beginnen.
247      * Das Ergebnis ist in Pages unterteilt.
248     */
249     Page<AProject>                               295
250         getProjectsByTitleStartingWith(String 299
251             titleToken, Pageable page);               300
252
253     /**
254      * Findet alle Projekte, deren Titel mit einem
255      * bestimmten Token enden.
256      * Das Ergebnis ist in Pages unterteilt.
257     */
258     Page<AProject>                               301
259         getProjectsByTitleEndsWith(String 305
260             titleToken, Pageable page);               306
261
262     /**
263      * Findet alle Projekte, deren Titel in einer
264      * Liste von Werten enthalten sind.
265      * Das Ergebnis ist in Pages unterteilt.
266    }
267
268 //-----// Test: IProjectRepository
269 //-----//
270 @Transactional
271 @RunWith(SpringRunner.class)
272 @SpringBootTest
273 @ActiveProfiles("test")
274 public class ProjectApplicationTests {
275
276     private static Logger log =
277         LoggerFactory.getLogger
278             (ProjectApplicationTests.class);
279
280     @Autowired
281     private IProjectRepository projectRepository;
282
283     @Test
284     public void testProjectRepository() {
285         Project project = projectRepository.
286             getProjectByTitle("Finite Element");
287
288         assertNotNull(project);
289
290     }
291
292     Page<Project> projects = projectRepository.
293         getProjectsByTitleIsLike("%sim%", 294
294             new PageRequest(0, 10));
295
296     assertEquals(2,
297         projects.getNumberOfElements());
298
299     projects = projectRepository.
300         getProjectsByTitleContains("sim", 295
301             new PageRequest(0, 10));
302
303     assertEquals(2,
304         projects.getNumberOfElements());
305
306     projects = projectRepository.
307         getProjectsByTitleIsLikeOrderByNameAsc("%sim%", 296
308             new PageRequest(0, 10));
309
310     assertEquals(2,
311         projects.getNumberOfElements());
312
313     projects = projectRepository.
314         getProjectsByTitleContainsAnd
315             DescriptionContaining("sim",
316                 "link", new PageRequest(0,
317                     10));
318
319     assertEquals(1,
320         projects.getNumberOfElements());
321
322     projects = projectRepository.
323         getProjectsByTitleContainingIgnoreCase("SIM", 307
324             new PageRequest(0, 10));
325
326     assertEquals(2,
327         projects.getNumberOfElements());
328
329     projects = projectRepository.
330         getProjectsByTitleIn(Arrays.asList("engine", 308
331             "simulation", "simulink", "hyper",
332                 "computation"), new PageRequest(0,10));
333
334     assertEquals(3,
335         projects.getNumberOfElements());
336
337 }


```

Month	Number of Projects
January	65
February	48
March	90
April	81
May	56



15.2.5 JpaRepository erweitern

Es gibt 3 Möglichkeiten die JpaRepository Schnittstelle zu erweitern.

► Auflistung: Erweitern des JpaRepositories ▾

- Signatur von Methoden ▾**

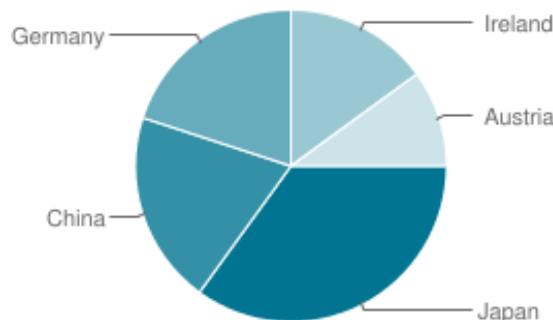
DURCH DAS EINHALTEN BESTIMMTER **Regeln** FÜR DIE **Signatur** VON METHODEN, generiert SPRING, DEN FÜR DIE LOGIK DER METHODE, NOTWENDIGEN Code.

- Übergabe von Queries ▾**

BEI DER **Programmierung** DES JPAREPOSITORIES KANN GEMEINSAM MIT EINER METHODE AUCH EINE **Query** ANGEgeben WERDEN. Die **Query** WIRD ZUR ERMITTlung DES ERGEBNISSES DER **Abfrage** HERANGEZOGEN.

- QueryDSL ▾**

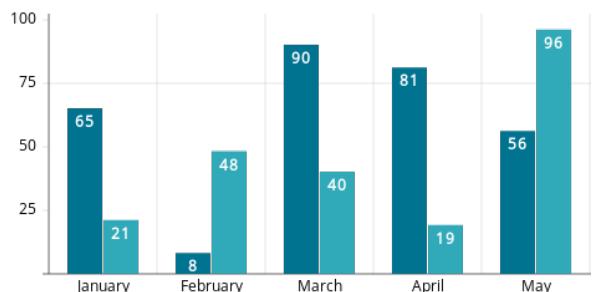
Queries KÖNNEN AUCH IN FORM VON **Objekten** AN METHODEN ÜBERGEBEN WERDEN. SPRING STELLT DAZU DAS **QueryDSL** FRAMEWORK ZUR VERFÜGUNG.



15.2.6 JPARepository - Übergabe von Queries

- ### ► Erklärung: Queries definieren ▾
- DIE **@QUERY** ANNOTATION WIRD VERWENDET UM EINE **Query** AN DIE **JpaRepository** SCHNITTSTELLE ZU ÜBERGEBEN.
 - BEIM AUFRUF DER METHODE, WIRD DIE QUERY VON DER **Datenbankengine** AUSGEFÜHRT UND DAS ERGEBNIS AN DIE ANWENDUNG ZURÜCKGEGEBEN.

- DIE **@PARAM** ANNOTATION VERBINDET DIE **Parameter** DER METHODE, MIT DEN PARAMETERN DER **Query**.
- FÜR DAS DEFINIEREN DER QUERY WIRD EINE EIGENE QUERSPRACHE VERWENDET: **JPA SQL**.



► Codebeispiel: Übergabe von Queries ▾

```

1  public interface ISubprojectRepository extends
2      JpaRepository<Subproject, Long>{
3
4      @Query("select s from Subproject s where
5          s.project_id = :projectId")
6      List<Subproject> findSubprojectsByProject(
7          @Param("projectId") Long projectId);
8
9      @Query("select s from Subproject s where
10         s.project_id = :projectId")
11     List<Subproject> findSubprojectsByProject(
12         @Param("projectId") Long projectId);
13
14      @Modifying
15      @Query("update AProject p set u.level = :level
16             where p.title = :title")
17      int updateProjectLevelByTitle(
18          @Param("titel") String title,
19          @Param("level") EProjectLevel level);
20
21      @Modifying
22      @Query("delete from AProject p where p.level =
23             :level")
24      void deleteBulkByProjectLevel(
25          @Param("level") EProjectLevel level);
26  }

```

15.3. JPA Query Language ▾



jpa sql ▾

JPA SQL IST EINE objektorientierte Abfrage- sprache FÜR relationale Datenbanken, ANGE- LEHNT AM SQL Standard.

► Erklärung: Motivation - JPA SQL ▾

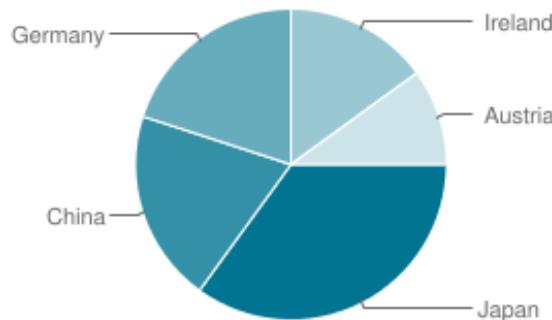
- SQL IST EINE standadisierte ABFRAGESPRACHE FÜR relationale Datenbanken.
- DIVERSE Datenbankhersteller HABEN DEN SQL Standard MIT DER ZEIT verändert UND WEITERENTWICKELT.
- ES GIBT KEINE GARANTIE DASS EINE SQL Abfrage, DIE FÜR EINE ORACLE DATENBANK GE SCHRIEBEN WURDE AUCH AUF EINER MySQL DATENBANK ausgeführt WERDEN KANN.
- JPA MUSS JEDOCH MIT DEN DATENBANKEN ALLER Datenbankhersteller FUNKTIONIEREN
- JPA SQL IST EIN Datenbankhersteller UN- ABHÄNGIGER STANDARD. JPA GENERIERT AUS JPA Queries FÜR JEDE DATENBANK, DIE ENTSPRECHENDEN QUERIES.



15.3.1 JPA SQL Grundlagen ▾

► Erklärung: JPA SQL ▾

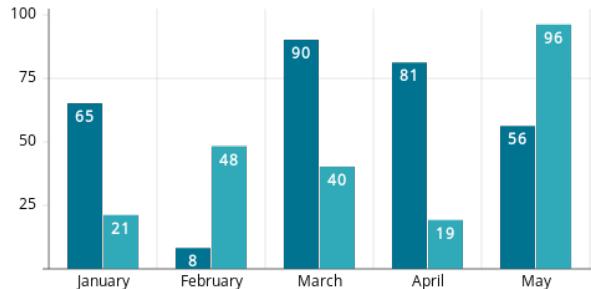
- JPA SQL ERINNERT IN SEINEN Grundzügen UND SEINER Struktur STARK AN SQL.
- IM GEGENSATZ ZU SQL, ARBEITET JPA SQL JEDOCH NICHT AUF DEN TABELLEN DER DATENBANK SONDERN AUF DEN Entitäten DER Modelschicht.



15.3.2 Pfadausdrücke ▾

Arbeiten wir mit JPA SQL, wird die Struktur der JPA Queries von der Struktur der Entitäten der Modelschicht vorgegeben.

In JPA SQL Queries arbeiten wir mit Objekten und Objektrelationen.



Pfadausdrücke ▾

Pfadausdrücke SIND FUNDAMENTALE TEILE DER JPA SQL SPRACHSPEZIFIKATION. SIE WERDEN VER- WENDET UM ENTLANG, DER IN Entities DEFINIERTEN Relationen, ZU NAVIGIEREN.

Pfadausdrücke KÖNNEN IN JEDER Klausel DER JPA Query VERWENDET WERDEN.



► Erklärung: Typen von Pfadausdrücken ▾

- Skalare Pfadausdrücke: Skalare Pfadausdrücke SIND PFADAUSDRÜCKE DIE ENTLANG VON @MANYToONE ODER @ONETOONE RELATIONEN DEFINIERT SIND.

```
select p.title from Funding f join
  f.fundingID.project p;
```

- Finite Pfadausdrücke: Finite Pfadausdrücke SIND PFADAUSDRÜCKE DIE ENTLANG VON @ONETO MANY ODER @MANYToMANY RELATION DEFINIERT WERDEN. Ein Pfad- ausdruck KANN NICHT ÜBER EINEN Finiten Pfadaus- druck HINAUSGEHEN.

```
select s from AProject p join p.subprojects s;
```



15.3.3 Select Klausel

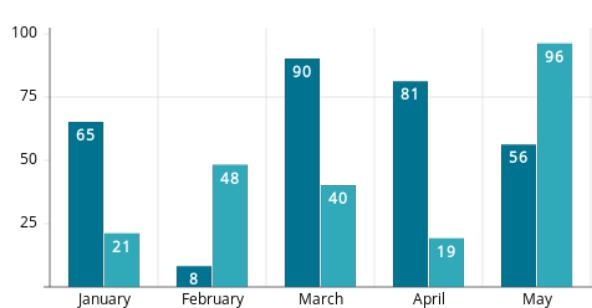
Die **Select Klausel** projiziert **Daten** der Datenbasis der **JPA SQL Query** auf die **Objekte** des Ergebnisses.

Die **Select Klausel** ist die letzte Klausel die im Rahmen der **Query** ausgewertet wird.



► Codebeispiel: Beispielqueries ▾

```
select p from RequestFundingProject p;
select p.titel from AProject p;
select p from AProject p;
select p from AProject p where type(p) =
    RequestFundingProject;
select s.institut from Subproject s;
```



15.3.4 From Klausel

In der **From Klausel** wird die **Datenbasis** der **JPA Query** definiert. Der Einsatz von **Joins** in der From Klausel, erlaubt uns Informationen über mehrere Entitäten hinweg zu **aggregieren**.

Joins werden entlang von **Pfadausdrücken** definiert.

► Codebeispiel: Pfadausdrücke in Joins ▾

```
select s from AProject p join p.subprojects s;
select p.title from Funding f join f.fundingID.
    project p;
select i from AProject a join a.subprojects s join
    s.institute i;
```

15.3.5 Schnittstellenbeispiel

► Codebeispiel: Vergleichsoperatoren ▾

```
1 //-----
2 // IProjectRepository
3 //-----
4 public interface IProjectRepository extends
    JpaRepository<AProject, Long>{
5
6     /**
7      * Fuer ein bestimmtes Projekt soll
8      * ermittelt werden wie hoch seine
9      * Foerderung ist
10     */
11    @Query("select sum(f.amount) from Funding f
12         join f.fundingID.project p where p =
13             :project group by p")
14    Integer getProjectFundingByProject
15        (@Param("project") AProject project);
16
17    /**
18     * Es sollen alle Projekte ermittelt
19     * werden deren Foerderung einen bestimmten
20     * Betrag uebersteigt
21     */
22    @Query("select p from Funding f join
23         f.fundingID.project p group by p having
24         sum(f.amount) > amount")
25    Integer getProjectFundingByProject
26        (@Param("amount") Integer amount);
27
28    /**
29     * Es sollen alle Projekte ermittelt
30     * werden die an einem bestimmten
31     * Institut durchgefuehrt werden
32     */
33    @Query("select distinct s.project from
34         Subproject s where s.institute =
35             :institute ")
36    List<AProject> getProjectsByInstitute
37        (@Param("institute") Institute institute);
38
39    /**
40     * Es sollen alle Projekte ermittelt
41     * werden die an einem bestimmten
42     * Institut durchgefuehrt werden
43     */
44    @Query("select distinct s.project from
45         Subproject s where s.institute =
46             :institute ")
47    List<AProject> getProjectsByInstitute
48        (@Param("institute") Institute institute);
49 }
```

15.4. Fetchgraphen

15.4.1 Objektgraphen



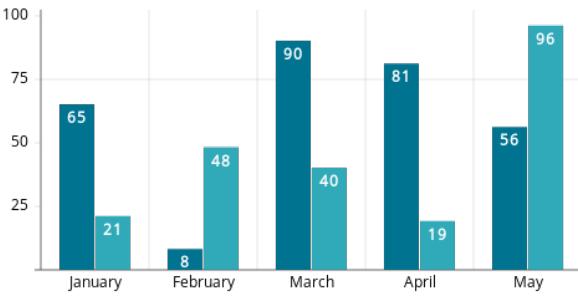
Objektgraph ▾

ALS **Objektgraphen** BEZEICHNET MAN DEN **Graphen** VON **OBJEKten** DER DURCH **Objektkomposition** ENTSTEHT. DIE **Wurzel** DES **OBJEKTGRAPHEN** IST STETS DAS **OBJEKT** VON DEM MAN AUSGEHT.

► Codebeispiel: Objektgraph AProject ▾

```

1 //-----
2 // Klasse: AProject
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Inheritance(strategy = InheritanceType.JOINED)
8 @Entity
9 @Table(name="projects")
10 @Access(AccessType.FIELD)
11 public abstract class AProject extends AEntity{
12
13     @OneToMany
14     @JoinColumn(name="PROJECT_ID")
15     private List<Subproject> subprojects = new
16     ArrayList<>();
17
18     @Column(name="title", length = 100, nullable =
19             false)
20     private String title;
21
22     @Column(name="description", length = 4000)
23     private String description;
24
25     @Temporal(TemporalType.DATE, , nullable =
26             false)
27     @Column(name = "creationDate")
28     private Date creationDate;
29
30 //-----
31 // Klasse: Subproject
32 //-----
33 @NoArgsConstructor
34 @Getter
35 @Setter
36 @Entity
37 @Table(name = "subprojects")
38 @Access(AccessType.FIELD)
39 public class Subproject extends AEntity{
40
41     @ManyToOne
42     @JoinColumn(name="INSTITUTE_ID")
43     private Institute institute;
44
45     @Min(0)
46     @Max(100)
47     @Column(name="FOCUS_RESEARCH", nullable=false)
48     private Integer focusResearch;
49
50     @NotNull
51     @Min(0)
52     @Max(100)
53     @Column(name="APPLIED_RESEARCH",
54             nullable=false)
55     private Integer appliedResearch;
56 }
57 //-----
58 //----- Klasse: Institute -----
59 //----- Klasse: Institute -----
60 @NoArgsConstructor
61 @Getter
62 @Setter
63 @Entity
64 @Table(name = "INSTITUTES")
65 @Access(AccessType.FIELD)
66 public class Institute extends AEntity{
67
68     @NotNull
69     @Size(min= 12, max=12)
70     @Column(name="INSTITUTE_CODE", length=12 ,
71             nullable=false, unique=true)
72     private String instituteCode;
73
74     @NotNull
75     @Size(min= 12, max=12)
76     @Column(name="INSTITUTE_NAME", length=12 ,
77             nullable=false, unique=true)
78     private String instituteName;
79 }
```



► Analyse: Project Objektgraph ▾

- PROJECT OBJEKTE SIND **Objektgraphen** BESTEHEND AUS MEHREREN **SUBPROJECT** OBJEKTKOMPOSITIONEN.
- SUBPROJECT OBJEKTE WIEDERUM SPEICHERN EINE REFERENZ AUF EIN INSTITUT OBJEKT.



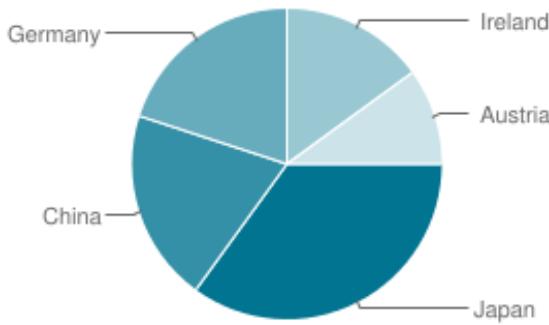
15.4.2 Laden von Objektgraphen

Wird eine **Entität** aus der Datenbank geladen stellt sich die Frage welche Teile des **Objektgraphen** mitgeladen werden sollen.



► Analyse: Laden von Objektgraphen ▾

- JPA BIETET DIE MÖGLICHKEIT, IM **Mapping** ANZUGEBEN, OB EINE BEZIEHUNG GLEICH MITGELADEN WERDEN SOLL, WENN DIE EIGENTLICHE ENTITÄT GELADEN WIRD.
- DIES IST MÖGLICH, INDEM MAN IM MAPPING FÜR DEN **FetchType EAGER** ANGIBT.
- DIESER IST BEI TO-ONE-BEZIEHUNGEN (ALSO **@ManyToOne** UND **@OneToOne**) DER DEFAULT.
- BEI **To-many** BEZIEHUNGEN (ALSO **@OneToMany** UND **@ManyToMany**) DAGEGEN MUSS ER EXPLIZIT ANGEgeben WERDEN.



► Codebeispiel: Defaultwerte - **fetchType** ▾

```

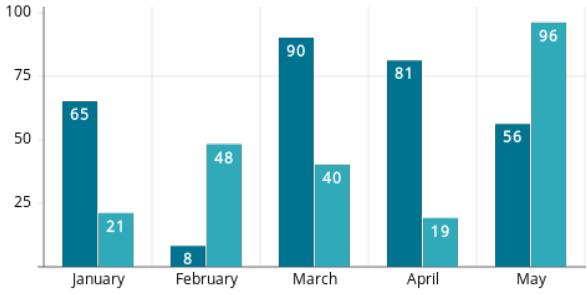
1 //-----
2 // Default: @OneToMany(fetch = FetchType.LAZY)
3 //-----
4 @NoArgsConstructor
5 @Getter
6 @Setter
7 @Inheritance(strategy = InheritanceType.JOINED)
8 @Entity
9 @Table(name="projects")
10 @Access(AccessType.FIELD)
11 public abstract class AProject extends AEntity{
12
13     @OneToMany(fetch = FetchType.LAZY)
14     @JoinColumn(name="PROJECT_ID")
15     private List<Subproject> subprojects = new
16         ArrayList<>();
17
18     @Column(name="title", length = 100, nullable =
19             false)
20     private String title;
21
22 }
```

```

19
20     @Column(name="description", length = 4000)
21     private String description;
22
23 }
24
25 //-----
26 // Klasse: Subproject
27 //-----
28 @NoArgsConstructor
29 @Getter
30 @Setter
31 @Entity
32 @Table(name = "subprojects")
33 @Access(AccessType.FIELD)
34 public class Subproject extends AEntity{
35
36     @ManyToOne(fetch = FetchType.EAGER)
37     @JoinColumn(name="INSTITUTE_ID")
38     private Institute institute;
39
40     @NotNull
41     @Min(0)
42     @Max(100)
43     @Column(name="APPLIED_RESEARCH",
44             nullable=false)
45     private Integer appliedResearch;
46 }
```

► Analyse: **FetchType bearbeiten** ▾

- ES BESTEHT DIE MÖGLICHKEIT DEN **fetchType** IN EINER ENTITÄTEN ZU VERÄNDERN.
- DAS BEDEUTET JEDOCH DASS EIN SO DEFINIERTES VERHALTEN FÜR ALLE ENTITÄTEN DER KLASSE ÜBERNOMMEN WIRD.
- DER **fetchType** SOLLTE IN DER REGEL NICHT IN DER KLASSE VERÄNDERT WERDEN.
- JPA VERWENDET **Entitygraphen** UM DAS **Ladeverhalten** AUF METHODENEBENE ZU STEUERN.



15.4.3 Entitygraphen

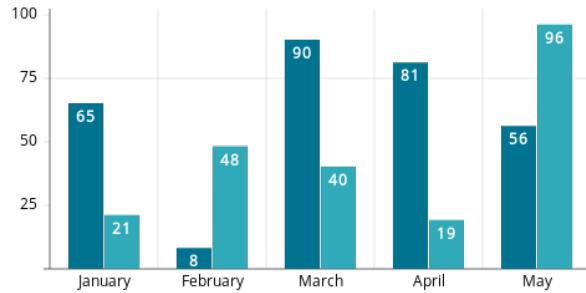
 Entitygraph ▾

MIT EINEM Entitygraph WIRD DAS Ladeverhalten DER Objektreferenzen IN Entitäten DEFINIERT.



► Analyse: Entitygraph ▾

- MIT EINEM Entitygraphen WIRD DEFINIERT WELCHE TEILE DES Objektgraphen GELADEN WERDEN SOLLEN.
- GELADEN WERDEN JENE Objektreferenzen DIE IM Entitygraphen ANGEgeben WERDEN.



► Codebeispiel: Entitygraph ▾

```

1 //-----
2 // Klasse: AProject
3 //-----
4 @NamedEntityGraph(
5     name = "projectWithSubprojects",
6     attributeNodes =
7         {@NamedAttributeNode("subprojects")}
8 )
9 @Inheritance(strategy = InheritanceType.JOINED)
10 @Entity
11 @Table(name = "projects")
12 public abstract class AProject extends AEntity{
13
14     @OneToMany
15     @JoinColumn(name = "PROJECT_ID")
16     private List<Subproject> subprojects = new
17         ArrayList<>();
18
19     @Column(name = "title", length = 100, nullable =
20             false)
21     private String title;
22
23     @Column(name = "description")
24     private String description;
25 }
```

```

24 //-----
25 // Klasse: IProjectRepository
26 //-----
27 public interface IProjectRepository extends
28     JpaRepository<AProject, Long>{
29
30     @EntityGraph(value = "projectWithSubprojects")
31     AProject findOne(Long id);
32 }
```

► Analyse: Subgraphen ▾

- IN DER REGEL IST ES NOTWENDIG EBENFALLS FÜR Subgraphen DES OBJEKTGRAPHEN EIN Fetchverhalten ZU DEFINIEREN.

► Codebeispiel: Subgraphen ▾

```

1 //-----
2 // Definieren von Subgraphen
3 //-----
4 @NamedEntityGraph(
5     name = "projectWithSubprojects",
6     attributeNodes = {
7         @NamedAttributeNode(
8             value = "subprojects",
9             subgraph = "subprojectInstitute")
10 },
11 subgraphs = {
12     @NamedSubgraph(
13         name = "subprojectInstitute",
14         attributeNodes = {
15             @NamedAttributeNode("institutes")
16         }
17     )
18 }
19 )
20 @Entity
21 @Table(name = "projects")
22 public abstract class AProject extends AEntity{
23
24     @OneToMany
25     @JoinColumn(name = "PROJECT_ID")
26     private List<Subproject> subprojects = new
27         ArrayList<>();
28
29     @OneToMany
30     @JoinColumn(name = "PROJECT_ID")
31     private List<Partner> partners = new
32         ArrayList<>();
33
34     @Column(name = "title", length = 100)
35     private String title;
36
37     @Column(name = "description")
38     private String description;
39 }
```



15.5. Kaskadierende Operationen ▾

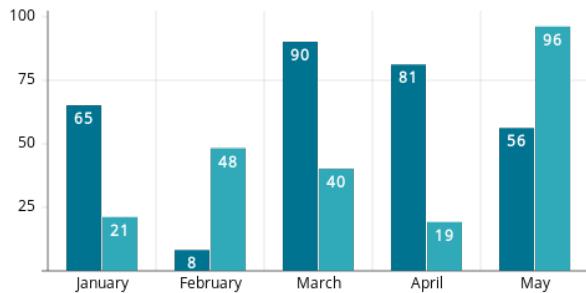
15.5.1 Datenkonsistenz



Datenkonsistenz ▾

UNTER **Datenintegrität** VERSTEHT MAN DIE WIDERSPRUCHSFREIHEIT VON DATENBESTÄTNDEN. EINE DATENBANK IST **konsistent**, WENN DIE DATEN FEHLERFREI ERFASST SIND UND DEN GEWÜNSCHEN INFORMATIONSGEHALT KORREKT WIEDERGEBEN.

Die **Datenintegrität** ist dagegen verletzt, wenn Mehrdeutigkeiten oder widersprüchliche Sachverhalte zutage treten.



Wir unterscheiden drei Arten von **Operationen**, die die **Integrität** von Daten gefährden können.

- Ändern der Attributwerte eines Tupels
- Einfügen von Tupeln
- Löschen von Tupeln



15.5.2 Kaskadierende Operationen



Kaskadierende Operationen ▾

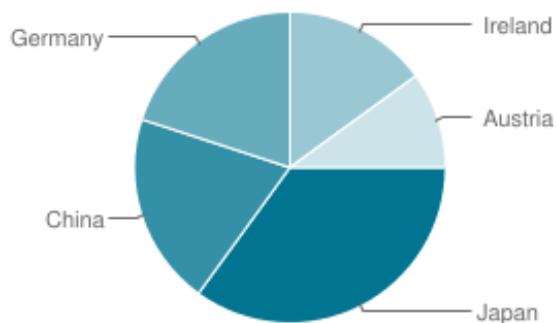
Kaskadierende Operationen ERLAUBEN ES FÜR OPERATIONEN **Folgeoperationen** ZU DEFINIEREN UM DIE **Datenintegrität** ZU SICHERN.

```

1  @Inheritance(strategy =
2      InheritanceStrategy.JOINED_TABLE)
3
4  @Getter
5  @Setter
6  @Entity
7  @Table(name = "PROJECTS")
8  @NoArgsConstructor
9  @Data
10 public abstract class AProject extends AEntity{
11
12     @OneToMany
13     @JoinColumn(name = "PROJECT_ID", cascade =
14         CascadeType.ALL, orphanRemoval =
15         true)
16     private List<Subproject> subprojects = new
17         ArrayList<>();
18
19     @Column(name = "NAME")
20     private String name;
21
22     @Column(name = "DESCRIPTION")
23     private String description;
24 }
```

► Analyse: Kaskadierende Operationen ▾

- **cascade:** DAS **cascade** ATTRIBUT ERLAUBT ES FÜR EINE INDIVIDUELLE OPERATION **Folgeoperationen** ZU DEFINIEREN.
- **CascadeType:** MIT DEM **CascadeType** WIRD DIE **Nachfolgeoperation** DEFINIERT.
- **orphanRemoval:** DAS **ORPHANREMOVAL** ATTRIBUT LEGT FEST DASS MIT DEM LÖSCHEN EINES DATENSATZES AUCH **Fremdschlüsselrelationen** AUFGELÖST WERDEN.



► Codebeispiel: Kaskadierende Operationen ▾

15.6. Transaktionsverwaltung ▾

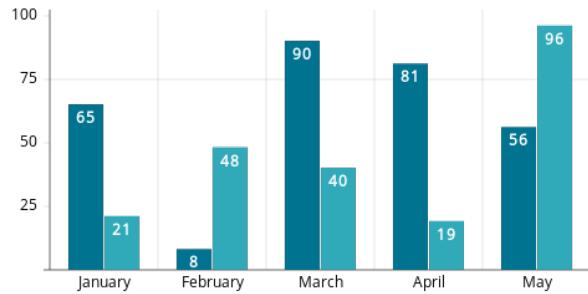
15.6.1 Transaktionen - Grundlagen



Transaktion ▾

ALS **Transaktion** BEZEICHNET MAN EINE FOLGE VON **Programmschritten**, DIE ALS EINE LOGISCHE EINHEIT BETRACHTET WERDEN, WEIL SIE DEN **Datenbestand** NACH FEHLERFREIER UND VOLLSTÄNDIGER AUSFÜHRUNG IN EINEM **konsistenten Zustand** HINTERLASSEN.

Daher wird für eine **Transaktion** insbesondere gefordert, dass sie entweder im Ganzen und **fehlerfrei** oder gar nicht **ausgeführt** wird.



15.6.2 Problemkontext

Wenn eine **Datenbank** von mehreren **Benutzern** oder Programmen verwendet wird, so ist es potenziell möglich, dass Datensätze konkurrierend **verändert** werden.

► Analyse: Konkurrenz Zugriff ▾

- IN DER REGEL MÖCHTE MAN DERARTIGE SITUATIONEN VERMEIDEN, INSBESONDERE MÖCHTE MAN VERHINDERN, DASS EIN **Benutzer** SEINE **Änderungen** VERLIERT, WEIL EIN ANDERER, DIESE UNBEMERKT **überschreibt**.
- **Transaktionen** DIENEN DAZU, EINE FOLGE VON **Aktionen** ALS GESAMTHEIT ZU BEARBEITEN.
- NUR WENN ALLE **Einzelaktionen** ERFOLGREICH **abgeschlossen** WERDEN KÖNNEN, WERDEN SIE AUSGEFÜHRT.
- BEI EINEM FEHLER MÜSSEN DIE **Änderungen** VOLLSTÄNDIG WIEDER RÜCKGÄNGIG GEMACHT WERDEN.

15.6.3 ACID Prinzip

Damit im Kontext des konkurrierenden **Zugriffs** keine Fehler auftreten werden für **Transaktionen** die **ACID Eigenschaften** gefordert.



► Auflistung: ACID Eigenschaften ▾



Atomicity ▾

EINE **Transaktion** WIRD ENTWEDER **ganz** ODER GAR NICHT AUSGEFÜHRT. TRANSAKTIONEN SIND ALSO NICHT **teilbar**. WENN EINE ATOMARE TRANSAKTION ABGE BROCHEN WIRD, IST DAS SYSTEM



Konsistenz ▾

NACH AUSFÜHRUNG DER **Transaktion** MUSS DER **Datenbestand** IN EINER **konsistenten Form** SEIN, WENN ER ES BEREITS ZU BEGINN DER TRANSAKTION WAR.



Isolation ▾

BEI GLEICHZEITIGER AUSFÜHRUNG MEHRER **Transaktionen** DÜRFEN SICH DIESSE NICHT GEGESEITIG beeinflussen.



Durability ▾

Die AUSWIRKUNGEN EINER **Transaktion** MÜSSEN IM **Datenbestand dauerhaft** BESTEHEN BLEIBEN. Die Effekte von Transaktionen dürfen also nicht verloren gehen oder mit der Zeit verblas sen.

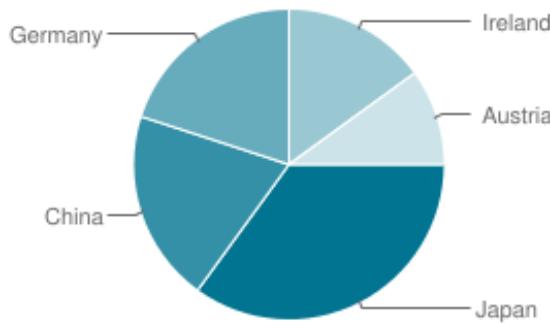
► Analyse: ACID Eigenschaften ▾

- DIE **ACID EIGENSCHAFTEN** GARANTIEREN IM KONTEXT DES konkurrierenden **Zugriffs** DIE **Konsistenz** DER DATEN.
- ERKAUFT WIRD DIESSE **Datensicherheit** JEDOCH DAMIT DASS TEILE DER DATENBANK FÜR ANDERE ZUGRIFFE GE SPERRT WERDEN.
- DIESES VERHALTEN FÜHRT IN DER REGEL JEDOCH ZU LANGEN **Antwortzeiten** DER ANWENDUNG.

15.6.4 Isolationslevel

Die **ACID** Eigenschaften garantieren uns die **Konsistenz** unserer Daten unterbinden jedoch in der Regel einen **konkurrierenden** Zugriff auf die Daten.

Solches Verhalten ist jedoch aus Performance Gründen nicht erwünscht, sodass **Datenbanken** in der Regel mehrere Varianten der **Isolation** kennen. Damit lässt sich die **Parallelausführung** verbessern, führt jedoch zu Dateninkonsistenzen.



► Auflistung: Arten von Dateninkonsistenzen ▼



Dirty Read ▾

DAMIT IST GEMEINT, DASS VON ANDEREN Transaktionen geschriebene Daten bereits sichtbar sind und gelesen werden können, obwohl die Transaktion noch gar nicht **committed** wurde.

POTENZIELL KÖNNTEN DIE DATEN **ungültig** sein und später durch eine **Rollback** zurückgesetzt werden.



Non repeatable Read ▾

Ein etwas weniger schlimmer Fall wird durch den **Non Repeatable Read** beschrieben.

Der Name röhrt daher, dass während einer **Transaktion** beim mehrfachen Lesen von Daten eventuell **unterschiedliche** Werte zurückgeliefert werden, wenn zwischenzeitlich andere Transaktionen Daten schreiben.

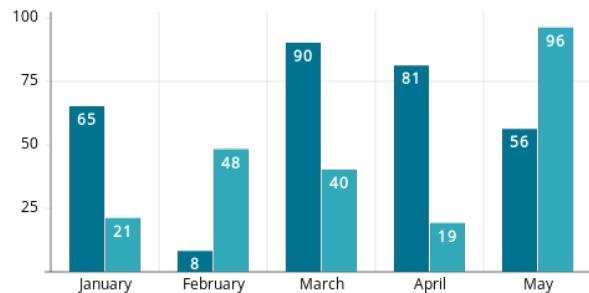


Phantom Read ▾

Die schwächste Form von **Inkonsistenzen** ist unter dem Namen **Phantom Read** bekannt.

HIERBEI KOMMT ES BEIM **Auslesen** VON DATEN ZU ZWEI UNTERSCHIEDLICHEN **Zeitpunkten** ZU EINER ABWEICHENDEN **Ergebnismenge**, WEIL DURCH ZWEISCHENZEITLICH BEENDETE TRANSAKTION DATENSÄTZE HINZUGEFÜGT ODER GELÖSCHT WURDEN.

Je nach gewähltem **Isolationslevel** kann nur eine Teilmenge der zuvor aufgelisteten **Fehlersituatoren** auftreten.



► Auflistung: Isolationslevel ▼

- **Read Uncommitted:** BIETET DIE GERINGSTE **Isolation**, GLEICHZEITIG ABER DIE BESTE **Performance**.

Es können allerdings alle der drei **Fehlerarten** auftreten, also **Dirty Reads**, **Non Repeatable Reads** und **Phantom Reads**.

- **Read Committed:** Es KÖNNEN NUR MEHR **committete Daten** GELESEN WERDEN, WODURCH KEINE **DIRTY READS** MEHR VORKOMMEN KÖNNEN.

Jedoch sind immer noch **Non Repeatable Reads** und **Phantom Reads** möglich.

- **Repeatable Read:** AUF DIESEM ISOLATIONSLEVEL WERDEN HIER VERLÄSSLICHE **Lesezugriffe** GARANTIERT, WORDURCH **DIRTY READS** UND **NON REPEATABLE READS** AUSGESCHLOSSEN SIND.

Trotzdem kann es zu **Phantom Reads** kommen.

- **Serializable:** DIESES LEVEL BIETET DIE HÖCHSTE **Isolationsstufe**, ABER AUCH DIE GERINGSTE PERFORMANCE. KEINER DER **problematischen Reads** KANN AUFTREten.

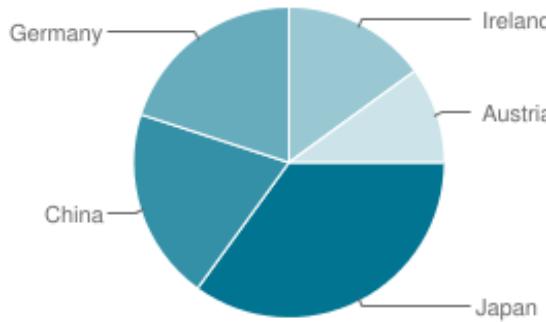
15.6.5 Optimistic Locking

Das **Optimistic Locking** Verfahren bietet **Datenkonsistenz** ohne **Performanceeinbussen** in Kauf nehmen zu müssen.



► Algorithmus: Optimistic Locking ▾

- JEDER **Datensatz** IN DER DATENBANK WIRD MIT EINER **Versionsnummer** VERSEHEN.
- WIRD EIN DATENSATZ GEÄNDERT WIRD, WIRD DIE **Versionsnummer** DES DATENSATZES UM 1 ERHÖHT.
- BEVOR EIN GEÄNDERTER DATENSATZ IN DIE DATENBANK **geschrieben** WERDEN KANN, WIRD DIE VERSIONSNUMMER DES GEÄNDERTEN DATENSATZES MIT DER VERSIONSNUMMER DES KORELIERENDEN DATENSATZES IN DER DATENBANK VERGLIECHEN.
- WENN DIE **Versionsnummern** BEIDER DATENSÄTZE GLEICH SIND, WERDEN DIE **Änderungen** IN DIE DATENBANK ÜBERNOMMEN UND DIE VERSIONSNUMBER DES DATENSATZES UM 1 ERHÖHT.
- SIND DIE VERSIONSNUMMERN UNTERSCHIEDLICH, WERDEN DIE ÄNDERUNGEN NICHT IN DATENBANK ÜBERNOMMEN UND ES WIRD EIN FEHLER AUSGELÖST.
- FÜR DEN FALL, DASS EINE ANDERE **Transaktion** DATEN IN DER DATENBANK ÄNDERT, WÄHREND DIE URSPRÜNGLICHE TRANSAKTION DIE **Daten** BEARBEITET, STELLT DER VERGLEICH DER VERSIONSNUMBER SICHER, DASS DIE **Datenkonsistenz** DER DATEN GEWÄHRT WERDEN KANN.



► Codebeispiel: Optimistic Locking ▾

```

1 //-----
2 // Optimistic Locking
3 //-----
4 @Table(name="PROJECTS")
5 @Entity
6 @RequiredArgsConstructor
7 @NoArgsConstructor
8 public class Project implements Serializable{
9
10    @GeneratedValue
11    @Id
12    @Getter
13    private Long id;
14
15    @NotBlank
16    @Size(max=50)
17    @Column(name="NAME", nullable=false, length=50)
18    @Getter
19    @Setter
20    private String name;
21
22    @Version
23    @Getter
24    private Long version;
25
26 }

```

► Analyse: Optimistic Locking ▾

- UM **Optimistic Locking** ALS **Transaktionsverfahren** EINSETZEN ZU KÖNNEN WIRD FÜR JEDE **Entität** EIN ATTRIBUT VOM TYP **LONG** DEFINIERT.
- DIE **@Version** ANNOTATION DESIGNIERT EIN **Attribut** DER KLASSE UM DIE **Versionsnummer** DER ENTITÄT ZU VERWALTEN.
- MIT DER VERWENDUNG DER **@Version** ANNOTATION WIRD DAS SYSTEM ANGEHALTEN **Optimistic Locking** IM RAHMEN DES **Transaktionsmanagements** EINZUSETZEN.
- DIE ANWENDUNG MUSS DAHINGEHEND NICHT WEITER konfiguriert WERDEN.
- DAMIT IST ES FÜR EINE ANWENDUNG MÖGLICH EINEN NIEDRIGEN **Isolationslevel** IN DER DATENBANK FESTZULEGEN UND TROTZDEM **Datenkonsistenz** ZU GEWÄHRLEISTEN.



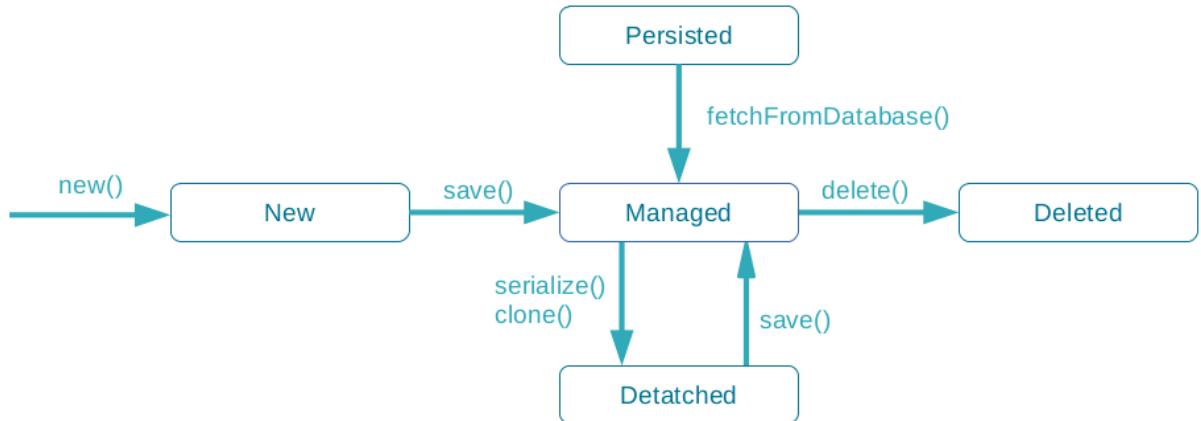


Abbildung 26. Lebenszyklus einer Entität

15.7. Eventhandling

15.7.1 Lebenszyklus einer Entität

Entitäten durchlaufen im Laufe ihrer Lebenszeit verschiedene Zustände.

► Auflistung: Zustände einer Entität ▶

- **New Zustand:** MIT DEM ANLEGEN EINER Entität IM PROGRAMMCODE, WIRD IHR DER Zustand new ZUGEWIESEN.

Die Entität besitzt zu diesem Zeitpunkt noch keine **Repräsentation** in der Datenbank und hat damit auch keine **id**. Solange sich die Entität im **new Zustand** befindet, wird sie nicht durch den **Entity-Manager** verwaltet.

- **Persisted Zustand:** Entitäten DIE EINE Repräsentation IN DER DATENBANK BESITZTEN UND ZUM GEGENWÄRTIGEN ZEITPUNKT NOCH NICHT IN DEN SPEICHER⁵⁵ GELADEN WORDEN SIND, BEFINDEN SICH IM Zustand persisted.

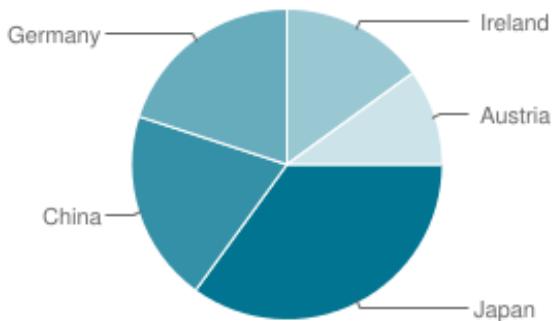
- **Managed Zustand:** MIT DEM SPEICHERN ODER LADEN EINER Entität WIRD IHR DER Zustand Managed ZUGEWIESEN.

Entitäten im Managed Zustand, werden vom **EntityManager** verwaltet. Der EntityManager dient als **1st Level Cache**.

■ **Detached Zustand:** WERDEN ENTITÄTEN ALS **DTOs**⁵⁶ AN DIE ANWENDUNG ÜBERGEBEN, WIRD IHNEN DER **detached** ZUSTAND ZUGEORDNET.

■ **Deleted Zustand:** WIRD EINE Entität GELÖSCHT, WIRD IHR DER ZUSTAND **deleted** ZUGEORDNET.

Eine gelöschte Entität besitzt keine **Repräsentation** in der Datenbank und wird nicht durch den **Entity-Manager** verwaltet.



⁵⁵ EntityManager

⁵⁶ Data Transfer Object - Webservice

15.7.2 Callback Methoden

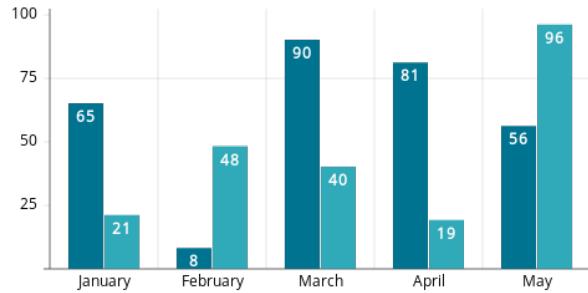


Rückruffunktionen ▾

EINE **Rückruffunktion** BEZEICHNET IN DER INFORMATIK EINE FUNKTION, DIE EINER ANDEREN **Funktion**, ALS **Parameter** ÜBERGEBEN UND VON DIESER UNTER DEFINIERTEN BEDINGUNGEN MIT DEFINIERTEN ARGUMENTEN AUFGERUFT WIRD.

DIESES VORGEHEN FOLGT DEM ENTWURFSMUSTER DER **Inversion of Control**.

JPA unterstützt vordefinierte **Rückruffunktionen** um benutzerdefinierte **Logik** im Prozess der Entitätsverwaltung auszuführen.



► Auflistung: JPA - Rückruffunktion ▾

- **@PrePersist** EINE MIT **@PrePersist** ANNOTIERTE METHODE WIRD AUFGERUFEN BEVOR EINE **Entität** GESPEICHERT WIRD.
- **@PostPersist** EINE MIT **@PostPersist** ANNOTIERTE METHODE WIRD AUFGERUFEN NACHDEM EINE **Entität** GESPEICHERT WORDEN IST.
- **@PostLoad** EINE MIT **@PostLoad** ANNOTIERTE METHODE WIRD AUFGERUFEN NACHDEM EINE **Entität** AUS DER DATENBANK GELADEN WIRD.
- **@PreUpdate** EINE MIT **@PreUpdate** ANNOTIERTE METHODE WIRD AUFGERUFEN BEVOR ÄNDERUNGEN AN EINER **Entität** GESPEICHERT WORDEN SIND.
- **@PostUpdate** EINE MIT **@PostUpdate** ANNOTIERTE METHODE WIRD AUFGERUFEN NACHDEM ÄNDERUNGEN IN DER **Entität** GESPEICHERT WORDEN SIND.
- **@PreRemove** EINE MIT **@PreRemove** ANNOTIERTE METHODE WIRD AUSGEFÜHRT BEVOR EINE **Entität** GELÖSCHT WIRD.

► Codebeispiel: JPA Rückruffunktionen ▾

```

1 //-----
2 // JPA Rückruffunktionen
3 //-----
4 @EntityListeners(ProjectListener.class)
5 @Table(name="PROJECTS")
6 @Entity
7 @RequiredArgsConstructor
8 @NoArgsConstructor
9 public class Project implements Serializable{
10
11     @GeneratedValue
12     @Id
13     @Getter
14     private Long id;
15
16     @NotBlank
17     @Size(max=50)
18     @Column(name="NAME", nullable=false, length=50)
19     @Getter
20     @Setter
21     private String name;
22
23     @Version
24     @Getter
25     private Long version;
26
27 }
28 //-----
29 // JPA Rückruffunktionen
30 //-----
31
32 public class ProjectListener{
33
34     @PrePersist
35     public void prePersist(Project project){
36         System.out.println("Listening Project:
37             prepersist");
38     }
39
40     @PostPersist
41     public void postPersist(Project project){
42         System.out.println("Listening Project:
43             postPersist");
44     }
45
46     @PreUpdate
47     public void preUpdate(Project project){
48         System.out.println("Listening Project:
49             preUpdate");
50     }
51
52     @PostUpdate
53     public void postUpdate(Project project){
54         System.out.println("Listening Project:
55             postUpdate");
56     }
57 }
```



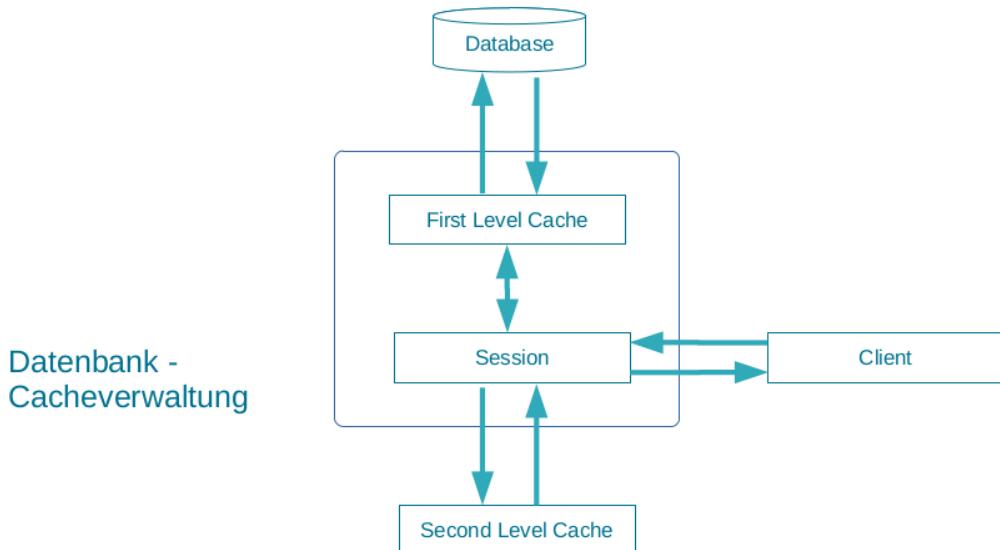


Abbildung 27. Cacheverwaltung für Datenbankanwendungen

15.8. Caching

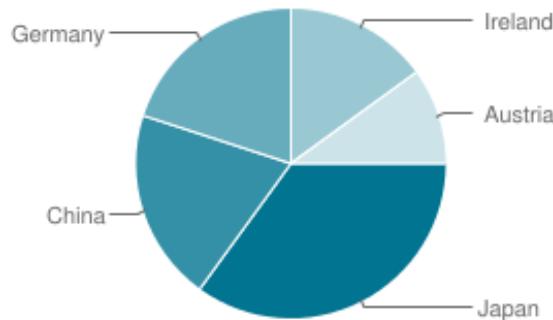
15.8.1 Grundlagen



Cache

EIN Cache BEZEICHNET EINEN Puffer Speicher, DER HILFT WIEDERHOLTE ZUGRiffe AUF DIE Datenbank ZU VERHINDERN.

Gerade im Bereich der Datenbanken kann Caching zu einer signifikanten Verbesserung der Performance führen.

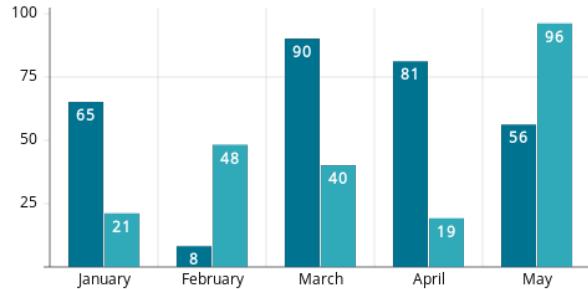


► Analyse: Laden einer Entität ▾

- EINE ENTITÄT SOLL ZUR WEITEREN VERARBEITUNG GELADEN WERDEN.
- EIN ENTSPRECHENDER REQUEST WIRD AN DIE ClientSession ABGESETZT.
- DIE ClientSession INITIERT EINE SUCHE NACH EINER REFERENZ DER ENTITÄT IM First Level Cache.
- ENTHÄLT DER FIRST LEVEL CACHE KEINE REFERENZ DER ENTITÄT MUSS DIE ENTITÄT AUS DER Datenbank GELADEN WERDEN.
- DIE ENTITÄT WIRD IM First Level Cache GEHASHED UND EINE REFERENZ DER ENTITÄT AN DEN SessionClient ZURÜCKGEGEBEN.
- IST EIN Second Level Cache FÜR DIE ANWENDUNG KONFIGURIERT, INITIERT DER First Level Cache ZUERST EINE SUCHE IM Second Level Cache, BEVOR DIE ENTITÄT AUS DER DATENBANK GELADEN WIRD.
- WIRD EINE REFERENZ IM Second Level Cache GEFUNDEN WIRD SIE IM First Level Cache GEHASHED UND EINE REFERENZ AN DEN SessionClient ZURÜCKGEGEBEN.

15.8.2 Arten von Caches

Ein Cache besteht in der Regel aus mehreren Schichten, die sich in Größe und Zugriffszeit unterscheiden.



► Auflistung: Arten von Caches ▾



First Level Cache ▾

FÜR JEDEN BENUTZER WIRD EIN EIGENER **First Level Cache** VERWALTET. DER CACHE WIRD DABEI FÜR JEDEN Request NEU ANGELEGT.



Second Level Cache ▾

DER **Second Level Cache** CACHED ENTITÄTEN ALLER User. DIE DES CACHES IST AN DIE Lebensdauer DER ANWENDUNG GEBUNDEN.



15.8.3 Verwendung von Caches

► Analyse: Vor- und Nachteile ▾

- DER EINSATZ VON **Caching** KANN EINE ENORME VERRBESSERUNG IN DER **Performance** FÜHREN, WEIL DATEN NICHT MEHR VON DER DATENBANK, SONDERN AUS DEM IM **Hauptspeicher** BEHEIMATEDEN CACHE **gelesen** WERDEN.
- ALLERDINGS SOLLTE MAN AUCH DIE **Nachteile** BEDENKEN: DER CACHE BELEGT **Hauptspeicher** - MITUNTER EINEN NICHT UNERHEBLICHEN TEIL.
- ZUDEM BRINGT DIE **Zwischenspeicherung** VON DATEN IM HAUPTSPEICHER DIE GEFahr FÜR **Inkonsistenzen** MIT DEN WERTEN AUS DER DATENBANK.



16. Service - Programmierung der REST Serviceschicht

03

Serviceschicht

01 ... Struktur eines Services	139
02 ... Http Protokoll	140
03 ... Service Endpoint	144
04 ... Json Repräsentation	148

16.1. Struktur eines Services

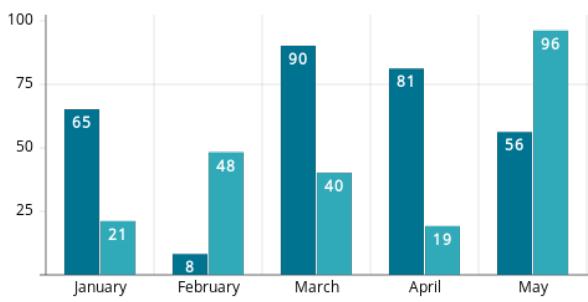


Service ▾

EIN Service IST EINE Softwarekomponente, AUSGEFÜHRT IN EINEM EIGENEN Betriebssystemprozess.

► Erklärung: SOA System ▾

- EINE SOA Anwendung IST EINE Komposition VON Servicen.
- SOA Systeme VERBINDELN SERVICE ZU EINEM Gesamtsystem.
- EIN Service Besteht IN DER Regel AUS 3 Schichten.
- JEDE DER Schichten ERFÜLLT EINE EIGENE AUFGABE.



16.1.1 Schichten eines Services

Ein Service ist softwaretechnisch in Schichten aufgeteilt. In der Regel unterscheiden wir für ein Microservice 3 Schichten.

► Erklärung: Schichten eines Service ▾

- **Modelschicht:** DIE Modelschicht ENTSPRICHT DER Struktur DES Datarepository⁵⁷ DES SERVICES.

Die Schicht beinhaltet jene Klassen⁵⁸, die notwendig sind um Daten aus dem Datenrepository zu kapseln. Klassen der Modelschicht implementieren kein Verhalten.

⁵⁷ SQL Datenbank, NoSQL Datenbank, Dateien usw.

⁵⁸ Objekte deren Aufgabe es ist in erster Linie Daten zu transportieren werden DataTransferObjects genannt

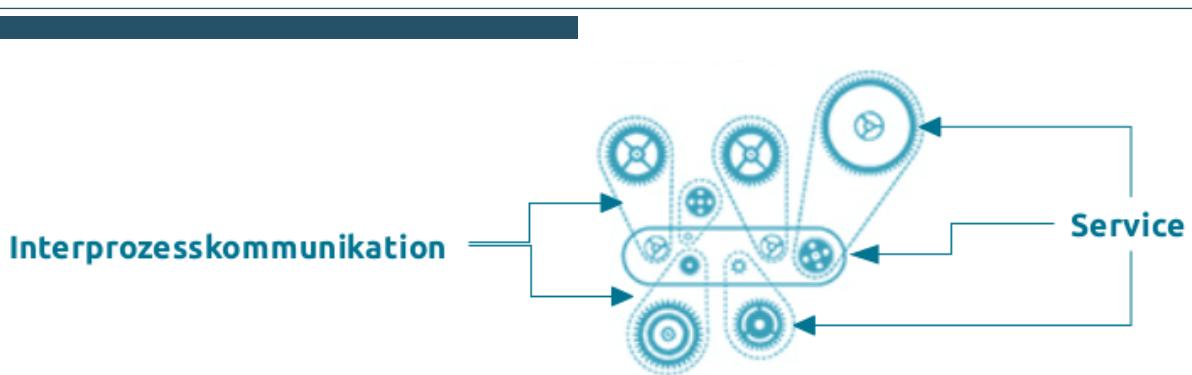


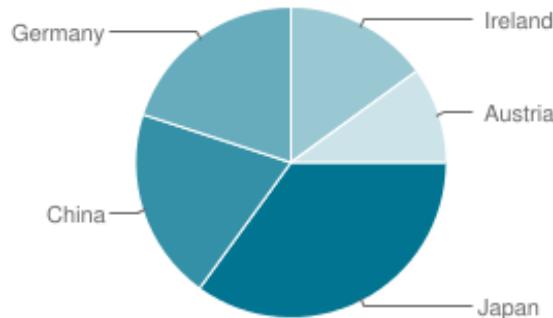
Abbildung 28. SOA - Anwendung

■ **Domainschicht:** DIE Domainschicht BEINHALTET DIE Logik DES Services.

Die Schicht kapselt die **datenlesende-** und **datenschreibende** Funktionalität des Services. Die Klassen der Domäinschicht verwalten in der Regel keinen Zustand sondern stellen **Verhalten** zur Verfügung.

■ **Serviceschicht:** DIE Serviceschicht IST VERANTWORTLICH FÜR DIE **Kommunikation** DER SERVICE UNTEREINANDER.

Die **Kommunikation** der Service erfolgt über **plattformunabhängige**⁵⁹ Protokolle. Damit ist es möglich Service in unterschiedlichen **Technologien** zu implementieren und sie trotzdem zu einer Anwendung zu **integrieren**.



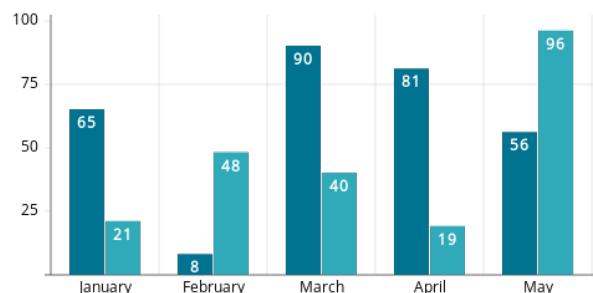
16.2. Http Protokoll



Serviceschicht

DIE Serviceschicht IST VERANTWORTLICH FÜR DIE Kommunikation DER SERVICE UNTEREINANDER.

Die **Kommunikation** der Service erfolgt über **plattformunabhängige** Protokolle.



16.2.1 HTTP Protokoll

Rest verwendet das **Http Protokoll** als **plattformunabhängiges** Protokoll.



http protokoll

DAS **http Protokoll** IST EIN **zustandsloses** PROTOKOLL ZUR **Übertragung** VON DATEN AUF DER ANWENDERSCHICHT IM NETZWERK.

⁵⁹ *Http Protokoll, SOAP usw.*

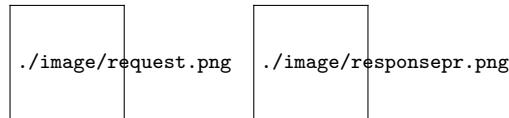
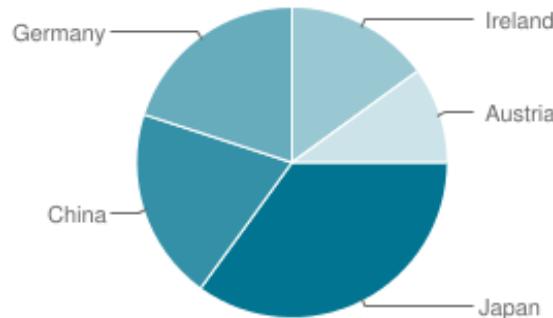


Abbildung 29. Http Request - Http Response

► Erklärung: HTTP Protokoll ▾

- DAS **http Protokoll** BILDET DIE BASIS DER **Kommunikation** IM INTERNET.
- BEIM **http Protokoll** HANDELT ES SICH UM EIN **Klar-textprotokoll** MIT EINER ZAHL VON VORGEgebenEN **Client Befehlen** UND WOHLDEFINIERTEN SERVER **Antworten**.
- DAS PROTOkOLL BERUHT DABEI AUF EINEM **Request/Response** Zyklus FÜR DIE KOMMUNIKATION.
- FÜR DIE **Kommunikation** ZWISCHEN CLIENT UND SERVER WERDEN **Nachrichten** AUSGETAUSCHT.
- WIR UNTERScheiden DABEI 2 ARTEN VON **Nachrichten**: DIE **Anfrage** (REQUEST) VOM CLIENT AN DEN SERVER UND DIE **Antwort** (RESPONSE) ALS REAKTION DARAUF VOM SERVER ZUM CLIENT.
- JEDE **Nachricht** BESTEHT DABEI AUS 2 TEILEN, DEM **Nachrichtenkopf** UND DEM **Nachrichtenrumpf**.
- DER **Nachrichtenkopf** ENTHÄLT **Informationen** ÜBER DEN NACHrichtenRUMPF WIE ETWA DIE VERWENDETE KODIERUNGEN ODER DEN INHALTSTYP.
- DER **Nachrichtenrumpf** ENTHÄLT SCHLIESSLICH DIE **Nutzdaten**.

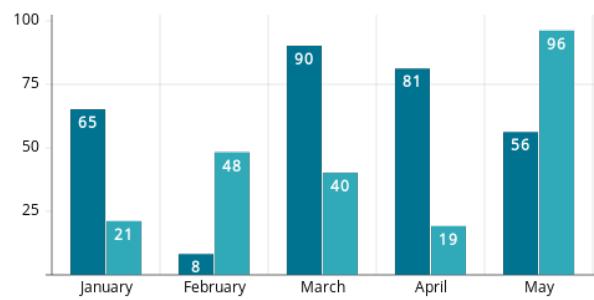


16.2.2 Anatomie einer Klientanfrage

Jede Nachricht besteht immer aus 2 Teilen, dem **Nachrichtenkopf** und dem **Nachrichtenrumpf**.

► Erklärung: Http Request ▾

- **http methode:** ALS ERSTES WIRD IM **Request** ANGEGEBEN WELCHE DER **Http Methoden** AM SERVER AUFGERUFEN WIRD.
- **host header:** IM **host header** WIRD DIE **hostadresse** DES GEWÜNSCHTEN SERVERS ANGEgeben.
- **accept header:** MIT DEM **accept header** WIRD ANGEGEBEN WELCHES **Repräsentationsformat** SICH DER CLIENT VOM SERVER ERWARTET.



16.2.3 Anatomie einer Serverantwort

Genau wie die **Klientanfrage** folgt die **Serverantwort** einer vorgegebenen Strukturierung.

► Erklärung: Http Response ▾

- **Statuscode:** DER **Statuscode** GIBT AN OB DER **Request** ERFOLGREICH VERARBEITET WURDE.
- **Responsebody:** IM **Responsebody** WIRD DIE EIGENTLICHE ANTWERFT DES SERVERS MITGESCHICKT.

Code	Bedeutung	Nachricht
100	Die laufende Anfrage an den Server wurde noch nicht zurückgewiesen.	Continue
101	Wird verwendet, wenn der Server eine Anfrage mit gesetztem „Upgrade“-Header-Feld empfangen hat und mit dem Wechsel zu einem anderen Protokoll einverstanden ist. Anwendung findet dieser Status-Code beispielsweise im Wechsel von HTTP zu WebSocket.	Switching Protocols
200	Die Anfrage wurde erfolgreich bearbeitet und das Ergebnis der Anfrage wird in der Antwort übertragen.	Ok
201	Die Anfrage wurde erfolgreich bearbeitet. Die angeforderte Ressource wurde vor dem Senden der Antwort erstellt. Das „Location“-Header-Feld enthält eventuell die Adresse der erstellten Ressource.	Created
202	Die Anfrage wurde akzeptiert, wird aber zu einem späteren Zeitpunkt ausgeführt. Das Gelingen der Anfrage kann nicht garantiert werden.	Accepted
400	Die Anfrage-Nachricht war fehlerhaft aufgebaut.	Bad Request
401	Die Anfrage kann nicht ohne gültige Authentifizierung durchgeführt werden. Wie die Authentifizierung durchgeführt werden soll, wird im „WWW-Authenticate“-Header-Feld der Antwort übermittelt.	Unauthorized
403	Die Anfrage wurde mangels Berechtigung des Clients nicht durchgeführt, bspw. weil der authentifizierte Benutzer nicht berechtigt ist, oder eine als HTTPS konfigurierte URL nur mit HTTP aufgerufen wurde.	Forbidden
404	Die angeforderte Ressource wurde nicht gefunden. Dieser Statuscode kann ebenfalls verwendet werden, um eine Anfrage ohne näheren Grund abzuweisen. Links, welche auf solche Fehlerseiten verweisen, werden auch als Tote Links bezeichnet.	Not Found
405	Die Anfrage darf nur mit anderen HTTP-Methoden (zum Beispiel GET statt POST) gestellt werden. Gültige Methoden für die betreffende Ressource werden im „Allow“-Header-Feld der Antwort übermittelt.	Method Not Allowed
500	Dies ist ein „Sammel-Statuscode“ für unerwartete Serverfehler.	Internal Server Error
504	Der Server konnte seine Funktion als Gateway oder Proxy nicht erfüllen, weil er innerhalb einer festgelegten Zeitspanne keine Antwort von seinerseits benutzten Servern oder Diensten erhalten hat.	Gateway Timeout
501	Die Funktionalität, um die Anfrage zu bearbeiten, wird von diesem Server nicht bereitgestellt. Ursache ist zum Beispiel eine unbekannte oder nicht unterstützte HTTP-Methode.	Not Implemented
511	Der Client muss sich zuerst authentifizieren um Zugang zum Netzwerk zu erhalten.	Network Authentication Required

Abbildung 30. Http Statuscode

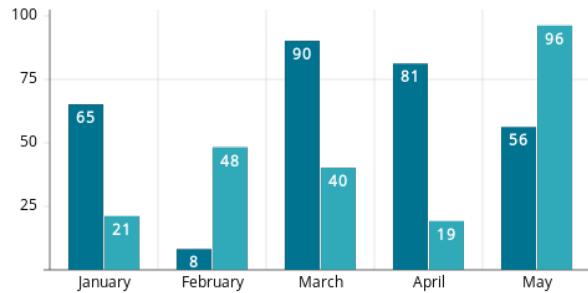
16.2.4 Statuscode

http Statuscode ▾

DER **http Statuscode** WIRD VON EINEM SERVER AUF JEDE **http Anfrage** ALS ANTWORT GELIEFERT. DER STATUSCODE ZEIGT DEM CLIENT AN OB DIE ANFRAGE ERFOLGREICH VERARBEITET WERDEN KONNTE.

► Erklärung: **Http Statuscode** ▾

- DER **http Statuscode** IST EIN 3 STELLIGER **Zahlencode**.
- MIT DER ERSTEN STELLE DES ZAHLENCODES WIRD DIE **Gruppe des Statuscodes** ANGEZEIGT. DIE RESTLICHEN STELLEN SIND FÜR DIE DIFFERENZIERUNG INNERHALB DER **Gruppe** VERANTWORTLICH.



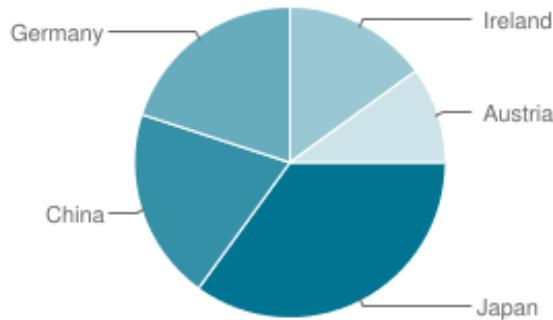
► Erklärung: **Statuscodegruppen** ▾

- **1xx - Status:** DER REQUEST BEFINDET SICH NOCH IN **Bearbeitung**.
- **2xx - Erfolgreiche Verarbeitung:** - DER REQUEST KONNTE VOM SERVER ERFOLGREICH VERARBEITET WERDEN.
- **3xx - Umleitung:** UM EINE ERFOLGREICHE BEARBEITUNG DER ANFRAGE SICHERZUSTELLEN, SIND WEITER SCHritte SEITENS DES CLIENTS ERFORDERLICH.
- **4xx Client Fehler:** - DIR URSACHE FÜR DAS SCHEITERN DES CLIENTS LIEGT IM VERANTWORTUNGSBEREICH DES CLIENTS.
- **5xx Server Fehler:** - DIE URSACHE FÜR DAS SCHEITERN DER ANFRAGE LIEGT IM VERANTWORTUNGSBEREICH DES SERVERS.

16.2.5 Http Methoden

Eine **Rest Anwendung** besteht im eigentlichen Sinn nicht aus Servicen. Eine REST Anwendung veraltet **Ressourcen**.

Aus diesem Grund werden **Rest Webservice**, oft auch nicht als **SOA**, sondern als **ROA**⁶⁰ bezeichnet. **Ressourcen** stellen zur Verwaltung **Schnittstellen** zur Verfügung.



► Erklärung: **Schnittstelle einer Ressource** ▾

- WIRD FÜR EINE REST ANWENDUNG **HTTP** ALS TRANSPORT PROTOKOLL VERWENDET, KANN JEDOCH **Ressource** ÜBER DIE **HTTP Methoden** BEARBEITET WERDEN.
- FÜR JEDOCH **Ressource** DEFINIERT DAS HTTP PROTOKOLL EIN **generisches Interface** BESTEHEND AUS DEN **Http Methoden**.
- ZUSÄTZLICH ZU DEN **Http Methoden** KÖNNEN RESOURCEN METHODEN ZUR VERWALTUNG IHRES ZUSTANDES ZUR VERFÜGUNG STELLEN.

► Codebeispiel: **Schnittstelle einer Ressource** ▾

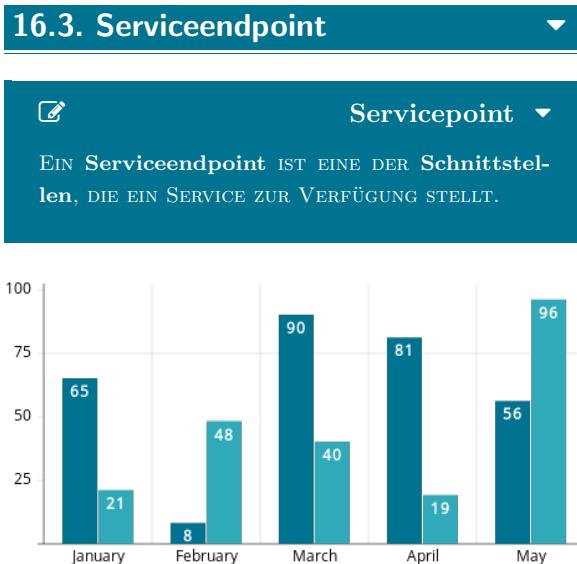
```

1 public interface IHttpRessource{
2     //Repraesentation der Ressource anfordern
3     HttpResponse get(HttpServletRequest request);
4
5     //Ressource verndern
6     HttpResponse put(HttpServletRequest request);
7
8     //Ressource anlegen
9     HttpResponse post(HttpServletRequest request);
10
11    //Ressource loeschen
12    HttpResponse delete(HttpServletRequest request);
13 }
```

⁶⁰ *Ressource Oriented Architecture*



Abbildung 31. Serviceendpoint url



16.3.1 Serviceendpoint Aufruf

Zum **Aufruf** der **Methoden** eines Serviceendpoints muß der Klient die **url** des Endpoint und die mit der Methode **assoziierte Httpmethode** angegeben.

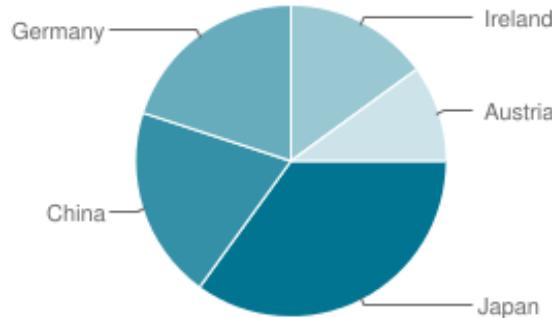
► Codebeispiel: Serviceendpoint ▾

```

1 //-----
2 // Aufruf einer Methode des Serviceendpoints
3 //-----
4 @RequestMapping(path = "/projects")
5 @RestController
6 public class ProjectService {
7
8     @Autowired
9     private IProjectRepository repository;
10
11    @GetMapping(path = "requestserviceping"
12                produces = MediaType.TEXT_PLAIN_VALUE)
13    @ResponseStatus(HttpStatus.OK)
14    public String ping(){
15        log.info("hallo welt");
16
17        return "hallo welt";
18    }
19 }
```

► Analyse: Serviceendpoint url ▾

- **Serverhost:** DIE **IP Adresse** UND DER **Port** DES HOSTS AUF DEM DAS **Service** AUSGEFÜHRT WIRD.
- **Servicename:** DER NAME DES **Webservices**. DER NAME DES SERVICES WIRD ÜBER DIE **SERVER.CONTEXT-PATH** PROPERTY DEFINIERT.
- **endpointURL:** DIE URL DES **Serviceendpoints** WIRD IN DER **@REQUESTMAPPING** ANNOTATION MIT DEM PATH ATTRIBUT ANGEGEBEN.
- **methodURL:** DIE URL DER GEWÜNSCHTEN **Methode** WIRD IN DER **@GETMAPPING** ANNOTATION MIT DEM PATH ATTRIBUT ANGEGEBEN.



► Analyse: Angebe der Httpmethode ▾

- WIR ERINNERN UNS: FÜR DEN AUFRUF DER **Methode** EINES **Serviceendpoints** MUSS DIE URL DES ENDPOINTS UND DIE MIT DER METHODE ASSOZIERTE **http Methode** ANGEGEBEN WERDEN.
- DAMIT EINER METHODE EINE DER **Httpmethoden** ZUGEORDNET WERDEN KANN, WERDEN EIGENS DAFÜR KONZIPIERTE ANNOTATIONEN VERWENDET: **GETMAPPING**, **PUTMAPPING**, **POSTMAPPING**, **DELETEMAPPING**.
- IN DIESEN ANNOTATIONEN WIRD AUCH ANGEGEBEN WELCHEM **Mimetype** DER RÜCKGABEWERT DER METHODE ENTSPRICHT. DIE ANGABE ERFOLGT HIER ÜBER DAS **PRODUCES** ATTRIBUT.

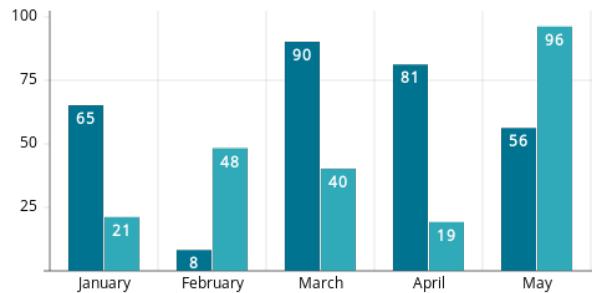
16.3.2 Anlegen von Ressourcen

Zum Anlegen von **Ressourcen** verwenden wir die **Post Methode**.

► Codebeispiel: Ressourcen anlegen ▾

```

1  //-----
2  // Ressourcen anlegen
3  //-----
4  @RequestMapping(path = "/projects")
5  @RestController
6  public class ProjectService {
7
8      private static Logger log =
9          LoggerFactory.getLogger(PService.class);
10
11     @Autowired
12     private IProjectRepository projectRepository;
13
14     @PostMapping(produces =
15         {MediaType.APPLICATION_JSON_UTF8_VALUE,
16          MediaType.APPLICATION_XML_VALUE})
17     @ResponseStatus(HttpStatus.CREATED)
18     public AProject create(@RequestBody @Valid
19                           AProject project){
20
21         projectRepository.save(project);
22         log.info("created project: " +
23                 project.getId());
24
25         return project;
26     }
27
28 }
```

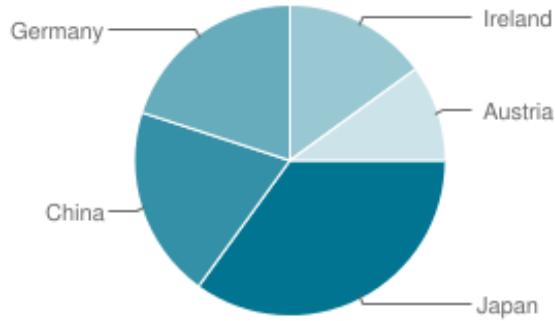


► Analyse: Ressourcen anlegen ▾

- **@RequestBody:** DIE DATEN FÜR DAS **APROJECT** OBJEKT WERDEN AUTOMATISCH AUS DEM **Httprequest** DURCH DIE VERWENDUNG DER **@REQUESTBODY** ANNOTATION EXTRAHIERT.
- **@PostMapping:** DAS **PRODUCES** ATTRIBUT DER **@POSTMAPPING** ANNOTATION GIBT DIE ART DER **Res-sourcenrepräsentation** AN DIE DIE METHODE AN DIE HTTPANTWORT ZURÜCKGIBT.

- **accept:** DIE ART DER **Ressourcenrepräsentation** WIRD DABEI IM **ACCEPT HEADER** DER **Httpanfrage** AN-**GEGBEN**.

- **ResponseStatus:** KONNTE DIE **Ressource** ERFOLG-**REICH ANGELEGT WERDEN, WIRD DER **Statuscode** CREATED FÜR DIE HTTPANTWORT GESETZT.**



► Codebeispiel: Aufruf der Endpointmethode ▾

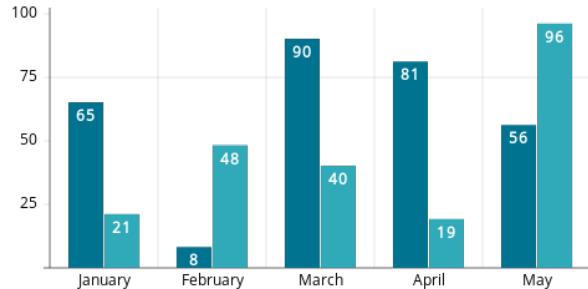
```

1  //-----
2  // Ressourcen anlegen
3  //-----
4  @RunWith(SpringRunner.class)
5  @SpringBootTest(webEnvironment =
6      SpringBootTest.WebEnvironment.DEFINED_PORT)
7  public class ProjectIntegrationTest {
8
9      @Test
10     public void createBranch(){
11         String requestURL =
12             "http://127.0.0.1:8181/projectservice/projects";
13         RestTemplate restClient = new
14             RestTemplate();
15
16         HttpHeaders headers = new HttpHeaders();
17         headers.setAccept(
18             Arrays.asList(MediaType.APPLICATION_JSON_UTF8));
19
20         AProject project =
21             ProjectFactroy.createProject();
22
23         HttpEntity<AProject> request = new
24             HttpEntity<>(project, headers);
25         ResponseEntity<AProject> response =
26             restClient.postForEntity(requestURL,
27             request, AProject.class);
28
29         assertEquals(HttpStatus.CREATED,
30             response.getStatusCode());
31         AProject createdProject =
32             response.getBody();
33
34         assertNotNull(createdProject.getId());
35     }
36 }
```



16.3.3 Ressourcen verwalten

Wir wollen uns nur ein Fallbeispiel für ein komplettes CRUD Service anschauen.



▶ Codebeispiel: Fallbeispiel: CRUD Service ▾

```

1 //-----
2 // Ressourcen verwalten
3 //-----
4 @RequestMapping(path = "/projects")
5 @RestController
6 public class ProjectService {
7
8     private static Logger log =
9         LoggerFactory.getLogger(PService.class);
10
11    @Autowired
12    private IProjectRepository repository;
13
14    @GetMapping(produces =
15        MediaType.TEXT_PLAIN_VALUE)
16    @ResponseStatus(HttpStatus.OK)
17    public String ping(){
18        log.info("hallo welt");
19
20        return "hallo welt";
21    }
22
23    @PostMapping(produces =
24        {MediaType.APPLICATION_JSON_UTF8_VALUE,
25         MediaType.APPLICATION_XML_VALUE})
26    @ResponseStatus(HttpStatus.CREATED)
27    public AProject create(@RequestBody @Valid
28                           AProject project){
29
30        repository.save(project);
31        log.info("created project: " +
32               project.getId());
33
34        return project;
35    }
36
37    @PutMapping("/{projectID}")
38    @ResponseStatus(HttpStatus.NO_CONTENT)
39    public void update(@RequestBody @Valid
40                       AProject project,
41                       @PathVariable("projectID") Long projectId){
42
43        repository.save(project);
44        log.info("updated project " + projectId);
45    }
46
47 }
```

```

35
36     repository.save(project);
37     log.info("updated project " + projectId);
38 }
39
40 @DeleteMapping("/{projectID}")
41 @ResponseStatus(HttpStatus.NO_CONTENT)
42 public void delete(@PathVariable("projectID")
43                     Long projectID){
44     repository.delete(projectID);
45     log.info("deleted project " + projectID);
46 }
47 }
```

▶ Codebeispiel: TestCode ▾

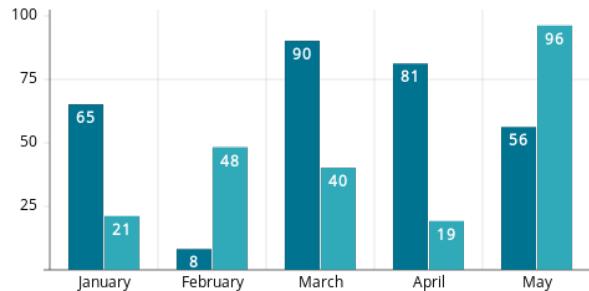
```

1 //-----
2 // Ressourcen verwalten
3 //-----
4 @RunWith(SpringRunner.class)
5 @SpringBootTest(webEnvironment =
6     SpringBootTest.WebEnvironment.DEFINED_PORT)
7 public class ProjectIntegrationTest {
8
9     @Autowired
10    private IProjectRepository projectRepository;
11
12    @Test
13    public void updateProject(){
14        AProject project =
15            projectRepository.findOne(11);
16        assertNotNull(project);
17
18        project.setTitle("updated project title");
19
20        String requestURL =
21            "http://localhost:8181/projectService/projects/1";
22        RestTemplate restClient = new
23            RestTemplate();
24
25        HttpEntity<AProject> request = new
26            HttpEntity<>(project);
27        restClient.exchange(requestURL,
28            HttpMethod.PUT, request, Void.class);
29    }
30
31    @Test
32    public void deleteProject(){
33        AProject project =
34            projectRepository.findOne(11);
35        assertNotNull(project);
36
37        String requestURL =
38            "http://localhost:8181/projectService/projects/1";
39        RestTemplate restClient = new
40            RestTemplate();
41
42        restClient.delete(requestURL);
43    }
44 }
```



16.3.4 Übergabeparameter validieren

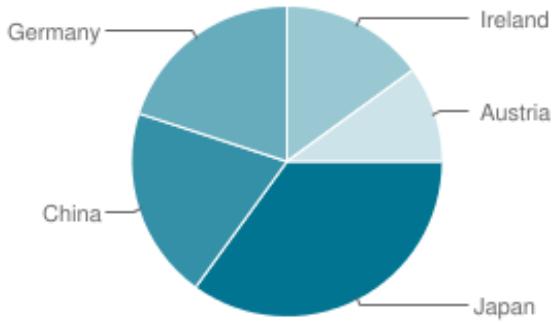
Bevor ein **Serviceendpoint** Daten verarbeiten soll, muß die **Richtigkeit** der **Serviceparameter** sichergestellt werden.



► Codebeispiel: Validieren von Parametern ▾

```

1 //-----
2 // Service Endpoint
3 //-----
4 @RequestMapping(path = "/projects")
5 @RestController
6 public class ProjectService {
7
8     private static Logger log =
9         LoggerFactory.getLogger(PService.class);
10
11    @Autowired
12    private IProjectRepository repository;
13
14    @PostMapping(produces =
15        {MediaType.APPLICATION_JSON_UTF8_VALUE,
16         MediaType.APPLICATION_XML_VALUE})
17    @ResponseStatus(HttpStatus.CREATED)
18    public AProject create(@RequestBody @Valid
19                           AProject project) {
20
21        repository.save(project);
22        log.info("created project: " +
23                project.getName());
24
25        return project;
26    }
27
28 //-----
29 // Error DTO
30 //-----
31 @Data
32 @AllArgsConstructor
33 public class ErrorDetails {
34
35     private Date timestamp;
36     private String message;
37     private List<String> errors;
38
39     //-----
40     // ValidationHandler
41     //-----
42     @ControllerAdvice
43     public class ValidationHandler extends
44         ResponseEntityExceptionHandler {
45
46         @Override
47         protected ResponseEntity<Object>
48             handleMethodArgumentNotValid(
49             MethodArgumentNotValidException ex,
50             HttpHeaders headers, HttpStatus status,
51             WebRequest request) {
52
53             List<String> errors = new
54                 ArrayList<String>();
55
56             for(FieldError error :
57                 ex.getBindingResult().getFieldErrors()){
58                 errors.add(error.getField() + ":" +
59                         error.getDefaultMessage());
60             }
61
62             for(ObjectError error :
63                 ex.getBindingResult().getGlobalErrors()){
64                 errors.add(error.getObjectName() + ":" +
65                         error.getDefaultMessage());
66             }
67
68             ErrorDetails errorDetails = new
69                 ErrorDetails(new Date(), "Validation
70                 Failed",
71                 errors);
72
73             return handleExceptionInternal(ex,
74                 errorDetails,headers,
75                 HttpStatus.BAD_REQUEST, request);
76         }
77     }
78 }
```



► Analyse: Validieren von Parametern ▾

- **@Valid:** DIE **@Valid** ANNOTATION LEITET DIE PRÜFUNG EINES Serviceparameters EIN.
- **ValidationHandler:** IM **ValidationHandler** LEGT DER PROGRAMMIERER FEST, WIE IM ETWAIGEN FEHLERFALL ZU VERFAHREN IST.



► Codebeispiel: Testcode ▾

```

1 //-----
2 // Service Endpoint
3 //-----
4 @RunWith(SpringRunner.class)
5 @SpringBootTest(webEnvironment =
6     SpringBootTest.WebEnvironment.DEFINED_PORT)
7 public class ProjectIntegrationTest {
8
9     @Test
10    public void createProjectWithInvalidData(){
11        String requestURL =
12            "http://127.0.0.1:8082/project/projects"
13        RestTemplate restClient = new
14            RestTemplate();
15
16        HttpHeaders headers = new HttpHeaders();
17        headers.setAccept(Arrays.asList(
18            MediaType.APPLICATION_JSON_UTF8));
19
20        AProject project = new
21            RequestFundingProject();
22
23        HttpEntity<AProject> request = new
24            HttpEntity<>(project, headers);
25        ResponseEntity<AProject> response = null;
26
27        try {
28            restClient.postForEntity(requestURL,
29                request, AProject.class);
30        }catch(HttpStatusConversionException exception){
31            log.info(exception.getResponseBodyAsString());
32        }
33    }
34 }
```

16.4. Json Repräsentation ▾



json ▾

DIE JavaScript Object Notation, KURZ **Json** IST EIN DATENFORMAT FÜR DEN **Datenaustausch** ZWISCHEN ANWENDUNGEN.

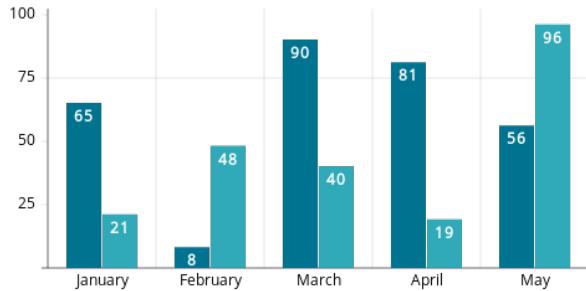


16.4.1 JSON Grundlagen

Json wird zur Übertragung und zum Speichern von strukturierten Daten eingesetzt.

► Erklärung: JSON ▾

- JSON STAMMT URSPRÜNGLICH AUS DEM **JavaScript** UMFELD. JEDES GÜLTIGE **Json Dokument** IST GÜLTIGER **JavaScript Code**.
- ABGESEHEN DAVON WIRD **JSON** IN ALLEN GÄNGIGEN PROGRAMMERSPRACHEN **unterstützt**.
- JSON IST IN ERSTER LINIE EIN **Datenaustauschformat**.



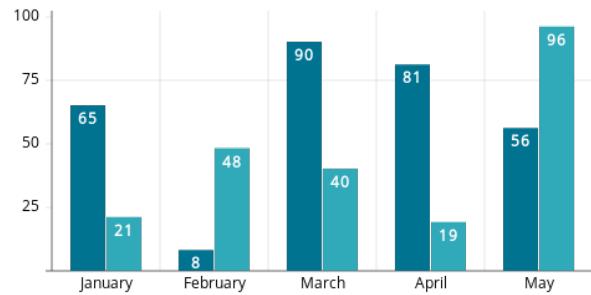
► Erklärung: JSON Grundstruktur ▾

- AUF OBERSTER EBENE UNTERSCHIEDEN WIR IN JSON ZWISCHEN **Objekten** UND **Arrays**.
- OBJEKTE UND ARRAYS SELBST KÖNNEN WIEDER AUS OBJEKten UND ARRAYS BESTEHEN.



16.4.2 JSON - Objekte und Arrays

Die grundlegenden Strukturen in Json sind **Objekte** und **Arrays**.



./image/array2.png

► Erklärung: JSON Grundstrukturen ▾

- **Objekte:** EINE SAMMLUNG VON **Schlüssel/Werte Paaren**
- **Arrays:** EINE GEORDNETE **Liste von Werten**.

► Codebeispiel: JSON Grundstrukturen ▾

```
1 //-----  
2 // JSON Objekt  
3 //-----  
4 var person = {  
5   string : value,  
6   string : value,  
7   string : value,  
8   string : value,  
9   string : value,  
10  string : value,  
11  string : value,  
12  string : value,  
13  string : value,  
14  string : [],  
15  string : value  
16 }  
17 }
```



./image/object2.png

16.4.3 JSON - Strings und Werte

Objekte und Arrays setzen sich aus **Strings** und **Werten** zusammen. Für Strings und Werte können wir die folgenden Diagramme angeben.

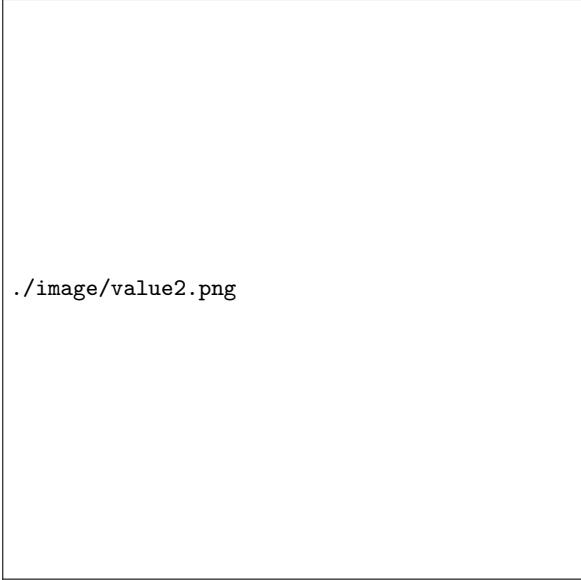
► Erklärung: String und Werte ▾

./image/string2.png

```

5   "firstName": "John",
6   "lastName" : "Smith",
7   "isAlive"  : "true",
8   "age"      : number,
9   "height_cm": number,
10
11  "address" : {
12    "street"   : "21 2nd Street",
13    "city"     : "New York",
14    "postalCode": "10021-2100"
15  },
16
17  "phoneNumbers" : [
18    {
19      "type"   : "home",
20      "number" : "212 5555-1234"
21    },
22    {
23      "type"   : "office",
24      "number" : "646 555-4567"
25    }
26  ],
27
28  "children" : [],
29  "spouse"   : null
30 }
```



./image/value2.png

► Codebeispiel: String und Werte ▾

```

1  //-----
2  // Aufloesen von Strings und Werten
3  //-----
4  var person = {
```

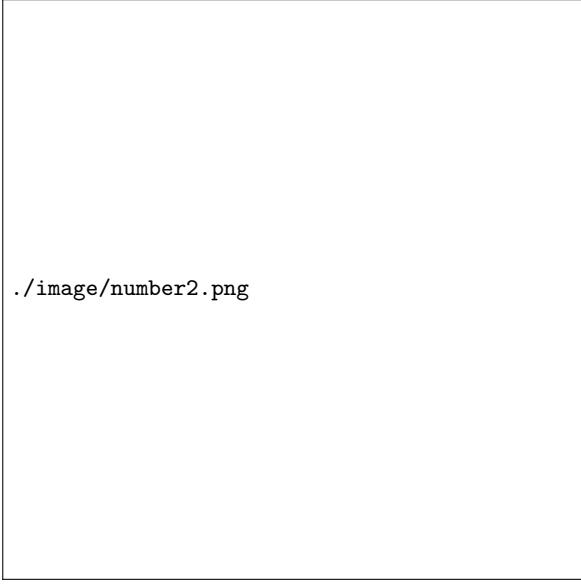
16.4.4 JSON - Number

Um die Menge der möglichen Werte auflösen zu können müssen wir noch definieren in welcher Form Numbers angegeben werden können.

```
29     "spouse" : null
30 }
```



► Erklärung: Number ▾

./image/number2.png

► Codebeispiel: Number ▾

```
1 //-----
2 // Aufloesen von Zahlenwerten
3 //-----
4 var person = {
5     "firstName": "John",
6     "lastName" : "Smith",
7     "isAlive"  : "true",
8     "age"      : 25,
9     "height_cm": 176.5,
10
11    "address" : {
12        "street"     : "21 2nd Street",
13        "city"       : "New York",
14        "postalCode" : "10021-2100"
15    },
16
17    "phoneNumbers" : [
18        {
19            "type"   : "home",
20            "number" : "212 5555-1234"
21        },
22        {
23            "type"   : "office",
24            "number" : "646 555-4567"
25        }
26    ],
27
28    "children" : []
},
```

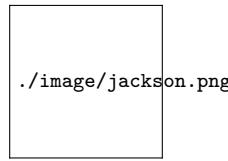
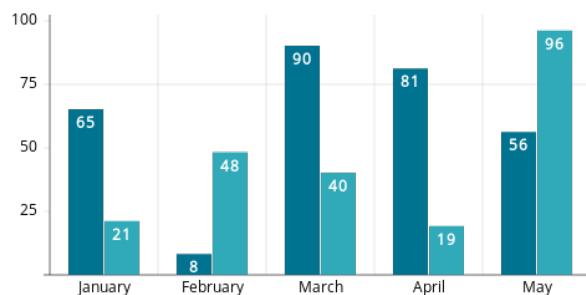


Abbildung 32. Jackson - Serialisierung, Deserialisierung

16.4.5 Jackson

Jackson ▾

Jackson ist ein hoch effizienter JSON Prozessor für Java.



```

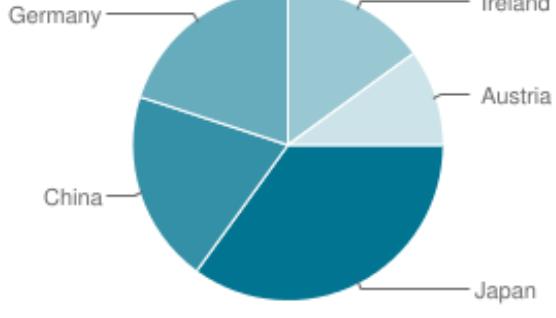
13 //-----
14 // Data
15 public class Project{
16
17     @JsonProperty("projectTitle")
18     @NotBlank
19     @Size(max=100)
20     private String titel;
21
22     @JsonProperty("code")
23     @NotBlank
24     @Size(min=12, max=12)
25     private String code;
26
27     @JsonProperty("description")
28     @NotBlank
29     @Size(max=4000)
30     private String description;
31
32 }
```

► Erklärung: Jackson ▾

- JACKSON IST VERANTWORTLICH FÜR DIE **Serialisierung** VON **Java Objekten** IN **Jason Dokumente** UND FÜR DIE **Deserialisierung** VON JSON DOKUMENTEN IN JAVA OBJEKTE.
- JACKSON VERWENDET **Annotationen** UM EIN **Mapping** ZWISCHEN **POJO** UND **Json Dokumenten** ZU DEFINIEREN.
- DABEI SETZT **Jackson** AUF **Defaulteinstellungen** UM DIE ZAHL DER ANNOTATIONEN GERING ZU HALTEN.

► Erklärung: **@JsonProperty** ▾

- MIT DER **@JsonProperty** ANNOTATION WIRD EIN MAPPING ZWISCHEN DEM ATTRIBUT EINES **Java Objekts** UND EINEM FELD IN EINEM **Json Dokument** ETABLIERT.
- WIRD DIE **@JsonProperty** ANNOTATION NICHT ANGEGEBEN MUSS DAS FELD IM JSON DOKUMENT UND DAS ATTRIBUT IM JAVA OBJEKT DENSELBEN NAMEN HABEN.



► Codebeispiel: **@JsonProperty** ▾

```

1 //-----
2 // Jason Dokument
3 //-----
4 {
5     "projectTitle": "Simulation Finiter Elemente",
6     "code": "SJD-543-4432",
7     "description": "Project to develop new ..."
8 }
9
10 //-----
11 // Java Objekt
12 }
```

16.4.6 Vererbung

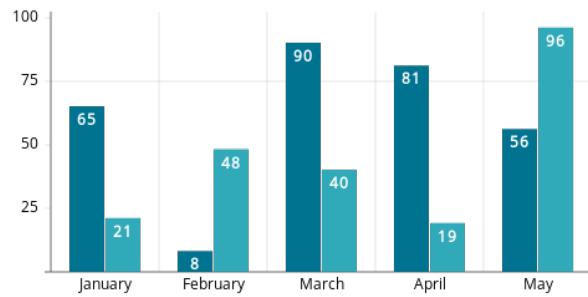
Für Objekte die Teil einer **Klassenhierarchie** sind muß zusätzlich Information über die **Vererbungsrelation** in Json codiert werden.



▶ Codebeispiel: `@JsonProperty`

```

1 //-----
2 // Java Objekte
3 //-----
4 @JsonTypeInfo(
5     use = JsonTypeInfo.Id.NAME,
6     include = JsonTypeInfo.As.PROPERTY,
7     property = "type"
8 )
9 @JsonSubTypes({
10     @JsonSubTypes.Type(value =
11         RequestFundingProject.class, name =
12         "request"),
13     @JsonSubTypes.Type(value =
14         ResearchFundingProject.class, name =
15         "research")
16 })
17 @Data
18 public abstract class AProject{
19
20     @JsonProperty("projectTitle")
21     @NotBlank
22     @Size(max=100)
23     private String titel;
24
25     @JsonProperty("code")
26     @NotBlank
27     @Size(min=12, max=12)
28     private String code;
29
30     @JsonProperty("description")
31     @NotBlank
32     @Size(max=4000)
33     private String description;
34
35 }
36
37 @JsonTypeName("request")
38 @Data
39 public RequestFundingProject extends AProject{
40
41     @JsonProperty("projectPartner")
42     @NotBlank
43     @Size(max=50)
44     private String partnerName;
45
46 }
47
48 @JsonTypeName("research")
49     @Data
50     public ResearchFundingProject extends AProject{
51
52         @NotNull
53         @Min(0)
54         private Integer fundingAmount;
55
56 //-----
57 // Json Dokumente
58 //-----
59 // RequestFundingProject
60 {
61     "type":"request"
62     "projectTitle":"Simulation Finiter Elemente",
63     "code":"SJD-543-4432",
64     "description":"Project to develop new ...",
65     "projectPartner":"TU Wien"
66 }
67 // ResearchFundingProject
68 //-----
69 // Jason Dokument
70 //-----
71 {
72     "type":"research"
73     "projectTitle":"Projektverwaltung",
74     "code":"SJD-543-4432",
75     "description":"Project to develop new ..."
76     "amount":2000
77 }
```



16.4.7 Bidirektionale Relationen mappen

Um **zirkuläre Abhängigkeiten** in Json Dokumenten zu vermeiden müssen **bidirektionale Relationen** in Entitäten zusätzlich gemapped werden.

▶ Codebeispiel: Bidirektionale Abhängigkeiten

```

1 //-----
2 // Java Objekte
3 //-----
4 @JsonTypeInfo(
5     use = JsonTypeInfo.Id.NAME,
6     include = JsonTypeInfo.As.PROPERTY,
```

```

7     property = "type"
8   )
9 @JsonSubTypes({
10   @JsonSubTypes.Type(value =
11     RequestFundingProject.class, name =
12     "request"),
13   @JsonSubTypes.Type(value =
14     ResearchFundingProject.class, name =
15     "research")
16 })
17 @Data
18 public abstract class AProject implements
19   Serializable{
20
21   @JsonManagedReference
22   private Set<Subproject> subprojects = new
23     HashSet<>();
24
25   @JsonProperty("projectTitle")
26   @NotBlank
27   @Size(max=100)
28   private String titel;
29
30   @Data
31   public class Subproject implements Serializable{
32
33     @JsonBackReference
34     private AProject project;
35
36     @JsonProperty("title")
37     @NotBlank
38     private String title;
39
40     @JsonProperty("description")
41     private String description;
42
43   }
44
45 //-----
46 //  Jason Dokument
47 //-----
48 {
49   "subprojects": [
50     {
51       "title": "FFG Institut",
52       "description": "Das Subprojekt ..."
53     },
54     {
55       "title": "FWF Institut",
56       "description": "Das Subprojekt ..."
57     }
58   ],
59   "type": "research",
60   "projectTitle": "Projektverwaltung",
61   "description": "Project to develop new ...",
62   "amount": 2000
63 }
```



16.4.8 Annotationen für Datumsformate

Für die Transformation von **temporale Daten** in das gewünschte Format wird eine eigene Annotation zur Verfügung gestellt.

► Codebeispiel: Datumsformate ▾

```

1 //-----
2 // Java Objekte
3 //-----
4 @Data
5 public class Project implements Serializable{
6
7     @JsonProperty("projectStart")
8     @JsonFormat(
9         shape = JsonFormat.Shape.STRING,
10        pattern = "dd.MM.yyyy"
11    )
12    private Date projectBegin;
13
14    @JsonProperty("projectEnd")
15    @JsonFormat(
16        shape = JsonFormat.Shape.STRING,
17        pattern = "dd.MM.yyyy"
18    )
19    private Date projectEnd;
20
21    @JsonProperty("projectTitle")
22    @NotBlank
23    @Size(max=100)
24    private String titel;
25
26    @JsonProperty("description")
27    @NotBlank
28    @Size(max=4000)
29    private String description;
30
31 }
32
33 //-----
34 // Unitest
35 //-----
36 @Test
37 public void testCreateProject(){
38     Project project = new Project();
39
40     project.setTitle();
41     project.setDescription();
42
43     SimpleDateFormat df = new
44         SimpleDateFormat("dd.MM.yyyy");
45     df.setTimeZone(TimeZone.getTimeZone("UTC"));
46
47     project.setProjectStart(df.parse("21.11.2004"));
48     project.setProjectEnd(df.parse("03.05.2020"));
49
50     projectRepository.create(project);
51 }
```

16.4.9

Index

- Architekturmuster, 26
- Ausfallsicherheit, 48
- Bigdata, 50
 - Variety, 50
 - Velocity, 50
 - Volume, 50
- Dezentrale Systeme, 48
- Entwurf Verteilte Systeme, 65
- Entwurfsmuster, 26
 - Erzeugungsmuster, 27
 - Verhaltensmuster, 34
- Erzeugungsmuster, 27
 - Singleton, 27
- Idiom, 26
- Interaktionskoppelung, 21
- Interoperabilität, 51
- Klassen, 15
- Kohäsion, 25
 - Klassenkohäsion, 25
 - Servicekohäsion, 25
- Kommunikationsverbund, 48
- Komponenten, 17
- Kompositum, 32
- Koppelung, 21
 - Interaktionskoppelung, 21
 - Vererbungskoppelung, 23
- Lastenverbund, 48
- Liskovsche Substitutionsprinzip, 41
- Microservice, 6, 10
 - Anwendung, 10
 - Architektur, 10
- Modularisierung, 7, 48
- Nebenläufigkeitstransparenz, 49
- Offenheit, 51
 - Interoperabilität, 51
 - Portabilität, 51
- Open Closed Prinzip, 39
- Ortstransparenz, 49
- parallele Verarbeitung, 7, 48
- Portabilität, 51
- Programmierparadigmen
 - Codestruktur, 14
- Prozess, 14
- Replikation, 7, 48
- Replikationstransparenz, 49
- Schichtenmodell, 15
- Shared Access, 7, 48
- Single Responsibility Prinzip, 38
- Singleton, 27
- Skalierbarkeit, 50
- Softwaremetrik
 - Koppelung, 21
- Softwaremetriken, 20
- SOLID Prinzipien, 37
 - Dependency Injection, 37
 - Interface Segregation Prinzip, 37
 - Open Closed Prinzip, 37
 - Single Responsibility Prinzip, 37
 - Subsitions Prinzip, 37
- Strukturierung, 14
 - Anweisung, 14
 - Code, 14, 90
 - Klasse, 15
 - Klassen, 15
 - Komponenten, 17
 - Schichten, 15
 - Unterprogramm, 14
- Strukturmuster
 - Kompositum, 32
- Transaktionsmodell, 7, 48
- Transparenz, 49
 - Nebenläufigkeitstransparenz, 49
 - Ortstransparenz, 49
 - Replikationstransparenz, 49
 - Zugriffstransparenz, 49
- Variety, 50
- Velocity, 50
- Vererbungskoppelung, 23
- Verhaltensmuster, 34
 - Strategie, 34
- Verteilte Systeme, 47
 - Modularisierung, 7, 48
 - parallele Verarbeitung, 7, 48
 - Replikation, 7, 48
 - Shared Access, 7, 48
 - Transaktionsmodell, 7, 48
 - Transparenz, 49
 - Volume, 50

Zeilenmetrik, 20
Zugriffstransparenz, 49