Architektur verteilter Anwendungen

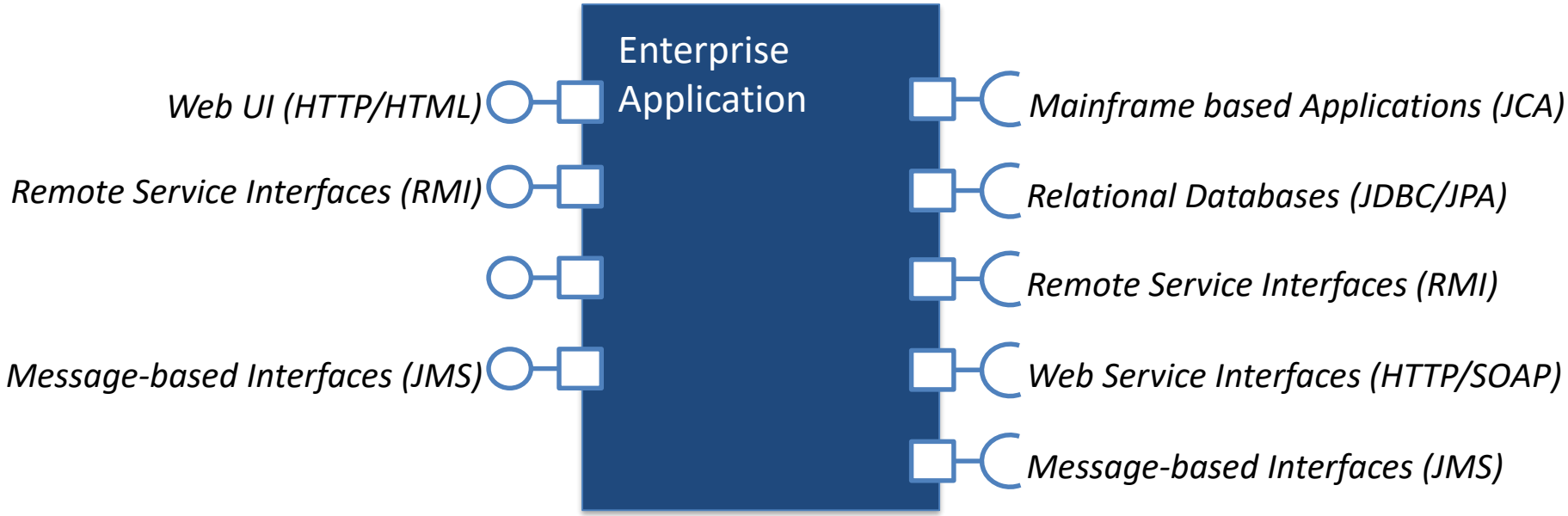
FWP Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Wer braucht schon einen Architekten?

ARCHITEKTUR VERTEILTER ANWENDUNGEN

Niemand ist eine Insel

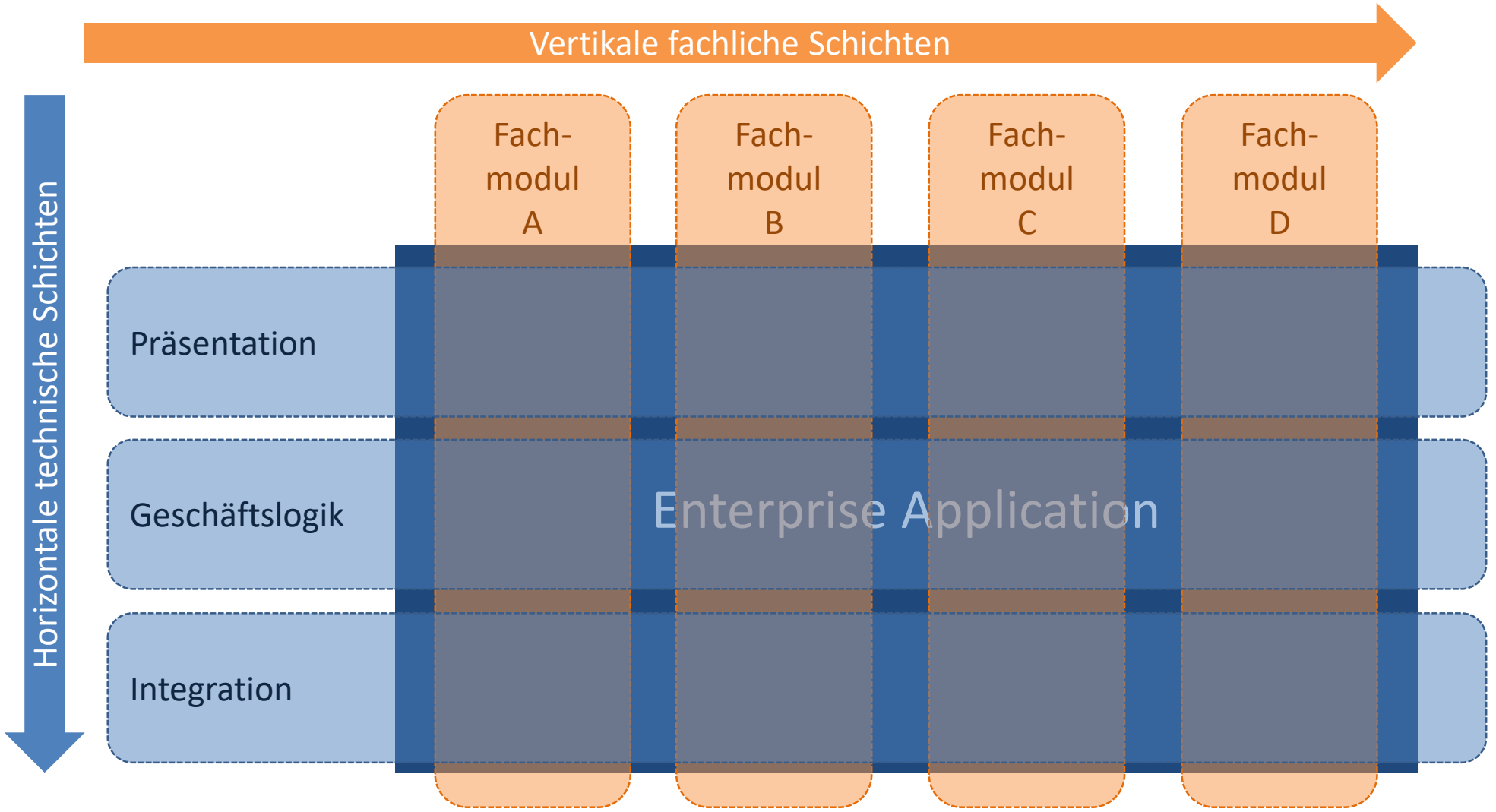
- Applikationen benötigen andere Systeme (Consumer)
- Applikationen unterstützen andere Systeme (Provider)



Wie wollen Sie diese Komplexität ohne Architektur bewältigen?

Schichtenmodell als Basis

Einheitliches Schichtenmodell



Merkmale der Schichten

- Wohldefinierte Schnittstellen zwischen den Schichten
- Zyklenfreie, gerichtete Abhängigkeiten zwischen den Schichten
- Getrennte Verantwortlichkeiten
- Lose Kopplung / Hohe Kohäsion
- Verteilung der Schichten auf verschiedene Lokationen möglich
- Jede Schicht hat eigenen Namensraum (in Java: Packages)

Drei prinzipielle Schichten

Präsentation
(Presentation)*

- Interaktion zwischen Anwendung und Benutzer
- Anzeige und Bearbeitung von Informationen

Geschäftslogik
(Business, Domain)*

- Kern der Anwendungen bestehend aus Diensten in einer Serviceschicht *(Service Layer)* und Domänenmodell *(Domain Model)*

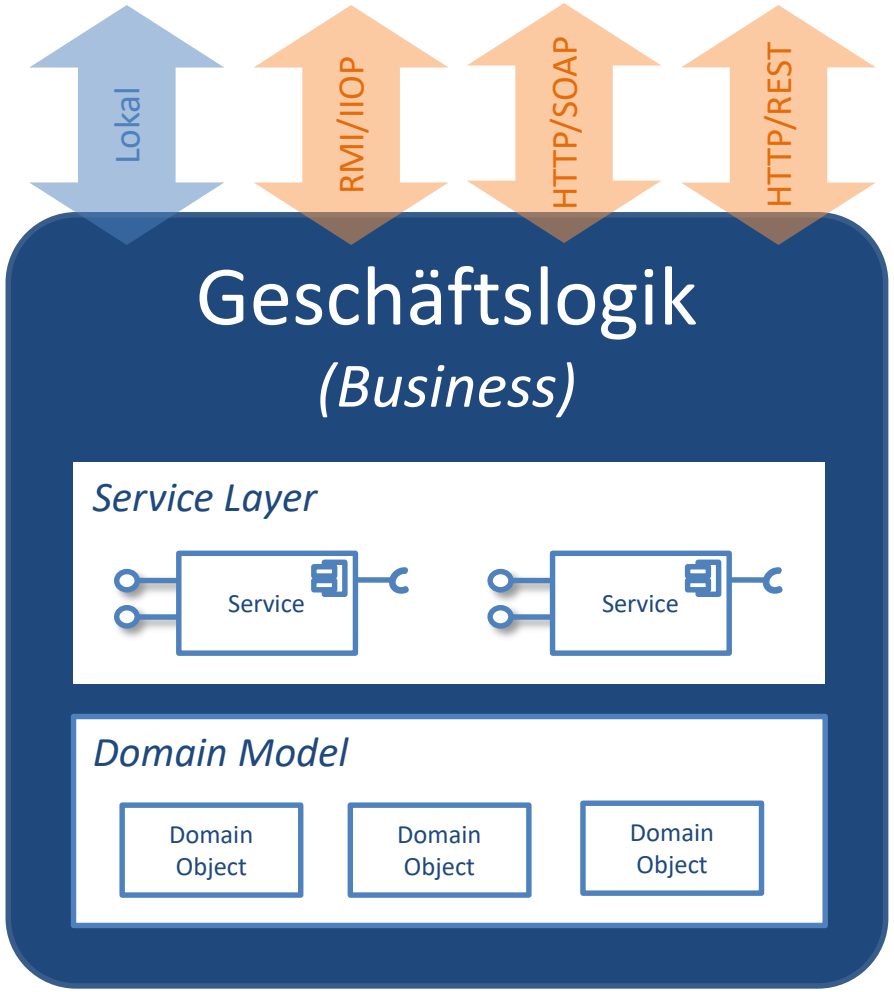
Integration
(Integration, Data Source)*

- Integration externer Ressourcen
- Transformation externes Domänenmodell / internes Domänenmodell

Präsentationsschicht

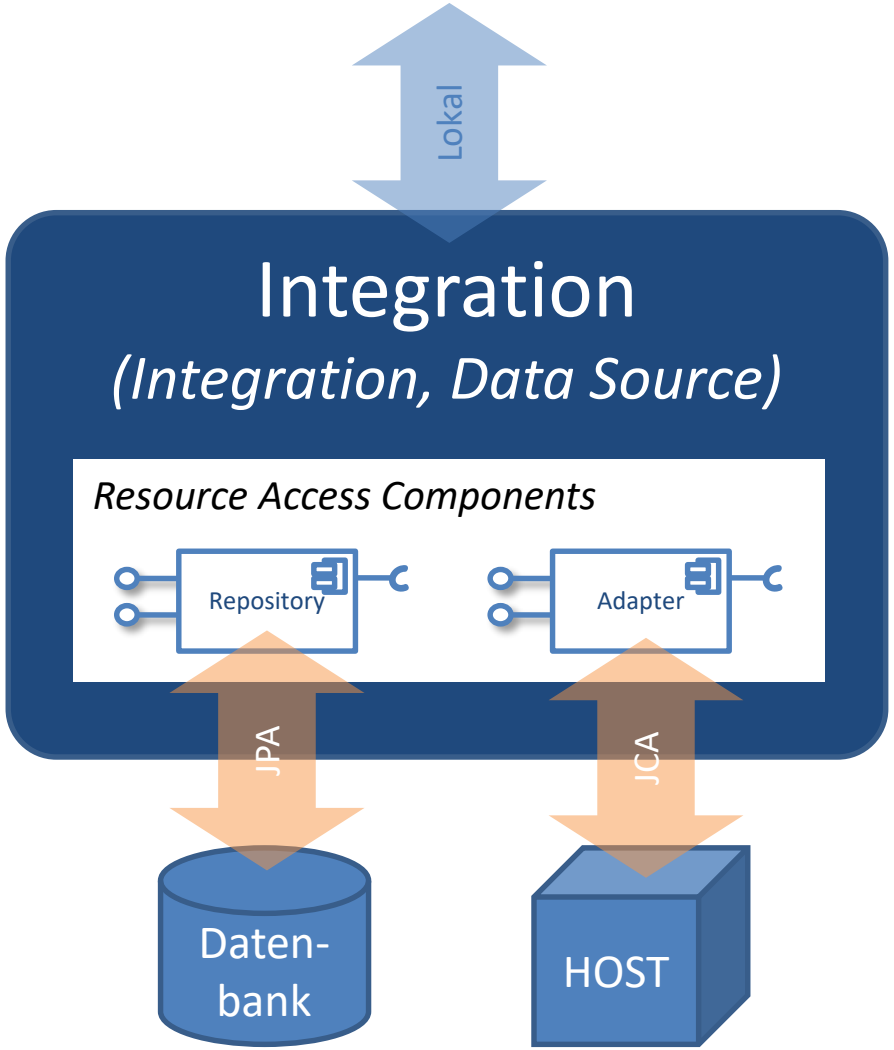
- Benutzeroberfläche für menschliche Benutzer
- Beinhaltet nur Präsentationslogik
- Höchste Änderungshäufigkeit innerhalb der Anwendung
- Rich Client oder Thin Client

Geschäftslogikschicht



- Kern der Anwendung
- Services kapseln Geschäftslogik
  - Lokal: Präsentationsschicht der eigenen Anwendung
  - Entfernt: Andere Anwendungen
- Services operieren auf einheitlichem Domänenmodell
- Nutzt Integrationsschicht für Integration von externen Ressourcen

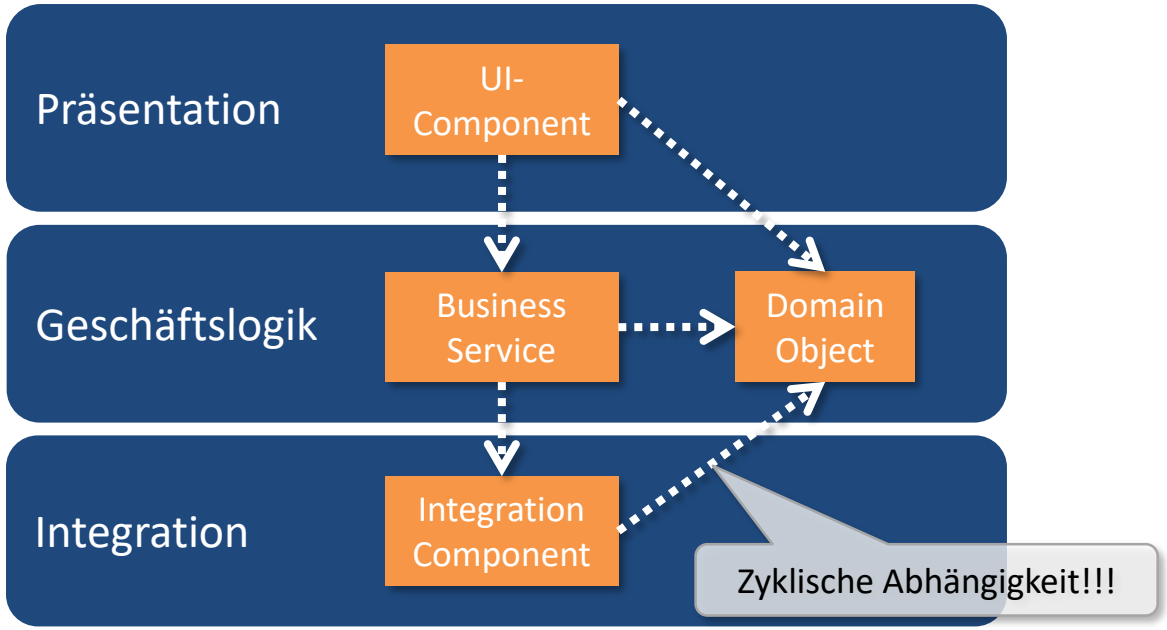
Integrationsschicht



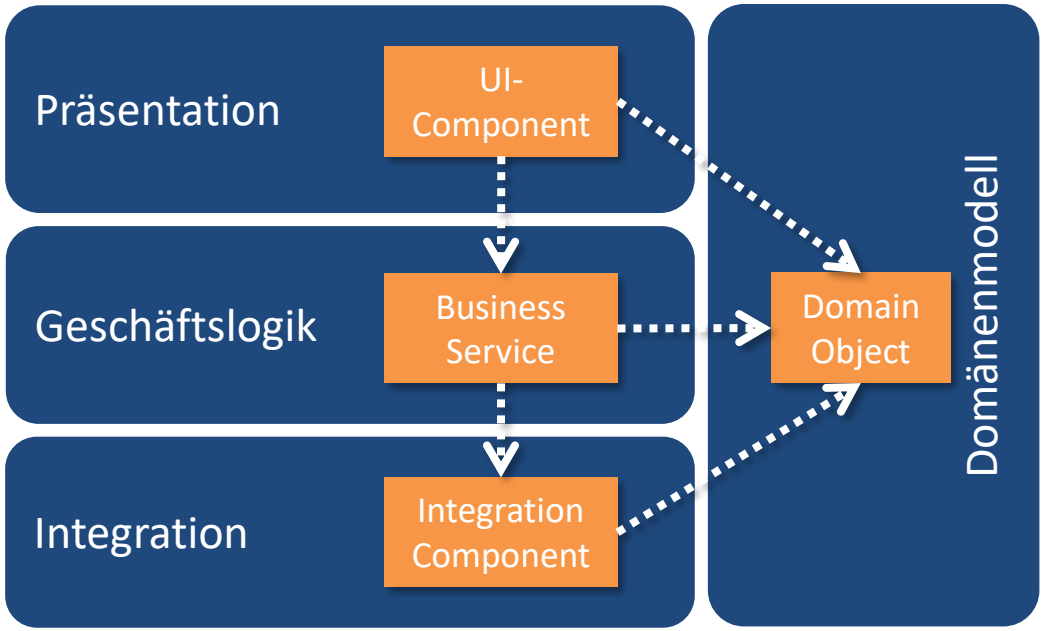
- Integriert externe Ressourcen
- Bildet externes Domänenmodell auf internes Domänenmodell ab
- Kapselt Information über konkrete Integration
  - Welche Datenbank?
  - Welches Kommunikationsprotokoll?
  - Welches externe System?

Problem: Zyklische Abhängigkeiten

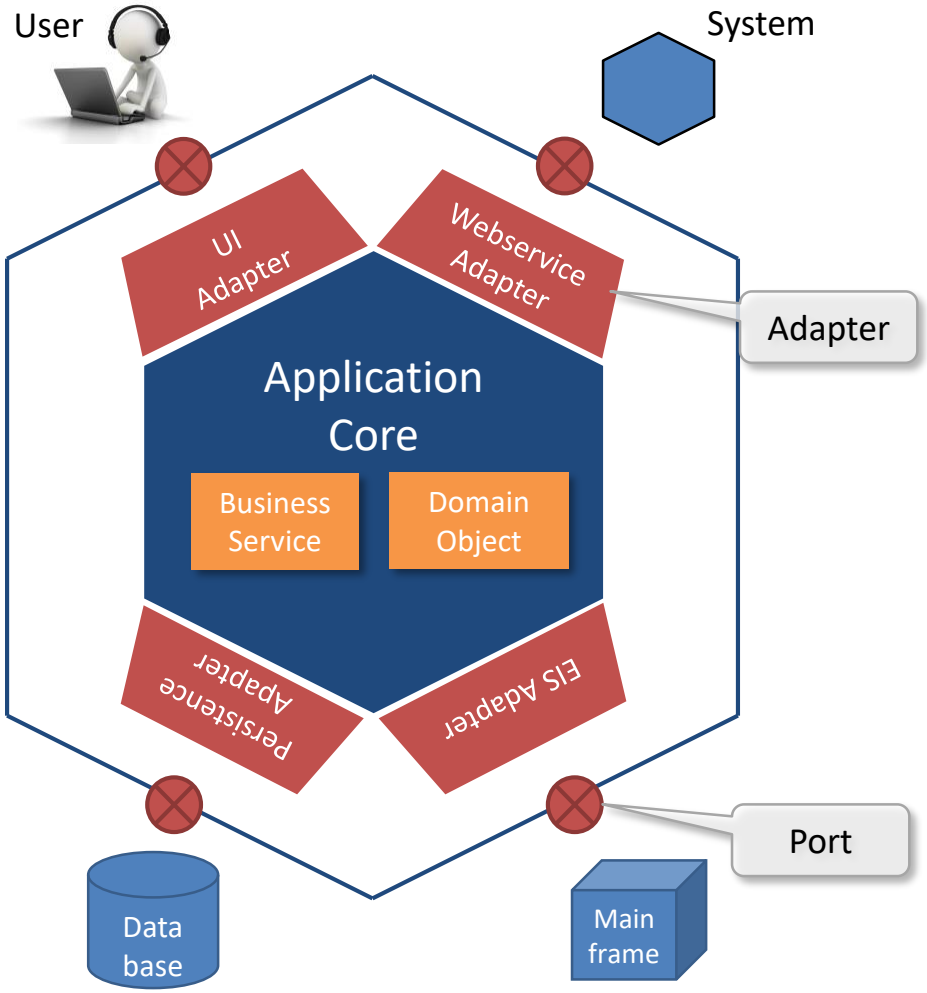
Schichtenmodell mit 3 Schichten



Bereinigtes Schichtenmodell



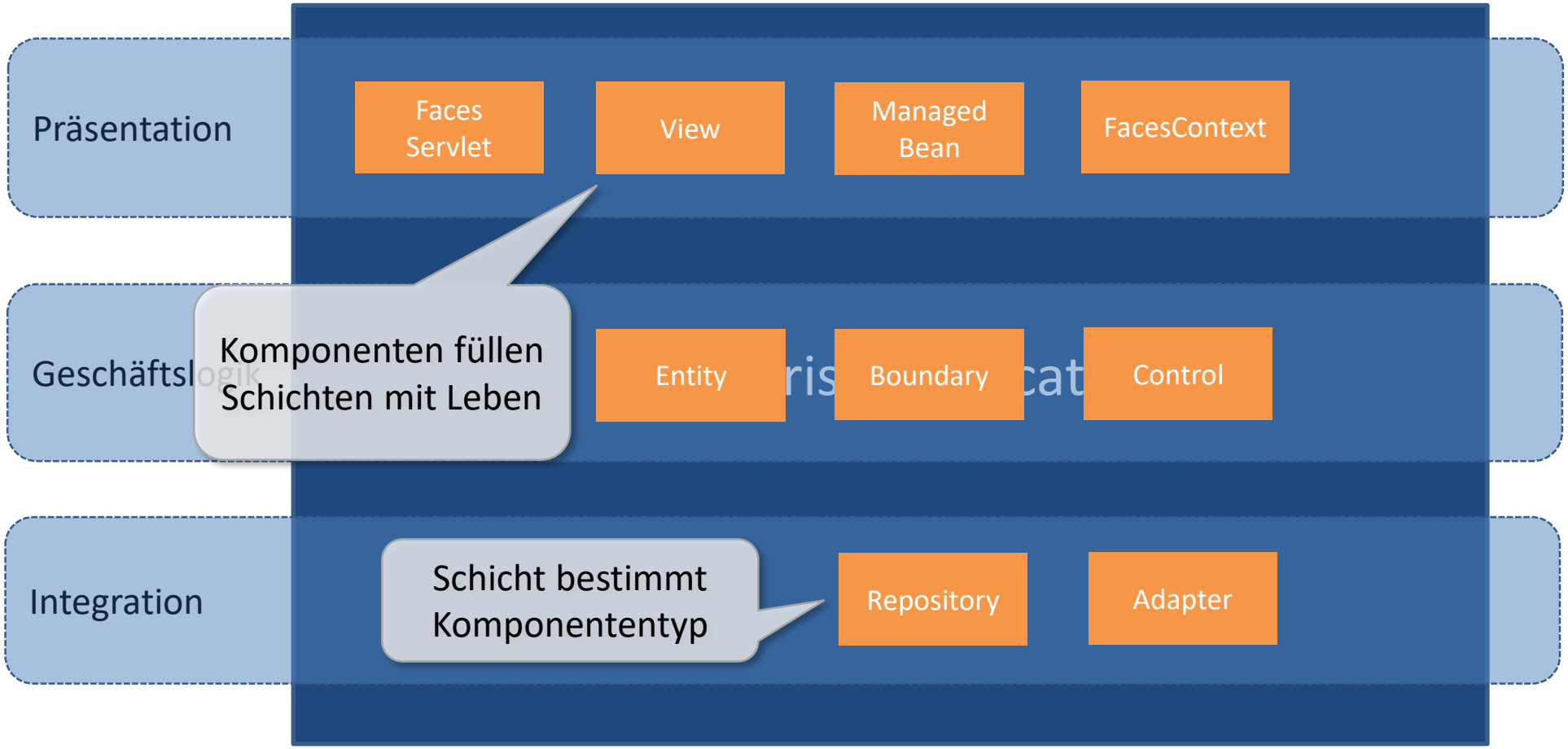
Alternative: Hexagonale Architektur



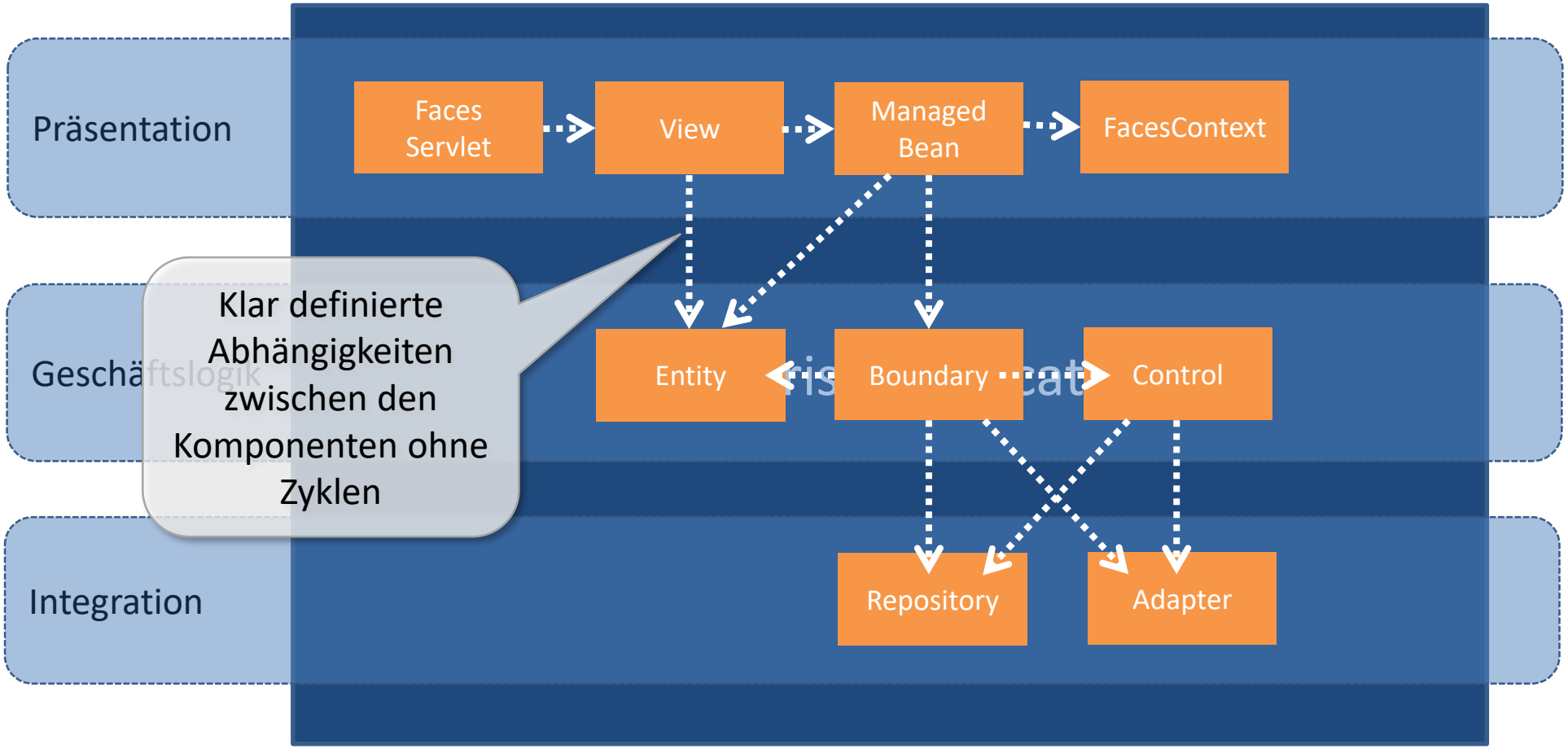
- Auch: Ports & Adapter
- Außenwelt kommuniziert mit Applikation über Ports
- Applikation kommuniziert mit Außenwelt über Ports
- Technologiespezifische Adapter hinter Ports übersetzen Kommunikation
- Applikation kennt keine Details der Außenwelt
- Außenwelt ist simulierbar

Aufgebaut aus Komponenten

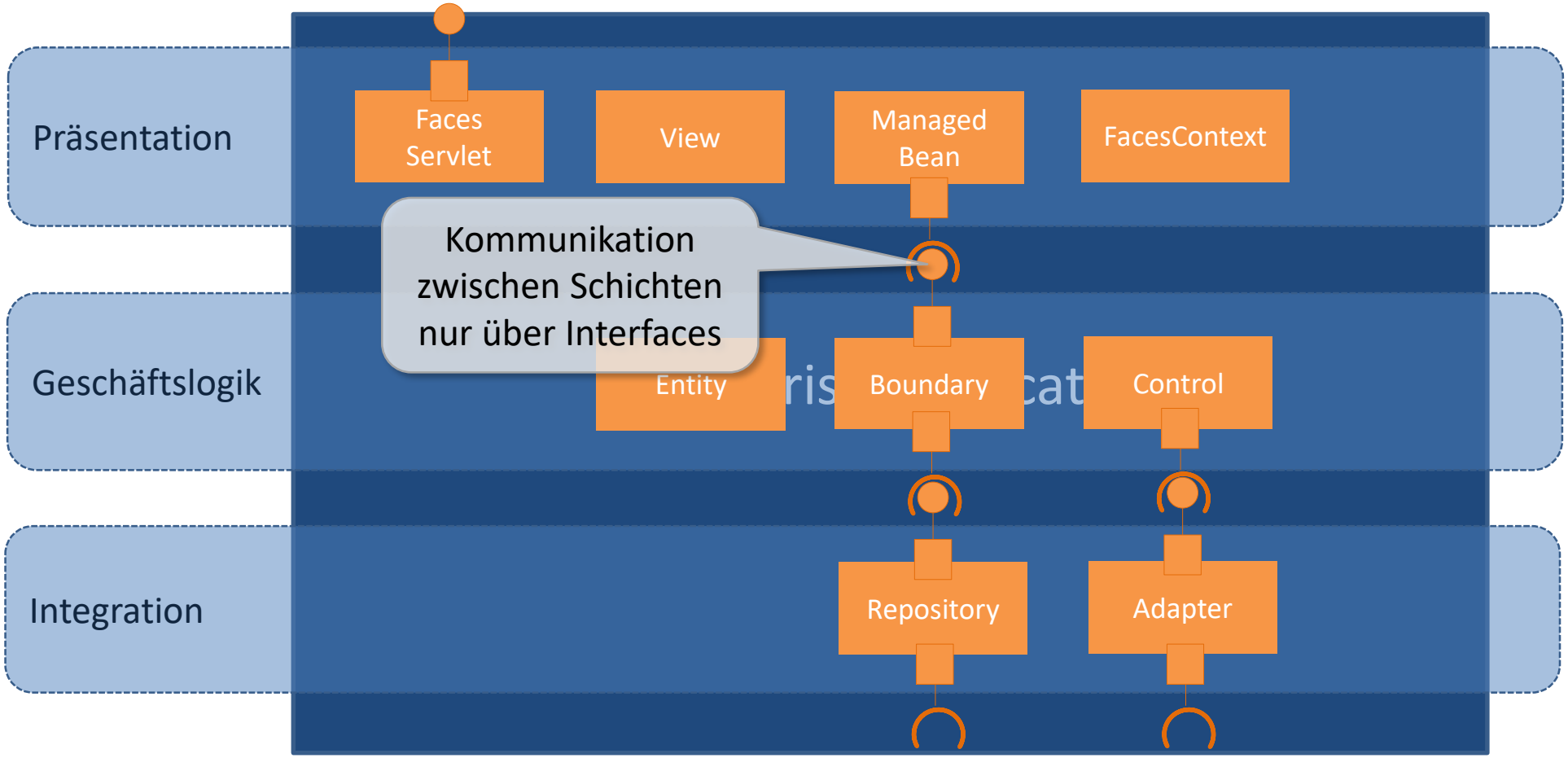
Einheitliches Komponentenmodell



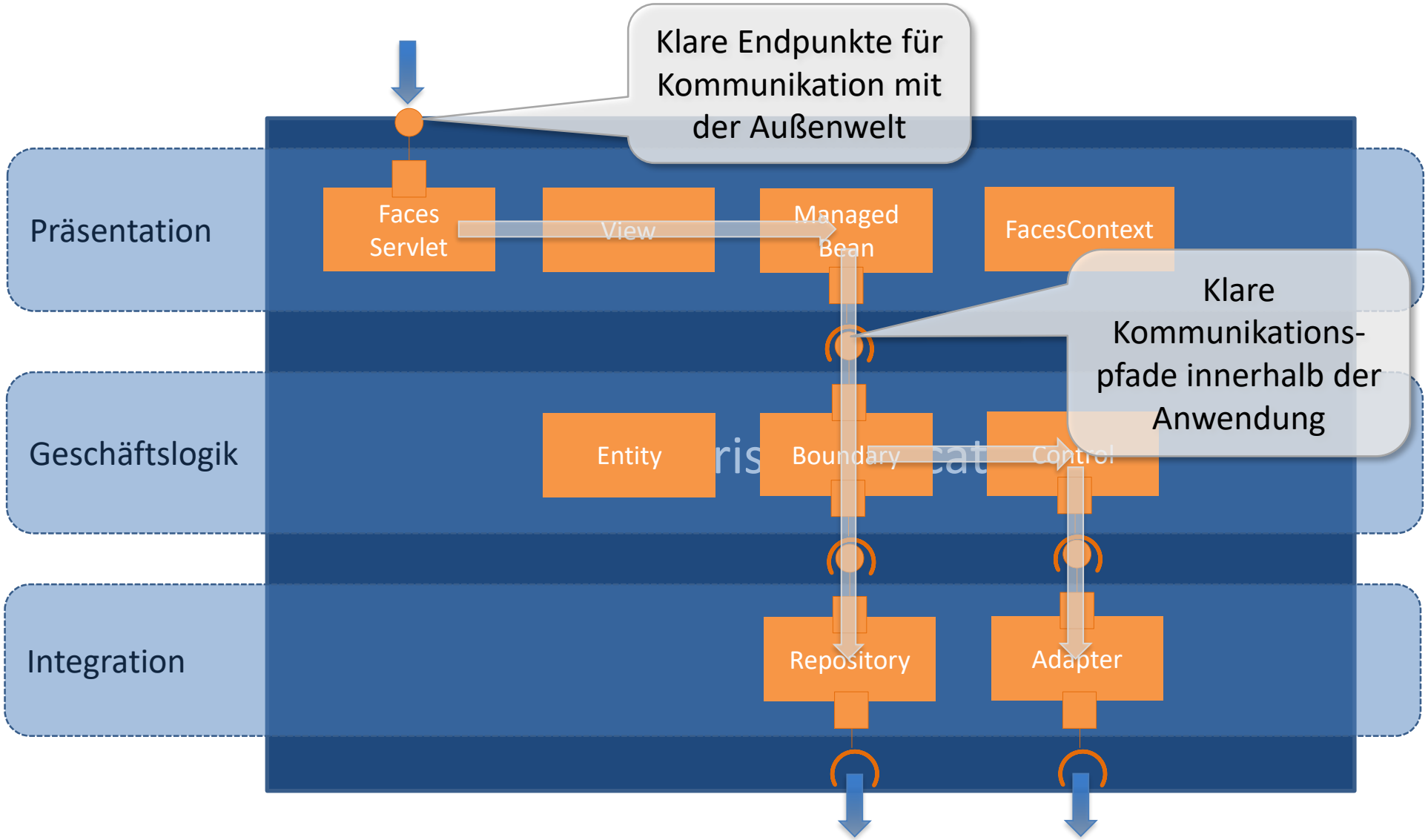
Einheitliches Komponentenmodell



Einheitliches Komponentenmodell



Einheitliches Komponentenmodell



Motivation und Umsetzung

- Ermöglicht Entkopplung, Kapselung und klare Trennung der Verantwortlichkeiten
- Erleichtert die Erstellung service-orientierter Applikationen
- Schafft gemeinsame Sprache zur Kommunikation des Designs
- Leicht umzusetzen
  - 1 Implementierungsklasse pro Komponente
  - 1 Interface pro Komponente (optional, aber empfehlenswert)
  - Gruppierung in Module/Packages gemäß Konvention

Mögliche Designmodelle als Basis

- Entscheidung für Designmodell zu Projektbeginn
- Wesentliche Modelle:
  - Serviceorientiert (Service Oriented Architecture SOA)
  - Domänengetrieben (Domain Driven Design DDD)
- Modelle können kombiniert werden, aber ein Modell sollte Hauptmodell sein

Gegenüberstellung SOA - DDD

Service Oriented Architecture	Domain Driven Design
● Services enthalten alle Logik ● Domänenobjekte sind dumme Datenträger ● Optimierung für Verteilung bestimmt internes Design ● Fördert Anti-Pattern Anemic Domain Model * ● Widerspricht objekt-orientiertem Design	● Domänenobjekte enthalten Daten und Logik ● Services enthalten nur Logik zur Orchestrierung von Domänenobjekten ● Fachlichkeit einziger Treiber für internes Design ● Verteilter Zugriff über Adapter ● Entspricht objekt-orientiertem Design

* Martin Fowler „AnemicDomainModel“ http://www.martinfowler.com/bliki/AnemicDomainModel.html

GEMEINSAME PATTERNS UND PRINZIPIEN

Gemeinsame Patterns und Prinzipien

DEPENDENCY INJECTION

Inversion of Control (IoC)

- Don´t call us, we call you!
- Kontrolle wandert in das verwendete Framework

> "One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This ***inversion of control*** gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application."
>
> *Designing Reusable Classes* von Ralph E. Johnson und Brian Foote
> Journal of Object-Oriented Programming Juni/Juli 1988, S. 22-35ff
> http://www.laputan.org/drc/drc.html

Dependency Injection (DI)

- Präzisierung des Inversion of Control-Patterns
  u.a. von Martin Fowler
- Ein sog. Assemblierer löst die Abhängigkeiten zwischen Objekten auf
- Lose gekoppelte POJOs leben in einem Inversion of Control-Container
  - Container bestimmt Lebenszyklus der Objekte
  - Container löst als Assemblierer die Abhängigkeiten auf

Consumer benötigt Service

Implementierungsklasse des Service

```
public class ServiceImpl implements Service
    public void doSomething() {
        // Hier wird etwas gemacht
    }
}
```

Interface des Service

```
public interface Service {
    public void doSomething();
}
```

Consumer ist abhängig vom Service-Interface

```
public class Consumer {
    private Service service;
    ...
    public void useService() {
        this.service.doSomething();
    }
```

Wie kommt der Consumer an den Service?

Traditionell

- Consumer erzeugt Objekt von *ServiceImpl* und initialisiert damit Feld *service*:

```
public class Consumer {
  private Service service =
        new ServiceImpl();
```

☹ Consumer wird abhängig von konkreter Implementierung

Mit Dependency Injection

- Consumer markiert Feld *service* als Ziel für Dependency Injection:

```
public class Consumer {
  @Inject
  private Service service;
```

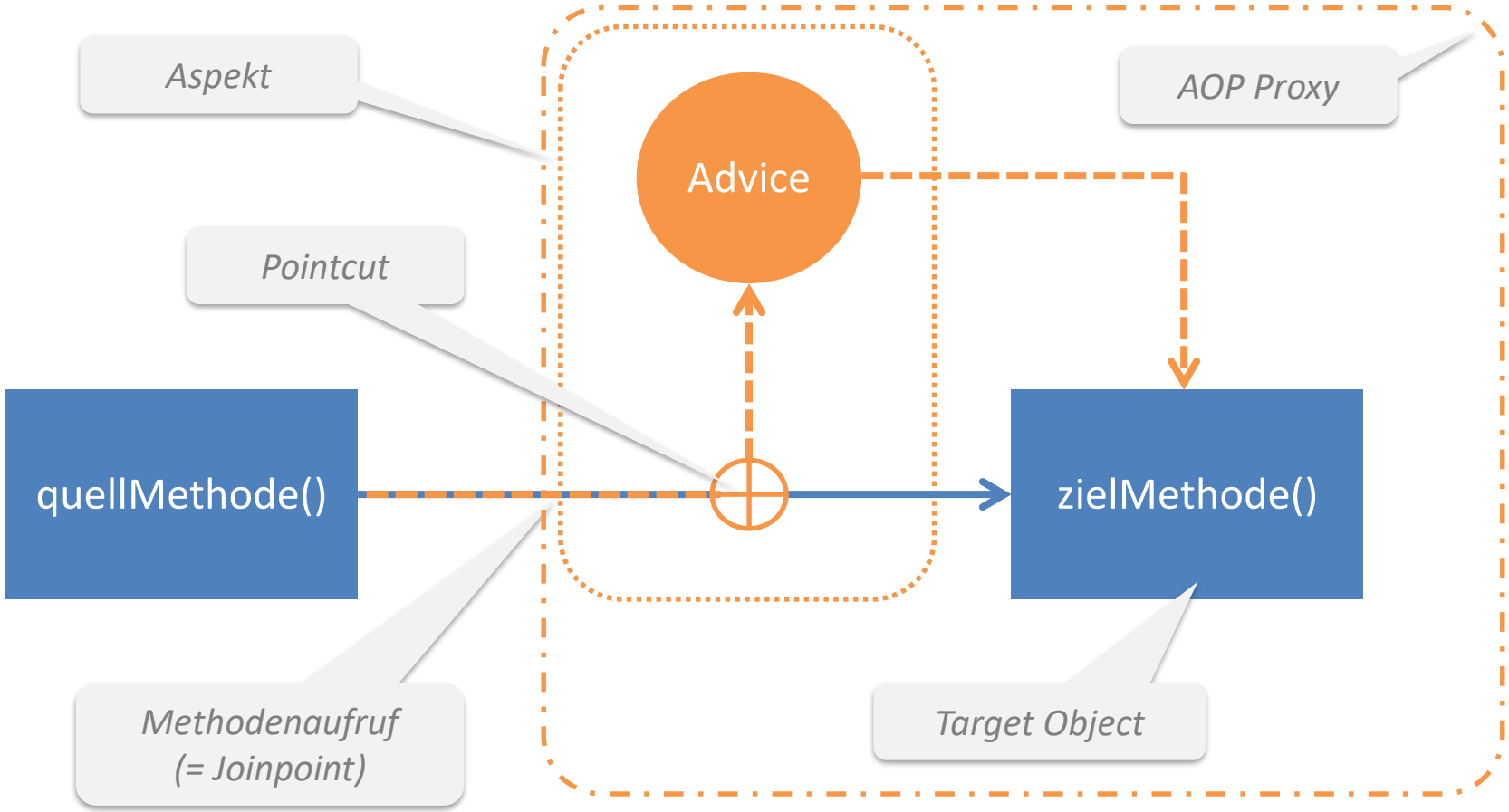
- Container erkennt annotiertes Feld, erzeugt ein Objekt von *ServiceImpl* und injiziert Referenz in Feld *service*

☺ Consumer kennt nicht die konkrete Implementierung

Gemeinsame Patterns und Prinzipien

ASPECT ORIENTED PROGRAMMING

Modularisierung von Cross Cutting Concerns



Beispiele für AOP in Java EE

- EJB-Container verwendet AOP für Transaktionsmanagement, Zugriffskontrolle und Remoting
- JAX-WS verwendet AOP für das Mapping von Methodenaufrufen mit Parametern auf SOAP-Nachrichten und umgekehrt
- JPA verwendet AOP für die Zustandsüberwachung von Entities

Gemeinsame Patterns und Prinzipien

CONVENTION OVER CONFIGURATION

Convention over Configuration (CoC)

- Einfaches Prinzip zur Erleichterung der Programmierung übernommen aus RUBY
- Standardmäßig verwendet die Laufzeitumgebung sinnvolle Voreinstellungen
- Nur im davon abweichenden Fall muss eine Konfiguration vorgenommen werden

Beispiele für CoC in Java EE

- Stateless Session Beans oder Managed Beans werden unter dem einfachen Klassennamen registriert, falls nichts anderes angegeben
- Methodenaufrufe eines Enterprise Java Beans laufen immer in einem transaktionalen Kontext (TRANSACTION_REQUIRED)
- Die Java Persistence Architecture (JPA) mappt automatisch Klassennamen von Entitäten auf Tabellennamen und Feldnamen auf Spaltennamen

ALLGEMEINE INFRASTRUKTUR-DIENSTE

Infrastrukturdienste für verteilte Apps

Verteilung

- Regelt Kommunikation verteilter Anwendung

Transaktionen

- Steuern Zugriff auf transaktionale Ressourcen
- Stellen Datenkonsistenz sicher

Persistenz

- Bietet die dauerhafte Speicherung von Daten
- Bildet Domänenmodell auf relationales Datenmodell ab

Sicherheit

- Schützt die Anwendung vor unberechtigten Zugriffen

Verteilung

Erstes Grundgesetz der Verteilung

“Don’t distribute your objects!”

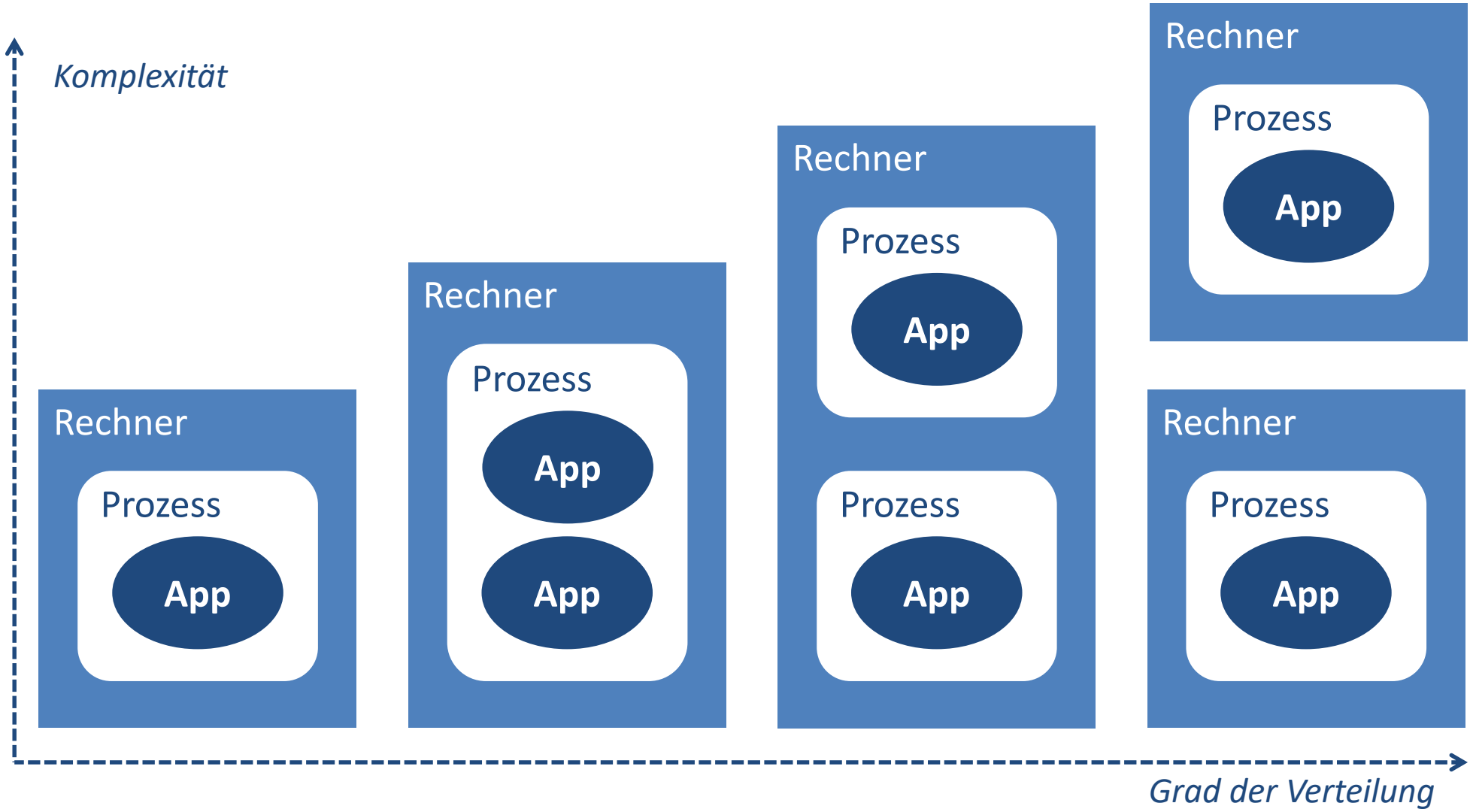
Quelle: Martin Fowler „Patterns of EAA“

Irrtümer der verteilten Verarbeitung

- Das Netzwerk ist zuverlässig
- Latenz[zeit] ist 0
- Bandbreite ist unendlich
- Das Netzwerk ist sicher
- [Netzwerk-]Topologie ändert sich nicht
- Es gibt nur einen Administrator
- Transportkosten sind 0
- Das Netzwerk ist homogen

Quelle: Peter Deutsch, James Gosling *Fallacies of Distributed Computing*
http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing

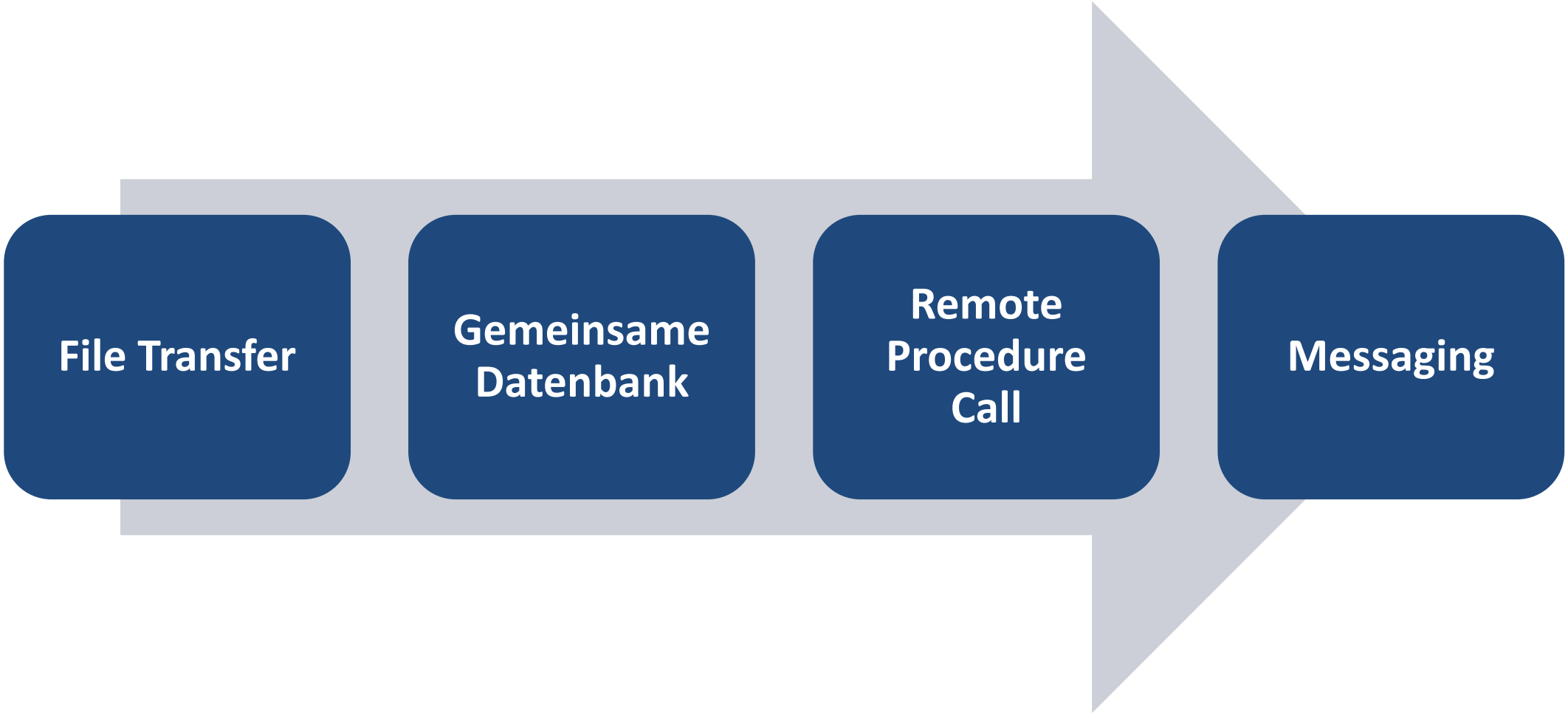
Verteilung erhöht Komplexität



Gründe für Verteilung

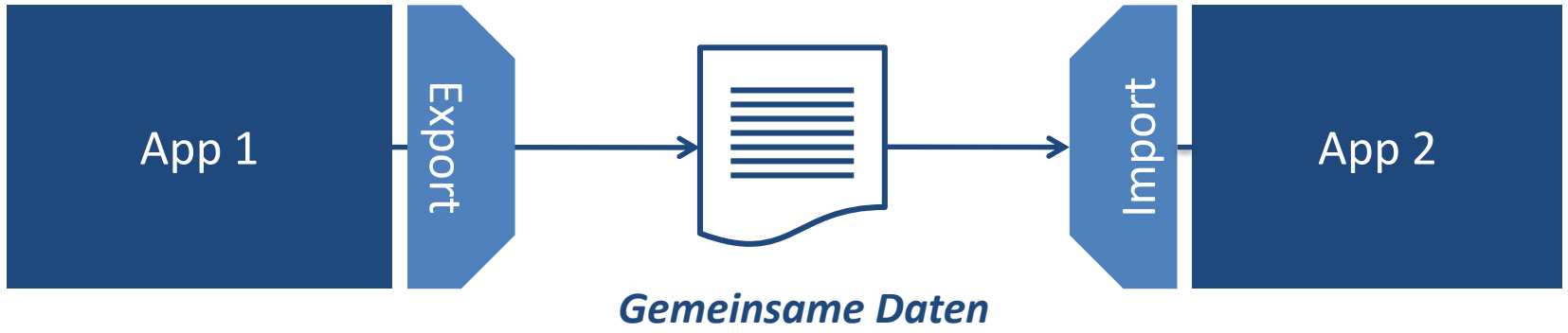
- Skalierbarkeit
- Ausfallsicherheit
- Technologiegrenzen
- Unterschiedliche Standorte
- Bereitstellung von Diensten für externe Consumer
- Komposition eigener Dienste aus Diensten externer Provider

Integrationsmuster verteilter Applikationen



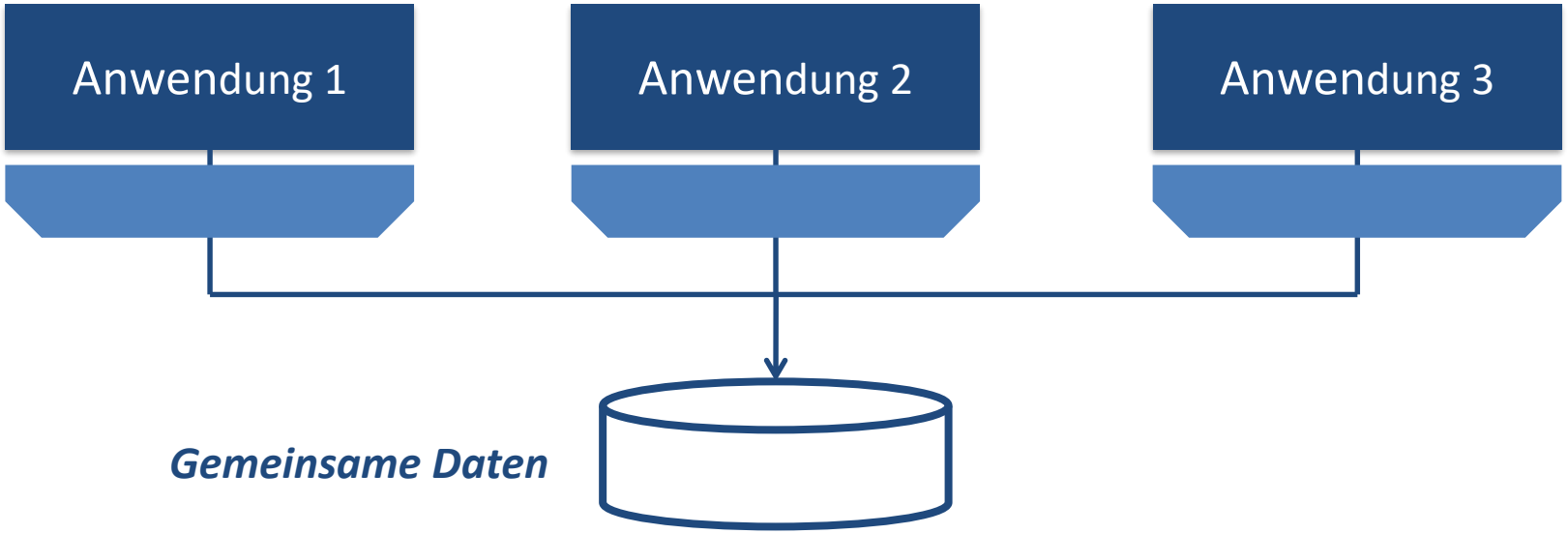
File Transfer

- Anwendungen tauschen Informationen über Dateien aus
- Anforderungen bestimmen Häufigkeit des Austausches



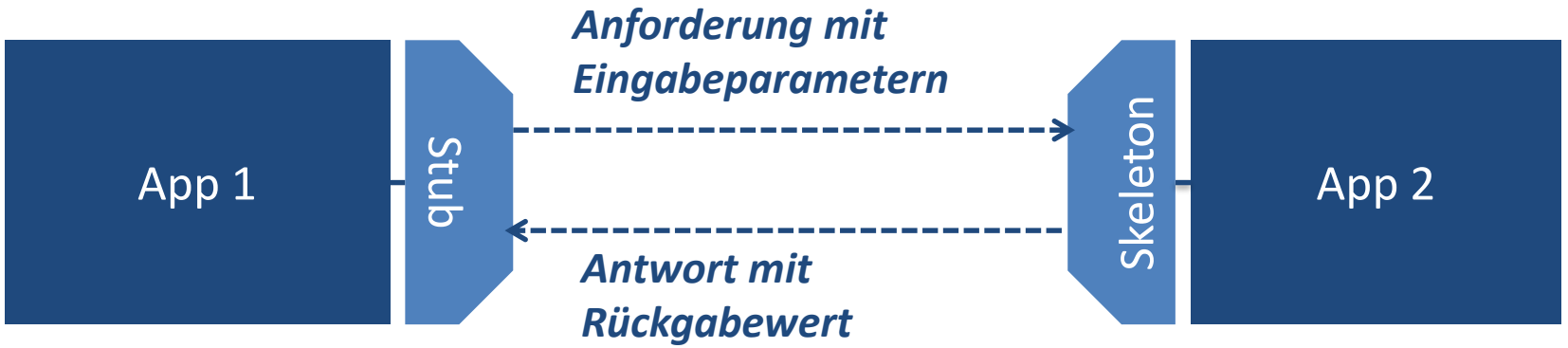
Gemeinsame Datenbank

- Anwendungen tauschen Informationen über gemeinsame Tabellen aus
- Datenbank-Schema bestimmt Datenformat



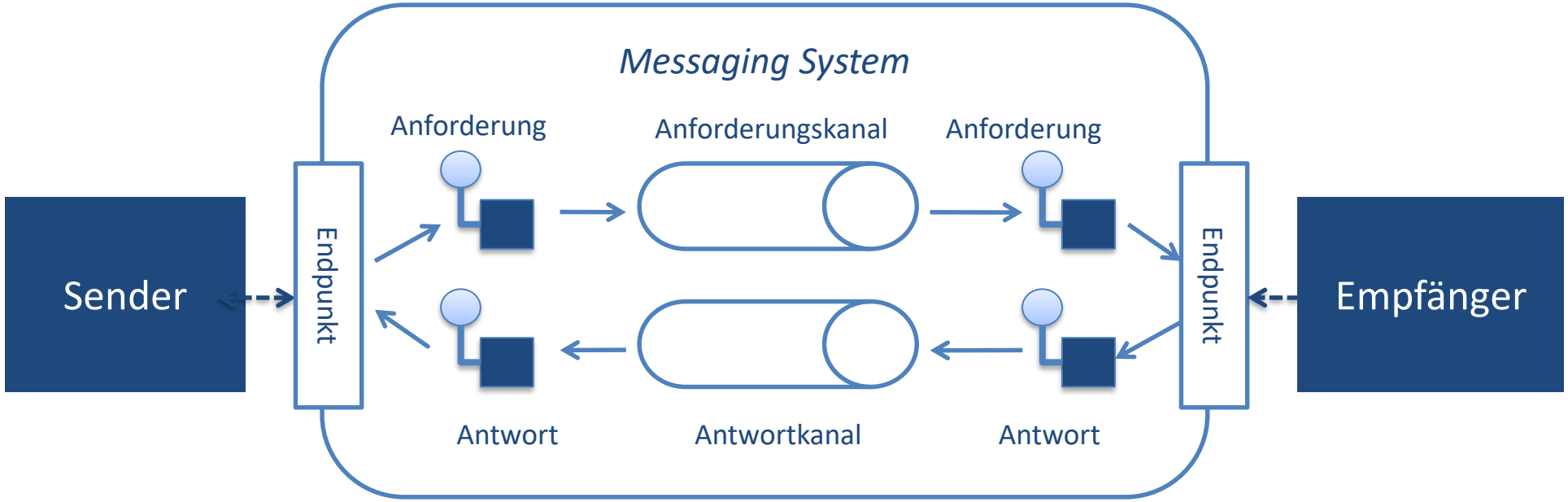
Remote Procedure Call

- Anwendungen kapseln ihre Daten und stellen Interfaces für den Zugriff zur Verfügung
- Austausch der Informationen erfolgt über synchronen entfernten Methodenaufruf



Messaging

- Häufiger, unmittelbarer, zuverlässiger und asynchroner Austausch von Nachrichten
- Flexibelste Integration, erfordert aber Umdenken



Transaktionen

ACID

Atomicity	• Eine Transaktion wird entweder ganz oder gar nicht ausgeführt
Consistency	• Eine Transaktion überführt eine transaktionale Ressource von einem bestehenden konsistenten Zustand in einen neuen konsistenten Zustand
Isolation	• Das Ergebnis einer Transaktion darf für andere Transaktionen solange nicht sichtbar sein, bis diese Transaktion erfolgreich abgeschlossen worden ist.
Durability	• Das Ergebnis einer erfolgreich abgeschlossenen Transaktion muss dauerhaft erhalten bleiben und Abstürze jeder Art überstehen.

Isolationsgrad bestimmt Durchsatz

Serializable

- Wiederholte Lesezugriffe liefern gleiche Sätze und gleiche Ergebnismengen
- Schreib-, Lese- und Bereichsoperationen verursachen Sperren

Repeatable reads

- Wiederholte Lesezugriffe liefern gleiche Sätze aber unterschiedliche Ergebnismengen möglich (Phantom Read)
- Schreib- und Leseoperationen verursachen Sperren

Read committed

- Lesezugriffe liefern nur committete Sätze
- Schreiboperationen verursachen Sperren

Read uncommitted

- Lesezugriffe liefern nicht committete Sätze
- Keine Sperren

Ablauf einer Transaktion

- Begin
  - Transaktion beginnt
- Commit
  - Transaktion endet erfolgreich
  - Beteiligte transaktionale Ressourcen werden verändert
- Rollback
  - Transaktion bricht nach Fehler ab
  - Beteiligte transaktionale Ressourcen bleiben unverändert

Propagation von Transaktionen

- Bei Aufrufen entfernter Anwendungen wird der Transaktionskontext mit übertragen
  - Aufgerufene Methode kann die die Transaktion des Aufrufers teilen (muss aber nicht)
- Koordination der Transaktionen übernimmt der aufrufende Container (Applikationsserver)
- Umsetzung teilweise aufwändig bzw. unmöglich

BASE für verteilte Datenspeicher

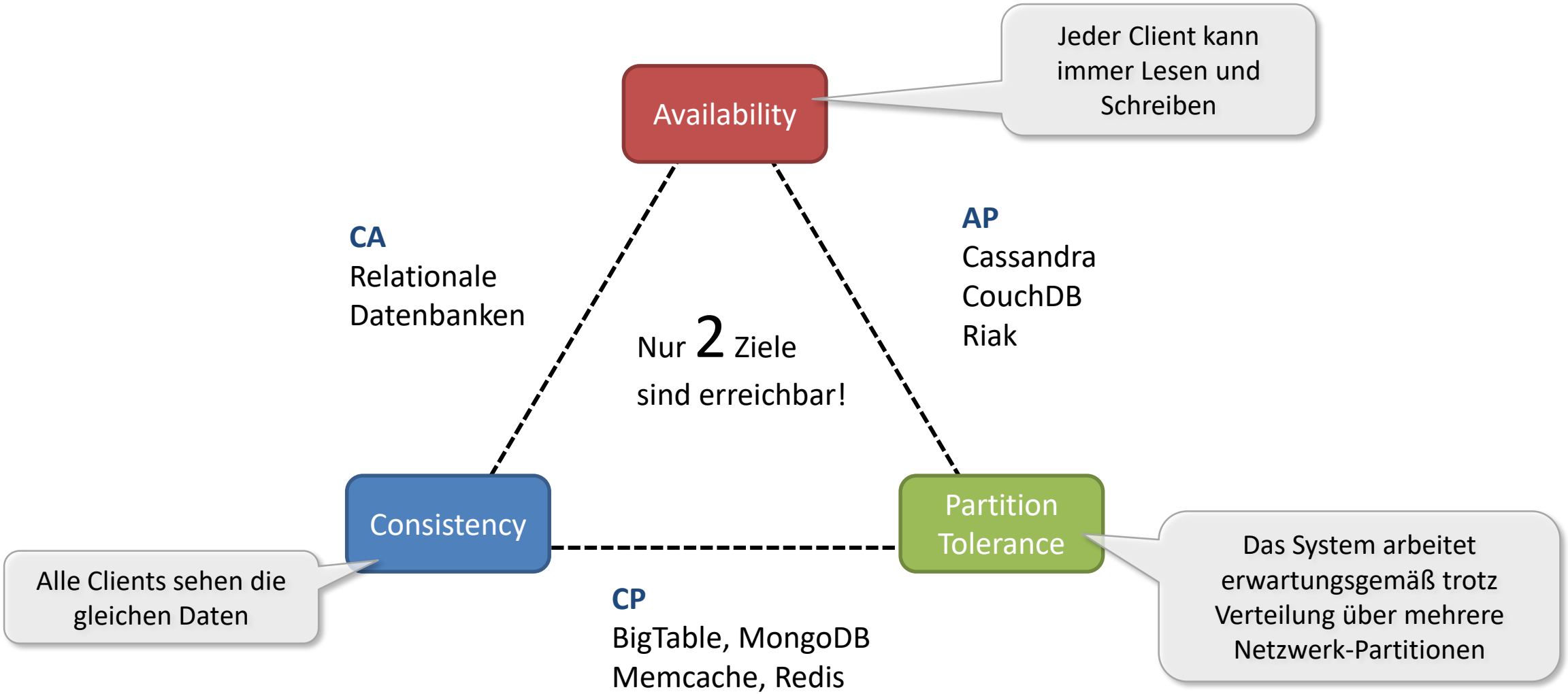
- ACID skaliert nur bis zu einem gewissen Grad
  - ACID bei verteilten Datenspeichern nur schwierig umzusetzen
  - Verfügbarkeit, Fehlertoleranz und Performance wichtiger als Konsistenz und Isolation
- BASE ersetzt ACID bei hochskalierenden Datenspeichern
  - Basically Available
  - Soft State
  - Eventual Consistency

Gegenüberstellung ACID - BASE

ACID	BASE
• Starke Konsistenz	• Schwache Konsistenz
• Isolation	• Hohe Verfügbarkeit
• Fokus auf Commit	• Best Effort
• Konservativ (pessimistisch)	• Aggressiv (optimistisch)
• Harte Strukturen, lassen sich nur schwer weiterentwickeln	• Weiche Strukturen, lassen sich leicht weiterentwickeln

Quelle: Dr. Eric A. Brewer, „Towards Robust Distributed Systems" PODC Keynote 19.07.2000

CAP-Theorem


Availability
Jeder Client kann immer Lesen und Schreiben
CA
Relationale Datenbanken
AP
Cassandra
CouchDB
Riak
Nur 2 Ziele sind erreichbar!
Consistency
Partition Tolerance
Alle Clients sehen die gleichen Daten
CP
BigTable, MongoDB
Memcache, Redis
Das System arbeitet erwartungsgemäß trotz Verteilung über mehrere Netzwerk-Partitionen


Security

Authentisierung / Autorisierung

Authentisierung (Wer bin ich?)

- Identifizierung eines Objekts und die Überprüfung der Identität
- Identität entspricht Benutzername
- Identifizierung erfolgt über Credentials
  - Passwort
  - Zertifikat
  - Biometrische Eigenschaften

Autorisierung (Was darf ich?)

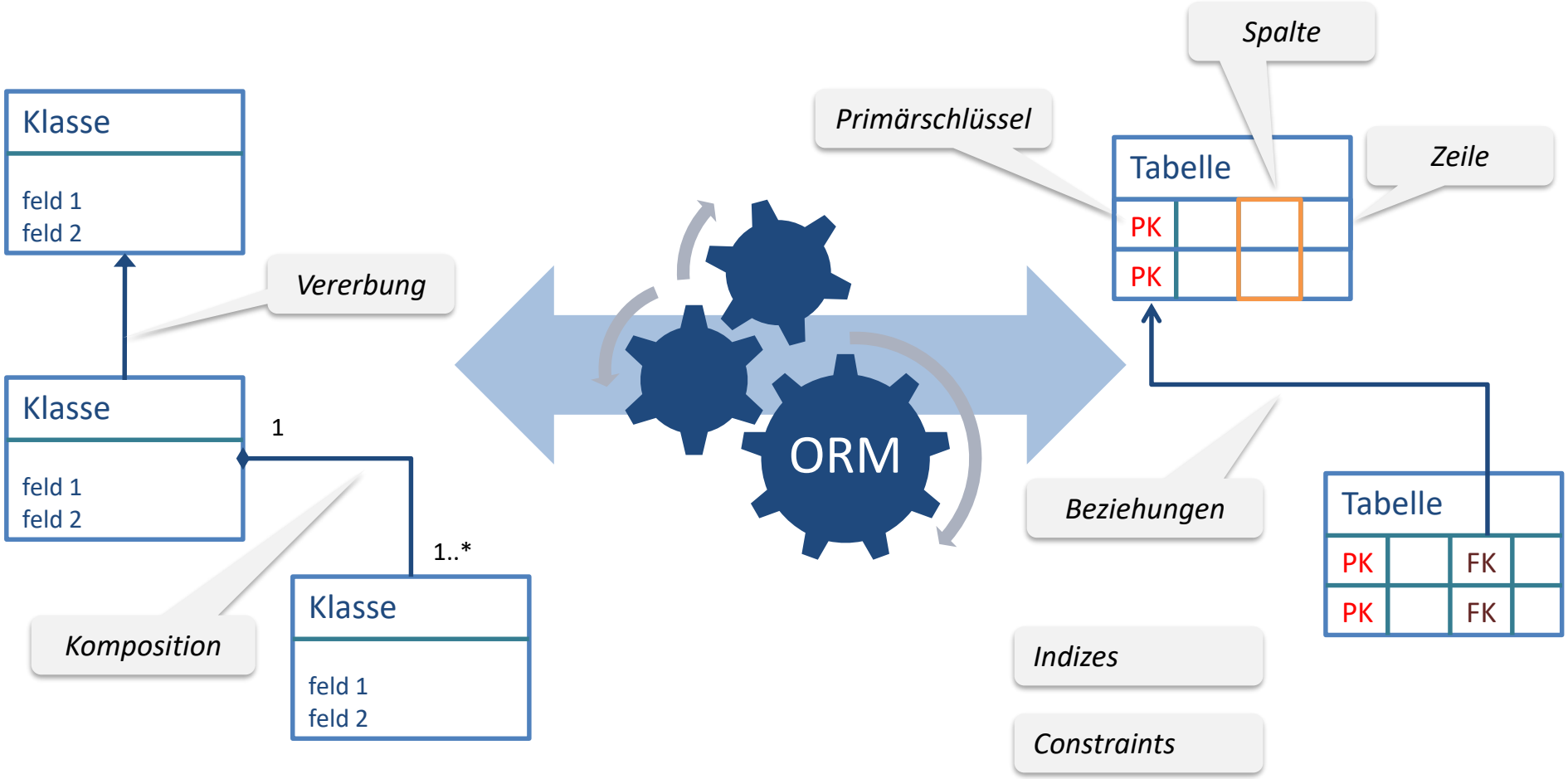
- Überprüfung von Berechtigungen eines angemeldeten Benutzers bezogen auf geschützte Ressourcen
- Berechtigungen entsprechen Rollen oder Gruppen

Propagation von Securitykontexten

- Bei Aufrufen entfernter Anwendungen wird der Securitykontext mit übertragen
  - Aufgerufene Methode läuft im gleichen Securitykontext wie der Aufrufer
  - Benutzer muss sich nicht erneut anmelden (Single Sign-On/SSO)
- Beteiligte Container (Application Server) müssen einander trauen
- Umsetzung teilweise aufwändig

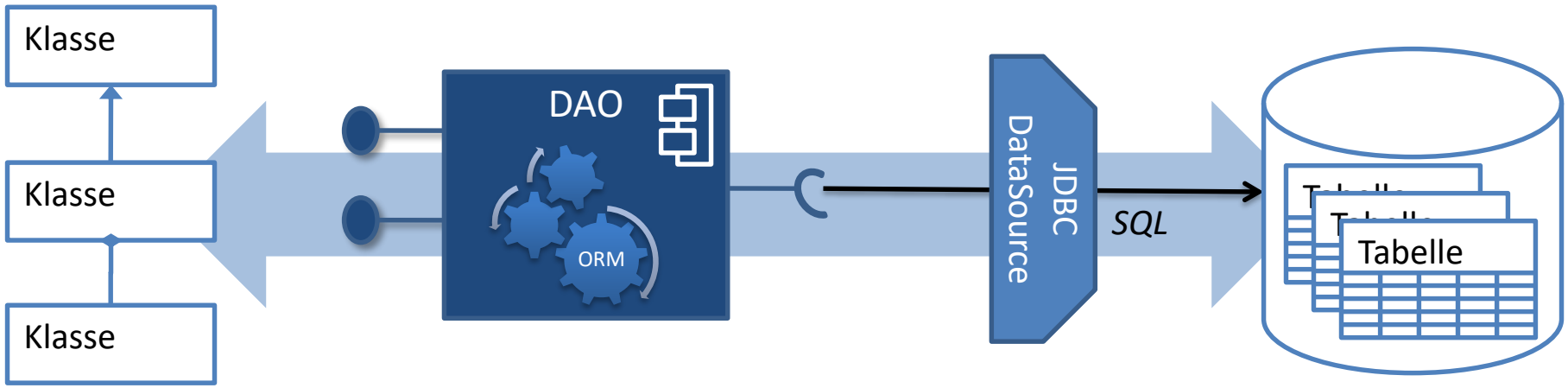
Persistenz

Object Relational Mapping (ORM)



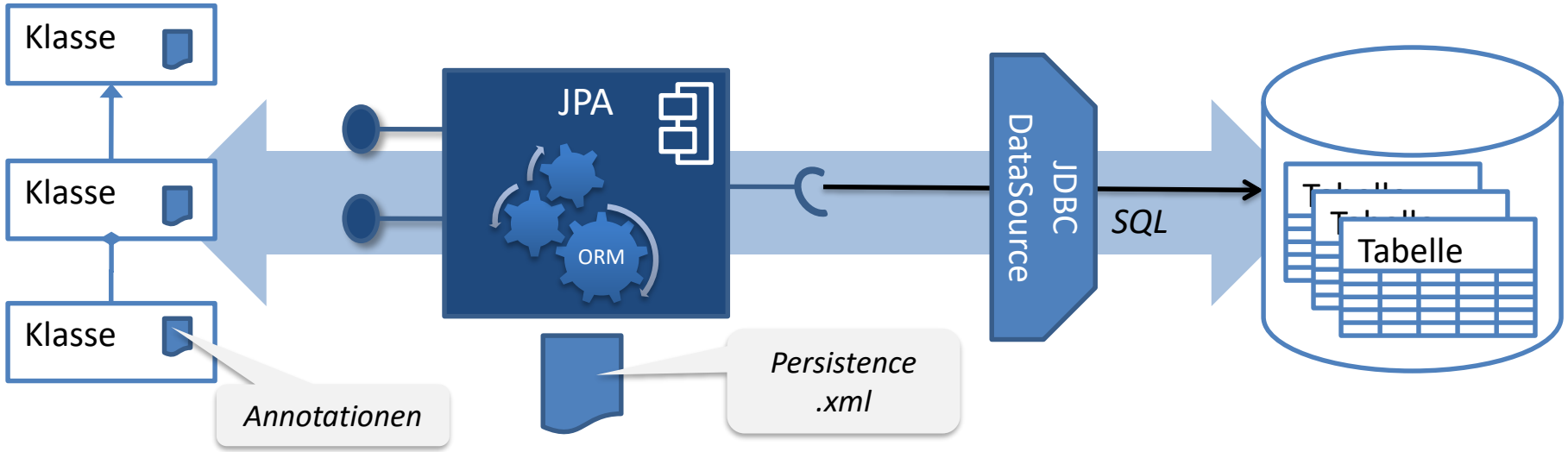
Nativer Datenzugriff mit JDBC

- Mapping muss programmiert werden
- Zugriff auf die Datenbank mit SQL muss programmiert werden

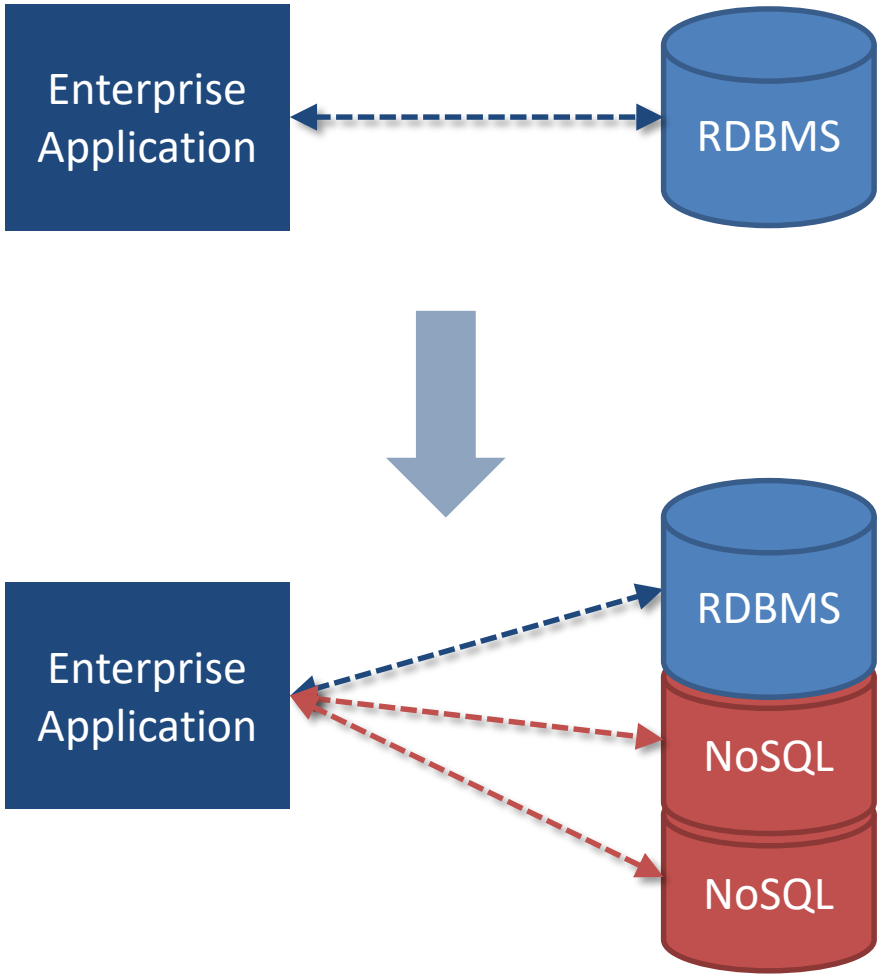


Datenzugriff mit Persistenzframework

- Persistente Klassen werden mit Mapping-Annotationen versehen und mit Datasource verknüpft
- Persistenzframework führt Mapping durch und generiert SQL zur Laufzeit



Ausblick: Polyglot Persistence



- Relationale Datenbanken stoßen zunehmen an ihre Grenzen
- Ablage der Daten in einem Datenspeicher nicht mehr zeitgemäß / sinnvoll
- Moderne Applikationen stützen sich je nach Anforderungen auf unter-schiedliche Datenbanken
- Kenntnisse über alternative Datenspeicher erforderlich

Fragen?

ANHANG

Quellen

- Martin Fowler: ***Patterns of Enterprise Application Architecture***
  Addison Wesley 2003; ISBN 0-321-12742-0
- Adam Bien: ***Real World Java EE Patterns: Rethinking Best Practices***
  press.adam-bien.com September 2012; ISBN 978-0-300-14931-6
- Gregor Hohpe, Bobby Wolfe
  Enterprise Integration Patterns
  Addison Wesley, 2004, ISBN: 0-321-20068-3
- Eric Evans: ***Domain Driven Design:***
  Tackling Complexity in the Heart of Software
  Addison Wesley 2004; ISBN 0-321-12521-5

Kontakt

Michael Theis
Lehrbeauftragter Hochschule München

email michael.theis@hm.edu
mobile + 49 170 5403805
web http://www.tschutschu.de/Lehrauftrag.html