Architektur verteilter Anwendungen

FWP Aktuelle Technologien zur Entwicklung
verteilter Java-Anwendungen

Wer braucht schon einen Architekten?

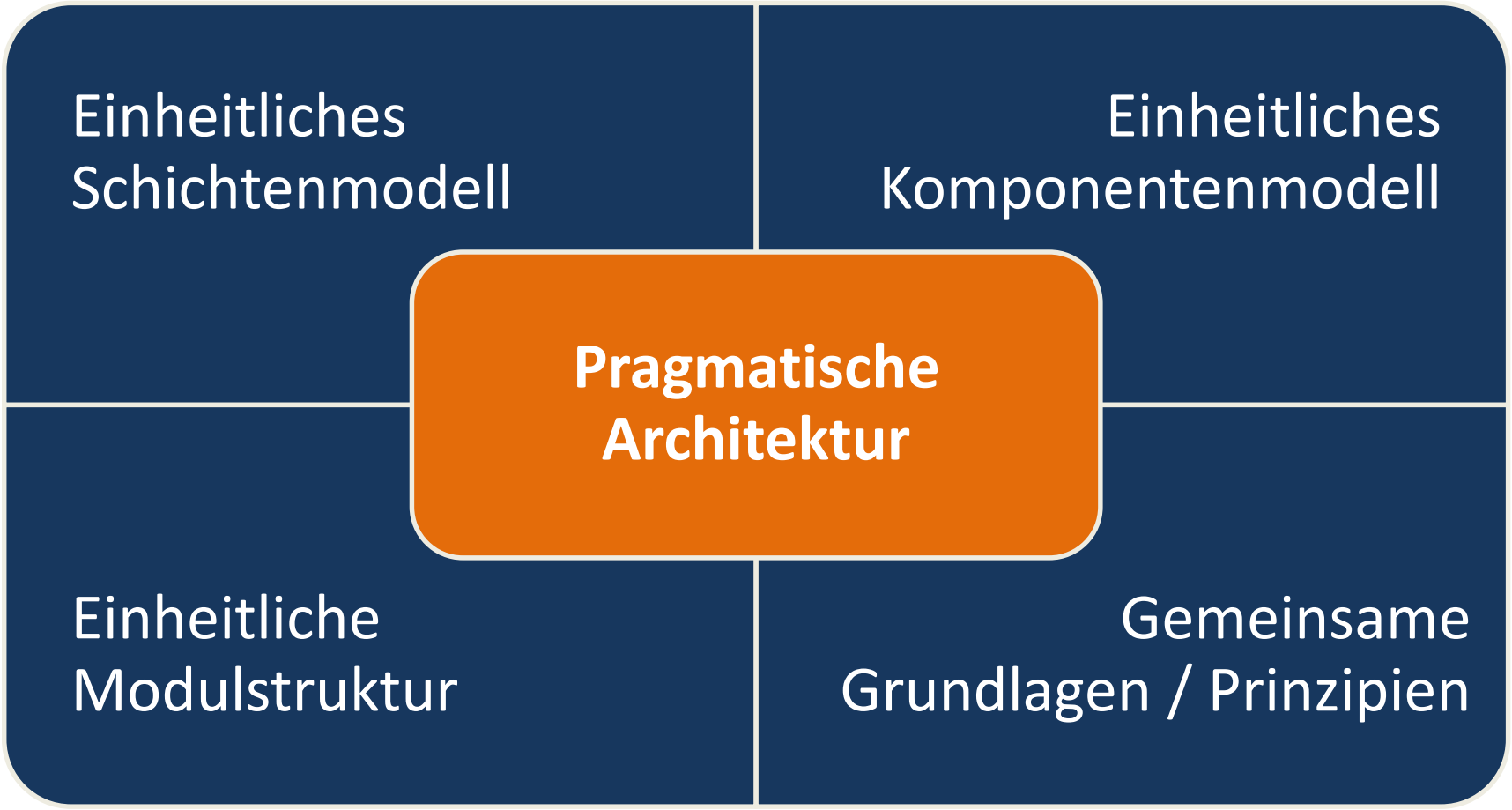
ARCHITEKTUR VERTEILTER ANWENDUNGEN

Anforderungen an Architektur

- Organisches Wachstum von Applikationen
- Beherrschbarkeit von zunehmender Komplexität
- Schlank bei hoher Leistungsfähigkeit
  - Architektur muss einfach und verständlich sein
  - Architektur muss helfen und darf nicht behindern

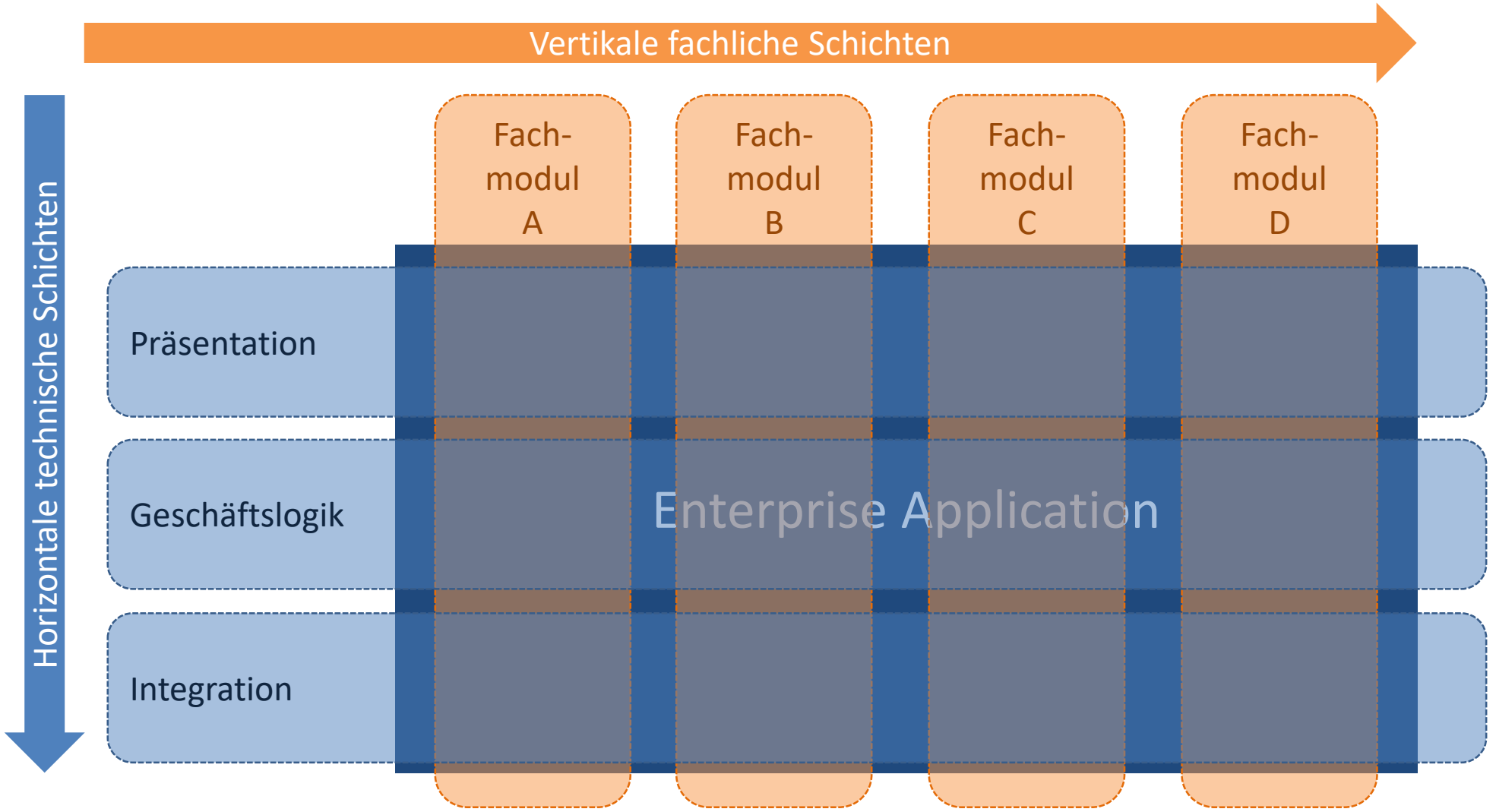
⇨ Forderung nach **Pragmatischer Architektur**

Bausteine einer Pragmatischen Architektur



Schichtenmodell als Basis

Einheitliches Schichtenmodell



Merkmale der Schichten

- Fachliche (horizontale) Unterteilung kommt vor technischer (vertikaler) Unterteilung
- 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 oder eigenes Modul

Drei prinzipielle Schichten

Präsentation
(Presentation)*

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

Geschäftslogik
(Business, Domain)*

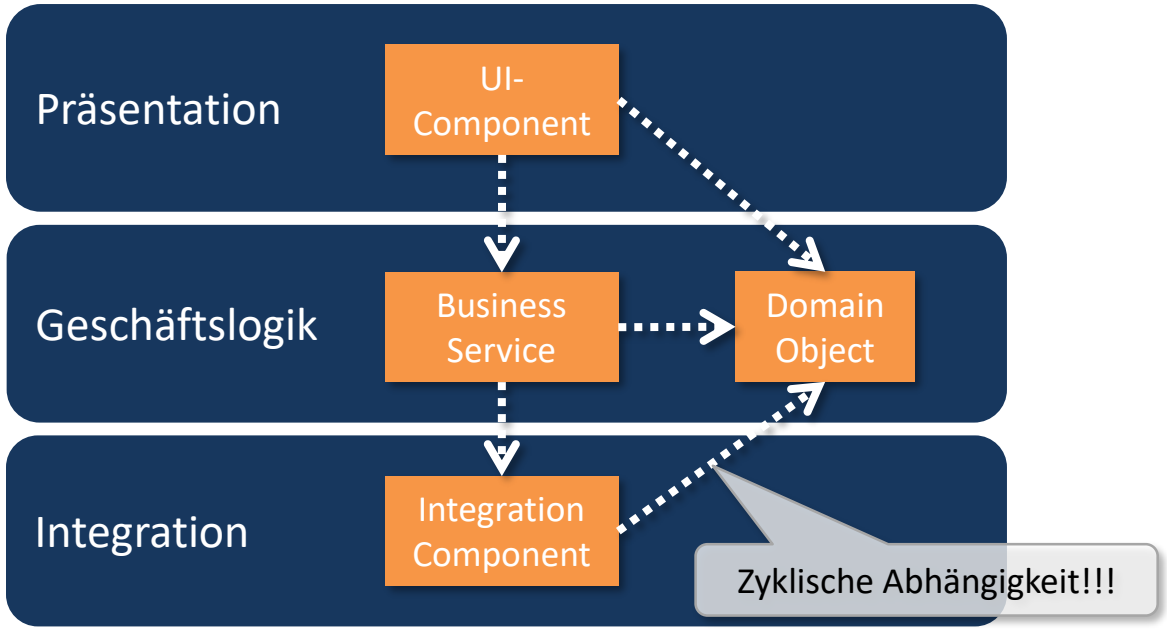
- Kern der Anwendungen bestehend aus Diensten in einer Serviceschicht *(Service Layer)* und Domänenmodell *(Domain Model)*

Integration
(Integration, Data Source)*

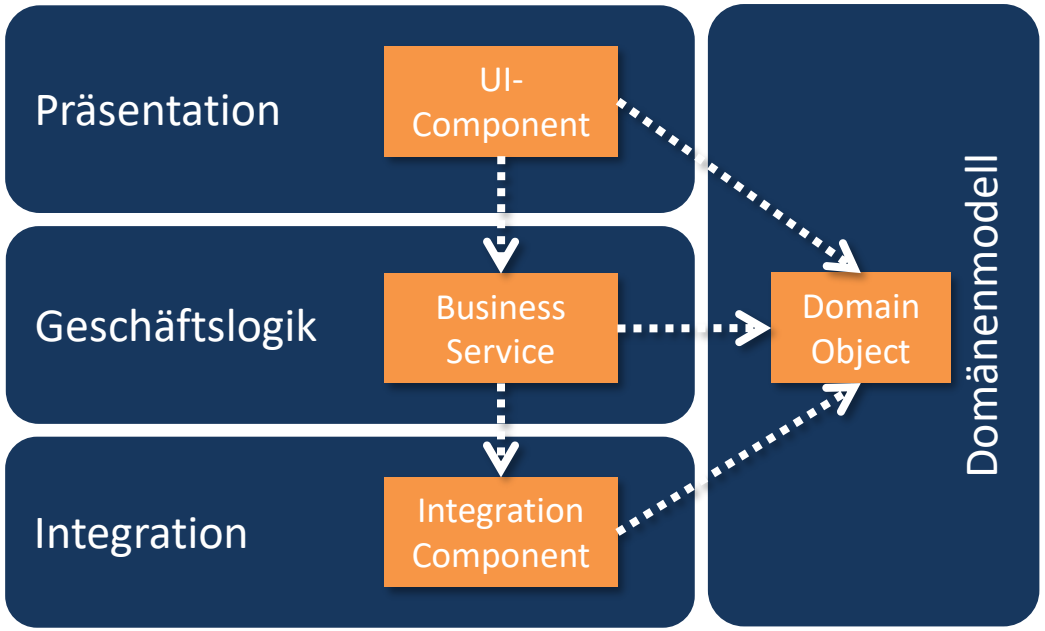
- Integration externer Ressourcen
- Transformation externes Domänenmodell / internes Domänenmodell

Problem: Zyklische Abhängigkeiten

Schichtenmodell mit 3 Schichten



Bereinigtes Schichtenmodell



Konventionen für 3 Schichten-Modell

- Gemeinsames Über-Package für Applikation:

  `${org-name}.${app-name}.*`

- Gemeinsames Über-Package für fachliche Schicht (Domäne):

  `${org-name}.${app-name}.${domain-name}.*`

- Gemeinsames Über-Package für technische Schicht:

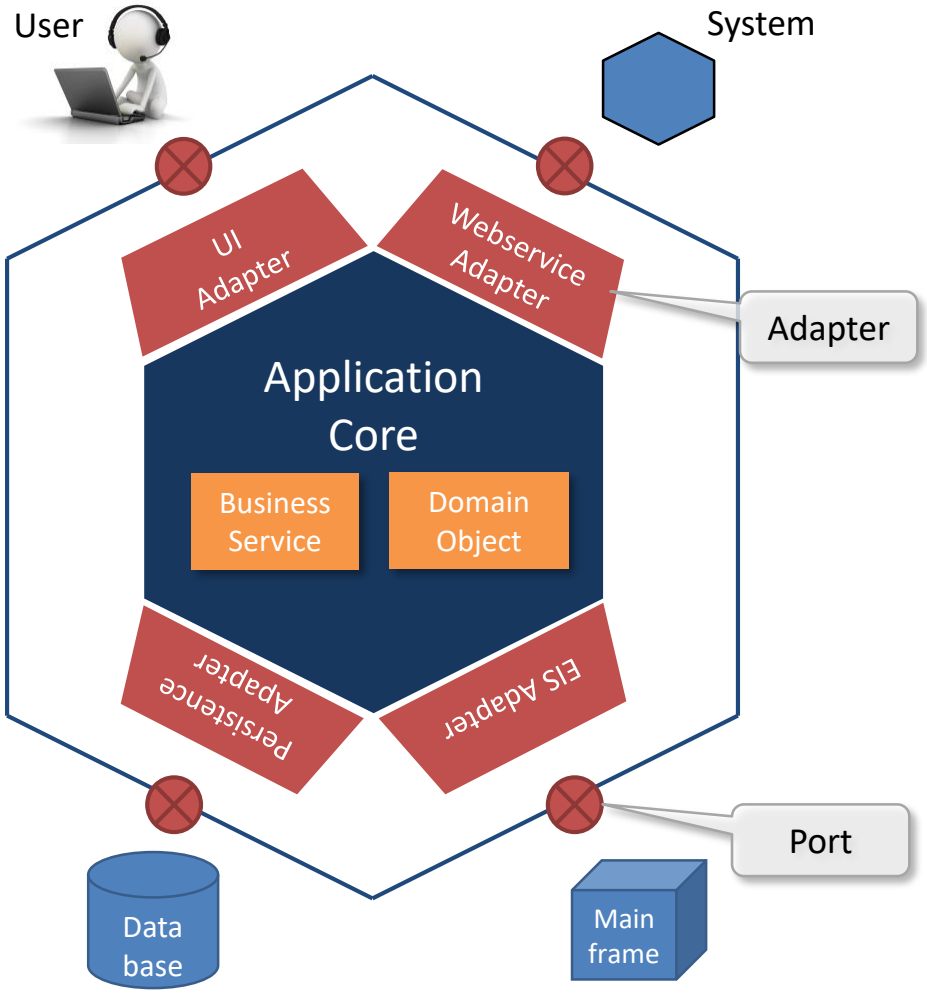
  `${org-name}.${app-name}.${domain-name}.presentation.*`

  `${org-name}.${app-name}.${domain-name}.business.*`

  `${org-name}.${app-name}.${domain-name}.integration.*`

- Optional: eigene Module pro Domäne oder pro Schicht

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

Konventionen für Hexagonale Architektur

- Über-Packages für Applikation und Fachdomäne wie bei Schichtenmodell
- Gemeinsames Über-Package für Kern:

  `${org-name}.${app-name}.${domain-name}.core.*`
- Gemeinsames Über-Package für Adapter:

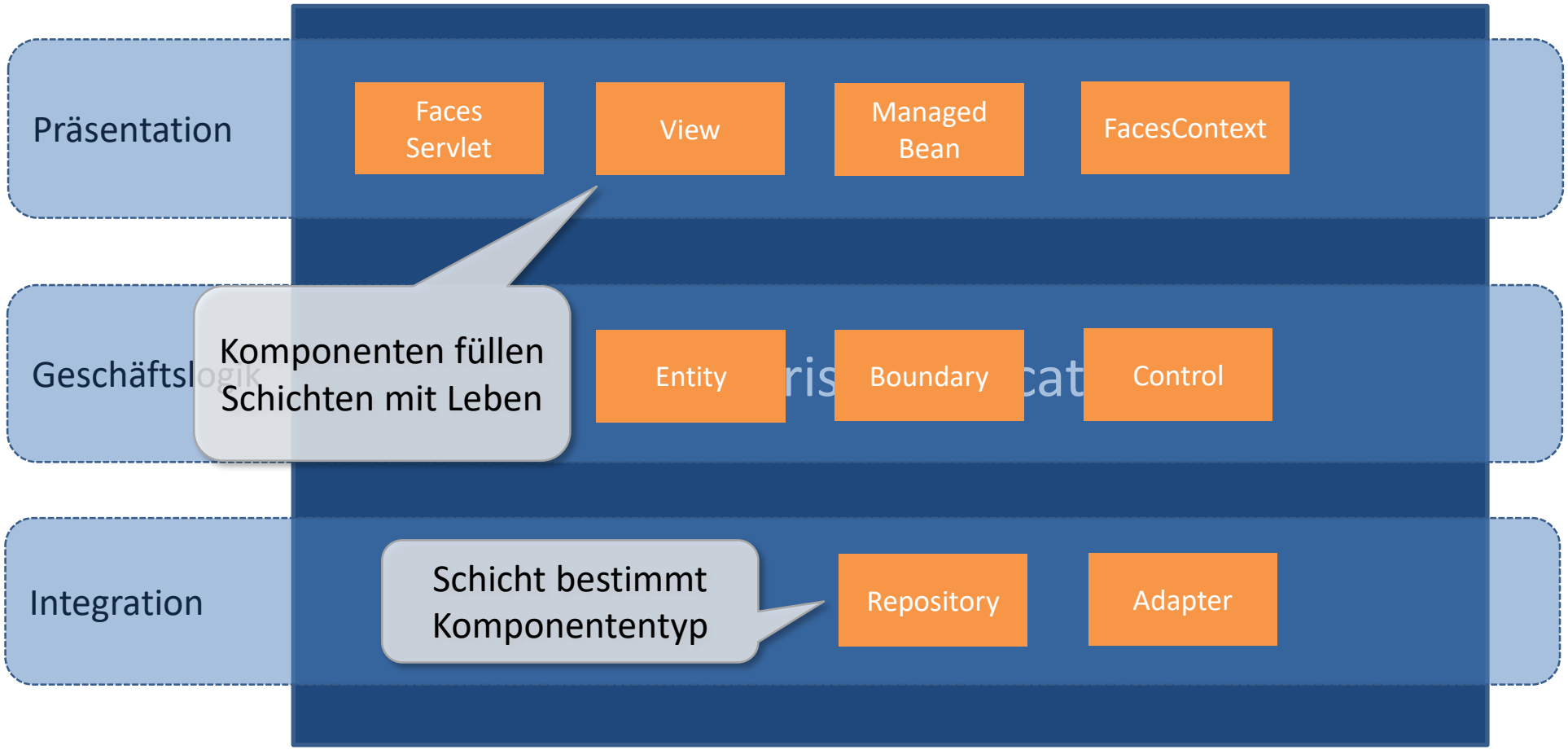
  `${org-name}.${app-name}.${domain-name}.adapter.ui.*`

  `${org-name}.${app-name}.${domain-name}.adapter.persistence.*`

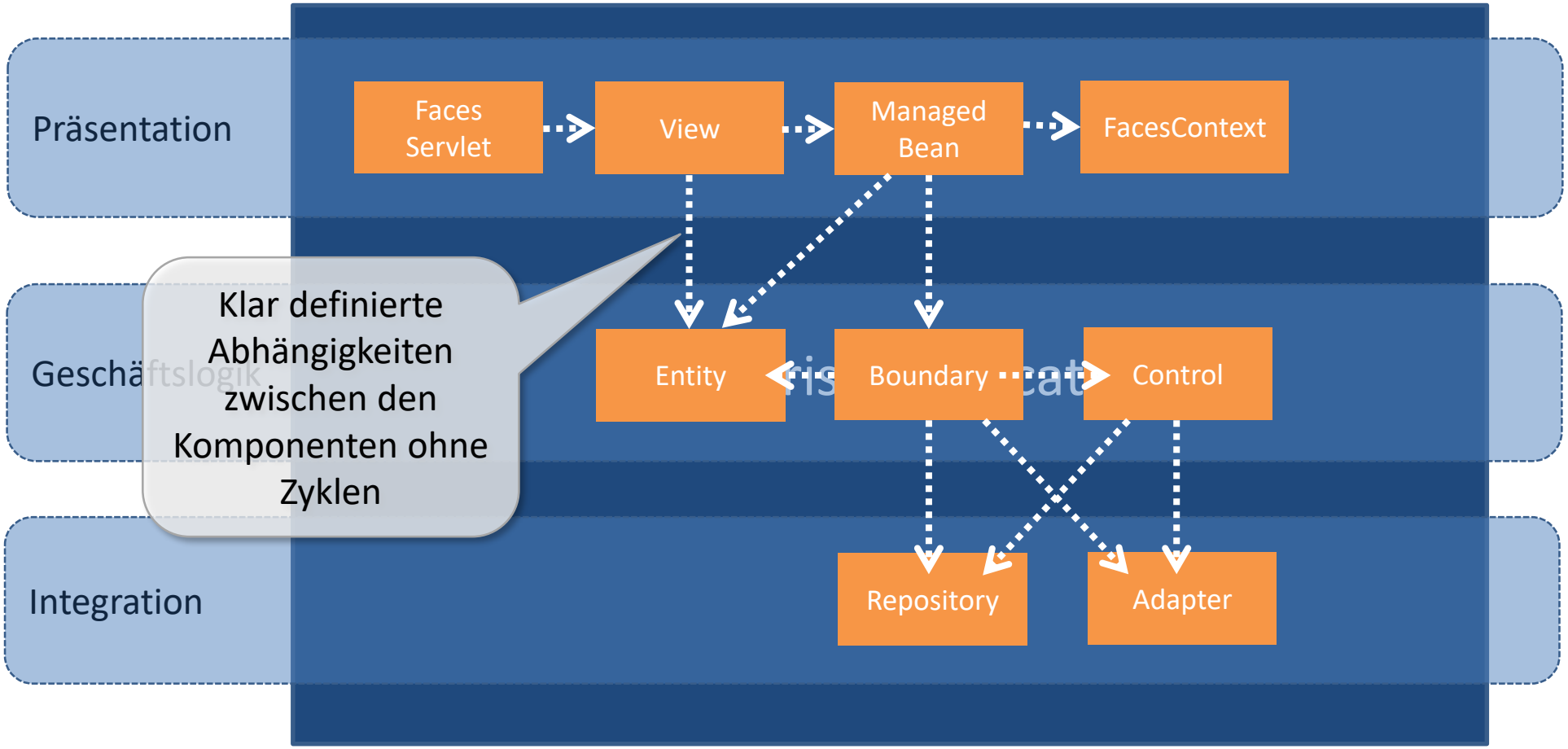
  `${org-name}.${app-name}.${domain-name}.adapter.rest.*`
- Optional: eigene Module pro Domäne, Kern und Adapter

Aufgebaut aus Komponenten

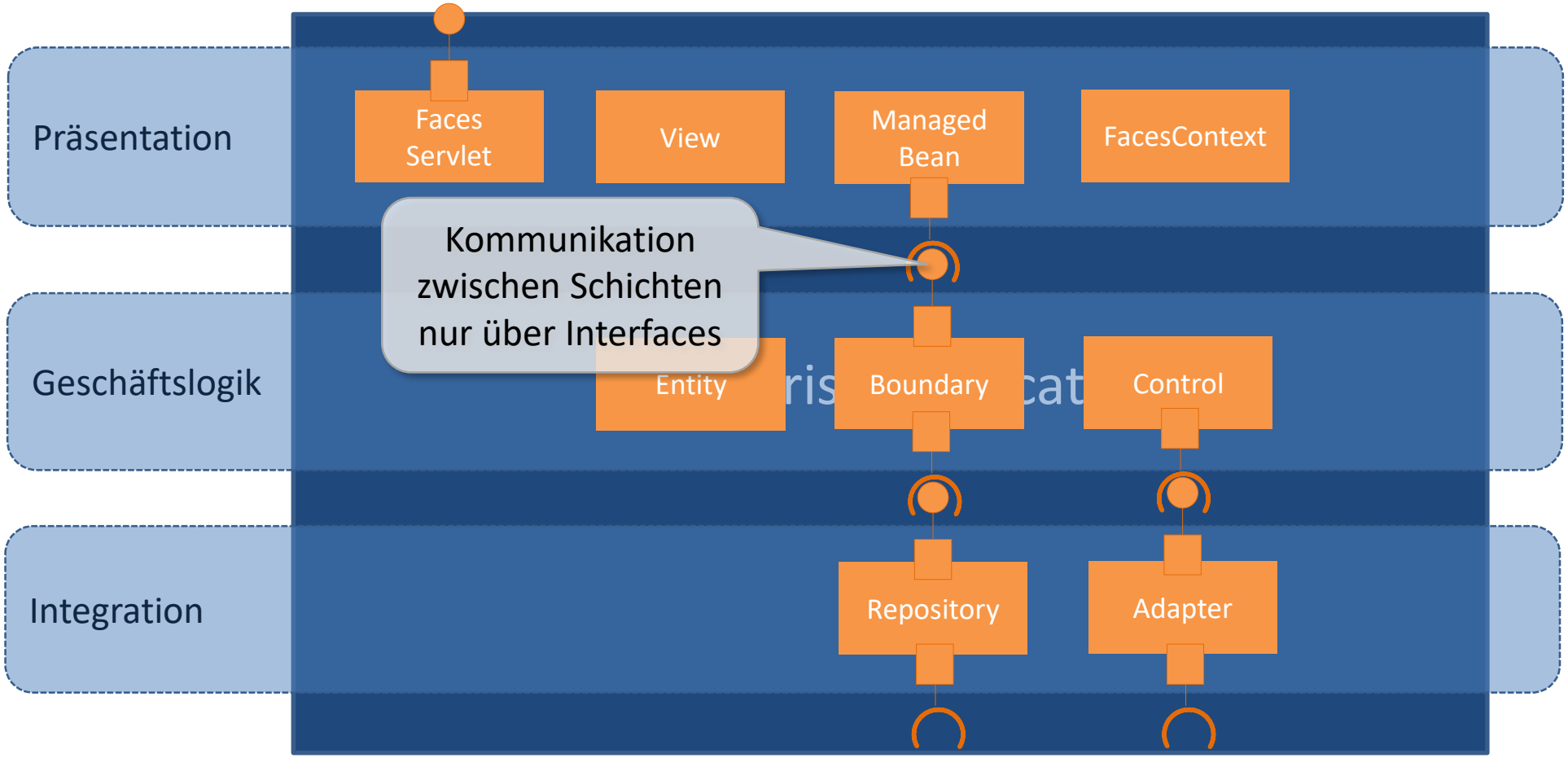
Einheitliches Komponentenmodell



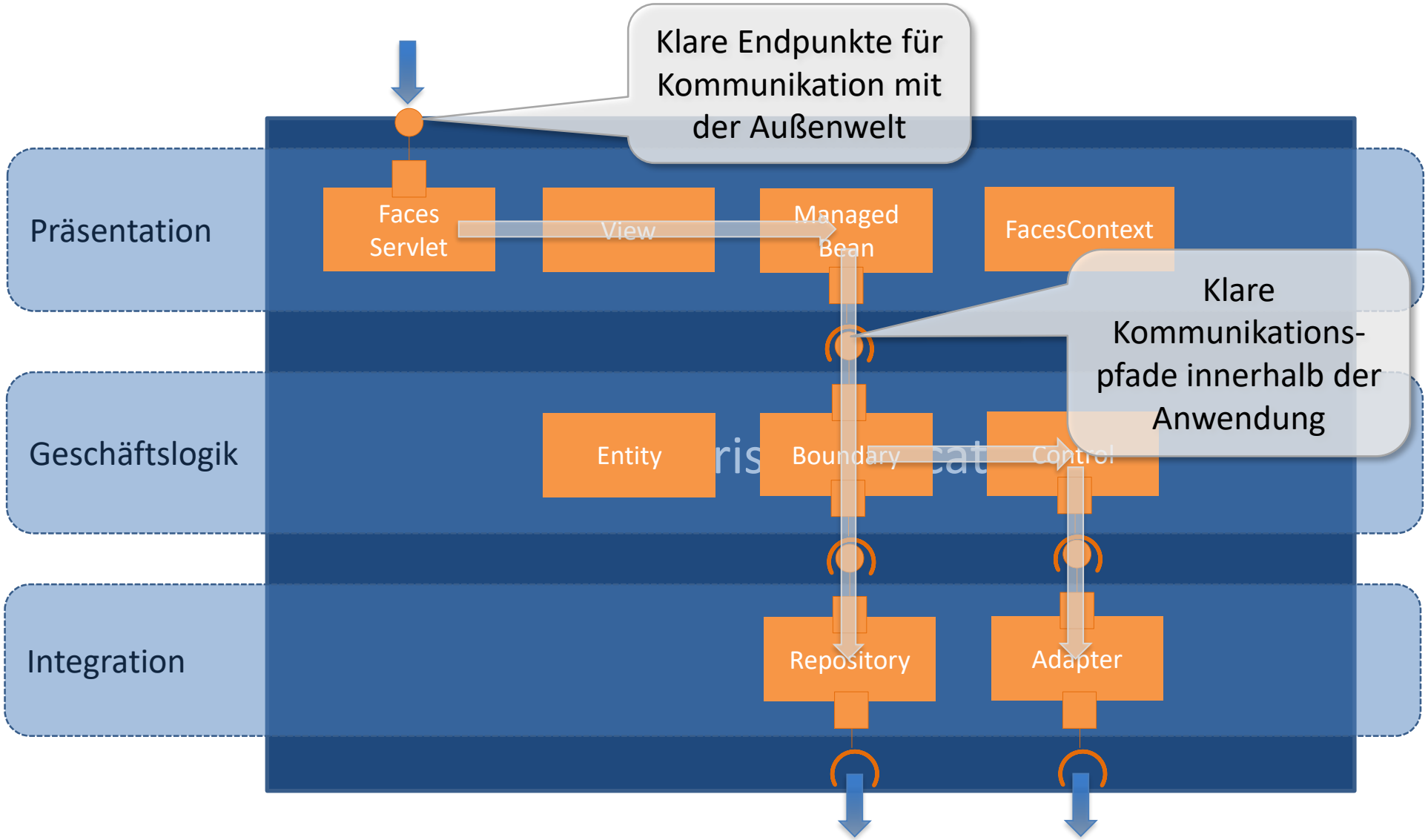
Einheitliches Komponentenmodell



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

Gepackt in Module

Einheitliche Modulstruktur

***Modul**: austauschbares, komplexes Element innerhalb eines Gesamtsystems [..], das eine geschlossene funktionale Einheit bildet.*
(Quelle: Duden)

Funktionseinheit (Functional Unit)

- Geschlossene fachliche Funktionseinheit
- Eigener Namensraum (Package)

Baueinheit (Build Unit)

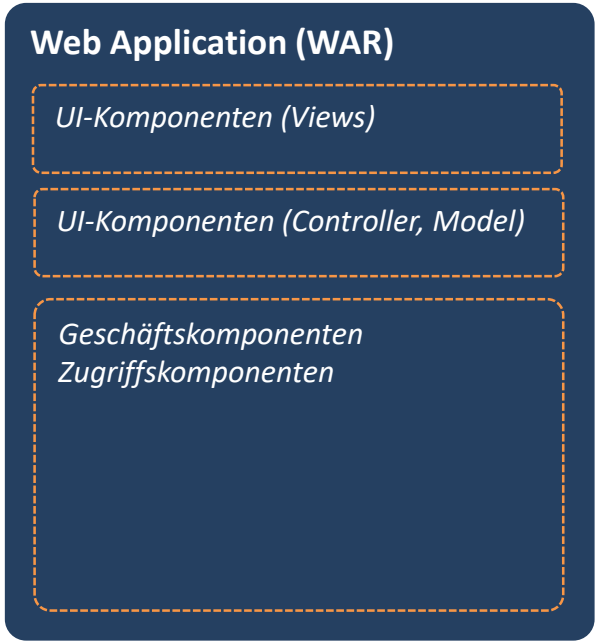
- Zusammenfassung von Funktions-einheiten
- Wird durch ein Build-Tool gebaut

Deploymenteinheit (Deployment Unit)

- Zusammenfassung von Baueinheiten
- Kann einzeln auf einer Laufzeit-umgebung bereit-gestellt und aus-geführt werden

Gewichtsklassen

Leichtgewicht

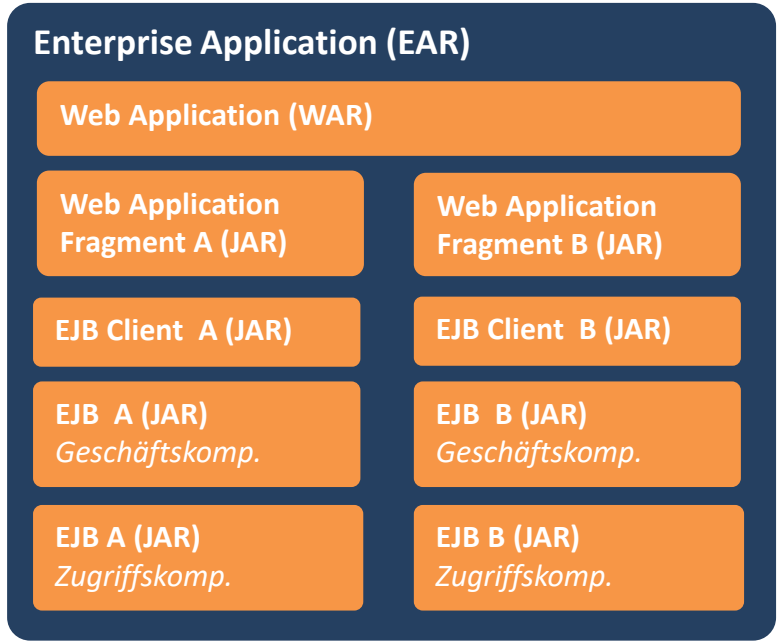


- Auslieferung als WAR
- Schichtentrennung über Packages
- Web Profile oder Micro Profile Application-Server ausreichend
- Ein Maven-Projekt

Mittelgewicht

- Auslieferung als EAR
- Schichtentrennung über Packages und Module
- Full Profile Server erforderlich
- Multi-Module Maven-Projekt

Schwergewicht



- wie Mittelgewicht plus:
- Pro fachliche Domäne eigener Modulstack

GEMEINSAME PATTERNS UND PRINZIPIEN

Gemeinsame Patterns und Prinzipien

STRIKTE TRENNUNG VON ZUSTÄNDIGKEITEN

Keine Mischung von Fachlichkeit und Technik

- Komponenten lassen sich in Kategorien (Blutgruppen) einteilen
- Vermischung von Blutgruppen erhöht Komplexität und verringert Wartbarkeit
- Pro Komponente nur eine Aufgabe

O-Software	A-Software	T-Software	AT-Software	R-Software
• unabhängig von Fachlichkeit und Technik • meist Klassenbibliotheken	• bestimmt durch Fachlichkeit • unabhängig von Technik • ändert sich nur aus fachlichen Gründen	• bestimmt durch Technik • unabhängig von Fachlichkeit • ändert sich nur aus technischen Gründen	• bestimmt durch Fachlichkeit UND Technik • ändert sich aus fachlichen und technischen Gründen • schwer zu warten • widersetzt sich Änderungen	• transformiert fachliche Objekte in externe Repräsentationen • Vermischung von Fachlichkeit und Technik zulässig

Quelle: Johannes Siedersleben, Moderne Software-Architektur, dpunkt.verlag 2004

Gemeinsame Patterns und Prinzipien

FACHLICHKEIT VOR TECHNIK

Design in der Sprache des Kunden

- Design der Applikation basiert auf fachlichen und nicht auf technischen Gesichtspunkten
- Alle Artefakte werden nach der Sprache der zugrundeliegenden Fachdomäne benannt
  - Module, Klassen, Interfaces, Felder, Methoden
- Ubiquitous Language schafft gemeinsames Vokabular zwischen Fachabteilungen und Softwareentwicklern

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;
```

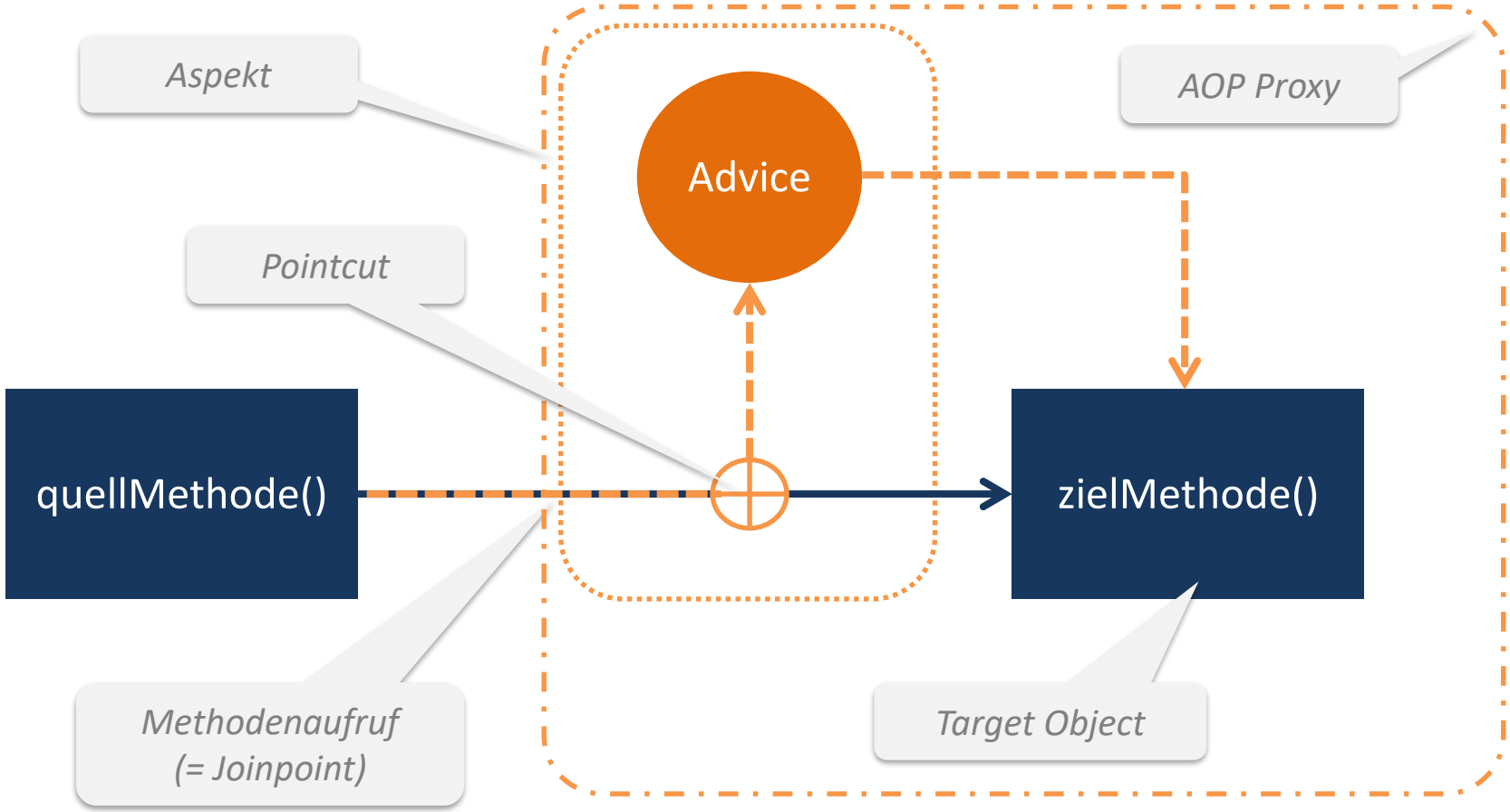
- Container erkennt annotiertes Feld, erzeugt ein Objekt von *ServiceImpl* und injiziert Referenz in Feld *service*

☺ 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

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