Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Softwarearchitektur und Design

Grundlagen und Überblick

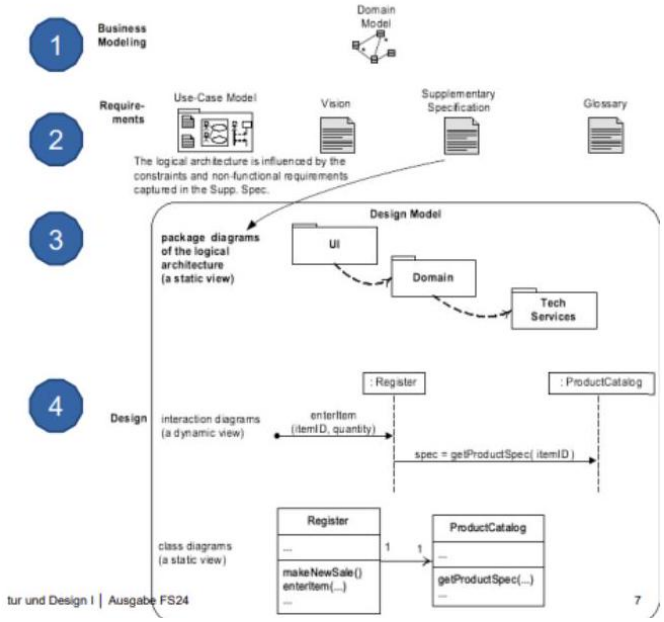
- Business Analyse vs Architektur vs Entwicklung x - Definition Softwarearchitektur x - Architekturanalyse und Twin Peaks Model x - ISO 25010 vs FURPS+ x

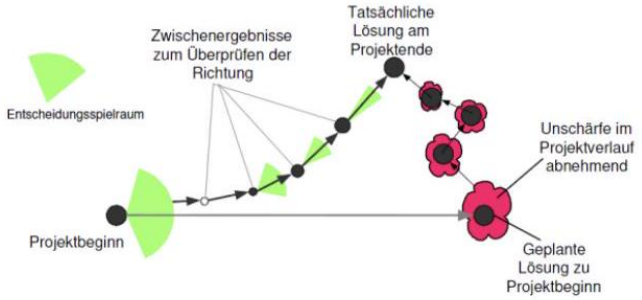
Grundlagen und Überblick

- **Business Analyse:**
  - Domänenmodell und Kontextdiagramm
  - Requirements (funktional und nicht-funktional)
  - Vision und Stakeholder
- **Architektur:**
  - Logische Struktur des Systems
  - Technische Konzeption
  - Qualitätsanforderungen
- **Entwicklung:**
  - Use Case / User Story Realisierung
  - Design-Klassendiagramm (DCD)
  - Implementierung und Tests

Architektur und Design sind eng verzahnt und bauen aufeinander auf:

- Architektur definiert das "große Ganze"
- Design spezifiziert die Details der Umsetzung
- Beides basiert auf Requirements und führt zur Implementation





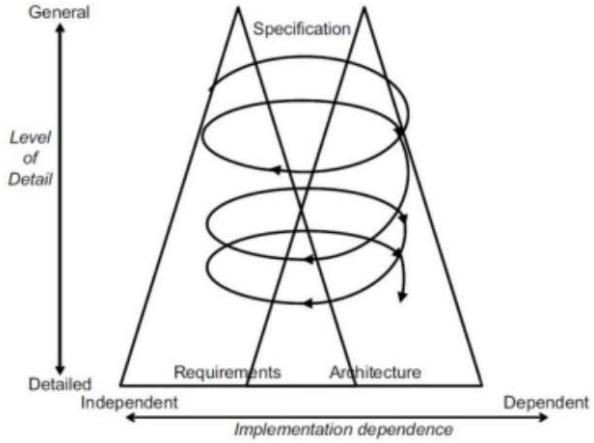
Softwarearchitektur Die Architektur eines Softwaresystems definiert:

- **Grundlegende Entscheidungen:**
  - Programmiersprachen und Plattformen
  - Aufteilung in Teilsysteme und Komponenten
  - Schnittstellen zwischen Komponenten
- **Strukturelle Aspekte:**
  - Verantwortlichkeiten der Teilsysteme
  - Abhängigkeiten zwischen Komponenten
  - Einsatz von Basis-Technologien/Frameworks
- **Qualitätsaspekte:**
  - Erfüllung nicht-funktionaler Anforderungen
  - Maßnahmen für Performance, Skalierbarkeit etc.
  - Fehlertoleranz und Ausfallsicherheit

Architekturanalyse

erfolgt iterativ mit den Anforderungen (Twin Peaks Model):

- **Anforderungsanalyse:**
  - Analyse funktionaler und nicht-funktionaler Anforderungen
  - Prüfung der Qualität und Stabilität der Anforderungen
  - Identifikation von Lücken und impliziten Anforderungen
- **Architekturentscheidungen:**
  - Abstimmung mit Stakeholdern
  - Berücksichtigung von Randbedingungen
  - Vorausschauende Planung für zukünftige Änderungen



Qualitätsanforderungen

ISO 25010:

- Hierachische Struktur für nicht-funktionale Anforderungen
- Definierte Hauptcharakteristiken und Subcharakteristiken
- Messbare Metriken für jede Anforderung
- Ermöglicht präzise Formulierung und Verifikation

FURPS+:

- **F**unctionality (Funktionalität)
- **U**sability (Benutzerfreundlichkeit)
- **R**eliability (Zuverlässigkeit)
- **P**erformance (Leistung)
- **S**upportability (Wartbarkeit)
- **+**: Implementation, Interface, Operations, Packaging, Legal

Architektur-Design

- Modulkonzept (Kohäsion/Kopplung) x - Architektursichten (4+1 View Model) x - Architektur-Entwurf und Best Practices x - Qualitätskriterien und deren Umsetzung

Modulkonzept

Ein Modul (Baustein, Komponente) wird bewertet nach:

- **Kohäsion:** Innerer Zusammenhang
- **Kopplung:** Externe Abhängigkeiten

Eigenschaften:

- Autarkes Teilsystem
- Minimale externe Schnittstellen
- Enthält alle benötigten Funktionen/Daten
- Verschiedene Formen: Paket, Library, Service

Schnittstellen

Module kommunizieren über definierte Schnittstellen:

- **Exportierte Schnittstellen:**
  - Definieren angebotene Funktionalität
  - Vertraglich garantierte Leistungen
  - Einzige nach außen sichtbare Information
- **Importierte Schnittstellen:**
  - Von anderen Modulen benötigte Funktionalität
  - Definieren Abhängigkeiten
  - Basis für Kopplung
  - Sollten minimiert werden (Low Coupling)

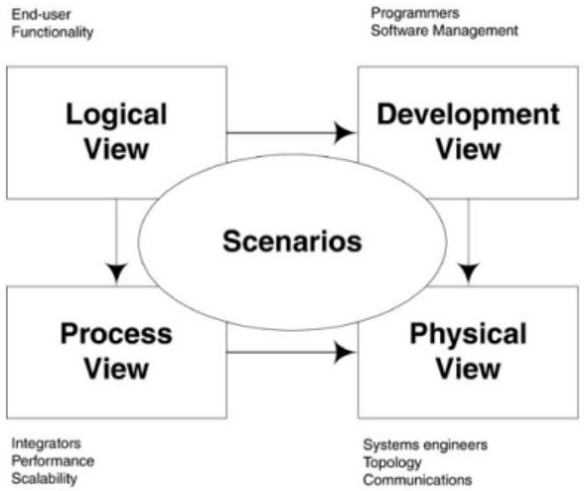
Architektursichten (4+1 View Model)

Verschiedene Perspektiven auf die Architektur:

- **Logical View:** End-User, Functionality
  - Funktionalität des Systems
  - Schichten, Subsysteme, Pakete
  - Klassen und Schnittstellen
- **Process View:** Integrators, Performance, Scalability
  - Laufzeitverhalten
  - Prozesse und Threads
  - Performance und Skalierung
- **Development View:** Programmers, Software Management
  - Implementierungsstruktur
  - Quellcode-Organisation
  - Build und Deployment
- **Physical View:** System Engineers, Topology, Communications
  - Hardware-Topologie
  - Verteilung der Software
  - Netzwerkkommunikation

+1: Scenarios:

- Wichtige Use Cases
- Validierung der Architektur
- Integration der anderen Views



Designprinzipien und Qualitätskriterien

Clean Architecture

Prinzipien nach Robert C. Martin:

Hauptprinzipien:

- Unabhängigkeit von Frameworks
- Testbare Business Rules
- Unabhängigkeit von UI
- Unabhängigkeit von Datenbank
- Unabhängigkeit von externen Systemen

Schichten (von innen nach außen):

1. Entities (Enterprise Business Rules)
2. Use Cases (Application Business Rules)
3. Interface Adapters (Controllers, Presenters)
4. Frameworks & Drivers (UI, DB, Devices)

Dependency Rule: Abhängigkeiten dürfen nur nach innen zeigen.

Architekturprinzipien

Grundlegende Prinzipien für gute Architektur:

Separation of Concerns:

- Trennung von Verantwortlichkeiten
- Klare Modulgrenzen
- Reduzierte Komplexität

Information Hiding:

- Kapselung von Implementierungsdetails
- Definierte Schnittstellen
- Änderbarkeit ohne Seiteneffekte

Loose Coupling:

- Minimale Abhängigkeiten
- Austauschbarkeit
- Unabhängige Entwicklung

Qualitätskriterien und deren Umsetzung

Strategien zur Erfüllung von Qualitätsanforderungen:

Performance:

- Effiziente Ressourcennutzung (Resource Pooling, Caching)
- Optimierte Verarbeitung (Parallelisierung, Lazy Loading)

Skalierbarkeit:

- Dynamische Anpassung (horizontale/vertikale Skalierung)
- Effiziente Lastverteilung (Load Balancing, Partitionierung)

Wartbarkeit:

- Klare Strukturen (Separation of Concerns, Modularisierung)
- Verbesserte Codequalität (Information Hiding, Standardisierung)

Zuverlässigkeit:

- Fehlerresistenz (Redundanz, Fehlertoleranz)
- Prävention und Wiederherstellung (Monitoring, Backup/Recovery)

Verfügbarkeit:

- Ausfallschutz (Redundanz, Failover-Mechanismen)
- Überwachung/Stabilisierung (Health Monitoring, Circuit Breaker)

Modularität:

- Gut definierte Grenzen (klare Modulgrenzen, hohe Kohäsion)
- Minimale Abhängigkeiten zwischen Modulen

Testbarkeit:

- Einfachheit von Tests (Isolation, Mockbarkeit)
- Automatisierung und Skalierung von Tests

Änderbarkeit:

- Anpassungsfähigkeit (Lokalisierung, Erweiterbarkeit)
- Sicherstellung der Kompatibilität (Backward Compatibility)

Erweiterbarkeit:

- Flexible Architekturen (offene Schnittstellen, Plugin-Systeme)
- Serviceorientierung für modulare Erweiterungen

Gesamter Architekturprozess

1. Initiale Phase
- Architekturanalyse durchführen
- Grundlegende Entscheidungen treffen
- Ersten Entwurf erstellen

2. Iterative Verfeinerung
- Review durchführen
- Evaluation vornehmen
- Anpassungen basierend auf Feedback

3. Kontinuierliche Verbesserung
- Regelmäßige Reviews
- Neue Anforderungen einarbeiten
- Technische Schulden adressieren

4. Dokumentation
- Entscheidungen festhalten
- Architektur dokumentieren
- Änderungen nachverfolgen

5. Qualitätssicherung
- Architektur-Konformität prüfen
- Performance-Tests durchführen
- Sicherheitsaudits durchführen

Architekturprozess-Komponenten

Architekturanalyse:
- Erster Schritt im Architekturprozess
- Analyse der funktionalen und nicht-funktionalen Anforderungen
- Identifikation von Qualitätszielen
- Parallel zur Anforderungserhebung (Twin Peaks)

Architektur-Entscheidungen:
- Konkrete Beschlüsse basierend auf der Analyse
- Technologiewahl und Strukturierung
- Dokumentation und Begründung
- Einschließlich verworfener Alternativen

Architektur-Entwurf:
- Praktischer Gestaltungsprozess
- Anwendung von Architekturmustern
- Umsetzung von Qualitätsanforderungen
- Erstellung konkreter Artefakte

Architektur-Review:
- Systematische Überprüfung
- Meist durch externe Experten
- Prüfung der Anforderungserfüllung
- Identifikation von Schwachstellen

Architektur-Evaluation:
- Bewertung anhand definierter Kriterien
- Quantitative und qualitative Analyse
- Szenario-basierte Prüfung
- Bewertung von Qualitätsattributen

Architekturanalyse

1. Anforderungen sammeln
- Funktionale Anforderungen gruppieren
- Nicht-funktionale Anforderungen identifizieren
- Randbedingungen dokumentieren

2. Qualitätsziele definieren
- Messbare Kriterien festlegen
- Priorisierung vornehmen
- Trade-offs identifizieren

3. Einflussfaktoren analysieren
- Technische Faktoren
- Organisatorische Faktoren
- Wirtschaftliche Faktoren

Architektur-Entscheidungen

1. Alternativen identifizieren
- Mögliche Lösungen sammeln
- Vor- und Nachteile analysieren
- Machbarkeit prüfen

2. Bewertungskriterien
- Erfüllung der Anforderungen
- Technische Umsetzbarkeit
- Kosten und Aufwand

3. Entscheidung dokumentieren
- Begründung
- Konsequenzen
- Verworfene Alternativen

Architektur-Entscheidungen dokumentieren

1. Entscheidung festhalten
- Dokumentation der getroffenen Architekturentscheidungen
- Begründungen und Alternativen
- Auswirkungen und Konsequenzen

2. Strukturierte Dokumentation
- Einheitliches Format für alle Entscheidungen
- Verwendung von Templates
- Nachvollziehbare Historie der Entscheidungen

3. Kommunikation
- Regelmäßige Updates an Stakeholder
- Transparenz über getroffene Entscheidungen
- Einbindung des gesamten Teams

4. Review und Anpassung
- Regelmäßige Überprüfung der Entscheidungen
- Anpassung bei geänderten Rahmenbedingungen
- Lessons Learned dokumentieren

Architekturentwurf

Schritte:
1. Anforderungen analysieren
2. Architekturstil wählen
3. Module identifizieren
4. Schnittstellen definieren
5. Mit Stakeholdern abstimmen

Qualitätskriterien:
- Änderbarkeit
- Wartbarkeit
- Erweiterbarkeit
- Testbarkeit

Architektur-Review durchführen

Vorgehen:
1. **Vorbereitung**
   - Architektur-Dokumentation zusammenstellen
   - Review-Team zusammenstellen
   - Checklisten vorbereiten
2. **Durchführung**
   - Architektur vorstellen
   - Anforderungen prüfen
   - Entscheidungen hinterfragen
   - Risiken identifizieren
3. **Nachbereitung**
   - Findings dokumentieren
   - Maßnahmen definieren
   - Follow-up planen

Prüfkriterien:
- Anforderungserfüllung
- Technische Machbarkeit
- Zukunftssicherheit
- Best Practices

Architektur-Evaluation

Systematische Bewertung einer Softwarearchitektur:

1. Qualitätsattribute identifizieren
- Performance
- Skalierbarkeit
- Wartbarkeit
- Sicherheit

2. Szenarien entwickeln
- Normale Nutzung
- Grenzfälle
- Fehlerfälle
- Wartungsszenarien

3. Architektur analysieren
- Strukturanalyse
- Verhaltensanalyse
- Trade-off Analyse

4. Risiken identifizieren
- Technische Risiken
- Geschäftsrisiken
- Architekturrisiken

Beispiele

Typische Prüfungsaufgabe: Architekturanalyse und Entscheidungen

Aufgabenstellung: Analysieren Sie folgende Anforderungen und leiten Sie architektonische Konsequenzen ab:

- System muss 24/7 verfügbar sein
- 10.000 gleichzeitige Benutzer
- Reaktionszeit unter 1 Sekunde
- Jährliche Wartungsfenster maximal 4 Stunden

Lösung:

- **Architekturentscheidungen:**
  - Verteilte Architektur für Hochverfügbarkeit
  - Load Balancing für gleichzeitige Benutzer
  - Caching-Strategien für Performanz
  - Blue-Green Deployment für Wartung
- **Begründungen:**
  - Verteilung minimiert Single Points of Failure
  - Load Balancer verteilt Last gleichmäßig
  - Caching reduziert Datenbankzugriffe
  - Blue-Green erlaubt Updates ohne Downtime

Architekturentwurf

Aufgabe: Entwerfen Sie die grundlegende Architektur für ein Online-Banking-System.

Lösung:

- **Anforderungsanalyse:**
  - Sicherheit (ISO 25010)
  - Performance (FURPS+)
  - Skalierbarkeit
- **Architekturentscheidungen:**
  - Mehrschichtige Architektur
  - Microservices für Skalierbarkeit
  - Sicherheitsschicht
- **Module:**
  - Authentifizierung
  - Transaktionen
  - Kontoführung

Architekturmuster

- Schichtenarchitektur - Client-Server Architektur - Microservices - Clean Architecture - Event-Driven Architecture - Integration Patterns

Objektorientiertes Design

- GRASP Prinzipien - Responsibility Driven Design - Design Patterns in der Architektur - Anti-Patterns

UML-Modellierung

- Statische vs Dynamische Modelle - Klassendiagramme - Sequenzdiagramme - Zustandsdiagramme - Aktivitätsdiagramme - Verteilungsdiagramme - Paketdiagramme

GRASP Prinzipien

General Responsibility Assignment Software Patterns - Grundlegende Prinzipien für die Zuweisung von Verantwortlichkeiten:

Information Expert:
- Zuständigkeit basierend auf Information
- Klasse mit relevanten Daten übernimmt Aufgabe
- Fördert Kapselung und Kohäsion

Creator:
- Verantwortung für Objekterstellung
- Basierend auf Beziehungen (enthält, aggregiert)
- Starke Verwendungsbeziehung

Controller:
- Koordination von Systemoperationen
- Erste Anlaufstelle nach UI
- Fassade für Subsystem

Low Coupling:
- Minimale Abhängigkeiten
- Erhöht Wiederverwendbarkeit
- Erleichtert Änderungen

High Cohesion:
- Fokussierte Verantwortlichkeiten
- Zusammengehörige Funktionalität
- Wartbare Klassen

Design nach GRASP

General Responsibility Assignment Software Patterns:

Grundprinzipien:
- **Information Expert:** Verantwortlichkeit dort, wo die Information liegt
- **Creator:** Objekterstellung durch eng verbundene Klassen
- **Controller:** Koordination von Systemoperationen
- **Low Coupling:** Minimale Abhängigkeiten zwischen Klassen
- **High Cohesion:** Starker innerer Zusammenhang in Klassen

Erweiterte Prinzipien:
- **Polymorphism:** Typenabhängiges Verhalten durch Polymorphie
- **Pure Fabrication:** Hilfsklassen für besseres Design
- **Indirection:** Vermittler für lose Kopplung
- **Protected Variations:** Kapselung von Änderungen

Responsibility Driven Design

Designansatz basierend auf Verantwortlichkeiten und Kollaborationen:

Verantwortlichkeiten:
- **Doing:**
  - Aktionen ausführen
  - Berechnungen durchführen
  - Andere Objekte steuern
- **Knowing:**
  - Eigene Daten kennen
  - Verwandte Objekte kennen
  - Berechnete Informationen

Kollaborationen:
- Klare Rollen definieren
- Aufgaben verteilen
- Interfaces abstimmen

Design Pattern Kategorien

Bewährte Lösungsmuster für wiederkehrende Designprobleme:

Erzeugungsmuster (Creational):
- Abstract Factory: Familien verwandter Objekte
- Factory Method: Objekterzeugung in Subklassen
- Singleton: Genau eine Instanz
- Builder: Komplexe Objektkonstruktion
- Prototype: Klonen existierender Objekte

Strukturmuster (Structural):
- Adapter: Schnittstellen anpassen
- Bridge: Implementation von Abstraktion trennen
- Composite: Teil-Ganzes Hierarchien
- Decorator: Dynamische Funktionserweiterung
- Facade: Vereinfachte Schnittstelle
- Proxy: Kontrollierter Zugriff

Verhaltensmuster (Behavioral):
- Command: Anfrage als Objekt
- Observer: Ereignisbenachrichtigung
- Strategy: Austauschbare Algorithmen
- Template Method: Algorithmus-Skelett
- State: Zustandsabhängiges Verhalten
- Visitor: Operation zu Objektstruktur hinzufügen

Architekturmuster

Übersicht Architekturmuster

Grundlegende Architekturmuster für Software-Systeme:

- **Layered Pattern:**
  - Strukturierung in horizontale Schichten
  - Klare Trennung der Verantwortlichkeiten
  - Abhängigkeiten nur nach unten
- **Client-Server Pattern:**
  - Verteilung von Diensten
  - Zentralisierte Ressourcen
  - Mehrere Clients pro Server
- **Master-Slave Pattern:**
  - Verteilung von Aufgaben
  - Zentrale Koordination
  - Parallelverarbeitung
- **Pipe-Filter Pattern:**
  - Datenstromverarbeitung
  - Verkettung von Operationen
  - Wiederverwendbare Filter
- **Broker Pattern:**
  - Vermittlung zwischen Komponenten
  - Entkopplung von Diensten
  - Zentrale Koordination
- **Event-Bus Pattern:**
  - Asynchrone Kommunikation
  - Publisher-Subscriber Modell
  - Lose Kopplung
- **MVC Pattern:**
  - Trennung von Daten, Präsentation und Logik
  - Wiederverwendbare Komponenten
  - Klare Strukturierung

Schichtenarchitektur (Layered Architecture)

Organisation des Systems in hierarchische Schichten:

Typische Schichten:

- Präsentationsschicht (UI)
- Anwendungsschicht (Application Logic)
- Geschäftslogikschicht (Domain Logic)
- Datenzugriffsschicht (Data Access)

Prinzipien:

- Schichten kommunizieren nur mit direkten Nachbarn
- Abhängigkeiten nur nach unten
- Jede Schicht kapselt ihre Implementierung
- Höhere Schichten sind von unteren abhängig

```java
// Praesentationsschicht
public class CustomerController {
    private CustomerService service;

    public CustomerDTO getCustomer(String id) {
        return service.findCustomer(id);
    }
}

// Anwendungsschicht
public class CustomerService {
    private CustomerRepository repository;

    public CustomerDTO findCustomer(String id) {
        Customer customer = repository.findById(id);
        return CustomerDTO.from(customer);
    }
}

// Geschaeftslogikschicht
public class Customer {
    private CustomerId id;
    private String name;

    public void updateName(String newName) {
        validateName(newName);
        this.name = newName;
    }
}

// Datenzugriffsschicht
public class CustomerRepository {
    public Customer findById(String id) {
        // Datenbankzugriff
    }
}
```

Clean Architecture

Architektur-Prinzipien nach Robert C. Martin:

Hauptprinzipien:

- Unabhängigkeit von Frameworks
- Unabhängigkeit von UI
- Unabhängigkeit von Datenbank
- Testbarkeit ohne externe Systeme

Schichten (von innen nach außen):

- **Entities:**
  - Zentrale Geschäftsregeln
  - Unternehmensweit gültig
  - Höchste Stabilität
- **Use Cases:**
  - Anwendungsspezifische Geschäftsregeln
  - Orchestrierung der Entities
  - Anwendungslogik
- **Interface Adapters:**
  - Konvertierung von Daten
  - Präsentation und Controller
  - Gateway-Implementierungen
- **Frameworks & Drivers:**
  - UI-Framework
  - Datenbank
  - Externe Schnittstellen

```java
// Entity (innerste Schicht)
public class Customer {
    private CustomerId id;
    private String name;

    public void validateName(String name) {
        // Domaenenregeln fuer Namen
    }
}

// Use Case (Business Rules)
public class RegisterCustomerUseCase {
    public void execute(RegisterCustomerCommand cmd) {
        Customer customer = new
            Customer(cmd.getName());
        customer.validateName(cmd.getName());
        repository.save(customer);
    }
}

// Interface Adapter
public class CustomerController {
    private RegisterCustomerUseCase useCase;

    public ResponseEntity<CustomerDTO> register(
            CustomerRequest request) {
        useCase.execute(
            new
                RegisterCustomerCommand(request.getName())
        );
        return ResponseEntity.ok().build();
    }
}
```

Microservices Architektur

Verteilte Architektur mit unabhängigen Services:

Charakteristiken:

- Unabhängig entwickelbar und deploybar
- Eigene Datenhaltung pro Service
- Lose Kopplung
- API-basierte Kommunikation

Patterns:

- Service Discovery
- API Gateway
- Circuit Breaker
- Event Sourcing
- CQRS (Command Query Responsibility Segregation)

```java
@Service
public class OrderService {
    private final CustomerClient customerClient;
    private final PaymentClient paymentClient;

    @CircuitBreaker(name = "order")
    public OrderResult createOrder(OrderRequest
         request) {
        // Kundeninformationen laden
        CustomerInfo customer =
            customerClient.getCustomer(request.getCustomerI

        // Zahlungsabwicklung
        PaymentResult payment =
            paymentClient.processPayment(request.getAmount(

        // Order erstellen
        return createOrderWithPayment(customer,
            payment);
    }
}
```

Event-Driven Architecture (EDA)

Architekturstil basierend auf der Erzeugung, Erkennung und Verarbeitung von Events:

Kernkomponenten:

- **Event Producer:** Erzeugt Events
- **Event Channel:** Transportiert Events
- **Event Consumer:** Verarbeitet Events
- **Event Processor:** Transformiert Events

```java
// Event Definition
public class OrderCreatedEvent {
    private final OrderId orderId;
    private final CustomerId customerId;
    private final Money totalAmount;
    private final LocalDateTime timestamp;
}

// Event Producer
@Service
public class OrderService {
    private final EventPublisher eventPublisher;

    public Order createOrder(OrderRequest request) {
        Order order = orderRepository.save(
            new Order(request));

        eventPublisher.publish(new OrderCreatedEvent(
            order.getId(),
            order.getCustomerId(),
            order.getTotalAmount(),
            LocalDateTime.now()
        ));

        return order;
    }
}

// Event Consumer
@Service
public class NotificationService {
    @EventListener
    public void handleOrderCreated(
            OrderCreatedEvent event) {
        sendConfirmationEmail(event.getCustomerId());
    }
}
```

Integration Patterns

Muster für die Integration verschiedener Systeme:

Hauptkategorien:

- **File Transfer:**
  - Datenaustausch über Dateien
  - Batch-Verarbeitung
  - Einfache Integration
- **Shared Database:**
  - Gemeinsame Datenbasis
  - Direkte Integration
  - Hohe Kopplung
- **Remote Procedure Call:**
  - Synchrone Kommunikation
  - Direkter Methodenaufruf
  - Service-Orientierung
- **Messaging:**
  - Asynchrone Kommunikation
  - Message Broker
  - Lose Kopplung

Spezifische Patterns:

- Message Router
- Message Translator
- Message Filter
- Content Enricher
- Message Store

UML-Modellierung

Grundlagen der UML-Modellierung

UML (Unified Modeling Language) wird im Design auf zwei Arten verwendet:

Statische Modelle:

- Struktur des Systems
- Klassendiagramme, Paketdiagramme
- Fokus auf Pakete, Klassen, Attribute
- Keine Methodenimplementierung

Dynamische Modelle:

- Verhalten des Systems
- Sequenz-, Zustands-, Aktivitätsdiagramme
- Fokus auf Logik und Verhalten
- Methodenimplementierung

UML Diagrammtypen

Klassendiagramm:

- Klassen mit Attributen und Methoden
- Beziehungen zwischen Klassen
- Vererbung und Implementierung
- Multiplizitäten und Rollen

Sequenzdiagramm:

- Zeitlicher Ablauf von Interaktionen
- Nachrichtenaustausch zwischen Objekten
- Synchrone und asynchrone Kommunikation
- Alternative Abläufe und Schleifen

Zustandsdiagramm:

- Zustandsübergänge eines Objekts
- Events und Guards
- Composite States
- Entry/Exit Actions

Aktivitätsdiagramm:

- Ablauf von Geschäftsprozessen
- Kontrollfluss und Datenfluss
- Parallelität und Synchronisation
- Swimlanes für Verantwortlichkeiten

Statische vs. Dynamische Modelle

UML bietet verschiedene Diagrammtypen für unterschiedliche Aspekte:

Statische Modelle:

- Fokus auf Struktur und Beziehungen
- UML-Klassendiagramm für Klassen, Attribute, Methoden
- UML-Paketdiagramm für Modularisierung
- UML-Komponentendiagramm für Systembausteine
- UML-Verteilungsdiagramm für Deployment

Dynamische Modelle:

- Fokus auf Verhalten und Interaktion
- UML-Sequenzdiagramm für Abläufe
- UML-Aktivitätsdiagramm für Prozesse
- UML-Zustandsdiagramm für Objektzustände
- UML-Kommunikationsdiagramm für Objektkollaborationen

UML im Design

Klassendiagramm für Order Management:

```java
public class Order {
    private OrderId id;
    private Customer customer;
    private List<OrderLine> lines;
    private OrderStatus status;

    public Money calculateTotal() {
        return lines.stream()
                    .map(OrderLine::getSubTotal)
                    .reduce(Money.ZERO, Money::add);
    }

    public void addProduct(Product product, int qty) {
        lines.add(new OrderLine(product, qty));
    }
}

public class OrderLine {
    private Product product;
    private int quantity;

    public Money getSubTotal() {
        return product.getPrice()
                      .multiply(quantity);
    }
}

@Service
public class OrderService {
    private OrderRepository repository;

    public Order createOrder(OrderRequest request) {
        Order order = new
             Order(request.getCustomerId());
        request.getItems().forEach(item ->
            order.addProduct(item.getProduct(),
                             item.getQuantity()));
        return repository.save(order);
    }
}
```

Sequenzdiagramm für Bestellprozess

Implementierung einer Bestellverarbeitung:

```java
@RestController
public class OrderController {
    private final OrderService orderService;
    private final PaymentService paymentService;

    public OrderResponse createOrder(
            OrderRequest request) {
        // Validiere Bestellung
        validateOrder(request);

        // Erstelle Order
        Order order =
             orderService.createOrder(request);

        // Prozessiere Zahlung
        PaymentResult payment =
            paymentService.processPayment(
                order.getId(),
                order.getTotal()
            );

        // Update Order Status
        if (payment.isSuccessful()) {
            order.confirm();
            orderService.save(order);
        }

        return OrderResponse.from(order);
    }
}
```

Zustandsdiagramm für Bestellstatus

Implementation des State Patterns:

```java
public interface OrderState {
    void process(Order order);
    void cancel(Order order);
    void ship(Order order);
}

public class NewOrderState implements OrderState {
    @Override
    public void process(Order order) {
        validateOrder(order);
        order.setState(new ProcessingState());
    }

    @Override
    public void cancel(Order order) {
        order.setState(new CancelledState());
    }

    @Override
    public void ship(Order order) {
        throw new IllegalStateException(
            "Cannot ship new order");
    }
}

public class Order {
    private OrderState state;

    public void process() {
        state.process(this);
    }

    void setState(OrderState newState) {
        this.state = newState;
    }
}
```

UML Diagrammauswahl

Auswahlkriterien:

1. **Ziel der Modellierung**
   - Struktur darstellen -> Klassendiagramm
   - Abläufe zeigen -> Sequenzdiagramm
   - Zustände dokumentieren -> Zustandsdiagramm
   - Prozesse beschreiben -> Aktivitätsdiagramm
2. **Zielgruppe**
   - Entwickler -> detaillierte technische Diagramme
   - Stakeholder -> vereinfachte Übersichtsdiagramme
   - Architekten -> Architekturdiagramme
3. **Detailgrad**
   - Überblick -> wenige wichtige Elemente
   - Detaildesign -> vollständige Details
   - Implementation -> code-nahe Darstellung
4. **Phase im Projekt**
   - Analyse -> konzeptuelle Modelle
   - Design -> Designmodelle
   - Implementation -> detaillierte Modelle

Aktivitätsdiagramm für Geschäftsprozess

Implementation eines Workflow:

```java
public class OrderProcessor {
    public void processOrder(Order order) {
        // Parallel processing
        CompletableFuture.allOf(
            validateInventory(order),
            validatePayment(order)
        ).thenRun(() -> {
            if (order.isValid()) {
                fulfillOrder(order);
            } else {
                handleValidationFailure(order);
            }
        });
    }

    private CompletableFuture<Void> validateInventory(
            Order order) {
        return CompletableFuture.runAsync(() -> {
            order.getItems().forEach(item -> {
                if
                    (!inventoryService.isAvailable(item))
                    {
                    throw new
                        OutOfStockException(item);
                }
            });
        });
    }
}
```

other examples

Gute Testbarkeit

```java
public class OrderService {
    private final OrderRepository repository;
    private final PaymentGateway paymentGateway;

    // Dependency Injection ermoeglicht einfaches Mocking
    public OrderService(
            OrderRepository repository,
            PaymentGateway paymentGateway) {
        this.repository = repository;
        this.paymentGateway = paymentGateway;
    }

    // Klare Methoden-Verantwortlichkeiten
    public OrderResult createOrder(OrderRequest request) {
        validateRequest(request);
        Order order = createOrderEntity(request);
        PaymentResult payment = processPayment(order);
        return createOrderResult(order, payment);
    }
}
```

Dokumentation Architektur

Architekturanalyse

Analyse für ein E-Commerce-System:

```java
// Dokumentation der Analyse
public class ArchitectureAnalysis {
    public class QualityRequirement {
        String name;
        String description;
        int priority;
        String measurementCriteria;
    }

    public class ArchitecturalConstraint {
        String type; // Technical, Organizational, Business
        String description;
        String impact;
    }

    // Beispiel Qualitaetsanforderung
    QualityRequirement performance = new QualityRequirement(
        "Response Time",
        "System responses within 200ms",
        1,
        "95th percentile < 200ms"
    );

    // Beispiel Randbedingung
    ArchitecturalConstraint technology = new ArchitecturalConstraint(
        "Technical",
        "Must use Java 17",
        "Affects framework selection"
    );
}
```

Architektur-Entscheidungen

Entscheidungsdokumentation:

```java
public class ArchitectureDecision {
    String id;
    String title;
    String context;
    String decision;
    String rationale;
    List<String> consequences;
    List<Alternative> alternatives;

    class Alternative {
        String description;
        List<String> pros;
        List<String> cons;
        String rejectionReason;
    }
}

// Beispiel:
ArchitectureDecision caching = new ArchitectureDecision(
    "AD001",
    "Caching Strategy",
    "High read load on product catalog",
    "Use Redis as distributed cache",
    "Better performance and scalability",
    List.of("Requires Redis expertise", "Additional infrastructure"),
    List.of(new Alternative(
        "In-memory cache",
        List.of("Simple", "No additional infrastructure"),
        List.of("Not distributed", "Memory limited"),
        "Doesn't scale horizontally"
    ))
);
```

Architektur-Review

Review-Protokoll:

```java
public class ArchitectureReview {
    public class Finding {
        String area;
        String observation;
        Risk risk;
        String recommendation;
        Priority priority;
    }

    public class Action {
        String description;
        String responsible;
        LocalDate dueDate;
        Status status;
    }

    List<Finding> findings = List.of(
        new Finding(
            "Security",
            "Missing rate limiting",
            Risk.HIGH,
            "Implement API gateway with rate limiting",
            Priority.HIGH
        )
    );

    List<Action> actions = List.of(
        new Action(
            "Implement API gateway",
            "Team A",
            LocalDate.now().plusWeeks(2),
            Status.OPEN
        )
    );
}
```