Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Softwarearchitektur und Design

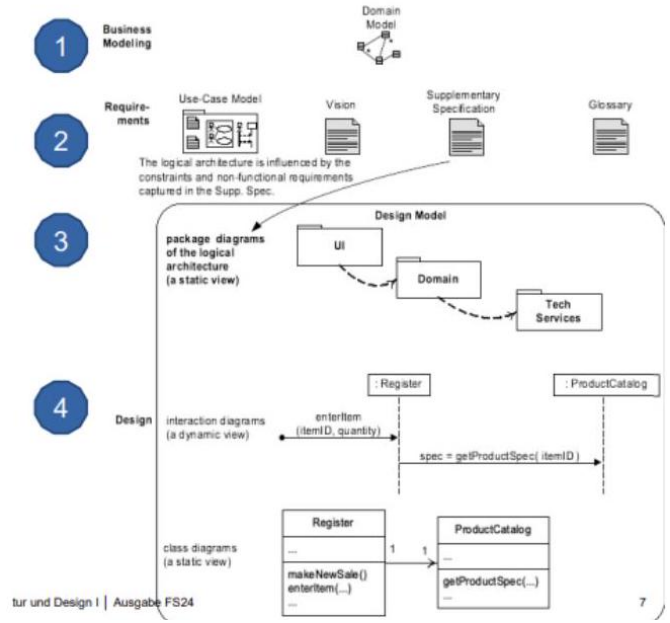
Grundlagen und Überblick

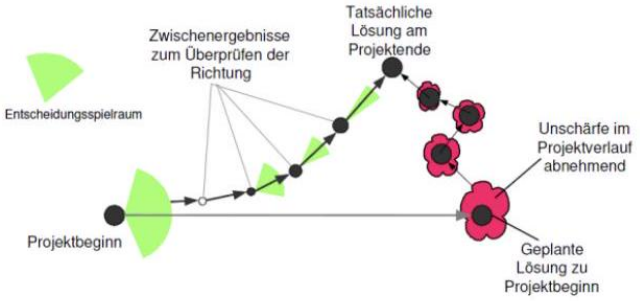
Grundlagen und Überblick

- **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





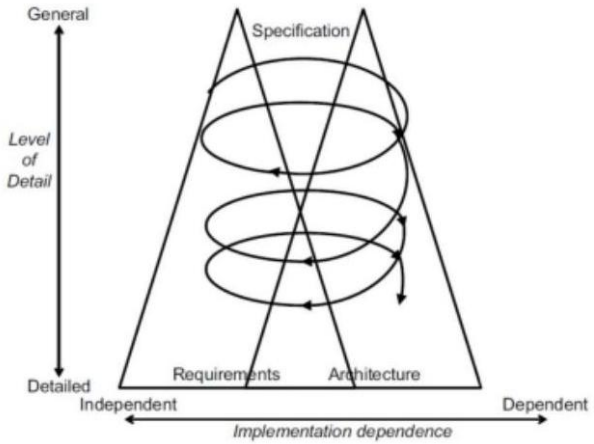
Softwarearchitektur Die Architektur eines Softwaresystems definiert:

- **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:

- Hierarchische 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

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)

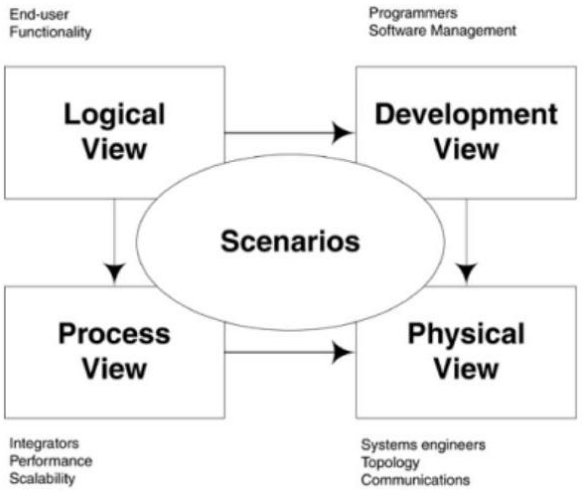
Architektursichten (4+1 View Model)

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



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

Architekturprozess und Best Practices

Best Practices im Architekturentwurf

1. Analyse und Planung

- Anforderungen priorisieren
- Qualitätsziele definieren
- Constraints identifizieren
- Stakeholder einbinden

2. Design-Prinzipien

- Separation of Concerns
- Single Responsibility
- Information Hiding
- Don't Repeat Yourself (DRY)

3. Strukturierung

- Klare Schichtenarchitektur
- Definierte Schnittstellen
- Lose Kopplung
- Hohe Kohäsion

4. Dokumentation

- Architekturentscheidungen
- Begründungen
- Alternativen
- Trade-offs

Gesamter Architekturprozess

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

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

Gesamter Architekturprozess

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

Architekturanalyse

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

Architektur-Entscheidungen

Architekturentwurf

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

Architekturentwurf

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-Review

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

Architektur-Evaluation

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

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

Clean Architecture

Prinzipien nach Robert C. Martin:

- **Unabhängigkeit von Frameworks**
  - Framework als Tool, nicht als Einschränkung
  - Geschäftslogik unabhängig von UI/DB
- **Testbarkeit**
  - Business Rules ohne externe Systeme testbar
  - Keine DB/UI für Tests notwendig
- **Unabhängigkeit von UI**
  - UI austauschbar ohne Business Logic Änderung
  - Web, Desktop, Mobile möglich
- **Unabhängigkeit von Datenbank**
  - DB-System austauschbar
  - Business Rules unabhängig von Datenpersistenz

Schichten von außen nach innen:

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

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.

Schichtenarchitektur (Layered Architecture)

Strukturierung eines Systems in horizontale Schichten:

Typische Schichten:

- Präsentationsschicht (UI)
- Anwendungsschicht (Application)
- Geschäftslogikschicht (Domain)
- Datenzugriffsschicht (Persistence)

Regeln:

- Kommunikation nur mit angrenzenden Schichten
- Abhängigkeiten nur nach unten
- Jede Schicht kapselt ihre Implementierung

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

```
// 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
    }
}
```

Schichtenarchitektur Implementation

Beispiel einer typischen Schichtenstruktur:

```java
// Presentation Layer
public class CustomerController {
    private CustomerService service;

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

// Application Layer
public class CustomerService {
    private CustomerRepository repository;

    public Customer findById(String id) {
        validateId(id);
        return repository.findById(id)
            .orElseThrow(CustomerNotFoundException::new);
    }
}

// Domain Layer
public class Customer {
    private CustomerId id;
    private String name;
    private Address address;

    public void updateAddress(Address newAddress) {
        validateAddress(newAddress);
        this.address = newAddress;
    }
}

// Persistence Layer
public class CustomerRepository {
    private JpaRepository<Customer, CustomerId>
        jpaRepo;

    public Optional<Customer> findById(String id) {
        return jpaRepo.findById(new CustomerId(id));
    }
}
```

Client-Server Architektur

Verteilung von Funktionalitäten zwischen Client und Server:

Charakteristiken:

- Klare Trennung von Zuständigkeiten
- Zentralisierte Ressourcenverwaltung
- Skalierbarkeit durch Server-Erweiterung
- Verschiedene Client-Typen möglich

Varianten:

- Thin Client: Minimale Client-Logik
- Rich Client: Komplexe Client-Funktionalität
- Web Client: Browser-basiert
- Mobile Client: Für mobile Geräte optimiert

Objektorientiertes Design

Objektorientiertes Design

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

GRASP in der Praxis

```java
// Information Expert: Order kennt seine Details
public class Order {
    private List<OrderLine> lines;

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

// Controller: Koordiniert Use Case
public class OrderController {
    private OrderService service;

    public OrderResponse
        createOrder(OrderRequest request) {
        // Koordination der Verarbeitung
        Order order =
            service.createOrder(request);
        return OrderResponse.from(order);
    }
}

// Protected Variations: Abstraktion von
    Implementierung
public interface PaymentGateway {
    PaymentResult process(Money amount);
}

public class StripePaymentGateway implements
    PaymentGateway {
    public PaymentResult process(Money amount) {
        // Stripe-spezifische Implementierung
    }
}
```

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

Design Patterns in der Architektur

Model-View-Controller (MVC)

Trennt Anwendung in drei Hauptkomponenten:

- **Model:** Geschäftslogik und Daten
- **View:** Darstellung der Daten
- **Controller:** Steuerung und Koordination

```
// Model
public class CustomerModel {
    private String name;
    private List<Order> orders;

    public void addOrder(Order order) {
        orders.add(order);
        notifyViews();
    }
}

// View
public class CustomerView {
    private CustomerModel model;

    public void displayCustomerInfo() {
        System.out.println("Customer: " +
            model.getName());
        System.out.println("Orders: " +
            model.getOrders().size());
    }
}

// Controller
public class CustomerController {
    private CustomerModel model;
    private CustomerView view;

    public void createOrder(OrderData data) {
        Order order = new Order(data);
        model.addOrder(order);
        view.displayCustomerInfo();
    }
}
```

Microservices und Event-Driven Architecture

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)

```
@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.getCustomerId

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

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

Microservices Architektur

Grundprinzipien:

- Unabhängig deploybare Services
- Lose Kopplung
- Eigene Datenhaltung pro Service
- REST/Message-basierte Kommunikation

Vorteile:

- Bessere Skalierbarkeit
- Unabhängige Entwicklung
- Technologiefreiheit
- Robustheit

Herausforderungen:

- Verteilte Transaktionen
- Service Discovery
- Datenkonvergenz
- Monitoring

Microservice Design

Service für Benutzerprofile:

```java
@RestController
@RequestMapping("/api/users")
public class UserProfileController {
    private final UserService userService;

    @GetMapping("/{id}")
    public UserProfileDTO getProfile(@PathVariable
         String id) {
        UserProfile profile = userService.findById(id);
        return UserProfileDTO.from(profile);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Void> updateProfile(
            @PathVariable String id,
            @RequestBody UpdateProfileCommand command)
                {
        userService.updateProfile(id, command);
        return ResponseEntity.ok().build();
    }
}

// Event fuer andere Services
public class UserProfileUpdatedEvent {
    private final String userId;
    private final String newEmail;
    private final LocalDateTime timestamp;

    // Konstruktor und Getter
}
```

Microservices Design Prinzipien

1. Service Boundaries

- Nach Business Capabilities trennen
- Bounded Context (DDD) beachten
- Datenhoheit festlegen

2. Service Kommunikation

- Synchron vs. Asynchron
- Event-Driven Design
- API Gateway Pattern

3. Datenmanagement

- Database per Service
- Event Sourcing
- CQRS Pattern

4. Resilience

- Circuit Breaker
- Bulkhead Pattern
- Fallback Mechanismen

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

Event-Driven Architecture (EDA)

Basiert auf der Produktion, Erkennung und Reaktion auf Events:

Komponenten:

- Event Producer
- Event Channel
- Event Consumer
- Event Processor

```java
// Event Definition
public class OrderCreatedEvent {
    private final String orderId;
    private final LocalDateTime timestamp;
    private final BigDecimal totalAmount;
}

// Event Producer
public class OrderService {
    private EventBus eventBus;

    public void createOrder(OrderData data) {
        Order order = orderRepository.save(data);
        OrderCreatedEvent event = new
             OrderCreatedEvent(
            order.getId(),
            LocalDateTime.now(),
            order.getTotalAmount()
        );
        eventBus.publish(event);
    }
}

// Event Consumer
@EventListener
public class InvoiceGenerator {
    public void handleOrderCreated(OrderCreatedEvent
         event) {
        generateInvoice(event.getOrderId());
    }
}
```

Architektur-Dokumentation

1. Überblick

- Systemkontext
- Hauptkomponenten
- Technologie-Stack

2. Architektur-Entscheidungen

- Begründungen
- Alternativen
- Trade-offs

3. Technische Konzepte

- Persistenz
- Sicherheit
- Integration
- Deployment

4. Qualitätsszenarien

- Performance
- Skalierbarkeit
- Verfügbarkeit
- Wartbarkeit

Architektur-Dokumentation: REST API

API-Design und Dokumentation:

```java
@RestController
@RequestMapping("/api/v1/orders")
public class OrderController {

    @GetMapping("/{id}")
    @Operation(summary = "Get order by ID",
               description = "Returns detailed order
                    information")
    @ApiResponses({
        @ApiResponse(responseCode = "200",
                     description = "Order found"),
        @ApiResponse(responseCode = "404",
                     description = "Order not found")
    })
    public OrderDTO getOrder(@PathVariable String id) {
        return orderService.findById(id)
                           .map(OrderDTO::from)
                           .orElseThrow(OrderNotFoundExcepti
    }
}
```

Qualitätsszenarien:

- Response Time $<$ 200ms (95. Perzentil)
- Verfügbarkeit 99.9
- Maximal 1000 req/s pro Instance
- Automatische Skalierung ab 70

Integrationsmuster

Integration Patterns

Muster für die Integration von Systemen und Services:

Hauptkategorien:

- **File Transfer:** Datenaustausch über Dateien
- **Shared Database:** Gemeinsame Datenbasis
- **Remote Procedure Call:** Direkter Methodenaufruf
- **Messaging:** Nachrichtenbasierte Kommunikation

Messaging Pattern Implementation

Message Producer und Consumer:

```java
// Message Definition
public class OrderMessage {
    private String orderId;
    private String customerId;
    private BigDecimal amount;
    private OrderStatus status;
}

// Message Producer
public class OrderProducer {
    private MessageQueue messageQueue;

    public void sendOrderCreated(Order order) {
        OrderMessage message = new OrderMessage(
            order.getId(),
            order.getCustomerId(),
            order.getAmount(),
            OrderStatus.CREATED
        );
        messageQueue.send("orders", message);
    }
}

// Message Consumer
public class OrderProcessor {
    @MessageListener(queue = "orders")
    public void processOrder(OrderMessage message) {
        if (message.getStatus() ==
             OrderStatus.CREATED) {
            processNewOrder(message);
        }
    }

    private void processNewOrder(OrderMessage message)
         {
        // Verarbeitung der Bestellung
        validateOrder(message);
        updateInventory(message);
        notifyCustomer(message);
    }
}
```

API Gateway Pattern

Zentraler Einstiegspunkt für Client-Anfragen:

Verantwortlichkeiten:

- Routing von Anfragen
- Authentifizierung/Autorisierung
- Last-Verteilung
- Caching
- Monitoring
- API-Versionierung

```java
@Component
public class ApiGateway {
    private final AuthService authService;
    private final ServiceRegistry registry;

    @GetMapping("/api/v1/**")
    public ResponseEntity<Object> routeRequest(
            HttpServletRequest request,
            @RequestHeader("Authorization") String
                token) {

        // Authentifizierung
        if (!authService.validateToken(token)) {
            return ResponseEntity.status(401).build();
        }

        // Service Discovery
        String serviceName =
            extractServiceName(request);
        ServiceInstance instance =
            registry.getInstance(serviceName);

        // Request Weiterleitung
        return forwardRequest(instance, request);
    }
}
```

API Design Best Practices

1. Ressourcen-Orientierung

- Klare Ressourcen-Namen
- Hierarchische Struktur
- Korrekte HTTP-Methoden

2. Versionierung

- Explizite Versions-Nummer
- Abwärtskompatibilität
- Migrations-Strategie

3. Fehlerbehandlung

- Standardisierte Fehler-Formate
- Aussagekräftige Fehlermeldungen
- Korrekte HTTP-Status-Codes

4. Dokumentation

- OpenAPI/Swagger
- Beispiele und Use Cases
- Fehlerszenarien

REST API Design

Ressourcen-Design für E-Commerce System:

```java
@RestController
@RequestMapping("/api/v1")
public class ProductController {

    // Collection Resource
    @GetMapping("/products")
    public PagedResponse<ProductDTO> getProducts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "20") int
                size) {
        return productService.findAll(page, size);
    }

    // Single Resource
    @GetMapping("/products/{id}")
    public ProductDTO getProduct(@PathVariable String
        id) {
        return productService.findById(id);
    }

    // Sub-Resource Collection
    @GetMapping("/products/{id}/reviews")
    public List<ReviewDTO> getProductReviews(
            @PathVariable String id) {
        return reviewService.findByProductId(id);
    }

    // Error Handling
    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse>
        handleNotFound(
            ProductNotFoundException ex) {
        ErrorResponse error = new ErrorResponse(
            "PRODUCT_NOT_FOUND",
            ex.getMessage()
        );
        return ResponseEntity.status(404).body(error);
    }
}
```

Design Pattern Anwendung

Systematisches Vorgehen bei der Pattern-Auswahl:

1. Problem analysieren

- Kernproblem identifizieren
- Qualitätsanforderungen beachten
- Kontext verstehen

2. Pattern auswählen

- Passende Pattern-Kategorie wählen
- Alternativen evaluieren
- Trade-offs abwägen

3. Pattern implementieren

- Struktur übertragen
- An Kontext anpassen
- Auf GRASP-Prinzipien achten

Factory Method Pattern

Problem: Objekterzeugung soll flexibel und erweiterbar sein.

```java
// Creator
public abstract class DocumentCreator {
    public abstract Document createDocument();

    public void openDocument() {
        Document doc = createDocument();
        doc.open();
    }
}

// Concrete Creator
public class PDFDocumentCreator extends
    DocumentCreator {
    @Override
    public Document createDocument() {
        return new PDFDocument();
    }
}

// Product Interface
public interface Document {
    void open();
    void save();
}

// Concrete Product
public class PDFDocument implements Document {
    @Override
    public void open() {
        // PDF-spezifische Implementation
    }

    @Override
    public void save() {
        // PDF-spezifische Implementation
    }
}
```

Strategy Pattern

Problem: Algorithmus soll zur Laufzeit austauschbar sein.

```java
// Strategy Interface
public interface PaymentStrategy {
    void pay(Money amount);
}

// Concrete Strategies
public class CreditCardStrategy implements
    PaymentStrategy {
    private String cardNumber;

    @Override
    public void pay(Money amount) {
        // Kreditkarten-Zahlung
    }
}

public class PayPalStrategy implements
    PaymentStrategy {
    private String email;

    @Override
    public void pay(Money amount) {
        // PayPal-Zahlung
    }
}

// Context
public class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void
        setPaymentStrategy(PaymentStrategy
        strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(Money amount) {
        paymentStrategy.pay(amount);
    }
}
```

Observer Pattern

Problem: Objekte sollen über Änderungen informiert werden.

```java
// Observer Interface
public interface OrderObserver {
    void onOrderStateChange(Order order);
}

// Concrete Observer
public class EmailNotifier implements
    OrderObserver {
    @Override
    public void onOrderStateChange(Order order)
        {
        sendEmail(order.getCustomer(),
                  "Order status: " +
                      order.getStatus());
    }
}

// Observable
public class Order {
    private List<OrderObserver> observers = new
        ArrayList<>();
    private OrderStatus status;

    public void addObserver(OrderObserver
        observer) {
        observers.add(observer);
    }

    public void setStatus(OrderStatus
        newStatus) {
        this.status = newStatus;
        notifyObservers();
    }

    private void notifyObservers() {
        observers.forEach(o ->
            o.onOrderStateChange(this));
    }
}
```

UML Modellierung im Design

Einsatz verschiedener UML-Diagramme im Design-Prozess:

1. Klassendiagramm
- Design der Klassenstruktur
- Beziehungen zwischen Klassen
- Attribute und Methoden
- Pattern-Strukturen

2. Sequenzdiagramm
- Interaktion zwischen Objekten
- Methodenaufrufe
- Zeitliche Abfolge
- Use-Case Realisierung

3. Zustandsdiagramm
- Objektzustände
- Zustandsübergänge
- Ereignisse und Aktionen
- Lifecycle Modellierung

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);
                }
            });
        });
    }
}
```

UML-Modellierung

Statische vs. Dynamische Modelle

Statische Modelle (Struktur):

- UML-Klassendiagramm
- Fokus auf Pakete, Klassen, Attribute
- Keine Methodenimplementierung

Dynamische Modelle (Verhalten):

- UML-Interaktionsdiagramme
- Fokus auf Logik und Verhalten
- Implementierung der Methoden

UML-Diagrammtypen

1. Klassendiagramm:

- Klassen und aktive Klassen
- Attribute und Operationen
- Sichtbarkeiten und Beziehungen
- Interfaces und Realisierungen

2. Sequenzdiagramm:

- Lebenslinien und Nachrichten
- Synchrone/Asynchrone Kommunikation
- Aktivierung und Deaktivierung
- Alternative Abläufe

3. Zustandsdiagramm:

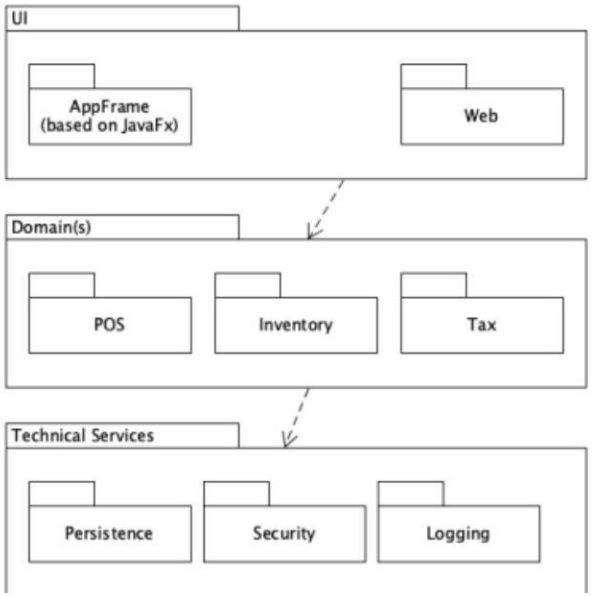
- Zustände und Übergänge
- Start- und Endzustände
- Composite States
- Historie und Parallelität

4. Aktivitätsdiagramm:

- Aktionen und Aktivitäten
- Kontroll- und Datenflüsse
- Verzweigungen und Zusammenführungen
- Partitionen (Swimlanes)

UML-Paketdiagramm:

- Definition von Teilsystemen
- Gruppierung von Elementen
- Abhängigkeiten zwischen Paketen



UML Diagrammauswahl

Entscheidungshilfe für die Wahl des UML-Diagrammtyps:

1. Strukturbeschreibung benötigt:

- Klassendiagramm für Typen und Beziehungen
- Paketdiagramm für Modularisierung
- Komponentendiagramm für Bausteinsicht
- Verteilungsdiagramm für Deployment

2. Verhaltensbeschreibung benötigt:

- Sequenzdiagramm für Interaktionsabläufe
- Aktivitätsdiagramm für Workflows
- Zustandsdiagramm für Objektlebenszyklen
- Kommunikationsdiagramm für Objektkollaborationen

3. Abstraktionsebene wählen:

- Analyse: Konzeptuelle Diagramme
- Design: Detaillierte Spezifikation
- Implementation: Codenahes Design

Responsibility Driven Design (RDD)

Design basierend auf Verantwortlichkeiten:

- Klassenentwurf nach Rollen
- Kollaborationsbeziehungen
- Implementierung durch Attribute/Methoden
- Anwendbar auf allen Ebenen

Prüfungsaufgabe: UML-Modellierung

Aufgabe: Modellieren Sie für ein Bibliothekssystem die Ausleihe eines Buches mit:

- Klassendiagramm der beteiligten Klassen
- Sequenzdiagramm des Ausleihvorgangs
- Zustandsdiagramm für ein Buchexemplar

Bewertungskriterien:

- Korrekte UML-Notation
- Vollständigkeit der Modellierung
- Konsistenz zwischen Diagrammen
- Angemessener Detaillierungsgrad

GRASP Prinzipien

General Responsibility Assignment Software Patterns:

- **Information Expert:** Verantwortung basierend auf Information
- **Creator:** Objekterstellung bei starker Beziehung
- **Controller:** Zentrale Steuerungslogik
- **Low Coupling:** Minimale Abhängigkeiten
- **High Cohesion:** Starker innerer Zusammenhang
- **Polymorphism:** Flexibilität durch Schnittstellen
- **Pure Fabrication:** Künstliche Klassen für besseres Design
- **Indirection:** Vermittler für Flexibilität
- **Protected Variations:** Kapselung von Änderungen

GRASP Anwendung

Szenario: Online-Shop Warenkorb-Funktionalität

GRASP-Prinzipien angewandt:

- **Information Expert:**
  - Warenkorb kennt seine Positionen
  - Berechnet selbst Gesamtsumme
- **Creator:**
  - Warenkorb erstellt Warenkorbpositionen
  - Bestellung erstellt aus Warenkorb
- **Controller:**
  - ShoppingController koordiniert UI und Domain
  - Keine Geschäftslogik im Controller
- **Low Coupling:**
  - UI kennt nur Controller
  - Domain unabhängig von UI

UML-Modellierung

UML Diagrammtypen Übersicht

UML bietet verschiedene Diagrammtypen für statische und dynamische Modellierung:

Statische Modelle:

- Klassendiagramm
- Paketdiagramm
- Komponentendiagramm
- Verteilungsdiagramm

Dynamische Modelle:

- Sequenzdiagramm
- Kommunikationsdiagramm
- Zustandsdiagramm
- Aktivitätsdiagramm

Klassendiagramm

Hauptelemente:

- **Klassen:**
  - Name der Klasse
  - Attribute mit Sichtbarkeit
  - Operationen mit Parametern
- **Beziehungen:**
  - Assoziation (normaler Pfeil)
  - Vererbung (geschlossener Pfeil)
  - Implementierung (gestrichelter Pfeil)
  - Aggregation (leere Raute)
  - Komposition (gefüllte Raute)
- **Interfaces:**
  - Stereotyp «interface»
  - Nur Methodensignaturen
  - Implementierungsbeziehung

Klassendiagramm: E-Commerce System

Domänenmodell mit wichtigen Beziehungen:

```java
public interface OrderRepository {
    Optional<Order> findById(OrderId id);
    void save(Order order);
}

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

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

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

    public Money getSubTotal() {
        return price.multiply(quantity);
    }
}
```

Sequenzdiagramm

Notationselemente:

- **Lebenslinien:**
  - Objekte als Rechtecke
  - Vertikale gestrichelte Linie
  - Aktivierungsbalken für Ausführung
- **Nachrichten:**
  - Synchron (durchgezogener Pfeil)
  - Asynchron (offener Pfeil)
  - Antwort (gestrichelter Pfeil)
  - Parameter und Rückgabewerte
- **Kontrollelemente:**
  - alt (Alternative)
  - loop (Schleife)
  - opt (Optional)
  - par (Parallel)

Sequenzdiagramm: Bestellprozess

Interaktion zwischen Komponenten:

```java
public class OrderService {
    private final OrderRepository orderRepo;
    private final PaymentService paymentService;

    public OrderConfirmation processOrder(OrderRequest
         request) {
        // Validiere Bestellung
        validateOrder(request);

        // Erstelle Order
        Order order = createOrder(request);
        orderRepo.save(order);

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

        // Bestaetige Bestellung
        if (result.isSuccessful()) {
            order.confirm();
            orderRepo.save(order);
            return new OrderConfirmation(order);
        }

        throw new PaymentFailedException();
    }
}
```

Zustandsdiagramm

Notationselemente:

- **Zustände:**
  - Startzustand (gefüllter Kreis)
  - Endzustand (Kreis mit Punkt)
  - Einfache Zustände (Rechteck)
  - Zusammengesetzte Zustände
- **Transitionen:**
  - Event [Guard] / Action
  - Interne Transitionen
  - Selbsttransitionen
- **Spezielle Elemente:**
  - History State (H)
  - Deep History (H*)
  - Entry/Exit Points
  - Choice Points

Zustandsdiagramm: Bestellstatus

Implementation eines 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;
    }
}
```

Aktivitätsdiagramm

Hauptelemente:

- **Aktionen:**
  - Atomare Aktionen
  - Call Behavior Action
  - Send/Receive Signal
- **Kontrollfluss:**
  - Verzweigungen (Diamond)
  - Parallelisierung (Balken)
  - Join/Merge Nodes
- **Strukturierung:**
  - Activity Partitions (Swimlanes)
  - Structured Activity Nodes
  - Interruptible Regions

Aktivitätsdiagramm: Bestellabwicklung

Implementation eines Geschäftsprozesses:

```java
public class OrderProcessor {
    public void processOrder(Order order) {
        // Parallele Verarbeitung
        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);
                }
            });
        });
    }
}
```

Verteilungsdiagramm

Elemente:

- **Nodes:**
  - Device Nodes
  - Execution Environment
  - Artifacts
- **Verbindungen:**
  - Kommunikationspfade
  - Protokolle
  - Multiplizitäten
- **Deployment:**
  - Deployment Specifications
  - Manifestationen

Verteilungsdiagramm: Microservice-Architektur

Deployment-Konfiguration:

```java
@Configuration
public class ServiceConfig {
    @Value("${service.host}")
    private String serviceHost;

    @Value("${service.port}")
    private int servicePort;

    @Bean
    public ServiceRegistry registry() {
        return ServiceRegistry.builder()
            .host(serviceHost)
            .port(servicePort)
            .healthCheck("/health")
            .build();
    }

    @Bean
    public LoadBalancer loadBalancer(
            ServiceRegistry registry) {
        return new RoundRobinLoadBalancer(registry);
    }
}
```

Kommunikationsdiagramm

Hauptelemente:

- **Objekte:**
  - Als Rechtecke dargestellt
  - Mit Objektname und Klasse
  - Verbunden durch Links
- **Nachrichten:**
  - Nummerierte Sequenz
  - Synchrone/Asynchrone Aufrufe
  - Parameter und Rückgabewerte
- **Steuerungselemente:**
  - Bedingte Nachrichten [condition]
  - Iterationen *
  - Parallele Ausführung ||

Kommunikationsdiagramm: Shopping Cart

Objektinteraktionen beim Checkout:

```java
public class ShoppingCart {
    private List<CartItem> items;
    private CheckoutService checkoutService;

    public Order checkout() {
        // 1: validateItems()
        validateItems();

        // 2: calculateTotal()
        Money total = calculateTotal();

        // 3: createOrder(items, total)
        Order order = checkoutService.createOrder(
            items, total);

        // 4: clearCart()
        items.clear();

        return order;
    }
}
```

Paketdiagramm

Elemente:

- **Pakete:**
  - Gruppierung von Modellelementen
  - Hierarchische Strukturierung
  - Namensräume
- **Abhängigkeiten:**
  - Import/Export von Elementen
  - «use» Beziehungen
  - Zugriffsrechte

UML Diagrammauswahl

Entscheidungshilfen für die Wahl des passenden Diagrammtyps:

1. Statische Struktur

- Klassendiagramm für Typen und Beziehungen
- Paketdiagramm für Modularisierung
- Komponentendiagramm für Bausteinsicht
- Verteilungsdiagramm für physische Verteilung

2. Dynamisches Verhalten

- Sequenzdiagramm für zeitliche Abläufe
- Kommunikationsdiagramm für Objektkollaborationen
- Zustandsdiagramm für Objektlebenszyklen
- Aktivitätsdiagramm für Geschäftsprozesse

3. Verwendungszweck

- Analyse: Konzeptuelle Modellierung
- Design: Detaillierte Spezifikation
- Implementation: Code-nahe Darstellung
- Dokumentation: Architekturübersicht

UML in der Praxis

Beispiel eines kompletten Designs:

```java
// Paketstruktur
package com.example.shop;

// Domain Model
public class Product {
    private ProductId id;
    private String name;
    private Money price;
    private Category category;
}

// Service Layer
@Service
public class ProductService {
    private final ProductRepository repository;
    private final PriceCalculator calculator;

    public Product updatePrice(
            ProductId id, Money newPrice) {
        Product product = repository.findById(id)
            .orElseThrow(ProductNotFoundException::new);

        Money calculatedPrice = calculator
            .calculateFinalPrice(newPrice);

        product.updatePrice(calculatedPrice);
        return repository.save(product);
    }
}

// Controller Layer
@RestController
@RequestMapping("/api/products")
public class ProductController {
    private final ProductService service;

    @PutMapping("/{id}/price")
    public ProductDTO updatePrice(
            @PathVariable ProductId id,
            @RequestBody PriceUpdateRequest request) {
        Product product = service.updatePrice(
            id, request.getNewPrice());
        return ProductDTO.from(product);
    }
}
```

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
        )
    );
}
```