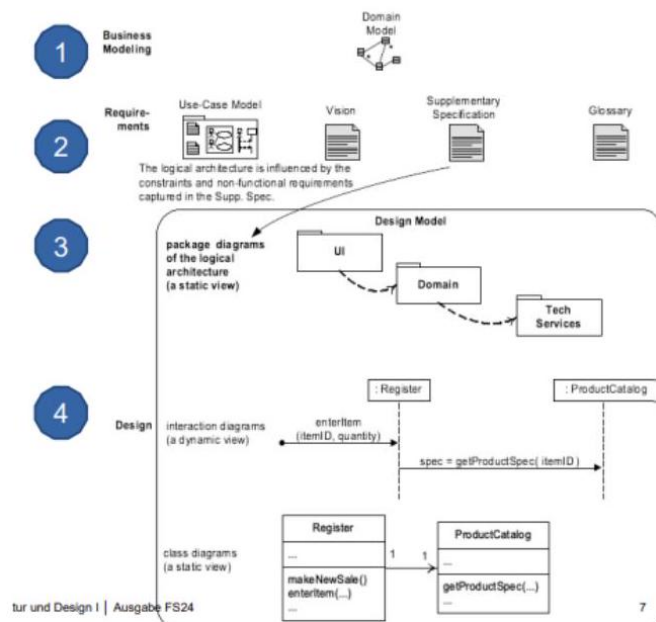


Grundlagen und Überblick

- **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

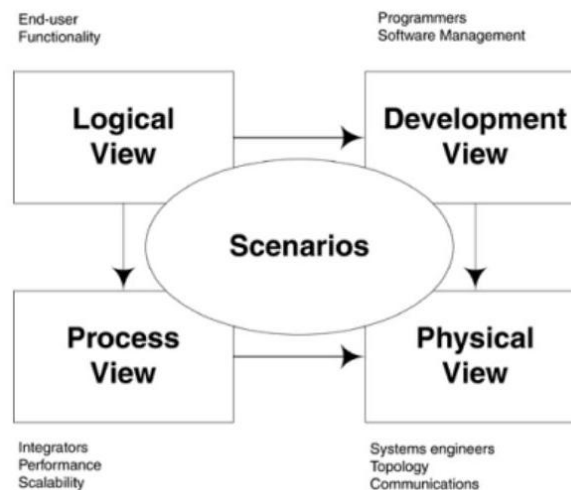
Architektursichten (4+1 View Model)

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

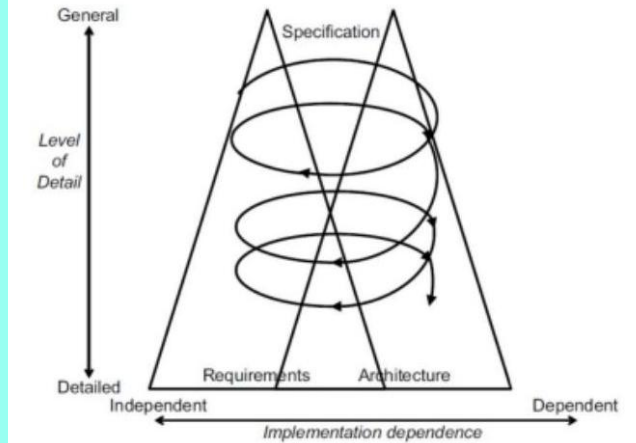
- Wichtige Use Cases
- Validierung der Architektur
- Integration der anderen Views



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



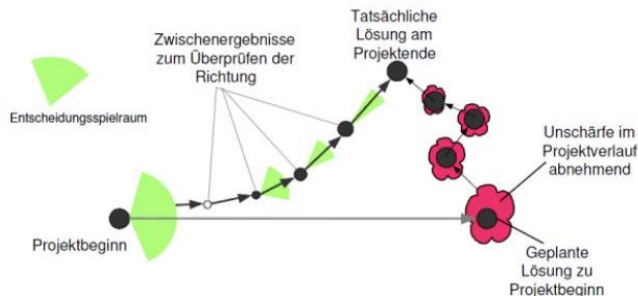
Qualitätsanforderungen

ISO 25010:

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

FURPS+:

- **Functionality** (Funktionalität)
- **Usability** (Benutzerfreundlichkeit)
- **Reliability** (Zuverlässigkeit)
- **Performance** (Leistung)
- **Supportability** (Wartbarkeit)
- **+**: Implementation, Interface, Operations, Packaging, Legal



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

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)

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

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

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

```

1 // Präsentationsschicht
2 public class CustomerController {
3     private CustomerService service;
4
5     public CustomerDTO getCustomer(String id) {
6         return service.findCustomer(id);
7     }
8 }
9
10 // Anwendungsschicht
11 public class CustomerService {
12     private CustomerRepository repository;
13
14     public CustomerDTO findCustomer(String id) {
15         Customer customer = repository.findById(id);
16         return CustomerDTO.from(customer);
17     }
18 }
19
20 // Geschäftslogikschicht
21 public class Customer {
22     private CustomerId id;
23     private String name;
24
25     public void updateName(String newName) {
26         validateName(newName);
27         this.name = newName;
28     }
29 }
30
31 // Datenzugriffsschicht
32 public class CustomerRepository {
33     public Customer findById(String id) {
34         // Datenbankzugriff
35     }
36 }
```

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

```

1 // Entity (innerste Schicht)
2 public class Customer {
3     private CustomerId id;
4     private String name;
5
6     public void validateName(String name) {
7         // Domainregeln fuer Namen
8     }
9 }
10
11 // Use Case (Business Rules)
12 public class RegisterCustomerUseCase {
13     public void execute(RegisterCustomerCommand cmd) {
14         Customer customer = new
15             Customer(cmd.getName());
16         customer.validateName(cmd.getName());
17         repository.save(customer);
18     }
19 }
20
21 // Interface Adapter
22 public class CustomerController {
23     private RegisterCustomerUseCase useCase;
24
25     public ResponseEntity<CustomerDTO> register(
26         CustomerRequest request) {
27         useCase.execute(
28             new
29                 RegisterCustomerCommand(request.getName())
30         );
31         return ResponseEntity.ok().build();
32     }
33 }
```

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)

```
1 @Service
2 public class OrderService {
3     private final CustomerClient customerClient;
4     private final PaymentClient paymentClient;
5
6     @CircuitBreaker(name = "order")
7     public OrderResult createOrder(OrderRequest
8         request) {
9         // Kundeninformationen laden
10        CustomerInfo customer =
11            customerClient.getCustomer(request.getCustomerId());
12
13        // Zahlungsabwicklung
14        PaymentResult payment =
15            paymentClient.processPayment(request.getAmount());
16
17        // Order erstellen
18        return createOrderWithPayment(customer,
19            payment);
20    }
21 }
```

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

```
1 // Event Definition
2 public class OrderCreatedEvent {
3     private final OrderId orderId;
4     private final CustomerId customerId;
5     private final Money totalAmount;
6     private final LocalDateTime timestamp;
7 }
8
9 // Event Producer
10 @Service
11 public class OrderService {
12     private final EventPublisher eventPublisher;
13
14     public Order createOrder(OrderRequest request) {
15         Order order = orderRepository.save(
16             new Order(request));
17
18         eventPublisher.publish(new OrderCreatedEvent(
19             order.getId(),
20             order.getCustomerId(),
21             order.getTotalAmount(),
22             LocalDateTime.now())
23         );
24
25         return order;
26     }
27 }
28
29 // Event Consumer
30 @Service
31 public class NotificationService {
32     @EventListener
33     public void handleOrderCreated(
34         OrderCreatedEvent event) {
35         sendConfirmationEmail(event.getCustomerId());
36     }
37 }
```

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:

```
1 public class Order {
2     private OrderId id;
3     private Customer customer;
4     private List<OrderLine> lines;
5     private OrderStatus status;
6
7     public Money calculateTotal() {
8         return lines.stream()
9             .map(OrderLine::getSubTotal)
10            .reduce(Money.ZERO, Money::add);
11    }
12
13    public void addProduct(Product product, int qty) {
14        lines.add(new OrderLine(product, qty));
15    }
16 }
17
18 public class OrderLine {
19     private Product product;
20     private int quantity;
21
22     public Money getSubTotal() {
23         return product.getPrice()
24             .multiply(quantity);
25     }
26 }
27
28 @Service
29 public class OrderService {
30     private OrderRepository repository;
31
32     public Order createOrder(OrderRequest request) {
33         Order order = new
34             Order(request.getCustomerId());
35         request.getItems().forEach(item ->
36             order.addProduct(item.getProduct(),
37                 item.getQuantity()));
38         return repository.save(order);
39     }
40 }
```

Sequenzdiagramm für Bestellprozess

Implementierung einer Bestellverarbeitung:

```
1 @RestController
2 public class OrderController {
3     private final OrderService orderService;
4     private final PaymentService paymentService;
5
6     public OrderResponse createOrder(
7         OrderRequest request) {
8         // Validiere Bestellung
9         validateOrder(request);
10
11        // Erstelle Order
12        Order order =
13            orderService.createOrder(request);
14
15        // Prozessiere Zahlung
16        PaymentResult payment =
17            paymentService.processPayment(
18                order.getId(),
19                order.getTotal()
20            );
21
22        // Update Order Status
23        if (payment.isSuccessful()) {
24            order.confirm();
25            orderService.save(order);
26        }
27
28        return OrderResponse.from(order);
29    }
30 }
```

Zustandsdiagramm für Bestellstatus

Implementation des State Patterns:

```
1 public interface OrderState {
2     void process(Order order);
3     void cancel(Order order);
4     void ship(Order order);
5 }
6
7 public class NewOrderState implements OrderState {
8     @Override
9     public void process(Order order) {
10         validateOrder(order);
11         order.setState(new ProcessingState());
12     }
13
14     @Override
15     public void cancel(Order order) {
16         order.setState(new CancelledState());
17     }
18
19     @Override
20     public void ship(Order order) {
21         throw new IllegalStateException(
22             "Cannot ship new order");
23     }
24 }
25
26 public class Order {
27     private OrderState state;
28
29     public void process() {
30         state.process(this);
31     }
32
33     void setState(OrderState newState) {
34         this.state = newState;
35     }
36 }
```

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:

```
1 public class OrderProcessor {
2     public void processOrder(Order order) {
3         // Parallel processing
4         CompletableFuture.allOf(
5             validateInventory(order),
6             validatePayment(order)
7         ).thenRun(() -> {
8             if (order.isValid()) {
9                 fulfillOrder(order);
10            } else {
11                handleValidationFailure(order);
12            }
13        });
14    }
15
16    private CompletableFuture<Void> validateInventory(
17        Order order) {
18        return CompletableFuture.runAsync(() -> {
19            order.getItems().forEach(item -> {
20                if
21                    (!inventoryService.isAvailable(item))
22                    {
23                        throw new
24                            OutOfStockException(item);
25                    }
26            });
27        });
28    }
29 }
```


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
- Verworfen 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

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

Gute Testbarkeit

```
1 public class OrderService {
2     private final OrderRepository repository;
3     private final PaymentGateway paymentGateway;
4
5     // Dependency Injection ermoeeglicht einfaches
6     // Mocking
7     public OrderService(
8         OrderRepository repository,
9         PaymentGateway paymentGateway) {
10         this.repository = repository;
11         this.paymentGateway = paymentGateway;
12     }
13
14     // Klare Methoden-Verantwortlichkeiten
15     public OrderResult createOrder(OrderRequest
16         request) {
17         validateRequest(request);
18         Order order = createOrderEntity(request);
19         PaymentResult payment = processPayment(order);
20         return createOrderResult(order, payment);
21     }
22 }
```

Dokumentation Architektur

Architekturanalyse
Analyse für ein E-Commerce-System:

```
1 // Dokumentation der Analyse
2 public class ArchitectureAnalysis {
3     public class QualityRequirement {
4         String name;
5         String description;
6         int priority;
7         String measurementCriteria;
8     }
9
10    public class ArchitecturalConstraint {
11        String type; // Technical, Organizational,
12        // Business
13        String description;
14        String impact;
15    }
16
17    // Beispiel Qualitaetsanforderung
18    QualityRequirement performance = new
19    QualityRequirement(
20        "Response Time",
21        "System responses within 200ms",
22        1,
23        "95th percentile < 200ms"
24    );
25
26    // Beispiel Randbedingung
27    ArchitecturalConstraint technology = new
28    ArchitecturalConstraint(
29        "Technical",
30        "Must use Java 17",
31        "Affects framework selection"
32    );
33 }
```

Architektur-Entscheidungen
Entscheidungsdokumentation:

```
1 public class ArchitectureDecision {
2     String id;
3     String title;
4     String context;
5     String decision;
6     String rationale;
7     List<String> consequences;
8     List<Alternative> alternatives;
9
10    class Alternative {
11        String description;
12        List<String> pros;
13        List<String> cons;
14        String rejectionReason;
15    }
16 }
17
18 // Beispiel:
19 ArchitectureDecision caching = new
20 ArchitectureDecision(
21     "AD001",
22     "Caching Strategy",
23     "High read load on product catalog",
24     "Use Redis as distributed cache",
25     "Better performance and scalability",
26     List.of("Requires Redis expertise",
27         "Additional infrastructure"),
28     List.of(new Alternative(
29         "In-memory cache",
30         List.of("Simple", "No additional
31         infrastructure"),
32         List.of("Not distributed", "Memory limited"),
33         "Doesn't scale horizontally"
34     ))
35 );
```

Architektur-Review

Review-Protokoll:

```
1 public class ArchitectureReview {
2     public class Finding {
3         String area;
4         String observation;
5         Risk risk;
6         String recommendation;
7         Priority priority;
8     }
9
10    public class Action {
11        String description;
12        String responsible;
13        LocalDate dueDate;
14        Status status;
15    }
16
17    List<Finding> findings = List.of(
18        new Finding(
19            "Security",
20            "Missing rate limiting",
21            Risk.HIGH,
22            "Implement API gateway with rate limiting",
23            Priority.HIGH
24        )
25    );
26
27    List<Action> actions = List.of(
28        new Action(
29            "Implement API gateway",
30            "Team A",
31            LocalDate.now().plusWeeks(2),
32            Status.OPEN
33        )
34    );
35 }
```