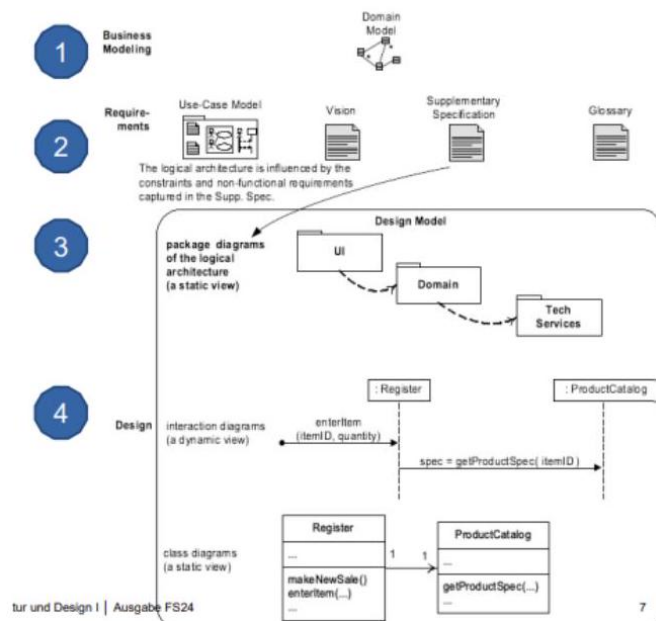


Grundlagen und Überblick

- **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



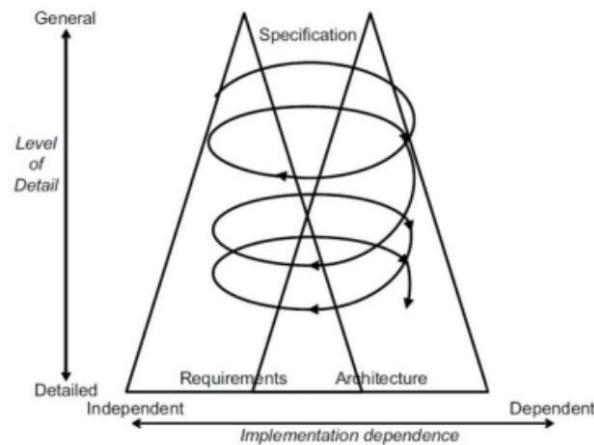
Softwarearchitektur Die Architektur eines Softwaresystems definiert:

- **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+:

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

Architekturanalyse durchführen

1. Anforderungen analysieren

- Funktionale Anforderungen gruppieren
- Nicht-funktionale Anforderungen priorisieren
- Randbedingungen identifizieren

2. Qualitätsziele definieren

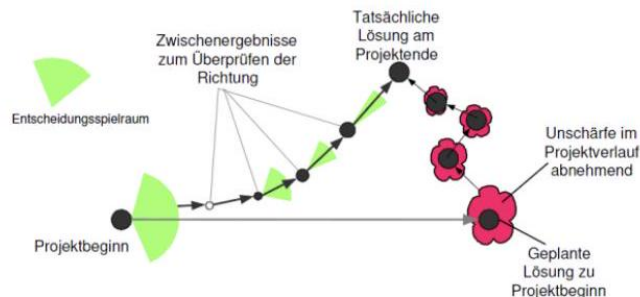
- Messbare Kriterien festlegen
- Priorisierung vornehmen
- Trade-offs identifizieren

3. Architekturentscheidungen treffen

- Alternativen evaluieren
- Entscheidungen dokumentieren
- Mit Stakeholdern abstimmen

4. Validierung durchführen

- Architektur-Reviews planen
- Prototypen erstellen
- Risiken bewerten



Modulkonzept

Ein Modul (Baustein, Komponente) wird charakterisiert durch:

- **Eigenschaften:**
 - Autarkes Teilsystem (geringe externe Abhängigkeiten)
 - Klar definierte Schnittstellen nach außen
 - Enthält alle benötigten Funktionen und Daten
 - Kann als Paket, Library, Komponente oder Service realisiert werden
- **Bewertungskriterien:**
 - **Kohäsion:** Stärke des inneren Zusammenhangs
 - **Kopplung:** Grad der Abhängigkeit zu anderen Modulen

Schnittstellen

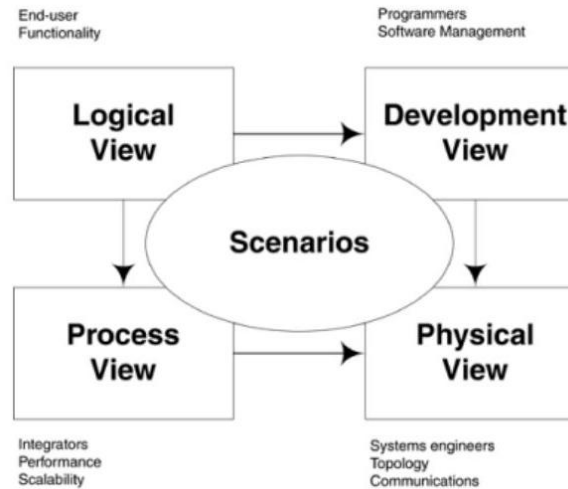
Module kommunizieren über definierte Schnittstellen:

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

Architektursichten (4+1 View Model)

Verschiedene Perspektiven auf die Architektur:

- **Logical View:**
 - Funktionalität des Systems
 - Schichten, Subsysteme, Pakete
 - Klassen und Schnittstellen
- **Process View:**
 - Laufzeitverhalten
 - Prozesse und Threads
 - Performance und Skalierung
- **Development View:**
 - Implementierungsstruktur
 - Quellcode-Organisation
 - Build und Deployment
- **Physical View:**
 - Hardware-Topologie
 - Verteilung der Software
 - Netzwerkkommunikation
- **+1: Scenarios:**
 - Wichtige Use Cases
 - Validierung der Architektur
 - Integration der anderen Views



Qualitätskriterien und deren Umsetzung

Strategien zur Erfüllung von Qualitätsanforderungen:

Performance:

- Resource Pooling (Wiederverwendung von Ressourcen)
- Caching (Zwischenspeicherung)
- Parallelisierung (Verteilung der Last)
- Lazy Loading (Verzögerte Initialisierung)

Skalierbarkeit:

- Horizontale Skalierung (mehr Instanzen)
- Vertikale Skalierung (mehr Ressourcen)
- Load Balancing (Lastverteilung)
- Partitionierung (Datenaufteilung)

Wartbarkeit:

- Separation of Concerns (Trennung der Zuständigkeiten)
- Information Hiding (Kapselung)
- Standardisierung (einheitliche Patterns)
- Modularisierung (unabhängige Komponenten)

Zuverlässigkeit:

- Redundanz (Mehrfachsysteme)
- Fehlertoleranz (Graceful Degradation)
- Monitoring (Überwachung)
- Backup und Recovery

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

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

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

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

```
1 // Information Expert
2 public class Order {
3     private List<OrderLine> lines;
4
5     // Order kennt seine eigenen Daten
6     public Money calculateTotal() {
7         return lines.stream()
8             .map(OrderLine::getSubTotal)
9             .reduce(Money.ZERO, Money::add);
10    }
11 }
12
13 // Creator
14 public class Order {
15     // Order erstellt OrderLines
16     // (enthaelt und verwendet sie)
17     public void addProduct(Product product, int
18         quantity) {
19         lines.add(new OrderLine(product, quantity));
20    }
21 }
22
23 // Controller
24 public class OrderController {
25     private OrderService orderService;
26
27     // Koordiniert Systemoperationen
28     public OrderResponse createOrder(OrderRequest
29         request) {
30         Order order =
31             orderService.createOrder(request);
32         return OrderResponse.from(order);
33    }
34 }
```

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

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

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 }
```

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 }
```

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