Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

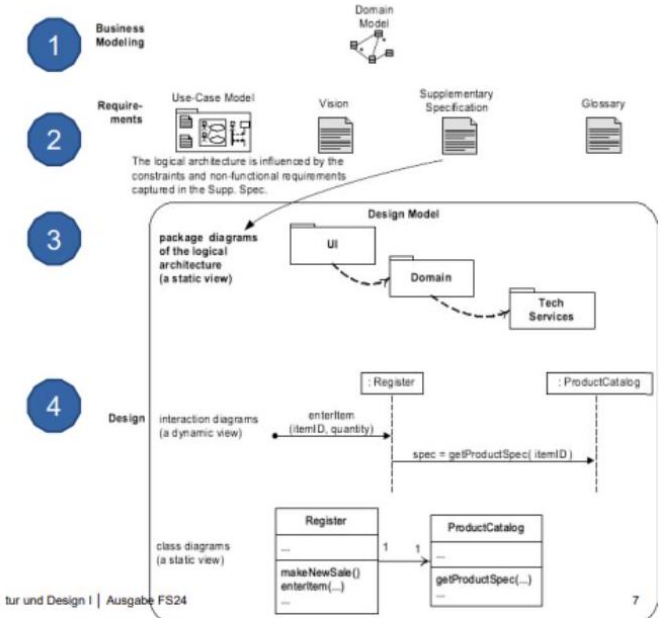
Softwarearchitektur und Design

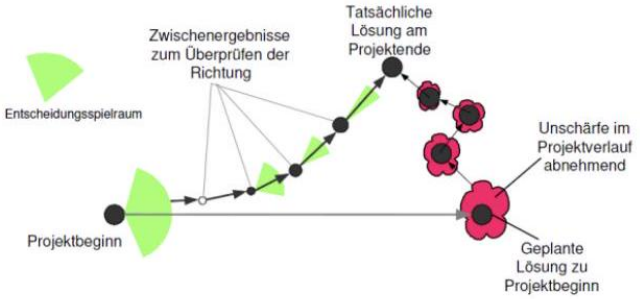
Grundlagen und Überblick

- **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

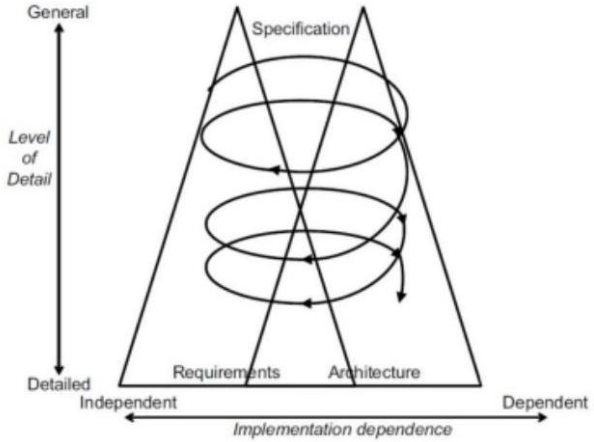
Die Architektur eines Softwaresystems definiert:

- **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

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

Architektur-Design

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

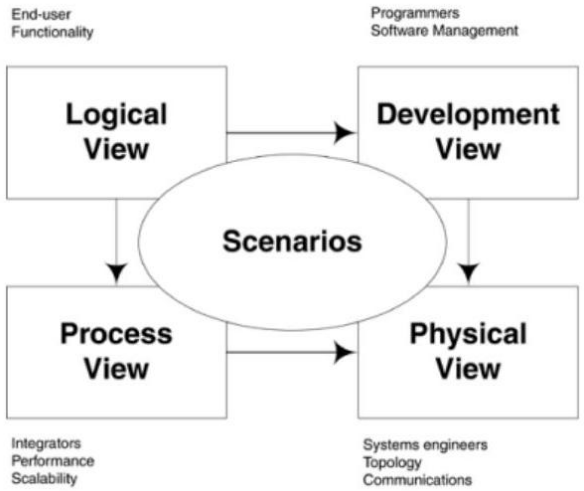
Module kommunizieren über definierte Schnittstellen:

- **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



Qualitaetskriterien 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

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

Schichtenarchitektur (Layered Architecture)

Organisation des Systems in hierarchische Schichten:

Typische Schichten:

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

Prinzipien:

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

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

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

// Anwendungsschicht
public class CustomerService {
    private CustomerRepository repository;

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

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

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

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

Clean Architecture

Architektur-Prinzipien nach Robert C. Martin:

Hauptprinzipien:

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

Schichten (von innen nach außen):

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

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

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

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

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

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

Microservices Architektur

Verteilte Architektur mit unabhängigen Services:

Charakteristiken:

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

Patterns:

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

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

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

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

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

Event-Driven Architecture (EDA)

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

Kernkomponenten:

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

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

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

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

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

        return order;
    }
}

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

Integration Patterns

Muster für die Integration verschiedener Systeme:

Hauptkategorien:

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

Spezifische Patterns:

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

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

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

    // Order kennt seine eigenen Daten
    public Money calculateTotal() {
        return lines.stream()
                    .map(OrderLine::getSubTotal)
                    .reduce(Money.ZERO, Money::add);
    }
}

// Creator
public class Order {
    // Order erstellt OrderLines
    // (enthaelt und verwendet sie)
    public void addProduct(Product product, int
         quantity) {
        lines.add(new OrderLine(product, quantity));
    }
}

// Controller
public class OrderController {
    private OrderService orderService;

    // Koordiniert Systemoperationen
    public OrderResponse createOrder(OrderRequest
         request) {
        Order order =
             orderService.createOrder(request);
        return OrderResponse.from(order);
    }
}
```

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:

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