Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Einführung und Überblick

Software Engineering

- Das Software Engineering (SE) beschäftigt sich mit der Herstellung oder Entwicklung von Software, der Organisation und Modellierung der zugehörigen Datenstrukturen und dem Betrieb von Softwaresystemen
- Disziplinen: Anforderungen, Architektur, Implementierung, Test und Wartung
- Ziel: Strukturierte Prozesse für Qualität, Risiko- und Fehlerminimierung
- Aufgrund des hohen Aufwandes zur Erstellung und Wartung komplexer Software erfolgt die Entwicklung durch Softwareentwickler anhand eines strukturierten (Projekt-)Planes

Vorgehen bei der Softwareentwicklung

1. Anforderungen verstehen und dokumentieren
2. Systemabgrenzung vornehmen
3. Architektur und Design entwerfen
4. Implementierung in kleinen, testbaren Schritten
5. Kontinuierliche Integration und Testing
6. Regelmäßige Reviews und Refactoring

Modellierung in der Softwareentwicklung

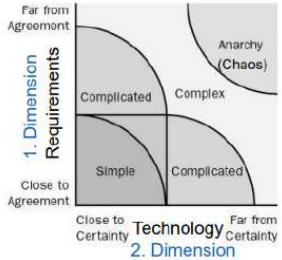
- Modelle als Abstraktionen: Anforderungen, Architekturen, Testfälle
- Einsatz von UML für:
  - Verstehen eines Gebildes
  - Kommunizieren über ein Gebilde
  - Spezifikation von Anforderungen
  - Durchführung von Experimenten
  - Aufstellen/Prüfen von Hypothesen
- Wieviel Modellierung?: Abhängig von Projektgröße und Komplexität

Modellierungsaufwand Analogie zur Bauplanung:
- Hundehütte: Minimale Planung nötig
- Einfamilienhaus: Mittlerer Planungsaufwand
- Wolkenkratzer: Extensive Planung erforderlich

Softwareentwicklungsprozesse

Klassifizierung Software-Entwicklungs-Probleme

Drei Dimensionen beeinflussen die Wahl des Entwicklungsprozesses:
- Requirements (Klarheit der Anforderungen)
- Technology (Bekanntheitsgrad der Technologie)
- People (Erfahrung und Fähigkeiten des Teams)



Quelle: Agile Project Mangement with Scrum, Ken Schwaber, 2003

Prozesse im Softwareengineering

Kernprozesse:
- Anforderungserhebung
- Systemdesign/technische Konzeption
- Implementierung
- Softwaretest
- Softwareeinführung
- Wartung/Pflege

Unterstützungsprozesse

- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

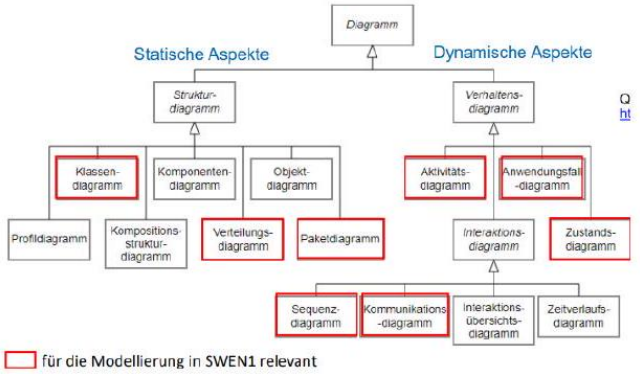
Prozessauswahl

Faktoren für die Wahl des Entwicklungsprozesses:
1. Analyse der Projektcharakteristika:
   - Projektgröße und Komplexität
   - Klarheit der Anforderungen
   - Technologische Risiken
   - Team-Erfahrung
2. Wahl des passenden Modells:
   - Code & Fix: Für sehr kleine, experimentelle Projekte
   - Wasserfall: Für Projekte mit klaren, stabilen Anforderungen
   - Iterativ-inkrementell: Für komplexe Projekte mit sich entwickelnden Anforderungen
   - Agil: Für Projekte mit hoher Änderungsdynamik

[Previous content continues with the definitions of Code and Fix, Wasserfallmodell, etc...]

UML in der Praxis

- UML as Sketch: Informelle Diagramme zur Kommunikation
- UML as Blueprint: Detaillierte Analyse- und Design-Diagramme
- UML as Programming Language: Ausführbare Spezifikationen



[Previous content continues with the formula for Incremental Model...]

Typischer Ablauf einer Iteration

```
// 1. Planungsphase
Iterator<UserStory> stories = iteration.getPlannedSto
while (stories.hasNext()) {
    UserStory story = stories.next();

    // 2. Analyse
    Requirements reqs = story.analyzeRequirements();

    // 3. Design
    Design design = story.createDesign();

    // 4. Implementierung
    Implementation impl = story.implement(design);

    // 5. Testing
    TestSuite tests = story.createTests();
    boolean passed = tests.executeOn(impl);

    // 6. Review und Integration
    if (passed) {
        mainBranch.integrate(impl);
    }
}
```

Anforderungsanalyse

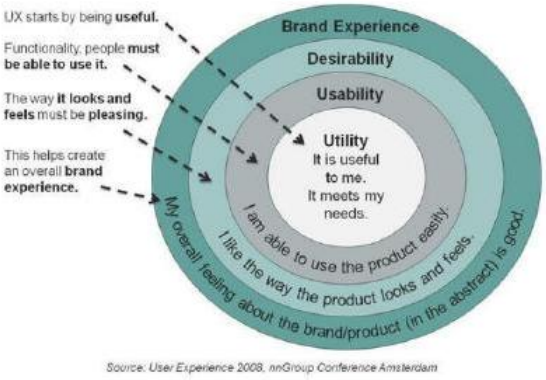
Übersicht Anforderungsanalyse

- Anforderungen sind nie vollständig im Voraus bekannt
- Entwickeln sich während des Projekts
- Müssen mit Stakeholdern erarbeitet werden
- Ï don't know what I want but I'll tell you when I see it!"

Usability und User Experience

Usability und User Experience Die drei Säulen der Benutzererfahrung:

- **Usability (Gebrauchstauglichkeit):** Grundlegende Nutzbarkeit des Systems
- **User Experience:** Usability + Desirability (Attraktivität)
- **Customer Experience:** UX + Brand Experience (Markenwahrnehmung)



Source: User Experience 2008, nnGroup Conference Amsterdam

Erfassen von Usability-Anforderungen

1. Nutzergruppen identifizieren
   - Primäre Nutzer
   - Gelegentliche Nutzer
   - Systemadministratoren
2. Nutzungskontext analysieren
   - Physische Umgebung
   - Technische Umgebung
   - Soziale Umgebung
3. Messbare Kriterien definieren
   - Erfolgsrate bei Aufgaben
   - Zeitbedarf für Aktionen
   - Fehlerrate bei Bedienung

[Previous content for Usability-Dimensionen and ISO 9241-110 remains...]

Messbare Usability-Kriterien Für ein Bankomat-System:

- **Effektivität:** 98% aller Geldabhebungen erfolgreich
- **Effizienz:** Standardabhebung in max. 30 Sekunden
- **Zufriedenheit:** Mind. 4 von 5 Punkten in Nutzerbefragung
- **Fehlertoleranz:** Max. 1% Abbrüche durch Bedienfehler

[Previous content for User-Centered Design remains...]

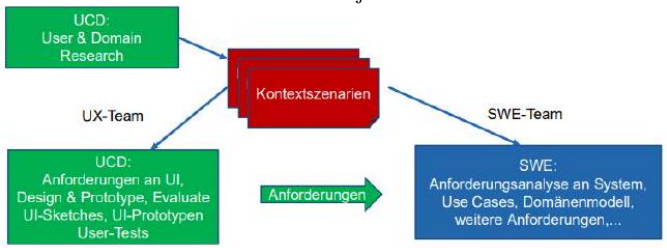
Erstellen einer Persona

1. Nutzungskontext recherchieren
   - Interviews durchführen
   - Nutzungsverhalten beobachten
   - Existierende Daten analysieren
2. Persona definieren
   - Name und Foto (fiktiv aber realistisch)
   - Demografische Daten
   - Technische Affinität
   - Ziele und Frustrationen
   - Typische Verhaltensweisen
3. Persona validieren
   - Mit Stakeholdern abstimmen
   - Mit realen Nutzerdaten vergleichen

Requirements Engineering

Requirements (Anforderungen)

- Funktionale Anforderungen: Was das System tun soll
- Nicht-funktionale Anforderungen: Wie das System sein soll
- Randbedingungen: Einschränkungen und Vorgaben
- Müssen mit allen Stakeholdern erarbeitet werden
- Entwickeln sich während des Projekts



Anforderungserhebung

1. Stakeholder identifizieren
2. Informationsquellen erschließen
   - Interviews
   - Workshops
   - Beobachtungen
   - Dokumente
3. Anforderungen dokumentieren
   - Use Cases
   - User Stories
   - Szenarien
4. Anforderungen priorisieren
5. Anforderungen validieren

[Previous content for Use Cases section remains...]

System Sequence Diagram für Bankomat

```
// Systemoperationen aus SSD
public interface ATMSystem {
    // Kunde authentifizieren
    boolean authenticate(Card card, PIN pin);

    // Kontostand abfragen
    Balance getBalance(AccountType type);

    // Geldbetrag abheben
    boolean withdraw(Amount amount, AccountType type);

    // Beleg drucken
    void printReceipt(Transaction transaction);
}
```

Prüfung von Use Cases Checkliste für qualitativ hochwertige Use Cases:

1. **Vollständigkeit**
   - Alle Stakeholder berücksichtigt?
   - Alle Szenarien abgedeckt?
   - Vor- und Nachbedingungen definiert?
2. **Konsistenz**
   - Begriffe einheitlich verwendet?
   - Keine Widersprüche?
   - Abstraktionsebene passend?
3. **Testbarkeit**
   - Eindeutige Erfolgskriterien?
   - Messbare Eigenschaften?
   - Nachvollziehbare Abläufe?

[Your previous example of the library system remains...]

Domänenmodellierung

Domänenmodellierung Die Domänenmodellierung ist ein essentieller Schritt zwischen Anforderungsanalyse und Software-Design:

- Abbildung der Fachdomäne in strukturierter Form
- Basis für späteres Objektdesign
- Kommunikationsmittel mit Stakeholdern
- Dokumentation des Problemverständnisses

Domänenmodell Ein Domänenmodell ist ein vereinfachtes UML-Klassendiagramm zur Darstellung der Fachdomäne:

- Konzepte als Klassen
- Eigenschaften als Attribute (ohne Typangabe)
- Beziehungen als Assoziationen mit Multiplizitäten
- Optional: Aggregationen/Kompositionen

Vorgehen bei der Domänenmodellierung

1. **Analyse der Dokumentation**
   - Use Cases durcharbeiten
   - Glossar berücksichtigen
   - Stakeholder-Interviews auswerten
2. **Konzepte identifizieren**
   - Substantive markieren
   - Kategorisieren (siehe Checkliste)
   - Redundanzen eliminieren
3. **Beziehungen analysieren**
   - Verben zwischen Konzepten suchen
   - Art der Beziehung bestimmen
   - Multiplizitäten festlegen
4. **Attribute hinzufügen**
   - Relevante Eigenschaften identifizieren
   - Vermeidung von Redundanz
   - Angemessene Detailtiefe wählen
5. **Review und Verfeinerung**
   - Mit Stakeholdern abstimmen
   - Konsistenz prüfen
   - Analysemuster anwenden

[Previous content for Analysemuster remains...]

Beschreibungsklassen **Problem:** Modellierung eines Bibliothekssystems

```java
// Schlechte Loesung: Redundante Daten
public class Book {
    private String title;
    private String author;
    private String isbn;
    private String description;
    private boolean isLent;
    private Date dueDate;
}

// Bessere Loesung: Beschreibungsklasse
public class BookDescription {
    private String title;
    private String author;
    private String isbn;
    private String description;
}

public class BookCopy {
    private BookDescription description;
    private boolean isLent;
    private Date dueDate;
}
```

Wertobjekte **Problem:** Modellierung von Geldbeträgen

```java
// Schlechte Loesung: Primitive Obsession
public class Order {
    private double amount;  // Problematisch!
    private String currency;
}

// Bessere Loesung: Money Value Object
public class Money {
    private BigDecimal amount;
    private Currency currency;

    public Money add(Money other) {
        if (!this.currency.equals(other.currency)) {
            throw new IllegalArgumentException("Differen
        }
        return new Money(amount.add(other.amount), curre
    }
}

public class Order {
    private Money price;  // Besser!
}
```

Validierung des Domänenmodells

Checkliste für die Qualitätssicherung:

1. **Vollständigkeit**
   - Alle wichtigen Konzepte vorhanden?
   - Alle relevanten Beziehungen modelliert?
   - Wichtige Attribute berücksichtigt?
2. **Korrektheit**
   - Konzepte richtig kategorisiert?
   - Beziehungen korrekt typisiert?
   - Multiplizitäten stimmen?
3. **Angemessenheit**
   - Abstraktionsniveau passend?
   - Detaillierungsgrad einheitlich?
   - Komplexität handhabbar?
4. **Verständlichkeit**
   - Eindeutige Bezeichnungen?
   - Klare Strukturierung?
   - Nachvollziehbare Beziehungen?

Komplettes Domänenmodell: E-Commerce System **Identifizierte Konzepte:**

- **Beschreibungsklassen:**
  - ProductCatalog
  - ProductDescription
- **Wertobjekte:**
  - Money
  - Address
- **Entitäten:**
  - Customer
  - Order
  - OrderLine
- **Zustände:**
  - OrderStatus
  - PaymentStatus

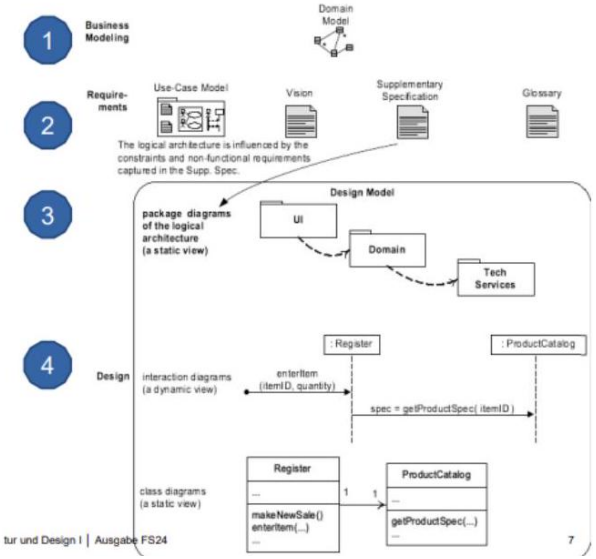
```java
// Beispielhafte Implementierung der Kernkonzepte
public class Order {
    private OrderId id;
    private Customer customer;
    private List<OrderLine> lines;
    private Money total;
    private OrderStatus status;
    private Address shippingAddress;
}

public enum OrderStatus {
    CREATED, CONFIRMED, PAID, SHIPPED, DELIVERED
}

public class OrderLine {
    private ProductDescription product;
    private int quantity;
    private Money lineTotal;
}
```

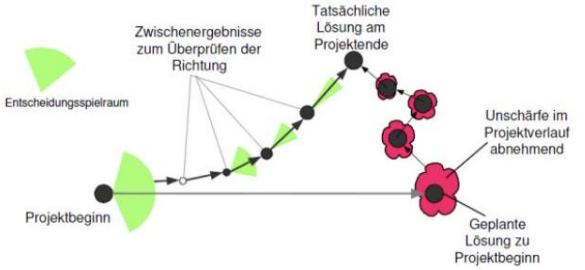
[Your previous content for avoiding modeling errors remains...]

Softwarearchitektur und Design

Überblick Softwareentwicklung Die Entwicklung von Software erfolgt in verschiedenen Ebenen:

- Business Analyse (Domänenmodell, Requirements)
- Architektur (Logische Struktur)
- Entwicklung (Konkrete Umsetzung)





Softwarearchitektur Die Architektur definiert:

- Grundlegende Strukturen und Komponenten
- Programmiersprachen und Plattformen
- Aufteilung in Teilsysteme und Bausteine
- Schnittstellen und deren Spezifikationen
- Basis-Technologien und Frameworks
- Besondere Maßnahmen für Anforderungserfüllung

Architekturentscheidungen treffen

1. Anforderungen analysieren
   - Funktionale Anforderungen identifizieren
   - Nicht-funktionale Anforderungen priorisieren
   - Randbedingungen berücksichtigen
2. Trade-offs evaluieren
   - Performance vs. Flexibilität
   - Skalierbarkeit vs. Komplexität
   - Time-to-Market vs. Qualität
3. Architekturstil wählen
   - Monolithisch
   - Microservices
   - Event-driven
   - Layered
4. Entscheidungen dokumentieren
   - Begründungen festhalten
   - Alternativen aufzeigen
   - Konsequenzen beschreiben

[Previous content for Architekturanalyse remains...]

Architekturentscheidungen Szenario: E-Commerce System

```
// Monolithische Architektur
public class OrderService {
    private InventorySystem inventory;
    private PaymentSystem payment;
    private ShippingSystem shipping;

    public Order processOrder(Cart cart) {
        // Direkte Aufrufe
        if (inventory.checkStock(cart)) {
            Payment p = payment.process(cart.getTotal());
            shipping.schedule(cart.getItems());
            return new Order(cart, p);
        }
        throw new OutOfStockException();
    }
}

// Microservices Architektur
public class OrderService {
    private OrderRepository repository;
    private EventBus eventBus;

    public Order processOrder(Cart cart) {
        // Asynchrone Event-basierte Kommunikation
        Order order = repository.create(cart);
        eventBus.publish(new OrderCreatedEvent(order));
        return order;
    }
}
```

Modulkonzept Module werden nach zwei Hauptkriterien bewertet:

Kohäsion (Zusammenhalt):

- Funktional: Alle Elemente tragen zu einer spezifischen Aufgabe bei
- Sequentiell: Ausgabe eines Elements ist Eingabe des nächsten
- Zeitlich: Elemente werden zur gleichen Zeit ausgeführt
- Logisch: Elemente gehören logisch zusammen

Kopplung (Abhängigkeiten):

- Daten: Nur Datenaustausch
- Nachricht: Nur über definierte Schnittstellen
- Inhalt: Direkter Zugriff auf interne Daten
- Global: Gemeinsame globale Daten

Modul-Design

1. Modul identifizieren
   - Fachliche Zusammengehörigkeit
   - Technische Abhängigkeiten
   - Wiederverwendbarkeit
2. Schnittstellen definieren
   - Exportierte Services
   - Benötigte Services
   - Datenformate
3. Interne Struktur entwerfen
   - Klassen und Komponenten
   - Datenstrukturen
   - Algorithmen

[Previous content for N+1 View Model and UML sections remains...]

UML-Modellierung in der Praxis Modellierung eines Bestellsystems:

1. Statisches Modell (Klassendiagramm):

```
public class Order {
    private List<OrderItem> items;
    private Customer customer;
    private OrderStatus status;

    public BigDecimal calculateTotal() { ... }
    public void addItem(Product p, int qty) { ... }
}

public class OrderItem {
    private Product product;
    private int quantity;
    private BigDecimal price;
}
```

2. Dynamisches Modell (Sequenzdiagramm Code):

```
// Prozess der im Sequenzdiagramm dargestellt wird
public class OrderProcess {
    public void processOrder(Order order) {
        // 1. Validierung
        validateOrder(order);

        // 2. Reservierung
        inventory.reserve(order);

        // 3. Zahlung
        payment.process(order);

        // 4. Bestaetigung
        notifyCustomer(order);
    }
}
```

[Previous content for RDD and GRASP remains...]

Anwendung von GRASP-Prinzipien

1. Information Expert
   - Identifiziere benötigte Informationen
   - Finde Klasse mit dieser Information
   - Weise Verantwortlichkeit zu
2. Creator
   - Prüfe Beziehungen zwischen Klassen
   - Wähle stärkste Beziehung
   - Weise Erstellungsverantwortung zu
3. Low Coupling/High Cohesion
   - Analysiere Abhängigkeiten
   - Gruppiere zusammengehörige Funktionalität
   - Minimiere externe Abhängigkeiten

Use Case Realisation

Use Case Realization Die Umsetzung von Use Cases erfolgt durch:

- Detaillierte Szenarien aus den Use Cases
- Systemantworten müssen realisiert werden
- UI statt System im SSD (System Sequence Diagram)
- Systemoperationen sind die zu implementierenden Elemente
- Berücksichtigung der Softwarearchitektur

UML im Implementierungsprozess UML dient als:

- Zwischenschritt bei wenig Erfahrung
- Kompakter Ersatz für Programmiercode
- Kommunikationsmittel (auch für Nicht-Techniker)
- Dokumentation von Design-Entscheidungen

Vorgehen bei der Use Case Realization **1. Vorbereitung und Analyse:**

```java
// Beispiel: Systemoperation aus SSD
public interface POSSystem {
    // Use Case: Process Sale
    void makeNewSale();
    void enterItem(String itemId, int quantity);
    void endSale();
    void makePayment(BigDecimal amount);
}
```

2. Design der Controller-Klasse:

```java
// Fassaden-Controller nach GRASP
public class Register implements POSSystem {
    private Sale currentSale;
    private ProductCatalog catalog;

    @Override
    public void makeNewSale() {
        currentSale = new Sale();
    }

    @Override
    public void enterItem(String itemId, int quantity
        ProductDescription desc =
            catalog.getProductDescription(itemId);
        currentSale.makeLineItem(desc, quantity);
    }
}
```

3. Implementierung der Domänenklassen:

```java
public class Sale {
    private List<SaleLineItem> items = new ArrayList
    private boolean isComplete = false;

    public void makeLineItem(ProductDescription desc
                             int quantity) {
        SaleLineItem item = new SaleLineItem(desc, q
        items.add(item);
    }

    public BigDecimal getTotal() {
        return items.stream()
            .map(SaleLineItem::getSubtotal)
            .reduce(BigDecimal.ZERO, BigDecimal::add
    }
}
```

Komplette Use Case Realization: Bestellung aufgeben **1. System Sequence Diagram (SSD):**

```java
// Systemoperationen aus SSD
public interface OrderSystem {
    String startOrder(String customerId);
    void addItem(String orderId, String productId, int qt
    OrderSummary submitOrder(String orderId);
}
```

2. Controller Implementation:

```java
@Service // Beispiel mit Spring Framework
public class OrderController implements OrderSystem {
    private final OrderRepository orders;
    private final CustomerService customers;
    private final ProductService products;

    @Override
    public String startOrder(String customerId) {
        Customer customer = customers.findById(customerId
        Order order = new Order(customer);
        return orders.save(order).getId();
    }

    @Override
    public void addItem(String orderId,
                        String productId,
                        int qty) {
        Order order = orders.findById(orderId);
        Product product = products.findById(productId);
        order.addItem(product, qty);
        orders.save(order);
    }
}
```

3. Domänenklassen:

```java
public class Order {
    private String id;
    private Customer customer;
    private List<OrderItem> items = new ArrayList<>();
    private OrderStatus status = OrderStatus.NEW;

    public void addItem(Product product, int quantity) {
        OrderItem item = new OrderItem(product, quantity)
        items.add(item);
    }

    public OrderSummary createSummary() {
        return new OrderSummary(
            id,
            customer.getName(),
            calculateTotal(),
            items.size()
        );
    }
}
```

GRASP-konforme Implementierung

1. **Information Expert**

```java
// Sale ist Expert fuer Preisberechnung
public class Sale {
    private List<SaleLineItem> items;

    public BigDecimal getTotal() {
        // Sale kennt seine Items
        return items.stream()
            .map(SaleLineItem::getSubtotal)
            .reduce(BigDecimal.ZERO,
                    BigDecimal::add);
    }
}
```

2. **Creator**

```java
// Sale erstellt SaleLineItems
public class Sale {
    public void makeLineItem(ProductDescriptio
                             int quantity) {
        // Sale erstellt und enthaelt SaleLine
        SaleLineItem item =
            new SaleLineItem(desc, quantity);
        items.add(item);
    }
}
```

3. **Low Coupling**

```java
// Verwendung von Interfaces
public class Register {
    private ProductCatalog catalog;
    // Kopplung nur ueber Interface
    private PaymentProcessor paymentProcessor;

    public void makePayment(BigDecimal amount)
        // Lose Kopplung durch Interface
        paymentProcessor.process(amount);
    }
}
```

[Your previous content for avoiding implementation errors remains...]

Design Patterns

Grundlagen Design Patterns Bewährte Lösungsmuster für wiederkehrende Probleme:

- Beschleunigen Entwicklung durch vorgefertigte Lösungen
- Verbessern Kommunikation im Team
- Bieten Balance zwischen Flexibilität und Komplexität
- **Wichtig:** Design Patterns sind kein Selbstzweck

Pattern-Auswahl und Anwendung

1. Problem identifizieren
   - Kernproblem isolieren
   - Anforderungen analysieren
   - Randbedingungen beachten
2. Patterns vergleichen
   - Ähnliche Probleme suchen
   - Lösungsansätze evaluieren
   - Komplexität vs. Nutzen abwägen
3. Pattern anwenden
   - An Kontext anpassen
   - Minimale Implementation wählen
   - Testbarkeit sicherstellen

Grundlegende Design Patterns

Adapter Pattern

```java
// Externes Interface
interface LegacyPaymentProvider {
    boolean doPayment(double amount, String currency);
}

// Gewuenschtes Interface
interface PaymentService {
    PaymentResult processPayment(Money money);
}

// Adapter
class PaymentAdapter implements PaymentService {
    private LegacyPaymentProvider legacy;

    @Override
    public PaymentResult processPayment(Money money) {
        boolean success = legacy.doPayment(
            money.getAmount().doubleValue(),
            money.getCurrency().getCode()
        );
        return new PaymentResult(success);
    }
}
```

Simple Factory

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

// Concrete Products
class PDFDocument implements Document { /*...*/ }
class WordDocument implements Document { /*...*/ }

// Factory
class DocumentFactory {
    public Document createDocument(String type) {
        switch(type.toLowerCase()) {
            case "pdf":
                return new PDFDocument();
            case "word":
                return new WordDocument();
            default:
                throw new IllegalArgumentException(
                    "Unknown type: " + type);
        }
    }
}
```

Singleton with Double-Checked Locking

```java
public class DatabaseConnection {
    private static volatile DatabaseConnection instance;
    private final Connection connection;

    private DatabaseConnection() {
        // Private constructor
        connection = createConnection();
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnection.class) {
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
        }
        return instance;
    }
}
```

Dependency Injection

```java
// Service interfaces
interface MessageService {
    void sendMessage(String msg);
}

interface UserService {
    User findUser(String id);
}

// Service implementation with DI
class NotificationService {
    private final MessageService messageService;
    private final UserService userService;

    // Constructor injection
    public NotificationService(
            MessageService messageService,
            UserService userService) {
        this.messageService = messageService;
        this.userService = userService;
    }

    public void notifyUser(String userId, String message
        User user = userService.findUser(userId);
        messageService.sendMessage(
            String.format("To: %s: %s",
                user.getEmail(), message));
    }
}
```

Chain of Responsibility

```java
abstract class AuthenticationHandler {
    private AuthenticationHandler next;

    public void setNext(AuthenticationHandler next) {
        this.next = next;
    }

    public abstract boolean handle(String username,
                                   String password);

    protected boolean handleNext(String username,
                                 String password) {
        if (next == null) {
            return false;
        }
        return next.handle(username, password);
    }
}

class DatabaseAuthHandler extends AuthenticationHandler
    @Override
    public boolean handle(String username,
                          String password) {
        // Check database
        boolean success = checkDatabase(username,
                                        password);
        if (success) {
            return true;
        }
        return handleNext(username, password);
    }
}

class LDAPAuthHandler extends AuthenticationHandler {
    @Override
    public boolean handle(String username,
                          String password) {
        // Check LDAP
        boolean success = checkLDAP(username, password);
        if (success) {
            return true;
        }
        return handleNext(username, password);
    }
}
```

[Continue with the rest of your original content, but with similar detailed examples for each pattern...]

Pattern Implementation Best Practices

1. **Interface Design**
   - Klar und minimalistisch
   - Erweiterbar gestalten
   - Semantik dokumentieren
2. **Testbarkeit**
   - Abhängigkeiten isolieren
   - Mocking ermöglichen
   - Verhalten verifizierbar
3. **Wartbarkeit**
   - SOLID Prinzipien befolgen
   - Dokumentation pflegen
   - Komplexität minimieren

Implementation, Refactoring und Testing

Von Design zu Code

Implementierungsstrategien **1. Bottom-Up Entwicklung:**

- Implementierung beginnt mit Basisbausteinen
- Schrittweise Integration zu größeren Komponenten
- **Vorteile:** Gründlich, solide Basis
- **Nachteile:** Spätes Feedback

2. Agile Entwicklung:

- Inkrementelle Entwicklung in Sprints
- Kontinuierliche Integration und Auslieferung
- **Vorteile:** Flexibilität, schnelles Feedback
- **Nachteile:** Mögliche Restrukturierung nötig

Bottom-Up vs. Agile Entwicklung **Szenario: Entwicklung eines Onlineshops**

```java
// Bottom-Up Ansatz
// 1. Basisklassen
public class Product {
    private String id;
    private String name;
    private BigDecimal price;
}

public class OrderItem {
    private Product product;
    private int quantity;
}

// 2. Zusammengesetzte Klassen
public class Order {
    private List<OrderItem> items;
    private Customer customer;
}

// Agiler Ansatz
// 1. Minimales funktionierendes System
public class SimpleOrder {
    public void addProduct(String productId) {
        // Minimale Implementation
    }
}

// 2. Inkrementelle Erweiterung
public class EnhancedOrder {
    public void addProduct(String productId, int qty) {
        // Erweiterte Funktionalitaet
    }

    public BigDecimal calculateTotal() {
        // Neue Funktion
    }
}
```

Test-Driven Development (TDD) Red-Green-Refactor Zyklus:

```java
// 1. Red: Test schreiben
@Test
void calculatesOrderTotal() {
    Order order = new Order();
    order.addItem(new Product("p1", new Money(10)));
    order.addItem(new Product("p2", new Money(20)));

    Money total = order.getTotal();

    assertEquals(new Money(30), total);
}

// 2. Green: Minimale Implementation
public class Order {
    private List<Product> items = new ArrayList<>();

    public void addItem(Product p) {
        items.add(p);
    }

    public Money getTotal() {
        return items.stream()
            .map(Product::getPrice)
            .reduce(Money.ZERO, Money::add);
    }
}

// 3. Refactor: Code verbessern
public class Order {
    private List<OrderItem> items = new ArrayList<>();

    public void addItem(Product p) {
        addItem(p, 1);
    }

    public void addItem(Product p, int quantity) {
        items.add(new OrderItem(p, quantity));
    }

    public Money getTotal() {
        return items.stream()
            .map(OrderItem::getSubtotal)
            .reduce(Money.ZERO, Money::add);
    }
}
```

Behavior-Driven Development (BDD)

```gherkin
Feature: Order Calculation
  As a customer
  I want to see my order total
  So that I know how much I need to pay

  Scenario: Calculate order with multiple items
    Given I have an empty shopping cart
    When I add 2 units of product "P1" at $10 each
    And I add 1 unit of product "P2" at $20
    Then my order total should be $40
```

```java
@Test
void calculatesOrderWithMultipleItems() {
    // Given
    ShoppingCart cart = new ShoppingCart();

    // When
    cart.addItem(new Product("P1", 10.00), 2);
    cart.addItem(new Product("P2", 20.00), 1);

    // Then
    assertEquals(40.00, cart.getTotal());
}
```

Effektives Refactoring 1. Code Smell: Lange Methode

```java
// Vor Refactoring
public class OrderProcessor {
    public void processOrder(Order order) {
        // Validierung
        if (order == null) throw new IllegalArgumentE
        if (order.getItems().isEmpty())
            throw new EmptyOrderException();

        // Lagerpruefung
        for (OrderItem item : order.getItems()) {
            if (!inventory.hasStock(item.getProduct()
                                    item.getQuantity())
                throw new OutOfStockException();
            }
        }

        // Bezahlung
        PaymentResult result =
            paymentService.process(order.getTotal());
        if (!result.isSuccessful()) {
            throw new PaymentFailedException();
        }

        // Versand
        shippingService.schedule(order);
    }
}

// Nach Refactoring
public class OrderProcessor {
    public void processOrder(Order order) {
        validateOrder(order);
        checkInventory(order);
        processPayment(order);
        scheduleShipping(order);
    }

    private void validateOrder(Order order) {
        if (order == null)
            throw new IllegalArgumentException();
        if (order.getItems().isEmpty())
            throw new EmptyOrderException();
    }

    private void checkInventory(Order order) {
        order.getItems().forEach(this::checkItemStock
    }

    private void checkItemStock(OrderItem item) {
        if (!inventory.hasStock(item.getProduct(),
                                item.getQuantity())) {
            throw new OutOfStockException();
        }
    }

    private void processPayment(Order order) {
        PaymentResult result =
            paymentService.process(order.getTotal());
        if (!result.isSuccessful()) {
```

[Continue with Testing section examples...]

Unit Testing Best Practices

```
public class OrderTest {
    private Order order;
    private Product product;

    @BeforeEach
    void setUp() {
        order = new Order();
        product = new Product("test", new Money(10));
    }

    @Test
    void newOrderHasNoItems() {
        assertTrue(order.isEmpty());
    }

    @Test
    void addingItemIncreasesTotal() {
        order.addItem(product, 2);
        assertEquals(new Money(20), order.getTotal());
    }

    @Test
    void throwsExceptionForNegativeQuantity() {
        assertThrows(IllegalArgumentException.class,
            () -> order.addItem(product, -1));
    }

    @Test
    void appliesDiscountCorrectly() {
        order.addItem(product, 10); // $100 total
        order.applyDiscount(0.1);   // 10% discount
        assertEquals(new Money(90), order.getTotal());
    }
}
```

Verteilte Systeme

Verteiltes System Ein Netzwerk aus autonomen Computern und Softwarekomponenten, die als einheitliches System erscheinen:

- Autonome Knoten und Komponenten
- Netzwerkverbindung
- Erscheint als ein System
- Gemeinsame Ressourcennutzung
- Transparente Verteilung

Verteiltes System in der Praxis

```
// Microservice-Architektur Beispiel
@RestController
public class OrderService {
    private final ProductService productService;
    private final PaymentService paymentService;

    @PostMapping("/orders")
    public OrderResponse createOrder(
            @RequestBody OrderRequest request) {
        // Synchrone Kommunikation mit Product-Service
        ProductInfo product =
            productService.getProduct(request.getProduct

        // Asynchrone Kommunikation via Message Broker
        paymentService.processPaymentAsync(
            new PaymentRequest(request.getPaymentDetails

        return new OrderResponse(/* ... */);
    }
}
```

Fehlerbehandlung in verteilten Systemen

```
public class ResilientServiceCaller {
    private final CircuitBreaker circuitBreaker;
    private final RetryTemplate retryTemplate;

    public <T> T callService(ServiceCall<T> serviceC
        return circuitBreaker.run(() ->
            retryTemplate.execute(context -> {
                try {
                    return serviceCall.execute();
                } catch (NetworkException e) {
                    // Exponential Backoff
                    long waitTime =
                        Math.pow(2, context.getRetryC
                    Thread.sleep(waitTime * 1000);
                    throw e;
                }
            })
        );
    }
}

// Verwendung
OrderInfo order = resilientCaller.callService(() ->
    orderService.getOrder(orderId));
```

Kommunikationsmuster **1. Synchrone Kommunikation:**

```
// REST-Client mit synchronem Aufruf
@FeignClient("product-service")
public interface ProductClient {
    @GetMapping("/products/{id}")
    ProductDTO getProduct(@PathVariable String id);
}

// Synchroner Service-Aufruf
ProductDTO product = productClient.getProduct(id);
```

2. Asynchrone Kommunikation:

```
// Message Producer
@Service
public class OrderEventPublisher {
    private final KafkaTemplate<String, OrderEvent> k

    public void publishOrderCreated(Order order) {
        OrderEvent event = new OrderEvent(order);
        kafka.send("order-events", event);
    }
}

// Message Consumer
@KafkaListener(topics = "order-events")
public void handleOrderEvent(OrderEvent event) {
    // Asynchrone Verarbeitung
}
```

Implementierung von Konsistenzstrategien

```
@Entity
public class Product {
    @Version
    private Long version;

    @Lock(LockModeType.OPTIMISTIC)
    public void updateStock(int quantity) {
        // Optimistic Locking durch @Version
        this.stockQuantity += quantity;
    }
}

// Verwendung mit Retry bei Konflikt
@Transactional
public void processOrder(Order order) {
    try {
        Product product = productRepo.findById(
            order.getProductId());
        product.updateStock(-order.getQuantity());
        productRepo.save(product);
    } catch (OptimisticLockException e) {
        // Retry mit neuem Versuch
        retryTemplate.execute(context -> {
            // Wiederhole Operation
            return null;
        });
    }
}
```

Service Discovery und Load Balancing

```java
// Service Registration
@SpringBootApplication
@EnableEurekaClient
public class ProductService {
    public static void main(String[] args) {
        SpringApplication.run(ProductService.class,
    }
}

// Load Balancer Configuration
@Configuration
public class LoadBalancerConfig {
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

// Service Discovery verwendung
@Service
public class ProductServiceClient {
    private final RestTemplate restTemplate;

    public ProductInfo getProduct(String id) {
        return restTemplate.getForObject(
            "http://product-service/products/" + id,
            ProductInfo.class
        );
    }
}
```

CAP-Theorem in der Praxis

- **Consistency (C):** Alle Knoten sehen dieselben Daten
- **Availability (A):** Jede Anfrage erhält eine Antwort
- **Partition Tolerance (P):** System funktioniert trotz Netzwerkausfällen

Beispiel-Implementierungen:

- **CP-System:** Distributed Database (z.B. MongoDB)
- **AP-System:** Content Delivery Network (CDN)
- **CA-System:** Traditional RDBMS (z.B. PostgreSQL)

[Previous content about Middleware Technologies and Error Sources remains...]

Persistenz

Persistenz Grundlagen

[Previous content remains the same...]

O/R-Mapping Probleme und Lösungen

```java
// Problem 1: Vererbung
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Payment {
    @Id private Long id;
    private BigDecimal amount;
}

@Entity
public class CreditCardPayment extends Payment {
    private String cardNumber;
    private String expiryDate;
}

// Problem 2: Beziehungen
@Entity
public class Order {
    @ManyToOne
    private Customer customer;

    @OneToMany(mappedBy = "order", cascade = CascadeType
    private List<OrderItem> items;
}

// Problem 3: Value Objects
@Embeddable
public class Money {
    private BigDecimal amount;
    private Currency currency;
}
```

JDBC Best Practices

```java
public class DatabaseUtils {
    // 1. Connection Pool verwenden
    private final DataSource dataSource;

    // 2. Try-with-resources fuer automatisches Schli
    public List<Customer> findCustomers(String name)
        String sql = "SELECT * FROM customers WHERE na

        try (Connection conn = dataSource.getConnectio
             PreparedStatement stmt = conn.prepareStat

            // 3. Prepared Statements gegen SQL-Injec
            stmt.setString(1, name);

            // 4. ResultSet verarbeiten
            try (ResultSet rs = stmt.executeQuery())
                List<Customer> customers = new ArrayL
                while (rs.next()) {
                    customers.add(mapCustomer(rs));
                }
                return customers;
            }
        }
    }

    // 5. Mapping in separate Methode
    private Customer mapCustomer(ResultSet rs)
            throws SQLException {
        Customer customer = new Customer();
        customer.setId(rs.getLong("id"));
        customer.setName(rs.getString("name"));
        return customer;
    }
}
```

DAO Pattern Implementation

```java
// 1. DAO Interface
public interface CustomerDao {
    Customer findById(Long id);
    List<Customer> findByName(String name);
    void save(Customer customer);
    void delete(Customer customer);
}

// 2. JDBC Implementation
public class JdbcCustomerDao implements CustomerDao {
    private final DataSource dataSource;

    @Override
    public Customer findById(Long id) {
        String sql = "SELECT * FROM customers WHERE id =
        try (Connection conn = dataSource.getConnection(
             PreparedStatement stmt =
                 conn.prepareStatement(sql)) {
            stmt.setLong(1, id);
            try (ResultSet rs = stmt.executeQuery()) {
                if (rs.next()) {
                    return mapCustomer(rs);
                }
                return null;
            }
        }
    }

    @Override
    public void save(Customer customer) {
        if (customer.getId() == null) {
            insert(customer);
        } else {
            update(customer);
        }
    }
}

// 3. JPA Implementation
@Repository
public class JpaCustomerDao implements CustomerDao {
    @PersistenceContext
    private EntityManager em;

    @Override
    public Customer findById(Long id) {
        return em.find(Customer.class, id);
    }

    @Override
    @Transactional
    public void save(Customer customer) {
        if (customer.getId() == null) {
            em.persist(customer);
        } else {
            em.merge(customer);
        }
    }
}
```

JPA Entity mit Beziehungen

```java
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "customer_id")
    private Customer customer;

    @OneToMany(mappedBy = "order",
               cascade = CascadeType.ALL,
               orphanRemoval = true)
    private List<OrderItem> items = new ArrayList<>();

    @Embedded
    private Address shippingAddress;

    @Enumerated(EnumType.STRING)
    private OrderStatus status;

    @Version
    private Long version;

    // Hilfsmethoden fuer Beziehungsverwaltung
    public void addItem(OrderItem item) {
        items.add(item);
        item.setOrder(this);
    }

    public void removeItem(OrderItem item) {
        items.remove(item);
        item.setOrder(null);
    }
}

@Entity
public class OrderItem {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "order_id")
    private Order order;

    @ManyToOne
    @JoinColumn(name = "product_id")
    private Product product;

    private int quantity;

    @Embedded
    private Money price;
}
```

Repository Pattern mit Spring Data JPA

```java
// 1. Repository Interface
public interface OrderRepository
        extends JpaRepository<Order, Long> {

    // Automatisch generierte Query
    List<Order> findByCustomerName(String name);

    // Custom Query mit JPQL
    @Query("SELECT o FROM Order o WHERE o.total > ?1"
    List<Order> findLargeOrders(Money threshold);

    // Native SQL Query
    @Query(value = "SELECT * FROM orders o " +
           "WHERE DATE(o.created_at) = CURDATE()",
           nativeQuery = true)
    List<Order> findTodaysOrders();
}

// 2. Service-Klasse mit Repository
@Service
@Transactional
public class OrderService {
    private final OrderRepository orderRepository;
    private final CustomerRepository customerReposito

    public Order createOrder(Long customerId,
                             OrderRequest request) {
        Customer customer = customerRepository
            .findById(customerId)
            .orElseThrow(() ->
                new CustomerNotFoundException(custome

        Order order = new Order(customer);
        request.getItems().forEach(item ->
            order.addItem(new OrderItem(
                item.getProductId(),
                item.getQuantity()
            )));

        return orderRepository.save(order);
    }

    @Transactional(readOnly = true)
    public List<Order> findCustomerOrders(String custo
        return orderRepository
            .findByCustomerName(customerName);
    }
}
```

[Previous content about Repository Pattern remains...]

Spring Data Repository Features

```
public interface ProductRepository
        extends JpaRepository<Product, Long> {
    // Verschiedene Abfragemethoden
    Optional<Product> findBySku(String sku);
    List<Product> findByPriceGreaterThan(Money price);

    // Paging und Sorting
    Page<Product> findByCategory(
        String category, Pageable pageable);

    // Spezifikationen fuer komplexe Queries
    List<Product> findAll(Specification<Product> spec);

    // Projections fuer optimierte Abfragen
    interface ProductSummary {
        String getName();
        Money getPrice();
    }
    List<ProductSummary> findAllProjectedBy();
}
```