Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Einführung und Überblick

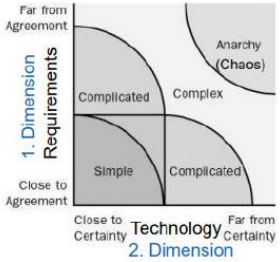
Software Engineering
- Disziplinen: Anforderungen, Architektur, Implementierung, Test und Wartung.
- Ziel: Strukturierte Prozesse für Qualität, Risiko- und Fehlerminimierung.

Modellierung in der Softwareentwicklung
- Modelle als Abstraktionen: Anforderungen, Architekturen, Testfälle.
- Einsatz von UML: Skizzen, detaillierte Blueprints, vollständige Spezifikationen.

Wrap-up
- Solide Analyse- und Entwurfskompetenzen sind essenziell.
- Iterativ-inkrementelle Modelle fördern agile Entwicklung.

Softwareentwicklungsprozesse

Klassifizierung Software-Entwicklungs-Probleme
Wir betrachten Wasserfall, iterativ-inkrementelle und agile Softwareentwicklungsprozesse.



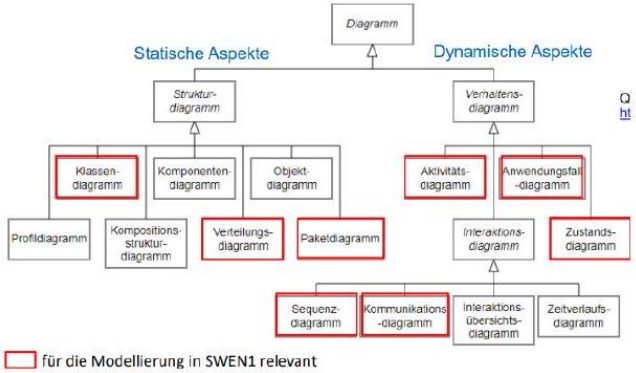
Quelle: Agile Project Mangement with Scrum, Ken Schwaber, 2003

Prozesse im Softwareengineering Kernprozesse
- Anforderungserhebung
- Systemdesign/technische Konzeption
- Implementierung
- Softwaretest
- Softwareeinführung
- Wartung/Pflege

Unterstützungsprozesse
- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

Begriffe
- Warum wird modelliert: Um Analyse- und Designentwürfe zu diskutieren, abstimmen und zu dokumentieren bzw. zu kommunizieren.
- Modell: Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhanden Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).
- Original: Das Original ist das abgebildete oder zu schaffende Gebilde.
- Modellierung: Modellierung gehört zum Fundament des Software Engineerings

Modelle in der Softwareentwicklung
- Software ist vielfach (immer?) selbst ein Modell
- Anforderungen sind Modelle der Problemstellung
- Architekturen und Entwürfe sind Modelle der Lösung
- Testfälle sind Modelle des korrekten Funktionierens des Codes usw.



Code and Fix
Vorgehen, bei dem Codierung oder Korrektur im Wechsel mit Ad-hoc-Tests die einzigen bewussten ausgeführten Tätigkeiten der Software-Entwicklung sind: Schnell, Agil, Einfach am Anfang, Schlecht Planbar, Schlecht Wartbar, Änderungen s. Aufwändig

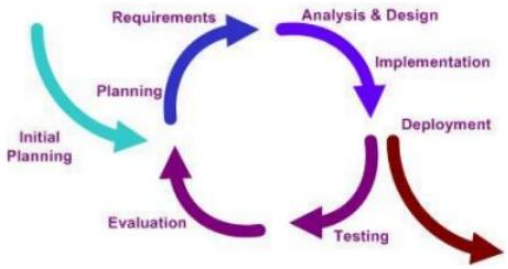
Wasserfallmodell
Die Software-Entwicklung wird als Folge von Aktivitäten/Phasen betrachtet, die durch Teilergebnisse (Dokumente) gekoppelt sind. Die Reihenfolge der Aktivitäten ist fest definiert. : gut planbar, klare Aufteilung in Phasen, Schlechtes Risikomanagment, nie alle Anforderungen zu Anfang bekannt

Iterativ-inkrementelle Modelle
Software wird in mehreren geplanten und kontrolliert durchgeführten Iterationen schrittweise (inkrementell) entwickelt: Flexibles Modell, Gutes Risikomanagement, Frühe Einsetzbarkeit, Planung upfront hat Grenzen, Kunde Involviert über ganze Entwicklung
Agile Softwareentwicklung Basiert auf interativ-inkrementellen Prozessmodell, Fokussiert auf gut dokumentierten und getesteten Code statt auf ausführlicher Dokumentation

Zweck und den Nutzen von Modellen in der Softwareentwicklung
Modell von Requirements (close to/ far from Agreement) & Technology (known / unknown)
Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhanden Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).

Unified Modelling Language (UML)
UML ist die Standardsprache für die graphische Modellierung von Anforderungen, Analyse und Entwürfen im Software Engineering (objektorientierte Modellierung). (As a sketch, blueprint, programminglanguage)

Incremental Model
Artefakte in einem iterativ-inkrementellen Prozess illustrieren und einordnen



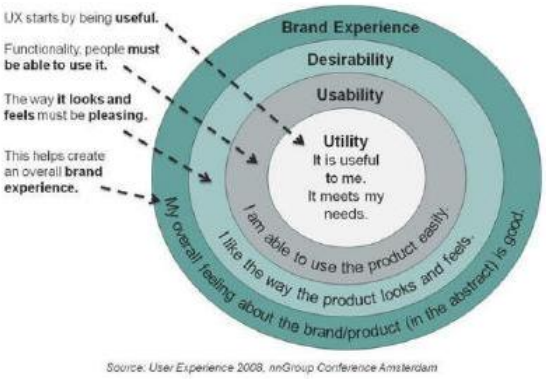
Anforderungsanalyse

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

Usability-Dimensionen

Die drei Hauptdimensionen der Usability:

- **Effektivität:**
  - Vollständige Aufgabenerfüllung
  - Gewünschte Genauigkeit
- **Effizienz:** Minimaler Aufwand
  - Mental
  - Physisch
  - Zeitlich
- **Zufriedenheit:**
  - Minimum: Keine Verärgerung
  - Standard: Zufriedenheit
  - Optimal: Begeisterung

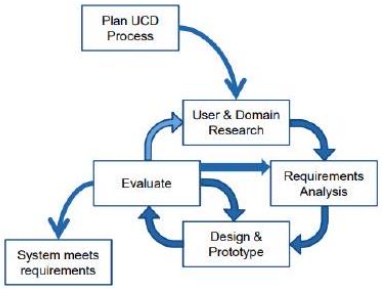
ISO 9241-110: Usability-Anforderungen

Die sieben Grundprinzipien:

- Aufgabenangemessenheit
- Lernförderlichkeit
- Individualisierbarkeit
- Erwartungskonformität
- Selbstbeschreibungsfähigkeit
- Steuerbarkeit
- Fehlertoleranz

User-Centered Design (UCD)

Ein iterativer Prozess zur nutzerzentrierten Entwicklung:



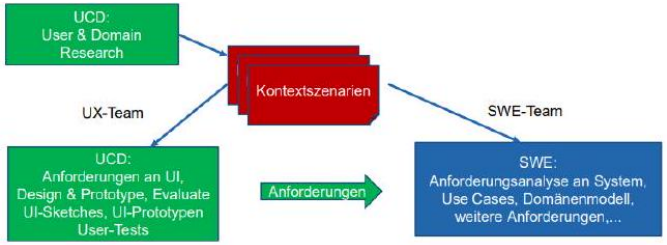
Wichtige Artefakte

- Personas: Repräsentative Nutzerprofile
- Usage-Szenarien: Konkrete Anwendungsfälle
- Mentales Modell: Nutzerverständnis
- Domänenmodell: Fachliches Verständnis
- Service Blueprint: Geschäftsprozessmodell
- Stakeholder Map: Beteiligte und Betroffene
- UI-Artefakte: Skizzen, Wireframes, Designs

Requirements Engineering

Requirements (Anforderungen)

- Leistungsfähigkeiten oder Eigenschaften
- Explizit oder implizit
- Müssen mit allen Stakeholdern erarbeitet werden
- Entwickeln sich während des Projekts



Use Cases

Use Case (Anwendungsfall)

Textuelle Beschreibung einer konkreten Interaktion zwischen Akteur und System:

- Aus Sicht des Akteurs
- Aktiv formuliert
- Konkreter Nutzen
- Essentieller Stil (Logik statt Implementierung)

Akteure in Use Cases

- **Primärakteur:** Initiiert den Use Case, erhält Hauptnutzen
- **Unterstützender Akteur:** Hilft bei der Durchführung
- **Offstage-Akteur:** Indirekt beteiligter Stakeholder

Use Case Erstellung

Schritte zur Erstellung eines vollständigen Use Cases:

1. **Identifikation:**
   - Systemgrenzen definieren
   - Primärakteure identifizieren
   - Ziele der Akteure ermitteln
2. **Dokumentation:**
   - Brief/Casual für erste Analyse
   - Fully-dressed für wichtige Use Cases
   - Standardablauf und Erweiterungen
3. **Review:**
   - Mit Stakeholdern abstimmen
   - Auf Vollständigkeit prüfen
   - Konsistenz sicherstellen

Brief Use Case **Verkauf abwickeln**

Kunde kommt mit Waren zur Kasse. Kassier erfasst alle Produkte. System berechnet Gesamtbetrag. Kassier nimmt Zahlung entgegen und gibt ggf. Wechselgeld. System druckt Beleg.

Fully-dressed Use Case **UC: Verkauf abwickeln**

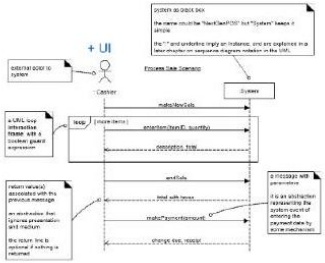
- **Umfang:** Kassensystem
- **Primärakteur:** Kassier
- **Stakeholder:** Kunde (schnelle Abwicklung), Geschäft (korrekte Abrechnung)
- **Vorbedingung:** Kasse ist geöffnet
- **Standardablauf:**
  1. Kassier startet neuen Verkauf
  2. System initialisiert neue Transaktion
  3. Kassier erfasst Produkte
  4. System zeigt Zwischensumme
  5. Kassier schliesst Verkauf ab
  6. System zeigt Gesamtbetrag
  7. Kunde bezahlt
  8. System druckt Beleg

Systemsequenzdiagramm (SSD)

Formalisierte Darstellung der System-Interaktionen:

- Zeigt Input/Output-Events
- Identifiziert Systemoperationen
- Basis für API-Design

- Links ist Primärakteur aufgeführt
  - Hier Cashier
    - Inkl. seiner Benutzerschnittstelle
  - Initiiert die Systemoperationen (via UI)
    - UI findet zusammen mit Akteur heraus, was dieser tun möchte
    - UI ruft sodann entsprechende Systemoperation auf
- Mitte das System (:System)
  - Muss die Systemoperationen zur Verfügung stellen
- Rechts
  - Sekundärakteure, falls nötig



SSD Erstellung

1. Use Case als Grundlage wählen
2. Akteur und System identifizieren
3. Methodenaufrufe definieren:
   - Namen aussagekräftig wählen
   - Parameter festlegen
   - Rückgabewerte bestimmen
4. Zeitliche Abfolge modellieren
5. Optional: Externe Systeme einbinden

Aufgabe: Erstellen Sie einen fully-dressed Use Case für ein Online-Bibliothekssystem. Fokus: "Buch ausleihen"

Lösung:

- **Umfang:** Online-Bibliothekssystem
- **Primärakteur:** Bibliotheksnutzer
- **Stakeholder:**
  - Bibliotheksnutzer: Möchte Buch einfach ausleihen
  - Bibliothek: Korrekte Erfassung der Ausleihe
- **Vorbedingung:** Nutzer ist eingeloggt
- **Standardablauf:**
  1. Nutzer sucht Buch
  2. System zeigt Verfügbarkeit
  3. Nutzer wählt Ausleihe
  4. System prüft Ausleihberechtigung
  5. System registriert Ausleihe
  6. System zeigt Bestätigung
- **Erweiterungen:**
  - 2a: Buch nicht verfügbar
  - 4a: Keine Ausleihberechtigung

Domänenmodellierung

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

Domänenmodell Erstellung

Schritt 1: Konzepte identifizieren

- Substantive aus Anforderungen extrahieren
- Kategorien prüfen:
  - Physische Objekte
  - Kataloge
  - Container
  - Externe Systeme
  - Rollen von Personen
  - Artefakte (Pläne, Dokumente)
  - Zahlungsinstrumente
- **Wichtig:** Keine Softwareklassen modellieren!

Schritt 2: Attribute definieren

- Nur wichtige/einfache Attribute
- Typische Kategorien:
  - Transaktionsdaten
  - Teil-Ganzes Beziehungen
  - Beschreibungen
  - Verwendungszwecke
- **Wichtig:** Beziehungen als Assoziationen, nicht als Attribute!

Schritt 3: Beziehungen modellieren

- Assoziationen zwischen Konzepten identifizieren
- Multiplizitäten festlegen
- Art der Beziehung bestimmen
- Richtung der Assoziation falls nötig

Analysemuster im Domänenmodell

Bewährte Strukturen für wiederkehrende Modellierungssituationen:

1. Beschreibungsklassen

- Trennung von Instanz und Beschreibung
- Beispiel: Artikel vs. Artikelbeschreibung
- Vermeidet Redundanz bei gleichen Eigenschaften

2. Generalisierung/Spezialisierung

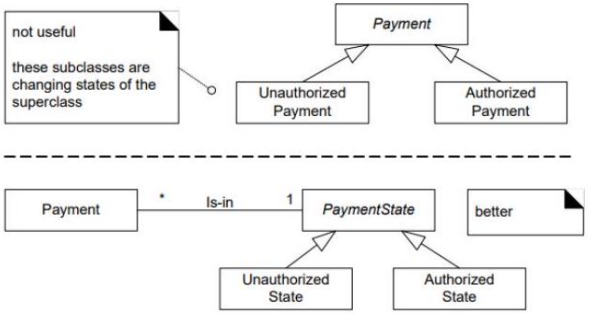
- 100% Regel: Alle Instanzen der Spezialisierung sind auch Instanzen der Generalisierung
- ÏS-A"Beziehung
- Gemeinsame Attribute/Assoziationen als Grund für Generalisierung

3. Komposition

- Starke Teil-Ganzes Beziehung
- Existenzabhängigkeit der Teile

4. Zustandsmodellierung

- Zustände als eigene Hierarchie
- Vermeidet problematische Vererbung

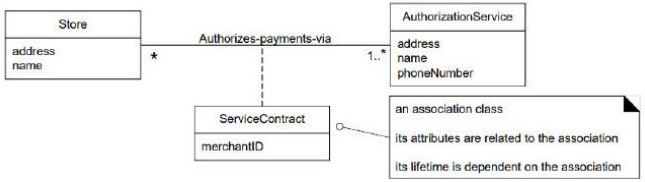


5. Rollen

- Unterschiedliche Rollen eines Konzepts
- Als eigene Konzepte oder Assoziationen

6. Assoziationsklassen

- Attribute einer Beziehung
- Eigene Klasse für die Assoziation



7. Wertobjekte

- Masseinheiten als eigene Konzepte
- Zeitintervalle als Konzepte
- Vermeidet primitive Obsession

Domänenmodell Online-Shop **Aufgabe:** Erstellen Sie ein Domänenmodell für einen Online-Shop mit Warenkorb-Funktion.

Lösung:

- **Konzepte identifizieren:**
  - Artikel (physisches Objekt)
  - Artikelbeschreibung (Beschreibungsklasse)
  - Warenkorb (Container)
  - Bestellung (Transaktion)
  - Kunde (Rolle)
- **Attribute:**
  - Artikelbeschreibung: name, preis, beschreibung
  - Bestellung: datum, status
  - Kunde: name, adresse
- **Beziehungen:**
  - Warenkorb gehört zu genau einem Kunde (Komposition)
  - Warenkorb enthält beliebig viele Artikel
  - Bestellung wird aus Warenkorb erstellt

Typische Modellierungsfehler vermeiden

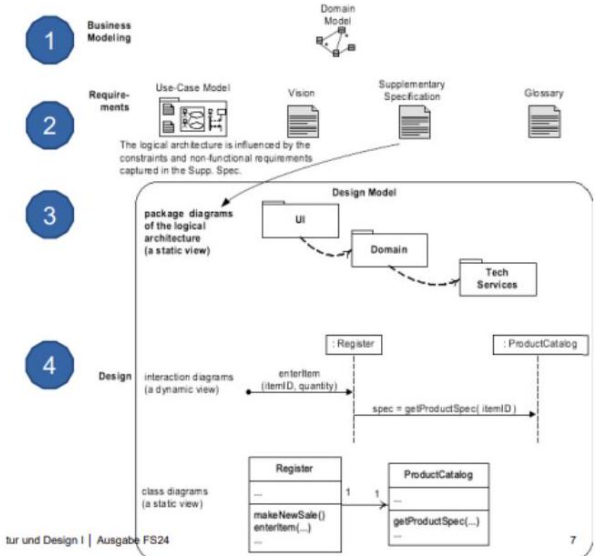
- **Keine Softwareklassen modellieren**
  - Manager-Klassen vermeiden
  - Keine technischen Helper-Klassen
- **Keine Operationen modellieren**
  - Fokus auf Struktur, nicht Verhalten
  - Keine CRUD-Operationen
- **Richtige Abstraktionsebene wählen**
  - Nicht zu detailliert
  - Nicht zu abstrakt
  - Fachliche Begriffe verwenden
- **Assoziationen statt Attribute**
  - Beziehungen als Assoziationen modellieren
  - Keine Objekt-IDs als Attribute

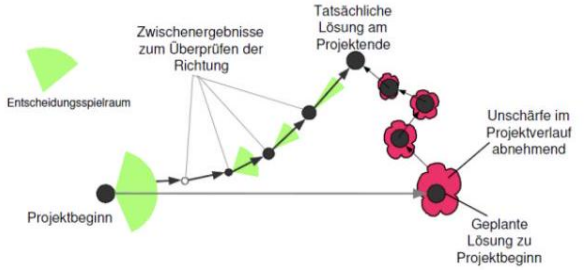
Softwarearchitektur und Design

Überblick Softwareentwicklung

Die Entwicklung von Software erfolgt in verschiedenen Ebenen:

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





Softwarearchitektur

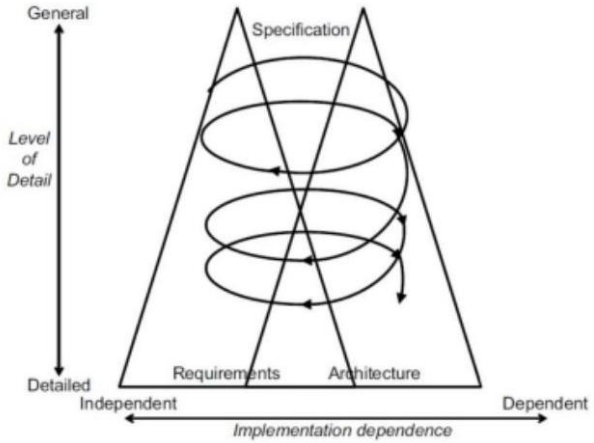
Die Architektur definiert:

- Grundlegende Strukturen und Komponenten
- Heutige und zukünftige Anforderungen
- Weiterentwicklungsmöglichkeiten
- Beziehungen zur Umgebung

Architekturanalyse

Die Analyse erfolgt iterativ mit den Anforderungen:

- Analyse funktionaler und nicht-funktionaler Anforderungen
- Abstimmung mit Stakeholdern
- Kontinuierliche Weiterentwicklung



ISO 25010 vs FURPS+

ISO 25010:

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

FURPS+:

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

Modulkonzept

Ein Modul (Baustein, Komponente) wird bewertet nach:

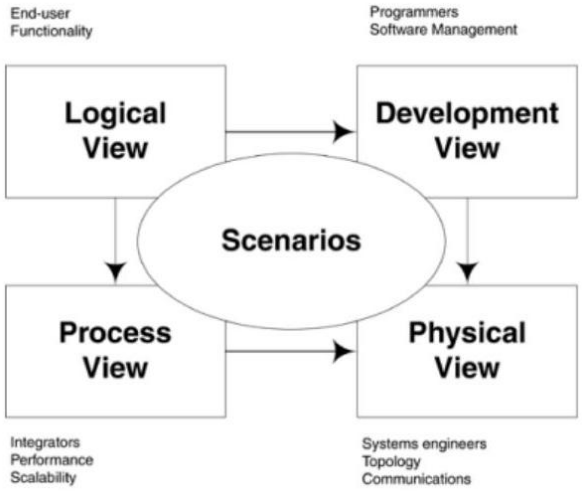
- **Kohäsion:** Innerer Zusammenhang
- **Kopplung:** Externe Abhängigkeiten

Eigenschaften:

- Autarkes Teilsystem
- Minimale externe Schnittstellen
- Enthält alle benötigten Funktionen/Daten
- Verschiedene Formen: Paket, Library, Service

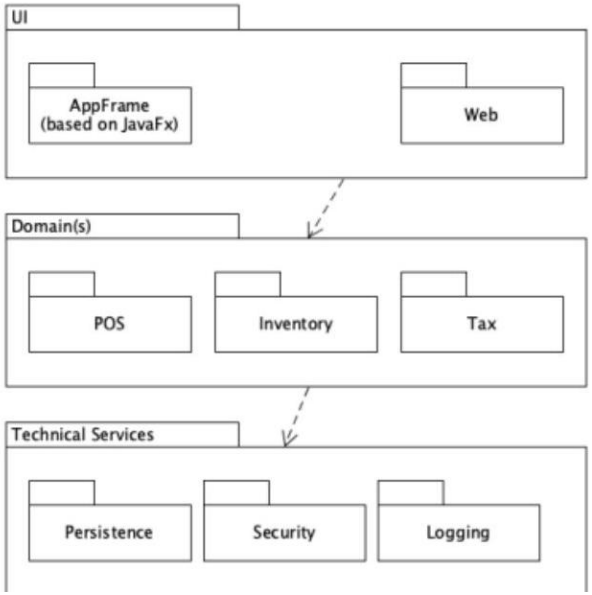
Architektursichten

Das N+1 View Model beschreibt verschiedene Perspektiven:



UML-Paketdiagramm:

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



Statische vs. Dynamische Modelle

Statische Modelle (Struktur):
- UML-Klassendiagramm
- Fokus auf Pakete, Klassen, Attribute
- Keine Methodenimplementierung

Dynamische Modelle (Verhalten):
- UML-Interaktionsdiagramme
- Fokus auf Logik und Verhalten
- Implementierung der Methoden

UML-Diagrammtypen

1. Klassendiagramm:
- Klassen und aktive Klassen
- Attribute und Operationen
- Sichtbarkeiten und Beziehungen
- Interfaces und Realisierungen

2. Sequenzdiagramm:
- Lebenslinien und Nachrichten
- Synchrone/Asynchrone Kommunikation
- Aktivierung und Deaktivierung
- Alternative Abläufe

3. Zustandsdiagramm:
- Zustände und Übergänge
- Start- und Endzustände
- Composite States
- Historie und Parallelität

4. Aktivitätsdiagramm:
- Aktionen und Aktivitäten
- Kontroll- und Datenflüsse
- Verzweigungen und Zusammenführungen
- Partitionen (Swimlanes)

Responsibility Driven Design (RDD)

Design basierend auf Verantwortlichkeiten:
- Klassenentwurf nach Rollen
- Kollaborationsbeziehungen
- Implementierung durch Attribute/Methoden
- Anwendbar auf allen Ebenen

GRASP Prinzipien

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

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

Architekturentwurf **Schritte:**

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

Qualitätskriterien:
- Änderbarkeit
- Wartbarkeit
- Erweiterbarkeit
- Testbarkeit

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
- Systemoperationen sind die zu implementierenden Elemente

UML im Implementierungsprozess

UML dient als:

- Zwischenschritt bei wenig Erfahrung
- Kompakter Ersatz für Programmiercode
- Kommunikationsmittel (auch für Nicht-Techniker)

Vorgehen bei der Use Case Realization

1. Vorbereitung:

- Use Case auswählen und SSD ableiten
- Systemoperation identifizieren
- Operation Contract erstellen/prüfen

2. Analyse:

- Aktuellen Code/Dokumentation analysieren
- DCD überprüfen/aktualisieren
- Vergleich mit Domänenmodell
- Neue Klassen gemäß Domänenmodell erstellen

3. Realisierung:

- Controller Klasse bestimmen
- Zu verändernde Klassen festlegen
- Weg zu diesen Klassen festlegen:
  - Parameter für Wege definieren
  - Klassen bei Bedarf erstellen
  - Verantwortlichkeiten zuweisen
  - Verschiedene Varianten evaluieren
- Veränderungen implementieren
- Review durchführen

Use Case Realization: Verkauf abwickeln **1. Vorbereitung:**

- **Use Case:** Verkauf abwickeln
- **Systemoperation:** makeNewSale()
- **Contract:** Neue Sale-Instanz wird erstellt

2. Analyse:

- **Klassen:** Register, Sale
- **DCD:** Beziehung Register-Sale prüfen
- **Neue Klassen:** Payment, SaleLineItem

3. Implementierung:

- Register als Controller
- Sale-Klasse erweitern
- Beziehungen implementieren

Typische Implementierungsfehler vermeiden

- **Architekturverletzungen:**
  - Schichtentrennung beachten
  - Abhängigkeiten richtig setzen
- **GRASP-Verletzungen:**
  - Information Expert beachten
  - Creator Pattern richtig anwenden
  - High Cohesion erhalten
- **Testbarkeit:**
  - Klassen isoliert testbar halten
  - Abhängigkeiten mockbar gestalten

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

Grundlegende Design Patterns

Adapter Pattern

Problem: Inkompatible Schnittstellen

- Objekte mit unterschiedlichen Interfaces sollen zusammenarbeiten
- Externe Dienste sollen austauschbar sein

Lösung: Adapter-Klasse als Vermittler

Simple Factory Pattern

Problem: Komplexe Objekterzeugung

- Objekterzeugung erfordert viele Schritte
- Konfiguration bei Erzeugung notwendig

Lösung: Eigene Klasse für Objekterzeugung

Singleton Pattern

Problem: Genau eine Instanz benötigt

- Globaler Zugriffspunkt notwendig
- Mehrfachinstanzierung verhindern

Lösung: Statische Instanz mit privater Erzeugung

Dependency Injection Pattern

Problem: Abhängigkeiten zu anderen Objekten

- Lose Kopplung erwünscht
- Flexibilität bei Abhängigkeiten

Lösung: Abhängigkeiten werden von außen injiziert

Proxy Pattern

Problem: Zugriffskontrolle auf Objekte

- Verzögertes Laden
- Zugriffsbeschränkungen
- Netzwerkkommunikation

Lösung: Stellvertreterobjekt mit gleichem Interface

- **Remote Proxy:** Für entfernte Objekte
- **Virtual Proxy:** Für spätes Laden
- **Protection Proxy:** Für Zugriffsschutz

Chain of Responsibility Pattern

Problem: Unklare Zuständigkeit für Anfragen

- Mehrere mögliche Handler
- Zuständigkeit erst zur Laufzeit klar

Lösung: Verkettete Handler-Objekte

Erweiterte Design Patterns

Decorator Pattern

Problem: Dynamische Erweiterung von Objekten

- Zusätzliche Verantwortlichkeiten
- Nur für einzelne Objekte

Lösung: Wrapper-Objekt mit gleichem Interface

Observer Pattern

Problem: Abhängige Objekte aktualisieren

- Lose Kopplung erwünscht
- Typ des Empfängers unbekannt

Lösung: Observer-Interface für Benachrichtigungen

Strategy Pattern

Problem: Austauschbare Algorithmen

- Verschiedene Implementierungen
- Zur Laufzeit wechselbar

Lösung: Interface für Algorithmus-Klassen

Composite Pattern

Problem: Baumstrukturen verwalten

- Einheitliche Behandlung
- Teil-Ganzes Hierarchie

Lösung: Gemeinsames Interface für Container und Inhalt

Design Pattern Auswahl

Schritt 1: Problem analysieren

- Art des Problems identifizieren
- Anforderungen klar definieren
- Kontext verstehen

Schritt 2: Pattern evaluieren

- Passende Patterns suchen
- Vor- und Nachteile abwägen
- Komplexität bewerten

Schritt 3: Implementation planen

- Klassenstruktur entwerfen
- Schnittstellen definieren
- Anpassungen vornehmen

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**

```
// 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:**

```
// 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)

```
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

@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());
}
```

```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

```java
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());
    }
}
```

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

Entwicklungsansätze

Code-Driven Development (CDD):
- Direkte Implementierung der Klassen
- Nachträgliches Testing

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

Behavior-Driven Development (BDD):
- Testbeschreibung aus Anwendersicht
- Gherkin-Syntax für Szenarios

Clean Code

1. Code-Guidelines:
- Einheitliche Formatierung
- Klare Namenskonventionen
- Dokumentationsrichtlinien

2. Fehlerbehandlung:
- Exceptions statt Fehlercodes
- Sinnvolle Error Messages
- Logging-Strategie

3. Namensgebung:
- Aussagekräftige Namen
- Konsistente Begriffe
- Domain-Driven Naming

Laufzeit-Optimierung

Grundregeln:
- Zuerst messen, dann optimieren
- Performance-Profile nutzen
- Bottlenecks identifizieren

Häufige Probleme:
- Datenbank-Zugriffe
- Ineffiziente Algorithmen
- Speicherlecks

Refactoring

Refactoring Grundlagen

Strukturierte Verbesserung des Codes ohne Änderung des externen Verhaltens:
- Kleine, kontrollierte Schritte
- Erhaltung der Funktionalität
- Verbesserung der Codequalität

Refactoring Durchführung

1. Code Smells identifizieren:
- Duplizierter Code
- Lange Methoden
- Große Klassen
- Hohe Kopplung

2. Refactoring durchführen:
- Tests sicherstellen
- Änderungen vornehmen
- Tests ausführen

3. Patterns anwenden:
- Extract Method
- Move Method
- Rename
- Introduce Variable

Testing

Testarten

Nach Sicht:
- **Black-Box:** Funktionaler Test ohne Codekenntnis
- **White-Box:** Strukturbezogener Test mit Codekenntnis

Nach Umfang:
- **Unit-Tests:** Einzelne Komponenten
- **Integrationstests:** Zusammenspiel
- **Systemtests:** Gesamtsystem
- **Akzeptanztests:** Kundenanforderungen

Testentwicklung

1. Testfall definieren:
- Vorbedingungen festlegen
- Testdaten vorbereiten
- Erwartetes Ergebnis definieren

2. Test implementieren:
- Setup vorbereiten
- Testlogik schreiben
- Assertions definieren

3. Test ausführen:
- Automatisiert ausführen
- Ergebnisse prüfen
- Dokumentation erstellen

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>

    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

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

// 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...]

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

Charakteristika verteilter Systeme

Typische Merkmale moderner verteilter Systeme:

- **Skalierbarkeit:** Oft sehr große Systeme
- **Datenorientierung:** Zentrale Datenbanken
- **Interaktivität:** GUI und Batch-Verarbeitung
- **Nebenläufigkeit:** Parallele Benutzerinteraktionen
- **Konsistenz:** Hohe Anforderungen an Datenkonsistenz

Grundlegende Konzepte

1. Kommunikation:

- Remote Procedure Calls (RPC)
- Message Queuing
- Publish-Subscribe-Systeme

2. Fehlertoleranz:

- Replikation von Komponenten
- Failover-Mechanismen
- Fehlererkennung und -behandlung

3. Fehlersemantik:

- Konsistenzgarantien
- Recovery-Verfahren
- Kompensationsmechanismen

Architekturmuster

Grundlegende Architekturstile für verteilte Systeme:

- **Client-Server:** Zentraler Server, multiple Clients
- **Peer-to-Peer:** Gleichberechtigte Knoten
- **Publish-Subscribe:** Event-basierte Kommunikation

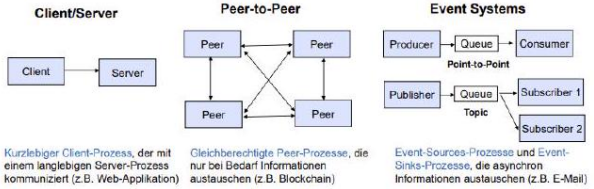


Abbildung 20: Architekturmodelle

Entwurf verteilter Systeme

1. Systemanalyse

- Anforderungen identifizieren
- Verteilungsaspekte analysieren
- Konsistenzanforderungen definieren

2. Architekturentscheidungen

- Architekturstil wählen
- Kommunikationsmuster festlegen
- Fehlertoleranzstrategie definieren

3. Technologieauswahl

- Middleware evaluieren
- Protokolle bestimmen
- Werkzeuge auswählen

Middleware-Technologien

Gängige Technologien für verteilte Systeme:

- **Message Broker:**
  - Apache Kafka
  - RabbitMQ
- **RPC Frameworks:**
  - gRPC
  - CORBA
- **Web Services:**
  - RESTful APIs
  - GraphQL

Typische Fehlerquellen

1. Netzwerkfehler

- Verbindungsabbrüche
- Timeouts
- Partitionierung

2. Konsistenzprobleme

- Race Conditions
- Veraltete Daten
- Lost Updates

3. Skalierungsprobleme

- Lastverteilung
- Resource-Management
- Bottlenecks

Lösungsstrategien:

- Circuit Breaker Pattern
- Retry mit Exponential Backoff
- Idempotente Operationen
- Optimistic Locking

Persistenz

Persistenz Grundlagen

Persistenz bezeichnet die dauerhafte Speicherung von Daten über das Programmende hinaus:

- Speicherung in Datenbankmanagementsystemen (DBMS)
- Haupttypen:
  - Relationale Datenbanksysteme (RDBMS)
  - NoSQL-Datenbanken (ohne fixes Schema)
- O/R-Mapping (Object Relational Mapping)
  - Abbildung zwischen Objekten und Datensätzen
  - Überwindung des Strukturbruchs (Impedance Mismatch)

O/R-Mismatch

Der Strukturbruch zwischen objektorientierter und relationaler Welt:

- **Typen-Systeme:**
  - Unterschiedliche NULL-Behandlung
  - Datum/Zeit-Darstellung
- **Beziehungen:**
  - Richtung der Beziehungen
  - Mehrfachbeziehungen
  - Vererbung
- **Identität:**
  - OO: Implizite Objektidentität
  - DB: Explizite Identität (Primary Key)

JDBC - Java Database Connectivity

JDBC Grundlagen

JDBC ist die standardisierte Schnittstelle für Datenbankzugriffe in Java:

- Seit JDK 1.1 (1997)
- Plattformunabhängig
- Datenbankunabhängig
- Aktuelle Version: 4.2

JDBC Verwendung

Grundlegende Schritte für Datenbankzugriff:

1. JDBC-Treiber installieren und laden
2. Verbindung zur Datenbank aufbauen
3. SQL-Statements ausführen
4. Ergebnisse verarbeiten
5. Transaktion abschließen (Commit/Rollback)
6. Verbindung schließen

Design Patterns für Persistenz

Persistenz Design Patterns

Drei grundlegende Ansätze für die Persistenzschicht:

- **Active Record (Anti-Pattern):**
  - Entität verwaltet eigene Persistenz
  - Vermischung von Fachlichkeit und Technik
  - Schlechte Testbarkeit
- **Data Access Object (DAO):**
  - Kapselung des Datenbankzugriffs
  - Trennung von Fachlichkeit und Technik
  - Gute Testbarkeit durch Mocking
- **Repository (DDD):**
  - Abstraktionsschicht über Data-Mapper
  - Zentralisierung von Datenbankabfragen
  - Komplexere Implementierung

DAO Implementation

Schritte zur Implementierung eines DAOs:

1. Interface definieren:
   - CRUD-Methoden (Create, Read, Update, Delete)
   - Spezifische Suchmethoden
2. Domänenklasse erstellen:
   - Nur fachliche Attribute
   - Keine Persistenzlogik
3. DAO-Implementierung:
   - Datenbankzugriff kapseln
   - O/R-Mapping implementieren
   - Transaktionshandling

Java Persistence API (JPA)

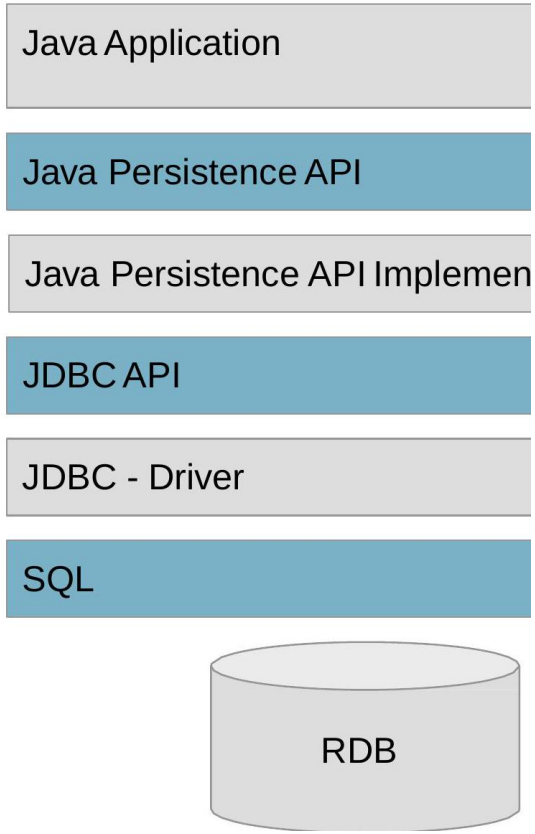
JPA Grundkonzepte

JPA ist der Java-Standard für O/R-Mapping:

- **Entity-Klassen:**
  - Plain Old Java Objects (POJOs)
  - Annotation @Entity
  - Keine JPA-spezifischen Abhängigkeiten
- **Referenzen:**
  - Eager/Lazy Loading
  - Automatisches Nachladen
- **Provider:**
  - Hibernate
  - EclipseLink
  - OpenJPA

JPA Technologie-Stack

- Java Application
- Java Persistence API
- JPA Provider (Hibernate, EclipseLink, etc.)
- JDBC Driver
- Relationale Datenbank



JPA Entity Erstellung

1. Entity-Klasse definieren:
   - @Entity Annotation
   - ID-Feld mit @Id markieren
2. Beziehungen definieren:
   - @OneToMany, @ManyToOne etc.
   - Navigationsrichtung festlegen
3. Validierung hinzufügen:
   - @NotNull, @Size etc.
   - Geschäftsregeln

Repository Pattern

Repository Pattern

Das Repository Pattern bietet eine zusätzliche Abstraktionsschicht über der Data-Mapper-Schicht:

- Zentralisierung von Datenbankabfragen
- Domänenorientierte Schnittstelle
- Unterstützung komplexer Abfragen
- Häufig in Kombination mit Spring Data

Spring Data unterstützt die automatische Generierung von Repository-Implementierungen basierend auf Methodennamen. Dies reduziert den Implementierungsaufwand erheblich.

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 Schl
    public List<Customer> findCustomers(String name)
        String sql = "SELECT * FROM customers WHERE
        try (Connection conn = dataSource.getConnect
             PreparedStatement stmt = conn.prepareSta

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

            // 4. ResultSet verarbeiten
            try (ResultSet rs = stmt.executeQuery())
                List<Customer> customers = new Array
                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 customerRepository;

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

        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 customerName) {
        return orderRepository
            .findByCustomerName(customerName);
    }
}
```

[Previous content about Repository Pattern remains...]

Spring Data Repository Features

```java
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();
}
```

[Previous content about Framework basics remains]

Prüfungsaufgabe: Framework-Analyse **Szenario:** Ein Framework für die Verarbeitung verschiedener Dokumentformate (PDF, DOC, TXT) soll entwickelt werden.
Aufgabe: Analysieren Sie die Design-Entscheidungen.
Lösung:

- **Erweiterungspunkte:**
  - Dokumenttyp-Erkennung
  - Parser für Formate
  - Konvertierungslogik
- **Design Patterns:**
  - Factory für Parser-Erzeugung
  - Strategy für Verarbeitungsalgorithmen
  - Template Method für Konvertierung
- **Schnittstellen:**
  - DocumentParser Interface
  - ConversionStrategy Interface
  - DocumentMetadata Klasse

Framework Design Principles **1. Abstraktionsebenen definieren**

- **Core API:**
  - Zentrale Interfaces
  - Hauptfunktionalität
  - Erweiterungspunkte
- **Extensions:**
  - Plugin-Mechanismen
  - Callback-Interfaces
  - Event-Systeme
- **Implementierung:**
  - Standard-Implementierungen
  - Utility-Klassen
  - Helper-Funktionen

2. Erweiterungsmechanismen

- **Interface-basiert:**
  - Klare Verträge
  - Lose Kopplung
  - Einfache Erweiterung
- **Annotations:**
  - Deklarative Konfiguration
  - Metadaten-getrieben
  - Runtime-Processing
- **Composition:**
  - Plugin-System
  - Service-Loader
  - Dependency Injection

Framework Design Pattern Anwendung **Aufgabe:** Implementieren Sie ein Plugin-System mit verschiedenen Design Patterns.
Analyse der Pattern-Kombination:

- **Abstract Factory:**
  - Plugin-Familie erzeugen
  - Zusammengehörige Komponenten
  - Austauschbare Implementierungen
- **Template Method:**
  - Plugin-Lifecycle definieren
  - Standardablauf vorgeben
  - Erweiterungspunkte bieten
- **Command:**
  - Plugin-Aktionen kapseln
  - Asynchrone Ausführung
  - Undo-Funktionalität

Framework Evaluation **1. Qualitätskriterien**

- **Usability:**
  - Intuitive API
  - Gute Dokumentation
  - Beispiele/Templates
- **Flexibilität:**
  - Erweiterbarkeit
  - Konfigurierbarkeit
  - Modularität
- **Wartbarkeit:**
  - Klare Struktur
  - Testbarkeit
  - Versionierung

2. Risikobewertung

- **Technisch:**
  - Kompatibilität
  - Performance
  - Skalierbarkeit
- **Organisatorisch:**
  - Learning Curve
  - Support/Community
  - Zukunftssicherheit

Typische Prüfungsaufgabe: Framework Migration **Szenario:** Ein bestehendes System soll von einem proprietären Framework auf ein Standard-Framework migriert werden.
Aufgabenstellung:

- Analysieren Sie die Herausforderungen
- Entwickeln Sie eine Migrationsstrategie
- Bewerten Sie Risiken

Lösungsansatz:

- **Analyse:**
  - Framework-Abhängigkeiten identifizieren
  - Geschäftskritische Funktionen isolieren
  - Testabdeckung prüfen
- **Strategie:**
  - Adapter für Framework-Bridging
  - Schrittweise Migration
  - Parallelbetrieb ermöglichen
- **Risikominimierung:**
  - Automated Testing
  - Feature Toggles
  - Rollback-Möglichkeit

[Previous content about modern framework patterns remains]

Framework Design

Framework Grundlagen

Ein Framework ist ein Programmiergerüst mit folgenden Eigenschaften:

- Bietet wiederverwendbare Funktionalität
- Definiert Erweiterungs- und Anpassungspunkte
- Verwendet Design Patterns
- Enthält keinen applikationsspezifischen Code
- Gibt Rahmen für anwendungsspezifischen Code vor
- Klassen arbeiten eng zusammen (vs. reine Bibliothek)

Framework Entwicklung

Die Entwicklung eines Frameworks erfordert:

- Höhere Zuverlässigkeit als normale Software
- Tiefergehende Analyse der Erweiterungspunkte
- Hoher Architektur- und Designaufwand
- Sorgfältige Planung der Schnittstellen

Kritische Betrachtung
Herausforderungen beim Framework-Einsatz:

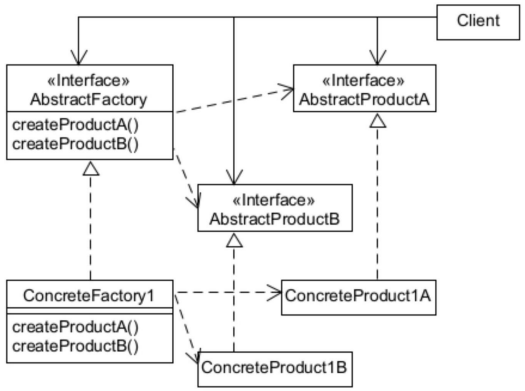
- Frameworks tendieren zu wachsender Funktionalität
- Gefahr von inkonsistentem Design
- Funktionale Überschneidungen möglich
- Hoher Einarbeitungsaufwand
- Schwierige SScheidung"nach Integration
- Trade-off zwischen Abhängigkeit und Nutzen

Design Patterns in Frameworks

Abstract Factory

Problem: Erzeugung verschiedener, zusammengehörender Objekte ohne Kenntnis konkreter Klassen
Lösung:

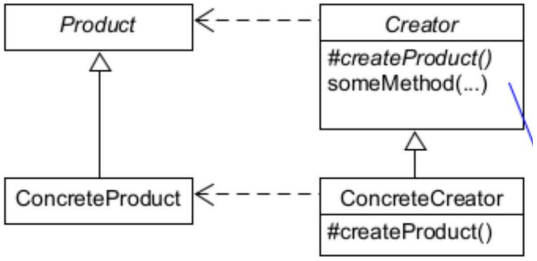
- AbstractFactory-Interface definieren
- Pro Produkt eine create-Methode
- Konkrete Factories implementieren Interface



Factory Method

Problem: Flexible Objekterzeugung in wiederverwendbarer Klasse
Lösung:

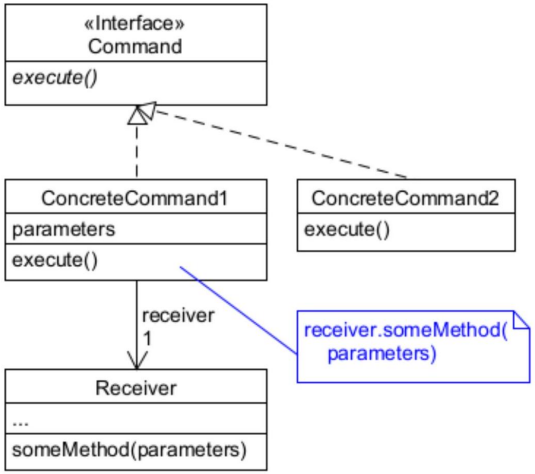
- Abstrakte Factory-Methode in Creator-Klasse
- Konkrete Subklassen überschreiben Methode
- Parallele Vererbungshierarchien



Command

Problem: Aktionen für späteren Gebrauch speichern und verwalten
Lösung:

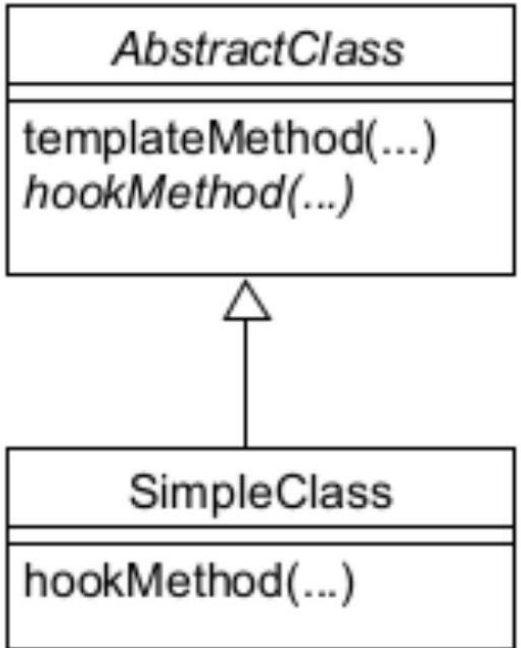
- Command-Interface definieren
- Konkrete Commands implementieren
- Parameter für Ausführung speichern
- Optional: Undo-Funktionalität



Template Method

Problem: Algorithmus mit anpassbaren Teilschritten
Lösung:

- Template Method in abstrakter Klasse
- Hook-Methoden für variable Teile
- Hollywood Principle: "Don't call us, we'll call you"



Moderne Framework Patterns

Annotation-basierte Konfiguration

Moderne Frameworks nutzen Annotationen für:

- Dependency Injection
- Konfiguration
- Interface-Implementation
- Funktionalitätserweiterung

Framework Integration

1. **Convention over Configuration**
   - Namenskonventionen einhalten
   - Standard-Verhalten nutzen
   - Nur Ausnahmen konfigurieren
2. **Dependency Injection**
   - Abhängigkeiten deklarieren
   - Framework übernimmt Injection
   - Constructor- oder Setter-Injection
3. **Interface-basierte Entwicklung**
   - Interfaces definieren
   - Framework generiert Implementation
   - Methodennamen als Spezifikation

Annotation-basierte Frameworks bieten:

- Geringere Kopplung zur Framework-API
- Deklarativen Programmierstil
- Reduzierte Boilerplate-Code
- Kann aber zu längeren Startzeiten führen