Software Entwicklung

Jil Zerndt, Lucien Perret
May 2024

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.

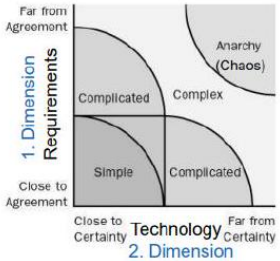
Wrap-up

- 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

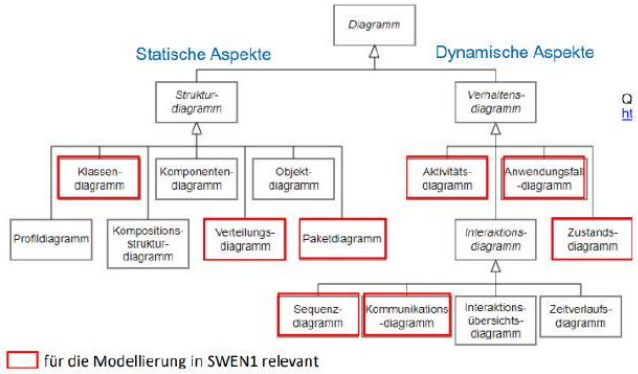
- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

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

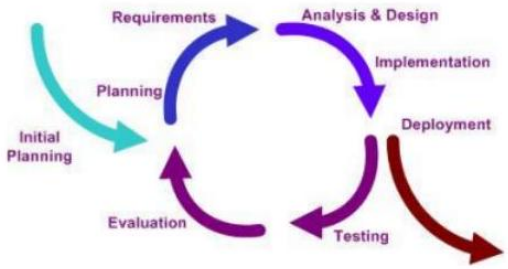
Modell von Requirements (close to/ far from Agreement) & Technology (known / unknown)
Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhanden Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).

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



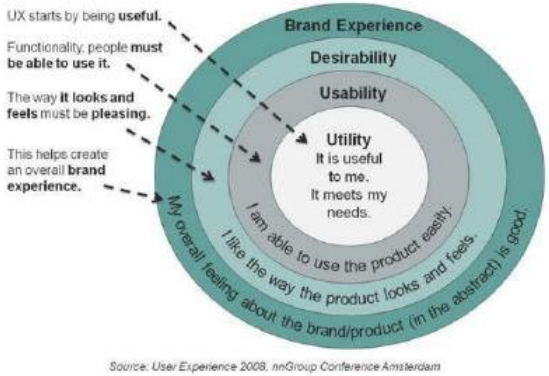
Anforderungsanalyse

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

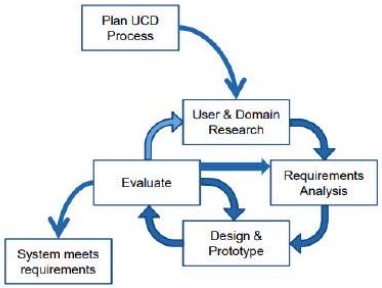
ISO 9241-110: Usability-Anforderungen

Die sieben Grundprinzipien:

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

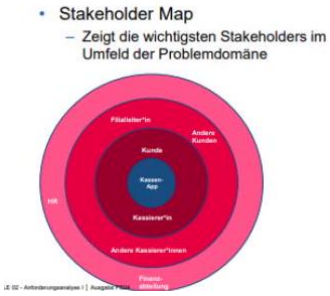
User-Centered Design (UCD)

Ein iterativer Prozess zur nutzerzentrierten Entwicklung:



Wichtige Artefakte

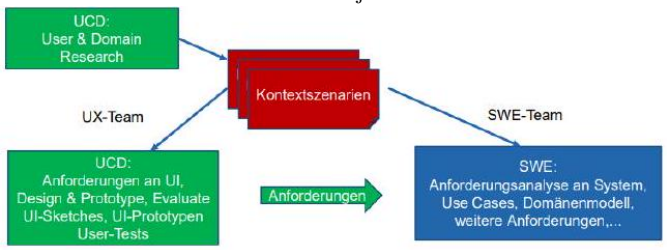
- Personas: Repräsentative Nutzerprofile
- Usage-Szenarien: Konkrete Anwendungsfälle
- Mentales Modell: Nutzerverständnis
- Domänenmodell: Fachliches Verständnis
- Service Blueprint: Geschäftsprozessmodell
- Stakeholder Map: Beteiligte und Betroffene
- UI-Artefakte: Skizzen, Wireframes, Designs



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.

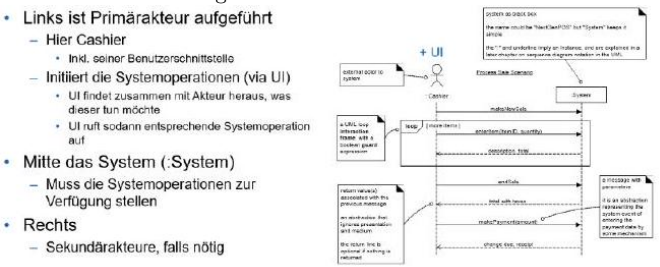
Fully-dressed Use Case **UC: Verkauf abwickeln**

- **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



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

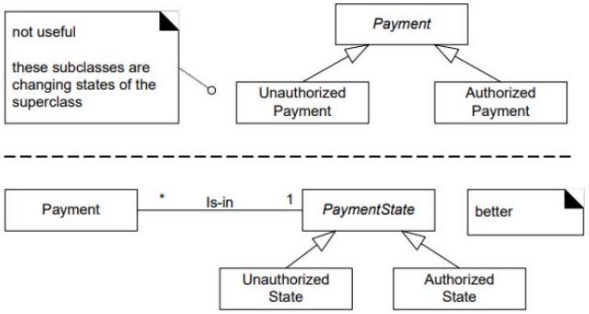
- 100% Regel: Alle Instanzen der Spezialisierung sind auch Instanzen der Generalisierung
- İS-A"Beziehung
- Gemeinsame Attribute/Assoziationen als Grund für Generalisierung

3. Komposition

- Starke Teil-Ganzes Beziehung
- Existenzabhängigkeit der Teile

4. Zustandsmodellierung

- Zustände als eigene Hierarchie
- Vermeidet problematische Vererbung

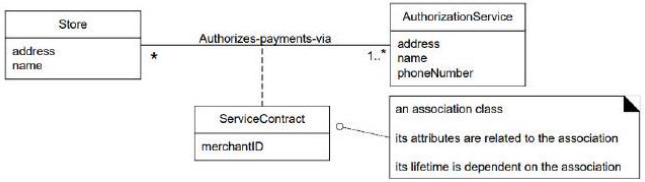


5. Rollen

- 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

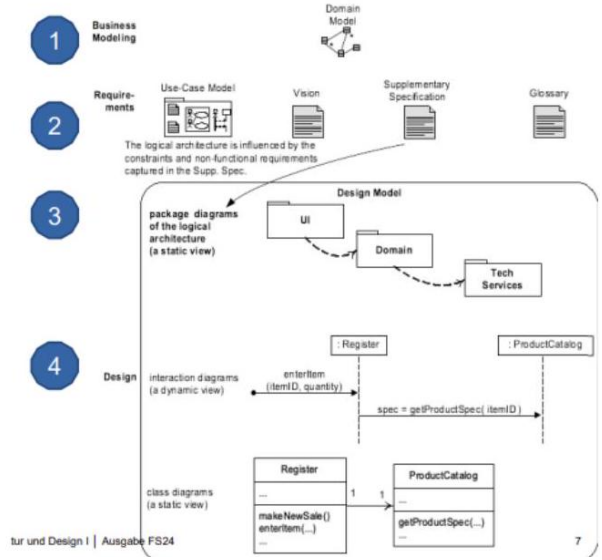
- **Keine Softwareklassen modellieren**
  - Manager-Klassen vermeiden
  - Keine technischen Helper-Klassen
- **Keine Operationen modellieren**
  - Fokus auf Struktur, nicht Verhalten
  - Keine CRUD-Operationen
- **Richtige Abstraktionsebene wählen**
  - Nicht zu detailliert
  - Nicht zu abstrakt
  -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

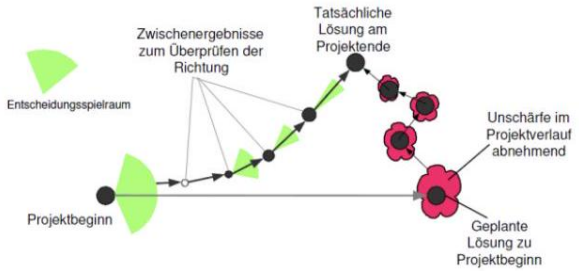
Softwarearchitektur und Design

Überblick Softwareentwicklung

Die Entwicklung von Software erfolgt in verschiedenen Ebenen:

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





Softwarearchitektur

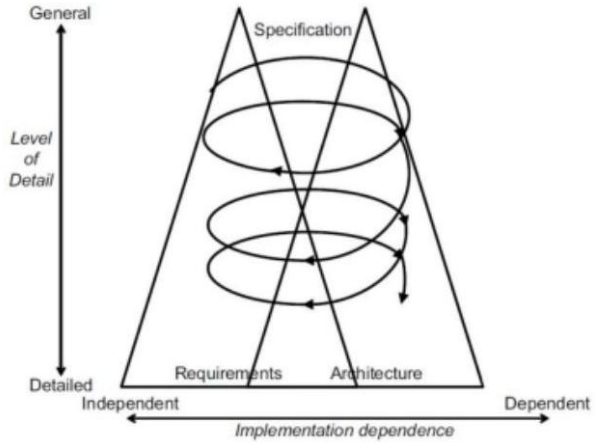
Die Architektur definiert:

- 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:

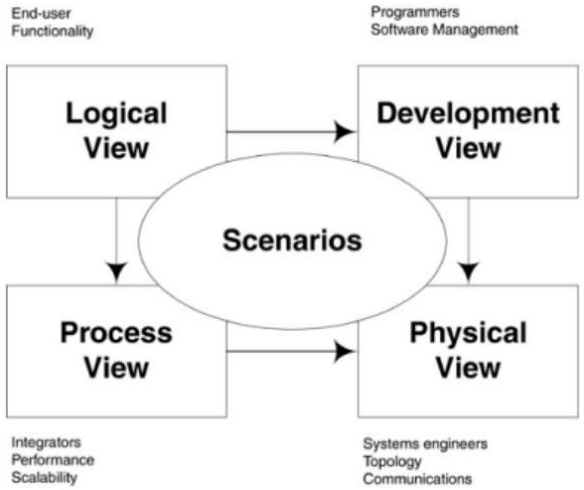
- **Kohäsion:** Innerer Zusammenhang
- **Kopplung:** Externe Abhängigkeiten

Eigenschaften:

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

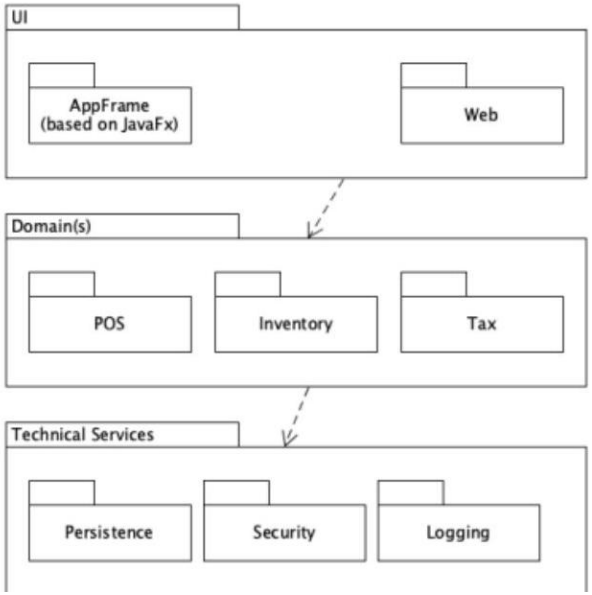
Architektursichten

Das N+1 View Model beschreibt verschiedene Perspektiven:



UML-Paketdiagramm:

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



UML-Modellierung

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

Factory Method Implementation

Aufgabe: Implementieren Sie eine Factory für verschiedene Dokumenttypen (PDF, Word, Text)

Lösung:

```
// Interface fuer Produkte
interface Document {
    void open();
    void save();
}

// Konkrete Produkte
class PdfDocument implements Document {
    public void open() { /* ... */ }
    public void save() { /* ... */ }
}

// Factory Method Pattern
abstract class DocumentCreator {
    abstract Document createDocument();

    // Template Method
    final void processDocument() {
        Document doc = createDocument();
        doc.open();
        doc.save();
    }
}

// Konkrete Factory
class PdfDocumentCreator extends DocumentCreator {
    Document createDocument() {
        return new PdfDocument();
    }
}
```

Observer Pattern Implementation

Aufgabe: Implementieren Sie ein Benachrichtigungssystem für Aktienkurse

Lösung:

```
interface StockObserver {
    void update(String stock, double price);
}

class StockMarket {
    private List<StockObserver> observers = new
        ArrayList<>();

    public void attach(StockObserver observer) {
        observers.add(observer);
    }

    public void notifyObservers(String stock, double
        price) {
        for(StockObserver observer : observers) {
            observer.update(stock, price);
        }
    }
}

class StockDisplay implements StockObserver {
    public void update(String stock, double price) {
        System.out.println("Stock: " + stock +
                           " updated to " + price);
    }
}
```

Adapter Pattern

Szenario: Altbestand an Drittanbieter-Bibliothek integrieren

```
// Bestehende Schnittstelle
interface ModernPrinter {
    void printDocument(String content);
}

// Alte Drittanbieter-Klasse
class LegacyPrinter {
    public void print(String[] pages) {
        for(String page : pages) {
            System.out.println(page);
        }
    }
}

// Adapter
class PrinterAdapter implements ModernPrinter {
    private LegacyPrinter legacyPrinter;

    public PrinterAdapter(LegacyPrinter printer) {
        this.legacyPrinter = printer;
    }

    public void printDocument(String content) {
        String[] pages = content.split("\n");
        legacyPrinter.print(pages);
    }
}
```

Simple Factory

Szenario: Erzeugung von verschiedenen Datenbankverbindungen

```
class DatabaseFactory {
    public static Database createDatabase(String type)
        {
        switch(type) {
            case "MySQL":
                return new MySQLDatabase();
            case "PostgreSQL":
                return new PostgreSQLDatabase();
            default:
                throw new
                    IllegalArgumentException("Unknown
                    DB type");
        }
    }
}

// Verwendung
Database db = DatabaseFactory.createDatabase("MySQL");
```

Singleton

Szenario: Globale Konfigurationsverwaltung

```
public class Configuration {
    private static Configuration instance;
    private Map<String, String> config;

    private Configuration() {
        config = new HashMap<>();
    }

    public static Configuration getInstance() {
        if(instance == null) {
            instance = new Configuration();
        }
        return instance;
    }
}
```

Dependency Injection

Szenario: Flexible Logger-Implementation

```
interface Logger {
    void log(String message);
}

class FileLogger implements Logger {
    public void log(String message) {
        // Log to file
    }
}

class UserService {
    private final Logger logger;

    public UserService(Logger logger) { // Dependency
        Injection
        this.logger = logger;
    }
}
```

Proxy
Szenario: Verzögertes Laden eines großen Bildes

```
interface Image {
    void display();
}

class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " + filename);
    }

    public void display() {
        System.out.println("Displaying " + filename);
    }
}

class ImageProxy implements Image {
    private RealImage realImage;
    private String filename;

    public ImageProxy(String filename) {
        this.filename = filename;
    }

    public void display() {
        if(realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

Chain of Responsibility
Szenario: Authentifizierungskette

```
abstract class AuthHandler {
    protected AuthHandler next;

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

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

class LocalAuthHandler extends AuthHandler {
    public boolean handle(String username, String
        password) {
        if(checkLocalDB(username, password)) {
            return true;
        }
        return next != null ? next.handle(username,
            password) : false;
    }
}

class LDAPAuthHandler extends AuthHandler {
    public boolean handle(String username, String
        password) {
        if(checkLDAP(username, password)) {
            return true;
        }
        return next != null ? next.handle(username,
            password) : false;
    }
}
```

Decorator
Szenario: Dynamische Erweiterung eines Text-Editors

```
interface TextComponent {
    String render();
}

class SimpleText implements TextComponent {
    private String text;

    public SimpleText(String text) {
        this.text = text;
    }

    public String render() {
        return text;
    }
}

class BoldDecorator implements TextComponent {
    private TextComponent component;

    public BoldDecorator(TextComponent component) {
        this.component = component;
    }

    public String render() {
        return "<b>" + component.render() + "</b>";
    }
}
```

Observer
Szenario: News-Benachrichtigungssystem

```
interface NewsObserver {
    void update(String news);
}

class NewsAgency {
    private List<NewsObserver> observers = new
        ArrayList<>();

    public void addObserver(NewsObserver observer) {
        observers.add(observer);
    }

    public void notifyObservers(String news) {
        for(NewsObserver observer : observers) {
            observer.update(news);
        }
    }
}

class NewsChannel implements NewsObserver {
    private String name;

    public NewsChannel(String name) {
        this.name = name;
    }

    public void update(String news) {
        System.out.println(name + " received: " +
            news);
    }
}
```

Strategy
Szenario: Verschiedene Zahlungsmethoden

```
interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public void pay(int amount) {
        System.out.println("Paid " + amount + " using
            Credit Card");
    }
}

class PayPalPayment implements PaymentStrategy {
    private String email;

    public void pay(int amount) {
        System.out.println("Paid " + amount + " using
            PayPal");
    }
}
```

Composite
Szenario: Dateisystem-Struktur

```
interface FileSystemComponent {
    void list(String prefix);
}

class File implements FileSystemComponent {
    private String name;

    public void list(String prefix) {
        System.out.println(prefix + name);
    }
}

class Directory implements FileSystemComponent {
    private String name;
    private List<FileSystemComponent> children = new
        ArrayList<>();

    public void add(FileSystemComponent component) {
        children.add(component);
    }

    public void list(String prefix) {
        System.out.println(prefix + name);
        for(FileSystemComponent child : children) {
            child.list(prefix + "  ");
        }
    }
}
```

State
Szenario: Verkaufsautomat

```
interface VendingMachineState {
    void insertCoin();
    void ejectCoin();
    void selectProduct();
    void dispense();
}

class HasCoinState implements VendingMachineState {
    private VendingMachine machine;

    public void selectProduct() {
        System.out.println("Product selected");
        machine.setState(machine.getSoldState());
    }

    public void insertCoin() {
        System.out.println("Already have coin");
    }
}

class VendingMachine {
    private VendingMachineState currentState;

    public void setState(VendingMachineState state) {
        this.currentState = state;
    }

    public void insertCoin() {
        currentState.insertCoin();
    }
}
```

Visitor
Szenario: Dokumentstruktur mit verschiedenen Operationen

```
interface DocumentElement {
    void accept(Visitor visitor);
}

interface Visitor {
    void visit(Paragraph paragraph);
    void visit(Heading heading);
}

class HTMLVisitor implements Visitor {
    public void visit(Paragraph p) {
        System.out.println("<p>" + p.getText() +
            "</p>");
    }

    public void visit(Heading h) {
        System.out.println("<h1>" + h.getText() +
            "</h1>");
    }
}
```

Facade
Szenario: Vereinfachte Multimedia-Bibliothek

```
class MultimediaFacade {
    private AudioSystem audio;
    private VideoSystem video;
    private SubtitleSystem subtitles;

    public void playMovie(String movie) {
        audio.initialize();
        video.initialize();
        subtitles.load(movie);
        video.play(movie);
        audio.play();
    }
}
```

Abstract Factory
Szenario: GUI-Elemente für verschiedene Betriebssysteme

```
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

class WindowsFactory implements GUIFactory {
    public Button createButton() {
        return new WindowsButton();
    }

    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}

class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }

    public Checkbox createCheckbox() {
        return new MacCheckbox();
    }
}
```

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

Extract Method Refactoring

Vorher:

```
void printOwing() {
    printBanner();

    // calculate outstanding
    double outstanding = 0.0;
    for (Order order : orders) {
        outstanding += order.getAmount();
    }

    // print details
    System.out.println("name: " + name);
    System.out.println("amount: " + outstanding);
}
```

Nachher:

```
void printOwing() {
    printBanner();
    double outstanding = calculateOutstanding();
    printDetails(outstanding);
}

double calculateOutstanding() {
    double result = 0.0;
    for (Order order : orders) {
        result += order.getAmount();
    }
    return result;
}

void printDetails(double outstanding) {
    System.out.println("name: " + name);
    System.out.println("amount: " + outstanding);
}
```

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

Unit Test

Zu testende Klasse:

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Test:

```
@Test
public class CalculatorTest {
    private Calculator calc;

    @Before
    public void setup() {
        calc = new Calculator();
    }

    @Test
    public void testAdd() {
        assertEquals(4, calc.add(2, 2));
        assertEquals(0, calc.add(-2, 2));
        assertEquals(-4, calc.add(-2, -2));
    }
}
```

BDD Test

Feature File:

```
Feature: Calculator Addition
  Scenario: Add two positive numbers
    Given I have a calculator
    When I add 2 and 2
    Then the result should be 4

  Scenario: Add positive and negative numbers
    Given I have a calculator
    When I add -2 and 2
    Then the result should be 0
```

Step Definitions:

```
public class CalculatorSteps {
    private Calculator calc;
    private int result;

    @Given("I have a calculator")
    public void createCalculator() {
        calc = new Calculator();
    }

    @When("I add {int} and {int}")
    public void addNumbers(int a, int b) {
        result = calc.add(a, b);
    }

    @Then("the result should be {int}")
    public void checkResult(int expected) {
        assertEquals(expected, result);
    }
}
```

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

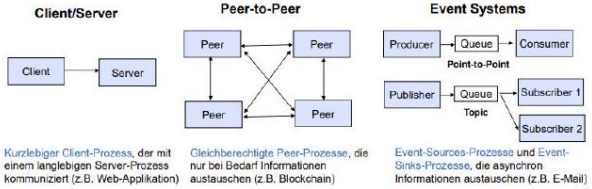


Abbildung 20: Architekturmodelle

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

Client-Server Implementation
Aufgabe: Implementieren Sie einen einfachen Echo-Server mit Java.
Lösung:

```java
// Server
public class EchoServer {
    public static void main(String[] args) {
        try (ServerSocket server = new
             ServerSocket(8080)) {
            while (true) {
                Socket client = server.accept();
                new Thread(() ->
                    handleClient(client)).start();
            }
        }
    }

    private static void handleClient(Socket client) {
        try (
            BufferedReader in = new BufferedReader(
                new
                    InputStreamReader(client.getInputStream
            PrintWriter out = new PrintWriter(
                client.getOutputStream(), true)
        ) {
            String line;
            while ((line = in.readLine()) != null) {
                out.println("Echo: " + line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// Client
public class EchoClient {
    public static void main(String[] args) {
        try (
            Socket socket = new Socket("localhost",
                 8080);
            PrintWriter out = new PrintWriter(
                socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new
                    InputStreamReader(socket.getInputStream
        ) {
            out.println("Hello Server!");
            System.out.println(in.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Publish-Subscribe Pattern
Aufgabe: Implementieren Sie ein einfaches Event-System.
Lösung:

```java
public class EventBus {
    private Map<String, List<EventHandler>> handlers =
        new HashMap<>();

    public void subscribe(String event, EventHandler
        handler) {
        handlers.computeIfAbsent(event, k -> new
            ArrayList<>())
            .add(handler);
    }

    public void publish(String event, String data) {
        if (handlers.containsKey(event)) {
            handlers.get(event)
                .forEach(handler ->
                    handler.handle(data));
        }
    }
}

interface EventHandler {
    void handle(String data);
}

// Verwendung
EventBus bus = new EventBus();
bus.subscribe("userLogin", data ->
    System.out.println("User logged in: " + data));
bus.publish("userLogin", "john_doe");
```

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

JDBC Basisbeispiel

```java
import java.sql.*;

public class DbTest {
    public static void main(String[] args)
            throws SQLException {
        // Verbindung aufbauen
        Connection con = DriverManager.getConnection(
            "jdbc:postgresql://test.zhaw.ch/testdb",
            "user", "password");

        // Statement erstellen und ausfuehren
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(
            "SELECT * FROM test ORDER BY name");

        // Ergebnisse verarbeiten
        while (rs.next()) {
            System.out.println(
                "Name: " + rs.getString("name"));
        }

        // Aufraeumen
        rs.close();
        stmt.close();
        con.close();
    }
}
```

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

DAO Implementation

```java
public interface ArticleDAO {
    void insert(Article item);
    void update(Article item);
    void delete(Article item);
    Article findById(int id);
    Collection<Article> findAll();
    Collection<Article> findByName(String name);
}

public class Article {
    private long id;
    private String name;
    private float price;

    // Getter/Setter
}

public class JdbcArticleDAO implements
    ArticleDAO {
    private Connection conn;

    public void insert(Article item) {
        PreparedStatement stmt =
            conn.prepareStatement(
            "INSERT INTO articles (name, price)
                VALUES (?, ?)");
        stmt.setString(1, item.getName());
        stmt.setFloat(2, item.getPrice());
        stmt.executeUpdate();
    }
    // weitere Implementierungen
}
```

Java Persistence API (JPA)

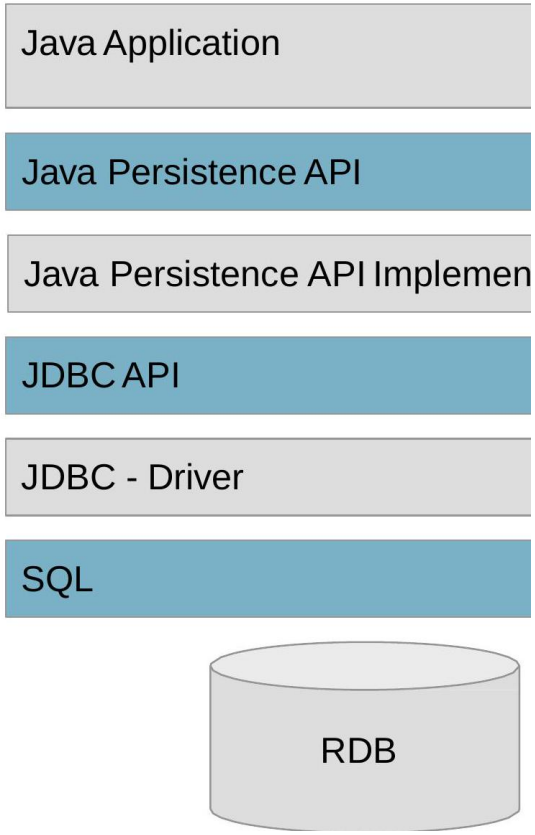
JPA Grundkonzepte

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

Parent-Child Beziehung mit JPA

```java
@Entity
public class Department {
    @Id @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;
}

@Entity
public class Employee {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    private String name;
    private double salary;
}
```

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 Repository

```java
@Repository
public interface SaleRepository
        extends CrudRepository<Sale, String> {

    List<Sale> findOrderByDateTime();

    List<Sale> findByDateTime(
        final LocalDateTime dateTime);
}

@Service
public class ProcessSaleHandler {
    private final ProductDescriptionRepository catalog;
    private final SaleRepository saleRepository;

    @Transactional
    public void endSale() {
        assert(currentSale != null
            && !currentSale.isComplete());
        this.currentSale.becomeComplete();
        this.saleRepository.save(currentSale);
    }
}
```

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

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:

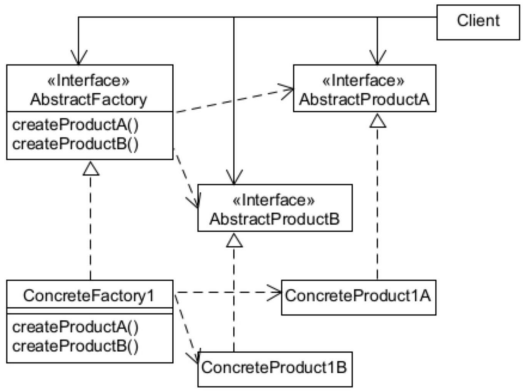
- Frameworks tendieren zu wachsender Funktionalität
- Gefahr von inkonsistentem Design
- Funktionale Überschneidungen möglich
- Hoher Einarbeitungsaufwand
- Schwierige SScheidung"nach Integration
- Trade-off zwischen Abhängigkeit und Nutzen

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



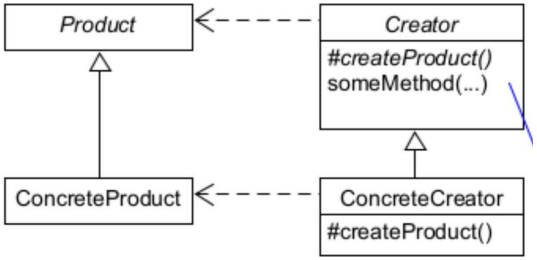
Abstract Factory: POS Terminal

```java
public interface IJavaPOSDevicesFactory {
    CashDrawer getNewCashDrawer();
    CoinDispenser getNewCoinDispenser();
    // weitere Methoden
}

public class IBMJavaPOSDevicesFactory
        implements IJavaPOSDevicesFactory {
    public CashDrawer getNewCashDrawer() {
        return new com.ibm.pos.jpos.CashDrawer();
    }
    // weitere Implementierungen
}
```

Factory Method

Problem: Flexible Objekterzeugung in wiederverwendbarer Klasse
Lösung:

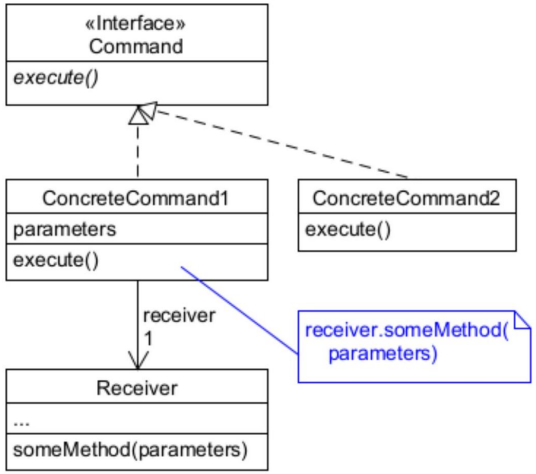
- Abstrakte Factory-Methode in Creator-Klasse
- Konkrete Subklassen überschreiben Methode
- Parallele Vererbungshierarchien



Command

Problem: Aktionen für späteren Gebrauch speichern und verwalten
Lösung:

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



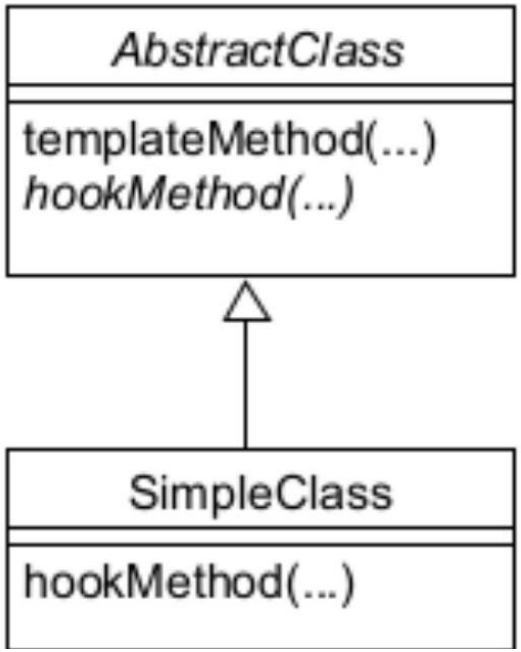
Command: Persistenz

```java
public interface ICommand {
    void execute();
    void undo();
}

public class DBUpdateCommand implements ICommand {
    private PersistentObject object;

    public void execute() {
        // Update in Datenbank
    }

    public void undo() {
        // Aenderung rueckgaengig machen
    }
}
```

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"



Template Method: GUI Framework

```java
public abstract class GUIComponent {
    // Template Method
    public final void update() {
        clearBackground();
        repaint(); // Hook Method
    }

    protected abstract void repaint();
}

public class MyButton extends GUIComponent {
    protected void repaint() {
        // Button-spezifische Implementation
    }
}
```

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

Spring Data Repository

```java
@Repository
public interface UserRepository
        extends JpaRepository<User, Long> {
    // Methode wird automatisch implementiert
    List<User> findByLastNameOrderByFirstNameAsc(
        String lastName);

    // SQL-Query via Annotation
    @Query("SELECT u FROM User u WHERE u.active =
        true")
    List<User> findActiveUsers();
}
```

Annotation-basierte Frameworks bieten:

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

Zusammenfassung

Iterativ-Inkrementeller Entwicklungsprozess

Der Softwareentwicklungsprozess in SWEN1/PM3:

- **Iterationen:**
  - 2-Wochen-Rhythmus
  - Definierte Ziele pro Iteration
  - Review nach Abschluss
- **Meilensteine:**
  - M1: Projektskizze
  - M2: Lösungsarchitektur
  - M3: Beta-Release
- **Pro Iteration:**
  - Anforderungsanalyse
  - Design
  - Implementation
  - Testing

Zentrale Artefakte

Die wichtigsten Ergebnisse im Entwicklungsprozess:

- **Anforderungsanalyse:**
  - Funktionale Anforderungen (Use Cases)
  - Qualitätsanforderungen und Randbedingungen
  - Domänenmodell
- **Design:**
  - Softwarearchitektur
  - Use Case Realisierung
  - Statische und dynamische Modelle
- **Implementation:**
  - Quellcode mit Javadoc
  - Refactoring bei Code Smells
- **Testing:**
  - Unit-Tests
  - Integrationstests
  - Systemtests

UML in der Praxis

Nach Martin Fowler gibt es drei Haupteinsatzarten:

- **UML as a Sketch:**
  - Informelle Diagramme
  - Kommunikationswerkzeug
  - Bevorzugt in agiler Entwicklung
- **UML as a Blueprint:**
  - Detaillierte Analyse/Design
  - Code-Generierung
  - Reverse-Engineering
- **UML as a Programming Language:**
  - Ausführbare Spezifikation
  - MDA-Werkzeuge

Objektorientierte Analyse (OOA)

Zentrale Aktivitäten der Anforderungsanalyse:

- **User Research:**
  - Personas entwickeln
  - Contextual Inquiry durchführen
  - Prototyping und Sketching
- **Use Cases:**
  - Modellierung und Dokumentation
  - UML-Use-Case-Diagramme
  - UI-Sketching
- **Domänenmodellierung:**
  - Konzeptuelles Klassenmodell
  - Fachbegriffe und Beziehungen
  - Problembezogene Sicht

Objektorientiertes Design (OOD)

Zentrale Design-Aktivitäten:

- **Architektur:**
  - UML-Paketdiagramm
  - UML-Verteilungsdiagramm
- **Use-Case Realisierung:**
  - Klassendesign mit Verantwortlichkeiten
  - Statische Modelle (Klassendiagramm)
  - Dynamische Modelle (Sequenz-, Zustands-, Aktivitätsdiagramme)
- **Design Patterns:**
  - GRASP Prinzipien
  - GoF Patterns
  - Architektur Patterns

Implementation und Testing

Implementation:

- Umsetzung des OO-Designs in Code
- Algorithmen und Datenstrukturen
- Kontinuierliches Refactoring
- Clean Code Prinzipien

Testing:

- Test-Driven Development
- Verschiedene Teststufen
- Testkonzept und -dokumentation