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.

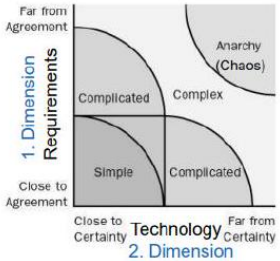
Wrap-up

- 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

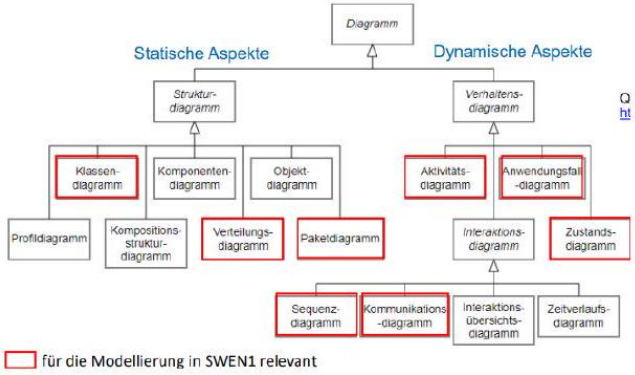
Unterstützungsprozesse

- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

Modelle und Diagramme

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

- 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

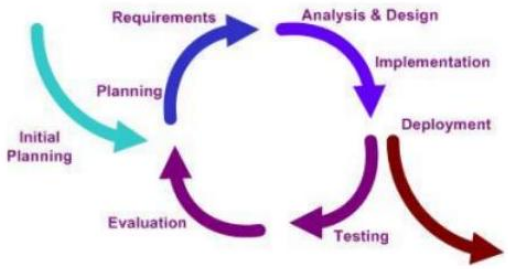
Modell von Requirements (close to/ far from Agreement) & Technology (known / unknown)
Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhanden Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).

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



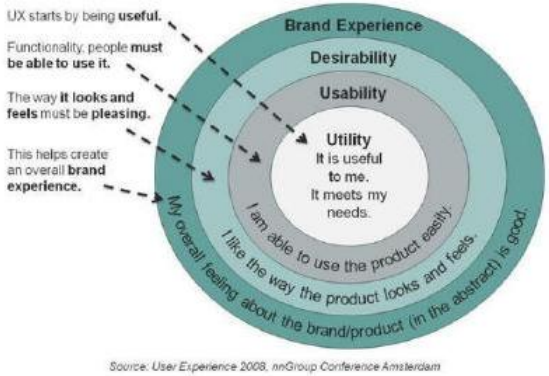
Anforderungsanalyse

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

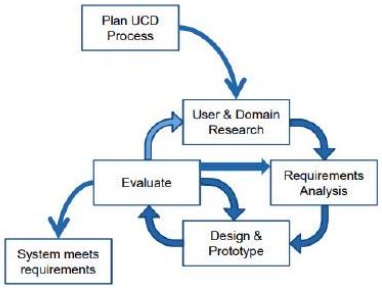
ISO 9241-110: Usability-Anforderungen

Die sieben Grundprinzipien:

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

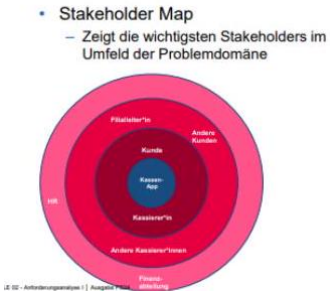
User-Centered Design (UCD)

Ein iterativer Prozess zur nutzerzentrierten Entwicklung:



Wichtige Artefakte

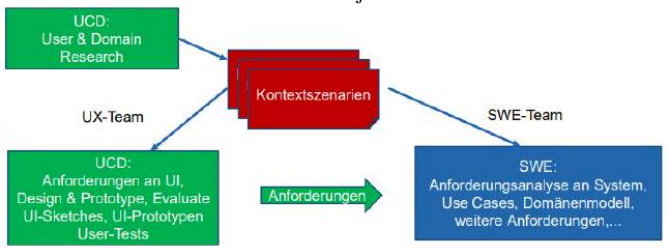
- Personas: Repräsentative Nutzerprofile
- Usage-Szenarien: Konkrete Anwendungsfälle
- Mentales Modell: Nutzerverständnis
- Domänenmodell: Fachliches Verständnis
- Service Blueprint: Geschäftsprozessmodell
- Stakeholder Map: Beteiligte und Betroffene
- UI-Artefakte: Skizzen, Wireframes, Designs



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.

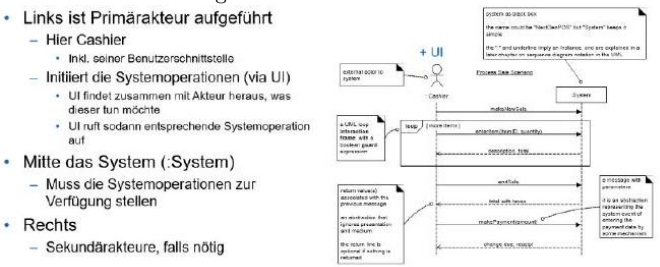
Fully-dressed Use Case **UC: Verkauf abwickeln**

- **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

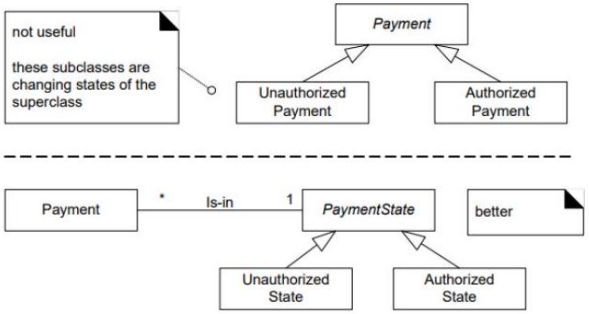
- 100% Regel: Alle Instanzen der Spezialisierung sind auch Instanzen der Generalisierung
- İS-A"Beziehung
- Gemeinsame Attribute/Assoziationen als Grund für Generalisierung

3. Komposition

- Starke Teil-Ganzes Beziehung
- Existenzabhängigkeit der Teile

4. Zustandsmodellierung

- Zustände als eigene Hierarchie
- Vermeidet problematische Vererbung

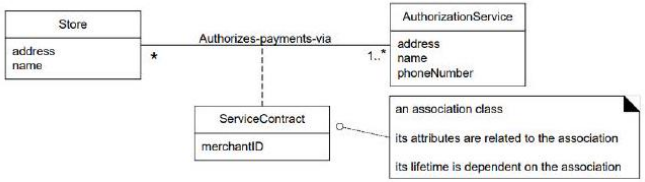


5. Rollen

- 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

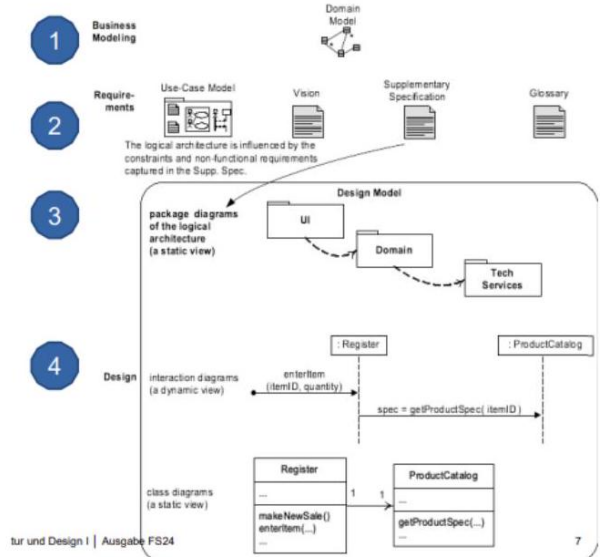
- **Keine Softwareklassen modellieren**
  - Manager-Klassen vermeiden
  - Keine technischen Helper-Klassen
- **Keine Operationen modellieren**
  - Fokus auf Struktur, nicht Verhalten
  - Keine CRUD-Operationen
- **Richtige Abstraktionsebene wählen**
  - Nicht zu detailliert
  - Nicht zu abstrakt
  -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

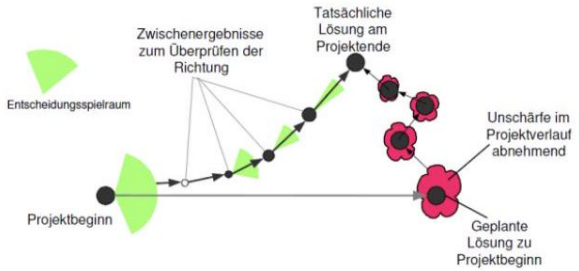
Softwarearchitektur und Design

Überblick Softwareentwicklung

Die Entwicklung von Software erfolgt in verschiedenen Ebenen:

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





Softwarearchitektur

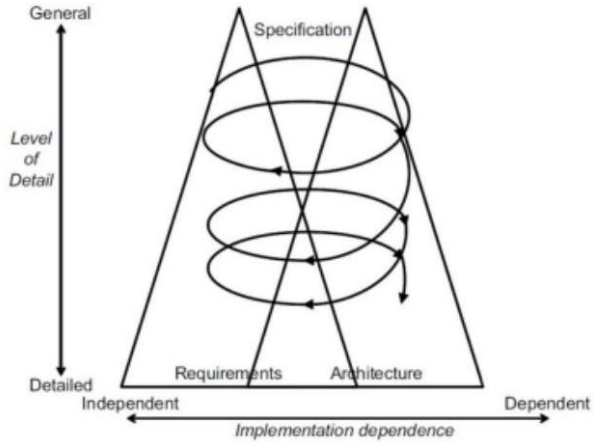
Die Architektur definiert:

- 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:

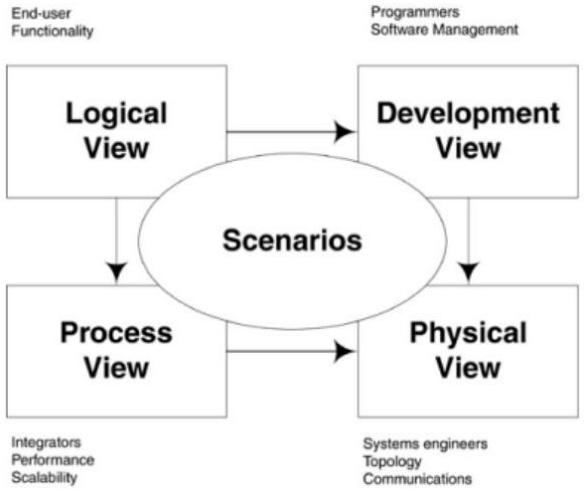
- **Kohäsion:** Innerer Zusammenhang
- **Kopplung:** Externe Abhängigkeiten

Eigenschaften:

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

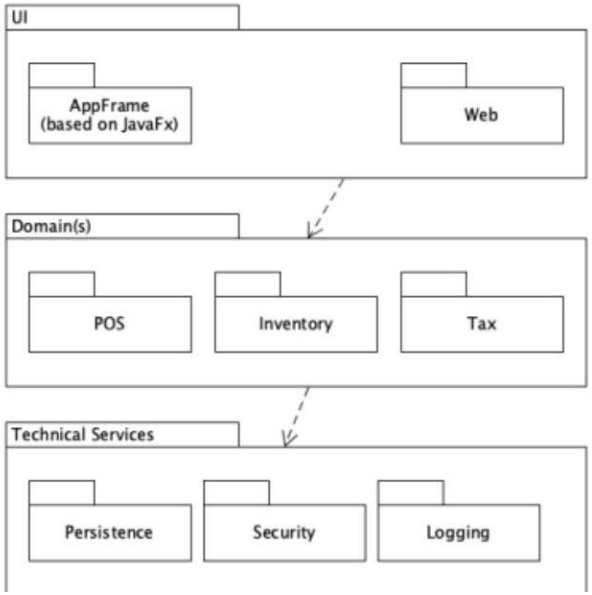
Architektursichten

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

11 Vorlesung 11

11.1 Verteiltes System Definition + Einsatz

Ein verteiltes System besteht aus einer Sammlung autonomer Computer (Knoten) und Softwarebausteinen (Komponenten), die über ein Netzwerk miteinander verbunden sind und gemeinsam als ein einziges System arbeiten. Sie werden in verschiedenen Bereichen eingesetzt, darunter Datenbanken, CloudComputing, verteilte Anwendungen usw.

- Oft sehr gross
- Sehr datenorientiert: Datenbanken im Zentrum der Anwendung
- Extrem interaktiv: GUI, aber auch Batch
- Sehr nebenläufig: Grosse Anzahl an parallel arbeitenden Benutzern
- Oft hohe Konsistenzanforderungen

11.2 Verteiltes System Konzepte + Architekturstil

Verteilte Systeme basieren auf verschiedenen Konzepten und Architekturstilen:

11.2.1 Kommunikationsverfahren

Kommunikationsverfahren umfassen Methoden, mit denen die einzelnen Knoten in einem verteilten System miteinander kommunizieren können. Dazu gehören beispielsweise Remote Procedure Calls (RPC), Message Queuing und Publish-Subscribe-Systeme.

11.2.2 Fehlertoleranz

Fehlertoleranz ist ein wichtiger Aspekt verteilter Systeme, der sicherstellt, dass das System auch bei Ausfällen oder Fehlern in einzelnen Komponenten weiterhin zuverlässig arbeitet. Hierzu werden Mechanismen wie Replikation, Failover und Fehlererkennung eingesetzt.

11.2.3 Fehlersemantik

Die Fehlersemantik beschreibt das Verhalten eines verteilten Systems im Falle von Fehlern oder Ausfällen. Dies umfasst Aspekte wie Konsistenzgarantien, Recovery-Verfahren und Kompensationsmechanismen.

11.3 Design- und Implementierungsaspekte von Client-ServerSystemen

Client-Server-Systeme sind eine häufige Architektur für verteilte Systeme, bei der Clients Anfragen an einen zentralen Server senden, der diese verarbeitet und entsprechende Antworten zurückgibt. Design- und Implementierungsaspekte umfassen unter anderem die Aufteilung von Funktionalitäten zwischen Client und Server, die Wahl der Kommunikationsprotokolle und die Skalierbarkeit des Systems.

11.4 Verteiltes System Architektur + Design Patterns

Die Architektur verteilter Systeme kann durch verschiedene Design Patterns strukturiert werden, um wiederkehrende Probleme effizient zu lösen. Dazu gehören Patterns wie Master-Slave, Peer-to-Peer, Publish-Subscribe, sowie verschiedene Replikations- und Verteilungsstrategien.

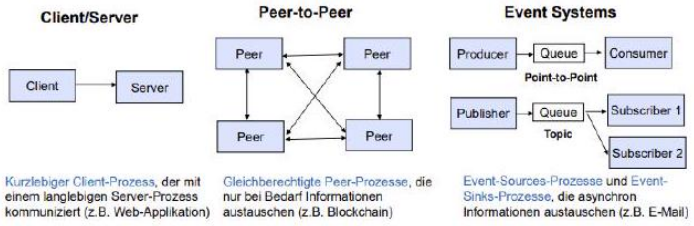


Abbildung 20: Architekturmodelle

11.5 Gängige Technologien (Middleware) f. Informationssysteme und Internet-basierte Systeme

Für die Entwicklung verteilter Systeme stehen verschiedene Middleware-Technologien zur Verfügung, die die Kommunikation und Integration von verteilten Komponenten erleichtern. Dazu gehören Messaging-Broker wie Apache
Kafka, Middleware-Frameworks wie CORBA (Common Object Request Broker Architecture) und RESTful Web Services.

Persistenz

Um was geht es?

- Wie kann ich meine Java Objekte dauerhaft speichern?

Java

- Welche Arten von Datenspeicherung gibt es?
- Welche Design Patterns stehen für die Realisierung von Persistenz in einer Applikation zur Verfügung?
- Wie kann ich mit Hilfe von den Java APIs JDBC (Java Database Connectivity) und JPA (Java Persistence API) meine Objekte dauerhaft in einer Datenbank speichern?

Applikation

```
DB API
```

Treiber

Lernziele LE 12 - Persistenz

- Sie sind in der Lage
- die Varianten der Datenspeicherung zu nennen,
- die unterschiedlichen Design Patterns für die Persistenz zu erklären,
- mit Hilfe des Design Patterns DAO (Data Access Object) und JDBC eine Persistenz in Java umzusetzen,
- mit JPA ein Objekt-Relationales-Mapping (O/R-Mapping) in Java anzuwenden.

1. Einführung in Persistenz

2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. O/R-Mapping mit JPA
6. Wrap-up und Ausblick

Problemstellung Persistenz (1/2)

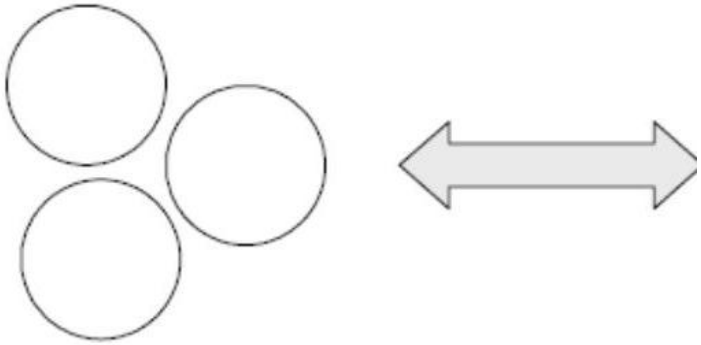
- In vielen Applikationen müssen unterschiedliche Daten verarbeitet, verwaltet und dauerhaft, d.h. über das Programmende hinaus gesichert werden.
- Letzteres bezeichnet man als Persistenz.
- Die dauerhafte Speicherung erfolgt in Datenbankmanagementsystemen (DBMS).
- Übliche Datenbanksysteme sind sogenannte Relationale Datenbanksysteme (RDBMS) und sogenannte NoSQL-Datenbanken.

- NoSQL-Datenbanken speichern Daten ohne fixes Schema und in verschiedenen Formaten (Dokument Stores, Key-Value Stores, Graph DB, ...).

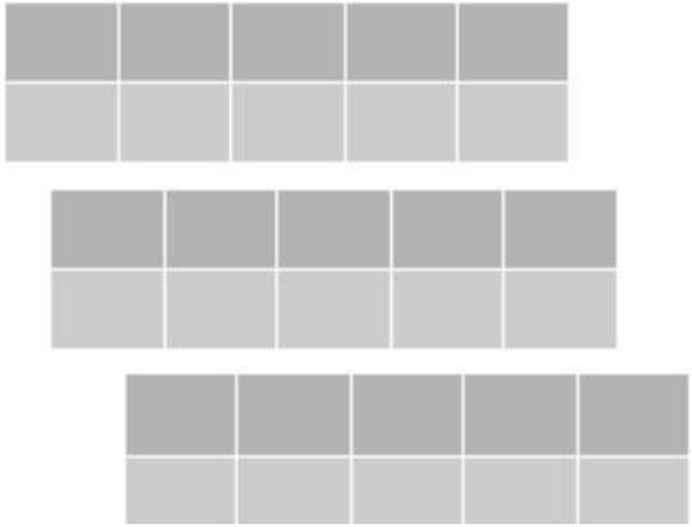
Problemstellung Persistenz (2/2)

- Die Abbildung zwischen Objekten und Datensätzen in Tabellen einer relationalen Datenbank wird auch als O/R-Mapping (Object Relational Mapping, ORM) bezeichnet.
- Verhältnismässig viel Java-Code wird benötigt, um die Datensätze des Ergebnisses zu verarbeiten und in Java-Objekte zu transformieren.
- Es besteht ein Strukturbruch (engl. Impedance Mismatch) aufgrund der unterschiedlichen Repräsentationsformen von Daten (flache Tabellenstruktur -Java-Objekte).

Objektorientiertes Programm



Relationale Datenbank



Denkpause

Aufgabe 12.1 (5')

Diskutieren Sie in Murmelgruppen folgende Frage:

- Was ist aktuell die vorherrschende Technologie zum Speichern von Daten im Enterprise-Umfeld?
- Recherchieren Sie dazu unter https://db-engines.com/en/ranking.
- Was sind die Gründe für dieses Ranking?

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapper mit DAO
5. O/R-Mapper mit JPA
6. Wrap-up und Ausblick

Herausforderung: Der O/R-Mismatch (1/2)

- Der O/R-Mismatch ist ein Fakt.
- Der O/R-Mismatch folgt aus konzeptionellen Unterschieden der zugrundeliegenden Technologien.
- Es gibt viele verschiedene Möglichkeiten (Patterns) den O/R-Mismatch zu überwinden.
- Active Record, O/R-Mapping resp. O/R-Mapping Frameworks oder Repositories (aus Domain Driven Design, DDD) sind ein möglicher Lösungsansatz.

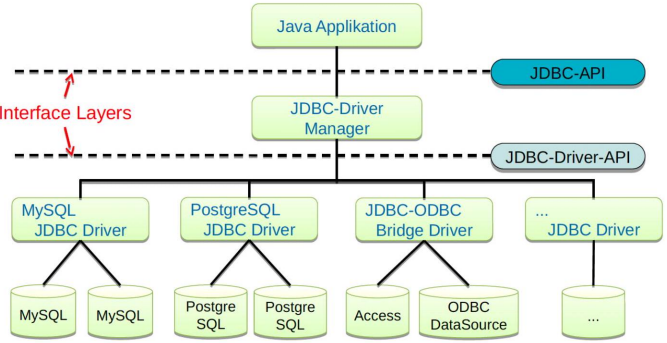
Herausforderung: Der O/R-Mismatch (2/2)

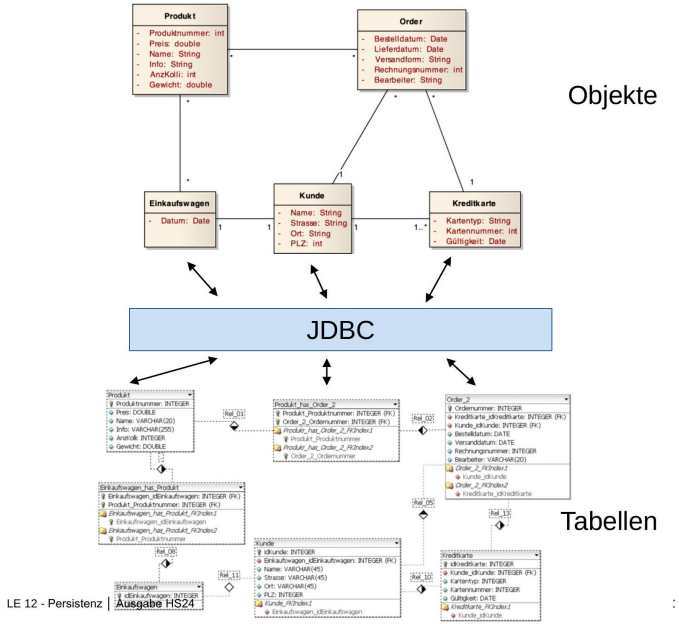
- Typen-Systeme
- Null
- Datum/Zeit
- Abbildung von Beziehungen
- Richtung
- Mehrfachbeziehungen
- Vererbung
- Identität
- Objekte haben eine implizite Identität
- Relationen haben eine explizite Identität (Primary Key)
- Transaktionen

JDBC: Java Database Connectivity (1997)

School of Engineering

- JDBC verbindet die Objektwelt mit der relationalen Datenbankwelt
- Herausforderung: Objekte vs. Tabellen,
- Verschiedene Datentypen etc.
- Die Programmierung ist aufwändig

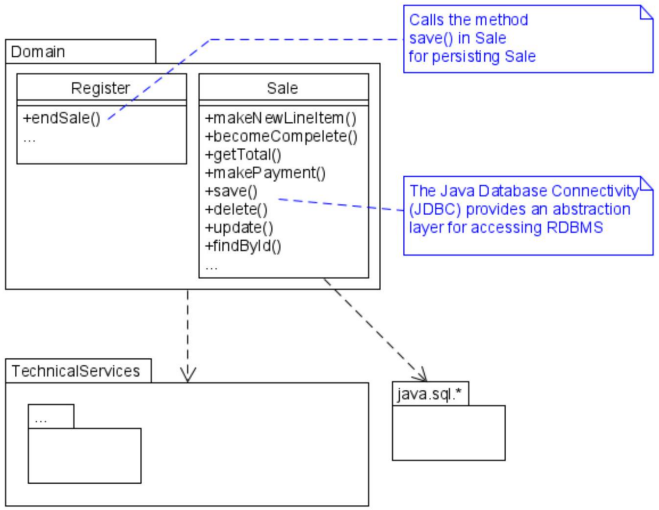




Design Pattern für Persistenz

Für eine Persistenz-Strategie muss eine Entscheidung getroffen werden, wo die Zuordnung (Mapping) zwischen Objekten und Tabellen stattfinden soll:

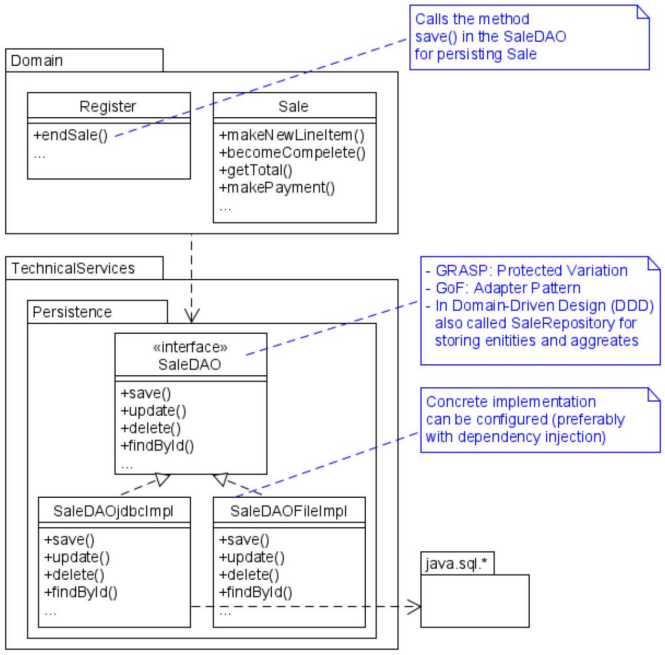
- Active Record (Anti Pattern): Jede Entität ist selber dafür zuständig
- Data Access Object (DAO): Abstrahiert und kapselt den Zugriff auf die Datenquelle
- O/R Data Mapper: Separate Klasse für das Mapping oder Einsatz eines ORM
- Zugriffscode auf Datenbank ist in der Domänenklasse
- Wrapper für eine Zeile einer Datenbanktabelle
- Spiegelt die Datenbankstruktur
- Enthält Daten und Verhalten
- Fachlichkeit und Technik alles in einer Klasse (GRASP: Information Expert?)
- Schlechte Testbarkeit der Domänenlogik ohne Datenbank



- Schlechte Wartbarkeit und Erweiterbarkeit (No separation of concerns!)

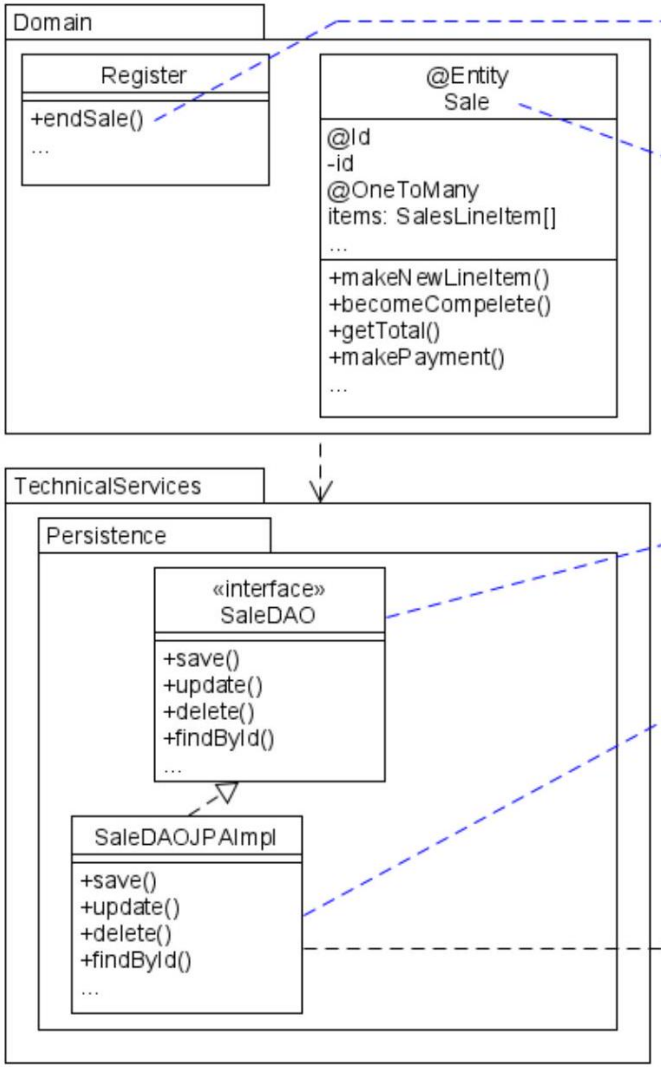
O/R-Mapping von Hand mit Hilfe eines Data Access Objects (DAO)

- Trennung von Fachlichkeit und Technik (Domänenklasse hat hohe Kohäsion)
- Gute Testbarkeit und Mocking der Persistenz
- Bevorzugtes Design ohne Einsatz eines O/RMappers



Verwendung eines O/R-Mappers (JPA mit Hibernate/Eclipselink o.ä. Framework)

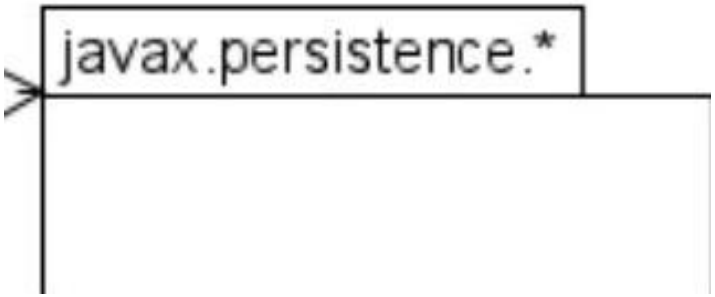
- Viel weniger Aufwand bzw. Code und standardisierte Schnittstelle
- Trennung von Fachlichkeit und Technik (Domänenklasse hat hohe Kohäsion)
- Gute Testbarkeit und Mocking der Persistenz
- DAO ist auch beim Einsatz von JPA empfehlenswert (Trennung von Fachlichkeit und Technik) - JPA könnte aber auch ohne DAO verwendet werden



Calls the method save() in the SaleDAO for persisting Sale

`Same design pattems apply as for`

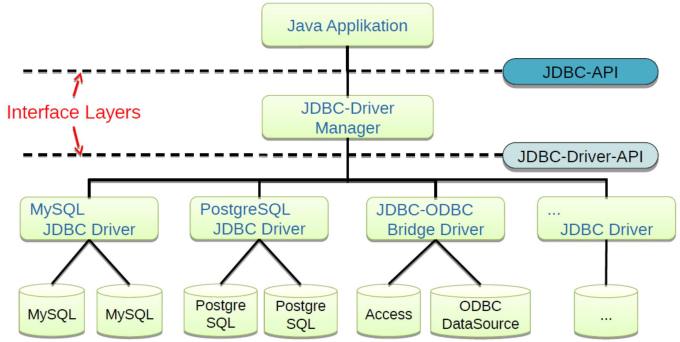
the option with handcrafted DAOs.
For implementing the DAO
the Java Persistence API the Java Persistence API (JPA) and a service provider (e.g. Hibernate) provide an orR mapping and the basic by using the annotations and a configuration file (persistence xml)For queries a Java Persistence Query Language (JPQL) is available.



1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapper mit DAO
5. O/R-Mapper mit JPA
6. Wrap-up und Ausblick

Was genau ist JDBC?

- JDBC = Java Data Base Connectivity
- Standardisierte Schnittstelle, um auf relationale Datenbanken mit Hilfe von SQL zuzugreifen
- Cross-Plattform und DB-independent
- JDBC-API ist Teil der Java-Plattfrom seit JDK1.1 (1997)
- Die aktuelle Version ist 4.2



JDBC-API: Interfaces and Classes

Anwendung von JDBC

Basisanweisungen:

1. Install and load JDBC driver
2. Connect to SQL database
3. Execute SQL statements
4. Process query results
5. Commit or Rollback DB updates
6. Close Connection to database

```
import java.sql.*;
public class DbTest {
    public static void main(String[] args)
        throws ClassNotFoundException, SQLException {
        Connection con = DriverManager.getConnection(
            "jdbc:postgresql://test.zhaw.ch/testdb",
            "user", "password");
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery(
            "SELECT * FROM test ORDER BY name");
        while (rs.next()) {
            System.out.println(
                "Column 1 contains '" +
                rs.getString(2) +"'");
        }
        con.close();
    }
}
```

1. Einführung in Persistenz 2. Design-Optionen für Persistenz 3. Persistenz mit JDBC 4. O/R-Mapping mit DAO 5. O/R-Mapping mit JPA 6. Wrap-up und Ausblick

O/R-Mapping Pattern

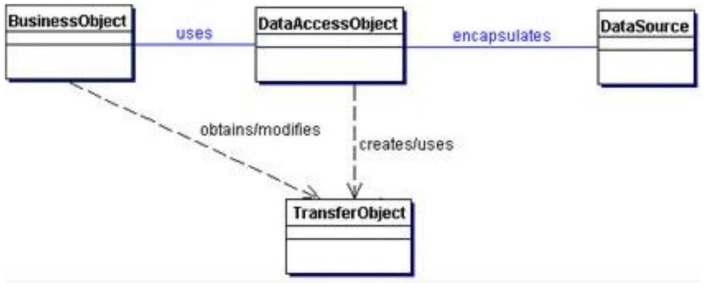
Es sollen beide Varianten des O/R-Mapper Patterns anhand eines praktischen Beispiels betrachtet werden: - DAO (Data Acess Object) ohne ein ORM (Object Relational Mapper) - Umsetzung von DAO mit Hilfe von JPA (Java Persistence API)

DAO - Data Access Object Pattern

- Das Artikel-Objekt repräsentiert das Domain-Model-Objekt. - Die Verbindung zur Datenbank wird durch das DAO sichergestellt. - Enthält die üblichen CRUD-Methoden wie create, read, update und delete. - Kann auch Methoden enthalten wie findAll, findByName, findByID um eine Kollektion von Daten aus der Datenbank abzufragen.

DAO - Data Access Object Pattern

School of



Sun Developer Network - Core J2EE Patterns
http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Das TransferObject aka. DTO kann zusätzlich für den Transport der Daten in einem verteilten System verwendet werden.

Beispiel Article und ArticleDAO

School of Engineering

Business Object

```
public class Article {
private long id;
private String name;
private float price;
public long getId(){
return id;
}
public void setId(long id){
this id = id
};
...
}


\section*{Data Access Object (DAO)}

//Interface to be implemented by all ArticleDAOs
public interface ArticleDAO {
public void insert(Article item);
public void update(Article item);
public void delete(Article item);
public Article findById(int id);
public Collection<Article> findAll();
public Collection<Article> findByName (String name);
public Collection<Article> findByPrice (float price);
}

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. O/R-Mapping mit JPA
6. Wrap-up und Ausblick
```

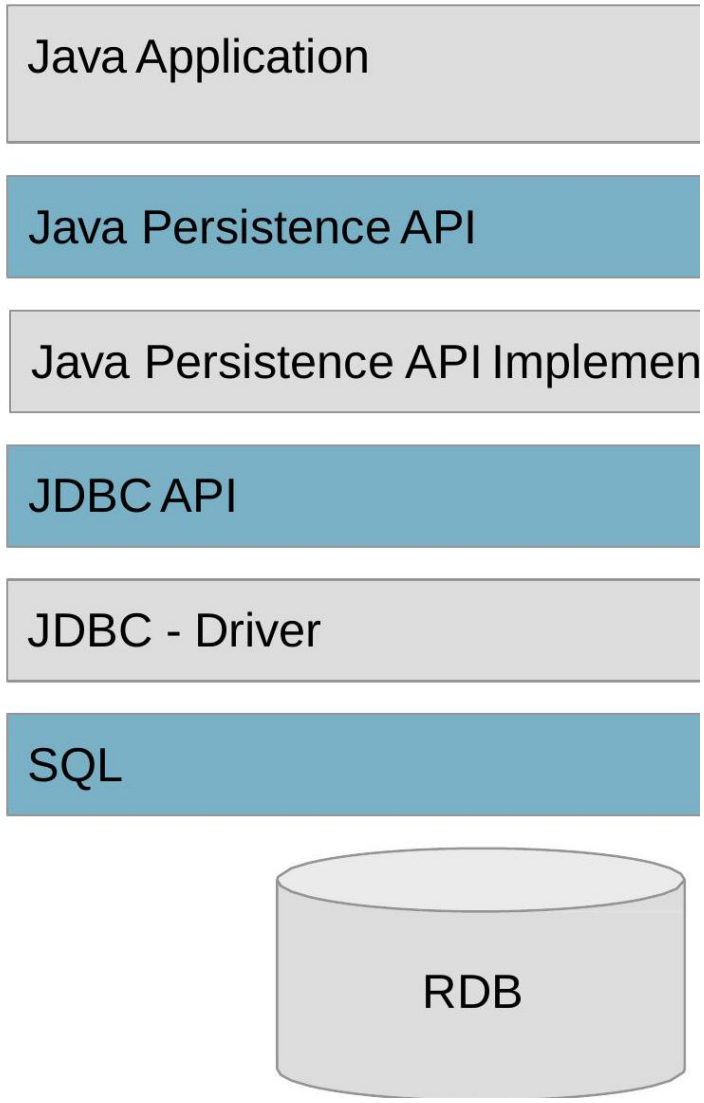
Versprechen von automatischem O/R-Mapping

- Die Applikation wird von der DB entkoppelt - Applikationsentwickler muss kein SQL beherrschen. - Das relationale Modell der Datenbank hat keinen Einfluss auf das OO-Design. - Automatische Persistenz - Automatisierte Abbildung der Objekte in die relationalen Strukturen. - Die Applikationsentwickler muss sich nicht um die «low-level»-Details kümmern. - Transparente Persistenz / Persistence Ignorance - Die Klassen des Domain-Models wissen nicht, dass sie persistiert und geladen werden können und haben keine Abhängigkeit zur Persistenz-Infrastruktur. - JPA ist ein Java Standard für O/R-Mapping - Verschiedene Implementationen, Hibernate vermutlich die bekannteste

JPA (Java Persistence API) Überblick

- Es folgt eine kurze, unvollständige Auflistung der wichtigsten Konzepte von JPA. - Starke Entkopplung der Anwendungslogik von der (relationalen) Datenbank. - Die Domänenklassen sind ganz normale Java Klassen (POJO) - Ausser Annotationen enthalten Sie keinen JPA spezifischen Code. - Referenzen - Werden entweder mit der referenzierenden Klasse (eager loading) oder erst, wenn die Referenz gebraucht wird (lazy loading), geladen. - Referenzen können direkt traversiert werden, JPA erledigt das Laden des referenzierten Objekts im Hintergrund. - Transaktionshandling und das Absetzen von Queries müssen über JPA spezifische Klassen abgewickelt werden. - EntityManagerFactory, EntityManager, EntityTransaction

Technologie-Stack



Java 5+
JPA Spezifikation EclipseLink (TopLink), Hibernate, OpenJPA etc.
JDBC 4.0 Herstellerspezifisch SQL (und Dialekte)

Entity Metadata

- Kennzeichnung mit Annotation @Entity oder Mapping mit XML - Klasse kann Basisklasse oder abgeleitet sein - Klasse kann abstrakt oder konkret sein - Serialisierbarkeit ist bezüglich Persistenz nicht erforderlich

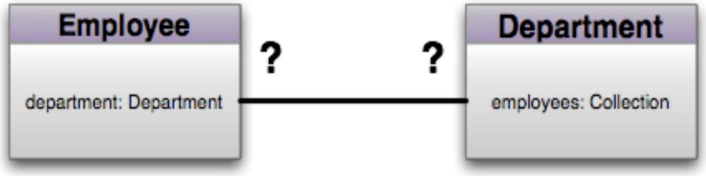
Beispiel Entity

School of - Minimale Anforderung an eine Entity
@Entity
public class Employee {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;
private String name;
private String lastName; }

- Primärschlüssel können in Zusammenarbeit mit der Datenbank gene

@Entity public class Employee {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public Integer id;
}
public class Employee {
@TableGenerator(name = Ëmp_Gen", table = ÏD_GEN", pkColumnName = "GEN_NAME",
valueColumnName = "GEN_VAL")
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = Ëmp_Gen")
private int id; }

Parent-Child Beziehung



- Mapping des Klassenmodells auf das DB-Schema mittels JPA: Metadata ist erforderlich. - Je nach Klassenmodell wird entweder eine many-to-one Beziehung oder eine one-to-many Beziehung gemappt. - Falls beide Richtungen gemappt werden sollen, so muss definiert werden, dass für beide derselbe Foreign-Key zugrunde liegt.

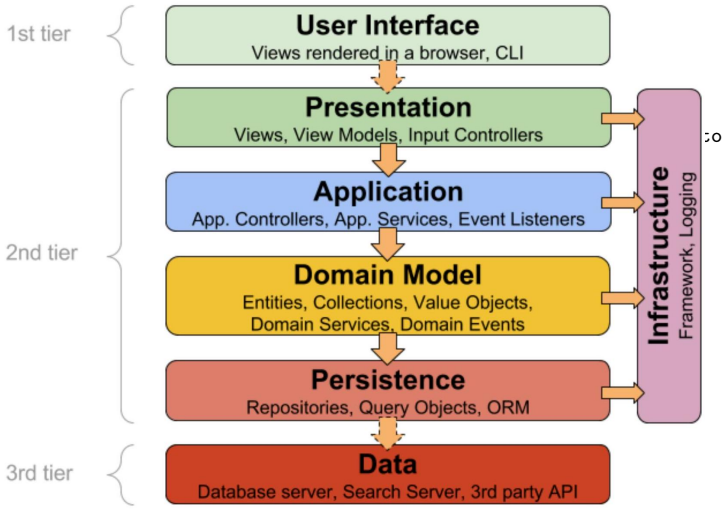
Employee
id
name
salary
department id

Ausblick Design Pattern Repository

- Ein System mit einem komplexen Domänen-Model profitiert wie vorher beschrieben von einer Data-Mapper-Schicht (mit JPA und DAOs), um die Details des Datenbankzugriffcodes zu isolieren. - Eine zusätzliche Abstraktionsschicht oberhalb des Data-Mappers kann helfen um die Konstruktion von Datenbank-Abfragen (Queries) an einem Ort zu konzentrieren. - Diese zusätzliche Schicht wird um so wichtiger je mehr Domänen-Klassen vorhanden sind, die viele Zugriffe auf die Datenbank vornehmen. - Die zusätzliche Schicht wird als Repository bezeichnet - Das Konzept stammt aus Domain Driven Design, DDD (Eric Evans). - Wird in den Wahlpflichtmodulen ASE1/2 anhand von Spring Data behandelt.

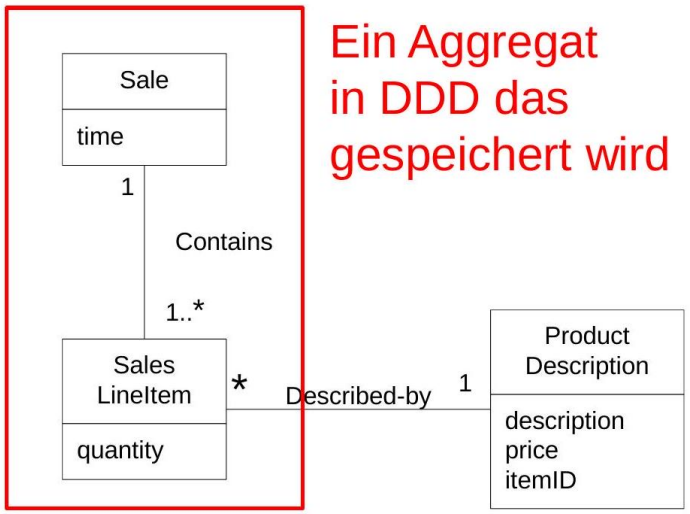
Design Pattern Repository: Schichtenmodell

- 3-Tier Architecture - Persistenz kann mittels Repositories umgesetzt werden



Idee und Beispiel Repository Pattern

School of Engineering InIT Institut für angewandte Informationstechnologie - Eine Repository vermittelt zwischen Domänen- und Data-Mapping Schicht
9 0/** * The GRASP controller for the use case process sale.
11 */
\emptysetpublic class ProcessSaleHandler {
} private ProductDescriptionRepository catalog;
private SaleRepository saleRepository;
private Sale currentSale;
public ProcessSaleHandler(ProductDescriptionRepository catalog, SaleRepository saleRepository) {
} public void makeNewSale() {
} public void enterItem(String id, int quantity) {
} public Money getTotalOfSale() {
} @Transactional
public void endSale() {
assert(currentSale != null && !currentSale.isComplete());
this.currentSale.becomeComplete()
this.saleRepository.save(currentSale);
}
public Money getTotalWithTaxesOfSale() {
} public void makePayment() {




```
/**
* Repository for Sale
* An implementation of CRUD and common search methods
* is automatically generated by Spring Data.
*/
6 @Repository
\emptysetpublic interface SaleRepository extends CrudRepository<Sale, String>
public List<Sale> findOrderByDateTime();
public List<Sale> findByDateTime(final LocalDateTime dateTime);
L}
```

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. O/R-Mapping mit JPA
6. Wrap-up und Ausblick

- Viele Applikationen verlangen, dass Daten dauerhaft gesichert werden müssen nach dem Programmende.
- Bei kleineren Applikationen kann diese Persistenz auch selber ausprogrammiert werden.
- Dabei sollte aber das Design Pattern Data Access Object (DAO) oder Repository angewendet werden.
- Für grössere Applikationen werden heute sogenannte O/R-Mapper eingesetzt.
- Java bietet mit dem Java Persistence API (JPA) eine standardisierte Schnittstelle für das O/R-Mapping, für die es viele Provider gibt (z.B. Hibernate).

Framework Design

Um was geht es?

- Ein Framework ist ein Programmiergerüst, das dem Anwendungsprogramm einen Rahmen gibt und wiederverwendbare Funktionalität zur Verfügung stellt.
- Es bietet gezielt Orte an, wo es erweitert oder angepasst werden kann.
- In Frameworks kommen gewisse Design Patterns zum Einsatz.
- Frameworks werden heutzutage sehr häufig eingesetzt.

Lernziele LE 13 - Framework Design

- Sie sind in der Lage:
- die Eigenschaften von Frameworks zu nennen,
- Design Patterns im Einsatz von Frameworks anzuwenden,
- Prinzipen von modernen Frameworks zu verstehen,
- die Auswahl und den Gebrauch von Frameworks kritisch einzuschätzen.

1. Einleitung und Definition

2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

Framework Charakterisierung

- Leider gibt es keine allgemein akzeptierte exakte Definition eines Frameworks und der Begriff wird für viele Programmbibliotheken eingesetzt.
- Für unsere Zwecke möchten wir den Begriff folgendermassen abgrenzen:
- Ein Framework enthält keinen applikationsspezifischen Code.
- Ein Framework gibt aber den Rahmen («Frame») des anwendungsspezifischen Codes vor.
- Die Klassen eines Frameworks arbeiten eng zusammen, dies im Gegensatz zu einer reinen Klassenbibliothek wie z.B. die Java Collection Klassen.
- Ein Framework muss für den Einsatz gezielt erweitert und/oder angepasst werden.
- Applikations-Container wie z.B. Spring Framework oder Java EE (neu Jakarta EE) schliessen wir ebenfalls ein.
- Die Entwicklung eines neuen Frameworks ist eine aufwändige Angelegenheit.
- Wiederverwendbare Software (und dazu gehören natürlich Frameworks) sollte ein höheres Level im Bereich Zuverlässigkeit besitzen, was ebenfalls mit zusätzlichem Aufwand verbundenen ist.
- Erweiterbare Software (und dazu gehören natürlich Frameworks) erfordert eine tiefergehende Analyse darüber, welche Teile erweiterbar sein sollen, was zu einem höheren Architektur- und Designaufwand führt.
- Eigentlich sprechen alle diese Punkte dagegen, selber ein Framework zu entwickeln. Weshalb wird dies trotzdem behandelt?

Framework Einsatz und Entwicklung erweiterbarer Software

- Alle Design Patterns, die wir heute behandeln, können für die Entwicklung erweiterbarer Software eingesetzt werden.
- Dies muss nicht zwingend ein Framework sein, das auf GitHub publiziert wird, sondern es kann auch einfach eine Komponente sein, die in mehreren eigenen Anwendungen in verschiedenen Kontexten eingesetzt wird.
- Das Wissen um den Aufbau von Frameworks hilft auch, deren Einsatz, aber auch deren Grenzen zu verstehen.

Kritische Bemerkungen zu Frameworks

- Frameworks tendieren dazu, im Laufe der Zeit immer mehr Funktionalität zu «sammeln».
- Was auf den ersten Blick positiv scheint, kann im zweiten Blick zu inkonsistentem Design und funktionalen Überschneidungen führen, die den Einsatz immer mehr erschweren.
- Der Einsatz eines Frameworks sollte gut überlegt werden.
- Einerseits erfordert dies gute Kenntnisse des Frameworks, andererseits ist nach der «Verheiratung» der Anwendung mit dem Framework eine «Scheidung» nur noch schwierig und mit hohem Aufwand möglich.
- Allenfalls sollte das Framework nur über eigene Schnittstellen verwendet werden (keine direkte Abhängigkeit), was aber unter Umständen die Nützlichkeit des Einsatzes in Frage stellt.

1. Einleitung und Definition
2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

Recap: Aufbau Design Patterns

- Beschreibungsschema
- Name
- Beschreibung Problem
- Beschreibung Lösung
- Hinweise für Anwendung
- Beispiele

Recap: Anwendung von Design Patterns

- Design Patterns sind ein wertvolles Werkzeug, um bewährte Lösungen für wiederkehrende Probleme schnell zu finden.
- Sie helfen, im Team effizient über Lösungsmöglichkeiten zu kommunizieren.
- Ihre Anwendung stellt aber immer einen Trade-Off zwischen Flexibilität und Komplexität dar.
- Es ist keineswegs so, dass ein Programm automatisch besser wird, wenn mehr Patterns angewendet werden.

Design Patterns

- Abstract Factory
- Factory Method
- Command
- Template Method

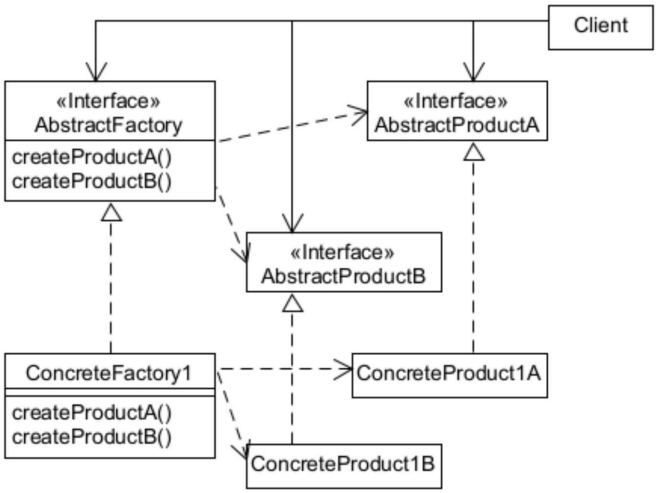
Abstract Factory: Problem und Lösung

- Problem

- Die Erzeugung verschiedener, inhaltlich zusammengehörender Objekte («Product»), ohne aber die konkreten Klassen zu kennen, damit diese austauschbar sind.

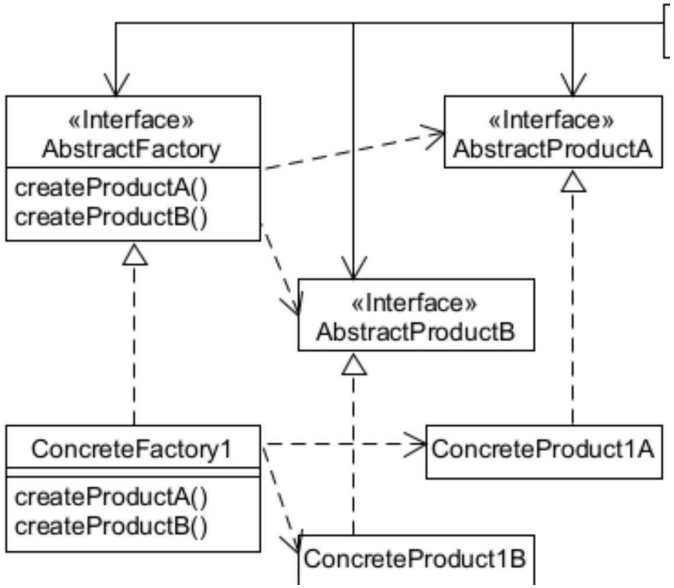
- Lösung

- Eine AbstractFactory und abstrakte «Products» definieren.
- Die AbstractFactory hat für jedes «Product» eine eigene «create» Methode.
- Eine konkrete Factory davon ableiten, die dann konkrete «Products» erzeugt.



Abstract Factory: Hinweise

- Hinweise
- Eigentlich «nur» eine Verallgemeinerung einer «SimpleFactory».
- Die verschiedenen Produkte hängen inhaltlich miteinander zusammen, zum Beispiel verschiedene Teile der anzusteuernden Hardware.



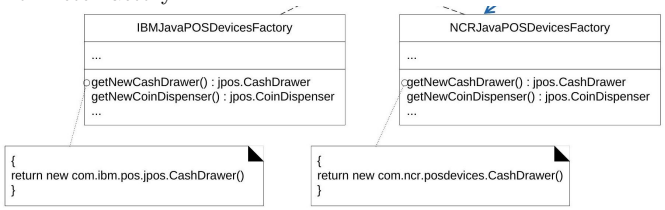
Abstract Factory: Beispiel Point Of Sale (POS) Terminal

School of Engineering

- Die elektronische Kasse muss Hardware wie z.B. die Notenschublade oder den Münzspender ansteuern.
- Typischerweise kommen die einzelnen Komponenten vom selben Hersteller.
- Pro Hersteller gibt es eine konkrete Implementation von IJavaPOSDevicesFactory.

```
this is the Abstract
Factory--an interface fo
creating a family of
related objects
```

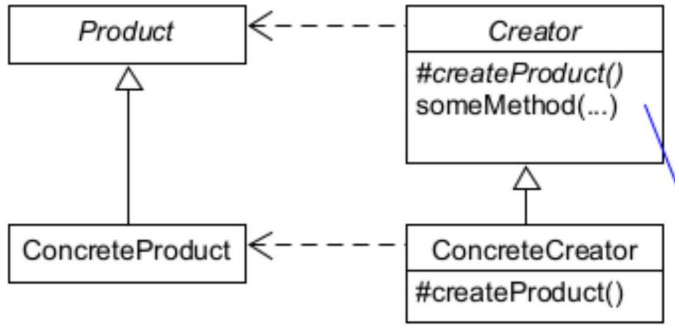
Abstract Factory ↙
Konkrete Factory



com.ncr.posdevices.CashDrawer
isDrawerOpened()

Factory Method: Problem und Lösung

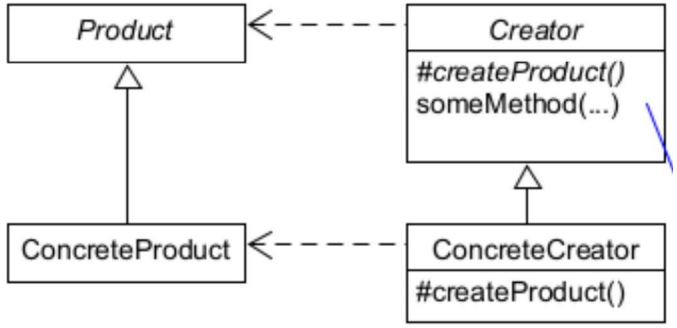
- Problem
- Eine (wiederverwendbare) Klasse Creator hat die Verantwortlichkeit, eine Instanz der Klasse Product zu erzeugen. Es ist aber klar, dass Product noch spezialisiert werden muss.
- Lösung
- Eine abstrakte Methode in der Klasse Creator definieren, die als Resultat Product zurückliefert.
- Konkrete Klassen von Creator können dann die



richtige Subklasse von Product erzeugen.

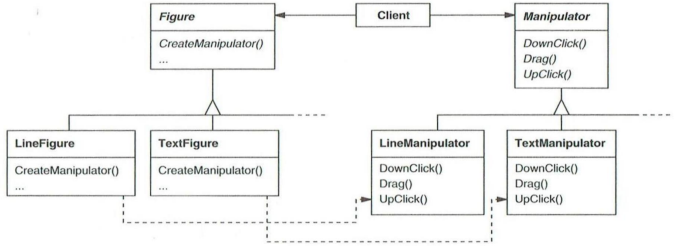
Factory Method: Hinweise

- Hinweise
- Es ist durchaus erlaubt, dass bereits Creator und Produkt konkret sind und somit eine Basisfunktionalität zur Verfügung stellen.
- Es gibt parallele Vererbungshierarchien mit Creator wie auch Product an der Spitze.
- Kann auch als Variante des Design Patterns «Template Method» interpretiert werden.



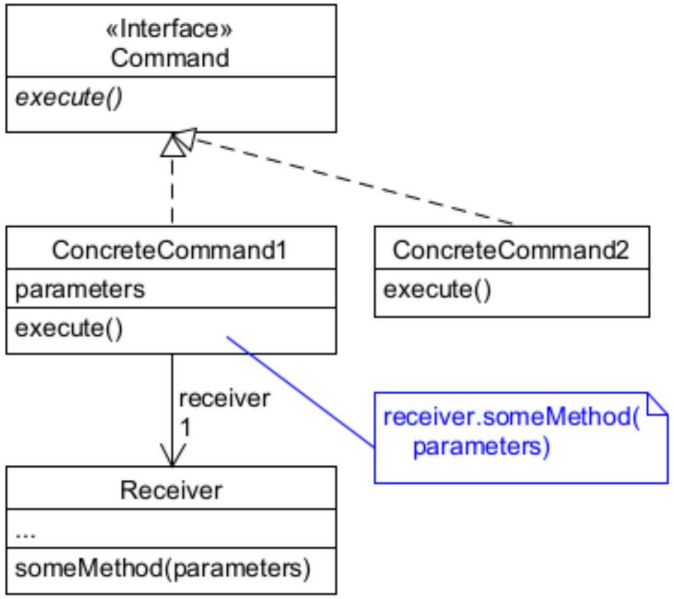
Factory Method: Beispiel GoF (2/2)

- Das Zeichenprogramm («Client») besitzt eine KlassenHierarchie von Figuren.
- Um Figuren übers UI verändern zu können, gibt es eine abstrakte Manipulator Klasse.
- Jede konkrete Figur hat nun die Aufgabe, seine Manipulator Klasse zu instanziieren.



Command: Problem und Lösung

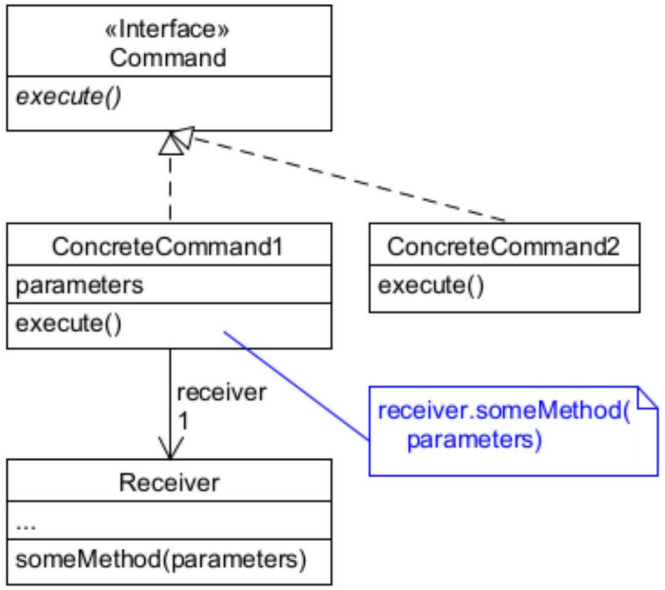
- Problem
- Aktionen müssen für einen späteren Gebrauch gespeichert werden und dabei können sie noch allenfalls priorisiert oder protokolliert werden und/oder Unterstützung für ein Undo anbieten.
- Lösung
- Ein Interface wird definiert, das nur die Auslösung der Aktion erlaubt.
- Implementationen dieses Interface überschreiben die Methode zur Auslösung der Aktion.
- Meistens bedeutet die Aktion, dass eine Methode auf



einem anderen Objekt aufgerufen wird.
- Dazu muss die Aktion die Parameter dieser Methode zwischenspeichern.

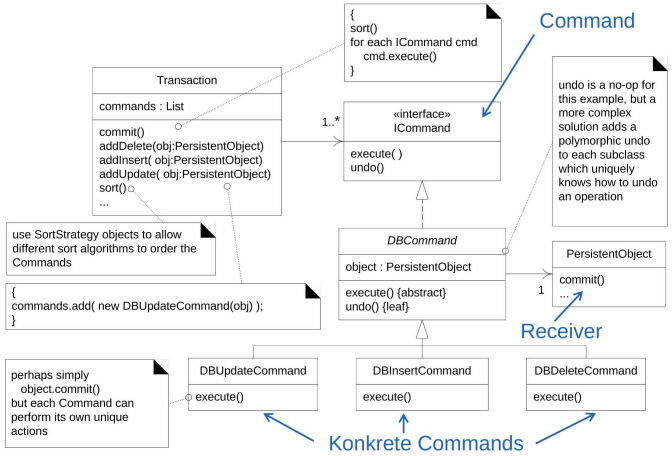
Command: Hinweise

- Hinweise
- Erstellung der Aktion und das Auslösen liegen zeitlich auseinander.
- Bevor Aktionen ausgelöst werden, können sie bei Bedarf noch sortiert oder priorisiert werden. Denken Sie dabei an eine Datenbank.
- Der Receiver muss nicht zwingend über eine Assoziation sichtbar sein. Es ist auch ein Lookup über z.B. einen Namen denkbar.
- Falls eine Rückabwicklung der Aktion («Undo») notwendig ist, kann die entsprechende Methode direkt in der Aktion eingefügt werden oder es gibt eine separate Aktion dafür.



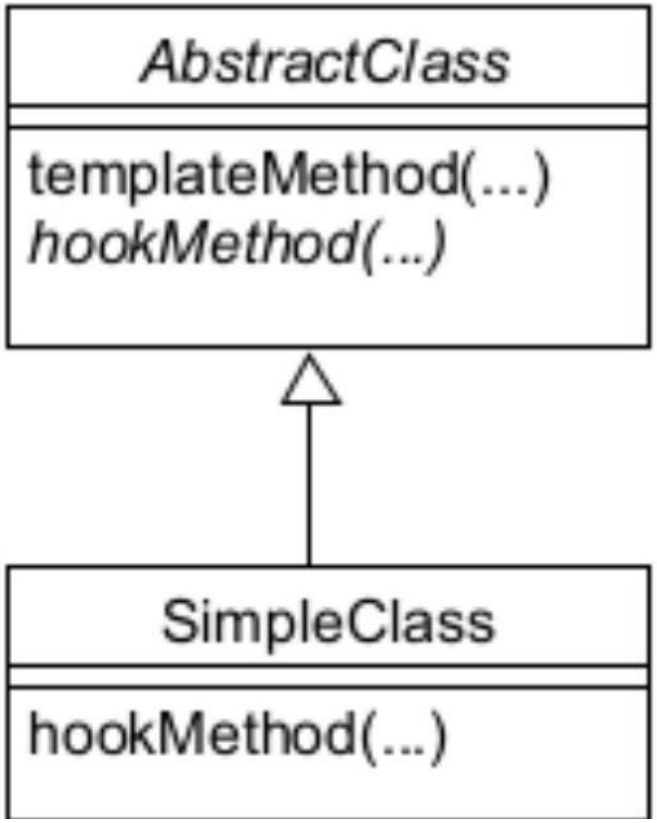
Command: Beispiel Point Of Sale Terminal

- Eine Transaktion eines Persistenz-Frameworks setzt sich aus den Aktionen für jedes veränderte Objekt zusammen.
- Aktionen sind update, insert und delete.
- Eine undo Methode ist ebenfalls vorhanden.



Template Method: Problem und Lösung

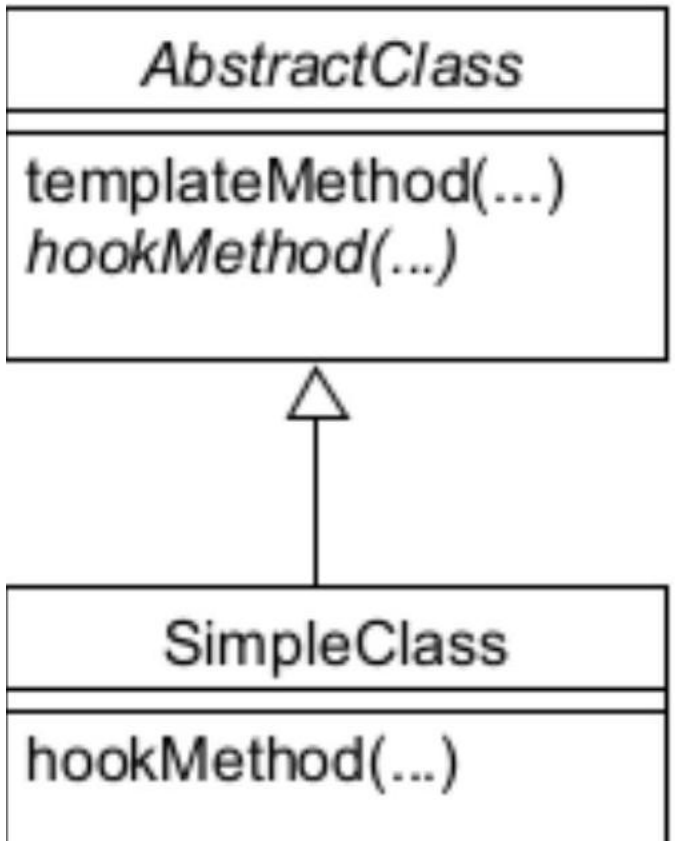
- Problem
- Ein Ablauf/Algorithmus soll so entworfen werden, dass er in gewissen Punkten angepasst werden kann.
- Lösung
- In einer abstrakten Klasse wird eine Template Method hinzugefügt, die diesen Ablauf/Algorithmus implementiert.
- Die Template Method ist fertig geschrieben, ruft aber noch abstrakte Methoden («hookMethod») auf.
- Diese Methoden dienen als Variations- resp. Erweiterungspunkte



und mit ihrer Implementation kann der Ablauf/Algorithmus auf den aktuellen Kontext angepasst werden.

Template Method: Hinweise

- Hinweise
- Die «hookMethod» kann entweder rein abstrakt sein oder bereits eine Standard-Implementation enthalten.
- Eine Factory Method kann in diesem Zusammenhang ebenfalls als «hookMethod» interpretiert werden.
- Es ist nicht einfach, im Voraus alle Orte zu identifizieren, wo Anpassungen notwendig sein müssen.
- Verwandtschaft mit einer Strategy. Eine Strategy benutzt Delegation, um einen ganzen Algorithmus zu variieren, während



Template Method Vererbung benutzt, um einen Teil des Algorithmus zu variieren.

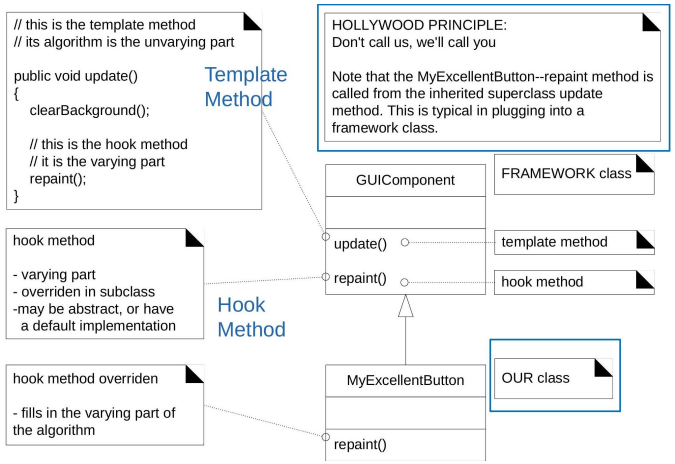
- Hollywood Prinzip: «Don't call us, we call you». Der eigene Code wird von fremdem Code aufgerufen (oder: der Code des Frameworks ruft den Code der Umsetzung auf).

Template Method: Beispiel Larman GUI Framework

School of Engineering
InIT Institut für angewandte Informationstechnologie

- Ein GUI Framework stellt Komponenten zur Verfügung.

- Die Basisklasse GUIComponent stellt die Template Method update() zur Verfügung, die repaint() aufruft.
- Die Methode repaint() muss dann von unserer Klasse überschrieben werden.



1. Einleitung und Definition
2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

Einleitung Persistenz-Framework im Buch von Larman

- Framework für Speicherung von Objekten (siehe [1] Kap. 38).
- Primäres Ziel: Prinzipien des Framework Designs zeigen.
- Sekundäres Ziel: Problemstellungen von Persistenz-Frameworks und mögliche Lösungsansätze zeigen.
- Was fehlt?
- Eigentliche RDB-Zugriffe. Im Buch werden verschiedene Lösungen skizziert.
- Eigentliche Behandlung von Collections und Assoziationen. Im Buch wird dafür die Verwendung vom Design Pattern «Virtual Proxy» erwähnt.
- Abfragen (Queries) werden gar nicht behandelt. Da ja beliebige Speichertechnologien unterstützt werden sollen, ist dies aber auch nicht verwunderlich.
- Der vollständige Programmcode.

Themen Persistenz-Framework von Larman

- Persistenz-Fassade
- Mapping auf RDB
- Mapper für jede Klasse
- Objekt-Identifikation
- Verfeinerung Mapper
- Zustandsverwaltung bezüglich Transaktionen
- Proxy für Lazy Loading von referenzierten Objekten

1. Einleitung und Definition
2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

Moderne Framework Patterns

- Die bewährten Design Patterns finden nach wie vor ihre Anwendung im Framework Design.
- In den letzten Jahren wurden aber noch weitere Mechanismen populär.
- Dependency Injection, meistens gesteuert über Annotationen
- Convention over Configuration: Nur durch das Einhalten von (Namens-)Konventionen wird das Framework aktiv und macht das Gewünschte.
- Implementation von Interfaces basierend auf den Methoden des Interfaces (z.B. Spring Data Repository-Interfaces). Der Methodenname spezifiziert sozusagen seine Implementation, allenfalls noch ergänzt mit Annotationen
- Ist ein Standard Java-Sprachelement ab Java 5 (z.B. @override).
- Können selber deklariert werden.
- Werden «normalen» Sprachelementen hinzugefügt
- Vorteil: Wenn beim Laden einer annotierten Klasse die Annotations-Klasse nicht gefunden wird, gibt es keine Fehlermeldung, sondern die Annotation wird stillschweigend entfernt.
- Anders gesagt fügen Annotationen keine harte Abhängigkeit hinzu und sind somit geeignet, die Domänenlogik frei von ungewünschten (technischen) Abhängigkeiten zu halten.

Steuerung über Annotationen

- Annotationen per se haben ja keine Funktionalität. Es braucht «jemand», der die Annotationen liest und dann Aktionen ausführt.
- Auswertung von Annotationen:
- Beim Starten der Anwendung wird das Framework ebenfalls gestartet.
- Das Framework sucht die Anwendungsklassen auf dem Klassenpfad ab, untersucht allfällige Annotationen und führt die gewünschten Aktionen aus.
- Mögliche Aktionen des Frameworks:
- Dependency Injection von Framework Objekten in Anwendungsobjekte (über Constructor oder Set-Methode).
- Automatisches Implementieren von Interfaces.
- Hinzufügen von Funktionalität zu Anwendungsklassen.
- Achtung: Dieser Vorgang kann zu unerwünschten Verzögerungen beim Start führen.

Java Mechanismen für das Hinzufügen von Funktionalität

- 2 Zeitpunkte
- Während (respektive am Schluss) der Kompilierung über einen AnnotationProcessor.
- Beim Starten einer Anwendung können Anwendungsklassen beim Laden (über einen FrameworkClassloader) noch verändert werden.
- Was wird verändert
- Quellcode hinzufügen.
- Bytecode hinzufügen und bestehenden abändern.
- Für das Implementieren von Interfaces kann java.lang.reflect.Proxy eingesetzt werden.
- Wer verändert
- AnnotationProcessor kann Quellcode und Bytecode hinzufügen.
- Beim Starten einer Anwendung kann Byte Code verändert und hinzugefügt, sowie die Proxy Klasse angewendet werden.

Agenda

1. Einleitung und Definition
2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

- Gerade Frameworks müssen sorgfältig mit bewährten Design Patterns entworfen werden.
- Traditionelle Framework Patterns sind die Template Methode und die Factory Method, die es erlauben, dass in Framework Klassen ein

Algorithmus realisiert wird, der aber in anwendungsspezifischen, abgeleiteten Klassen noch an den aktuellen Kontext angepasst werden kann.
- Das Command Pattern erlaubt es, dass das Framework anwendungsspezifischen Code aufrufen kann, ohne dass das Framework angepasst werden muss.
- AbstractFactory dient dazu, die Erzeugung einer Familie verwandter Objekte zu ermöglichen.
- Larman hat die Grundzüge eines Persistenz-Frameworks in seinem Buch entworfen, das didaktischen Zwecken dient und den Entwurf eines Frameworks an einem umfangreicheren Beispiel demonstriert.
- Moderne Frameworks setzen auf die Steuerung durch Annotationen, vor allem für Dependency Injection.
- In der nächsten Lerneinheit werden wir:
- den ganzen Stoff SWEN1 kurz repetieren und
- eine alte Semesterendprüfung (SEP) gemeinsam lösen.

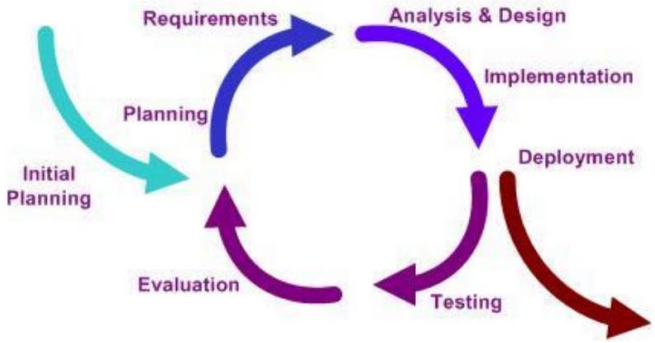
wrapup

Angewendeter iterativ-inkrementeller Softwareentwicklungsprozess in SWEN1/PM3

- Der Softwareentwicklungsprozess wurde so angepasst (engl. tailoring), dass die wesentlichen Artefakte in einem Softwareprojekt im Kontext eingeführt werden können.
- Die Software wird in Iterationen entwickelt (2 Wochen Rhythmus).
- Jede Iteration hat ein Ziel und wird nach Abschluss reviewed.
- Es gibt drei Meilensteine, die im Projektverlauf ein besonderes Ereignis darstellen bzw. den Abschluss einer Phase: Projektskizze (M1), Lösungsarchitektur (M2) und Beta-Release (M3)
- In jeder Iteration werden Anforderungen, Analyse & Design, Implementation und Testing gemacht (Software entsteht in Inkrementen).
- Der angewendete Softwareentwicklungsprozess und das Projektmanagement eines iterativ-inkrementellen Projektes wird in PM3 noch detaillierter erklärt.

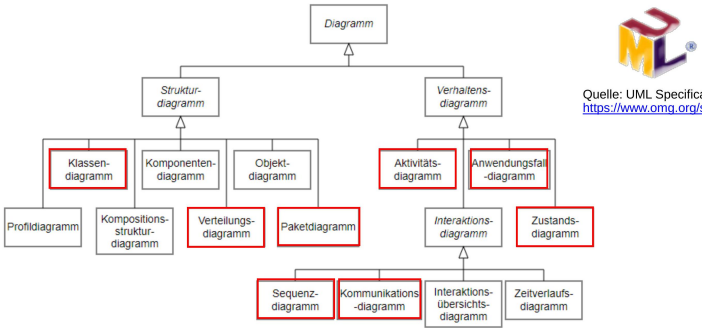
Wesentliche Resultate bzw. Artefakte

- Anforderungsanalyse
- Funktionale Anforderungen mit Use Cases
- Qualitätsanforderungen und Randbedingungen
- Domänenmodell
- Design
- Softwarearchitektur
- Use Case Realisierung (statische und dynamische Modelle)
- Implementation
- Quellcode (inkl. Javadoc)
- Testing
- Unit-Tests



- Integrations- und Systemtests

Modellierung und Modelle mit der UML



Quelle: UML Specifica
https://www.omg.org/s

Quelle: UML Specification, https://www.omg.org/spec/UML/
☐ für die Modellierung in SWEN1 relevant

Gebrauch der UML (nach Martin Fowler)

- UML as a Sketch

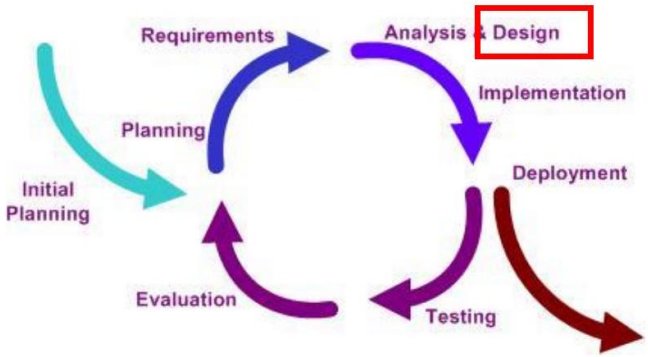
- Informelle und unvollständige Diagramme (z.T. von Hand gezeichnet), um schwierige Teile des Problems oder der Lösung zu verstehen und zu kommunizieren
- Die agile Community bevorzugt diese Anwendungsart von UML
- UML as a Blueprint
- Relativ detaillierte Analyse und Design-Diagramme für Code-Generierung oder um existierenden Code besser zu verstehen
- Klassische UML-Tools für ein Forward- und Reverse-Engineering (Roundtrip)
- UML as a Programming Language
- Komplete, ausführbare Spezifikation eines Software-Systems in UML
- MDA-Tools zur Modellierung und Generierung

Überblick Anforderungen & Analyse

- User Research (Personas und Szenarien, Contextual Inquiry)
- Sketching und Protoyping
- Ableiten und Modellieren von Use Cases (dt. Anwendungsfälle)
- Detaillierung der Use Case (UML-Use-Case-Diagramm, Use-CaseSpezifikationen, UI-Sketching
- Qualitätsanforderungen und Randbedingungen erheben und festhalten.
- Modellierung der Fachlichkeit und Begriffe des Anwenders in einem Domänenmodell (konzeptuelles UML-Klassendiagramm)
- Bei der objektorientierten Analyse (OOA) liegt die Betonung darauf, die Objekte - oder Konzepte in dem Problembereich zu finden und zu beschreiben!

Überblick Design

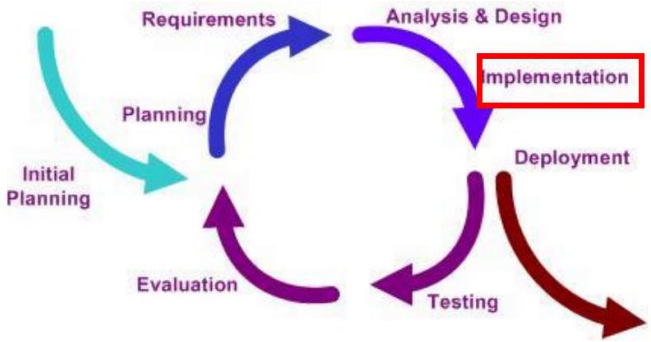
- Design und Modellierung einer für die Problemstellung geeigneten Softwarearchitektur (UML-Paketdiagramm, UML-Verteilungsdiagramm)
- Use-Case-Realisierung und Klassendesign mit Verantwortlichkeiten (UML-Klassendiagramm, UML-Sequenzdiagramm, UMLKommunikationsdiagramm, UML-Zustandsdiagramm, UML-



Aktivitätsdiagramm)
- Entwurf mit bewährten Design Patterns
- Beim objektorientierten Design (OOD) liegt die Betonung darauf, geeignete Softwareobjekte und ihr Zusammenwirken (engl. collaboration) zu definieren, um die Anforderungen zu erfüllen!

Überblick Implementation

- Umsetzung des Designs in Code der entsprechenden (objektorientierten) Programmiersprache
- Verwendung von geeigneten Algorithmen und Datenstrukturen zur Implementierung des Designs
- Code Smells sofort bei deren Aufdeckung verbessern (Refactoring)



- Laufende Dokumentation des Quellcodes (nach Clean CodePrinzipien)

Überblick Testing

- Laufendes Design und Implementierung von Unit-Tests
- Planung, Design und Durchführung von weiteren Tests auf den Teststufen Integration und System je nach Problemstellung
- Dokumentation des Testkonzepts und der Tests