Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Einführung und Überblick

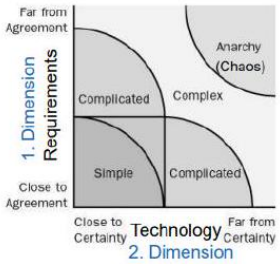
Software Engineering

- **Disziplinen**:
  Anforderungen, Architektur, Implementierung, Test und Wartung
- **Ziel**:
  Strukturierte Prozesse für Qualität, Risiko- & Fehlerminimierung

Softwareentwicklungsprozesse

Dimensionen der Softwareentwicklung

- Requirements (Bekannt - Unbekannt)
- Technology (Bekannt - Unbekannt)
- Skills/Experience (Vorhanden - Nicht vorhanden)



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

Kernprozesse

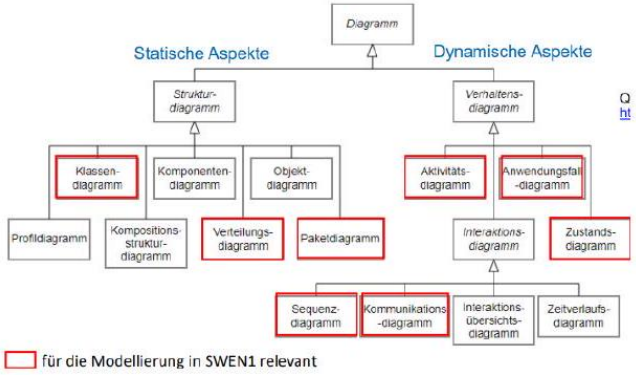
- Anforderungserhebung
- Systemdesign/technische Konzeption
- Implementierung
- Softwaretest
- Softwareeinführung
- Wartung/Pflege

Unterstützungsprozesse

- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

Modelle in der Softwareentwicklung

- Software ist selbst ein Modell der Realität
- Anforderungsmodelle beschreiben das Problem
- Architektur-/Entwurfsmodelle beschreiben die Lösung
- Testmodelle beschreiben korrektes Verhalten



Code and Fix

Codierung und Korrektur im Wechsel

Vorteile:
- Schnell und agil
- Einfach am Anfang

Nachteile:
- Schlecht planbar
- Schwer wartbar
- Änderungen aufwändig

Wasserfallmodell

Sequentielle Phasen mit definierten Ergebnisdokumenten

Vorteile:
- Gut planbar
- Klare Aufteilung in Phasen
- Definierte Meilensteine

Nachteile:
- Schlechtes Risikomanagement
- Spätes Kundenfeedback
- Unflexibel bei Änderungen
- Anforderungen nie vollständig zu Beginn bekannt

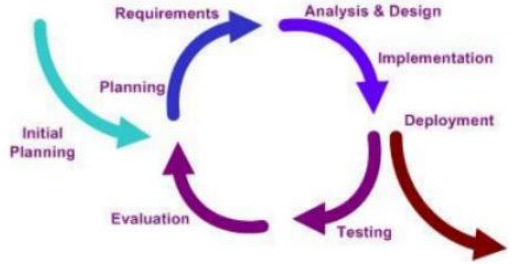
Iterativ-inkrementelle Modelle

Schrittweise Entwicklung in geplanten Iterationen

Vorteile:
- Flexibles Modell
- Gutes Risikomanagement
- Frühe Einsetzbarkeit
- Kontinuierliches Kundenfeedback

Nachteile:
- Planung upfront hat Grenzen
- Höherer Koordinationsaufwand

Basis für agile Entwicklung:
- Fokus auf funktionierender Software
- Kurze Iterationen
- Enge Kundeneinbindung



Charakteristiken iterativ-inkrementeller Prozesse

- Projekt-Abwicklung in Iterationen (Mini-Projekte)
- Inkrementelle Entwicklung (Stück für Stück)
- Risiko-getriebene Iterationsziele
- Reviews und Learnings nach jeder Iteration
- Demming-Cycle: Plan, Do, Check, Act

Typische Prüfungsaufgabe: Prozessmodelle vergleichen

Vergleichen Sie das Wasserfallmodell mit einem iterativ-inkrementellen Ansatz anhand folgender Kriterien:
Umgang mit sich ändernden Anforderungen, Risikomanagement, Planbarkeit, Kundeneinbindung

Musterlösung:
- Wasserfall:
  - Änderungen schwierig zu integrieren
  - Risiken erst spät erkennbar
  - Gut planbar durch feste Phasen
  - Kunde hauptsächlich am Anfang und Ende involviert
- Iterativ-inkrementell:
  - Flexibel bei Änderungen
  - Frühes Erkennen von Risiken
  - Planung pro Iteration
  - Kontinuierliches Kundenfeedback

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

Modellierung

Modellierung in der Softwareentwicklung

- Modelle als Abstraktionen: Anforderungen, Architekturen, Testfälle.
- Einsatz von UML: Skizzen, detaillierte Blueprints, vollständige Spezifikationen.
- Zweck:
  - Verstehen eines Gebildes
  - Kommunizieren über ein Gebilde
  - Gedankliches Hilfsmittel zum Gestalten, Bewerten, Kritisieren
  - Spezifikation von Anforderungen
  - Durchführung von Experimenten

Modellierungsumfang bestimmen

Der benötigte Modellierungsumfang hängt ab von:
- Komplexität der Problemstellung
- Anzahl beteiligter Stakeholder
- Kritikalität des Systems
- Domänenspezifische Anforderungen
- Analogie: Planung einer Hundehütte vs. Haus vs. Wolkenkratzer

Prüfungsfrage zur Modellierung

Erklären Sie anhand eines selbst gewählten Beispiels, warum der Modellierungsaufwand je nach Projekt stark variieren kann. Nennen Sie mindestens drei Faktoren, die den Modellierungsumfang beeinflussen.

Mögliche Antwort:
- Beispiel: Entwicklung einer Smartphone-App vs. Medizinisches Gerät
- Faktoren:
  - Komplexität der Domäne
  - Regulatorische Anforderungen
  - Anzahl beteiligter Stakeholder
  - Sicherheitsanforderungen

Unified Modeling Language (UML)

Standardsprache für grafische Modellierung:
- **Einsatz als:**
  - Sketch: Informelle Kommunikation und Verständnis
  - Blueprint: Detaillierte Design-Spezifikation
  - Programming Language: Ausführbare Modellierung
- **Vorteile:**
  - Standardisierte Notation
  - Verschiedene Abstraktionsebenen
  - Unterstützung des gesamten Entwicklungszyklus

Anforderungsanalyse

Software Engineering

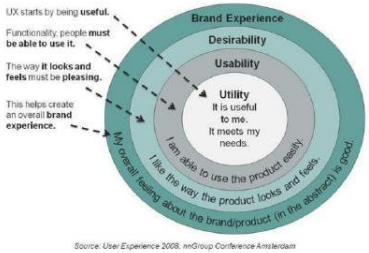
- **Disziplinen:** Anforderungen, Architektur, Implementierung, Test und Wartung
- **Ziel:** Strukturierte Prozesse für Qualität, Risiko- & Fehlerminimierung

Usability und User Experience

Usability und User Experience

drei Säulen der Benutzererfahrung:

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



Wichtige Aspekte: Benutzer und seine Ziele/Aufgaben, Kontext der Nutzung, Softwaresystem (inkl. UI)

Usability-Dimensionen nach ISO 9241

- **Effektivität:**
  - Der Benutzer kann alle Aufgaben vollständig erfüllen
  - Gewünschte Genauigkeit wird erreicht
  - Ziele werden im vorgegebenen Kontext erreicht
- **Effizienz:** Minimaler Aufwand für:
  - Mentale Belastung
  - Physische Anstrengung
  - Zeitlicher Aufwand
  - Ressourceneinsatz
- **Zufriedenheit:**
  - Minimum: Keine Verärgerung
  - Standard: Zufriedenheit
  - Optimal: Begeisterung
  - Subjektive Nutzererfahrung

Usability-Evaluation durchführen

1. **Vorbereitung**
   - Testziele definieren
   - Testpersonen auswählen
   - Testaufgaben erstellen
2. **Durchführung**
   - Beobachtung der Nutzer
   - Protokollierung von Problemen
   - Zeitmessung der Aufgaben
3. **Auswertung**
   - Probleme klassifizieren
   - Schweregrad bestimmen
   - Verbesserungen vorschlagen
4. **Dokumentation**
   - Ergebnisse zusammenfassen
   - Empfehlungen formulieren
   - Maßnahmen priorisieren

ISO 9241-110: Usability-Anforderungen

- **Aufgabenangemessenheit:**
  - Funktionalität unterstützt Arbeitsaufgaben
  - Keine unnötige Komplexität
- **Selbstbeschreibungsfähigkeit:**
  - Verständliche Benutzerführung
  - Klare Statusanzeigen
- **Steuerbarkeit:**
  - Benutzer kontrolliert Ablauf
  - Geschwindigkeit anpassbar
- **Erwartungskonformität:**
  - Konsistentes Verhalten
  - Bekannte Konventionen
- **Fehlertoleranz:**
  - Fehler vermeiden
  - Fehlerkorrektur ermöglichen
- **Individualisierbarkeit:**
  - Anpassung an Benutzergruppen
  - Flexible Nutzung
- **Lernförderlichkeit:**
  - Einfacher Einstieg
  - Unterstützung beim Lernen

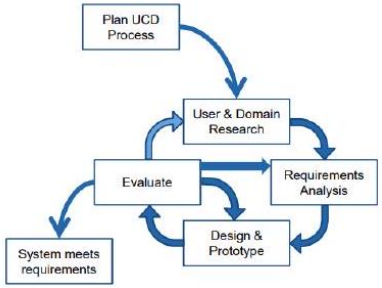
User-Centered Design (UCD)

UCD Prozess

Ein iterativer Prozess zur nutzerzentrierten Entwicklung, der die Bedürfnisse, Wünsche und Einschränkungen der Benutzer in jeder Phase des Design-Prozesses berücksichtigt.

Hauptziele:

- Benutzerfreundlichkeit
- Effektive Nutzung
- Hohe Akzeptanz

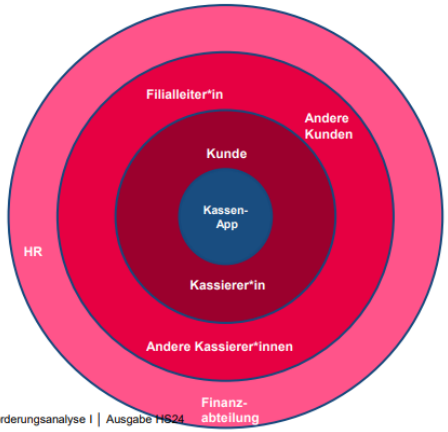


Wichtige Artefakte

- 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

Stakeholder Map

Zeigt die wichtigsten Stakeholder im Umfeld der Problemdomäne.



UCD Prozess-Phasen

1. **User & Domain Research** (see KR)
2. **Requirements Analysis** (see KR)
3. **Design & Prototype**
   - Interaktionskonzept entwickeln
   - Wireframes erstellen
   - Prototypen bauen
   - Design iterativ verbessern
4. **Evaluate**
   - Mit Benutzern testen
   - Feedback sammeln
   - Probleme identifizieren
   - Verbesserungen einarbeiten

User & Domain Research

1. **Zielgruppe identifizieren**
   - Wer sind die Benutzer?
   - Was sind ihre Aufgaben/Ziele?
   - Wie sieht ihre Arbeitsumgebung aus?
   - Welche Sprache/Begriffe verwenden sie?
2. **Daten sammeln durch**
   - Contextual Inquiry
   - Interviews
   - Beobachtung
   - Fokusgruppen
   - Nutzungsauswertung
3. **Ergebnisse dokumentieren in**
   - Personas
   - Usage-Szenarien
   - Mentales Modell

UCD Artefakte erstellen

1. **Personas**
   - Daten aus User Research sammeln
   - Gemeinsame Merkmale identifizieren
   - Repräsentative Person definieren
   - Details ausarbeiten:
     - Demografische Daten
     - Ziele und Motivation
     - Fähigkeiten/Kenntnisse
     - Frustrationspunkte
2. **Usage-Szenarien**
   - Kontext beschreiben
   - Akteure identifizieren
   - Ablauf definieren
   - Probleme/Lösungen darstellen
3. **Mentales Modell**
   - Nutzerverständnis dokumentieren
   - Konzepte und Beziehungen visualisieren
   - Mit Fachmodell abgleichen

Usage-Szenario: Online-Banking

Kontext: Sarah möchte eine Überweisung tätigen

Aktuelles Szenario: Sarah loggt sich in ihr Online-Banking ein. Sie sucht nach der letzten Überweisung an ihren Vermieter, um die Kontodetails zu finden. Nach mehreren Klicks findet sie die Information und kopiert die IBAN. Sie öffnet das Überweisungsformular und fügt die Daten ein. Beim Absenden erscheint eine Fehlermeldung, weil sie vergessen hat, den Verwendungszweck einzutragen.

Probleme:

- Umständliche Suche nach Kontodetails
- Fehleranfällige manuelle Dateneingabe
- Späte Validierung der Eingaben

Verbessertes Szenario: Sarah wählt aus einer Liste ihrer häufigen Empfänger ihren Vermieter aus. Das System füllt automatisch alle bekannten Daten ein. Fehlende Pflichtfelder sind deutlich markiert. Sarah ergänzt den Verwendungszweck und sendet die Überweisung ab.

Weitere Beispiele z.B. Persona erstellen auf nächster Seite

Persona erstellen

Aufgabe: Erstellen Sie eine Persona für ein Online-Banking-System.

Lösung: Sarah Schmidt, 34, Projektmanagerin

- **Hintergrund:**
  - Arbeitet Vollzeit in IT-Firma
  - Technik-affin, aber keine Expertin
  - Nutzt Smartphone für die meisten Aufgaben
- **Ziele:**
  - Schnelle Überweisungen zwischen Konten
  - Überblick über Ausgaben
  - Sichere Authentifizierung
- **Frustrationen:**
  - Komplexe Menüführung
  - Lange Ladezeiten
  - Mehrfache Login-Prozesse

Persona für E-Learning-System

Thomas Weber, 19, Informatik-Student

Hintergrund:

- Erstsemester-Student
- Arbeitet nebenbei 10h/Woche
- Pendelt zur Universität (1h pro Weg)

Technische Fähigkeiten:

- Versiert im Umgang mit Computern
- Nutzt hauptsächlich Smartphone für Online-Aktivitäten
- Kennt gängige Learning-Management-Systeme

Ziele:

- Effizientes Lernen trotz Zeitdruck
- Flexible Zugriffsmöglichkeiten auf Lernmaterialien
- Gute Prüfungsvorbereitung

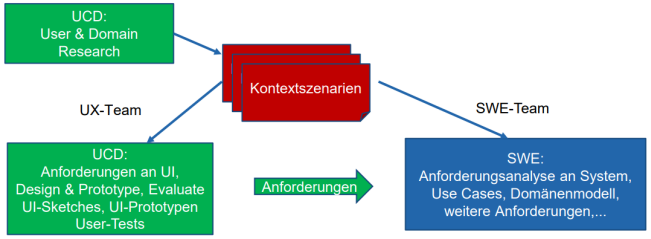
Frustrationen:

- Unübersichtliche Kursstrukturen
- Fehlende Mobile-Optimierung
- Schwierige Navigation zwischen Materialien

Requirements Engineering

Requirements (Anforderungen)

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



Charakteristiken:

- Können explizit oder implizit sein
- Sind fast nie im Vorneherein vollständig bekannt
- Müssen mit allen Stakeholdern erarbeitet werden
- Entwickeln sich während des Projekts
- Müssen verifizierbar und messbar sein

Herkunft:

- Benutzer (Ziele, Bedürfnisse, Kontext)
- Weitere Stakeholder (Management, IT, etc.)
- Regulatorien, Gesetze, Normen

Arten von Anforderungen

Funktionale Anforderungen:

- Beschreiben, WAS das System tun soll
- Werden in Use Cases dokumentiert
- Müssen konkret und testbar sein

Nicht-funktionale Anforderungen (ISO 25010):

- Performance Efficiency
  - Time Behaviour
  - Resource Utilization
  - Capacity
- Compatibility
  - Co-existence
  - Interoperability
- Usability (siehe oben)
- Reliability
  - Maturity
  - Availability
  - Fault Tolerance
  - Recoverability
- Security
- Maintainability
- Portability

Randbedingungen:

- Technische Einschränkungen
- Rechtliche Vorgaben
- Budgetäre Grenzen
- Zeitliche Limitationen

Requirements Analysis

- Benutzeranforderungen ableiten
- Kontextszenarien erstellen
- UI-Skizzen entwickeln
- Use Cases definieren

Stakeholder identifizieren

- Benutzer
- Auftraggeber
- Weitere Interessengruppen

Anforderungsquellen analysieren

- Interviews und Workshops
- Existierende Dokumente
- Beobachtung der Arbeitsabläufe

Anforderungen dokumentieren

- Funktionale Anforderungen (Use Cases)
- Nicht-funktionale Anforderungen
- Randbedingungen

Anforderungen validieren

- Review mit Stakeholdern
- Priorisierung
- Machbarkeitsanalyse

Anforderungsanalyse: Onlineshop

Ausgangssituation: Ein traditioneller Buchladen möchte einen Onlineshop entwickeln.

Funktionale Anforderungen:

- Produktkatalog durchsuchen
- Warenkorb verwalten
- Bestellung aufgeben
- Kundenkonto verwalten

Nicht-funktionale Anforderungen:

- Performance:
  - Seitenaufbau < 2 Sekunden
  - Suche < 1 Sekunde
- Sicherheit:
  - HTTPS-Verschlüsselung
  - Zwei-Faktor-Authentifizierung
- Usability:
  - Responsive Design
  - Max. 3 Klicks zur Bestellung

Randbedingungen:

- DSGVO-Konformität
- Integration mit bestehendem ERP
- Budget: 100.000 EUR
- Launch in 6 Monaten

Use Cases

Use Case (Anwendungsfall)

Ein Use Case beschreibt eine konkrete Interaktion zwischen Akteur und System mit folgenden Eigenschaften:

Grundprinzipien:

- Aus Sicht des Akteurs beschrieben
- Aktiv formuliert (Verb + Objekt)
- Konkreter Nutzen für Akteur
- Mehr als eine einzelne Interaktion
- Essentieller Stil (Logik statt Implementierung)

Qualitätskriterien:

- Boss-Test: Sinnvolle Arbeitseinheit
- EBP-Test: Elementary Business Process
- Size-Test: Mehrere Interaktionen

Use Case Erstellung

Schritte zur Erstellung eines vollständigen Use Cases:

1. **Identifikation:** siehe **Use Case Identifikation**
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

Use Case Identifikation

1. **Systemgrenzen definieren**
   - Was gehört zum System?
   - Was ist externe Umgebung?
2. **Akteure identifizieren**
3. **Ziele ermitteln**
   - Geschäftsziele
   - Benutzerziele
   - Systemziele

Akteure in Use Cases

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

Use Case Beziehungen

Include-Beziehung:

- Ein UC schließt einen anderen UC ein
- Wiederverwendung von Funktionalität
- Obligatorische Beziehung

Extend-Beziehung:

- Optionale Erweiterung eines UC
- Unter bestimmten Bedingungen
- Ursprünglicher UC bleibt unverändert

Generalisierung:

- Spezialisierung von Akteuren/UCs
- Vererbung von Eigenschaften
- ïst-einBeziehung

Use Case Granularität

1. **Brief Use Case**
   - Kurze Zusammenfassung
   - Hauptablauf skizzieren
   - Keine Details zu Varianten
2. **Casual Use Case**
   - Mehrere Absätze
   - Hauptvarianten beschreiben
   - Informeller Stil
3. **Fully-dressed Use Case**
   - Vollständige Struktur
   - Alle Varianten
   - Vor- und Nachbedingungen
   - Garantien definieren

Fully-dressed Use Case erstellen

1. Grundinformationen

- Aussagekräftiger Name (aktiv)
- Umfang (Scope)
- Ebene (Level)
- Primärakteur

2. Stakeholder und Interessen

- Alle beteiligten Parteien
- Deren spezifische Interessen

3. Vor- und Nachbedingungen

- Was muss vorher erfüllt sein?
- Was ist nachher garantiert?

4. Standardablauf

- Nummerierte Schritte
- Akteur-System-Interaktion
- Klare Erfolgskriterien

5. Erweiterungen

- Alternative Abläufe
- Fehlerszenarien
- Verzweigungen

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.

Casual Use Case **UC: Verkauf abwickeln**

Der Umfang des Use Cases ist das Kassensystem. Der Primärakteur ist der Kassier. Der Stakeholder ist der Kunde, der eine schnelle Abwicklung wünscht, und das Geschäft, das eine korrekte Abrechnung benötigt. Die Vorbedingung ist, dass die Kasse geöffnet ist.

Der Standardablauf ist wie folgt: Kassier startet neuen Verkauf und System initialisiert neue Transaktion. Kassier erfasst Produkte und System zeigt Zwischensumme. Kassier schliesst Verkauf ab und System zeigt Gesamtbetrag. Kunde bezahlt und 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

Fully-dressed Use Case

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

Typische Prüfungsaufgabe: Use Case Analyse

Aufgabe: Analysieren Sie den folgenden Use Case und identifizieren Sie mögliche Probleme:

Use Case: "Der Benutzer loggt sich ein und das System zeigt die Startseite. Er klickt auf den Button und die Daten werden in der Datenbank gespeichert."

Probleme:

- Zu technisch/implementierungsnah
- Fehlende Akteurperspektive
- Unklarer Nutzen/Ziel
- Fehlende Alternativszenarien
- Keine Fehlerbehandlung

Verbesserter Use Case: "Der Kunde möchte seine Bestelldaten speichern. Er bestätigt die Eingaben und erhält eine Bestätigung über die erfolgreiche Speicherung."

Prüfungsaufgabe: Use Case Analyse

Aufgabe: Analysieren Sie folgenden Use Case und verbessern Sie ihn.

Ursprünglicher Use Case: "Der User loggt sich ein. Das System überprüft seine Credentials in der Datenbank. Bei erfolgreicher Validierung wird die Startseite angezeigt. Der User klickt auf 'Profil bearbeiten' und das System speichert die Änderungen in der Datenbank."

Probleme:

- Technische Implementierungsdetails
- Fehlende Akteurperspektive
- Keine Alternativen/Fehlerbehandlung
- Unklarer Nutzen/Ziel

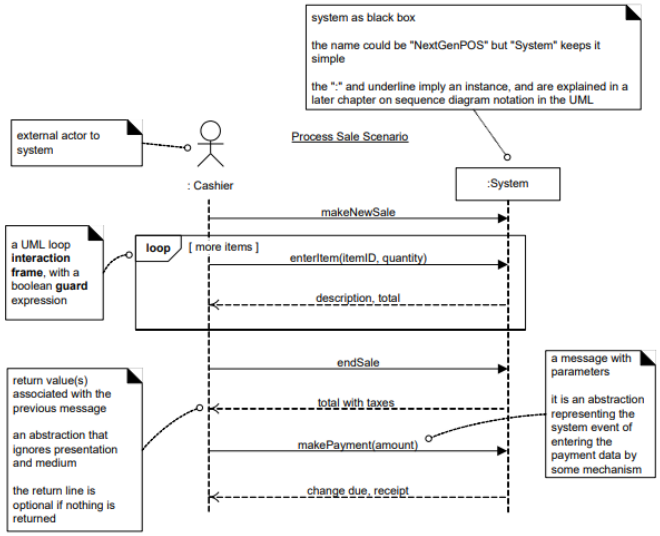
Verbesserter Use Case: "Profilinformationen aktualisieren"

- **Primärakteur:** Registrierter Benutzer
- **Vorbedingung:** Benutzer ist authentifiziert
- **Standardablauf:**
  1. Benutzer wählt Profilbearbeitung
  2. System zeigt aktuelle Profildaten
  3. Benutzer ändert gewünschte Informationen
  4. System prüft Änderungen
  5. System bestätigt erfolgreiche Aktualisierung
- **Erweiterungen:**
  - 4a: Ungültige Eingaben
  - 4b: Verbindungsfehler

System Sequence Diagrams

Systemsequenzdiagramm (SSD)

Formalisierte Darstellung der System-Interaktionen:



System Sequence Diagram

Ein SSD visualisiert die Interaktion zwischen Akteur und System auf einer höheren Abstraktionsebene:

Hauptmerkmale:

- Zeigt Input/Output-Events
- Identifiziert Systemoperationen
- Bildet Basis für API-Design
- Abstrahiert von UI-Details

Notationselemente:

- Akteur und System als Lebenslinien
- Methodenaufrufe als durchgezogene Pfeile
- Rückgabewerte als gestrichelte Pfeile
- Parameter für benötigte Informationen

System Sequence Diagram erstellen

1. Vorbereitung

- Use Case als Grundlage wählen
- Standardablauf identifizieren
- Akteur und System festlegen

2. Methodenaufrufe definieren

- Aussagekräftige Namen wählen
- Notwendige Parameter bestimmen
- Rückgabewerte festlegen

3. Zeitliche Abfolge

- Sequenz der Aufrufe modellieren
- Abhängigkeiten beachten
- Kontrollstrukturen einbauen (alt, loop, etc.)

4. Externe Systeme

- Bei Bedarf weitere Akteure einbinden
- Schnittstellen definieren
- Kommunikationsfluss darstellen

Systemoperationen definieren

Namenskonventionen:

- Verben für Aktionen
- Substantive für Entitäten
- Präzise, aber nicht technisch

Parameter:

- Nur notwendige Information
- Domänenorientierte Typen
- Sinnvolle Standardwerte

Rückgabewerte:

- Eindeutige Bestätigungen
- Relevante Geschäftsobjekte
- Fehlerindikationen

Beispiele guter Operationen:

```
// Gut - klar und domaenenorientiert
createOrder(customer: CustomerId): OrderId
addOrderItem(orderId: OrderId,
             product: ProductId,
             quantity: int)

// Schlecht - zu technisch/implementierungsnah
insertIntoOrderTable(customerData: Map)
updateOrderItemList(items: ArrayList)
```

Contracts für Systemoperationen

Ein Contract definiert die Vor- und Nachbedingungen einer Systemoperation:

1. Struktur

- Name und Parameter
- Querverweis zum Use Case
- Vorbedingungen
- Nachbedingungen

2. Vorbedingungen

- Systemzustand vor Aufruf
- Notwendige Initialisierungen
- Gültige Parameter

3. Nachbedingungen

- Erstellte/gelöschte Instanzen
- Geänderte Attribute
- Neue/gelöschte Assoziationen

Contract für enterItem()

Operation: enterItem(itemId: ItemID, quantity: int)
Querverweis: UC "Process Sale"
Vorbedingungen:

- Verkauf ist gestartet
- ItemID existiert im System

Nachbedingungen:

- SalesLineItem-Instanz wurde erstellt
- Verknüpfung mit aktueller Sale-Instanz
- quantity wurde gesetzt
- Verknüpfung mit ProductDescription

SSD Übungsaufgabe

Aufgabe: Erstellen Sie ein Systemsequenzdiagramm für den Use Case 'Geld abheben' an einem Bankautomaten.

Wichtige Aspekte:

- Kartenvalidierung
- PIN-Eingabe
- Betragseingabe
- Kontostandsprüfung
- Geldausgabe
- Belegdruck

Essentielle Systemoperationen:

- validateCard(cardNumber)
- checkPIN(pin)
- withdrawMoney(amount)
- printReceipt()

Sequenzdiagramm: TO BE ADDED

SSD: Online-Banking Überweisung

Use Case: Überweisung durchführen
Systemoperationen:

```
// Kontostand pruefen
checkBalance(): Money

// Ueberweisung initiieren
initiateTransfer(recipient: String,
                 iban: String,
                 amount: Money,
                 purpose: String): TransferId

// TAN anfordern
requestTAN(transferId: TransferId): void

// Ueberweisung bestaetigen
confirmTransfer(transferId: TransferId,
                tan: String): Boolean
```

Wichtige Aspekte:

- Validierung vor Ausführung
- Zweistufige Bestätigung
- Klare Rückmeldungen
- Fehlerbehandlung

Sequenzdiagramm: TO BE ADDED

SSD: Typische Prüfungsaufgabe

Aufgabe: Erstellen Sie ein SSD für den Use Case "Produkt bestellenïn einem Webshop.
Analyse:

- Identifiziere Hauptaktionen:
  - Warenkorb verwalten
  - Bestellung aufgeben
  - Zahlung durchführen
- Definiere Systemoperationen:
  - addToCart(productId, quantity)
  - showCart(): CartContents
  - checkout(shippingAddress, paymentMethod)
  - confirmOrder(): OrderId
- Berücksichtige Rückgabewerte:
  - Bestätigungen
  - Zwischensummen
  - Fehlermeldungen

Sequenzdiagramm: TO BE ADDED

SSD: Integration mit externen Systemen

Use Case: Kreditkartenzahlung durchführen
Beteiligte Systeme:

- Verkaufssystem (SuD)
- Kreditkarten-Autorisierungssystem
- Buchhaltungssystem

Systemoperationen:

```
// Request credit card approval
requestApproval(cardNum: String,
                expiryDate: Date,
                amount: Money): Boolean

// Post transaction to accounting
postTransaction(transactionData:
    TransactionData)
```

Wichtige Aspekte:

- Asynchrone Kommunikation
- Fehlerbehandlung über mehrere Systeme
- Transaktionsmanagement
- Logging und Nachvollziehbarkeit

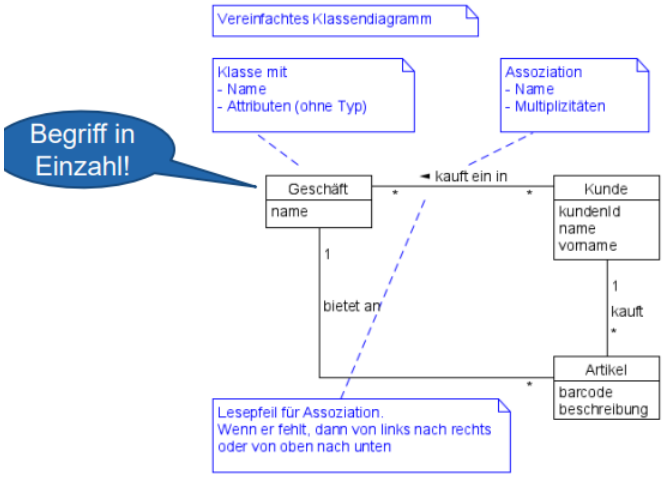
Sequenzdiagramm: TO BE ADDED

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



Domänenmodell Erstellung

Schritt 1: Konzepte identifizieren

- Substantive aus Anforderungen extrahieren
- Kategorien prüfen:
  - Physische Objekte (Produkte, Geräte)
  - Kataloge und Listen
  - Container (Warenkorb, Lager)
  - Externe Systeme
  - Rollen von Personen
  - Artefakte (Pläne, Dokumente, Verträge)
  - Zahlungsinstrumente (Transaktionen)
  - **Wichtig:** Keine Softwareklassen modellieren!
- Irrelevante Konzepte ausschließen
- Synonyme vereinheitlichen

Schritt 2: Attribute definieren

- Nur wichtige/zentrale Attribute
- Typische Kategorien:
  - Transaktionsdaten
  - Teil-Ganzes Beziehungen
  - Beschreibungen
  - Verwendungszwecke
- **Wichtig:** Beziehungen als Assoziationen, nicht als Attribute!
- Keine technischen IDs
- Keine abgeleiteten Werte

Schritt 3: Beziehungen modellieren

- Assoziationen zwischen Konzepten identifizieren
- Multiplizitäten festlegen
- Art der Beziehung bestimmen
- Richtung der Assoziation falls nötig
- Rollen an Assoziationsenden benennen

Domänenmodell Zweck

- Visualisierung der Fachdomäne für alle Stakeholder
- Grundlage für das spätere Softwaredesign
- Gemeinsames Verständnis der Begriffe und Zusammenhänge
- Dokumentation der fachlichen Strukturen
- Basis für die Kommunikation zwischen Entwicklung und Fachbereich

Analysemuster im Domänenmodell

Analysemuster im Überblick

Details siehe nächste Seite

Bewährte Strukturen für wiederkehrende Modellierungssituationen:

- **Beschreibungsklassen:** Trennung von Typ und Instanz
- **Generalisierung:** ïst-ein"Beziehungen
- **Komposition:** Starke Teil-Ganzes Beziehung
- **Zustände:** Eigene Zustandshierarchie
- **Rollen:** Verschiedene Funktionen eines Konzepts
- **Assoziationsklasse:** Attribute einer Beziehung

Musterauswahl und Kombination

Systematisches Vorgehen bei der Anwendung von Analysemustern:

1. **Analyse der Situation**
   - Konzepte und Beziehungen identifizieren, Attribute zuordnen
   - Probleme im einfachen Modell erkennen (Bei Kombination)
2. **Passende Muster identifizieren**
   - Beschreibungsklassen bei gleichartigen Objekten
   - Generalisierung bei ïst-einBeziehungen
   - Komposition bei existenzabhängigen Teilen
   - Zustände bei Objektlebenszyklen
   - Rollen bei verschiedenen Funktionen
   - Assoziationsklassen bei Beziehungsattributen
   - Wertobjekte bei komplexen Werten
3. **Musterauswahl**
   - Vor- und Nachteile abwägen
   - Komplexität vs. Nutzen bewerten
4. **Muster anwenden**
   - Struktur des Musters übernehmen, an Kontext anpassen
   - Konsistenz und fachliche Korrektheit sicherstellen
5. **Muster kombinieren**
   - An Kontext anpassen und mit bestehenden Elementen verbinden
   - Überschneidungen identifizieren und Konflikte auflösen
   - Gesamtmodell harmonisieren
   - Konsistenz und fachliche Korrektheit sicherstellen

Prüfungsaufgabe: Konzeptidentifikation

Aufgabentext: Ën Bibliothekssystem verwaltet Bücher, die von Mitgliedern ausgeliehen werden können. Jedes Buch hat eine ISBN und mehrere Exemplare. Mitglieder können maximal 5 Bücher gleichzeitig für 4 Wochen ausleihen. Bei Überschreitung wird eine Mahngebühr fällig."

Identifizierte Konzepte: Buch (Beschreibungsklasse), Exemplar (Physisches Objekt), Mitglied (Rolle), Ausleihe (Transaktion), Mahnung (Artefakt)

Begründung:

- Buch/Exemplar:
  Trennung wegen mehrfacher Exemplare (Beschreibungsmuster)
- Ausleihe: Verbindet Exemplar und Mitglied, hat Zeitbezug
- Mahnung: Entsteht bei Fristüberschreitung

Review eines Domänenmodells

Checkliste für die Überprüfung:

- **Fachliche Korrektheit**
  - Alle relevanten Konzepte vorhanden?
  - Begriffe aus der Fachdomäne verwendet?
  - Beziehungen fachlich sinnvoll?
- **Technische Korrektheit**
  - UML-Notation korrekt?
  - Multiplizitäten angegeben?
  - Assoziationsnamen vorhanden?
- **Modellqualität**
  - Angemessener Detaillierungsgrad?
  - Analysemuster sinnvoll eingesetzt?
  - Keine Implementation vorweggenommen?

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 oder abstrakt
  - Fachliche Begriffe verwenden
- **Assoziationen statt Attribute**
  - Beziehungen als Assoziationen modellieren
  - Keine Objekt-IDs als Attribute

Typische Modellierungsfehler

Fehler 1: Technische statt fachliche Klassen

- **Falsch:** CustomerManager, OrderController, DatabaseHandler
- **Richtig:** Kunde, Bestellung, Produkt

Fehler 2: IDs als Attribute statt Assoziationen

- **Falsch:** customerId: String, orderId: Integer
- **Richtig:** Direkte Assoziation zwischen Kunde und Bestellung

Fehler 3: Implementierungsdetails

- **Falsch:** saveToDatabase(), validateInput(), createPDF()
- **Richtig:** Keine Operationen im Domänenmodell

Typische Prüfungsaufgabe: Modell verbessern

Fehlerhaftes Modell:

- Klasse 'UserManager' mit CRUD-Operationen
- Attribute 'customerID' und 'orderID' statt Assoziationen
- Operation 'calculateTotal()' in Bestellung
- Technische Klasse "DatabaseConnection"

Verbesserungen:

- 'UserManager' entfernen, stattdessen Beziehungen modellieren
- IDs durch direkte Assoziationen ersetzen
- Operationen entfernen (gehören ins Design)
- Technische Klassen entfernen

Detaillierte Analysemuster

1. Beschreibungsklassen

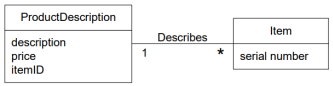
Trennt die Beschreibung eines Typs von seinen konkreten Instanzen.

Anwendung:

- Bei mehreren gleichartigen Objekten
- Gemeinsame unveränderliche Eigenschaften
- Vermeidung von Redundanz

Beispielstruktur:

- ProductDescription (Typ)
  - price, description, itemID
- Product (Instanz)
  - serialNumber



Beschreibungsklassen in der Praxis

Szenario: Bibliothekssystem

Problem: Ein Buch kann mehrere physische Exemplare haben, die alle dieselben Grunddaten (Titel, Autor, ISBN) aber unterschiedliche Zustände (ausgeliehen, verfügbar) haben.

Lösung:

- **Book** (Beschreibungsklasse)
  - title, author, isbn, publisher
- **BookCopy** (Instanzklasse)
  - inventoryNumber, status, location
- Assoziation: BookCopy "beschrieben durch"Book

2. Generalisierung/Spezialisierung

Regeln:

- 100% Regel: Jede Instanz der Spezialisierung ist auch Instanz der Generalisierung
- 'IS-A' Beziehung
- Gemeinsame Attribute/Assoziationen als Grund für Generalisierung
- Gemeinsame Eigenschaften in Basisklasse
- Spezifische Eigenschaften in Unterklassen

Beispiele:

- Person → Student, Dozent
- Zahlung → Barzahlung, Kreditkartenzahlung
- Dokument → Rechnung, Lieferschein

Generalisierung im Online-Shop

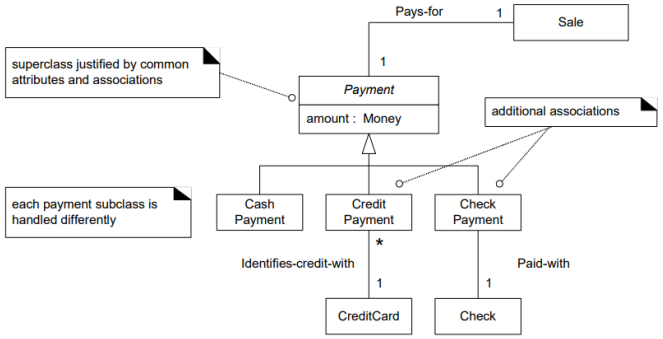
Szenario: Versch. Zahlungsarten

Struktur:

- **Payment** (abstrakt)
  - amount, date, status
- **CashPayment**
  - receivedAmount, changeAmount
- **CreditCardPayment**
  - cardType, authorizationCode

Begründung:

- Gemeinsame Attribute in Payment
- Spezifische Attribute in Unterklassen
- Jede Zahlung ist genau ein Typ



3. Komposition

Modelliert eine starke Teil-Ganzes Beziehung mit Existenzabhängigkeit der Teile.

Eigenschaften:

- Teile können nicht ohne Ganzes existieren
- Teil gehört zu genau einem Ganzen
- Löschen des Ganzen löscht alle Teile

Notation:

- Ausgefüllte Raute am "GanzesEnde
- Multiplizität am "TeilEnde

Komposition im Bestellsystem

Szenario: Bestellung mit Bestellpositionen

Struktur:

- **Order**
  - orderDate, status
- **OrderItem**
  - quantity, price
- Komposition von Order zu OrderItem (1 zu *)

Begründung:

- OrderItems existieren nur im Kontext einer Order
- Löschen der Order löscht alle OrderItems
- Ein OrderItem gehört zu genau einer Order

4. Zustandsmodellierung

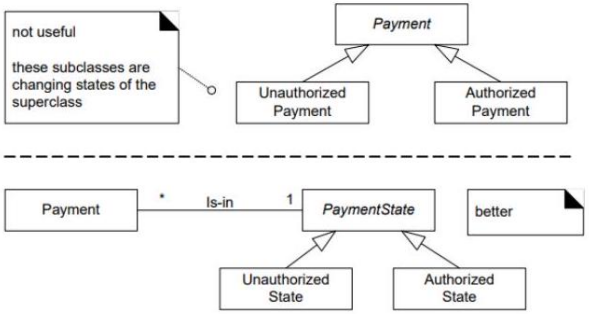
Modelliert Zustände als eigene Konzepthierarchie statt als Attribut.

Vorteile:

- Klare Strukturierung der Zustände
- Vermeidet problematische Vererbung
- Erweiterbarkeit durch neue Zustandsklassen
- Vermeidung von if/else Kaskaden
- Zustandsspezifisches Verhalten möglich

Beispielstruktur:

- OrderState (abstrakt)
  - New, InProgress, Completed
- Order ïst inÖrderState



Zustandsmodellierung: Ticketsystem

Szenario: Support-Tickets mit verschiedenen Status

Falsche Modellierung:

```
class Ticket {
    enum Status {NEW, OPEN, IN_PROGRESS,
        RESOLVED, CLOSED}
    private Status status;
}
```

Bessere Modellierung:

- **TicketState** (abstrakt)
  - timestamp, changedBy
- Konkrete Zustände:
  - NewState: assignedTo
  - OpenState: priority
  - InProgressState: estimatedCompletion
  - ResolvedState: solution
  - ClosedState: closureReason

5. Rollen

Modelliert verschiedene Funktionen eines Konzepts.

Varianten:

- Rollen als eigene Konzepte
- Rollen als Assoziationsenden
- Rollen durch Generalisierung

Anwendung:

- Bei verschiedenen Verantwortlichkeiten
- Wenn Rollen wechseln können
- Bei unterschiedlichen Beziehungen

Rollenmuster: Universitätssystem

Szenario: Person kann gleichzeitig Student und Tutor sein

Variante 1: Rollen als Konzepte

- **Person**
  - name, birthDate, address
- **StudentRole**
  - matriculationNumber, program
- **TutorRole**
  - department, hourlyRate

Variante 2: Generalisierung

- Person als Basisklasse
- Student und Tutor als Spezialisierungen
- Problem: Mehrfachrollen schwierig

6. Assoziationsklassen

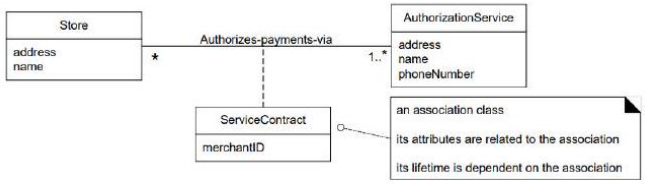
Modelliert Attribute einer Beziehung zwischen Konzepten, eigene Klasse für die Assoziation

Einsatz wenn:

- Attribute zur Beziehung gehören
- Beziehung eigene Identität hat
- Mehrere Beziehungen möglich sind

Notation:

- Gestrichelte Linie zur Assoziation
- Klasse enthält beziehungsspezifische Attribute



Assoziationsklasse: Kursbuchungssystem

Szenario: Studenten können sich für Kurse einschreiben

Struktur:

- **Student** und **Course** als Hauptkonzepte
- **Enrollment** als Assoziationsklasse:
  - enrollmentDate
  - grade
  - attendance
  - status

Begründung:

- Noten gehören zur Einschreibung
- Student kann mehrere Kurse belegen
- Kurs hat mehrere Studenten
- Einschreibungsdaten sind beziehungsspezifisch

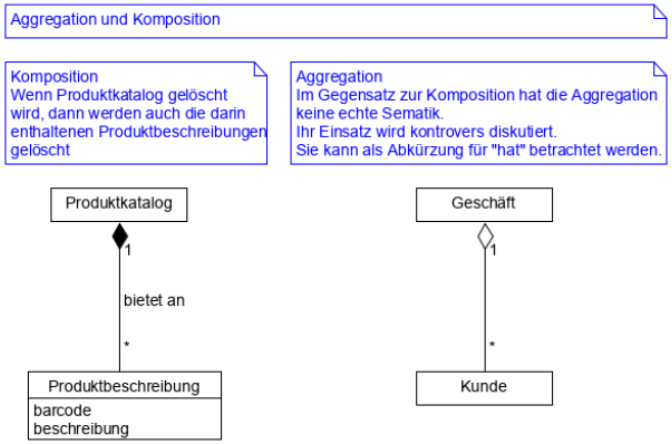
7. Wertobjekte

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

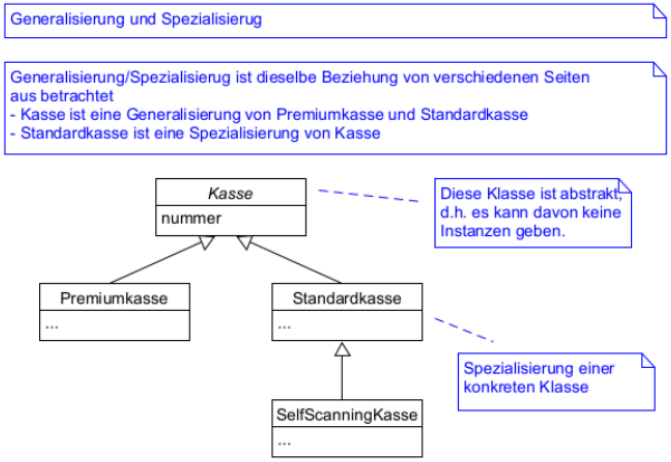
Mehr Info und Beispiele

Optionale Elemente im Domänenmodell

- Optional: Aggregationen/Kompositionen



- Optional: Generalisierung/Spezialisierung



Vollständige Beispiele Domänenmodell

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

Komplexes Domänenmodell: Reisebuchungssystem

Anforderung: Modellieren Sie ein System für Pauschalreisen mit Flügen, Hotels und Aktivitäten.

Verwendete Analysemuster:

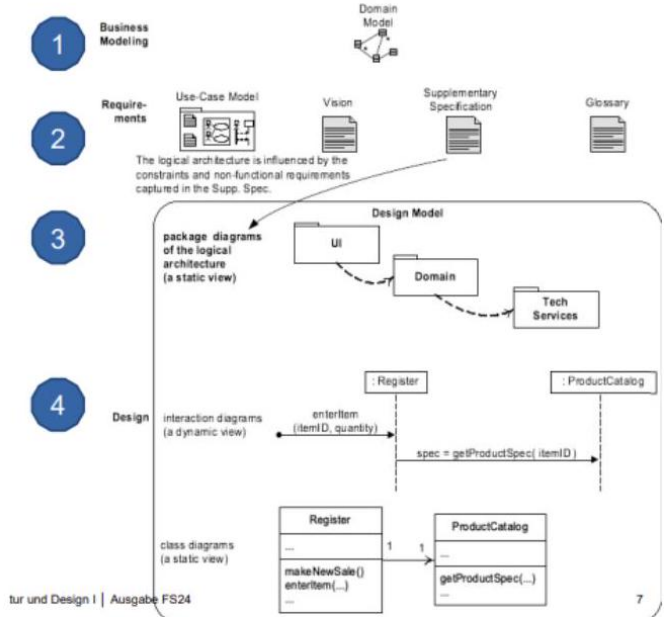
- **Beschreibungsklassen:**
  - Flugverbindung vs. konkreter Flug
  - Hotelkategorie vs. konkretes Zimmer
  - Aktivitätstyp vs. konkrete Durchführung
- **Zustände:**
  - Buchungszustände: angefragt, bestätigt, storniert
  - Zahlungszustände: offen, teilbezahlt, vollständig
- **Rollen:**
  - Person als: Kunde, Reiseleiter, Kontaktperson
- **Wertobjekte:**
  - Geldbetrag mit Währung
  - Zeitraum für Reisedauer

Softwarearchitektur und Design

Grundlagen der Software-Architektur

Grundlagen und Überblick

- **Business Analyse:**
  - Domänenmodell und Kontextdiagramm
  - Requirements (funktional und nicht-funktional)
  - Vision und Stakeholder
- **Architektur:**
  - Logische Struktur des Systems
  - Technische Konzeption
  - Qualitätsanforderungen
- **Entwicklung:**
  - Use Case / User Story Realisierung
  - Design-Klassendiagramm (DCD)
  - Implementierung und Tests



Softwarearchitektur

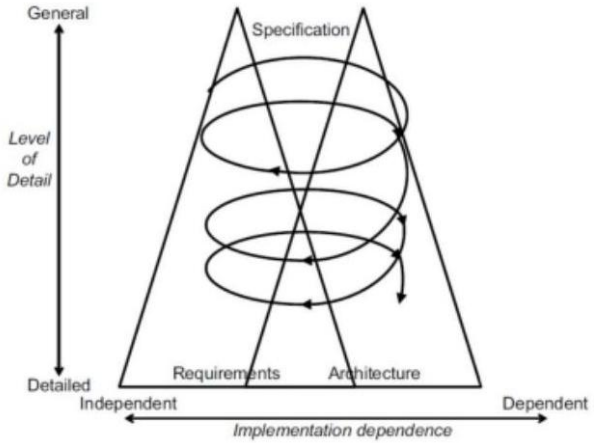
Die Architektur eines Softwaresystems definiert:

- **Grundlegende Entscheidungen:**
  - Programmiersprachen und Plattformen
  - Aufteilung in Teilsysteme und Komponenten
  - Schnittstellen zwischen Komponenten
- **Strukturelle Aspekte:**
  - Verantwortlichkeiten der Teilsysteme
  - Abhängigkeiten zwischen Komponenten
  - Einsatz von Basis-Technologien/Frameworks
- **Qualitätsaspekte:**
  - Erfüllung nicht-funktionaler Anforderungen
  - Maßnahmen für Performance, Skalierbarkeit etc.
  - Fehlertoleranz und Ausfallsicherheit

Architekturanalyse

erfolgt iterativ mit den Anforderungen (Twin Peaks Model):

- **Anforderungsanalyse:**
  - Analyse funktionaler und nicht-funktionaler Anforderungen
  - Prüfung der Qualität und Stabilität der Anforderungen
  - Identifikation von Lücken und impliziten Anforderungen
- **Architekturentscheidungen:**
  - Abstimmung mit Stakeholdern
  - Berücksichtigung von Randbedingungen
  - Vorausschauende Planung für zukünftige Änderungen



Qualitätsanforderungen

ISO 25010:

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

FURPS+:

- **F**unctionality (Funktionalität)
- **U**sability (Benutzerfreundlichkeit)
- **R**eliability (Zuverlässigkeit)
- **P**erformance (Leistung)
- **S**upportability (Wartbarkeit)
- +: Implementation, Interface, Operations, Packaging, Legal

Architekturprinzipien

Grundlegende Prinzipien für gute Architektur:

Separation of Concerns:

- Trennung von Verantwortlichkeiten
- Klare Modulgrenzen
- Reduzierte Komplexität

Information Hiding:

- Kapselung von Implementierungsdetails
- Definierte Schnittstellen
- Änderbarkeit ohne Seiteneffekte

Loose Coupling:

- Minimale Abhängigkeiten
- Austauschbarkeit
- Unabhängige Entwicklung

Qualitätskriterien und deren Umsetzung

Strategien zur Erfüllung von Qualitätsanforderungen:

Performance:

- Effiziente Ressourcennutzung (Resource Pooling, Caching)
- Optimierte Verarbeitung (Parallelisierung, Lazy Loading)

Skalierbarkeit:

- Dynamische Anpassung (horizontale/vertikale Skalierung)
- Effiziente Lastverteilung (Load Balancing, Partitionierung)

Wartbarkeit:

- Klare Strukturen (Separation of Concerns, Modularisierung)
- Verbesserte Codequalität (Information Hiding, Standardisierung)

Zuverlässigkeit:

- Fehlerresistenz (Redundanz, Fehlertoleranz)
- Prävention und Wiederherstellung (Monitoring, Backup/Recovery)

Verfügbarkeit:

- Ausfallschutz (Redundanz, Failover-Mechanismen)
- Überwachung/Stabilisierung (Health Monitoring, Circuit Breaker)

Modularität:

- Gut definierte Grenzen (klare Modulgrenzen, hohe Kohäsion)
- Minimale Abhängigkeiten zwischen Modulen

Testbarkeit:

- Einfachheit von Tests (Isolation, Mockbarkeit)
- Automatisierung und Skalierung von Tests

Änderbarkeit:

- Anpassungsfähigkeit (Lokalisierung, Erweiterbarkeit)
- Sicherstellung der Kompatibilität (Backward Compatibility)

Erweiterbarkeit:

- Flexible Architekturen (offene Schnittstellen, Plugin-Systeme)
- Serviceorientierung für modulare Erweiterungen

Architekturprozess und Best Practices

Best Practices im Architekturentwurf

1. Analyse und Planung

- Anforderungen priorisieren
- Qualitätsziele definieren
- Constraints identifizieren
- Stakeholder einbinden

2. Design-Prinzipien

- Separation of Concerns
- Single Responsibility
- Information Hiding
- Don't Repeat Yourself (DRY)

3. Strukturierung

- Klare Schichtenarchitektur
- Definierte Schnittstellen
- Lose Kopplung
- Hohe Kohäsion

4. Dokumentation

- Architekturentscheidungen
- Begründungen
- Alternativen
- Trade-offs

Architektur-Design

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

Schnittstellen

Module kommunizieren über definierte Schnittstellen:

- **Exportierte Schnittstellen:**
  - Definieren angebotene Funktionalität
  - Vertraglich garantierte Leistungen
  - Einzige nach außen sichtbare Information
- **Importierte Schnittstellen:**
  - Von anderen Modulen benötigte Funktionalität
  - Definieren Abhängigkeiten
  - Basis für Kopplung
  - Sollten minimiert werden (Low Coupling)

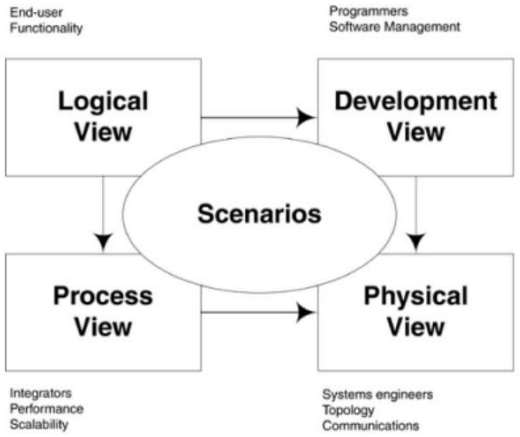
Architektursichten (4+1 View Model)

Verschiedene Perspektiven auf die Architektur:

- **Logical View:** End-User, Functionality
  - Funktionalität des Systems
  - Schichten, Subsysteme, Pakete, Klassen und Schnittstellen
- **Process View:** Integrators, Performance, Scalability
  - Laufzeitverhalten, Performance und Skalierung
  - Prozesse und Threads
- **Development View:** Programmers, Software Management
  - Build und Deployment
  - Implementierungsstruktur und Quellcode-Organisation
- **Physical View:** System Engineers, Topology, Communications
  - Hardware-Topologie und Verteilung der Software
  - Netzwerkkommunikation

+1: Scenarios:

- Wichtige Use Cases, Validierung der Architektur, Integration der anderen Views



Architektur-Muster und Clean Code

Architekturmuster

Grundlegende Architekturmuster für Software-Systeme:

Layered Pattern:

- Strukturierung in horizontale Schichten
- Klare Trennung der Verantwortlichkeiten
- Abhängigkeiten nur nach unten

Client-Server Pattern:

- Verteilung von Diensten
- Zentralisierte Ressourcen
- Mehrere Clients pro Server

Master-Slave Pattern:

- Verteilung von Aufgaben
- Zentrale Koordination
- Parallelverarbeitung

Pipe-Filter Pattern:

- Datenstromverarbeitung
- Verkettung von Operationen
- Wiederverwendbare Filter

Broker Pattern:

- Vermittlung zwischen Komponenten
- Entkopplung von Diensten
- Zentrale Koordination

Event-Bus Pattern:

- Asynchrone Kommunikation
- Publisher-Subscriber Modell
- Lose Kopplung

Event-Bus Pattern

Implementierung eines Event-Bus Systems:

1. Event-Bus

- Publisher-Subscriber Mechanismus implementieren
- Event-Routing einrichten
- Event-Persistenz berücksichtigen
- Ordering und Filtering ermöglichen

2. Publisher

- Event-Typen definieren
- Event-Publikation implementieren
- Transaktionshandling berücksichtigen
- Fehlerbehandlung vorsehen

3. Subscriber

- Event-Handler implementieren
- Idempotenz sicherstellen
- Fehlertoleranz einbauen
- Dead-Letter-Queue vorsehen

Layered Pattern

Anwendung des Schichtenmusters:

1. Schichten identifizieren

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

2. Regeln definieren

- Kommunikation nur mit angrenzenden Schichten
- Abhängigkeiten nur nach unten
- Jede Schicht kapselt ihre Implementierung

3. Schnittstellen festlegen

- Klare Service-Interfaces pro Schicht
- Dependency Injection für lose Kopplung
- Daten-DTOs für Schichtübergänge

Client-Server Pattern

Implementierung:

1. Server-Design

- Services definieren
- Schnittstellen dokumentieren
- Sicherheitsaspekte berücksichtigen
- Skalierbarkeit einplanen

2. Client-Design

- Client-Typen festlegen (Thin/Rich/Web)
- Fehlerbehandlung implementieren
- Offline-Fähigkeit berücksichtigen
- Caching-Strategie entwickeln

3. Kommunikation

- Protokoll wählen (REST, GraphQL, etc.)
- API-Versionierung einplanen
- Rate Limiting implementieren
- Monitoring einrichten

Master-Slave Pattern

Implementierung:

1. Master-Komponente

- Aufgabenverteilung implementieren
- Slave-Management einrichten
- Fehlertoleranz berücksichtigen
- Load Balancing implementieren

2. Slave-Komponenten

- Aufgabenbearbeitung implementieren
- Statusmeldungen einrichten
- Fehlerbehandlung implementieren
- Recovery-Mechanismen vorsehen

3. Koordination

- Kommunikationsprotokoll definieren
- Synchronisation implementieren
- Monitoring einrichten
- Failover-Strategien entwickeln

Pipe-Filter Pattern

Implementierung einer Pipe-Filter Architektur:

1. Filter-Design

- Atomare Transformationen definieren
- Ein- und Ausgabeformat festlegen
- Fehlerbehandlung implementieren
- Unabhängige Verarbeitung sicherstellen

2. Pipe-Design

- Datentransfer implementieren
- Pufferung einrichten
- Threading-Modell festlegen
- Backpressure berücksichtigen

3. Pipeline-Konfiguration

- Filter-Reihenfolge definieren
- Verzweigungen ermöglichen
- Monitoring einrichten
- Fehlerszenarien behandeln

Broker Pattern

Implementierung eines Broker Systems:

1. Broker-Komponente

- Service-Registry implementieren
- Request-Routing einrichten
- Load Balancing implementieren
- Service-Discovery ermöglichen

2. Service-Provider

- Service-Interface definieren
- Bei Broker registrieren
- Health Checks implementieren
- Fehlerbehandlung vorsehen

3. Service-Consumer

- Service-Lookup implementieren
- Fehlertoleranz einbauen
- Caching-Strategie entwickeln
- Retry-Mechanismen vorsehen

Clean Architecture

Clean Architecture

Architektur-Prinzipien nach Robert C. Martin:

Hauptprinzipien:

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

Schichten (von innen nach außen):

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

Clean Architecture

Implementierung der Clean Architecture:

1. Schichten definieren

- **Entities:**
  - Zentrale Geschäftsobjekte identifizieren
  - Geschäftsregeln definieren
  - Unabhängig von externen Frameworks
- **Use Cases:**
  - Anwendungsfälle implementieren
  - Geschäftslogik orchestrieren
  - Input/Output Boundaries definieren
- **Interface Adapters:**
  - Controller implementieren
  - Presenter erstellen
  - Gateways definieren
- **Frameworks & Drivers:**
  - UI-Framework einbinden
  - Datenbankzugriff implementieren
  - Externe Services anbinden

2. Dependency Rule beachten

- Abhängigkeiten nur nach innen
- Interfaces für Richtungsumkehr
- DTOs für Datentransfer

3. Clean Architecture Testing

- Unit Tests für Entities
- Use Case Tests ohne externe Systeme
- Integrationstests für Adapter
- End-to-End Tests für das Gesamtsystem

Gesamter Architekturprozess

Architekturprozess-Komponenten

Architekturanalyse:

- Erster Schritt im Architekturprozess
- Analyse der funktionalen und nicht-funktionalen Anforderungen
- Identifikation von Qualitätszielen
- Parallel zur Anforderungserhebung (Twin Peaks)

Architektur-Entscheidungen:

- Konkrete Beschlüsse basierend auf der Analyse
- Technologiewahl und Strukturierung
- Dokumentation und Begründung
- Einschließlich verworfener Alternativen

Architektur-Entwurf:

- Praktischer Gestaltungsprozess
- Anwendung von Architekturmustern
- Umsetzung von Qualitätsanforderungen
- Erstellung konkreter Artefakte

Architektur-Review:

- Systematische Überprüfung
- Meist durch externe Experten
- Prüfung der Anforderungserfüllung
- Identifikation von Schwachstellen

Architektur-Evaluation:

- Bewertung anhand definierter Kriterien
- Quantitative und qualitative Analyse
- Szenario-basierte Prüfung
- Bewertung von Qualitätsattributen

Gesamter Architekturprozess

1. Initiale Phase

- Architekturanalyse durchführen
- Grundlegende Entscheidungen treffen
- Ersten Entwurf erstellen

2. Iterative Verfeinerung

- Review durchführen
- Evaluation vornehmen
- Anpassungen basierend auf Feedback

3. Kontinuierliche Verbesserung

- Regelmäßige Reviews
- Neue Anforderungen einarbeiten
- Technische Schulden adressieren

4. Dokumentation

- Entscheidungen festhalten
- Architektur dokumentieren
- Änderungen nachverfolgen

5. Qualitätssicherung

- Architektur-Konformität prüfen
- Performance-Tests durchführen
- Sicherheitsaudits durchführen

Gesamter Architekturprozess

Architekturanalyse

1. Anforderungen sammeln

- Funktionale Anforderungen gruppieren
- Nicht-funktionale Anforderungen identifizieren
- Randbedingungen dokumentieren

2. Qualitätsziele definieren

- Messbare Kriterien festlegen
- Priorisierung vornehmen
- Trade-offs identifizieren

3. Einflussfaktoren analysieren

- Technische Faktoren
- Organisatorische Faktoren
- Wirtschaftliche Faktoren

Architekturanalyse

Architektur-Entscheidungen

1. Alternativen identifizieren

- Mögliche Lösungen sammeln
- Vor- und Nachteile analysieren
- Machbarkeit prüfen

2. Bewertungskriterien

- Erfüllung der Anforderungen
- Technische Umsetzbarkeit
- Kosten und Aufwand

3. Entscheidung dokumentieren

- Begründung
- Konsequenzen
- Verworfene Alternativen

Architektur-Entscheidungen dokumentieren

1. Entscheidung festhalten

- Dokumentation der getroffenen Architekturentscheidungen
- Begründungen und Alternativen
- Auswirkungen und Konsequenzen

2. Strukturierte Dokumentation

- Einheitliches Format für alle Entscheidungen
- Verwendung von Templates
- Nachvollziehbare Historie der Entscheidungen

3. Kommunikation

- Regelmäßige Updates an Stakeholder
- Transparenz über getroffene Entscheidungen
- Einbindung des gesamten Teams

4. Review und Anpassung

- Regelmäßige Überprüfung der Entscheidungen
- Anpassung bei geänderten Rahmenbedingungen
- Lessons Learned dokumentieren

Architektur-Entscheidungen

Architekturentwurf

Schritte:

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

Architekturentwurf

Architektur-Review durchführen

Vorgehen:

1. **Vorbereitung**
   - Architektur-Dokumentation zusammenstellen
   - Review-Team zusammenstellen
   - Checklisten vorbereiten
2. **Durchführung**
   - Architektur vorstellen
   - Anforderungen prüfen
   - Entscheidungen hinterfragen
   - Risiken identifizieren
3. **Nachbereitung**
   - Findings dokumentieren
   - Maßnahmen definieren
   - Follow-up planen

Prüfkriterien:

- Anforderungserfüllung
- Technische Machbarkeit
- Zukunftssicherheit
- Best Practices

Architektur-Review

Architektur-Evaluation

Systematische Bewertung einer Softwarearchitektur:

1. Qualitätsattribute identifizieren

- Performance
- Skalierbarkeit
- Wartbarkeit
- Sicherheit

2. Szenarien entwickeln

- Normale Nutzung
- Grenzfälle
- Fehlerfälle
- Wartungsszenarien

3. Architektur analysieren

- Strukturanalyse
- Verhaltensanalyse
- Trade-off Analyse

4. Risiken identifizieren

- Technische Risiken
- Geschäftsrisiken
- Architekturrisiken

Architektur-Evaluation

Beispiele Architekturentwurf

Typische Prüfungsaufgabe: Architekturanalyse und Entscheidungen

Aufgabenstellung: Analysieren Sie folgende Anforderungen und leiten Sie architektonische Konsequenzen ab:

- System muss 24/7 verfügbar sein
- 10.000 gleichzeitige Benutzer
- Reaktionszeit unter 1 Sekunde
- Jährliche Wartungsfenster maximal 4 Stunden

Lösung:

- **Architekturentscheidungen:**
  - Verteilte Architektur für Hochverfügbarkeit
  - Load Balancing für gleichzeitige Benutzer
  - Caching-Strategien für Performanz
  - Blue-Green Deployment für Wartung
- **Begründungen:**
  - Verteilung minimiert Single Points of Failure
  - Load Balancer verteilt Last gleichmäßig
  - Caching reduziert Datenbankzugriffe
  - Blue-Green erlaubt Updates ohne Downtime

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

Architektur-Evaluation: Performance

Szenario: Online-Shop während Black Friday

Analyse:

- **Last-Annahmen:**
  - 10.000 gleichzeitige Nutzer
  - 1.000 Bestellungen pro Minute
  - 100.000 Produktaufrufe pro Minute
- **Architektur-Maßnahmen:**
  - Caching-Strategie für Produkte
  - Load Balancing für Anfragen
  - Asynchrone Bestellverarbeitung
  - Datenbank-Replikation
- **Monitoring:**
  - Response-Zeiten
  - Server-Auslastung
  - Cache-Hit-Rate
  - Fehlerraten

UML-Modellierung

Grundlagen der UML-Modellierung

UML (Unified Modeling Language) wird im Design auf zwei Arten verwendet:

Statische Modelle:
- Struktur des Systems
- Klassendiagramme, Paketdiagramme
- Fokus auf Pakete, Klassen, Attribute
- Keine Methodenimplementierung

Dynamische Modelle:
- Verhalten des Systems
- Sequenz-, Zustands-, Aktivitätsdiagramme
- Fokus auf Logik und Verhalten
- Methodenimplementierung

UML Diagrammtypen

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

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

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

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

UML Diagrammauswahl

1. Statische Struktur
- Klassendiagramm für Typen und Beziehungen
- Paketdiagramm für Modularisierung
- Komponentendiagramm für Bausteinsicht
- Verteilungsdiagramm für physische Verteilung

2. Dynamisches Verhalten
- Sequenzdiagramm für zeitliche Abläufe
- Kommunikationsdiagramm für Objektkollaborationen
- Zustandsdiagramm für Objektlebenszyklen
- Aktivitätsdiagramm für Geschäftsprozesse

3. Verwendungszweck
- Analyse: Konzeptuelle Modellierung
- Design: Detaillierte Spezifikation
- Implementation: Code-nahe Darstellung
- Dokumentation: Architekturübersicht

Kommunikationsdiagramm

Hauptelemente:
- **Objekte:**
  - Als Rechtecke dargestellt
  - Mit Objektname und Klasse
  - Verbunden durch Links
- **Nachrichten:**
  - Nummerierte Sequenz
  - Synchrone/Asynchrone Aufrufe
  - Parameter und Rückgabewerte
- **Steuerungselemente:**
  - Bedingte Nachrichten [condition]
  - Iterationen *
  - Parallele Ausführung ||

Verteilungsdiagramm **Elemente:**

- **Nodes:**
  - Device Nodes
  - Execution Environment
  - Artifacts
- **Verbindungen:**
  - Kommunikationspfade
  - Protokolle
  - Multiplizitäten
- **Deployment:**
  - Deployment Specifications
  - Manifestationen

Best Practices für UML-Modellierung

1. Allgemeine Richtlinien
- Nur relevante Details zeigen
- Konsistente Notation verwenden
- Diagramme dokumentieren
- Lesbarkeit priorisieren

2. Diagrammspezifische Richtlinien
- Klassendiagramm: Wichtige Beziehungen hervorheben
- Sequenzdiagramm: Kritische Interaktionen zeigen
- Zustandsdiagramm: Komplexe Zustände hierarchisch strukturieren
- Aktivitätsdiagramm: Parallelität klar darstellen

3. Tooling
- UML-Tool auswählen
- Versionskontrolle einsetzen
- Templates definieren
- Review-Prozess etablieren

Objektorientiertes Design

GRASP Prinzipien

General Responsibility Assignment Software Patterns - Grundlegende Prinzipien für die Zuweisung von Verantwortlichkeiten:

Information Expert:
- Zuständigkeit basierend auf Information
- Klasse mit relevanten Daten übernimmt Aufgabe
- Fördert Kapselung und Kohäsion

Creator:
- Verantwortung für Objekterstellung
- Basierend auf Beziehungen (enthält, aggregiert)
- Starke Verwendungsbeziehung

Controller:
- Koordination von Systemoperationen
- Erste Anlaufstelle nach UI
- Fassade für Subsystem

Low Coupling:
- Minimale Abhängigkeiten
- Erhöht Wiederverwendbarkeit
- Erleichtert Änderungen

High Cohesion:
- Fokussierte Verantwortlichkeiten
- Zusammengehörige Funktionalität
- Wartbare Klassen

Polymorphism:
- Verhaltensänderungen durch Vererbung
- Type-dependent behavior
- Alternative zu if/else Ketten

Pure Fabrication:
- Hilfsklassen für besseres Design
- Keine direkte Domänenentsprechung
- Unterstützt High Cohesion

Indirection:
- Vermittler für lose Kopplung
- Intermediate object
- Reduziert direkte Abhängigkeiten

Protected Variations:
- Kapselung von Änderungen
- Interface für Variation Points
- Stabilität bei Änderungen

Grundlagen des RDD

Design basierend auf Verantwortlichkeiten und Kollaborationen:

Verantwortlichkeiten:

- **Doing:**
  - Aktionen ausführen
  - Berechnungen durchführen
  - Andere Objekte steuern
  - Aktivitäten koordinieren
- **Knowing:**
  - Eigene Daten kennen
  - Verwandte Objekte kennen
  - Berechnete Informationen
  - Private enkapsulierte Daten

Kollaborationen:

- Klare Rollen definieren
- Aufgaben verteilen
- Interfaces abstimmen
- Verantwortlichkeiten zuweisen

RDD Anwendung

1. **Verantwortlichkeiten identifizieren**
   - Systemoperationen analysieren
   - Notwendige Aktionen auflisten
   - Benötigte Daten identifizieren
   - Abhängigkeiten erkennen
2. **Rollen definieren**
   - Klassen nach Verantwortlichkeiten gruppieren
   - Schnittstellen festlegen
   - Kollaborationen planen
   - GRASP Prinzipien anwenden
3. **Implementierung**
   - Interfaces definieren
   - Klassen implementieren
   - Kollaborationen umsetzen
   - Tests schreiben

Best Practices im RDD

1. **Klare Verantwortlichkeiten**
   - Eine Hauptverantwortung pro Klasse
   - Logisch zusammenhängende Aufgaben
   - Überschaubare Klassengröße
2. **Effektive Kollaborationen**
   - Minimale Abhängigkeiten
   - Klare Schnittstellen
   - Wiederverwendbare Komponenten
3. **Wartbarkeit**
   - Testbare Komponenten
   - Dokumentierte Verantwortlichkeiten
   - Erweiterbare Struktur

Design Patterns in der Architektur

Design Patterns im Architekturkontext

Architektur-relevante Patterns:

- **Structural Patterns:**
  - Facade für Subsystem-Zugriff
  - Adapter für System-Integration
  - Proxy für Remote-Zugriff
  - Bridge für Implementierungs-Entkopplung
- **Behavioral Patterns:**
  - Observer für Event-Handling
  - Command für Service-Aufrufe
  - Strategy für austauschbare Algorithmen
  - Template Method für Framework-Hooks
- **Creational Patterns:**
  - Factory Method für Komponenten-Erstellung
  - Abstract Factory für Familien von Komponenten
  - Builder für komplexe Objektkonstruktion
  - Singleton für zentrale Dienste

Anti-Patterns

Häufige Anti-Patterns

1. **Big Ball of Mud**
   - Keine klare Struktur
   - Vermischung von Zuständigkeiten
   - Schwer wartbar und erweiterbar
2. **Golden Hammer**
   - Ein Pattern/Tool für alle Probleme
   - Ignorieren besserer Alternativen
   - Übermäßige Komplexität
3. **Spaghetti Code**
   - Unstrukturierter Code
   - Keine klaren Abhängigkeiten
   - Schwer zu verstehen und zu ändern
4. **God Class**
   - Zu viele Verantwortlichkeiten
   - Verletzt Single Responsibility
   - Schwer zu testen und zu warten
5. **Lava Flow**
   - Veralteter, ungenutzter Code
   - Niemand traut sich zu löschen
   - Erhöht Komplexität unnötig

Anti-Pattern Erkennung und Vermeidung

1. **Code-Review Checkliste**
   - Single Responsibility prüfen
   - Abhängigkeiten analysieren
   - Testbarkeit bewerten
   - Dokumentation prüfen
2. **Refactoring-Strategien**
   - Klassen aufteilen
   - Verantwortlichkeiten extrahieren
   - Interfaces einführen
   - Tests schreiben
3. **Präventive Maßnahmen**
   - Design Reviews durchführen
   - Architektur-Guidelines definieren
   - Code-Qualität messen
   - Kontinuierliches Refactoring

Use Case Realization

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

Use Case Realization Ziele

- Umsetzung der fachlichen Anforderungen in Code
- Einhaltung der Architekturvorgaben
- Implementierung der GRASP-Prinzipien
- Erstellung wartbaren und testbaren Codes
- Dokumentation der Design-Entscheidungen

Verantwortlichkeiten (Responsibilities)

Im objektorientierten Design unterscheiden wir zwei Arten von Verantwortlichkeiten:

Doing-Verantwortlichkeiten:

- Selbst etwas tun
- Aktionen anderer Objekte anstoßen
- Aktivitäten anderer Objekte kontrollieren

Knowing-Verantwortlichkeiten:

- Private eingekapselte Daten kennen
- Verwandte Objekte kennen
- Dinge berechnen/ableiten können

Architekturbezogene Aspekte

Bei der Use Case Realization müssen folgende architektonische Aspekte beachtet werden:

Schichtenarchitektur:

- Presentation Layer (UI)
- Application Layer (Use Cases)
- Domain Layer (Business Logic)
- Infrastructure Layer (Persistence, External Services)

Abhängigkeitsregeln:

- Abhängigkeiten nur nach unten
- Interfaces für externe Services
- Dependency Injection für lose Kopplung

Cross-Cutting Concerns:

- Logging
- Security
- Transaction Management
- Exception Handling

Vorgehen bei der Use Case Realization

Einfluss der Analyse-Artefakte

- **Use Cases:**
  - Standardszenario
  - Erweiterungen
  - Definiert Systemoperationen
- **Systemverträge:**
  - Definieren Vorbedingungen
  - Definieren Nachbedingungen
  - Legen Invarianten fest
- **Domänenmodell:**
  - Inspiriert Softwareklassen
  - Definiert Attribute
  - Zeigt Beziehungen auf

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 Dokumentation

1. Analysephase

- Use Case und Systemoperationen dokumentieren
- Domänenmodell-Ausschnitt zeigen
- Relevante Anforderungen auflisten

2. Design

- Design Class Diagram erstellen
- Sequenzdiagramme für komplexe Abläufe
- GRASP-Prinzipien begründen

3. Implementation

- Code-Struktur dokumentieren
- Wichtige Algorithmen erläutern
- Test-Strategie beschreiben

Design und Implementation

UML im Design-Prozess

UML dient als:

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

Arten der Modellierung:

- **Statische Modelle:** Klassenstruktur
- **Dynamische Modelle:** Verhalten

Design to Code

Aus Design-Klassen-Diagramm (DCD):

- Klassen und deren Attribute
- Methoden und deren Signaturen
- Beziehungen zwischen Klassen

Aus Interaktionsdiagrammen: (SSD):

- Methodenaufrufe
- Parameter
- Sequenz der Aufrufe

System Sequence Diagrams (SSD)

System Sequence Diagrams (SSD) erstellen

Interaction Diagrams in der Use Case Realization

Sequenzdiagramm für enterItem():

```
:Register -> :ProductCatalog: getDescription(itemId)
:ProductCatalog --> :Register: desc
:Register -> currentSale: makeLineItem(desc, quantity)
currentSale -> :SalesLineItem: create(desc, quantity)
currentSale -> lineItems: add(sl)
```

Begründung der Interaktionen:

- Register als Controller empfängt Systemoperation
- ProductCatalog als Information Expert für Produkte
- Sale als Creator für SalesLineItem
- Sale als Container verwaltet seine LineItems

Design Class Diagram (DCD)

Design Class Diagram (DCD) erstellen

1. Klassen identifizieren

- Aus Domänenmodell übernehmen
- Technische Klassen ergänzen
- Controller bestimmen

2. Attribute definieren

- Datentypen festlegen
- Sichtbarkeiten bestimmen
- Validierungen vorsehen

3. Methoden hinzufügen

- Systemoperationen verteilen
- GRASP-Prinzipien anwenden
- Signaturen definieren

4. Beziehungen modellieren

- Assoziationen aus Domänenmodell
- Navigierbarkeit festlegen
- Abhängigkeiten minimieren

Design Class Diagram

GRASP in der Use Case Realization

GRASP Patterns in der Realization

Die 5 wichtigsten Prinzipien:

- **Information Expert:**
  - Verantwortlichkeit dort, wo Information liegt
  - Basis für Methodenzuweisung
- **Creator:**
  - Regeln für Objekterzeugung
  - Container erzeugt Inhalt
- **Controller:**
  - Erste Systemanlaufstelle
  - Koordiniert Systemoperationen
- **Low Coupling:**
  - Minimale Abhängigkeiten
  - Entscheidungshilfe bei Alternativen
- **High Cohesion:**
  - Fokussierte Verantwortlichkeiten
  - Zusammengehörige Funktionalität

Testing und Refactoring

1. Funktionale Prüfung

- Use Case Szenarien durchspielen
- Randfälle testen
- Fehlersituationen prüfen

2. Strukturelle Prüfung

- Architekturkonformität
- GRASP-Prinzipien
- Clean Code Regeln

3. Qualitätsprüfung

- Testabdeckung
- Wartbarkeit
- Performance

Implementierung und Prüfung

Typische Prüfungsaufgaben

1. Use Case Realization dokumentieren

- System Sequence Diagram erstellen
- Operation Contracts definieren
- Design Class Diagram zeichnen
- GRASP Prinzipien begründen
- Sequenzdiagramm für wichtige Operationen

2. Implementation analysieren

- GRASP Verletzungen identifizieren
- Verbesserungen vorschlagen
- Alternative Designs diskutieren

3. Architektur evaluieren

- Schichtenarchitektur prüfen
- Kopplung analysieren
- Kohäsion bewerten

Vollständige Use Case Realization

Use Case: Bestellung aufgeben

1. Systemoperationen:

- createOrder()
- addItem(productId, quantity)
- removeItem(itemId)
- submitOrder()

2. Design-Entscheidungen:

- OrderController als Fassade
- Order aggregiert OrderItems
- OrderService für Geschäftslogik
- Repository für Persistenz

3. GRASP-Anwendung:

- Information Expert:
  - Order berechnet Gesamtsumme
  - OrderItem verwaltet Produktdaten
- Creator:
  - Order erstellt OrderItems
  - OrderService erstellt Orders
- Low Coupling:
  - Repository-Interface für Persistenz
  - Service-Interface für Geschäftslogik

4. Implementierung:

```java
public class OrderController {
    private OrderService orderService;
    private Order currentOrder;

    public void createOrder() {
        currentOrder = orderService.createOrder();
    }

    public void addItem(String productId, int
         quantity) {
        currentOrder.addItem(productId, quantity);
    }

    public void submitOrder() {
        orderService.submitOrder(currentOrder);
    }
}
```

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

Strukturelle Patterns

Adapter Pattern **Problem:** Inkompatible Schnittstellen integrieren

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

Lösung: Adapter-Klasse als Vermittler

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

Decorator Pattern **Problem:** Dynamische Erweiterung von Objekten

- Zusätzliche Verantwortlichkeiten
- Nur für einzelne Objekte

Lösung: Wrapper-Objekt mit gleichem Interface

Composite Pattern **Problem:** Baumstrukturen verwalten

- Einheitliche Behandlung
- Teil-Ganzes Hierarchie

Lösung: Gemeinsames Interface für Container und Inhalt

Verhaltensmuster

Chain of Responsibility Pattern **Problem:** Unklare Zuständigkeit für Anfragen

- Mehrere mögliche Handler
- Zuständigkeit erst zur Laufzeit klar

Lösung: Verkettete Handler-Objekte

Observer Pattern **Problem:** Abhängige Objekte aktualisieren

- Lose Kopplung erwünscht
- Typ des Empfängers unbekannt

Lösung: Observer-Interface für Benachrichtigungen

Strategy Pattern **Problem:** Austauschbare Algorithmen

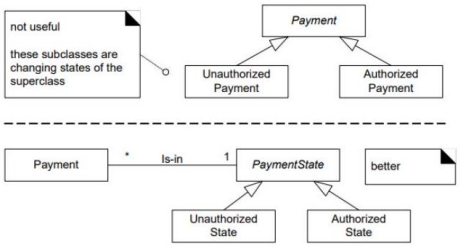
- Verschiedene Implementierungen
- Zur Laufzeit wechselbar

Lösung: Interface für Algorithmus-Klassen

State Pattern **Problem:** Zustandsabhängiges Verhalten

- Verhalten abhängig vom inneren Zustand
- Viele bedingte Anweisungen

Lösung: Eigene Klassen für verschiedene Zustände



Erzeugungsmuster

Factory Method Pattern **Problem:** Flexible Objekterzeugung

- Entscheidung über konkrete Klasse zur Laufzeit
- Basis für Frameworks/Libraries

Lösung: Abstrakte Methode zur Objekterzeugung

Singleton Pattern **Problem:** Genau eine Instanz benötigt

- Globaler Zugriffspunkt
- Mehrfachinstanzierung verhindern

Lösung: Statische Instanz mit privater Erzeugung

Pattern-Analyse für Prüfung **Systematisches Vorgehen:**

1. **Problem identifizieren und analysieren**
   - Art des Problems identifizieren
   - Anforderungen klar definieren
   - Kontext verstehen
2. **Pattern auswählen und evaluieren**
   - Passende Patterns suchen
   - Trade-offs abwägen
   - Kombinationsmöglichkeiten prüfen
3. **Lösung skizzieren**
   - Klassenstruktur entwerfen
   - Beziehungen definieren
   - Vor- und Nachteile nennen

Implementation, Refactoring und Testing

Design to Code

Umsetzungsstrategien **Code-Driven Development:**
- 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

Implementation Grundsätze **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

Refactoring

Refactoring Grundlagen Strukturierte Verbesserung des Codes ohne Änderung des externen Verhaltens:
- Kleine, kontrollierte Schritte
- Erhaltung der Funktionalität
- Verbesserung der Codequalität und interner Struktur
- Ziel: Bessere Wartbarkeit und Erweiterbarkeit

Code Smells Anzeichen für mögliche Probleme im Code:
- Duplizierter Code
- Lange Methoden
- Klassen mit vielen Instanzvariablen
- Klassen mit sehr viel Code
- Auffällig ähnliche Unterklassen
- Keine Interfaces
- Hohe Kopplung zwischen Klassen

Refactoring Patterns **1. Extract Method**
- Code in eigene Methode auslagern
- Verbessert Lesbarkeit und Wiederverwendbarkeit
- Reduziert Duplikation

2. Move Method/Field
- Methode/Feld in andere Klasse verschieben
- Verbessert Kohäsion
- Folgt Information Expert

3. Extract Class
- Teil einer Klasse in neue Klasse auslagern
- Trennt Verantwortlichkeiten
- Erhöht Kohäsion

4. Rename Method/Class/Variable
- Bessere Namen für besseres Verständnis
- Dokumentiert Zweck
- Erleichtert Wartung

Testing

Teststufen
- **Unit Tests:**
  - Einzelne Komponenten
  - Isolation durch Mocks/Stubs
  - Schnelle Ausführung
- **Integrationstests:**
  - Zusammenspiel mehrerer Komponenten
  - Schnittstellen-Tests
  - Externe Systeme
- **Systemtests:**
  - End-to-End Szenarien
  - Nicht-funktionale Anforderungen
- **Abnahmetests:**
  - Gegen Kundenanforderungen
  - User Acceptance Testing (UAT)

Testdesign **1. Funktionaler Test (Black-Box)**
- Test aus Benutzersicht
- Ohne Codekenntnis
- Fokus auf Input/Output

2. Strukturbezogener Test (White-Box)
- Test mit Codekenntnis
- Code Coverage
- Pfadtests

3. Änderungsbezogener Test
- Regressionstest
- Verifizierung von Änderungen
- Sicherstellung der Gesamtfunktionalität

Testbegriffe
- **Testling/Testobjekt:** Das zu testende Element
- **Fehler:** Fehler des Entwicklers bei der Implementation
- **Fehlerwirkung/Bug:** Abweichung vom spezifizierten Verhalten
- **Testfall:** Spezifische Testkonfiguration mit Testdaten
- **Testtreiber:** Programm zur Testausführung

Verteilte Systeme

Verteiltes System Ein Netzwerk aus autonomen Computern und Softwarekomponenten:

- Unabhängige Knoten und Komponenten
- Netzwerkverbindung
- Erscheint dem Benutzer wie ein einzelnes, kohärentes 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

Architekturmodelle

Architekturmodelle Heute finden vor allem folgende Architekturmodelle ihren Einsatz:

1. Client/Server

- Kurzlebiger Client-Prozess kommuniziert mit langlebigem Server-Prozess
- Beispiel: Web-Applikation

2. Peer-to-Peer

- Gleichberechtigte Peer-Prozesse
- Informationsaustausch nur bei Bedarf
- Beispiel: Blockchain

3. Event Systems (Publish-Subscribe)

- Event-Sources-Prozesse und Event-Sinks-Prozesse
- Asynchroner Informationsaustausch
- Beispiel: E-Mail-System

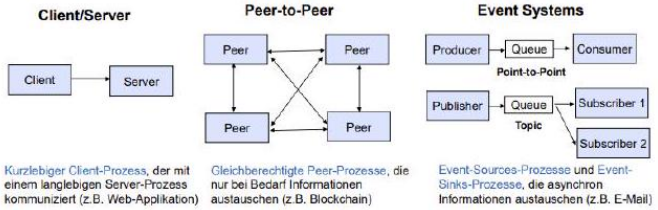


Abbildung 20: Architekturmodelle

Kommunikation und Middleware

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

Kommunikationsmodelle **1. Synchrone Kommunikation**

- Synchroner entfernter Dienstaufruf → blockierend
- Sender wartet auf Ergebnis
- Typisch für Request-Response Pattern

2. Asynchrone Kommunikation

- Asynchroner entfernter Serviceaufruf → nicht blockierend
- Sender kann direkt weitermachen
- Senden und Empfangen zeitlich versetzt

Middleware Middleware ist eine Softwareschicht, die standardisierte Dienste über ein API bereitstellt:

Middleware-Kategorien:

- **Anwendungsorientiert:**
  - Java Enterprise Edition (Jakarta EE)
  - Spring-Framework
- **Kommunikationsorientiert:**
  - RPC, RMI, REST, WebSocket
- **Nachrichtenorientiert:**
  - Message Oriented Middleware (MOM)
  - Java Messaging Service (JMS)

Design und Implementation

Entwurf verteilter Systeme **1. Systemanalyse**

- Anforderungen identifizieren
- Verteilungsaspekte analysieren
- Konsistenzanforderungen definieren

2. Architekturentscheidungen

- Architekturstil wählen
- Kommunikationsmuster festlegen
- Fehlertoleranzstrategie definieren

3. Technische Maßnahmen

- Middleware evaluieren
- Protokolle bestimmen
- Werkzeuge auswählen

Verteilungsprobleme analysieren **1. Probleme identifizieren**

- **Netzwerk:** Latenz, Bandbreite, Ausfälle
- **Daten:** Konsistenz, Replikation
- **System:** Skalierung, Verfügbarkeit

2. Lösungsstrategien

- **Netzwerk:** Caching, Compression
- **Daten:** Eventual Consistency, Master-Slave Replikation
- **System:** Load Balancing, Service Discovery

Common Pitfalls und Best Practices

Common Pitfalls in JPA Implementation **N+1 Problem:**
- **Symptom:** Für jedes Objekt wird eine zusätzliche Query ausgeführt
- **Lösung:** Join Fetch oder Eager Loading strategisch einsetzen

LazyInitializationException:
- **Symptom:** Zugriff auf lazy geladene Referenz außerhalb der Session
- **Lösung:** Transaktionen korrekt abgrenzen

Bidirektionale Beziehungen:
- **Symptom:** Inkonsistente Objektzustände
- **Lösung:** Helper-Methoden für Beziehungspflege

Best Practices für Persistenz **1. Architektur-Ebene**
- Repository für Datenzugriff
- Service für Geschäftslogik
- DTO für Datentransfer

2. Entity Design
- Immutable wenn möglich
- Bean Validation nutzen
- Geschäftsregeln in Entity-Klassen

3. Performance
- Caching Strategien
- Batch Processing
- Query Optimierung

Parent-Child Beziehungen

Parent-Child Mapping **Implementationsaspekte:**
- Cascade-Typen definieren
- Bidirektionale Navigation
- Lazy Loading konfigurieren
- Orphan Removal festlegen

JPA Annotationen:
- @OneToMany / @ManyToOne
- @JoinColumn
- mappedBy Parameter
- fetch = FetchType.LAZY/EAGER

Repository Pattern

Spring Data Repository **Vorteile:**
- Standardisierte CRUD-Operationen
- Query-Methoden aus Methodennamen
- Paginierung und Sortierung
- Einfache Integration mit Spring

Repository Hierarchie:
- Repository (Marker Interface)
- CrudRepository (Basis CRUD)
- PagingAndSortingRepository
- JpaRepository (JPA-spezifisch)

Repository Design **1. Interface Definition**
- Domänenspezifische Methoden
- Query-Methoden
- Custom Implementations

2. Query Methoden
- Methodennamen-Konventionen
- @Query Annotation
- Native Queries

3. Transaktionshandling
- @Transactional Annotation
- Isolation Level
- Propagation Rules

Performance Optimierung

Optimierungsstrategien **1. Fetch-Strategien**
- Lazy Loading als Default
- Joins für häufig benötigte Daten
- EntityGraphs für komplexe Szenarien

2. Caching
- First-Level Cache (Session)
- Second-Level Cache
- Query Cache

3. Batch-Verarbeitung
- Batch Inserts/Updates
- JDBC Batch Size
- Pagination für große Datensätze

Transaktionsmanagement **ACID-Eigenschaften:**
- Atomicity (Atomarität)
- Consistency (Konsistenz)
- Isolation (Isolation)
- Durability (Dauerhaftigkeit)

Isolation Levels:
- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

Framework Design

Framework Grundlagen Ein Framework ist ein Programmiergerüst mit folgenden Eigenschaften:

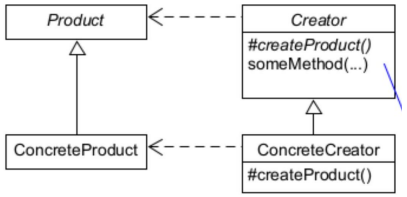
- Bietet wiederverwendbare Funktionalität
- Definiert Erweiterungs- und Anpassungspunkte
- Verwendet Design Patterns
- Enthält keinen applikationsspezifischen Code
- Gibt Rahmen für anwendungsspezifischen Code vor
- Klassen arbeiten eng zusammen (vs. reine Bibliothek)

Design Patterns in Frameworks

Factory Method **Problem:** Flexible Objekterzeugung in wiederverwendbarer Klasse

Lösung:

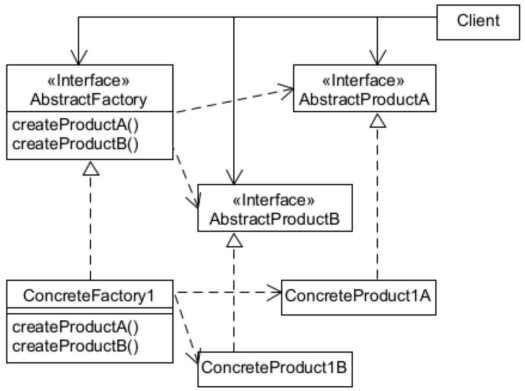
- Abstrakte Factory-Methode in Creator-Klasse
- Konkrete Subklassen überschreiben Methode
- Parallele Vererbungshierarchien



Abstract Factory **Problem:** Erzeugung zusammengehörender Objekte ohne Kenntnis konkreter Klassen

Lösung:

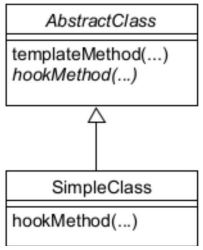
- AbstractFactory-Interface definieren
- Pro Produkt eine create-Methode
- Konkrete Factories implementieren Interface



Template Method **Problem:** Algorithmus mit anpassbaren Teilschritten

Lösung:

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



Moderne Framework Mechanismen

Annotation-basierte Konfiguration Moderne Frameworks nutzen Annotationen für:

- Dependency Injection
- Konfiguration
- Interface-Implementation
- Funktionalitätserweiterung

Vorteile:

- Keine harte Abhängigkeit zum Framework
- Deklarativer Programmierstil
- Reduzierung von Boilerplate-Code

Annotations als Steuerungsmechanismus **Auswertung von Annotationen:**

- Framework wird mit Anwendung gestartet
- Sucht Anwendungsklassen auf dem Klassenpfad
- Untersucht Annotationen

Mögliche Framework-Aktionen:

- Dependency Injection in Anwendungsobjekte
- Automatische Interface-Implementierung
- Funktionalität zu Klassen hinzufügen

Aspekt-orientierte Programmierung **Querschnittliche Belange:**

- Logging
- Sicherheit
- Transaktionsmanagement
- Performance Monitoring

Implementation mit Annotations:

- Aspekte definieren
- Join Points festlegen
- Advice implementieren

Framework Design Principles **1. Abstraktionsebenen definieren**

- Core API
- Extensions
- Standard-Implementierungen

2. Erweiterungsmechanismen

- Interface-basiert
- Annotations
- Composition

3. Qualitätskriterien

- Usability der API
- Flexibilität
- Wartbarkeit

Framework-Extensions entwickeln **1. Extension Points identifizieren**

- Core-Funktionalität analysieren
- Variationspunkte bestimmen
- Interface-Hierarchie planen

2. Extension Mechanismen

- Interface-basiert
- Annotation-basiert
- Discovery Mechanism implementieren

Framework Integration und Testing

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