Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Einführung und Überblick

Software Engineering Software Engineering ist eine systematische und strukturierte Entwicklung von Software:

Kernprozesse

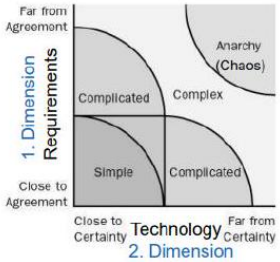
- Anforderungserhebung
- Systemdesign/technische Konzeption
- Implementierung
- Softwaretest
- Softwareeinführung
- Wartung/Pflege

Unterstützungsprozesse

- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

Dimensionen der Softwareentwicklung

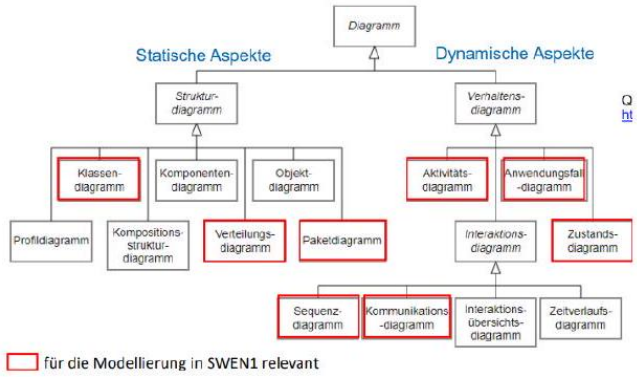
- Requirements (Bekannt - Unbekannt)
- Technology (Bekannt - Unbekannt)
- Skills/Experience (Vorhanden - Nicht vorhanden)



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

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

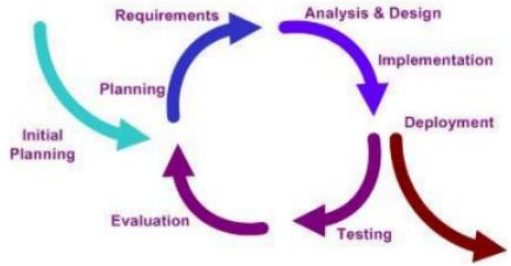
- Codierung und Korrektur im Wechsel
- Vorteile:
  - Schnell und agil
  - Einfach am Anfang
- Nachteile:
  - Schlecht planbar
  - Schwer wartbar
  - Änderungen aufwändig

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

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

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

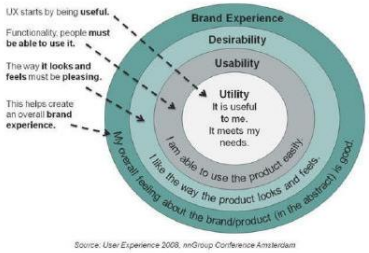
- **Disziplinen:** Anforderungen, Architektur, Implementierung, Test und Wartung
- **Ziel:** Strukturierte Prozesse für Qualität, Risiko- & Fehlerminimierung

Usability und User Experience

Usability und User Experience

Drei Säulen der Benutzererfahrung:

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



Source: User Experience 2008, nnGroup Conference Amsterdam

Usability-Dimensionen nach ISO 9241

- **Effektivität:** Vollständige und genaue Zielerreichung
- **Effizienz:** Minimaler Aufwand für die Zielerreichung
- **Zufriedenheit:** Positive Nutzererfahrung

ISO 9241-110: Usability-Anforderungen

- **Aufgabenangemessenheit:** Unterstützung der Arbeitsaufgaben
- **Selbstbeschreibungsfähigkeit:** Verständliche Benutzerführung
- **Steuerbarkeit:** Kontrolle über Ablauf
- **Erwartungskonformität:** Konsistentes Verhalten
- **Fehlertoleranz:** Fehlervermeidung und -korrektur
- **Individualisierbarkeit:** Anpassung an Benutzergruppen
- **Lernförderlichkeit:** Unterstützung beim Lernen

Usability-Evaluation durchführen

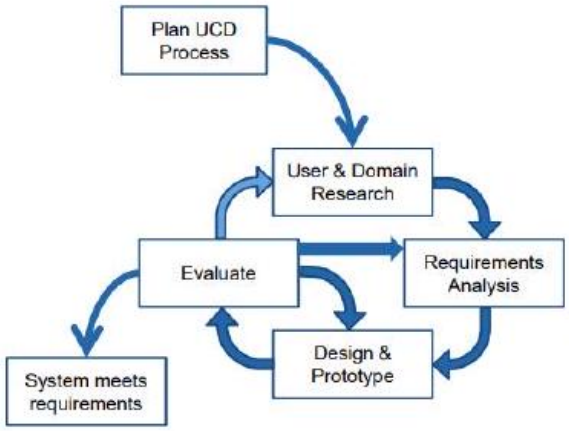
1. **Vorbereitung**
   - Testziele definieren
   - Testpersonen auswählen
   - Testaufgaben erstellen
2. **Durchführung**
   - Beobachtung der Nutzer
   - Protokollierung von Problemen
   - Zeitmessung der Aufgaben
3. **Auswertung**
   - Probleme klassifizieren
   - Schweregrad bestimmen
   - Verbesserungen vorschlagen

User-Centered Design (UCD)

UCD Process

Ein iterativer Prozess zur nutzerzentrierten Entwicklung:

- User & Domain Research
- Requirements Analysis
- Design & Prototype
- Evaluate



User & Domain Research

1. **Zielgruppe identifizieren**
   - Wer sind die Benutzer?
   - Aufgaben/Ziele verstehen
   - Arbeitsumgebung analysieren
2. **Daten sammeln**
   - Contextual Inquiry
   - Interviews/Beobachtungen
   - Fokusgruppen
3. **Ergebnisse dokumentieren**
   - Personas erstellen
   - Usage-Szenarien beschreiben
   - Mentales Modell entwickeln

Requirements Engineering

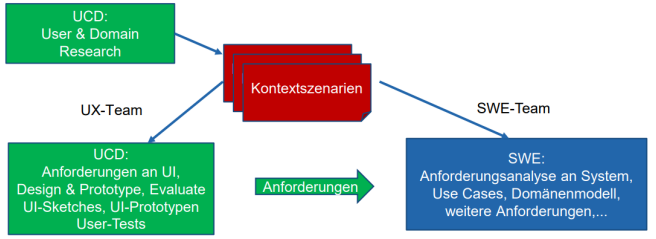
Requirements (Anforderungen)

Kern-Eigenschaften von Anforderungen:

- Explizit oder implizit
- Fast nie vollständig zu Beginn bekannt
- Mit allen Stakeholdern zu erarbeiten
- 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
- Compatibility
- Usability
- Reliability
- Security
- Maintainability
- Portability

Randbedingungen:

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

Use Cases

Use Case (Anwendungsfall)

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

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 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

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

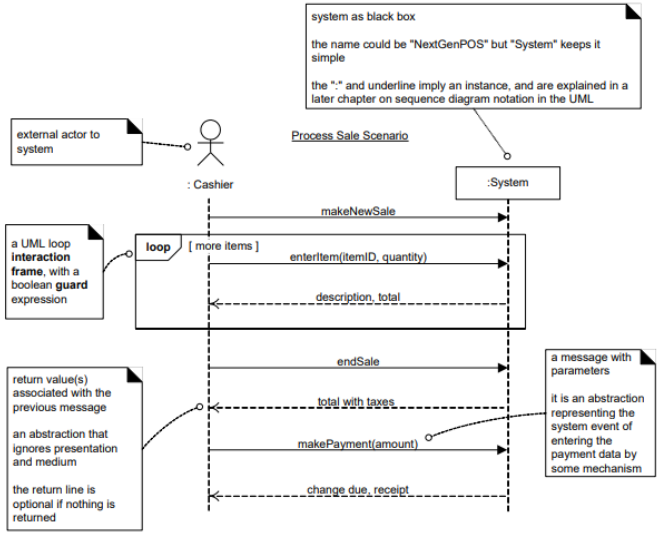
Fully-dressed Use Case erstellen

- **Grundinformationen**
  - Name (aktiv)
  - Umfang (Scope)
  - Ebene (Level)
  - Primärakteur
- **Stakeholder und Interessen**
  - Alle beteiligten Parteien
  - Deren spezifische Interessen
- **Vor- und Nachbedingungen**
  - Was muss vorher erfüllt sein?
  - Was ist nachher garantiert?
- **Standardablauf**
  - Nummerierte Schritte
  - Akteur-System-Interaktion
  - Klare Erfolgskriterien
- **Erweiterungen**
  - Alternative Abläufe
  - Fehlerszenarien
  - Verzweigungen

System Sequence Diagrams

System Sequence Diagrams (SSD)

- Formalisierte Darstellung der System-Interaktionen
- Identifiziert Systemoperationen
- Basis für API-Design
- Abstrahiert von UI-Details



Contracts für Systemoperationen

1. Struktur
- Name und Parameter
- Querverweis zum Use Case
- Vorbedingungen
- Nachbedingungen

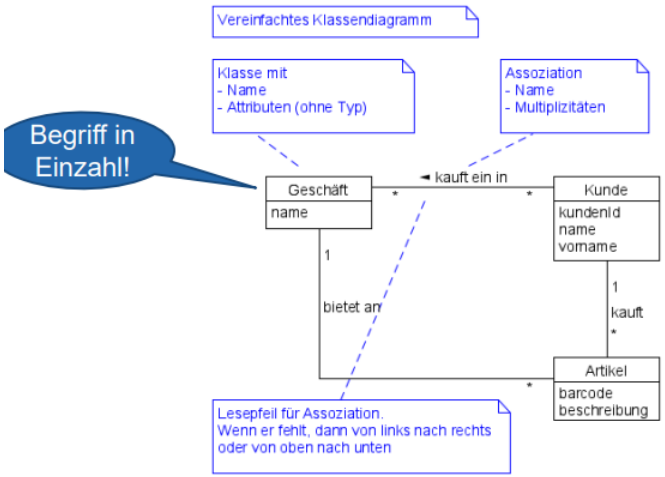
2. Vorbedingungen
- Systemzustand vor Aufruf
- Notwendige Initialisierungen
- Gültige Parameter

3. Nachbedingungen
- Erstellte/gelöschte Instanzen
- Geänderte Attribute
- Neue/gelöschte Assoziationen

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 **1. Konzepte identifizieren**

- **Kategorien prüfen:**
  - Physische Objekte (Produkte, Geräte)
  - Kataloge und Listen
  - Container (Warenkorb, Lager)
  - Externe Systeme
  - Rollen von Personen
  - Artefakte (Dokumente, Verträge)
  - Zahlungsinstrumente
- **Wichtig:** Keine Softwareklassen modellieren!

2. Attribute definieren

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

3. Beziehungen modellieren

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

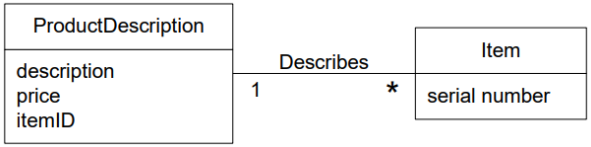
Analysemuster

Analysemuster im Überblick 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

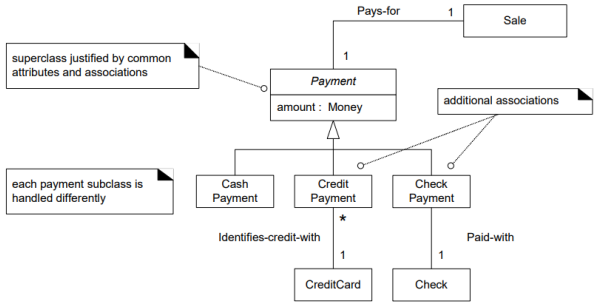
Beschreibungsklassen Trennt die Beschreibung eines Typs von seinen konkreten Instanzen:

- Verwendung bei mehreren gleichartigen Objekten
- Gemeinsame unveränderliche Eigenschaften
- Vermeidung von Redundanz



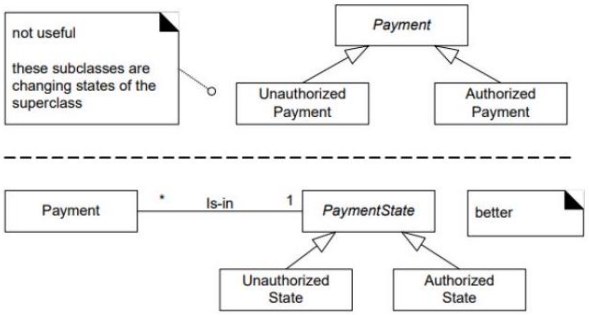
Generalisierung/Spezialisierung **Regeln:**

- 100
- 'IS-A' Beziehung
- Gemeinsame Attribute/Assoziationen in Basisklasse
- Spezifische Eigenschaften in Unterklassen



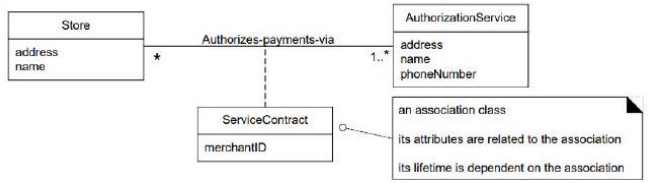
Zustände

- Modellierung als eigene Konzepthierarchie
- Vermeidung problematischer Vererbung
- Erweiterbarkeit durch neue Zustandsklassen



Assoziationsklassen **Einsatz wenn:**

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



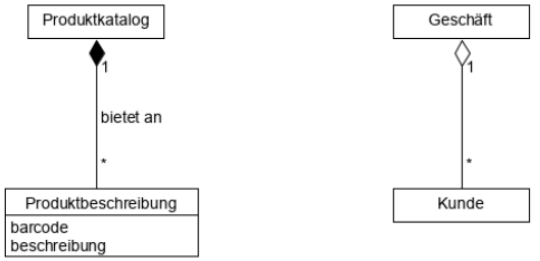
Optionale Elemente im Domänenmodell

- Optional: Aggregationen/Kompositionen

Aggregation und Komposition

Komposition
Wenn Produktkatalog gelöscht wird, dann werden auch die darin enthaltenen Produktbeschreibungen gelöscht

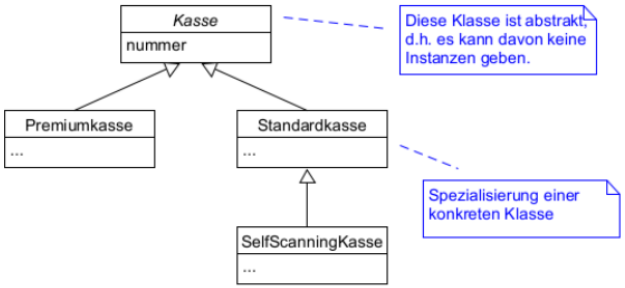
Aggregation
Im Gegensatz zur Komposition hat die Aggregation keine echte Sematik.
Ihr Einsatz wird kontrovers diskutiert.
Sie kann als Abkürzung für "hat" betrachtet werden.



- Optional: Generalisierung/Spezialisierung

Generalisierung und Spezialisierug

Generalisierung/Spezialisierug ist dieselbe Beziehung von verschiedenen Seiten aus betrachtet
- Kasse ist eine Generalisierung von Premiumkasse und Standardkasse
- Standardkasse ist eine Spezialisierung von Kasse

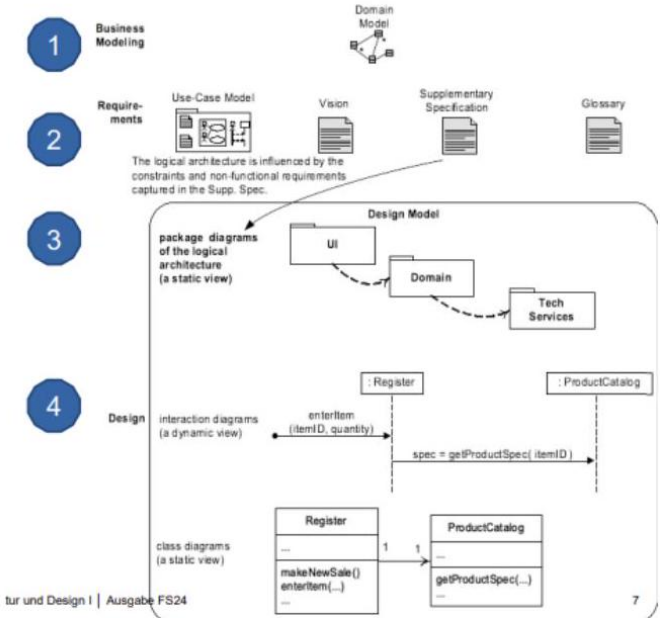


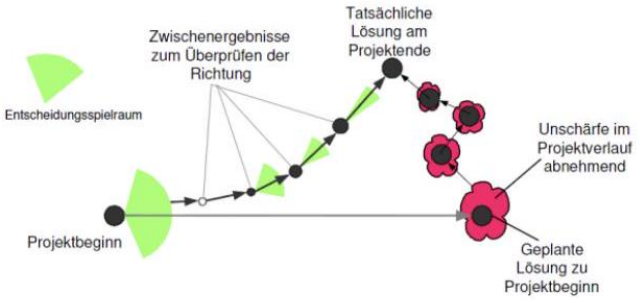
Softwarearchitektur und Design

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





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

Architektur-Design

Modulkonzept Ein Modul 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

Architektur-Prozess

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

Architektur-Evaluation **1. Qualitätsattribute identifizieren**

- Performance
- Skalierbarkeit
- Wartbarkeit
- Sicherheit

2. Szenarien entwickeln

- Normale Nutzung
- Grenzfälle
- Fehlerfälle
- Wartungsszenarien

3. Architektur analysieren

- Strukturanalyse
- Verhaltensanalyse
- Trade-off Analyse

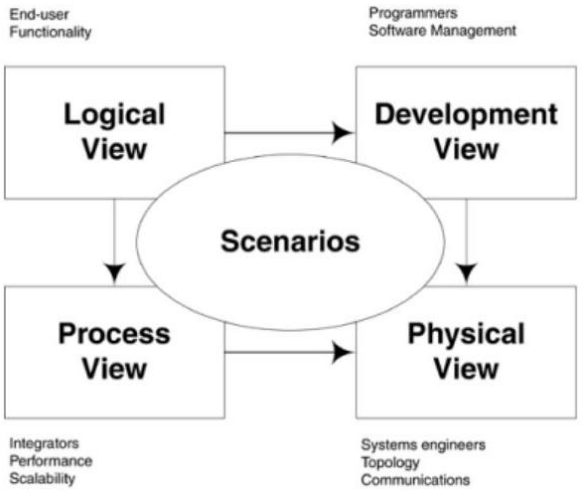
Architektur-Muster und Clean Code

Architekturmuster Übersicht 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
- **Event-Bus Pattern:**
  - Asynchrone Kommunikation
  - Publisher-Subscriber Modell
  - Lose Kopplung

Clean Architecture **Hauptprinzipien:**
- Unabhängigkeit von Frameworks
- Unabhängigkeit von UI
- Unabhängigkeit von Datenbank
- Testbarkeit ohne externe Systeme

Schichten (von innen nach außen):
- **Entities:** Zentrale Business Rules
- **Use Cases:** Anwendungsspezifische Business Rules
- **Interface Adapters:** Datenkonvertierung
- **Frameworks & Drivers:** Externe Tools



UML-Modellierung

UML im Design UML wird in zwei Hauptaspekten 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 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

Objektorientiertes Design

GRASP Prinzipien General Responsibility Assignment Software Patterns:
- **Information Expert:**
  - Zuständigkeit basierend auf Information
  - Klasse mit relevanten Daten übernimmt Aufgabe
- **Creator:**
  - Verantwortung für Objekterstellung
  - Basierend auf Beziehungen
- **Controller:**
  - Koordination von Systemoperationen
  - Erste Anlaufstelle nach UI
- **Low Coupling:**
  - Minimale Abhängigkeiten
  - Erhöht Wiederverwendbarkeit
- **High Cohesion:**
  - Fokussierte Verantwortlichkeiten
  - Zusammengehörige Funktionalität

Responsibility Driven Design (RDD) **1. Verantwortlichkeiten**
- **Doing:**
  - Selbst etwas tun
  - Aktionen anderer Objekte anstoßen
  - Aktivitäten koordinieren
- **Knowing:**
  - Private eingekapselte Daten
  - Verwandte Objekte kennen
  - Berechnete Informationen

2. Kollaborationen
- Klare Rollen definieren
- Aufgaben verteilen
- Interfaces abstimmen
- Verantwortlichkeiten zuweisen

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

Analyse-Artefakte

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
- Code/Dokumentation analysieren
- DCD überprüfen/aktualisieren

2. Realisierung
- Controller Klasse bestimmen
- Zu verändernde Klassen festlegen
- Weg zu diesen Klassen festlegen
- Veränderungen implementieren
- Review durchführen

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:
- Methodenaufrufe
- Parameter
- Sequenz der Aufrufe

GRASP Anwendung

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

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

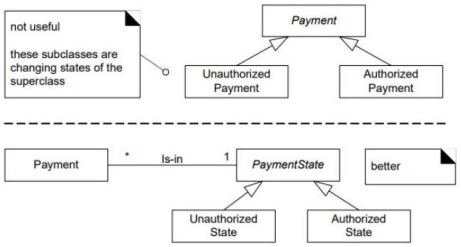
- Verschiedene Implementierungen
- Zur Laufzeit wechselbar

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

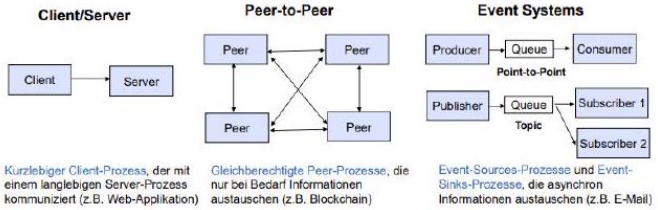


Abbildung 20: Architekturmodelle

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:

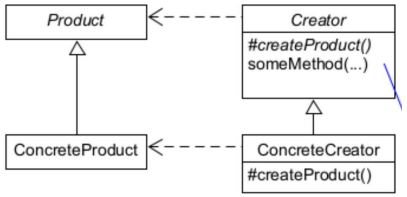
- Bietet wiederverwendbare Funktionalität
- Definiert Erweiterungs- und Anpassungspunkte
- Verwendet Design Patterns
- Enthält keinen applikationsspezifischen Code
- Gibt Rahmen für anwendungsspezifischen Code vor
- Klassen arbeiten eng zusammen (vs. reine Bibliothek)

Design Patterns in Frameworks

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

Lösung:

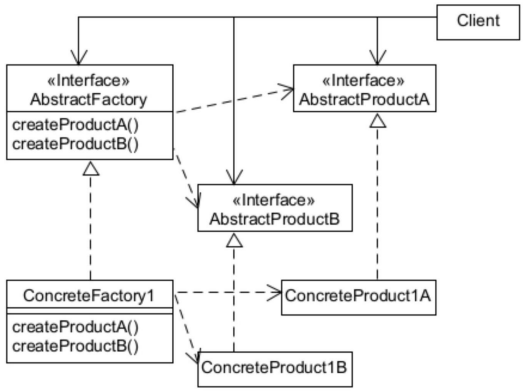
- Abstrakte Factory-Methode in Creator-Klasse
- Konkrete Subklassen überschreiben Methode
- Parallele Vererbungshierarchien



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

Lösung:

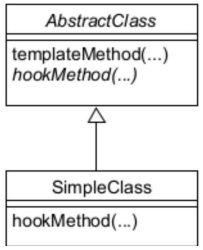
- AbstractFactory-Interface definieren
- Pro Produkt eine create-Methode
- Konkrete Factories implementieren Interface



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