Software Entwicklung 1

Asha Schwegler

16. Juni 2022

Inhaltsverzeichnis

1 Software Engineering

- Herstellung / Entwicklung von Software
- Organisation und Modellierung (Zugehörigen Datenstrukturen, Bettrieb von Softwaresystemen)
- Strukturiertes Projektplan f. Entwicklung
- Unterteilung Entwicklungsprozess
  - Schritte (überschaubar, zeitlich und inhaltlich begrenzt)
  - Phasen
  - Meilensteine
- Disziplinen während Entwicklungsprozess sind verzahnt.

Disziplinen

- **Kerndisziplinen**
  - Anforderungsanalyse
  - Softwarearchitekur und Design
  - Implementierung / Test
  - Softwareverteilung
  - Softwareeinführung
  - Wartung / Pflege
- **Unterstützungsdisziplinen**
  - Projektmanagement
  - Konfigurationsmanagemesnt
  - Risikomanagement

2 Prozess und Prozess-Modell

Warum strukturierte Softwareentwicklung notwendig?

- Strukturierung der wichtigsten Aktivitäten
- Früherkennung von Fehlern
- Minimierung von Risiken

Begriffe:

- Prozess
  - Beschreibung Aktivitäten, Rollen und Artefakte(Informationen)
  - Software-Entwicklung und Wartung
- Prozessmodell
  - Präskriptives Modell (Vorgehensmodell und Organisationsstrukturen)
  - Planung und Lenkung
  - Unified Process, V-Modell, Scrum,...

Klassifizierung Software Probleme:

1.Dimension: Requirements (y-Achse Agreement)

2.Dimension: Technology (x-Achse Certainty)

3.Dimension: Skills,Intelligence, Experience, Attitudes, Prejudices

2.1 Vorgehensmodelle

1. Code and Fix
2. Wasserfallmodell
3. Iterative und inkrementelle Modelle

Code and Fix

- **Definition**
  - Codierung / Korrektur im Wechsel mit Ad-hoc Tests
- **Vorteile**
  - Schnell vorankommen
  - Schnelle Ergebnisse
  - Einfache Tätigkeiten (Codieren, Testen, Fixen)
- **Nachteile**
  - Schlecht planbar und keine Unterstützung im Team
  - Aufwand hoch für Korrekturen
  - Sclecht wartbare Software

Wassefallmodell

- **Definition**
  - Folge von Aktivitäten/Phasen, gekoppelt durch Teilergebnisse (Dokumente). Reihenfolge ist fest definiert.
- **Vorteile**
  - hohe Planbarkeit
  - Klare Aufteilung der SWE (Analyse, Design, Test,...)
- **Nachteile**
  - Schlechtes Risikomanagement (Lösungskonzept nur auf Papier validiert)
  - Anforderungen zu Beginn nie alle bekannt

Iterativ-inkrementelle Modelle

- **Definition**
  - Geplante und kontrollierte Iterationen inkrementell entwickelt
- **Vorteile**
  - Flexibles Modell bei unklaren Anforderungen
  - Gutes Risikomanagement (Mitarbeiter und Technologie)
  - Frühe Einsetzbarkeit der Software und Feedback
- **Nachteile**
  - Upfront Planbarkeit hat Grenzen (Funktionalität, Zeit und Kosten)
  - Braucht Involvierung und Steuerung durch den Kunden über ganze Projektdauer

2.2 Agile SWE

- Basiert auf iterativ-inkrementellen Prozessmodell
- Fokussiert auf gut dokumentierten und getesteten Code statt auf ausführlicher Dokumentation
- Sammlung von Ideen SWE Prozess flexibler und schlanker zu machern
- Adressiert bekannten Probleme bei klassischen Software-Prozessmodellen

Strategie

- Definierte Prozesskontrolle
  - Planung am Anfang, Prozess gesteuert und überwacht
  - Geeignet für gut planbare Problemstellungen
  - Strategie: Steuerung
- Empirische Prozesskontrolle (Agil)
  - Nur Grobplanung am Anfang
  - Prozess fortlaufend überwacht
  - Rollende Planung
  - Geeignet für komplexe Problemstellungen
  - Strategie: Regelung, Deming-Cycle (Plan-Do-Check-Act)

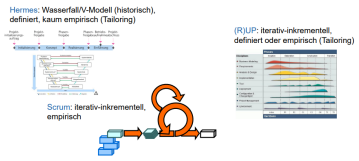


Abbildung 1: CharaktersierungProzessmodellen

3 Modellierung

Modell: Abbild oder Vorbild für ein zu schaffendes Gebilde.
Original: Abgebildete oder zu schaffende Gebilde

- **Modellierung**
  - Software = selbst ein Modell
  - Anforderungen = Modelle der Problemstellung
  - Architekturen und Entwürfe = Modelle der Lösung
  - Tesfälle = Modelle des Korrekten Funktionierens des Codes

3.1 UML

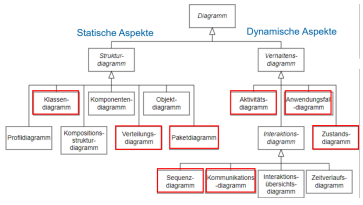


Abbildung 2: Guetereinteilung.

3.1.1 Gebrauch der UML

- UML as a sketch
  - Informell, unvollständig
  - Bevorzugt von agile Community
- UML as blueprint
  - Detallierte Analyse und Design-Diagramme für Code
  - Forward - und Reverse-Engineering
- UML as a Programming Language
  - Komplette, ausführbare Spezifikation eines Software-Systems in UML
  - MDA-Tools zur Modellierung und Generierung

4 Wesentliche Artefakte

- **Anfoderungsanalyse**
  - Funktionale Anforderungen mit Use Cases
  - Qualitätsanforderungen und Randbedingungen
  - Domänenmodell
- **Design**
  - Softwarearchitektur
  - Use Case Realisierung (Statische und dynamische Modelle)
- **Implementation**
  - Quellcode (inkl.Javadoc)
- **Testing**
  - Unit-Tests
  - Integrations- und Systemtests

4.1 Überblick Anforderungsanalyse

- **User Research**
  - Personas
  - Szenarien
  - Contextual Inquiry
- Sketching und Prototyping
- **Use Cases**
  - Ableiten und Modellieren
  - Detaillierung (UML-Use-Case-Diagramm, Use-Case-Spezifikation, UI-Sketching)
- **Qualitätsanforderungen, Randbedingungen** erheben
- **Domänenmodell**
  - Konzeptuelles UML-Klassendiagramm
- **objektorientierte Analyse(OOA)**
  - Objekte/Konzepte in dem Problembereich zu finden und zu beschreiben

4.2 Überblick Design

- **Softwarearfchitektur**
  - UML-Paketdiagramm
  - UML-Deploymentdiagramm
- **Use-Case-Realisierung und Klassendesign**
  - UML-Klassendiagramm
  - UML-Sequenzdiagramm
  - UML-Kommunikationsdiagramm
  - UML-Zustandsdiagramm
  - UML-Aktivitätsdiagramm
- Entwurf **Design Patterns**
- **Objektorientierte Design (OOD)**
  - Geeignete Softwareobjekte und ihr Zusammenwirken definieren

4.3 Überblick Implementation

- **Code**
  - Umsezung Design in entspr. OOP-Sprache
- **Refactoring**
  - Code smells aufdecken und verbessern
- **laufende Dokumentierung**
  - vom Quellcode

4.4 Überblick Testing

- **Unit-Tests**
  - Laufendes Design und Implementierung
- **Teststufen Integration und System**
  - Planung, Design und Durchführung
- **Dokumentation**
  - Testkonzept und Test

5 Usability und User Experience (UX)

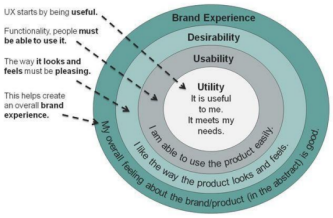


Abbildung 3: Usability.

5.1 Usability

Definition: Effektivität, Effizienz, Zufriedenheit -> Ziele erreichen im spezifischen Kontext

- **4 wichtige Aspekte**
  1. Benutzer
  2. Seine Ziele/Aufgaben
  3. Sein Kontext
  4. Softwaresystem (inkl.UI)

5.2 Usability Engineering

Ziel: Software entwickeln, die drei Anfroderungen erfüllt

- **Drei Anforderungen:**
  1. **Effektivität**
     - Aufgaben vollständig erfüllen
     - Genauigkeit
  2. **Effizienz**
     - Mit minimalem Aufwand (Mental, Physisch, Zeit)
  3. **Zufriedenheit**
     - **Minimum:** nicht verärgert
     - **Normal:** Zufrieden
     - **Optimal:** Erfreut

5.3 Usability Anforderungen

7 Anforderungen:

1. Aufgabengemessenheit
2. Lernförderlichkeit
3. Individualisierbarkeit
4. Erwartungskonformität
5. Selbstbeschreibungsfähigkeit
6. Steuerbarkeit
7. Fehlertoleranz

5.3.1 Aufgabenangemessenheit

- Minimale Anz. Schritte f. Aufgabe
- Nur wichtige Informationen
- Kontextabhängige Hilfe
- Minimale Anz. Benutzereingaben
  - Jede Eingabe nur 1x
  - Standardwerte
  - Liste vordefinierter Werte (z.B Länder)
  - Ableitbare Eingaben vorschlagen

5.3.2 Selbstbeschreibungsfähigkeit

- Benutzer ausreichend informieren
  - Wo er ist
  - Was er tun soll / kann
  - Wie er es tun soll (Formate, Werte)
- Begriffe des Benutzers verwenden (Labels, Fehlermeldungen)
- Affordanzen

5.3.3 Kontrolle

- Mit Interaktion Benutzer steuern
  - Initiative, Tempo
  - Dialogfluss
  - Darstellungsformate
  - Inputmodalität (Maus, Tastatur, Touch, Sprache)
- Benutzeraktionen rückgängig machen können
- Benutzeraktionen jeder Zeit abbrechen können

5.3.4 Erwartungskonformität

- Bezüglich
  - Design
  - Interaktion
  - Struktur
  - Komplexität
  - Funktionalität
- Konsistenz
  - Terminologie
  - Verhalten (Reihenfolge Aktionen, Änderungen)
  - Informationsdarstellung (Platzierung, Wortwahl)

5.3.5 Fehlertoleranz

- Benutzerfehler vermeiden
  - Klar kommunizieren (Erwarteter Input, Funktionen aktiv resp. sinnvoll)
- Benutzereingaben vor Aktion überprüfen
- Nicht unbedingt beim Tippen
- Benutzer helfen
  - Fehler zu erkennen
  - Ursache zu verstehen
  - Aus Fehlerzustand zu kommen
- Einfache Korrektur
- Kein Datenverlust

5.3.6 Individualisierbarkeit

- System anpassbar sein:
  - Know-How
  - Sprache
  - Kultur
  - Benutzer mit Einschränkungen

5.3.7 Lernförderlichkeit

- Informationen über unterliegende Konzepte und Regeln anbieten
  - Um mentales Modell anzugleichen
  - Nur auf Verlangen des Users
  - eifache Tasks ohne Vorkentnisse
  - komplexere Konzepte bei der Verwendung zu erlernen

6 User-Centered Design (UCD)

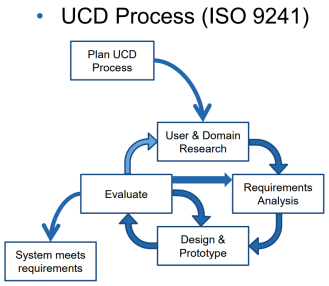


Abbildung 4: UCDProcess

6.1 User & Domain Research

- **Ziele bez. Benutzer:**
  - Wer ist Benutzer
  - Was ist die Arbeit (Aufgaben, Ziele)
  - Wie sieht Arbeitsumgebung aus
  - Was wird gebraucht um Ziele zu erreichen
- Welche Sprache, Begriffe
- Normen (organisatorisch, kulturell, sozial)
- Pain Points (Brüche, Workarounds)
- Für mobile Apps:
  - Nutzungskontext
    - Wo wird App benutzt (Umgebung)
    - Wann wird App benutzt (Tageszeit, involvierte Personen, Randbedingungen)
    - Warum wird App benutzt (Nutzen, Motivation, Trigger)
- **Ziele bez. Domäne:**
  - Buisiness der Firma verstehen
  - Domäne verstehen (Sprache, Wichtigste Konzepte, Prozesse)

6.1.1 GUI Design Process

Methoden User & Domain Research

- Contextual Inquiry
  - Was ist das?(Beobachtung, Interview)
  - Was braucht es dazu? (Für den inquiry, z.B Videogerät)
- Interviews
- Beobachtung
- Fokusgruppen
- Umfragen
- Nutzungsauswertung
- Desktop Research (Dokumentenstudium, Mitbewerber)

6.1.2 Wichtige Artekfakte

- Personas
- Usage-Szenarien
  - Kurze Geschichte
    - Usage Szenarien
      - aktuelle Situation
      - in User and domain research verwendet
    - Kontextszenarien
      - Zukünftige gewünschte Situation
      - in Anforderungsanalyse verwendet
- Mentales Modell
- Domänenmodell
- Stakeholder Map

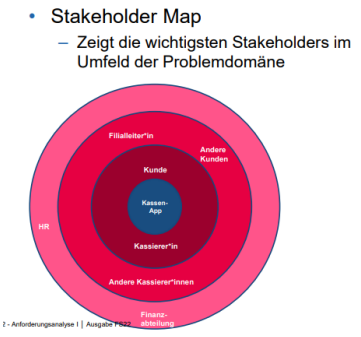


Abbildung 5: Stakeholdermap

- Service Blueprint/Geschäftsprozessmodell

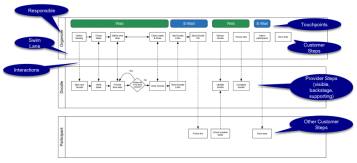


Abbildung 6: Blueprint

6.2 Anforderungsanalyse

Ziel:

- Ausgehend von den Resultaten des UCD -> User-Anforderungen ableiten:
  - Funktionale Abläufe, Interaktionen
    - **Kontextszenarien**
    - **Storyboards**
    - **UI-Skizzen**
    - **Use cases**
  - Konzepte, Beziehungen, Quantitäten
    - **Kontextszenarien**
    - **FURPS-Modell (Functionality, Usability, Reliability, Performance, Supportablility**

6.2.1 Use Cases

- **Akteur**
  - Primärakteur
  - Unterstützender Akteur
  - Offstage-Akteur
- **Keine Kann-Formulierungen**
- **3 Ausprägungen:**
  1. Kurz
     - Titel + 1 Absatz (Standardablauf)
  2. Informell
     - Titel + Informelle Beschreibung (können mehrere Absätze sein, beschreibt auch Varianten)
  3. Vollständig
     - Titel + alle Schritte und alle Varianten im Detail
     - UC-Name
     - Umfang
     - Ebene
     - Primärakteur
     - Stakeholders und Interessen
     - Vorbedingungen
     - Erfolgsgarantie/Nachbedingungen
     - Standardablauf
     - Erweiterungen
     - Spezielle Anforderungen
     - Liste der Technik und Datavariationen
     - Häfigkeit des Auftretens
     - Verschiedenes
- **Notation = Nomen + Verb**

6.2.2 Wie schreibt man Use Cases

Brief UC:

- Kurze Beschreibung des Anwendungsfalls in einem Paragraph
  - Nur Erfolgszenario
  - Sollte enthalten:
    - Trigger des UCs
    - Akteure
    - summarischen Ablauf des UCs
  - Zu beginn der Analyse

Casual UC:

- Informelle Beschreibung des Anwendungsfalls in mehreren Paragraphen
  - Nur Erfolgszenario + wichtigste Alternativszenarien
  - Sollte enthalten:
    - Trigger des UCs
    - Akteure
    - Interaktion Akteurs mit System
  - Zu beginn der Analyse

Fully-dressed UC:

- Detaillierte Beschreibung des ablaufs mit allen Alternativszenarien
- ende der Inception und v.a. in Elaborationphase
- Die wichtigsten UCs(10%), die die Architektur bestimmen

Formaler Aufbau:

1. **UC-Name**
   - Aktiv formulieren (Verb + ev.Objekt)
   - Beschreibt Job(Ziel,Aufgabe),den Akteur ausführen will
2. **Umfang(Scope)**
   - Beschreibt das zu entwickelnde System (SuD= System under Development)
3. **Ebene(Level)**
   - Anwenderziel oder
   - Subfunktion
4. **Primärakteur**
   - Hauptakteur des UCs
     - Primärer Nutzniesser des UC
     - initiiert den UC
     - Interagiert hauptsächlich mit dem System
5. **Stakeholders und Interessen**
   - Für wen ist der UC sonst noch relevant und welche Interessen hat er daran?

6. **Vorbedingungen(Preconditions)**
   - Was ist die unmittelbare Voraussetzung, damit UC ablaufen kann? (Nur die wichtigsten,offensichtliche Voraussetzungen)
7. **Erfolgsgarantie/Nachbedingungen(Success Guarantee)**
   - Was muss gewährleistet sein nach erfolgreichem Ablauf von UC
8. **Standardablauf (Main Success Scenario)**
   - Wichtigster Teil des UCs(Beschreibung erfolgreichen Ablaufs des UCs)
   - Beschreibung Interaktion Primärakteurs mit dem System(+allenfalls Interaktion mit unterstützenden Akteuren)
   - Startpunkt= nach den Vorbedingungen
   - Keine Lösungsdetails
9. **Erweiterungen (Extensions)**
   - Alternative Erfolgs-aber auch Misserfolgszenarien
   - Erweiterungen / alternative Abläufe
   - * = Alternativablauf zu jeder Zeit auftreten kann
   - Interaktion des alternativen Ablaufs analog zu Hautptszenario

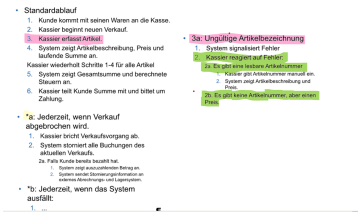


Abbildung 7: Beispiel-Fully-Dressed.

-
- **Spezielle Anforderungen (Special Requirements)**
  - Weitere Anforderungen, die aus diesem UC resultieren
- **Liste der Technik und Datavariationen (Technology and Data Variations)**
  - Alternative I/O-Methoden, Datenformate, etc.(Notation wie bei Erweiterung)
- **Häufigkeit des Auftretens(Frequency of Occurance)**
  - Wie häufig tritt dieser UC auf?
- **Verschiedenes (Miscellaneous)**
  - Offene Fragen/ Probleme

von Anwendungsfällen zu konkreten Funktionalitäten:

- Systemsequenzdiagramme
- Operation Contracts

6.2.3 UML Sequenzdiagramm (SSD)

Zeigt: Interaktion der Akteure mit dem System

- Welche Input-Events auf das System einwirken
- Welche Output-Events das System erzeugt

SSD können auch Interaktionen zwischen SuD und externen unterstützenden System zeigen.

Ziel: Wichtigste Systemoperationen identifizieren, die das System zur Verfügung stellen muss (API) für einen gegebenen Anwendungsfall

Wie Systemoperationen finden:

1. Szenario für UC Schritt für Schritt durchgehen
2. Für jeden Schritt Systemoperation überlegen
    - Geeigneten, präzisen Namen wählen (POV Akteur)
    - Welche Info braucht das System um Systemop. auszuführen? (Wenn nicht vorhanden im System -> Parameter)

6.2.4 Operation Contract

Definition: Spezifiziert (System)Operation

- Name plus Parameterliste
- Vorbedingung (Was muss zwingend erfüllt sein damit Systemoperation aufgerufen werden kann)
- Nachbedingung
    - Was hat sich alles geändert nach Ausführung (Erstellte / gelöschte Instanzen, Assoziationen, geänderte Attribute), Im Präsens schreiben
    - basierend auf Domänenmodell

Wann Operation Contracts?

- Nur wenn aus Anwendungsfall nicht klar wird, was Systemoperation genau machen muss
    - Meist nur bei sehr komplizierten Operationen und/oder
    - Wenn Entwicklung der Systemoperation ausgelagert wird
- Erst gegen Ende des Meilensteins Lösungsarchitektur oder kurz vor Start des Designs der Systemoperation

6.2.5 Zusätzliche Anforderungen

- weitere Funktionale (grosser Teil schon von UCs beschrieben)
- Nicht-Funktionale

Formulierung

- Anforderungstatements
  - Als Anforderung formuliert
  - messbar/verifizierbar
- So wenig wie nötig
  - Nur diejenige, die begründet gefordert werden
  - Keine ersten Lösungsideen als Forderungen
- User-Stories
  - In einem Satz wer,was,warum fordert
  - Erfüllen einige Bedingungen automatisch
  - die anderen Bedingungen (Messbarkeit etc.) solten auch erfüllt sein

Checkliste FURPS+

- **F**unctionality
  - Features, Fähigkeiten, Sicherheit
- **U**sability
  - Usability Anforderungen (Kap.5.3)
  - Accessibility
- **R**eliability
  - Fehlerrate, Wiederanlauffähigkeit, Vorhersagbarkeit, Datensicherung
- **P**erformance
  - Reaktionszeiten, Durchsatz, Genauigkeit, Verfügbarkeit, Ressourceneinsatz
- **S**upportability
  - Anpassungsfähigkeit, Wartbarkeit, Internationalisierung, Konfigurierbarkeit
- +
  - Implementation (HW,BS,Sprachen, Tests...)
  - Interface
  - Operations
  - Packaging
  - Legal

Glossar

- Einfaches Glossar
  - Begriffe im Projekt und SW-Produkt
  - beliebige Elemente
- Data Dictionary
  - Zusätzliche Datenformate, Wertebereiche, Validierungsregeln

7 Domänenmodellierung

Definition: Vereinfachtes UML-Klassendiagramm.

Aufbau: Fachliche Begriffe mit ihren Attributen, setzt Begriffe in Beziehung zueinander. Geht nur um die Problemstellung und das Fachgebiet.

Wie Domänen finden:

- Substantive markieren in Szenario (Achtung nicht alle sind Konzepte, gewisse sind auch Attribute oder gehören nicht zum Fachgebiet)

7.1 Domänenmodell als vereinfachtes UML Klassendiagramm

Konzepte = Klassen
Eigenschaften = Attributen (Typenangabe entfällt)
Assoziatonen = Beziehungen zwischen Konzepte mit Multiplizitäten an beiden Enden.

Nur wenn es einen guten Grund gibt:

- **Aggregation** = Keine echte Semantik, als Abkürzung für "hat".
- **Komposition** = z.B wenn Produktkatalog gelöscht wird, dann auch die darin enthaltenen Produktbeschreibungen. Abkürzung "bietet an".

7.2 Vorgehen

1. Konzepte identifizieren
   (a) Fachwissen und Erfahrung verwenden
   (b) Substantive aus Anwendungsfällen
   (c) Kategorienliste verwenden
2. Attributen
   (a) Fachwissen verwenden
3. Konzepte in Verbindung zueinander setzen
   (a) Fachwissen verwenden
   (b) Kategorienliste verwenden
4. Auftraggeben und/oder Fachexperten beiziehen
5. Vorgehensweise eines Kartografen

7.2.1 Kategorienliste

Kategorie	Mögliche Konzepte für DM
Geschäftstransaktionen	
• Transaktionen als Ganzes	Sale
• Transaktionsposition	SalesLineItem
• Produkt, das damit verbunden ist	Item
• Wo wird Transaktion registriert?	Register
• Rollen von beteiligten Personen	Cashier
• Ort der Transaktion	Store
• Beschreibung von Dingen	ProductDescription
• Ereignisse mit Ort/Zeit	Sale

Abbildung 8: Kategorienliste1

Kategorie	Mögliche Konzepte für DM
Physische Objekte	Register
Kataloge	ProductCatalog
Container von Dingen	Store
Dinge in den Containern	Item
Andere beteiligte Systeme	CreditAuthorizationSystem
Rollen von beteiligten Personen	Cashier
Artefakte (Pläne, Finanzen, Arbeit, Verträge, ...)	Receipt
Zahlungsinstrumente	Cash, Credit Card

Abbildung 9: Kategorienliste2

Kategorie	Mögliche Assoziation für DM
Transaktion	Payment - Sale
• Position	SalesLineItem - Sale
• Produkt	Item - SalesLineItem
• Rolle	Customer - Payment
Teil zum Ganzen	Register - Store
Beschreibung zum Gegenstand	ProductDescription - Item
Protokoll zum Gegenstand	Sale - Register
Verwendung	Cashier - Register

Abbildung 10: Kategorienliste3

7.3 Datentypen von Attributen

- Wenn nötig: eigene Datentypen als **Konzepte**
- Dann definieren wenn:
  - Typ aus mehreren Abschnitten (wie Tel.Nr)
  - Operationen darauf sind möglich (Validierung Kreditkartennummer)
  - Hat selber eigene Attribute (Verkaufspreis mit Anfangs & Enddatum)
  - Verknüpft mit Einheit (Preis mit Währung)

Anti-Pattern: Assoziationen statt Attribute, um Konzepte in Beziehung zueinander zu setzen.

7.4 Vorgehensweise eines Kartografen

- Vorhandene Begriffe oder Wissen einsetzen (Gebiete besuchen, Bewohner nach Begriffen befragen)
- Unwichtiges weglassen
- Nichts hinzufügen, was es (noch) nicht gibt
  - Ausnahme: System, das enwickelt wird, kann eingetragen werden
- Nur analysieren, (noch) keine Lösungen entwerfen

Anti-Pattern: Keine Software Klassen im Domänenmodell

7.5 Analysemuster

- **Beschreibungsklassen**
  - Item = Physischer Gegenstand oder Dienstleistung
  - Mehrere Artikel desselben Typs
  - Attribute (description, price, serial number, itemID)
- **Generalisierung / Spezialisierung**
  - Spezialisierung als ïs a"Beziehung zu
- **Komposition**
- **Zustände**
  - Eigene Hierarchie für Zustände definieren:
- **Rollen**
  - Dasselbe Konzept kann unterschiedliche Rollen einnehmen:
- **Assoziationsklasse**
  - Wenn Assoziationen eigene Attribute haben (MerchantID für Kreditkarte Geschäft¡-¿AuthorizationService)
- **Einheiten**
  - Manchmal sinnvoll explizit als Konzept zu modellieren
- **Zeitintervalle**
  - Gültigkeitsintervall für sich ändernede Attribute

8 Softwarearchtiketur und Design I

- Gesamtheit der wichtigsten Entwurfs-Enscheidungen.
  - Programmiersprachen, Plattformen
  - Aufteilung: Teilsysteme, Bausteine, Schnittstellen
  - Veratnwortlichkeiten und Abhängigkeiten der Teilsysteme
  - Basis-Technologie oder Frameworks (z.B Java EE)
  - Besondere Massnahmen um Anforderungen zu erfüllen
- Grundlagen
  - Anforderungen (vorallem nicht-funktionale)
  - Systemkontext mit Schnittstellen
- Top Level View (das grosse Ganze)

8.0.1 Übersicht Buisiness Analyse vs Architektur vs Entwicklung

Business Analyse

- Domänenmodell
- Kontext Diagramm
- Requirements
  - Liste Stakeholder
  - Vision
  - Funktionale Anforderungen:
    - Use Cases / User Stories
  - Nicht funktionale Anforderungen:
    - Supplementary Specification
  - Randbedingungen
  - Glossar

Architektur

- Logische Architektur

Entwicklung

- Use Case / User Story Realisierung
- Anwendung GRASP
- DCD - Design-Klassen-Diagramm
- Interaktionsdiagramme
- Programmierung
- Erstellen der Unit-/Integrations-Tests

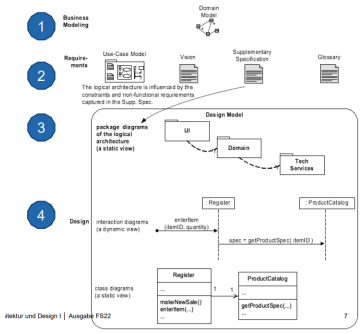


Abbildung 11: Übersicht

8.0.2 Wie entstehen Architekturen

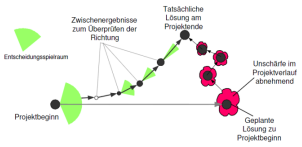


Abbildung 12: ArchitekturEntstehung

Architektur aus Anforderungen ableiten

- Muss heutige und zukünftige Anfroderungen erfüllen können
- Aufgabe Architekturanalyse
  - Analyse funktionale und nicht funktionale Anforderungen und deren Konzsequenzen
  - Berücksichtigung Randbedingung und zukünftige Veränderungen
  - Qualität, Stabilität der Anforderungen prüfen
    - Lücken in Anforderungen aufdecken

Twin peak Model

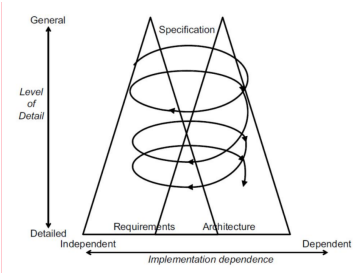


Abbildung 13: TwinPeak

Entwurfsentscheidungen

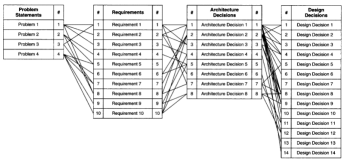


Abbildung 14: EntwurfsEntscheidungen

Nichtfunktionale Anforderungen

Abbildung 15: NichtfunktionaleAnforderungen

8.1 Modulkonzept

- Modul(Baustein, Komponente):
  - Autarkes Teilsystem
  - Klare, minimale Schnittstelle gegen Aussen
  - Software-Modul enthält alle Funktionen und Datenstrukturen
  - Modul: Paket, Programmierkonstrukt, Library, Komponente, Service
- Konzept in allen Ingenierdisziplinen angwendet

Schnitstellen Kapselung und Austauschbarkeit

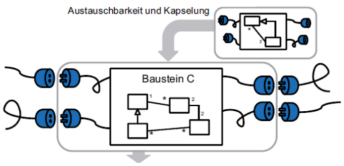


Abbildung 16: Schnittstellen

Prinzip modularen Strukur

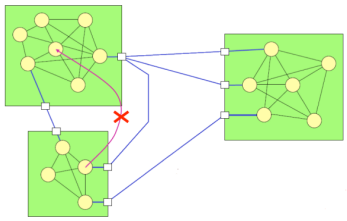


Abbildung 17: ModularenStruktur

8.1.1 Messung der Güte einer Modularisierung

- Kohäsion (Stärke inneren Zusammenhangs)
  - schlecht: zufällig, zeitlich
  - gut: funktional, objektbezogen
  - je höher Kohäsion innerhalb Modul, desto besser die Modularisierung
- Kopplung (Abhängigkeit zwischen 2 Modulen)
  - schlecht: Globale Kopplung
  - akzeptabel: Datenbereichskopplung (Referenzen auf gemeinsame Daten)
  - gut: Datenkopplung (alle Daten werden beim Aufruf der Schnittstelle übergeben)
  - Je geringer die wechselseitige Kopplung desto besser die Modularisierung

Clean Architecture

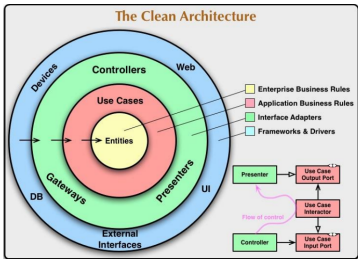


Abbildung 18: CleanArchitecture

- Unabhängikeit:
  - von Framework
  - voneinander getestet werden
  - von UI
  - von DB

8.2 Architektur Beschreiben

Aufgeteilt in Views:

- Logical View
  - Funktionalität gegen aussen
  - Aspekte:
    - Schichten
    - Subsysteme
    - Pakete
    - Frameworks
    - Klassen
    - Interfaces
  - UML:
    - Systemsequenzdiagramme
    - Interaktionsdiagramme
    - Klassendiagramm
    - Zustandiagramme
- Process View
  - Wo und wie im System
  - Aspekte:
    - Prozesse
    - Threads
    - Wie Anforderungen erreicht
  - UML:
    - Klassendiagramme
    - Interaktionsdiagramme
    - Aktiviitätsdiagramme
- Development View (Implementation View):

  - Wie Struktur umgesetzt
  - Aspekte:
    * Source Code
    * Executables
    * Artefakte
  - UML:
    * Paketdiagramme
    * Komponentendiagramme

- Physical View (Deployment View)
  - Auf welcher Infrastruktur wird System ausgeliefert /betrieben
  - Aspekte:
    * Prozessknoten
    * Netzwerke
    * Protokolle
  - UML:
    * Deployment Diagram

- +1 View: Scenarios (Use Cases)
  - Wichtigste Use Cases und ihre nicht funktionale Anforderungen? Wie umgesetzt?
  - Aspekte:
    * Architektonisch wichtige UCs
    * deren nichtfunktionale Anforderungen
    * deren Implementation
  - UML:
    * UC-Diagramm
    * Systemsequenzdiagramme
    * UC-Realisierungen

- Daten-Sicht
- Sicherheit

Logische Architektur vs Physikalische Architeckur

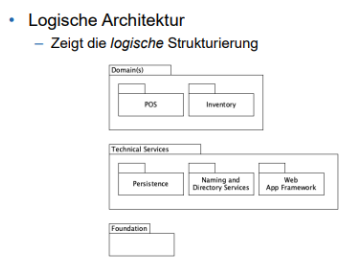


Abbildung 19: LogischeArchitektur

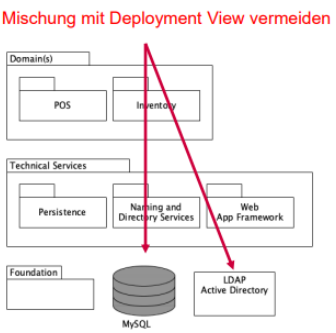


Abbildung 20: VermeidungArch

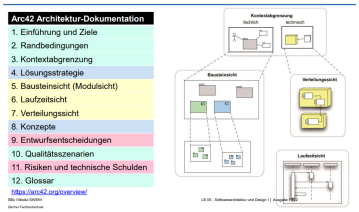


Abbildung 21: Arc42

8.3 UML-Paketdiagramme

- Mittel, zum Teilsysteme zu definieren
- Mittel zur Gruppierung von Elementen
- Paket enthält Klassen und andere Pakete
- Abhängigkeit zwischen Paketen

8.4 Verteilungsdiagramm

- Darstellung Verteilung von Komponenten auf Rechenknoten mit Abhängigkeiten, Schnittstellen und Verbindungen
- Statische Modellierung

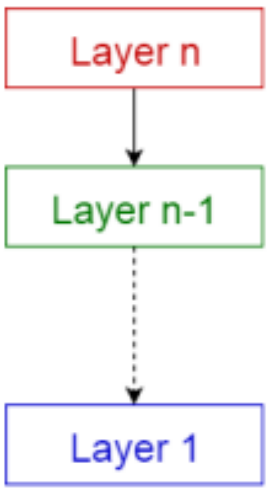
8.5 Ausgewählte Architekturpatterns

Pattern	**Beschreibung**
Layered Pattern	Strukturierung eins Programms in Schichten
Client-Server Pattern	Server stellt Services für mehrere Clients zur Verfügung
Master-Slave Pattern	Master verteilt Arbeit auf mehrere Slaves
Pipe-Filter Pattern	Verarbeitung einses Datenstroms (filtern, zuordnen, speichern)
Broker Pattern	Meldungsvermittler zwischen verschiedenen Endpunkten
Event-Bus Pattern	Datenquellen publizieren Meldungen an einen Kanal auf dem Event-Bus. Datensenken abonnieren einen bestimmten Kanal
MVC Pattern	Ineraktive Anwendung in 3 Komponenten aufgeteilt: -Model -View - Informationsanzeige -Controller - Verarbeitung Benutzereingabe

8.5.1 Layered Pattern

- Je weiter unten, desto allgemeiner
- Je höher, desto anwendungs-spezifischer
- Zuoberst ist das Benutzerinterface

Kopplung von oben nach unten NIE von unten nach oben.



Anrufszenarien

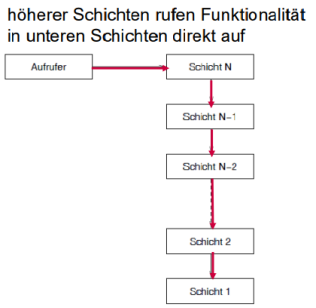


Abbildung 23: AnrufSzenarienH

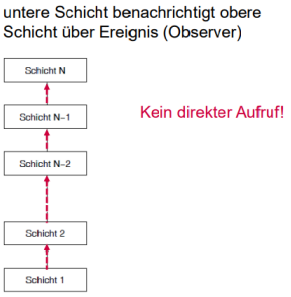


Abbildung 24: AnrufSzenarienObserver

- UI
  - Presentation, Windows, Dialoge, Reports, WEB, Mobile
- Application
  - Requests von UI Layer, Workflow, Sessions
- Domain
  - Requests von Application Layer, Domain Rules, Services
- Business Infrastructure
  - Low level business Services (z.B CurrencyConverter)
- Technical Services
  - Persistence, Security, Logging
- Foundation
  - Datenstrukturen, Threads, Dateien, Netwerk IO

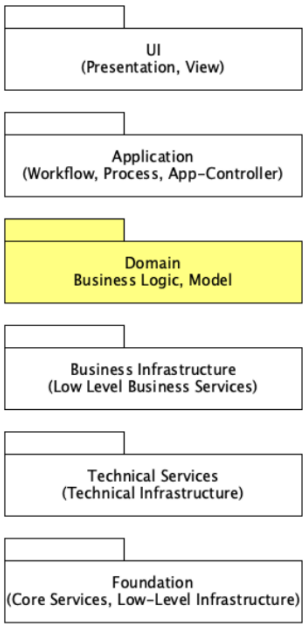


Abbildung 25: LayeredPattern2

8.5.2 Client-Server

- 1 Server und mehrere Clients
- 1 Server stellt einen oder mehrere Services zur Verügung
- Client macht Request zum Server
- Server sendet Response zurück

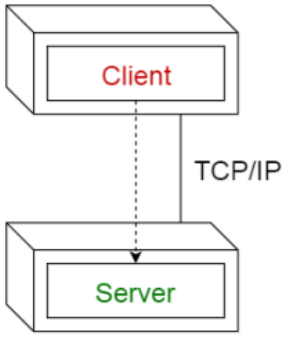


Abbildung 26: ServerClient

8.5.3 Master-Slave Pattern

- Master verteilt die Aufgaben auf mehrere Slaves
- Slaves führen Berechnungen aus und senden Ergebnis zum Master
- Maseter berechnet Endergebnis

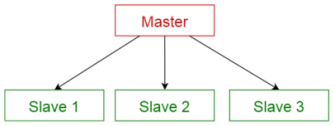


Abbildung 27: MasterSlave

8.5.4 Pipe-Filter Pattern

- Verarbeitung von Datenströmen (Linux Pipe, RxJS Observable Streams, Java Streams,...)
- Verarbeitungsschritt durch Operator wie Filter, Maper, etc. umgesetzt

Abbildung 28: PipeFilter

8.5.5 Broker Pattern

- verteilte Systeme mit entkoppelten Subsysteme zu koordinieren
- Broker(Vermittler) ermittelt Kommunikation zwischen einem Client und dem entspr. Subsystem
- Bsp.: Message Broker

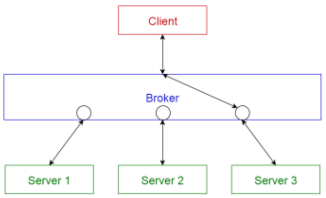


Abbildung 29: BrokerPattern

8.5.6 Event-Bus Pattern

- 4 Hauptkomponenten:
  1. EventSource
  2. Eventlistener
  3. Channel
  4. Event Bus
- Event Sources publizieren Meldungen zu einem bestimmten Kanal auf dem Event Bus
- EventListeners:
  - Melden sich für bestimmte Events an
  - werden informiert, sobald entsprechende Meldungen auf dem Kanal befinden

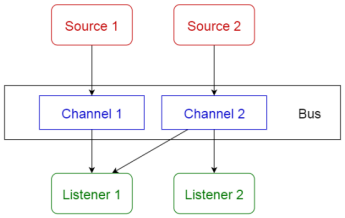


Abbildung 30: EventBus

8.5.7 Model View Controller Pattern

- Interaktive Anwendung in 3 Komponenten:
  - Model: Daten und Logik
  - View: Informationsanzeige
  - Controller: Verarbeitung der Benutzereingabe
- Entkopllung UI und Logik
- Erlaubt Austauschbarkeit des UIs
- Alternativen:
  - MVVM: Model View View Model
  - MVP: Model View Presenter

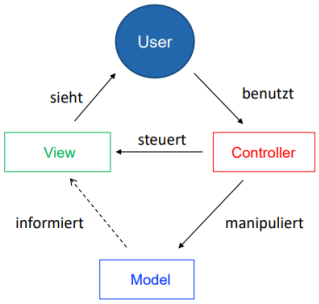


Abbildung 31: MVC

9 Softwarearchitekur und Design II

9.1 Objektorientierung

Grundidee: Reale Welt besteht aus Objekten, die untereinander in Beziehungen stehen.

- Klasse:
  - Daten(Attribute)
  - Funktionalität(Operationen, Methoden)
- Objekte:
  - In der Lage Nachrichten (= Methodenaufrufe) zu empfangen
  - Daten verarbeiten
  - Nachrichten senden
  - können einmal erstellt werden
  - in verschiedene Kontexten wiederverwendet werden

Objektorientierte Analyse(OOA): Objekte-Konzepte-in Domäne zu finden und zu beschreiben.

Objektorientierten Design (OOD): Geeignete Softwareobjekte und ihr Kollaboration zu definieren um Anforderungen zu erfüllen.

Domänenschicht: Klassen abgeleitet aus dem Domänenmodell (Low-Representational-Gap)

9.1.1 Use Cases und System-Sequenzdiagramm

Basis für das Design:

1. Szenarien
2. Systemoperationen
3. Domänenmodell

Was programmiert werden muss:

1. Systemoperationen bzw. deren Antworten

Use-Case-Realisierung: Wie ein bestimmter Use Case innerhalb Design mit kollaborierenden Objekten realisiert wird.

Systemoperationen: Jedes Szeanario schrittweise entworfen und implementiert

UML-Diagramme: Gemeinsame Sprache um Use-Case-Realisierungen zu veranschaulichen und zu diskutieren.

9.1.2 Klassen entwerfen:

Zwei arten von Design-Modellen (ergänzen sich und werden parallel erstellt):

- Statische Modelle:
  - UML-Klassendiagramm- Unterstützen Entwurf Paketen, Klassennamen, Atrributen und Methodensignaturen (ohne Methodenkörperd)
- Dynamische Modelle:
  - UML-Interaktionsdiagramme Untersützen Entwurf der Logik, des Verhaltens des Codes und der Methodenkörper.

9.2 UML-Diagramme für das Design

9.2.1 Grundelemente der UML

- Primitiver Datentyp
- Literal
- Schlüsselwort, Stereotyp
- Randbedingung (constraint)
- Kommentar
- Diagrammrahmen (optional)

9.2.2 UML-Klassendiagramm

- Statische struktur
- Konzeptuell: Problemdomäne (Domänenmodell)
- Design: Lösungsdomäne (DCD)

Notationselemente

- Klasse
- Attribut
- Operation
- Sichtbarkeit von Attributen und Operationen
- Assoziationsname, Rollen an den Assoziationsenden
- Multiplizität (Objekte der betreffenden Klasse)
- Navigierbarkeit in Assoziationen
- Datentypen und Enumerationen

- Generalisierung / Spezialisierung
- Abstrakte Klassen
- Komposition
- Aggregation
- Interface
- Interface - Realisierung (Menge von öffentlichen Operationen, Merkmalen und Verpflichtungen, die durch eine Klasse, die die Schnittstelle implementiert, zwingend zur Verfügung gestellt werden müssen.
- Assoziationsklasse (Da wenn ** Beziehung existiert)
- Aktive Klasse (Instanz wird in einem separaten Thread ausgeführt)

9.2.3 UML-Interaktionsdiagramme

Spezifiziert, auf welche Weise Nachrichten und Daten zwischen Interaktionspartnern ausgetauscht werden.

2 Arten:

1. Sequenzdiagramm
2. Kommunikationsdiagramm

Anwendung: Kollaborationen bzw. Informationsaustausch zwischen Objekten zu modellieren.

- Wer tauscht mit Wem Information aus?
- in welcher Reihenfolge
- Zeitlicher Ablauf
- Schachtelung und Flussteuerung (Bedingung, Schleifen, Verzweigungen) möglich.

Kann in mehreren Perspektiven verwendet werden:

- Analyse
  - mit SSD Input-/Output-Erignisse (Systemoperationen mit Rückgabeantworten) für ein Szenario eines Use Cases modelliert
- Design
  - mit SSD Interaktion zwischen Objekten zur Realisierung eines konkreten Use-Case Szenarios zu modellieren

Notationselemente:

- Lebenslinie
- Aktionssequenz
- Synchrone Nachricht
- Antwortnachricht
- Gefundene, verlorene Nachricht
- kombiniertes Fragment

- Erzeugnis-, Löschereignis
- Selbstaufruf
- Interaktionsreferenz
- Lebenslinie mit aktiver Klasse
- Asynchrone Nachricht

9.2.4 UML-Kommunikationsdiagramm

Beantwortet Frage:
Wer kommuniziert mit werm? Wer arbeitet im System zusammen? Darstellung Informationsaustausch zwischen Kommunikationspartnern. Überblick im Vordergrund.

Notationselemente:

- Lebenslinie (Box)
- Synchrone Nachricht (=Aufruf einer Operation)
- Antwortnachricht (=Rückgabewert)
- Bedingte Nachrichten "[]"
- Iteration "*"
- Nummerierung der Nachrichten (Festlegung Zeitliche Abfolge)
  - Systemoperation (msg1, ohne Nummer)
  - Nummern gleicher Hierarchie(1,2,3...)
  - Nummer unterer Hierarchie:
    - Werden innerhalb der Operation darüber ausgeführt (verschachtelt)
    - Operation 1.1:msg3 wird innerhalb von Operation 1:msg2 aufgerufen

9.2.5 UML-Zustandsdiagramm

Beantwortet zentrale Frage:
Welche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, ...bei welchen Ereignissen annehmen?
Präzise Abbildung eines Zustandmodells (endlicher Automat):

- Zuständen
- Ereignissen
- Nebenläufigkeiten
- Bedingungen
- Ein- und Austrittsaktionen

Verwendung: Modellierung von Echtzeitsystemen, Steuerungen und Protokollen

Notationselemente:

- Start-, Endzustand
- einfacher Zustand
- Zusammengesetzter bzw. geschachtelter Zustand
- Transition
- Orthogonaler Zustand
- Parallelisierungsknoten
- Synchronisationsknoten
- Einstiepunkt
- Ausstigespunkt
- Unterzustandautomat
- Zusammengesetzter Zustand
- Flache und tiefe Historie

9.2.6 UML-Aktivitätsdiagramm

Beantwortet zentrale Frage:
Wie läuft ein bestimmter Prozess oder ein Algortithmus ab?
Detaillierte Visualisierung von Abläufen:

- Bedingungen
- Schleifen
- Verweigungen

Parallelisierung und Synchronisation von Aktionen möglich.

Notationselemente:

- Aktivität
- Aktionsknoten (Aktion)
- Objektknoten (Objekt)
- Entscheidungs- und Vereinigungsknoten
- Initialknoten
- Aktivitätsendknoten
- Partition / Swimlane
- Parallelisierungsknoten
- Synchronisationsknoten
- SendSignal-Aktion
- Ereignis- bzw. Zeitereignisannahmeaktion
- CallBehavior-Aktion

9.3 Verantwortlichkeiten und Responsibility-Driven-Design

Methode über Entwurf Softwareklassen nachzudenken: **Verantwortlichkeiten, Rollen, Kollaborationsbeziehungen** = **RDD / Responsibility Driven Design**

Softwareobjekte haben Verantwortlichkeiten und arbeiten mit anderen Objekten zusammen.
Verantwortlichkeiten werden durch Attribute und Methoden implementiert.
Kann auf jeder Ebene des Designs angwendet werden:

- Klasse
- Komponente
- Schicht

2 Ausprägungen von Verantwortlichkeiten:

Doing-Verantwortlichkeiten / Algorithmen, Code:

- Selbst etwas tun
- Aktionen anderer Objekte anstossen
- Aktivitäten anderer Objekte kontrollieren und steuern

Knowing-Verantwortlichkeit / Daten, Attribute:

- Private eingekapselte Daten
- Dinge kennen, die es ableiten oder berechnen kann
- Daten/Objekte zur Verfügung stellen, die aus bekannten Daten/Objekten abgeleitet oder berechnet werden können

9.3.1 GRASP- 9 Patterns

GRASP: Methodischer Ansatz für das OO-Design = General Responsibility Assignment Software Patterns)

Idee: Mit Pattern-Language, gutes wiederverwendbares Design zu kodifizieren.

Definition Pattern: Ein benanntes Problem-Lösungspaar

1. **Information Expert**
   - Derjenige, der was weiss
2. **Creator**
   - Zuständig erstellen Objekte(new())im Framework ABER delegiert als Dependency Injection
3. **Controller /Domain Controller = Service**
4. **Low Coupling**
   - Weniger Abhängigkeiten
5. **High Cohesion**
   - Abhängigkeiten zusammentun
6. **Polymorphism**
   - Unterschiedliche Objekte können gleiche Nachricht empfangen und interpretieren

7. **Pure Fabrication**
8. **Indirection**
9. **Protected Variations**

9.3.2 Information Expert

Problem:
Gibt es ein grundlegendes Prinzip, um Objekten Verantwortlichkeiten zuzuweisen?
Lösung:
Verantwortlichkeit einer Klasse zuweisen, die über die erforderlichen Informationen verfügt, um sie zu erfüllen. Partielle Verantwortlichkeiten sind auch möglich.
Alternativen:
Low Coupling, High Cohesion (Künstliche Klasse)

9.3.3 Creator

Problem:
Wer verantwortlich, neue Instanz einer Klasse zu erzeugen? Denn Nachteil von new..() = Kopplung
Lösung:
Klasse A zuständig eine neue Instanz einer **Klasse B** zu erzeugen, wenn eine oder mehrerere der folgenden Aussagen wahr ist(je mehr desto besser):

- A eine Aggregation oder ein Kompositum von B
- A registriert oder erfasst B-Objekte
- A arbeitet eng mit B-Objekten zusammen oder hat enge Kopplung
- A verfügt über Initialisierungsdaten für B (A ist Experte bezüglich Erzeugung von B)

Wenn mehrere Optionen anwendbar sind, Klasse A vorziehen, die ein Aggregat oder ein Kompositum ist.
Alternative:
Factory Pattern, Dependency Injection(DI)

9.3.4 Controller

Service = 1.Schicht = Domain Controller
Problem.
Welches erste Objekt jenseits der UI-Schicht empfängt und koordiniert(kontrolliert) eine Systemoperation?
Lösung:
Verantwortlichkeit der Klasse zuweisen, der eine der folgenden Bedingungen erfüllt:

- Variante 1: Fassaden Controller:
  - Repräsentiert „Root-Objekt“, System bzw. übergeordnetes System auf dem die Sowftware läuft.
- Variante 2: Use Case Controller:
  - Pro Use-Case-Szenario eine künstliche Klasse, in der die Systemoperationen abläuft.

Controller macht selber nur wenig und delegiert fast alles!

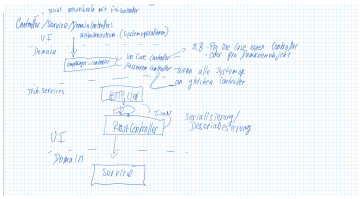


Abbildung 32: Controller.

9.3.5 Low Coupling

Problem:
Wie geringe Abhängigkeit erreichen, Auswirkungen von Veränderungen begrenzen und wie Wiederverwendbarkeit verbessern?
Ziel: geringe Abhängigkeit (besser für Refactoring)

- Kopplung = Mass für die Abhängigkeit von anderen Elementen (Klassen, Subsystem,Systeme)
- Hohe Kopplung= Element ist von vielen anderen Elementen abhängig
  - Nachteil: Refactoring mühsam, schwieriger zu verstehen, schwieriger wiederzuverwenden
- Niederige Kopplung = Element ist nur von wenigen anderen Elementen abhängig.

Lösung:
Verwantwortlichkeiten so zuweisen, dass Kopplung gering bleibt, Delegation Information Expert.

9.3.6 High Cohesion

Problem:
Erreichen, dass Objekte fokussiert, verständlich handbar bleiben und nebenbei Low Coupling unterstützen

- Kohäsion = Mass für Verwandschaft und Fokussierung eines Elements
- Hohe Kohäsion = Element erledigt nur wenige Aufgaben, die eng miteinander verwandt sind
- Geringe Kohäsion = Element, das für viele unzusammenhängende Dinge verantwortlich ist

Nachteil geringe Kohäsion: Schwierig zu verstehen, schwierig wiederzuverwenden, brüchig und instabil, sind laufend von Änderungen betroffen.
Lösung:
Verwantwortlichkeiten so zuweisen, dass Kohäsion hoch bleibt.

9.3.7 Polymorphismus

Problem:
Typabhängige Alternativen handhaben.

- Operation weist viele if-then-else bzw. grosse switch-case Anweisungen auf
- Bestimmtes Verhalten (z.B Einsatz eines externen Dienstes konfigurierbar machen.

Lösung:
Typabhängige Verhalten mit polymorphen Operationen der Klasse zuweisen, dessen Verhalten variiert.

- Grundlegende Idee OOP (Generalisierung / Spezialisierung)
- Überprüfen dass Beziehung „is a“ ist zwischen Superklase und Subklassen
- Liskov-Substituions-Prinzip einhalten

9.3.8 Pure Fabrication

Problem:
Wollen nicht gegen High Cohesion und Low Coupling oder andere Ziele verstossen, aber Lösungen passen nicht

- Viele Design-Klassen können direkt aus dem Fachbereich (Domänenmodell) abgeleitet werden und erfüllen das Low Representational Gap
- viele Situationen wo es Probleme gibt wenn Verantwortlichkeiten der Klasse in der Domänenschicht zugewiesen werden

Lösung:

- Hoch kohäsiven Satz von Verantwortlichkeiten einer künstlichen Hilfsklasse zuweisen
- Nur erstellt um höhere Kohäsion, geringe Kopplung oder eine bessere Wiederverwendbarkeit zu realisieren.

9.3.9 Indirection

Problem:
Verantwortlichkeit zuweisen, so dass direkte Kopplung zwischen 2 oder mehrere Objekte vermeidet wird. Objekte Entkoppeln, für geringere Kopplung und Wiederverwendungspotential grösser.
Ziel: Zwischen Adapter vermittelt
Lösung:

- Verantwortlichkeit einem zwischengeschalteten Objekt zuweisen
- Vermittler schafft eine Indirektion zwischen den anderen Komponenten
- Alternativen = Protected Variations
- GoF Patterns wie Adapter, Bridge, Facade, Observer oder Mediator verwenden dieses Prinzip
- viele Indirections sind Pure Fabrications

9.3.10 Protected Variations

Problem:
Objekte, Subsystem und Systeme entwerfen, so dass Veränderungen und Instabilitäten in diesen Elementen keinen Einfluss auf andere Elemente haben.
Ziel: Im Vorfeld Erweiterungen vorbereiten für Zukunft.
Lösung:
Punkte identifizieren, an denen Veränderungen und Instabilitäten zu erwarten sind. Verantwortlichkeiten so zuweisen, dass diese Punkte durch ein stabiles Interface eingekapselt werden.
Unterscheidung der Änderungspunkten:

- Variationspunkt:
  - Veränderung sind sicher(in Anforderung), zwingend PV Konzepte einbauen
  - Entwicklungspunkt:
    - Veränderungen nicht sicher, treffen mit hoher Wahrscheinlichkeit ein, sind nicht in Anforderungen enthalten.

Spekulative Anwendungen vermeiden, sonst unnötige Komplexität.

10 Implementation,Refactoring,Testing

10.0.1 Design to Code

- Quellcode aus Design Artefakten ableiten
- Praxis: Nur Teile des gesamten Quellcodes zusätzlich als Design Artefakte abgebildet.

Einsatz von Collection Klassen: Erforderlich bei 1:n Beziehungen

Fehlerbehandlung:

- Exceptions verwenden
- Exceptions nur für Fehlersituationen, nicht für reguläre Rückgabewerte
- Standard Exceptions verwenden
- Wo sinnvoll eigene Klassen definieren
- Jede Schicht kapselt Exception Handling ab

Umsetzungs-Reihenfolge

Bottom-Up Strategie: Wenn alle umzusetzende Klassen als Design Artefakte vorhanden sind

Variante Agile:

- Nur für die Iteration notwendigen Klassen bekannt. Funktionen Schritt für Schritt umgesetzt.
- Vorhandene Klassen müssen angepasst werden (Refaktoriert)
- Umsetzung über verschiedene Schichten der Architektur vollzogen (Model, Controller, Services, Repository)
- Ausgangspunkt oft Schnittstellenbeschreibung:
  - Benutzerschnittstelle (UI-Designer)
  - Systemschnittstelle (OpenApi Swagger)

Methoden:

- High Cohesion
- Eher kleine Methoden mit startkem inneren Zusammenhang
- CQS - Command Query Separation anwenden (setter und getter?)
- Wenn viele if's: Polymorphismus einsetzen

10.1 Implementation

3 Verschiedene Implementierungsstrategien:

1. Code-Driven Development (Zuerst die Klasse implementieren)
2. Test-Driven Development (TDD) (Zuerst Tests für Klassen/Komponenten dann Code entwickeln)
3. Behaviour-Driven Development(BDD:
   - Test aus Benutzerschicht beschreiben
   - Z.B durch die Business Analystem mit Hilfe von Gherking

Unabhängig der gewählten Strategie, jedes stück Code muss am Schluss Tests haben.

10.2 Refactoring

Definition:

- Strukturierte, disziplinierte Methode, vorhandenen Code umzuschreiben
- Externes Verhalten bleibt gleich!
- Viele kleine Schritte
- Interne Struktur wird verbessert (Um Erweiterungen einzuleiten)
- Trennen von eigentlichen Weiterentwicklung
- Low Level Design-Programmiertechnik

Code verbessern:

- DRY: Keinen duplizierten Code
- Namensgebung: Klarheit erhöhen, Aussagekräftige Namen
- Lange Methoden verkürzen
- Algorithmen strukturieren in:
  - Initialisierung
  - Berechnung
  - Aufbereiten des Resultats
- Sichtbarkeit verbessern
- Testbarkeit verbessern

Code Smells:

- Duplizierter Code
- Lange Methoden
- Klassen mit vielen Instatnzvariablen
- Klassen mit sehr viel Code
- Auffällig ähnliche Unterklassen
- Keine Interface, nur Klassen
- Hohe Kopplung zwischen Klassen

Nach Refactorn wieder testen ob Code noch funktioniert.

10.2.1 Refactoring Patterns

- Rename Method/Class/Variable
- Pull Up / Push Down
  - Methode in Superklasse / Subklasse verschoben
- Extract Interface / Superclass
  - Teil bestehendes Interfaces/Klasse wird in eine Superinterface / Superklasse extrahiert.
- Extract Method

  - Teil einer Methode wird in eine private Methode ausgelagert
- Extract Constant
  - Symbolische Konstante verwenden
- Introduce Explaining Variable
  - Grossen Ausdruck aufteilen, erklärende Zwischenvariablen einfügen.

10.3 Testing

Testarten:

1. Funktionaler Test (Black-Box Verfahren)
2. Nicht funktionaler Test (Lasttest etc)
3. Strukturbezogener Test (White-Box Verfahren)
4. Änderungsbezogener Test (Regressionstest)

Weitere Teststufen und Testarten

- Integrationstest
- Systemtest
- Abnahmetest
- Regressionstest

10.3.1 Integrationstest

- Eine Klasse wird im Anwendungskontext eingesetzt
- Keine Mockups sondern die richtigen referenzierten Klassen eingesetzt
- Ganzes Subsystem oder ganzes System getestet
- Black-Box-Test mit zusätzlichem Wissen über Internas

10.3.2 Systemtest

- Ganzes System oder gesamte Anwenderlogik wird getestet
- Typischerweise Black-Box-Test
- Nicht nur während Entwicklung sondern auch vor Auslieferung an Kunden
- Anwendungsfälle beiziehen

10.3.3 Abnahmetest

- Nach der Auslieferung wird die gesamte Software vom Kunden getestet
- Meist Systemtest über das UI
- Reiner Black-Box-test
- Orientiert sich an Anforderungen des Kunden
- Oft relevant für die Bezahlung

10.3.4 Regressionstest

- Automatische Wiederholung von Tests nach Veränderungen am Quelltext
- Nach Refactoring
- Nach Weiterentwicklung für Funktionen, die nicht geändert haben.

10.3.5 Reproduktion von Fehlern

- Tesfall schreiben nach Meldung Fehler
- Reproduziert Fehler möglichst exakt
- Am besten Systemtest Anwendungslogik sonst über UI
- Eher White-Box-Test

10.3.6 Einbindung in den Prozess

- Testfall vor der Implementation schreiben:
  - Black-Box Test, den der Entwickler selber schreibt
- Testfall nach der Implementation schreiben:
  - Black-Box Test, mit White-Box Test Bereicherungen
  - Unit-, Integration-und/oder Systemtests, Entwickler
- Qualitätssicherung
  - Black-Box System Test, eigene Organisationseinheit
- Abnahmetest
  - Black-Box System Test, Kunde
- Reproduktion von Fehlern

10.3.7 Wichtige Begriffe

- Testling, Testobjekt
  - Objekt, das getestet wird
- Fehler
  - Entwickler macht einen Fehler
- Fehlerwirkung, Bug
  - Jedes zu den Spezifikationen abweichende Verhalten
- Testfall
  - Satz von Testdaten zur vollständigen Ausführung eines Tests
- Testtreiber
  - Rahmenprogramm, das den Test startet und ausführt

10.3.8 Merkmale

- Was wird getestet
  - Einheit/Klasse (Unit-Test)
  - Zusammenarbeit mehrerer Klassen
  - Gesamte Applikationslogik (ohne UI)
  - gesamte Anwendung (über UI)
- Wie wird getestet
  - Dynamisch: Testling wird ausgeführt
    - Black-Box Test
    - White-Box Test
  - Statisch: Qelltext wird analysiert
    - Walkthrough, Review, Inspektion
- Wann wird der Test geschrieben?
  - Vor dem Implementieren (TDD)
  - Nach dem Implementieren
- Wer testet?
  - Entwickler
  - Tester, Qualitätssicherung
  - Kunde, Endbenutzer

11 Entwurf mit Design Pattern I

11.1 Liste der Design Patterns für die Lerneinheit

- Adapter
- Simple Factory
- Singleton
- Dependency Injection
- Proxy
- Chain of Responsibility

11.1.1 Adapter

- **Problem**
  - Einsatz einer Klasse ist inkompatibel mit bereits definierten domänenspezifischen Interface
- **Lösung**
  - Eigene Adapter Klasse dazwischen schalten
- **Hinweise**
  - Oft so ein externer Dienst in eigene Anwendung integriert, insbesondere wenn Dienst austauschbar sein soll
  - Target Interface bewusst für Domänenlogik optimiert, während Adaptee von extern bezogen wird.
  - Falls Adaptee einen externen Dienst, dann im Adapter allenfalls Kommunikation integrieren.

11.1.2 Simple Factory

- **Problem**
  - Das Erzeugen eines neuen Objekts ist aufwändig
- **Lösung**
- Eine eigene Klasse für das Erzeugen eines neuen Objekts wird geschrieben
- **Hinweise**
  - Erzeugung oft abhängig von Konfiguration
  - Auch möglich create() mit Parametern zu ergänzen
  - Delegation Implementieren von Interfaces z.B
  - Factory kann allenfalls erzeugten Objekte zwischenspeichern und wiederverwenden
- **Alternativen**
  - GoF Abstract Factory: Erzeugung einer Familie von verwandten Objekte
  - GoF Factory Method: Basisklasse abstrakte Methode definiert, Objekt eines bestimmten Interfaces zu erzeugen. In abgeleiteten Klassen Methode überschrieben und erzeugt gewünschtes Objekt.

11.1.3 Singleton

- **Problem**
  - Benötigt von einer Klasse eine einzige Instanz
  - Instanz global sichtibar sein
- **Lösung**
  - Klasse mit statischen Methode, liefert immer dasselbe Objekt zurück
  - Statische Methode public deklarieren
- **Hinweise**
  - Globale Sichtbarkeit kritisch
  - Lazy Creation für Instanz möglich, dann aber getInstance() synchronisiert werden.
- **Allgemein**
  - Singletons dann wichtig, wenn einen zentralen Ort braucht um Ressourcen zu verwalten
  - Speicherplatz wird gespart mit nur einem Objekt
  - Java Enum Instanzen sind ebenfalls Singletons
  - Globale Sichtbarkeit eher problematisch

11.1.4 Dependency Injection

- **Problem**
  - Klasse braucht Referenz auf ein anderes Objekt. Dieses Objekt muss ein bestimmtes Interface definieren, je nach Konfiguration mit einer anderen Funktionalität.
- **Lösung**
  - Anstelle Klasse abhängige Objekt selber erzeugt, Objekt von aussen (Injector) gesetzt
- **Hinweise**

  - Ersatz für Factory Pattern
  - Direkter Widerspruch zu GRASP Creator Prinzip
  - Unterstüzt von vielen Frameworks kann auch ohne angewendet werden
  - Erleichtert schreiben von Testfällen insbesondere Gebrauch von Mocks
- Varianten
  - DI über setter Methoden
  - Service bei Constructor vom Client übergeben werden
  - DI initialisiert ganze Anwendung
- Frameworks verwenden Annotationen um anzuzeigen welche Attribute über DI gesetzt werden sollen.

11.1.5 Proxy

- **Problem**
  - Objekt ist nicht oder noch nicht im selben Adressraum verfügbar
- **Lösung**
  - Stellvertreter Objekt mit demselben Interface anstelle des richtigen Objekts verwendet
  - Proxy Objekt leitet alle Methodenaufrufe zum richtigen Objekt weiter
- Einsatz als (Struktur ist dieselbe!)
  - Remote Proxy = Proxy für Objekt in einem anderen Adressraum und übernimmt Kommunikation mit diesem
  - Virtual Proxy = verzögert Erzeugen des richtigen Objekts auf das 1. Mal, dass dieses benutzt wird
  - Protection Proxy = Kontrolliert Zugriff auf das richtige Objekt
- **Hinweise**
  - Unterschied zu Adapter: Ädapteeïn diesem Fall dasselbe Interface implementiert wie Ädapter"
  - Vom Aufbau identisch mit Decorator Pattern hat aber einen anderen Zweck
  - Persistenzframeworks verwenden Virtual Proxy Objekte, um das Erzeugen von Objekten und damit das Herunterladen der Daten zu verzögern, bis die Daten wirklich gebraucht werden.

11.1.6 Chain of Responsibility

- **Problem**
  - Für Anfrage potentiell mehrere handler, aber richtigen Handler im Vornherein nicht möglich herauszufinden
- **Lösung**
  - Handler in einer einfach verketteten Liste hintereinandergeschachtelt.
  - Jeder Handler entscheidet, ob der die Anfrage selber beantworten möchte oder sie an den nächsten Handler weiterleitet.
- **Hinweise**
  - Variante davon:Jeder Handler leitet Anfrage an den nächsten, unabhängig dabon, ob er sie selber behandelt oder nicht

  - Könnte auch sein, dass gar kein Handler Anfrage behandelt

- **Allgemein**
  - Bei allen hierarchischen Stufen: Wenn Referenz auf Parent Node vorhanden, Anfrage zum Parent weiterleiten wenn dies Node selber nicht beabeiten kann.

12 Entwurf mit Design Pattern II

12.1 Decorator

- Problem
  - Objekt(nicht eine ganze Klasse) soll mit zusätzlichen Verantwortlichkeiten versehen werden.
- Lösung
  - Decorator, der dieselbe Schnittstelle hat wir das ursprüngliche Objekt, wird vor dieses geschaltet. Decorator kann nun jeden Methodenaufruf entweder
    1. selber bearbeiten,
    2. ihn an das ursprüngliche Objekt weiterleiten oder
    3. eine Mischung aus beidem machen.
- Hinweise
  - Strukturell identisch mit dem Proxy Design Pattern. Hat aber andere Absicht.
  - Identisch mit Composite Design Pattern wenn Anzahl Elemente 1 ist. Hat aber andere Absicht.

12.2 Observer

- **Problem**
  - Objekt soll ein anderes Objekt benachrichtigen, ohne Typ des Empfängers zu kennen.
- **Lösung**
  - Interface definieren, dient nur dazu, Objekt über eine Änderung zu informieren. Interface vom Observer implementiert. Observable Objekt benachrichtigt ale registrierten Observer über eine Änderung.
- **Hinweise**
  - (Observer-Observable) = (Publisher-Subscriber)= (Listener-Observable)
  - Observable kennt nur Observer aber nicht wahren Typ
  - 2 Phasen:
    1. Registrierung vom Observer
    2. Benachrichtigung vom Observable
- **Erweiterung**
  - Mithilfe Mediatior Pattern kann ein Objekt zwischen Observer und Observable vermitteln. Sowohl Objekt wie auch Observable registrieren sich bei diesem Objekt. Beide benennen eine Eventquelle, die Observer abonniert und Observable mit Events bedient.
  - Beispiel:
    * IoT mit Netzwerkprotokoll MQTT- Server/Client verwendet dieses Prinzip

12.3 Strategy

- **Problem**
  - Algorithmus soll einfach austauschbar sein
- **Lösung**
  - Algorithmus in eine eigene Klasse verschieben, die nur eine Methode mit diesem Algorithmus hat.
  - Interface für diese Klasse definieren, das von alternativen Algorithmen implementiert werden muss.
- **Hinweise**
  - Motivation: technische oder fachspezifische Gründe zur Austauschung
  - Interface:
    - Nur eine Methode
    - Parameter: Alle Daten übergeben welche Algorithmus benötigt. Parameter heisst Context.

12.4 Composite

- **Problem**
  - Menge von Objekten haben dasselbe Interface und müssen für viele Verantwortlichkeiten als Gesamtheit betrachtet werden.
- **Lösung**
  - Composite definieren, das dasselbe Interface implementiert und Methoden an die darin enthaltenen objekte weiterleitet.
- **Hinweise**
  - Hierarchische Struktur of vom Fachgebiet gegeben
  - Nicht alle Methoden delegieren einfach auf enthaltenen Elemente. vor- und Nachbearbeitung ist üblich. Gewisse Methoden müssen ganz anders implementiert werden.

12.5 State

- **Problem**
  - Verhalten eines Objekts ist abhängig von seinem inneren Zustand
- **Lösung**
  - Objekt hat ein darin enthaltenes Zustandsobjekt
  - Alle Methoden, deren Verhalten vom Zustand abhängig sind, über Zustandsobjekt geführt.
- **Hinweise**
  - Zustands-Klassen implementieren Zustand-Interface
  - Zustands-Objekte sind nichts anderes als Strategy Objekte und können Singletons sein.
  - Zustandsobjekt hat entweder direkt den Code (als innere Klasse) oder delegiert an eine Methode des Objekts weiter.
- **Allgemein**
  - In Geschäftsanwendungen State Pattern selten. Häufig in technischen Anwendungen wie Protokollhandler oder Maschinensteuerungen.
- Forum(Inklusive Code)
  - Verschiedene Stufe der Änderungsmöglichkeiten

12.6 Visitor

- **Problem**
  - Klassenhierarchie um Verantwortlichkeiten (weniger wichtige) erweitert werden, ohne dass viele neue Methoden hinzukommen
- **Lösung**
  - Klassenhierarchie mit einer Visitor-Infrastruktur erweitern
  - Alle weiteren neuen Verantwortlichkeiten mit spezigischen Visitor-Klassen realisiert
- **Hinweise**
  - Widerspruch zum Information Expert. Darum wiichtige Methoden weiterhin direkt der Klasse hinzufügen
  - Oft Auswertungen an Visitor-Klassen delegieren
  - Frage bei mehrstufigen Objekthierarchie. wer die darin enthaltenen Elemente aufruft.

12.7 Facade

- **Problem**
  - Einsatz kompliziertes Subsystem mit vielen Klassen. Wie Verwendung vereinfachen, so dass alle Team-Mitglieder es korrekt und einfach verwenden?
- **Lösung**
  - Facade Klasse diefinieren,die vereinfachte Schnittstelle zum Subsystem anbietet und die meisten Anwendungen abdeckt.
- **Hinweise**
  - Facade vs Adapter: Fassade kapselt Subsystem nicht vollständig. Erlaubt dass Methoden der Facade Parameter und Rückgabewerte haben, die Bezug auf das Subsystem nehmen.
  - Oft vom Ersteller eines Frameworks entwickelt

13 vertiefung 1: Verteilte Systeme