1. März, 2024; rev. 7. Juni 2024

Linda Riesen (rieselin)

1 Vorlesung 01

1.1 Charakteristiken von Wasserfall, iterativ-inkrementellen und agilen Softwareentwicklungsprozessen

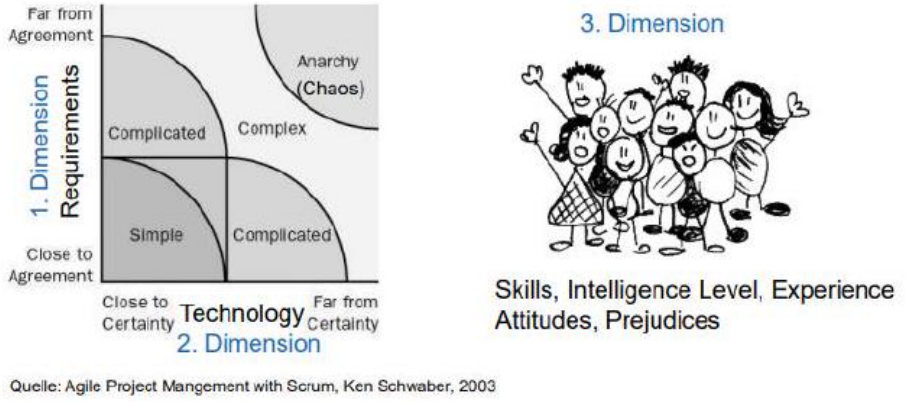

Abbildung 1: Klassifizierung Software-Entwicklungs-Probleme

Prozesse im Softwareengineering Kernprozesse

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

Unterstützungsprozesse

- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

Begriffe Warum wird modelliert: Um Analyse- und Designentwürfe zu diskutieren, abstimmen und zu dokumentieren bzw. zu kommunizieren. Modell: Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhanden Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).
Original: Das Original ist das abgebildete oder zu schaffende Gebilde.
Modellierung: Modellierung gehört zum Fundament des Software Engineerings

- Software ist vielfach (immer?) selbst ein Modell
- Anforderungen sind Modelle der Problemstellung
- Architekturen und Entwürfe sind Modelle der Lösung
- Testfälle sind Modelle des korrekten Funktionierens des Codes usw.

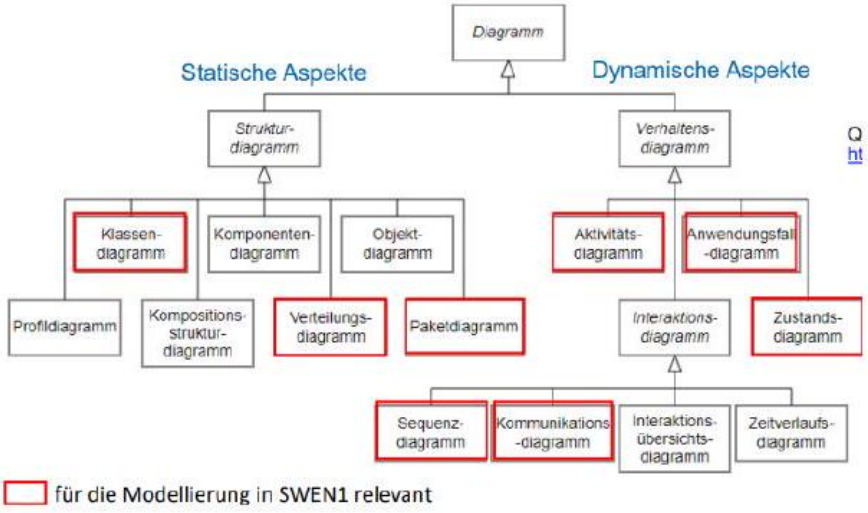

Abbildung 2: Diagramme UML Übersicht

1.1.1 Code and Fix

Vorgehen, bei dem Codierung oder Korrektur im Wechsel mit Ad-hoc-Tests die einzigen bewussten ausgeführten Tätigkeiten der Software-Entwicklung sind: Schnell, Agil, Einfach am Anfang, Schlecht Planbar, Schlecht Wartbar, Änderungen s. Aufwändig

1.1.2 Wasserfallmodell

Die Software-Entwicklung wird als Folge von Aktivitäten/Phasen betrachtet, die durch Teilergebnisse (Dokumente) gekoppelt sind. Die Reihenfolge der Aktivitäten ist fest definiert. : gut planbar, klare Aufteilung in Phasen, Schlechtes Risikomanagment, nie alle Anforderungen zu Anfang bekannt

1.1.3 Iterativ-inkrementelle Modelle

Software wird in mehreren geplanten und kontrolliert durchgeführten Iterationen schrittweise (inkrementell) entwickelt: Flexibles Modell, Gutes Risikomanagement, Frühe Einsetzbarkeit, Planung upfront hat Grenzen, Kunde Involviert über ganze Entwicklung

Agile Softwareentwicklung Basiert auf interativ-inkrementellen Prozessmodell, Fokussiert auf gut dokumentierten und getesteten Code statt auf ausführlicher Dokumentation

1.2 Zweck und den Nutzen von Modellen in der Softwareentwicklung

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

1.2.1 Unified Modelling Language (UML)

UML ist die Standardsprache für die graphische Modellierung von Anforderungen, Analyse und Entwürfen im Software Engineering (objektorientierte Modellierung). (As a sketch, blueprint, programminglanguage)

1.3 Artefakte in einem iterativinkrementellen Prozess illustrieren und einzuordnen

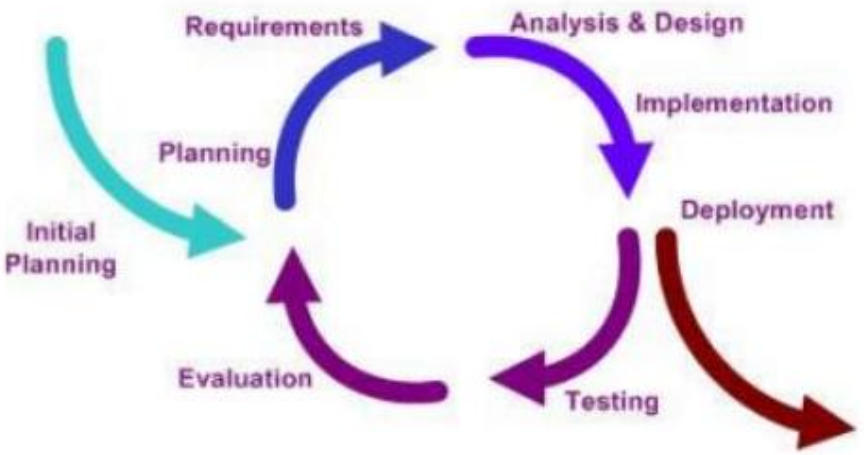

Abbildung 3: Incremental Model

2 Vorlesung 02

2.1 wichtigste Begriffe des Usability-Engineering

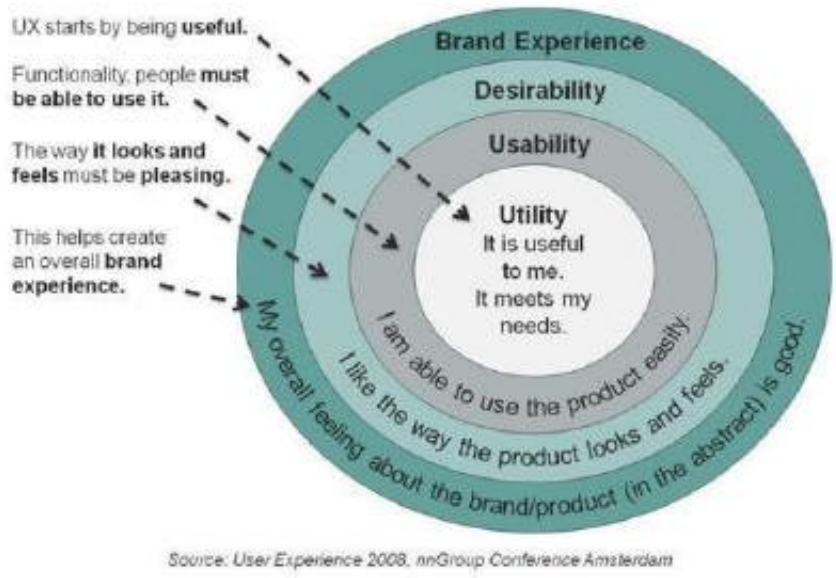

Abbildung 4: Usability und User Experience (UX)

Usability = Deutsch: Gebrauchstauglichkeit
User Experience = Usability + Desirability
Customer Experience = Usability + Desirability + Brand Experience

2.2 Usability-Anforderungen

Die Effektivität, Effizienz und Zufriedenheit mit der die adressierten Benutzer ihre Ziele erreichen in ihren spezifischen Kontexten.

Wichtigste Aspekte

- Benutzer
- Seine Ziele/Aufgaben
- Sein Kontext
- Softwaresystem (inkl. UI)

Effektivität

- Der Benutzer kann alle seine Aufgaben vollständig erfüllen
- Mit der gewünschten Genauigkeit

Effizienz Der Benutzer kann seine Aufgaben mit minimalem/angemessenem Aufwand erledigen

- Mental
- Physisch
- Zeit

Zufriedenheit Mit dem System / der Interaktion

- Minimum: Benutzer ist nicht verärgert
- Normal: Benutzer ist zufrieden
- Optimal: Benutzer ist erfreut

2.2.1 7 wichtige Anforderungsbereiche bezüglich Usability

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

2.3 User-Centered Designs

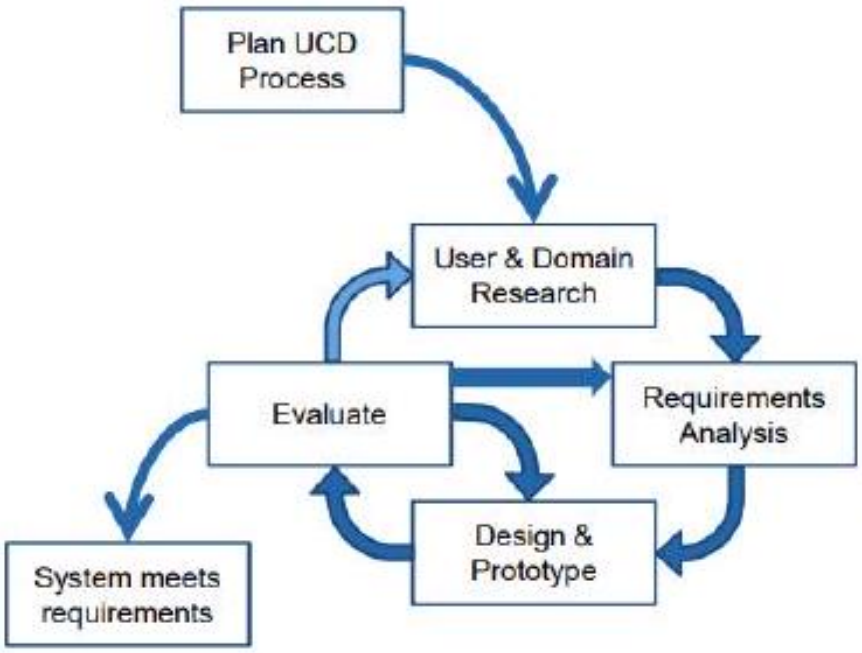

Abbildung 5: Usercentered Design

2.4 Ziele, Methoden und Artefakte der einzelnen Phasen des UCD

- Personas (Fiktive Person mit Eigenschaften / Fähigkeiten): Repräsentitert eine bestimmte Benutzergruppe
- Usage-Szenarien
- Mentales Modell
- Domänenmodell
- Service Blueprint / Geschäftsprozessmodell: Skizze wie Funktioniert was im Geschäft und was für Interaktionen geschehen wo
- Stakeholder Map

Abbildung 6: Stakeholdermap Example

- Zusätzlich: UI-Skizzen der wichtigsten Screens, Wireframes (UIPrototypes), UI-Design, weitere Usability - Anforderungen

3 Vorlesung 03

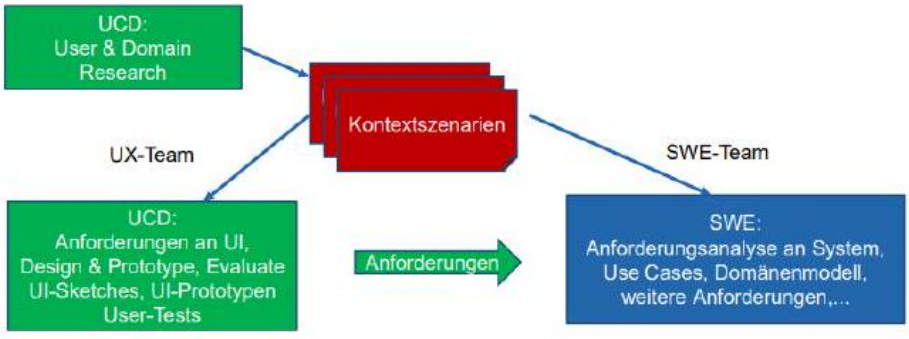

Abbildung 7: Anforderungsanalyse Übersicht

3.1 Anforderungen aus Artefakten des UCD

Anforderungen (Requirements): Forderungen bezüglich (Leistungs-) Fähigkeiten oder Eigenschaften die das System unter gegebenen Bedingungen erfüllen muss (explizit oder implizit)

- Meist sind nie alle Anforderungen im Voraus vollständig bekannt, entwickeln sich im Laufe des Projekts
- Müssen mit den Benutzern und Stakeholdern erarbeitet werden

3.2 Anforderungen in Form von Use Cases

Textuelle Beschreibung einer konkreten Interaktion eines Benutzers mit zukünfigem System (Beschreiben aus Sicht des Akteurs, Impkizite und Explizite Anforderungen, Ziel des Akteurs, Kontext)

3 Arten von Akteuren

- Primärakteur (Primary Actor)
- Initiert einen Anwendungsfall, um sein (Teil-)Ziel zu erreichen
  Erhält den Hauptnutzen des Anwendungsfalls
  Beispiel Kasse: Kassier
- Unterstützender Akteur (Supporting Actor)

Hilft dem SuD bei der Bearbeitung eines Anwendungsfalls
Beispiel Kasse: externer Dienstleister wie Zahlungsdienst für Kreditkarten

- Offstage-Akteur (Offstage Actor)
- Weitere Stakeholder, die nicht direkt mit dem System interagierten
  Beispiel Kasse: Steuerbehörde

Abbildung 8: Arten von Akteuren

- Aus Sicht des Akteurs
- Aktiv formuliert (Titel auch aktiv)
- Konkreter Nutzen
- Mehr als eine einzelne Interaktion / UseCase

- im Essentiellen, nicht Konkreten Stil (Logik, nicht Umsetzung)

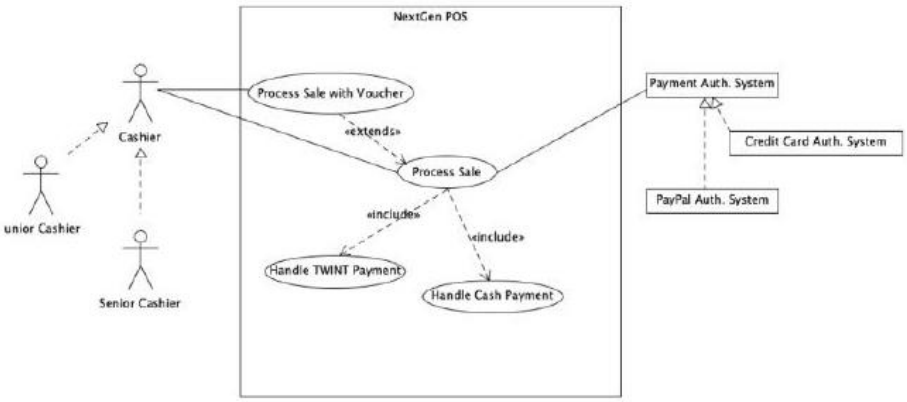

Abbildung 9: Use Case Diagramm

3.2.1 Brief UC

Kurze Beschreibung des Anwendungsfalls in einem Paragraphen

- Nur Erfolgsszenario
- Sollte enthalten:
- Trigger des UCs
- Akteure
- Summarischen Ablauf des UCs
- Wann?: Zu Beginn der Analyse

3.2.2 Casual UC

Informelle Beschreibung des Anwendungsfalls in mehreren Paragraphen

- Erfolgsszenario plus wichtigste Alternativszenarien
- Sollte enthalten:
- Trigger des UCs
- Akteure
- Interaktion des Akteurs mit System
- Wann?: Zu Beginn der Analyse

Formaler Aufbau

- UC-Name
- Umfang (Scope)
- Ebene (Level)
- Primärakteur (Primary Actor)
- Stakeholders und Interessen
- Vorbedingungen (Preconditions)
- Erfolgsgarantie/Nachbedingungen (Success
- Erfolgsgarantie/Nachbedingungen (Succe

  Guarantee)
- Standardablauf (Main Sucess Scenario)
- Erweiterungen (Extensions)
- Spezielle Anforderungen (Special Requirements)
- Liste der Technik und Datavariationen
  (Technology and Data Variations)
- Häufigkeit des Auftretens (Frequency of Occurance
- Verschiedenes (Miscellaneous)

Abbildung 10: Aufbau Fully- Dressed Use Case (UC)

3.2.3 Systemsequenzdiagramm SSD

Ist formal ein UML Sequenzdiagramm: Zeigt Interaktionen der Akteure mit dem System

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

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

- Formal wie Methodenaufruf, evtl mit Parametern, Details zu Parametern sollen im Glossar erklärt werden
- Durchgezogener Pfel für Methodenaufruf

- Rückgabewert kann fehlen falls unwichtig, indirektes Update des UI, deshalb gestrichelte Linie

SSD können auch Interaktionen zwischen SuD und externen unterstützenden System zeigen

- Links ist Primärakteur aufgeführt
  - Hier Cashier
    - Inkl. seiner Benutzerschnittstelle
  - Initiiert die Systemoperationen (via UI)
    - UI findet zusammen mit Akteur heraus, was dieser tun möchte
    - UI ruft sodann entsprechende Systemoperation auf
- Mitte das System (:System)
  - Muss die Systemoperationen zur Verfügung stellen
- Rechts
  - Sekundärakteure, falls nötig

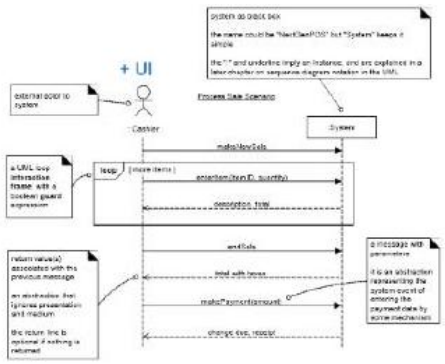

Abbildung 11: Systemsequenzdiagramm (SSD)

4 Vorlesung 04

4.1 UML Klassendiagramm = Domänenmodell (vereinfachtes UML Klassendiagramm)

Konzepte werden als Klassen modelliert, Eigenschaften als Attribute (ohne Typangabe), Assoziationen mit Multiplizitäten als Beziehung zw. Konzepten (wenn notwendig noch Aggregation (Beschriftung d Pfeile))

4.1.1 Konzepte: Substantive

- Physische Objekte
- Kataloge
- Container von Dingen
- Andere beteiligte Systeme
- Rollen von beteiligten Personen
- Artefakte (Pläne, Finanzen, Arbeit, Verträge)
- Zahlungsinstrumente
- Keine Softwareklassen

4.1.2 Attribute: sollen einfach/wichtig sein

- Transaktion
- Teil zum Ganzen
- Beschreibung/ Protokoll zum Gegenstand
- Verwendung

Attribute an Stelle von Assoziationen Verwenden Sie Assoziationen und nicht Attribute, um Konzepte in Beziehung zueinander zu setzen.

4.2 Analysemuster

4.2.1 Beschreibungsklassen

Artikel, Physischer Gegenstand, Diensleitung: hat Preis, Serie Nummer u Code

4.2.2 Generalisierung / Spezialisierung

Wenn 100% Regel: alle instanzen eines spezialisierten Konzepts sind auch Instanzen des generalisierten Konzepts und ÏS A"
Assoziationen und Attribute dienen umgekehrt als Begründung für eine gemeinsame generalisierte Klasse.

4.2.3 Komposition

4.2.4 Zustände

Sollen durch eigene Hierarchie dargestellt werden

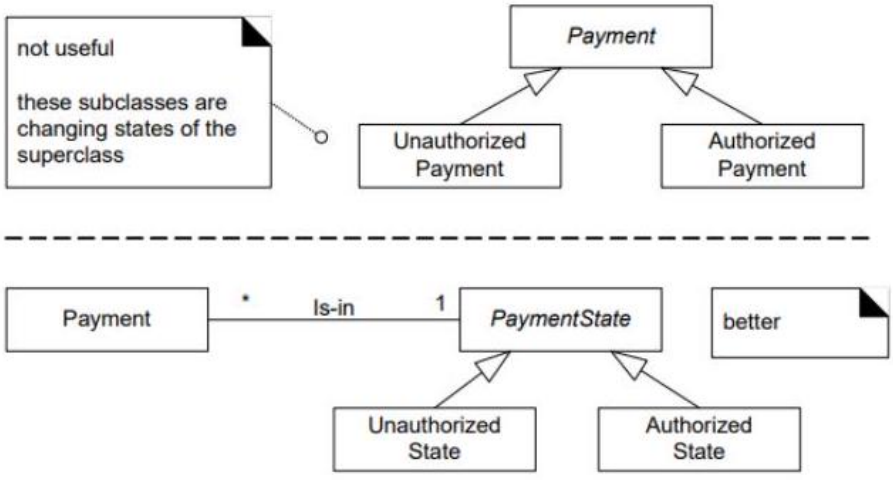

Abbildung 12: Zustände Domänenmodell(DM) Beispiel

4.2.5 Rollen (Manager etc.)

Dasselbe Konzept (aber selten dieselbe Instanz) kann unterschiedliche Rollen einnehmen.
Dargestellt als Konzepte / Assoziationen

4.2.6 Assoziationsklasse

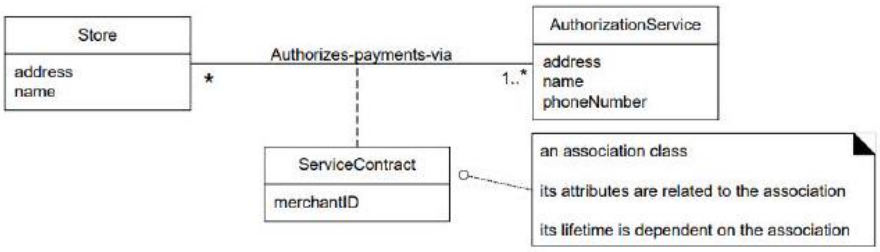

Abbildung 13: Assoziationsklasse Beispiel

4.2.7 Masseinheiten / Zeitintervalle

Oft Sinnvollerweise als Konzept modelliert

5 Vorlesung 05

5.1 Ubersicht Business Analyse vs Architektur vs Entwicklung

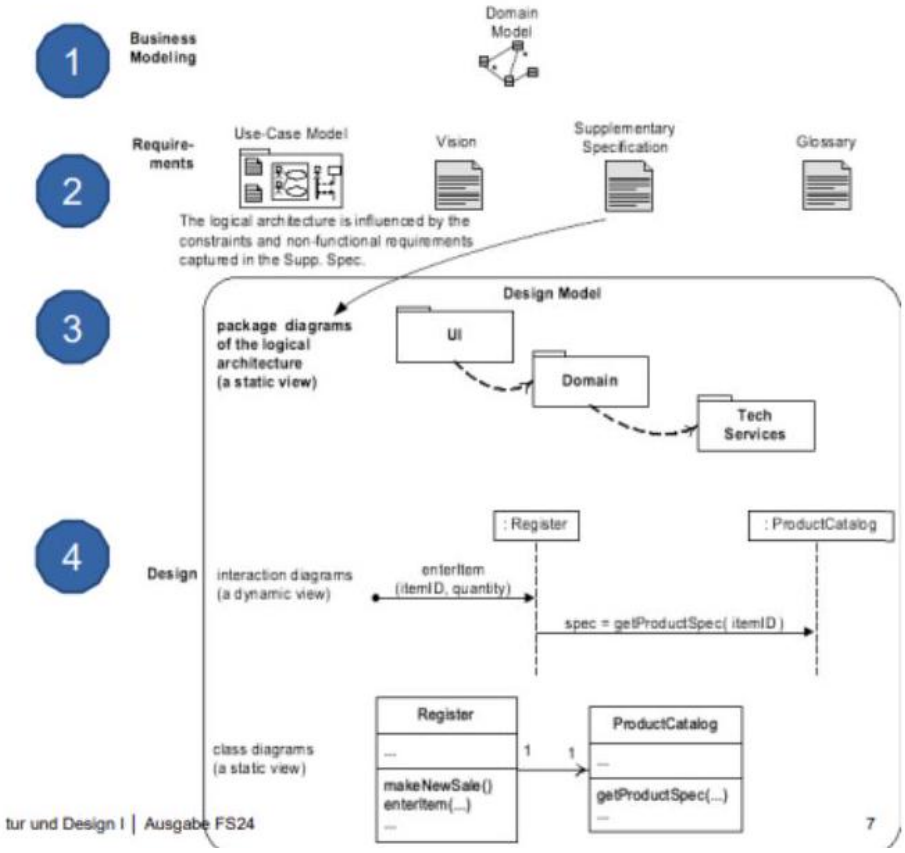

Abbildung 14: Übersicht Business Analyse vs Architektur vs Entwicklung

1. Domänenmodell (Business Modelling) Kontext Diagramm (Business Analyst)
2. Requirement (Business Analyst)
3. Logische Architektur (Software Architekt)
4. Umsetzung (Entwicklung)

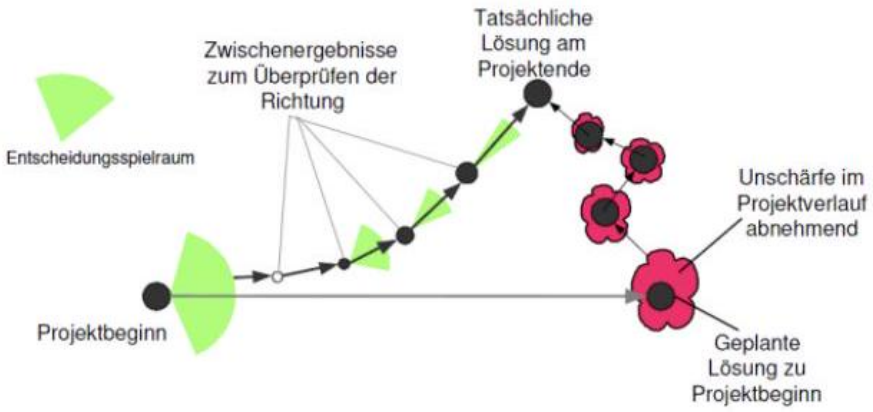

Abbildung 15: Entstehung Archtektur

5.2 Architektur aus Anforderungen

Die Architektur muss heutige und zukünftige Anforderungen erfüllen können und Weiterentwicklungen der Software und seiner Umgebung ermöglichen

5.2.1 Architekturanalyse

Analyse der funktionalen und nichtfunktionalen Anforderungen:

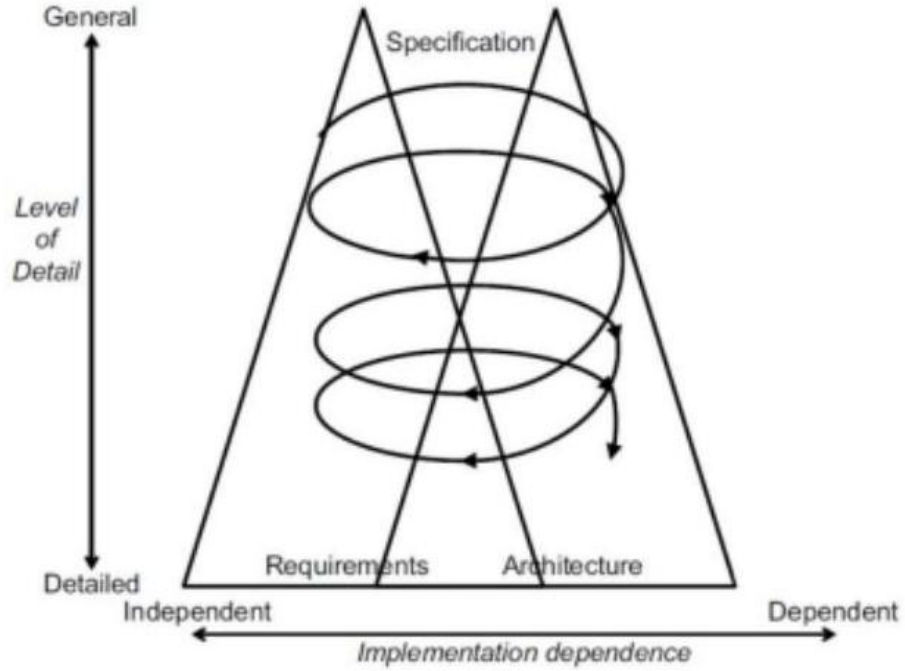

Abbildung 16: Twin Peak Model
Entwurfsentscheidungen sollten in erster Linie aus den Anforderungen abgeleitet werden, Architekturentscheidungen und die Konsequenzen daraus müssen mit den Stakeholdern abgestimmt werden.

5.2.2 ISO 25010

- ISO 25010 provides a hierarchical structure for non-functional requirements.
- It defines main characteristics, sub-characteristics, and metrics.
- Each non-functional requirement in ISO 25010 is associated with metrics.
- Metrics include a description of the requirement, a measurement method to check requirement fulfillment, and guidance for interpreting results.
- This allows for more precise and measurable formulation of requirements, which can be verified later.
  5.2.3 Difference from FURPS + (Functionality, Usiability, Reliability, Performance, Supportability (Anpassungsfähigkeit, Wartbarkeit, etc), + = Implementation, Interface, Operations, Packaging, Legal)

- FURPS+ is an acronym and not a standard.
- FURPS+ includes Functionality, Usability, Reliability, Performance, Supportability, and other terms.

5.2.4 Grundprinzip: Modulkonzept

Modul (Baustein, Komponente): Güte wird gemessen mit Kohäsion und Kopplung

- Möglichst autarkes Teilsystem (wenig Kopplung nach aussen)
- Hat eine klare minimale Schnittstelle gegen aussen
- Software-Modul enthält alle Funktionen und Datenstrukturen, die es benötigt
- Modul kann sein: Paket, Programmierkonstrukt, Library, Komponente, Service

5.2.5 Architektur beschreiben

Architektur umfasst verschiedene Aspekte, die je nach Sichtweise wichtig sind

N+1 View Model + 1 View: Use Cases

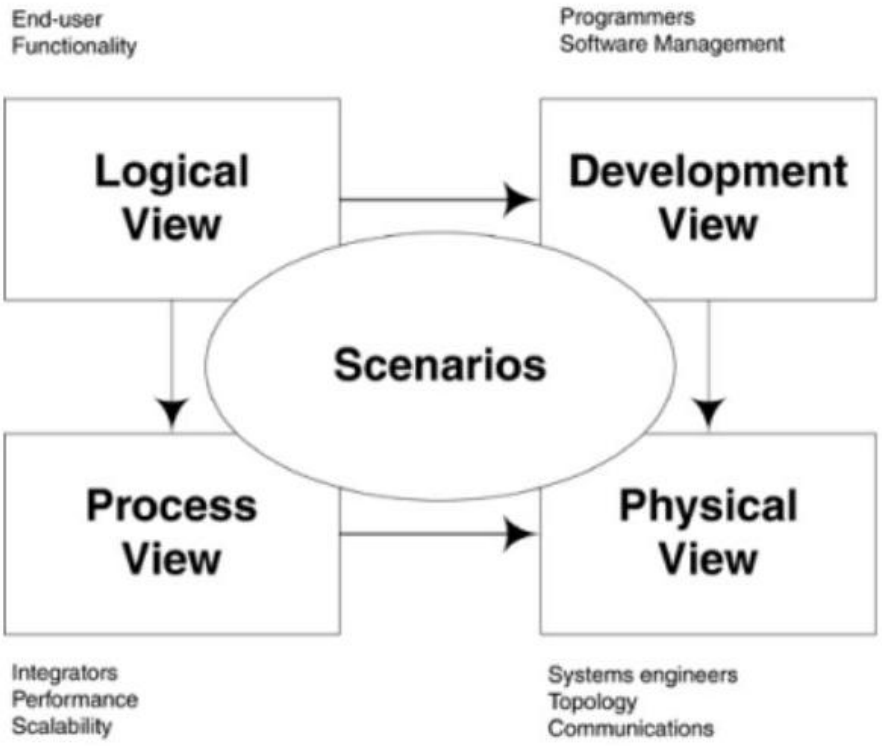

Abbildung 17: View Model

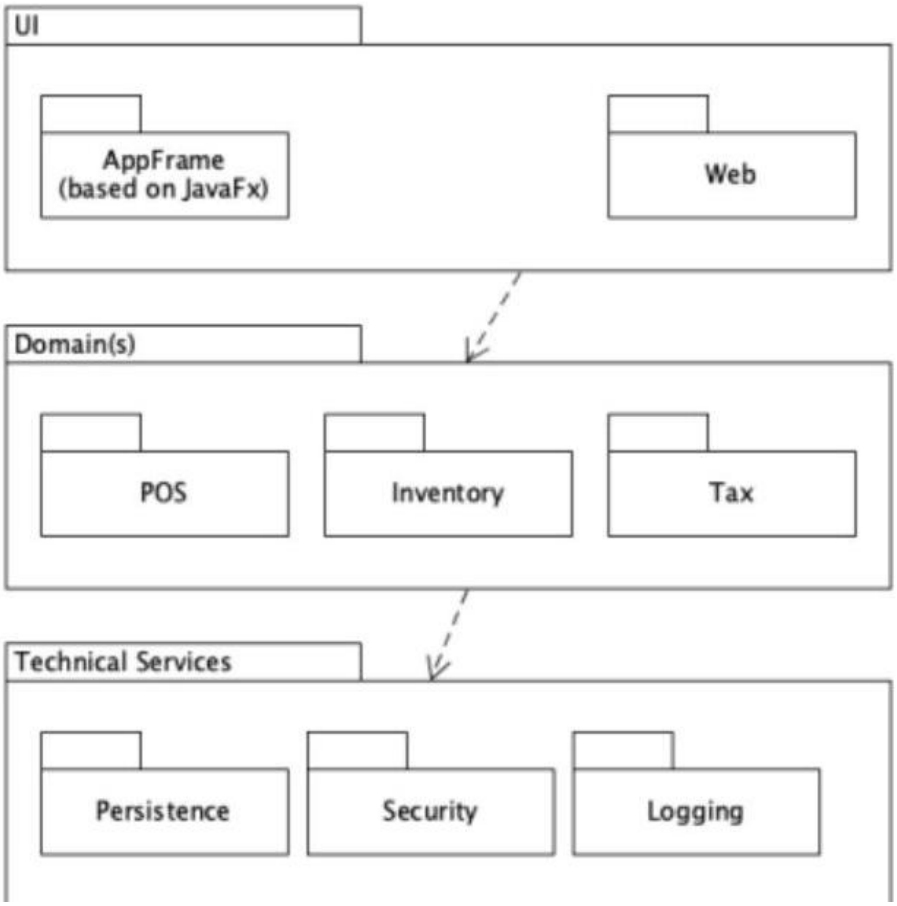

Abbildung 18: UML- Paketdiagram

- Mittel, um Teilsysteme zu definieren
- Mittel zur Gruppierung von Elementen

Paket enthält Klassen und andere Pakete

- Ähnlich, aber allgemeiner als Java Packages

Abhängigkeiten zwischen Paketen

Pattern	Beschreibung
Layered Pattern	Strukturierung eines Programms in Schichten
Client-Server Pattern	Ein Server stellt Services für mehrere Clients zur Verfügung
Master-Slave Pattern	Ein Master verteilt die Arbeit auf mehrere Slaves
Pipe-Filter Pattern	Verarbeitung eines 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	Eine interaktive Anwendung wird in 3 Komponenten aufgeteilt: Model, View – Informationsanzeige, Controller – Verarbeitung der Benutzereingabe

Abbildung 19: Architekturpatterns

6 Vorlesung 06

6.1 Zweck und Anwendung von Statischen und Dynamischen Modellen im Design

- Statische Modelle, wie beispielsweise das UML-Klassendiagramm, unterstützen den Entwurf von Paketen, Klassennamen, Attributen und Methodensignaturen (ohne Methodenkörper).
- Dynamische Modelle, wie beispielsweise UMLInteraktionsdiagramme, unterstützten den Entwurf der Logik, des Verhaltens des Codes und der Methodenkörper.

Statische u Dynamische ergänzen sich, werden parallel erstellt

6.2 Objektentwurf mit UML-Klassen-, UML-Interaktions-, UML-Zustands- und UML-Aktivitätsdiagrammen

6.2.1 UML-Klassendiagramm

Notationselemente:

- Klasse, aktive Klasse
- Attribut
- Operation
- Sichtbarkeit von Attributen und Operationen
- Assoziationsname, Rollen an den Assoziationsenden
- Multiplizität (Bezieht sich auf die Objekte der betreffenden Klasse)
- Navigierbarkeit in Assoziationen

- Datentypen und Enumerationen
- Generalisierung / Spezialisierung
- Abstrakte Klassen
- Assoziation, Assoziationsklasse: Komposition, Aggregation
- Interface, Interface Realisierung

6.2.2 UML-Interkationsdiagramm

Modellieren die Kollaborationen bzw. den Informationsaustausch zwischen Objekten (Dynamik).

Sequenzdiagramm Notationselemente:

- Lebenslinie
- Aktionssequenz
- Synchrone Nachricht
- Antwortnachricht
- Gefundene, verlorene Nachricht
- Kombiniertes Fragment
- Erzeugungs-, Löschereignis
- Selbstaufruf
- Interaktionsreferenz
- Lebensline mit aktiver Klasse
- Asynchrone Nachricht

Kommuninaktionsdiagramm

- Lebenslinie (Box)
- Synchrone Nachricht (= Aufruf einer Operation) (Nummeriert)
- Antwortnachricht (= Rückgabewert)
- Bedingte Nachrichten «[]»
- Iteration «*»

6.2.3 UML-Zustandsdiagramm

Welche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, ... bei welchen Ereignissen annehmen?

- Start-, Endzustand
- einfacher Zustand
- Zusammengesetzter bzw. geschachtelter Zustand
- Flache und tiefe Historie
- Transition
- Orthogonaler Zustand
- Parallelisierungsknoten
- Synchronisationsknoten
- Einstiegpunkt
- Ausstiegspunkt
- Unterzustandsautomat

6.2.4 UML-Aktivitätsdiagramm

Wie läuft ein bestimmter Prozess oder ein Algorithmus ab?

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

6.3 Responsibility Driven Design (RDD)

Denken in Verantwortlichkeiten, Rollen und Kollaborationsbeziehungen für den Entwurf von Softwareklassen.
RDD kann auf jeder Ebene des Designs angewendet werden (Klasse, Komponente, Schicht).
Verantwortlichkeiten werden durch Attribute und Methoden implementiert.

6.4 Prinizpien für Klassenentwurf: GRASP, SOLID

6.4.1 SOLID: Missing from Slides

6.4.2 GRASP (General Responsibility Assignment Software Patterns)

- welche Klassen und Objekte wofür zuständig
- für erleichterung Kommunikation d Entwickler
- grundlegenden Prinzipen bzw. Pattern
- Information Expert: Ein Objekt sollte die Verantwortung für eine Aufgabe übernehmen, wenn es die benötigten Informationen dazu besitzt.
- Creator: Ein Objekt sollte für die Erstellung von anderen Objekten verantwortlich sein, wenn eine starke Beziehung zwischen ihnen besteht.
- Controller: Ein Objekt sollte die zentrale Steuerungslogik in einem System repräsentieren.
- Low Coupling: Objekte sollten lose miteinander gekoppelt sein, um die Flexibilität und Wiederverwendbarkeit des Systems zu erhöhen.
- High Cohesion: Eine Klasse sollte nur zusammengehörige Funktionen und Daten enthalten, um ihre Verständlichkeit und Wartbarkeit zu verbessern.
- Polymorphism: Objekte sollten so entworfen werden, dass sie anhand ihrer Schnittstellen verwendet werden können, unabhängig von ihrer spezifischen Implementierung.
- Pure Fabrication: Künstliche Klassen sollten erstellt werden, um eine hohe Kohäsion und niedrige Kopplung zu erreichen, wenn keine natürliche Klasse die Verantwortung übernehmen kann.
- Indirection: Zwischen Objekten sollten indirekte Verbindungen hergestellt werden, um die Flexibilität und Wartbarkeit zu erhöhen.
- Protected Variations: Mechanismen sollten eingeführt werden, um Variationen in den Systemkomponenten zu schützen und unerwünschte Auswirkungen von Änderungen zu minimieren.

7 Vorlesung 07

7.1 Use Cases und Use-Case-Realization

Die Planung erfolgt anhand von Use-Cases, Realisierung v Use-Cases
Der wichtigste Teil sind die detaillierten Szenarien (Standardszenario und Erweiterungen), und davon die Systemantworten. Diese müssen schlussendlich realisiert werden.

7.1.1 Vergleich SSD

UI statt System, Systemoperationen sind Elemente die realisiert werden

7.1.2 Warum UML

- Zwischenschritt bei wenig Erfahrung
- Ersatz für Programmiersprache, Kompakt
- auch für Laien zu verstehen

7.2 Vorgehen UC Realization

1. Use Case auswählen, offene Fragen klären, SSD ableiten
2. Systemoperation auswählen
3. Operation Contract (Systemvertrag) erstellen/überlegen
4. Aktuellen Code/Dokumentation analysieren
   (a) DCD überprüfen/aktualisieren
   (b) Vergleich mit Domänenmodell durchführen
   (c) Neue Klassen gemäß Domänenmodell erstellen
5. Falls notwendig, Refactorings durchführen
   (a) Controller Klasse bestimmen
   (b) Zu verändernde Klassen festlegen
   (c) Weg zu Klassen festlegen
   i. Weg mit Parametern wählen
   ii. Klassen ggf. neu erstellen
   iii. Verantwortlichkeiten zuweisen
   iv. Varianten bewerten
   (d) Veränderungen programmieren
   (e) Review durchführen

8 Vorlesung 08

8.1 Allgemeiner Aufbau u Zweck von Design Pattern (DP)

- bewährte Lösungen f wiederkehrende Probleme schnell finden
- effiziente Kommunikation
- immer tradeoff zw. Flexibilität u Kompatibilität
- Programm wird nicht besser mit DP

8.1.1 Adapter

Ermöglicht die Zusammenarbeit von Objekten mit inkompatiblen Schnittstellen. (Überall wo Dienste angesprochen werden, die austauschbar sein sollten)

8.1.2 Simple Factory

(eigene Klasse) erstellt Objekte, die aufwändig zu erzeugen sind

8.1.3 Singleton

Stellt sicher, dass eine Klasse nur eine Instanz hat und einen globalen Zugriffspunkt darauf bereitstellt.

8.1.4 Dependency Injection

Ermöglicht es, einem abhängigen Objekt die benötigten Abhängigkeiten bereitzustellen. (Ersatz f Facotry Pattern)

8.1.5 Proxy

Bietet einen Platzhalter (mit demselben Interface) für ein anderes Objekt, um den Zugriff darauf zu kontrollieren. Proxy Objekt leitet alle Methodenaufrufe zum richtigen Objekt weiter

- Remote Proxy: Stellvertreter für ein Objekt in einem anderen Adressraum und übernimmt die Kommunikation mit diesem.
- Virtual Proxy: Verzögert das Erzeugen des richtigen Objekts bis zum ersten Mal, dass es benutzt wird.
- Protection Proxy: Kontrolliert den Zugriff auf das richtige Objekt.

8.1.6 Chain of Responsibility

Ermöglicht es einem Objekt, eine Anfrage entlang einer Kette potenzieller Handler zu senden, bis einer die Anfrage behandelt. (wenn unklar im vorraus welcher Handler zuständig sein wird)

9 Vorlesung 09

9.1 Decorator

9.1.1 Problem

Ein Objekt (nicht eine ganze Klasse) soll mit zusätzlichen Verantwortlichkeiten versehen werden.

9.1.2 Solution

Ein Decorator, der dieselbe Schnittstelle hat wie das ursprüngliche Objekt, wird vor dieses geschaltet. Der Decorator kann nun jeden Methodenaufruf entweder selber bearbeiten, ihn an das ursprüngliche Objekt weiterleiten oder eine Mischung aus beidem machen.

9.2 Observer

9.2.1 Problem

Ein Objekt soll ein anderes Objekt benachrichtigen, ohne dass es den genauen Typ des Empfängers kennt.

9.2.2 Solution

Ein Interface wird definiert, das nur dazu dient, ein Objekt über eine Änderung zu informieren. Dieses Interface wird vom «Observer» implementiert. Das «Observable» Objekt benachrichtigt alle registrierten «Observer» über eine Änderung.

9.3 Strategy

9.3.1 Problem

Ein Algorithmus soll einfach austauschbar sein.

9.3.2 Solution

Den Algorithmus in eine eigene Klasse verschieben, die nur eine Methode mit diesem Algorithmus hat.
Ein Interface für diese Klasse definieren, das von alternativen Algorithmen implementiert werden muss

9.4 Composite

9.4.1 Problem

Eine Menge von Objekten haben dasselbe Interface und müssen für viele Verantwortlichkeiten als Gesamtheit betrachtet werden.

9.4.2 Solution

Sie definieren ein Composite, das ebenfalls dasselbe Interface implementiert und Methoden an die darin enthaltenen Objekte weiterleitet

9.5 State

9.5.1 Problem

Das Verhalten eines Objekts ist abhängig von seinem inneren Zustand.

9.5.2 Solution

Das Objekt hat ein darin enthaltenes Zustandsobjekt.
Alle Methoden, deren Verhalten vom Zustand abhängig sind, werden über das Zustandsobjekt geführt.

9.6 Visitor

9.6.1 Problem

Eine Klassenhierarchie soll um (weniger wichtige) Verantwortlichkeiten erweitert werden, ohne dass viele neue Methoden hinzukommen.

9.6.2 Solution

Die Klassenhierarchie wird mit einer Visitor-Infrastruktur erweitert. Alle weiteren neuen Verantwortlichkeiten werden dann mit spezifischen VisitorKlassen realisiert.

9.7 Facade

9.7.1 Problem

Sie setzen ein ziemlich kompliziertes Subsystem mit vielen Klassen ein. Wie können Sie seine Verwendung so vereinfachen, dass alle Team-Mitglieder es korrekt und einfach verwenden können?

9.7.2 Solution

Eine Facade (Fassade) Klasse wird definiert, welche eine vereinfachte Schnittstelle zum Subsystem anbietet und die meisten Anwendungen abdeckt.

9.8 Abstract Factory

9.8.1 Problem

Sie müssen verschiedene, aber verwandte Objekte erstellen, ohne ihre konkreten Klassen anzugeben. Wie können Sie die Erstellung der Objektfamilien zentralisieren und von ihrer konkreten Implementierung abstrahieren?

9.8.2 Solution

Das Abstract Factory Muster definiert eine Schnittstelle zur Erstellung von Familien verwandter oder abhängiger Objekte, ohne ihre konkreten Klassen zu benennen. Es bietet Methoden, um Objekte der verschiedenen Produktfamilien zu erstellen, und ermöglicht es, ganze Produktfamilien konsistent zu verwenden.

9.9 Factory Method

9.9.1 Problem

Es gibt eine Oberklasse, aber die genauen Unterklassen sollen durch eine spezielle Logik zur Laufzeit bestimmt werden. Wie können Sie die Instanziierung dieser Klassen so gestalten, dass sie flexibel und erweiterbar bleibt?

9.9.2 Solution

Das Factory Method Muster definiert eine Schnittstelle zur Erstellung eines Objekts, lässt aber die Unterklassen entscheiden, welche Klasse instanziiert wird. Dadurch wird die Erstellung der Objekte auf Unterklassen delegiert, wodurch die Klasse flexibler und erweiterbar wird.

9.10 Command

9.10.1 Problem

Sie müssen eine Anforderung in Form eines Objekts kapseln, um parameterisierte Clients, Warteschlangen oder Log-Requests sowie Operationen rückgängig machen zu können. Wie können Sie dies strukturieren?

9.10.2 Solution

Das Command Muster kapselt eine Anforderung als Objekt, wodurch Sie Parameter für Clients, Warteschlangen oder Log-Requests einfügen und Operationen rückgängig machen können. Es besteht aus einem Kommando-Objekt, das eine bestimmte Aktion mit ihren Parametern, Empfänger und eventuell einer Methode zur Rückgängigmachung enthält.

9.11 Template Method

9.11.1 Problem

In einer Methode einer Oberklasse gibt es einige Schritte, die in Unterklassen unterschiedlich implementiert werden sollen, während die Struktur der Methode erhalten bleiben muss. Wie können Sie die Schritte anpassbar machen?

9.11.2 Solution

Das Template Method Muster definiert das Skelett eines Algorithmus in einer Methode, wobei einige Schritte von Unterklassen implementiert werden. Es ermöglicht Unterklassen, bestimmte Schritte des Algorithmus zu überschreiben, ohne die Struktur des Algorithmus zu verändern.

10 Vorlesung 10

10.1 Quellcode aus Design Artefakten ableiten

10.1.1 Umsetzungs-Reihenfolge: Variante Bottom-Up

Kurze Erklärung: Implementierung beginnt mit Basisbausteinen, die schrittweise zu größeren Teilen kombiniert werden.
Vorgehen: Start mit Basisfunktionalitäten, dann schrittweise Erweiterung und Integration.
Eigenschaften: Gründlich, bietet solide Basis, gut für sich ändernde Anforderungen.

10.1.2 Umsetzungs-Reihenfolge: Variante Agile

Kurze Erklärung: Flexible, inkrementelle Entwicklung in kurzen Iterationen.
Vorgehen: Kontinuierliche Lieferung funktionsfähiger Teile in Sprints, Anpassung an sich ändernde Anforderungen.
Eigenschaften: Hohe Flexibilität, schnelles Feedback, geeignet für sich ändernde Anforderungen.

10.2 Codier-Richtlinien

Abmachung für:

- Fehlerbehandlung
- Codierrichtlinien (Gross/Kleinschreibung, Einrücken, Klammernsetzung, Prüf programme)
- Namensgebung f. Klasse, Attribute, Methoden, Variablen

10.3 Implementierungs- / Umsetzungsstrategie

Code-Driven Development

- Zuerst die Klasse implementieren

TDD: Test-Driven Development

- Zuerst Tests für Klassen/Komponenten schreiben, dann den Code entwickeln

BDD: Behavior-Driven Development

- Tests aus Benutzersicht beschreiben
- Zum Beispiel durch die Business Analysten mit Hilfe von Gherkin

10.4 Laufzeit Optimierung, Optimierungsregeln

- Optimiere nicht sofort, sondern analysiere zuerst, wo Zeit tatsächlich verbraucht wird.
- Verwende Performance-Monitoring-Tools, um Zeitfresser zu identifizieren.
- Besonders kritisch sind Datenbankzugriffe pro Objekt über eine Liste.
- Überprüfe und optimiere Algorithmen, z.B. Collections.sort() in Java 7.
- Bedenke, dass moderne Compiler bereits viel Optimierung leisten.
- Ziehe Berechnungen aus Schleifen heraus, da die Java VM und Just-In-Time-Compilation diese optimieren.

10.5 Refactoring

Definition: Strukturierte Methode zum Umstrukturieren vorhandenen Codes, ohne das externe Verhalten zu ändern. Ziele:

- Verbesserung der internen Struktur durch viele kleine Schritte.
- Trennung vom eigentlichen Entwicklungsprozess.
- Verbesserung des Low-Level-Designs und der Programmierpraktiken.

Methoden zur Code-Verbesserung:

- DRY: Vermeidung von dupliziertem Code.
- Klare Namensgebung für erhöhte Lesbarkeit.
- Aufteilung langer Methoden in kleinere, klar definierte Teile.
- Strukturierung von Algorithmen in Initialisierung, Berechnung und Ergebnisverarbeitung.
- Verbesserung der Sichtbarkeit und Testbarkeit.

Code Smells:

- Duplizierter Code
- Lange Methoden
- Klassen mit vielen Instanzvariablen oder viel Code
- Auffällig ähnliche Unterklassen
- Fehlen von Interfaces oder hohe Kopplung zwischen Klassen

Unterstützung durch:

- Automatisierte Tests zur Sicherstellung der Funktionsfähigkeit nach Refactoring.
- Moderne Entwicklungsumgebungen, die abhängige Arbeitsschritte automatisieren.

Refactoring Patterns:

- Umbenennung von Methoden, Klassen und Variablen für klarere Bezeichnungen.
- Verschieben von Methoden in Super- oder Subklassen.
- Extrahieren von Teilfunktionen in separate Methoden oder Konstanten.
- Einführung erklärender Variablen zur Verbesserung der Lesbarkeit.

10.6 Testing

10.6.1 Grundlegende Testarten

- Funktionaler Test (Black-Box Verfahren): Überprüft die Funktionalität des Systems, ohne den internen Code zu kennen.
- Nicht funktionaler Test (Lasttest etc.): Testet nicht-funktionale Anforderungen wie Leistung, Skalierbarkeit, usw.
- Strukturbezogener Test (White-Box Verfahren): Überprüft die interne Struktur des Codes, um sicherzustellen, dass alle Pfade abgedeckt sind.
- Änderungsbezogener Test (Regressionstest etc.): Überprüft, ob durch Änderungen im Code keine neuen Fehler eingeführt wurden.
- Integrationstest
- Systemtest
- Abnahmetest
- Regressionstest

10.6.2 Wichtige Begriffe

- Testling, Testobjekt: Objekt, das getestet wird
- Fehler: Fehler, den der Entwickler macht
- Fehlerwirkung, Bug: Jedes Verhalten, das von den Spezifikationen abweicht
- Testfall: Satz von Testdaten zur vollständigen Ausführung eines Tests
- Testtreiber: Programm, das den Test startet und ausführt

10.6.3 Merkmale

Was wird getestet?

- Einheit / Klasse (Unit-Test)
- Zusammenarbeit mehrerer Klassen
- Gesamte Applikationslogik (ohne UI)
- Gesamte Anwendung (über UI)

Wie wird getestet?

- Dynamisch: Testfall wird ausgeführt (Black-Box / White-Box Test)
- Statisch: Quelltext wird analysiert (Walkthrough, Review, Inspektion)

Wann wird der Test geschrieben?

- Vor dem Implementieren (Test-Driven Development, TDD)
- Nach dem Implementieren

Wer testet?

- Entwickler
- Tester, Qualitätssicherungsabteilung
- Kunde, Endbenutzer

11 Vorlesung 11

11.1 Verteiltes System Definition + Einsatz

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

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

11.2 Verteiltes System Konzepte + Architekturstil

Verteilte Systeme basieren auf verschiedenen Konzepten und Architekturstilen:

11.2.1 Kommunikationsverfahren

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

11.2.2 Fehlertoleranz

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

11.2.3 Fehlersemantik

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

11.3 Design- und Implementierungsaspekte von Client-ServerSystemen

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

11.4 Verteiltes System Architektur + Design Patterns

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

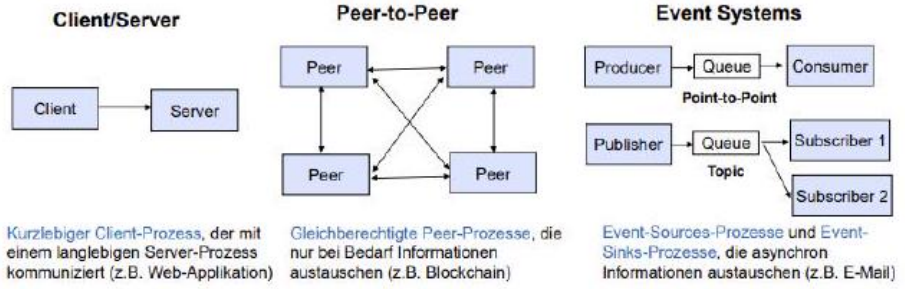

Abbildung 20: Architekturmodelle

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

Für die Entwicklung verteilter Systeme stehen verschiedene MiddlewareTechnologien zur Verfügung, die die Kommunikation und Integration von verteilten Komponenten erleichtern. Dazu gehören Messaging-Broker wie Apache

Kafka, Middleware-Frameworks wie CORBA (Common Object Request Broker Architecture) und RESTful Web Services.